DIRECTORIO DE PROFESORES DEL CURSO: METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION 1985.

1. M. EN C. EFRAIN PARDO ORTIZ
DIRECTOR GENERAL
DINAMICA ORGANIZACIONAL
REVOLUCION NO. 1134-2
MEXICO,D.F.
680 04 88 y 680 60 96

2. ING. LUIS CORDERO BORBOA (COORDINADOR)
JEFE DEL DEPARTAMENTO DE COMPUTACION
DIVISION DE INGENIERIA MECANICA Y ELECTRICA
FACULTAD DE INGENIERIA
UNAM
MEXICO,D.F.
550 52 15 EXT. 3750 y 3746

3. ING. SALVADOR PEREZ VIRAMONTES
CONSULTOR
CONSULTORES EN INFORMATICA SICISA
SAN FRANCISCO 706
COL. DEL VALLE
MEXICO,D.F.
543 87 05

4. ING.DANIEL RIOS ZERTUCHE
SUBDIRECTOR DE COMPUTO
DIRECCION GENERAL DE POLITICA,ECONOMICA Y SOCIAL
S. P. P.
IZAZAGA NO. 38-11° PISO
MEXICO 1, D.F.
521 98 98

7. ING. JUAN CARREON
JEFE DEL CENTRO DE INFORMACION
FACULTAD DE INGENIERIA
UNAM
MEXICO,D.F.

8. M. EN I. GABRIEL SANCHEZ GUERRERO
PROFESOR
DEPFI
UNAM
MEXICO,D.F.
550 52 15 EXT. 4486

9. DR. ALEJANDRO MENDOZA F.
PROFESOR
DEPFI
UNAM
MEXICO,D.F.
550 52 15 EXT. 4491

10. DR. JOSE DE JESUS ACOSTA FLORES
SUBJEFE DEL AREA DE INGENIERIA.DEDSISTEMAS
D E P F I
UNAM
MEXICO,D.F.
550 52 15 EXT. 4482

11. ING. JACINTO VIQUEIRA LANDA
JEFE DE LA DIVISION DE INGENIERIA
MECANICA Y ELECTRICA
FACULTAD DE INGENIERIA
UNAM
MEXICO,D.F.
550 52 15 ext. 3746

EVALUACION DEL PERSONAL DOCENTE

CURSO: METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

FECHA: DEL 31 de mayo al 28 de junio 1985.

	CONFERENCISTA	DOMINIO DEL TEMA	EFICIENCIA EN EL USO DE AYUDAS AUDIOVISUALES	MANTENIMIENTO DEL INTERES.(COMUNICACION CON LOS ASISTENTES, AMENIDAD, FACILIDAD DE EXPRESION).	PUNTUALIDAD	
1.	M. EN C. FERAIN PARDO ORTIZ					
2.	ING. SALVADOR PEREZ VIRAMONTES					
3.	ING. DANIEL RIOS ZERTUCHE					
4.						
5.						
6.						
7.						
8.						
9.						
	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	

ESCALA DE EVALUACION DE 1 A 10

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 EDUCACIÓN CONTINUA	PERIODICO NOVEDADES ANUNCIO TITULADO DIVISION DE EDUCACIÓN 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 intensivo ¿Cuál es el horario - más conveniente para usted?

LUNES 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:

_____

_____

_____

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

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

METODOLOGIA Y ESTANDARES

EFRAIN PARDO ORTIZ

MAYO. 1985

Metodología y Estandares:

La metodología para desarrollo de sistemas es el conjunto - organizado de procedimientos que integran el proceso para el desarrollo de sistemas.

Los estandares para desarrollo de sistemas son el conjunto de normas, reglas, medidas y modelos utilizados en la planeación y control del proceso de desarrollo de sistemas.

Un proceso es el conjunto de fases sucesivas en este caso - las fases sucesivas para el desarrollo de sistemas.

Un proyecto es una forma de planeación sinónimo de un pro-- grama.

La primera estandarización que podemos realizar es en la -- terminología; para evitar confusiones con los terminos del proceso de administración de la función informática en general.

Llamaremos: metodología a los procedimientos integrados y - organizados para el desarrollo de sistemas; estandares a sus normas, reglas, medidas y modelos y proyectos a sus programas de -- trabajo.

Ciclo de Desarrollo de Sistemas:

Al proceso de desarrollo de sistemas lo podemos llamar ci-- clo dado que todos los sistemas de cómputo son sustituidos periodicamente, repitiendose su proceso de desarrollo en forma cíclica.

Las fases que conforman el ciclo de desarrollo de sistemas dividen un proyecto en subproyectos cada uno de los cuales da la oportunidad de revisar (controlar) el cumplimiento de los requerimientos estipulados (planeados).

Cuales fases son establecidas no es tan importante como el hecho de que sean fijas y bien definadas. Para mejorar los resultados todos los proyectos y áreas funcionales deberían adoptarlas.

Algunos de los beneficios de estandarizar las fases de los proyectos de desarrollo de sistemas son:

- Control.- Permite la revisión en varios puntos, en los que se mide la calidad y se ajustan los planes.

- Comunicación.- Estandarizar la terminología permite que los integrantes del proyecto, los usuarios y los directivos hablen el mismo lenguaje.

- Participación del usuario.- Al respaldar la metodología aumenta el compromiso de participación.

- Documentación.- Cada fase requiere una salida que debe producerse antes de terminar.

- Calidad.- La precisión en las salidas de cada fase aseguran que se termine el producto y su documentación.

- Estimaciones.- Permite calendarizar cada véz con mejores bases.

Las fases son a su véz divididas en tareas estandarizadas o actividades genéricas. Estas se presentan como actividades específicas para cada parte de un sistema en un proyecto particular.

G L O S A R I O.

METODO.- Modo de hacer con orden una cosa.

METODOLOGIA.- Ciencia del método.

PROCEDIMIENTO.- Método de ejecutar algunas cosas.

PROCESO.- Conjunto de fases sucesivas de un fenómeno.

TECNICA.- Conjunto de procedimientos y recursos se que se sirve una ciencia.

DOCUMENTAR.- Probar, justificar la verdad de una cosa con documentos.

ESTANDARD.- (Anglisismo) Norma, medida, patron, modelo, regla fija.

REGLA.- Modo de ejecutar una cosa.

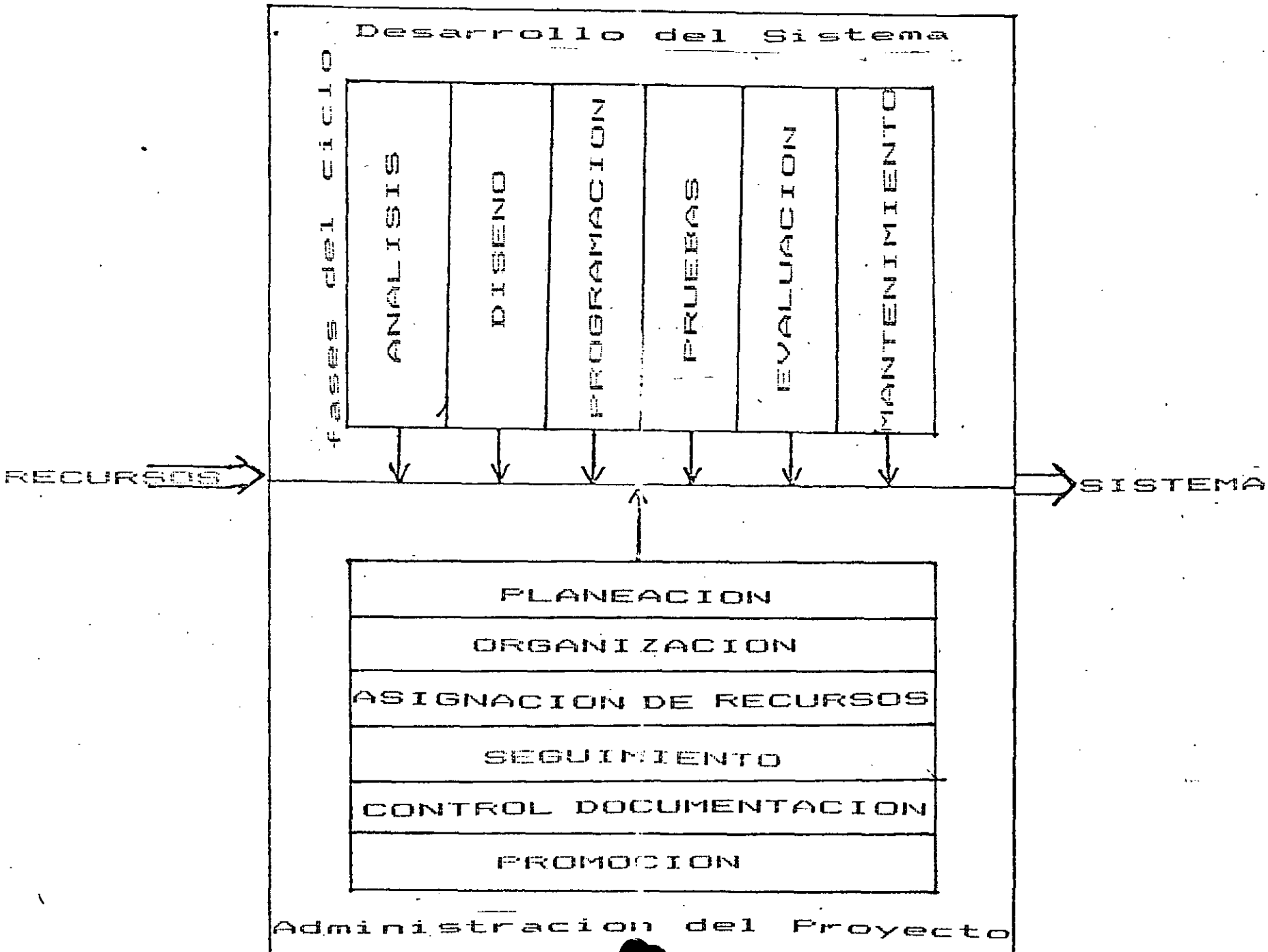
NORMA.- Regla que se debe seguir o a que se deben ajustar las operaciones.

MODELO.- Ejemplar o forma que uno se propone y sigue en la ejecución de una obra.

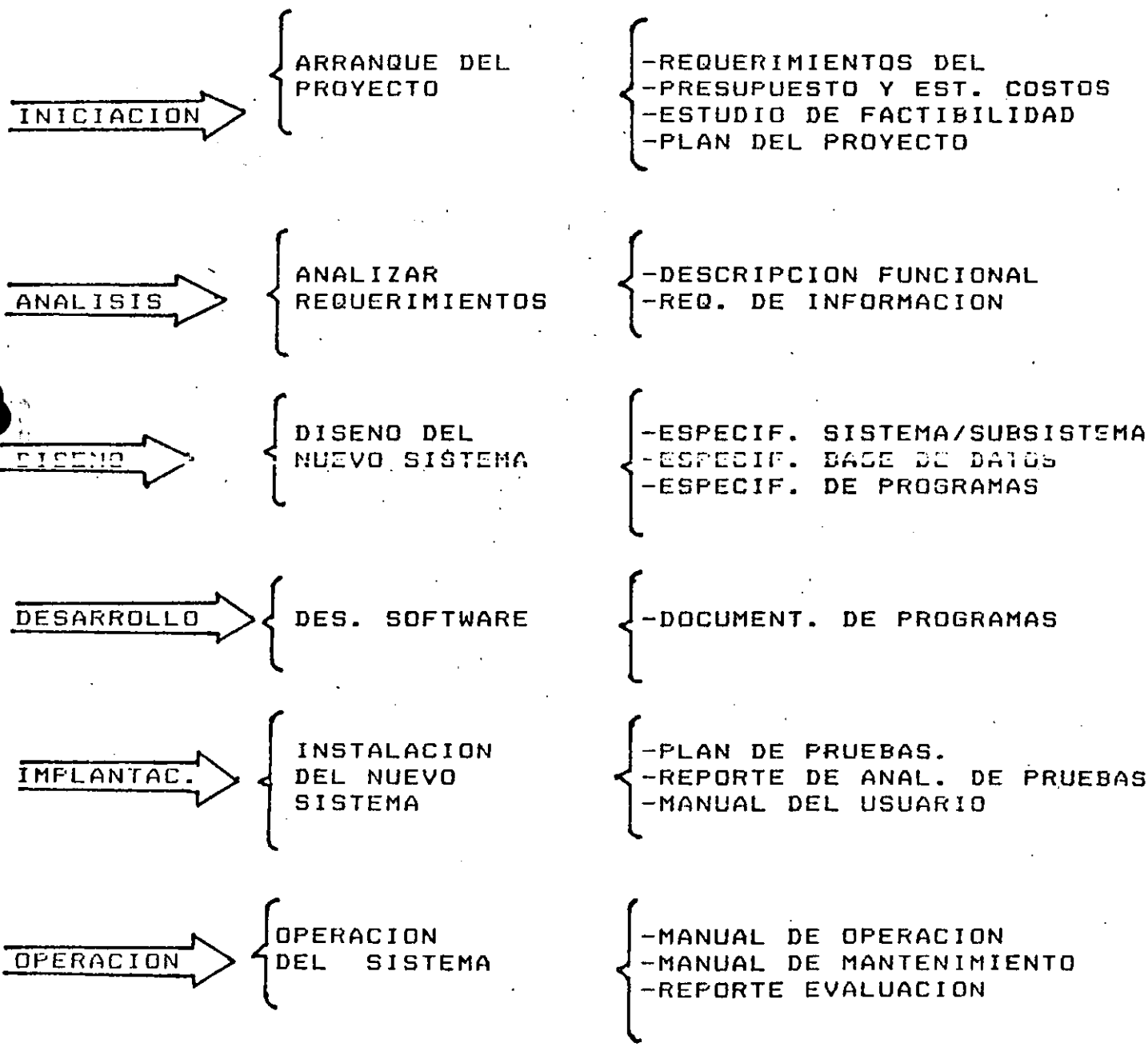
MEDIDA.- Unidades que se contemplan para medir un trabajo.

PATRON.- Que sirve como muestra para sacar otro igual.

DESARROLLO DE PROYECTOS



FUNCIONES Y PRODUCTOS DE VIDA DE UN SISTEMA



CARPETAS PARA DOCUMENTACION DE PROYECTOS

CARPETA	CONTENIDO
DEL PROYECTO	-CORRESPONDENCIA -REQ. DEL USUARIO -PRESUPUESTO Y EST. COSTOS -PLAN DEL PROYECTO
DEL SISTEMA	-EST. DE FACTIBILIDAD -DESCRIP. FUNCIONAL -DOCTO. REQ. INFORMACION -REPORTE DISENO SISTEMA: *ESP. SISTEMA/SUBSISTEMA *ESP. BASE DE DATOS *ESP. PROGRAMAS *PLAN DE PRUEBAS *MANUAL DEL USUARIO -REPORTE DE EVALUACION
DEL PROGRAMA	-DOCUMENTACION DEL PROGRAMA -REPORTE ANALISIS PRUEBAS
DE PRODUCCION	-MANUAL DE OPERACION -MANUAL DE MANTENIMIENTO
DE USUARIO	-MANUAL DEL USUARIO

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

TEMA I

- LA NECESIDAD DE INFORMACION
- EL CONCEPTO DE SISTEMA Y SUS CARACTERISTICAS
- LAS ORGANIZACIONES COMERCIALES
- SISTEMAS DE INFORMACION
- CONCEPTOS SOBRE INFORMACION Y DATOS
- EL ESTUDIO DE LOS SISTEMAS

ING. JORGE I. EUAN AVILA

MAYO, 1985

LA NECESIDAD DE INFORMACION.

Las actividades que se desarrollan para el proceso de datos dan como resultado información. La información es algo que requerimos las personas, los animales, las organizaciones, etc. que sirve para guiar el comportamiento. En nuestro caso, cuando vamos manejando nuestro automovil estamos procesando datos, para obtener información que nos permita conducir el automovil de la mejor manera. De forma similar, una organizacion procesa datos para obtener información que guie sus actividades hacia el logro de sus objetivos.

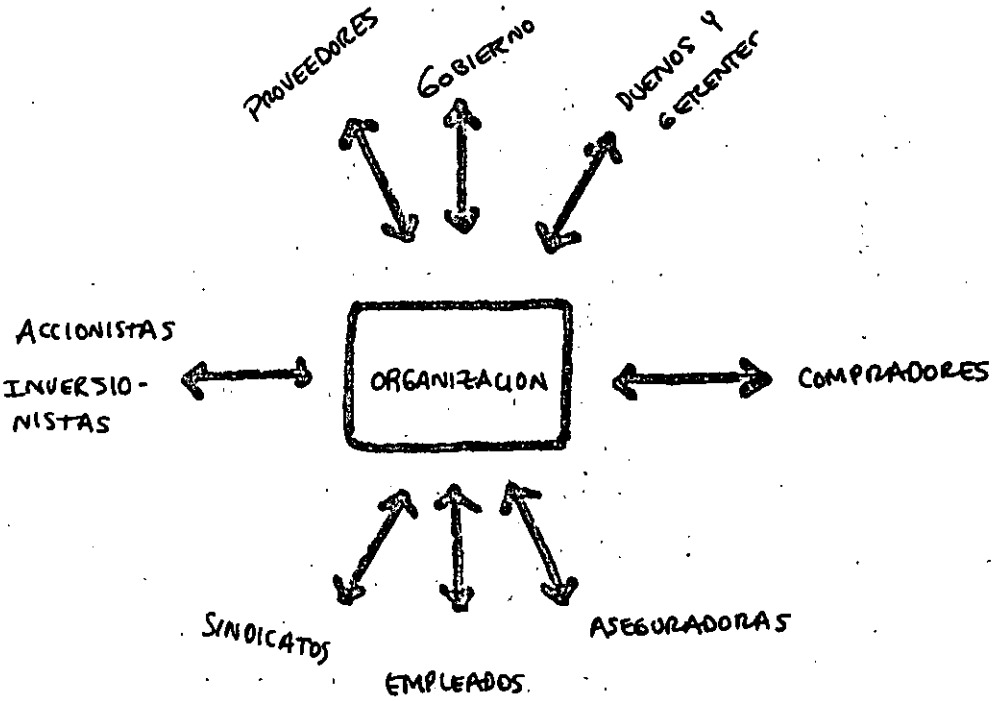
En las actividades del proceso de datos la participación de las personas es cada dia mayor; en los EU se estima que cerca del 60% de la fuerza de trabajo esta orientada hacia estas actividades. Para tener una idea de lo que este porcentaje significa, en los EU solo el 2% de la fuerza de trabajo esta dedicada a la agricultura. Esto, refleja la importancia del procesamiento de datos como una actividad fundamental en las organizaciones actuales y se debe fundamentalmente a :

-lo complejo de las relaciones actuales

-las nuevas tecnicas de administración(administración cientifica)

-la tecnología de las computadoras

En la siguiente figura puede verse la cantidad de entidades con las que una organización tiene que relacionarse y por consiguiente demanda de información.



Como ejemplo podemos resaltar algunos aspectos de las formas de relación considerando dos medios ambientes diferentes:la tienda de la esquina y una cadena de tiendas de autoservicio. La tienda de la esquina, seguramente la relación con el gobierno es siemple -pago de cuotas fijas- ; pero la de una cadena de tiendas seguramente es complejo -cálculos diversos para el pago de cuotas- por el estilo podriamos extender esto hacia otros sectores.

Frederick Taylor, Franck y Lilian Gilbreth desarrollaron técnicas que permitieran operar en forma eficiente a las organizaciones, estas constituyen la base de lo que se conoce como técnicas de administración científica.

La disponibilidad de la tecnología de las computadoras se demanda a diario para su uso. Información que antes era demasiado costosa o impráctica de obtener es ahora posible a un costo rasonable. Con esto a crecido la demanda de este tipo de información.

EL CONCEPTO DE SISTEMA Y SUS CARACTERISTICAS.

La información, es fundamental para el hombre, la familia, los negocios, las escuelas, los gobiernos, etc. y cualquiera de ellos es un sistema; dentro del cual, existe un subsistema llamado sistema de información que los provee de información para el logro de sus objetivos.

El hombre a observado que el medio ambiente en el que vive requiere organización, interacción y orden para funcionar adecuadamente. Esto, no es una excepción en otras actividades donde se desenvuelve. El desarrollo de la partida doble en la contabilidad es una forma sistemática de manejar transacciones en un negocio.

-Definición.

El término sistema, se refiere a un grupo organizado de componentes relacionados funcionalmente entre si. Un sistema existe debido a que es diseñado para lograr un objetivo.
El cuerpo humano es un sistema, compuesto de esqueleto, aparato respiratorio, aparato circulatorio, sistema nervioso, etc. solamente cuando estos subsistema se encuentran funcionado de manera coordinada. De forma similar podemos hablar de un sistema productivo y de su organización, como un sistema compuesto de departamentos interrelacionados llamados subsistemas. De hecho,cualquier cosa puede considerarse como un sistema el hombre,las organizaciones,las fabricas,el sol, son ejemplos de sistemas.

Ninguno de los subsistemas es independiente de los demas y cuando estan propiamente coordinados podemos decir que el sistema funciona exitosamente y cumple con su cometido.

3

De lo anterior, podemos decir que un sistema es un grupo organizado de componentes llamados subsistemas conjuntamente ligados de acuerdo a un plan para el logro de un objetivo.

-Características primarias de un sistema.

Las caracteristicas que se han observado en un sistema son:

-es abierto(cerrado) cuando interactua con el medio ambiente

-tiene dos o mas subsistemas

-hay interdependencia entre los subsistemas

-es auto-ajustable

-es auto-regulable

-tiene un proposito

-tiene estructura

-tiene un comportamiento

-tiene un ciclo de vida

-Clasificación de los sistemas

Los sistemas en general son clasificados por el tipo y grado de complejidad. Un sistema puede ser determinístico o probabilístico y en cada caso simple, complejo o excesivamente complejo.

Un sistema al que puede predecirsele sus salidas, debido a que no se espera que su comportaminto varie' es un sistema detrministico. Por ejemplo una sumadora.
Los sistema probabilisticos, son descritos en terminos de probabilidad. En la medida que estos se hacen excesivamente complejos la salida se vuelve menos predecible.

-Representacion de los sistemas

En el trabajo del analista de sistemas, el analista espera poder definir al sistema. El uso de un modelo hace facil visualizar las relaciones entre los elementos del sistema y de explorar formas que mejoren el entendimiento del sistema. Un modelo es una representacion para un sistema real o planeado. El objetivo del uso de un modelo es señalar los elementos significantes y las interrelaciones de un sistema.

4

Algunos tipos de modelos son:

-modelos esquemáticos

-modelos para sistemas de flujo

-modelos para sistemas estáticos

-modelos para sistemas dinámicos

LAS ORGANIZACIONES COMERCIALES

La tendencia del hombre es crear organizaciones. La razon, es que tales organizaciones las crea para alcanzar objetivos que de forma individual no lograria. Una organización comercial tiene como objetivos la comercializacion de productos o servicios que la clasifica como comercial.

La información que se maneja en una organización comercial puede clasificarse como aquella que se genera de las actividades internas de la organizacion(información interna) y la proveniente del medio ambiente(informacion externa).

La informacion en una organizacion se requiere por varios motivos:

> Es utilizada para comunicarle a la gente que en ella participa los objetivos que se persiguen e instruirlos en las políticas y procedimientos requeridos para el logro de los objetivos.
> Es usada para determinar las percepciones y deducciones a sus empleados; la forma de servir a sus clientes, responder a preguntas que surgen durante la operacion normal de la empresa, etc.

La demanda de información en una organización se encuentra clasificada por el nivel donde surge . Existen tres niveles organizacionales :

-nivel operacional	donde se desarrollan las actidiarias
-nivel táctico	donde se superviza y planean las actividades diarias
-nivel estratégico	donde se encuentra la administración de alto nivel y la planeacion a largo plazo

-Objetivos de una organización

5

Los objetivos en una organización se encuentran clasificados de acuerdo a los niveles organizacionales que existen.

-objetivos del nivel operacional

-objetivos del nivel táctico

-objetivos del nivel estratégico

Para establecer los objetivos en cada nivel, se requiere de una gran cantidad de información. Para establecer los objetivos de ventas en un negocio, se requiere conocer que factores influyen en las ventas y como se espera que estos factores varíen en el período para el que fueron establecidos los objetivos.

-Componentes de una organización comercial

Las organizaciones comerciales estan compuestas de personas que trabajan en diversas actividades llamadas funciones comerciales, estas funciones pueden considerarse como subsistemas y constituyen los componentes principales de la organización. Para hacer su trabajo en forma eficiente las personas hacen uso de edificios, equipos, etc.

Los componentes principales de una organización son los siguientes:

-mercado

-ventas

-producción

-control de inventarios

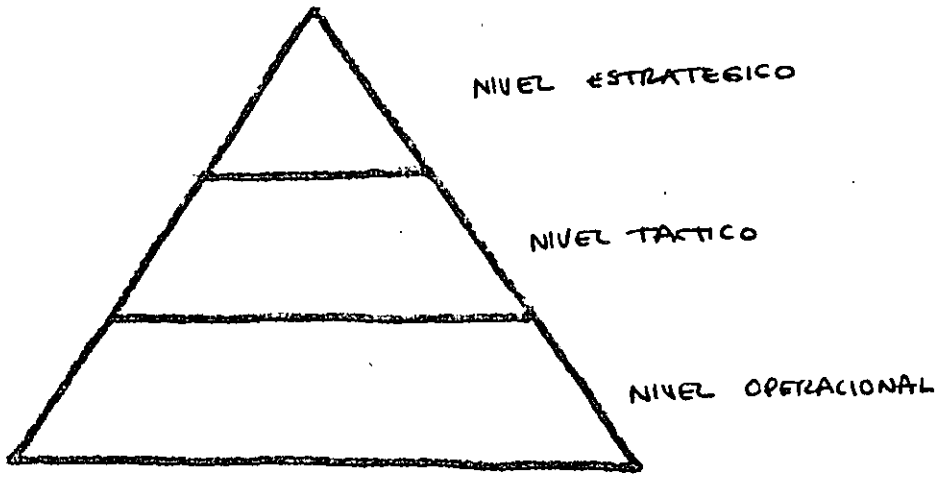
-contabilidad

-jurídico

-seguridad

-relaciones públicas

6



organización piramidal

-Estructura de una organización comercial

La estructura de una organización comercial es la forma en la cual la autoridad y las responsabilidades se distribuyen entre los empleados y los gerentes.

-Comportamiento

El comportamiento de una organización esta determinado por sus procedimientos, los cuales especifican la secuencia de actividades que deben realizarse de acuerdo a las politicas de la organización para el logro de los objetivos.

Los procedimientos, son guias para los empleados acerca de como deben realizar sus tareas y son entrenados para realizar eficientemente estas tareas. El entrenamiento, es una forma de transferir información y de compartir los beneficios de las experiencia con los nuevos empleados.

La generalidad de procedimientos se dictan dentro de la empresa aunque algunos son dictados fuera de la organización. Como ejemplo de esto tenemos cierto tipo de demandas del gobierno.

-Ciclo de vida

Las estadisticas en los EU muestran que para las organizaciones comerciales los tres primeros años de vida son dificiles, pero despues de esto la organización madura y alcanza sus objetivos. Despues de veinticinco años se observa que las empresas tienen fallas y que sus objetivos ya resultan demasiado viejos.

7

Es frecuente encontar altos costos de administración por componentes que estan fuera de su vida útil pero que continúan existiendo. Para evitar esto, hay que revisas constantemente estos componentes y cuando los objetivos han sido alcanzados debera el componente ser abolido o tomarlo sobre nuevos objetivos.

Zero-base budgeting es un ejemplo de una técnica usada para controlar el ciclo de vida de un componente.

SISTEMAS DE INFORMACION

Un sistema de información es un componente de una organización. Su propósito, es obtener información dentro y fuera de la organización y hacerla disponible a todos los otros componentes en la forma como estos la demanden y tambien presentar información a los que se encuentran fuera en la forma como la necesiten.

-Objetivos

Los sistemas de información tienen como objetivo responder a las necesidades de información en los tres niveles organizacionales. La información debe ser presentada el la forma adecuada para aquellos que la requieran (textos,graficas,grabaciones,etc).

La información para que sea útil debe presentarse en el tiempo adecuado y estar disponible a un costo razonable.

-Componentes de un sistema de información

Un sistema de información tiene tres tipos de componentes, datos, sitemas para procesar datos y canales de comunicación.

Fechas,cantidades,nombres, son ejemplos de datos. Los datos son adquiridos por el sistema de su medio ambiente y estos datos son conocidos como entradas al sistema. Un tipo particular de dato es la realimentación. Datos que resultan de las actividades del sistema de información y que son tomados como entradas al sistema son llàmados datos de realimentación. La realimentación es importante para medir el éxito de la organización.

Otro tipo de componente es el sistema de procesamiento de datos, con el cual se manipulan los datos. Dentro del sistema de información pueden existir varios sistemas de procesamiento de datos.

En los sistemas de procesamiento de datos son las gentes quienes procesan los datos, auxiliandose en algunos casos de máquinas como podria ser una computadora.

El tercer componente son los canales de comunicación entendido como los mecánismos que permiten pasar información de un componente a otro. Ejemplo de canales de comunicacion son el sistema telefónico de la organización el correo o mensajería, memorándums,seminarios. etc.

-Estructura de un sistema de información

La forma como diferentes sistemas de procesamiento de datos estan relacionados unos con otros y con los usuarios constituye la estructura del sistema de información.

-Comportamiento

El comportamiento que se espera de un sistema de información es que logre los objetivos de almacenamiento de información y que provea de información a la organización en la forma, tiempo y costo que resulte apropiado. Para lograr este comportamiento se requiere del establecimiento de procedimientos.

Identificar la fuente de los datos, los componentes del procesador de datos para ser utilizados y especificar la forma, el costo y el tiempo de la información constituyen los procedimientos que gobiernan el comportamiento del sistema. Los empleados de las organizaciones son usualmente entrenados en los procedimientos requeridos por la organizacion para el manejo de la información.

-Ciclo de vida

Como una organización está en cambio permanente, las necesiades de información también. La vida de un sistema de información cesa cuando se decide que la necesidade de información ya no existe.

Cuando un proyecto se inicia nace un sistema de información y cuando este termina muere el sistema de información. En algunos casos puede morir parte del sistema o todo antes que el proyecto termine.

-Ejemplos

Algunos ejemplos de sistemas de información son:

-sistema de información contable (contabilidad)

-sistema de control de inventarios

-sistema de cuentas por cobrar

-nómina

-otros

CONCEPTOS SOBRE INFORMACION Y DATOS

-Objetos de información

Entidades.- Una parte esencial para conducir las actividades de una organización es obtener información acerca de las entidades a las cuales sirve o son usadas por la organización. Son ejemplo de entidades en una organización comercial los clientes, los dueños , los productos, los empleados, etc.. En otras palabras una entidad es una persona, cosa o lugar.

Eventos.- Otro aspecto esencial es el de obtener información sobre los eventos que ocurren durante el curso de las operaciones en una organización. Un evento es una cosa que sucede en un tiempo particular. Son ejemplo de eventos cobrar una factura, pagarle a un empleado, aseguar un equipo, modificar el precio unitario de un artículo, etc.

En las operaciones comerciales eventos que resultan por el intercambio de valores son llamados transacciones. El registro de transacciones asi como de otros eventos son usados para actualizar registros de entidades tales como cantidad en un inventario, precio unitario de un producto, etc.

Las entidades y eventos, pueden ser reconocidos, recordados, y descritos en terminos de sus atributos. Los atributos, son hechos acerca de las entidades y eventos. Algunos tipos de atributos que existen son:

-identificadores	hechos usados para distinguir objetos de informacion de unos a otros.
-descriptores	hechos relacionados con la percepcion de los sentidos acerca de los objetos de informacion.(color, peso estatura,..)
-localizadores	hechos que permiten determinar el lugar donde el evento ocurre o posicion de la entidad.(direccion, cubiculo,casilla,apartado postal,..)
-temporales	hechos que permiten determinar el momento en que ocurre un evento. (fecha,hora del dia,..)

-relacionales	Hechos que permiten describir las relaciones que existen entre eventos o entidades.(padre,hijo,gerente, subordinado,..)
-clasificadores	hechos que determinan la manera en la cual eventos y entidades estan relacionados con la organización. (cliente,empleado,proveedor,..)
-condicionales	hechos que describen el estado de eventos o entidades.(pendiente, en proceso,entregado,...)

Estos atributos, son usados en combinación para describir completamente un evento o entidad.

-Definiciones

Dato.- Los datos, son hechos que describen eventos y entidades. Los datos son comunicados por varios tipos de símbolos tales como letras, números, dibujos, etc. La combinación o arreglo de estos símbolos permiten la representación de un hecho.

Información.- La información, es una colección de datos significativos y relevantes que describen eventos y entidades. Un dato significativo es aquel que consiste de símbolos reconocibles, es completo y expresa una idea no ambigua. Un datos relevante es aquel que puede ser usado para dar respuesta a una pregunta.

Registro de la información.-El hecho de registrar datos selectos para referencias futuras es llamado captura de datos. Esencial a la captura de datos es el registro del contexto, el cual debe ser explícito y es el hombre quien los proporciona. El contexto para un data-item (cadena de caracteres o dato numerico) es frecuentemente indicado por su nombre. Ejemplo:

nombre de un data-item (contexto)	valor de un data-item
sexo	masculino
edad	25
color	moreno

Cierto tipo de información tal como nombre, dirección, fecha son llamados agrupaciones de data-items ya que consisten de una agrupación de data-items elementales.
Ejemplo:

fecha de nacimiento

11

dia mes ano

Registro.- Una colección de data-items que comparten un contexto común acerca de una entidad o evento es llamada un registro. Los registros con el objeto de facilitarles su identificación se les da un nombre.

-Formatos

La forma establecida para definir un registro es llamada formato del registro. Las características de un data-item (en este caso también llamado campo) que deben indicarse en la especificación del formato son:

-nombre

-secuencia

-valores validos

-longitud

-tipo

-Archivos

De la misma forma como los data-items son organizados en registros, estos se organizan en archivos. Cuando todos los registros del mismo tipo son agrupados en una sola colección de información, la colección es llamada archivo.

A.

12

EL ESTUDIO DE LOS SISTEMAS

-El enfoque de sistemas

El enfoque de sistemas(ES) es posiblemente la técnica más utilizada para el estudio de los sistemas. El ES es un proceso de desarrollo ordenado y analítico que se puede utilizar continuamente para analizar,evaluar y diagnosticar la naturaleza de un sistema, asi como los resultados de su desempeño para captar todo lo necesario a esos fines y proveer la continua autocorrección del funcionamiento del sistema con el propósito de alcanzar los objetivos propuestos.

-Pasos del enfoque de sistemas

El proceso de desarrollo puede resumirse en las siguientes etapas o pasos.

-análisis

-diseño

-desarrollo

-instrumentación

-evaluación

El ES es un metodo que sirve para descubrir los problemas existentes en el sistema para elegir o disenar mejores y modernos recursos para hacerlo funcionar adecuadamente.

El ES es un punto de vista de actuar de manera logica,ordenada y cientifica. Es en este sentido que el ES se opone a la conjetura, intucion o al jucio subjetivo que tan a menudo es usado.

El ES se basa el los 5 pasos fundamentales mostrados anterirmente; las actividades que se desarrollan en cada paso podemos resumirlas en forma general de la siguiente manera:

Análisis

Representación o caracterización del sistema

-se describen dando detalles concretos y útiles toda la información acerca del sistema

-se definen y analizan las entradas

-se describen los procesos y sus características

-las salidas se definen,identifican y cuantifican

13

-la estructura del sistema se describe para aclarar las relaciones entre los componentes

-el ambiente se define para aclarar su interacción con el sistema

-el flujo de información se define y en especial los mecanismos de realimentación

-finalmente se estudia la relación entre las salidas del sistema y los objetivos y metas

Descubrimiento de problemas dentro del sistema

el descubrimiento de anomalías dentro del sistema constituye el estudio minucioso de la manera en que el sistema logra sus salidas, la eficacia conque lo hace y el grado que alcanza su rendimiento.

Diseño

Con el análisis se han descubierto fallas en el sistema y el paso siguiente es diseñar nuevos métodos para el sistema. Segun sea la naturaleza del problema el diseño puede comprender:

-un nuevo sistema

-cambio de compónentes

-cambio en las entradas y/o salidas

-cambios en los procesos

Para llevar acabo el diseño, se prepara un plan detallado con todas las especificaciones requeridas para modoficar al sistema.

Desarrollo

Sobre la base de los diseños se procede a la construcción, edificación, explicación o lo que sea necesario para el desarrollo del sistema.

Istrumentación

Una vez que se ha desarrollado el nuevo componente, elemento, proceso o sistema debe incorporarse a lo ya existente.

La instrumentación requiere del establecimiento de prioridades y secuencia de pasos para incorporar el nuevo

14

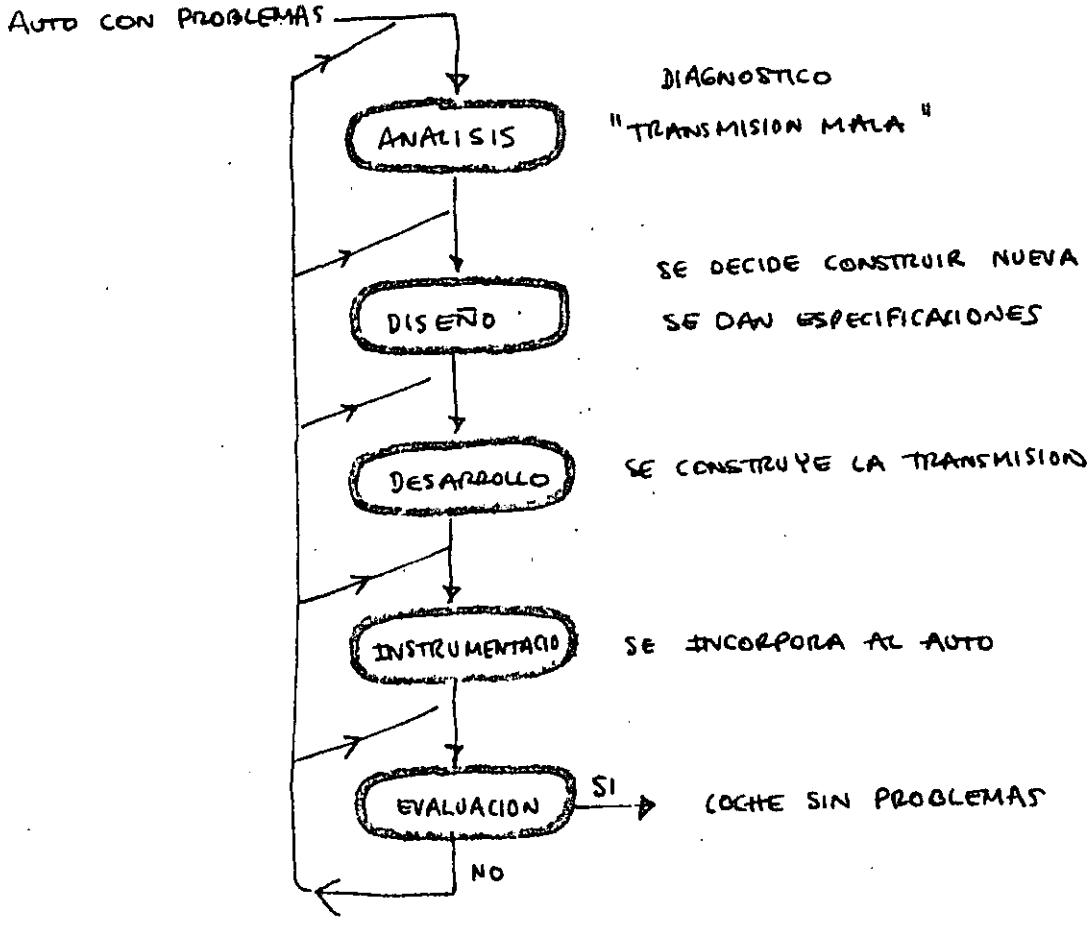
elemento, ya que si se fracasa en la instrumentación los demás elementos pueden transtornar al sistema temporalmente al tratar de ajustarse a los requerimientos del nuevo. Otra falla sería que el componente en si mismo fallara.

Para asegurar una exitosa instrumentación es necesario anticipar los cambios que exijiran los demás componentes.

Evaluación

Despues de haber istrumentado el nuevo componente este debe evaluarse. ?pudo el nuevo componente resolver los problemas planteados?, ?provoco nuevos problemas?.
La evaluación debe hacerse en terrminos de los objetivos que se establecieron en las etapas de análisis y diseño.

Consideremos el siguiente ejemplo para ilustrar los pasos del enfoque de sistemas.



-El enfoque de sistemas aplicado al desarrollo de sistemas de información

En cualquier plan para el desarrollo de sistemas de información, encon-

15
tramos que los pasos del ES gobiernan de forma general el proceso. Este plan tambien es llamado "ciclo de vida del desarrollo de un sistema".

-Ciclo de vida

Dependiendo de las diversas corrientes que existen para el desarrollo de sistemas encontramos que los pasos son:

Para Elias M Awad los pasos que sugiere son:

- análisis del sistema
  - definición inicial del problema
  - recolección de datos
  - organización de los datos
  - análisis de costo beneficio
  - definición final del problema
- diseño
  - diseño de salidas
  - diseño de entradas
  - diseño de archivos
  - diseño de procesos
  - documentación
- prueba e implementación
  - codificación de los programas
  - preparación de datos de prueba
  - prueba general de los archivos
  - corrida en paralelo

Para Kenniston W.Lord y james B.Steiner los pasos que sugieren son:

- analisis del sistema
- diseño del sistema

16

-selección y adquisición del equipo

-programación

-prueba y conversión

-instalación

-operación

-mantenimiento

-evaluación

A.

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

TEMA II

ANALISIS ESTRUCTURADO

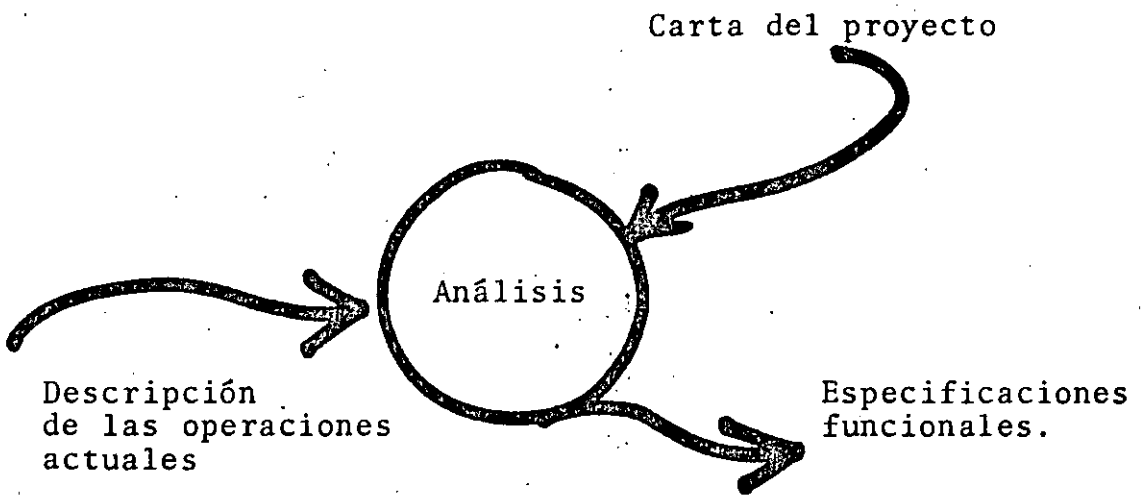
- LA ETAPA DEL ANALISIS
- HERRAMIENTAS DEL ANALISIS ESTRUCTURADO
- CONSTRUCCION DEL DIAGRAMA DE FLUJO DE DATOS
- CONSTRUCCION DEL DICCIONARIO DE DATOS
- CONSTRUCCION DE MINIESPECIFICACIONES

MAYO, 1985

Análisis estructurado

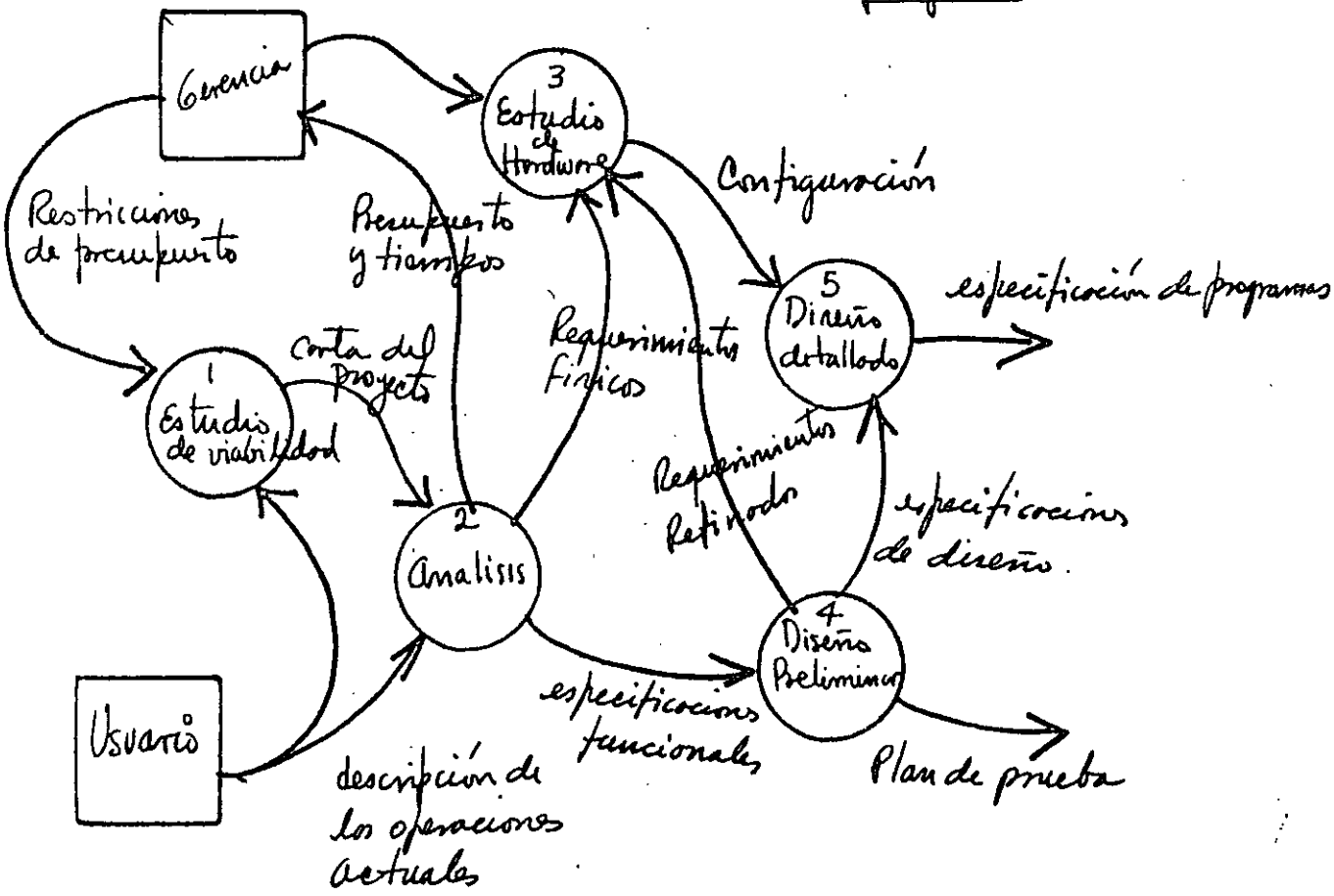
Cuando Ed Yourdon habló de análisis estructurado, la idea fue largamente especulada. El no tenía resultados actuales sobre proyectos completos para reportarlo. Sus ideas estaban basadas de la simple observación de algunos principios de particiones de arriba hacia abajo (top-down) usadas por los diseñadores y que se podían aplicar en la fase de análisis. Desde este tiempo ha habido una revolución en la metodología del análisis.

¿Qué es el análisis?



Análisis es el proceso de transformar una cadena de información acerca de las operaciones corrientes o actuales y de los nuevos requerimientos a una descripción ordenada y rigurosa de un sistema para ser construido. Esta descripción es tàmbién llamada especificaciones funcionales o especificacioń del sistema.

Figura: Análisis en el contexto del ciclo de vida del proyecto.



En el contexto de el ciclo de vida del proyecto para el desarrollo de sistemas, el análisis se encuentra muy al principio, solamente precedido por el estudio de viabilidad durante el cual la carta del proyecto es generada. La carta de proyecto contiene consideraciones que gobiernan el desarrollo, cambios que deben ser considerados etc. La fase de análisis concierne principalemente con la generación --- de especificaciones de el sistema para ser construído.

Especificamente la tarea esta compuesta de las siguientes actividades:

- interacción con el usuario
- estudio del medio ambiente actual
- negociación
- diseño externo del nuevo sistema
- diseño de formatos de E/S

. estudio de costo beneficio
. escritura de las especificaciones y
. estimación

¿Que es el análisis estructurado?

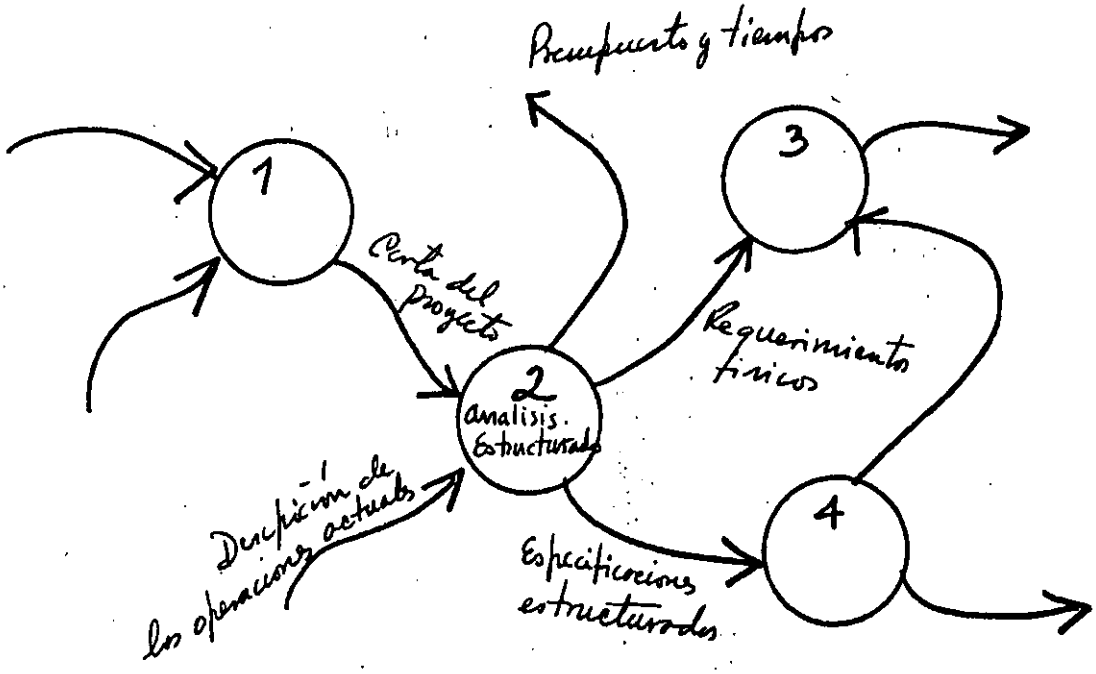


Figura 3:
Analisis estructurado en el contexto de ciclo de vida del proyecto.

El análisis estructurado es una disciplina moderna para conducir la fase de análisis. En el contexto de el -- ciclo de vida del proyecto su única diferencia aparente_ es un nuevo producto llamado "especificaciones estruc-- turadas". Esta nueva clase de especificación tiene las

siguientes características:

- . es gráfica, compuesta mayormente de diagramas
- . es particionada, no es una sola especificación sino una red conectada de "mini especificaciones".
- . es de arriba-abajo (top-down), presentada en modo -- jerarquico y progresivamente de los niveles superiores mas abstractos hasta los niveles inferiores mas_ detallados.
- . es mantenible, una especificación puede ser actualizada para reflejar cambios en los requerimientos.
- . es un modelo en papel del sistema, el usuario puede_ trabajar con el para perfeccionar su visión de las - operaciones tal y como seran en el nuevo sistema.

En la siguiente figura se representa en un solo proceso el análisis estructurado. Veamos en detalle el proceso.

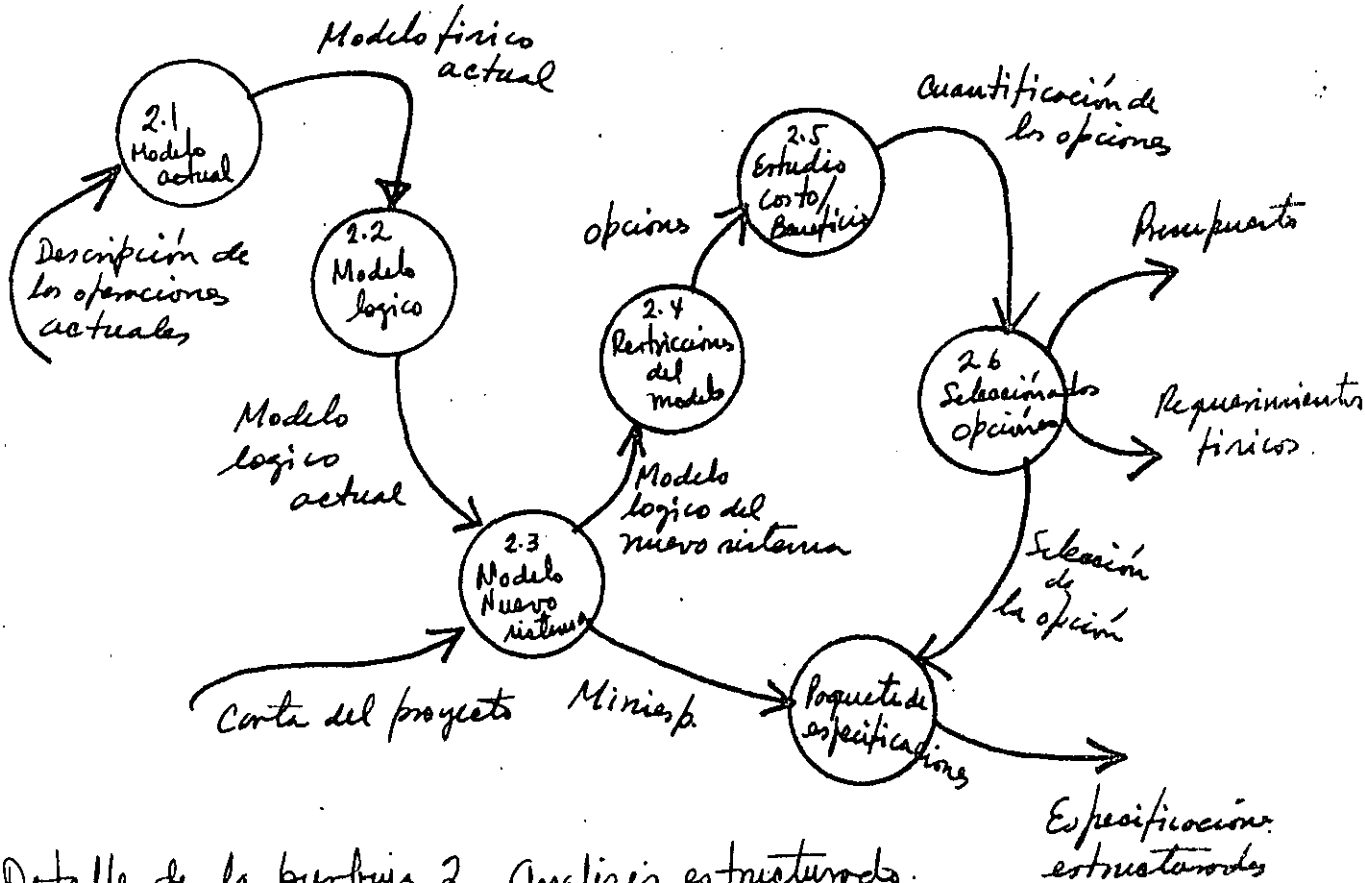


Figura: Detalle de la burbuja 2. Analisis estructurado.

Note que la figura tiene las mismas entradas y salidas que la burbuja 2 de la figura anterior. Pero muestra las -- transformaciones en considerable mas detalle. Consideran los autores que esta figura da mas información que - - cientos de palabras. Bastará una definición de los componentes y del flujo de información para terminar de comprender el proceso.

Proceso 2.1. Modelo del sistema actual: Hay casi siempre un sistema actual. El análisis estructurado nos - - tendrá que construir un modelo "fisico" en papel del sistema actual y usarlo para perfeccionar el entendimiento - del medio ambiente actual. La justificación para la - naturaleza física de este primer modelo es que su propósito es ser una representación verificable de las operaciones actuales y deberán ser fácilmente entendibles por el grupo de desarrollo.

Proceso 2.2. Derivar un equivalente lógico.- El equivalente lógico del modelo físico es uno que esta divorciado de los "comos" de las operaciones actuales y en su lugar se concentra en los "que". En lugar de una descripción de la forma como se lleva a cabo la política se hace una descripción de la política en sí misma.

Proceso 2.3. Modelo del nuevo sistema.- Aquí es donde el trabajo mas importante de la fase de análisis se realiza, la invención del nuevo sistema. La carta de proyecto que ha recogido las diferencias y las diferencias potenciales entre el medio ambiente actual y el nuevo y el - - modelo lógico del sistema existente son los elementos del análista para construir el nuevo modelo junto con la documentación del futuro medio ambiente. El nuevo modelo - presenta al sistema como un grupo particionado de procesos elementales. El detalle de estos procesos son - - ahora especificados utilizando una "miniespecificación" - por proceso.

Proceso 2.4. Restricciones del modelo. El nuevo modelo es demasiado lógico para nuestros propósitos debido a que no se ha establecido cuanto de lo declarado se hace dentro y cuanto fuera de la máquina. Es en este punto donde el analista establece los límites entre el hombre y la máquina y se limita el alcance de la automatización.

Típicamente estos se hace mas de una vez con el objeto de dar varias alternativas para la selección de procesos.

Proceso 2.5 Medidas de costo y beneficio.- El estudio de costo-beneficio es realizado para cada una de las opciones. Cada uno de los modelos tentativos junto con sus parámetros asociados de costo-beneficio son presentados en forma cuantificada.

Proceso 2.6. Selección de la opción.- Las opciones cuantificadas son analizadas y una de ellas es seleccionada como la mejor.

Proceso 2.7. Presentación de especificaciones.- Ahora todos los elementos de la especificación estructurada son armados y presentados. El resultado es: el nuevo modelo -- físico seleccionado, el conjunto integrado de miniespeci-- ficaciones y posiblemente agunas tablas de contenido, ---- resumenes, etc.

¿Qué es el modelo?

El modelo al que se ha referido para la representación del sistema es un modelo escrito. En la convención de Análisis estructurado este modelo se logra con el uso de un - - "diagrama de flujo de datos" y de un "diccionario de da-- tos".

1.- Diagrama de flujo de datos.- Es una representación en forma de red de un sistema. Representa al sistema en términos de los componentes de los procesos y declara todas las interfases entre los componentes.

2.- Diccionario de datos.- Es un conjunto de definiciones de las interfases declarados en el diagrama de flujo de datos. Se define cada una de esas interfases en términos de sus componentes.

El modelo del sistema tiene diferentes usos en el análisis estructurado.

1.- Es una herramienta de comunicación.- Los analistas y los usuarios tienen fallas en la comunicación, el modelo es esencial para las discusiones y que estas puedan conducir a un entendimiento común. Se le considera al modelo como la mas importante ayuda para la comunicación.

2.- Es el marco de referencia para las especificaciones. El modelo declara las piezas que componen el sistema y las partes de esas piezas, hasta esas que ya no puede ser subdivididas. Los niveles básicos, son acompañados de una miniespecificación para completar su especificación.

3.- Es el punto de inicio para el diseño. Debido a que el modelo es el mas elocuente elemento que dió pie a los requerimientos tendrá una gran influencia en el trabajo que se haga en la fase de diseño.

Algunas fallas del análisis clásico según Yourdon.

Las principales fallas del análisis clásico los encontramos en el proceso de especificación del sistema. Algunos de estos son:

1.- Las especificaciones funcionales en el análisis clásico son generalmente una narrativa en un documento solamente leible en forma serial del principio al fin.

2.- Las especificaciones funcionales son imposibles de actualizar.

3.- Debido a que las especificaciones funcionales no son entendibles por si mismas no podemos mostrarle nada al usuario sino hasta el final del análisis, esto no permite la interacción que podría servir para refinar y perfeccionar el producto.

Como ayuda el análisis estructurado según Yourdon.

El análisis estructurado ayuda a resolver los problemas de la fase de análisis ya que:

1.- Ataca el problema del tamaño por particiones
2.- Ataca el problema de la comunicación en forma interactiva y con una inversión de los puntos de vista.
3.- Ataca el problema del mantenimiento de especificaciones por redundancia limitada.

El concepto de particionar o descomponer funcionalmente puede ser familiar para los diseñadores como el primer paso para crear un diseño estructurado. Su potencial valor en el análisis fue evidente desde el principio, sistemas grandes no podían ser analizados sin alguna forma de particionar concurrentemente. La directa aplicación de las herramientas del diseño como HIPO causaron mas problemas en lugar de resolverlos. Despues de algunos experimentos el análisis estructurado fue apoyado con una nueva herramienta llamada diagrama de flujo de datos.

Sus ventajas son las siguientes:

1.- La apariencia del diagrama de flujo de datos no es para todos espantosa. Parece ser una simple fotografía del elemento que esta en discusión. Nunca se tendrá que explicar una convensión arbitraria, nunca se explicará una cosa a todos. Simplemente se usan diagramas.

2.- Los modelos tipo red como los que se construyen para el diagrama de flujo de datos puede resultar familiar para algunos usuarios. Pueden los usuarios tener nombres diferentes para este tipo de herramientas pero el concepto es similar.

3.- El hecho de particionar con el diagrama de flujo de datos llama la atención de las interfases que resultan del proceso de particionar. La complejidad de los interfases es un indicador importante del esfuerzo que se haga por particionar.

El término "comunicación iterativa" esta usado para describir el intercambio de información en una doble dirección es una de las características de las sesiones mas productivos de trabajo. El periodo de dialogo entre el analista y los usuarios debe reducirse y procurar realimentación. Un entendimiento temprano es siempre imperfecto. También se menciona como parte del análisis la "inversión de puntos de vista". Las especificaciones clásicas describen que hace la computadora, en que orden lo hace, usando términos que son releventas para la computadora y para las gentes de la computadora. Frecuentemente se limita también a la discusión de los procesos internos de la máquina y la transferencia de datos y casi nunca se especifica los casos que suceden fuera de los límites entre el hombre y la máquina.

Desde el punto de vista de la máquina es natural y útil para el grupo de desarrollo pero para los usuarios y su grupo les concierne lo que sucede fuera de la máquina. - El análisis estructurado adopta un punto de vista diferente, el diagrama de flujo de datos sigue las trayectorias de los datos por donde ellos pasen, ya sean procesos manuales o automáticos.

El uso de la "redundancia limitada". Uno de los aspectos mas importantes de esta profesión y que representa un cuello de botella es el congelamiento de las especificaciones. No aceptar cambios en las especificaciones porque estas no pueden ser actualizadas o aceptarlas, pero sin actualizar las especificaciones resulta un grave peligro para el desarrollo del sistema.

El concepto llave que el análisis estructurado propone para especificar los requerimientos es tener poca o nada de redundancia para obtener especificaciones considerablemente mas consistentes.

¿Qué es una especificación estructurada?

El análisis estructurado se involucra en la construcción de una nueva clase de especificación, una especificación estructurada hecha a base de diagramas de flujo de datos, diccionario de datos y miniespecificaciones.

1.- Diagrama de flujo de datos (DFD) sirve para particionar el sistema. El sistema que es tratado con un diagrama de flujo de datos puede incluir partes manuales y automatizadas. El propósito del DFD es declarar los componentes de los procesos que integran el sistema y las interfases entre los componentes.

2.- Diccionario de datos.- Define las interfases que -- fueron declarados en el DFD. Esto se hace con una notación convencional que permita representar los -- flujos de datos y guardarlos en términos de sus componentes.

3.- Las miniespecificaciones definen un proceso elemental declarado en el DFD. Un proceso es considerado elemental o primitivo cuando no puede ya ser subidividido en niveles mas bajos. Antes de que la especificación estructurada este completa deberán estar todos los procesos primitivos acompañados de una miniespecificación. El método usado para escribir miniespecificaciones utiliza: Ingles estructurado, tablas - de decisión, árboles de decisión, etc.

Resumen

Los fundamentos del análisis estructurado no son nuevos. Muchas de las ideas han sido ya usadas por años. ¿Que_ es lo nuevo de esta disciplina emergente? Podemos decir que intenta sintematizar el proceso de análisis para especificar requerimientos.

CONSTRUCCION DEL DIAGRAMA DE FLUJO DE DATOS

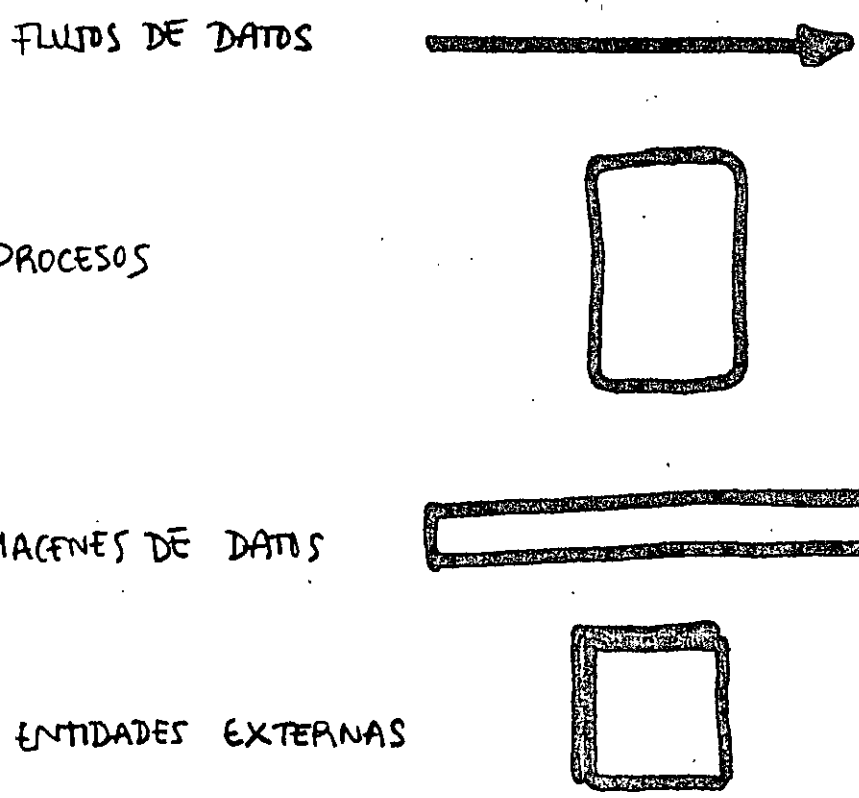
Definición

Un diagrama de flujo de datos (DFD) es una representación en forma de red de un sistema. El DFD es una herramienta para modelar un sistema que puede ser automatizado, manual o mixto. El DFD presenta al sistema en términos de sus piezas componentes -- con todas las interfases entre los componentes indicados.

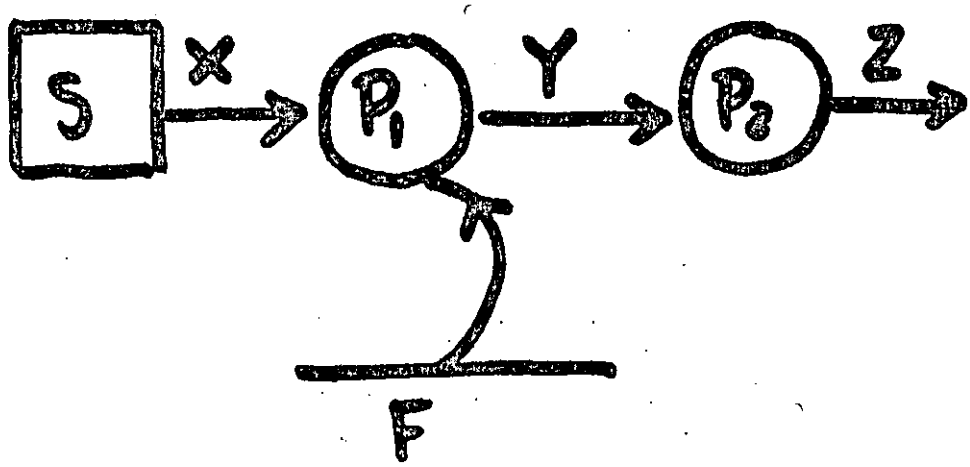
Elementos del DFD

Los diagramas de flujos de datos se construyen con cuatro elementos básicos:

1.- Flujo de datos	-	representados por vectores
2.- Procesos	-	representados por círculos o rectángulos redondeados.
3.- Archivos	-	representados por líneas rectas o rectángulos abiertos.
4.- Datos fuentes o destino	-	representados por cajas o cuadros.



Ejemplo

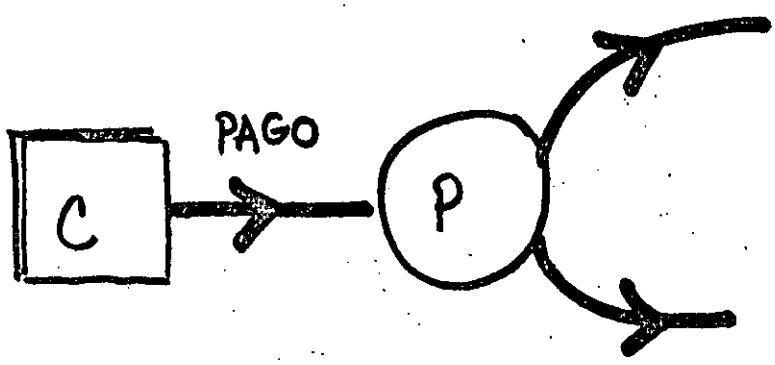


El diagrama de flujo de datos se leería:

X llega de la fuente S y es transformada a Y por el proceso P_1 el cual requiere acceso al archivo F para hacer su trabajo. - Y es posteriormente transformada a Z por el proceso P_2 .

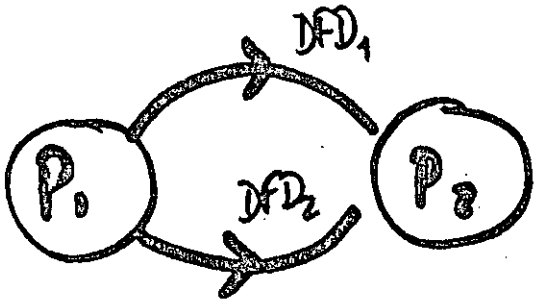
1.- Flujo de datos

El flujo de datos ⟶ representa a una interfase entre los -- componentes del diagrama de flujo de datos. Estos, se encuentran entre procesos y van o vienen de archivos o de fuentes y destinos.



El flujo de datos → es una línea a través de la cual, fluye un paquete de información conocida. El pago (flujo de datos del ejemplo anterior) puede consistir del cheque y de la copia de la factura y no por esto, se colocan 2 líneas entre la fuente y el proceso.

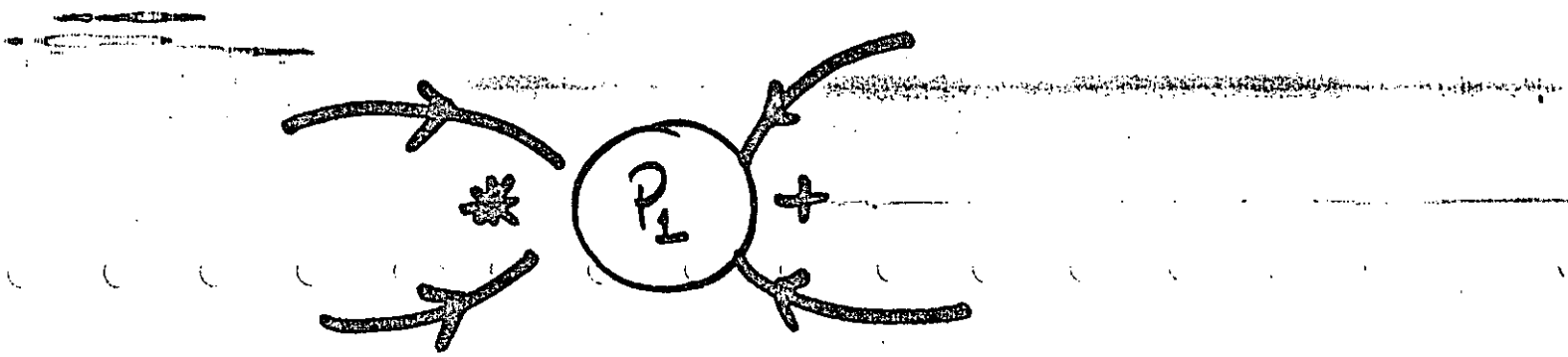
En algunos casos si es posible encontrar que de un proceso a otro existan 2 líneas, pero estas llevan información que juntas, no constituirán un paquete. (El tiempo podría ser una causa)



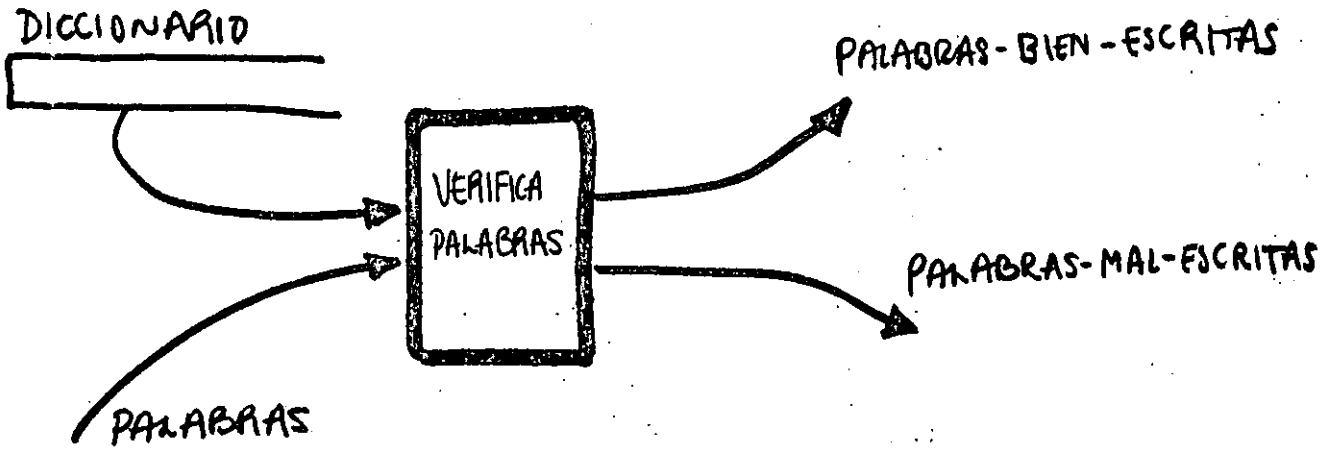
Convenciones para asignar un nombre al flujo de datos:

- Los nombres estan ligados con guiones y se usan letras mayúscula. NUMERO-DE-CUENTA
- No hay dos flujos de datos con el mismo nombre
- Los nombres no solamente deben representar los datos que se muevan en el flujo de datos, si sabemos más de ellos pueden aparecer en el nombre.
- La misma línea de flujo de datos puede llegar a 2 procesos o dos salidas pueden juntarse en una sola línea.
- Las líneas que salen o llegan a archivos no requieren nombre.

En algunos casos, es necesario indicar que dos flujos de datos, deben presentarse al proceso o que uno o el otro pero no ambos.



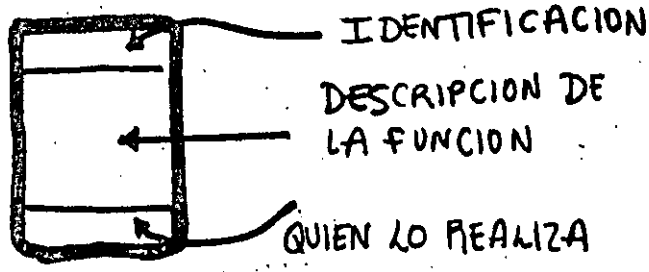
El proceso invariablemente muestra alguna cantidad de trabajo - realizado sobre los datos.



Aquí se muestra un proceso cuya tarea es dividir el flujo de entrada de datos en 2 flujos de datos de salida (Palabras bien -- escritas - Palabras mal escritas).

Un proceso es una tranformación del flujo de datos de entrada a un flujo de datos de salida. Cada proceso, debe tener un nombre.

Convenciones para referenciar el proceso:



Identificación: es un número cuya función es identificar al -- proceso

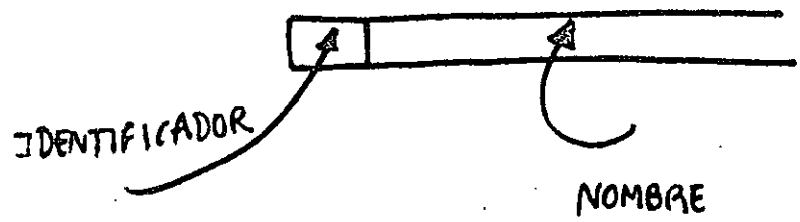
Descripción de la función: es una oración corta que describe al proceso.

Quien lo realiza: Es el nombre de un departamento, persona, programa, etc., que realiza la función.

3.- El archivo

Para los propósitos del DFD el archivo es considerado como un depósito temporal de datos.

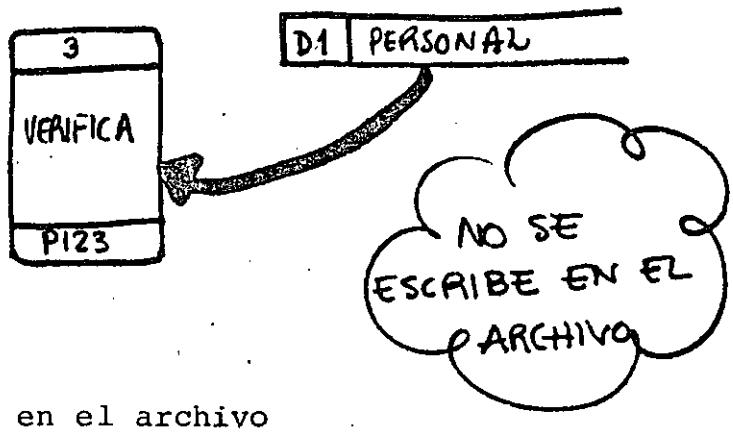
Convenciones:



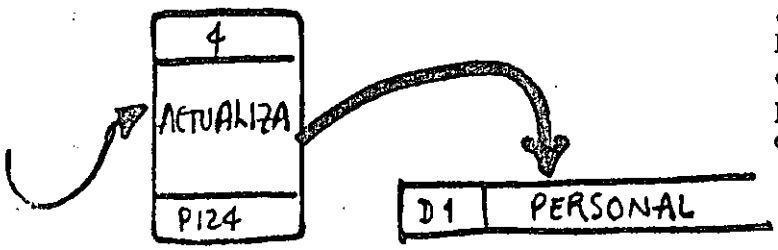
Identificador: Para facilitar su referencia y se utiliza la letra D seguida de un número.

Nombre: Un nombre lo mas descriptivo que se pueda.

El flujo de las flechas que salen o llegan de o hacia los procesos es significante por ejemplo:



Este proceso escribe en el archivo



¿Cómo actualiza sin leer? Es cierto, pero debe en el diagrama solo indicarse el propósito fundamental que en este caso es hacia afuera.

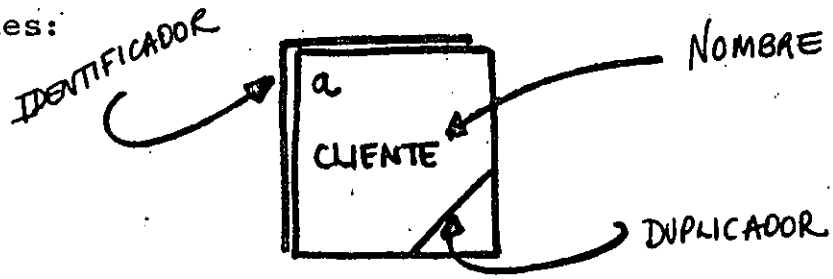
este proceso solo escribe el archivo.

(Si es necesario leer y escribir la flecha puede ser ↕)

4.- La fuente o el destino

La fuente o el destino es una persona o una organización situado fuera del contexto del sistema, es el orígen o el receptor de los datos del sistema.

Convenciones:



Identificador: letras minúsculas asociadas a las entidades externas

Nombre: nombre de la entidad

Duplicador: indica que el símbolo se repite en el diagrama.

Diagramas de flujo de datos jerárquicos.

Este concepto se refiere a explotar un diagrama en la modalidad de TOP-DOWN. Estableciendo una relación padre-hijo por cada proceso de explosión. Por ejemplo:

Diagrama 0

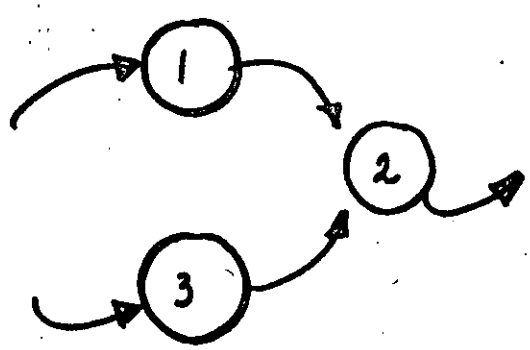


Diagrama 2

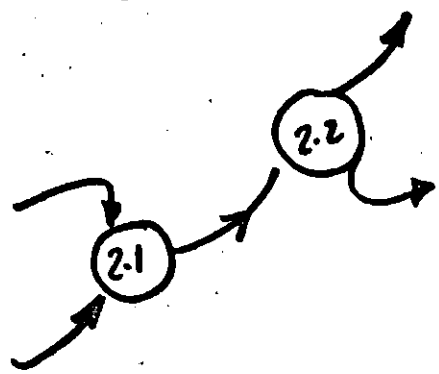


Diagrama 1

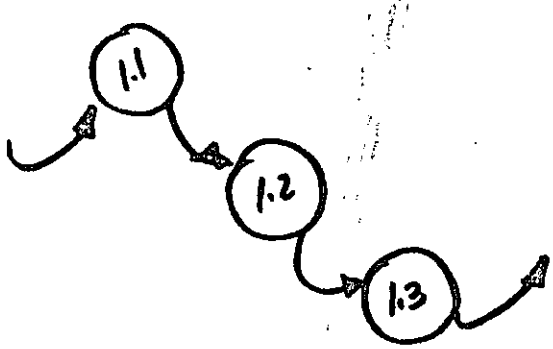
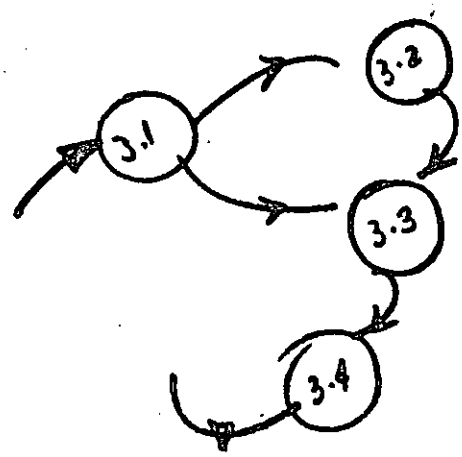


Diagrama 3



Diagramas de flujos de datos balanceados

Es un concepto en el que al hacer alguna explosión la entrada y la salida en el nivel inferior se mantienen

Por ejemplo

El diagrama 2 es desbalanceado y los diagramas 1 y 3 son balanceados.

Guía para la elaboración del diagrama de flujo de datos.

1) Identificar las entidades externas involucradas. Hay que recordar que los flujos de datos, son creados cuando un evento se desarrolla fuera de los límites del sistema:

   Una persona decide comprar alguna cosa
   Un camión llega a la terminal.

2) Identificar las entradas y salidas. Si la persona decide -- comprar algo las entradas pueden ser: Una orden de compra - por teléfono, una carta o pedido por correo, presentarse personalmente a comprar algo, etc. Se sugiere hacer una lista_ de las entradas y las salidas.

3) Identificar las preguntas y los requerimientos de información que puedan presentarse. Hay que especificar un flujo de datos que define la información dada al sistema y otro para indicar que es lo requerido.

4) Hacer un diagrama colocando las entidades externas y el flujo de datos que se origina de cada uno de ellos, los procesos y los almacenes necesarios.

   En este punto, no hay que poner mucho énfasis en las consideraciones de tiempo, excepto en la procedencia natural de los_ eventos. Hay que dibujar un diagrama que nunca inicia y nun ca termina. No hay que poner decisiones.

5) El primer diagrama puede ser hecho a mano libre

6) Es posible que se requiera al menos 3 variaciones del dibujo_ inicial. Ya que el primero a lo mejor no es claro.

7) Cuando se tiene el primer diagrama, verificar que todas las - entradas y salidas se han incluido. Excepto aquello que resultan salidas de error o excepciones.

8) Ahora producir un nuevo diagrama:
   - Tratando de minimizar el número de cruces de líneas de flujo.
   - Duplicar entidades externas si es necesario
   - Duplicar archivos si es necesario

9) Validar el diagrama con alguno de los usuarios, mostrarle el diagrama y pasearlo por el y explicarle que es solo una aproximación y si tiene alguna observación que la haga.

10) Hacer una explosión de cada uno de los procesos incorporando errores y salidas de excepción.
Es posible que esto pueda hacer que se modifique el diagrama del nivel anterior.

11) Si es posible, construir el diagrama final en una hoja de -- 36 x 48 pulgadas. Que servirá como ayuda invaluable en la presentación a los usuarios, diseñadores, revisores y a todos los que tengan que ver con el sistema.

Ejemplo

Consideremos el caso de automatizar el proceso de enseñanza-aprendizaje tomando como punto de partida el modelo de Anderson y Faust.

Este modelo sugiere la aplicación de las siguientes etapas:

1) Especificación de objetivos
   Los objetivos son entregados al profesor por el comité de carrera como la entrada principal al sistema. El profesor los revisa y establece en definitiva cuales serán los objetivos que guiarán su proceso de E.A.

2) Elaboración de instrumentos de Medición
   Tomando a sus objetivos el profesor elabora los instrumentos de medición (cuestionarios, prácticas, elaboración de programas, exámenes, etc.) y establece las siguientes clases:

- Evaluación diagnóstica
- Evaluación formativa
- Evaluación Sumaria

La evaluación diagnóstica es la que le permite saber el estado inicial del alumno, la formativa es complementaria a instrucción y la sumaria le permite establecer en que grado se lograron los objetivos.

3) Aplicación de la evaluación diagnóstica.

   Antes de iniciar la instrucción el maestro aplica la evaluación diagnóstica al alumno para conocer el estado inicial, de este resultado el maestro puede optar por lo siguiente:

- No se le instruye porque ya conoce el material
- No se le instruye porque no tiene los antecedentes
- Se le instruye

4) Planeación de la instrucción

   El maestro de acuerdo a sus objetivos selecciona y/o elabora el método de instrucción y las técnicas de enseñanza, la selección de las actividades de aprendizaje y la selección y/o elaboración de los materiales y medios educativos.

5) Aplicación de la instrucción

En este punto, el maestro realiza el acto de enseñar a sus -- alumnos.

6) Aplicación de la evaluación sumaria

Cuando se ha concluído con la instrucción el maestro aplica - una evaluación sumaria, la cual le permitirá establecer el -- grado con el que se lograron los objetivos. Ahora bien si - se lograron los objetivos podemos dar por terminados el proce so. Si no se lograron los objetivos hay que investigar las causas en las etapas del modelo.

7) Administración del proceso de E.A.

La administración del proceso de E.A incluye una fase de inscripción, de seguimiento y de emisión de una calificación para el alumno. Otra, en la que se inscribe, se dan anteceden tes y consecuentes, temarios, guías de estudios de los cursos.

PROCESO DE CONSTRUCCION DEL DFD.

1) Identificar las entidades externas involucradas.

COMITE DE CARRERA
ALUMNO

2) Identificar entradas y salidas

ENTRADAS: COMITE DE CARRERA
- OBJETIVOS DE APRENDIZAJE
- DATOS DEL CURSO

: ALUMNO

- DATOS DEL ALUMNO
- CONTESTACION DIAGNOSTICA
- CONTESTACION FORMATIVA
- CONTESTACION SUMARIA

SALIDAS: COMITE DE CARRERA

- RESULTADOS SUMARIOS

: ALUMNO
- RESULTADOS SUMARIOS

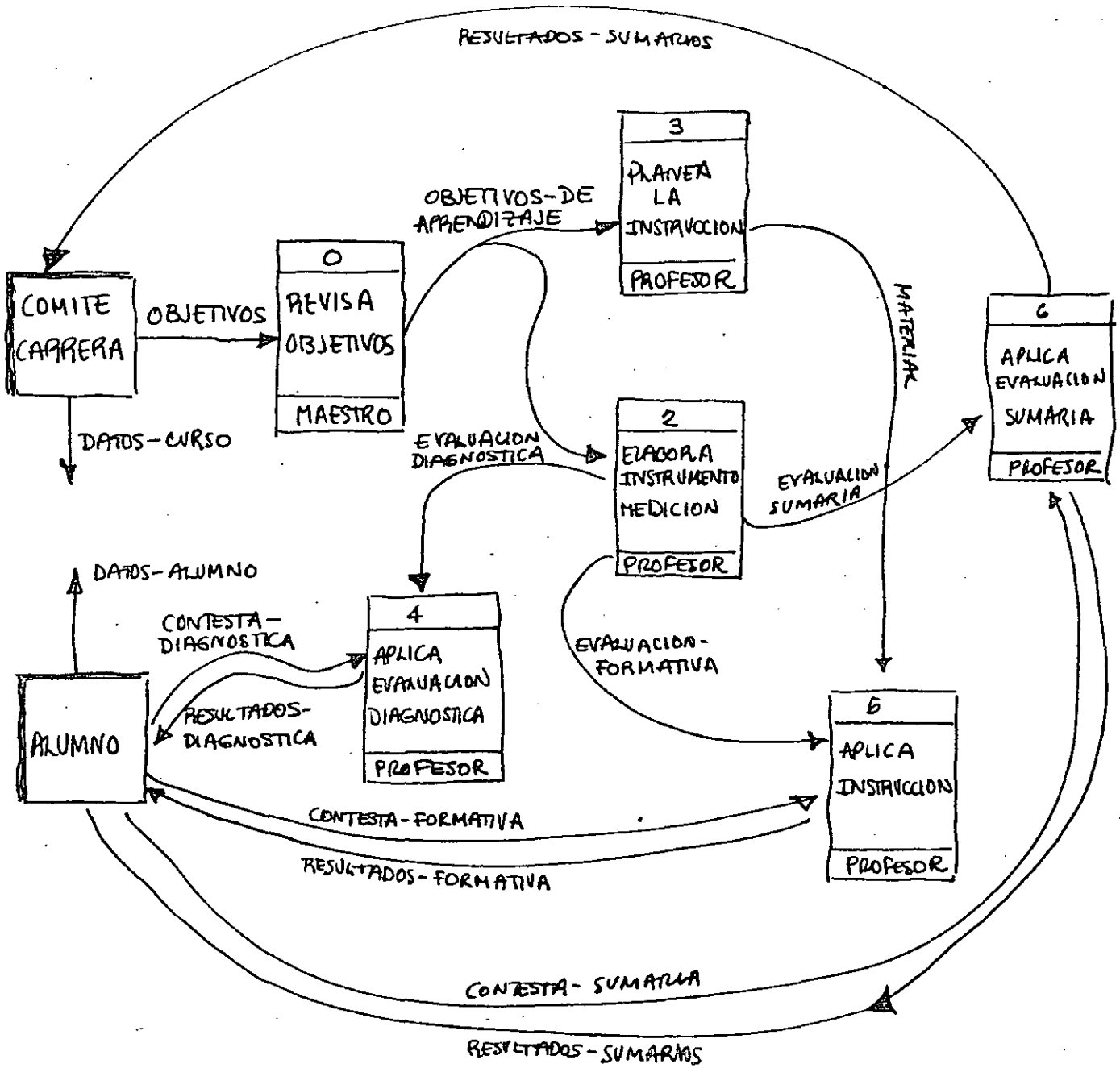
3) Preguntas al sistema

COMITE DE CARRERA:

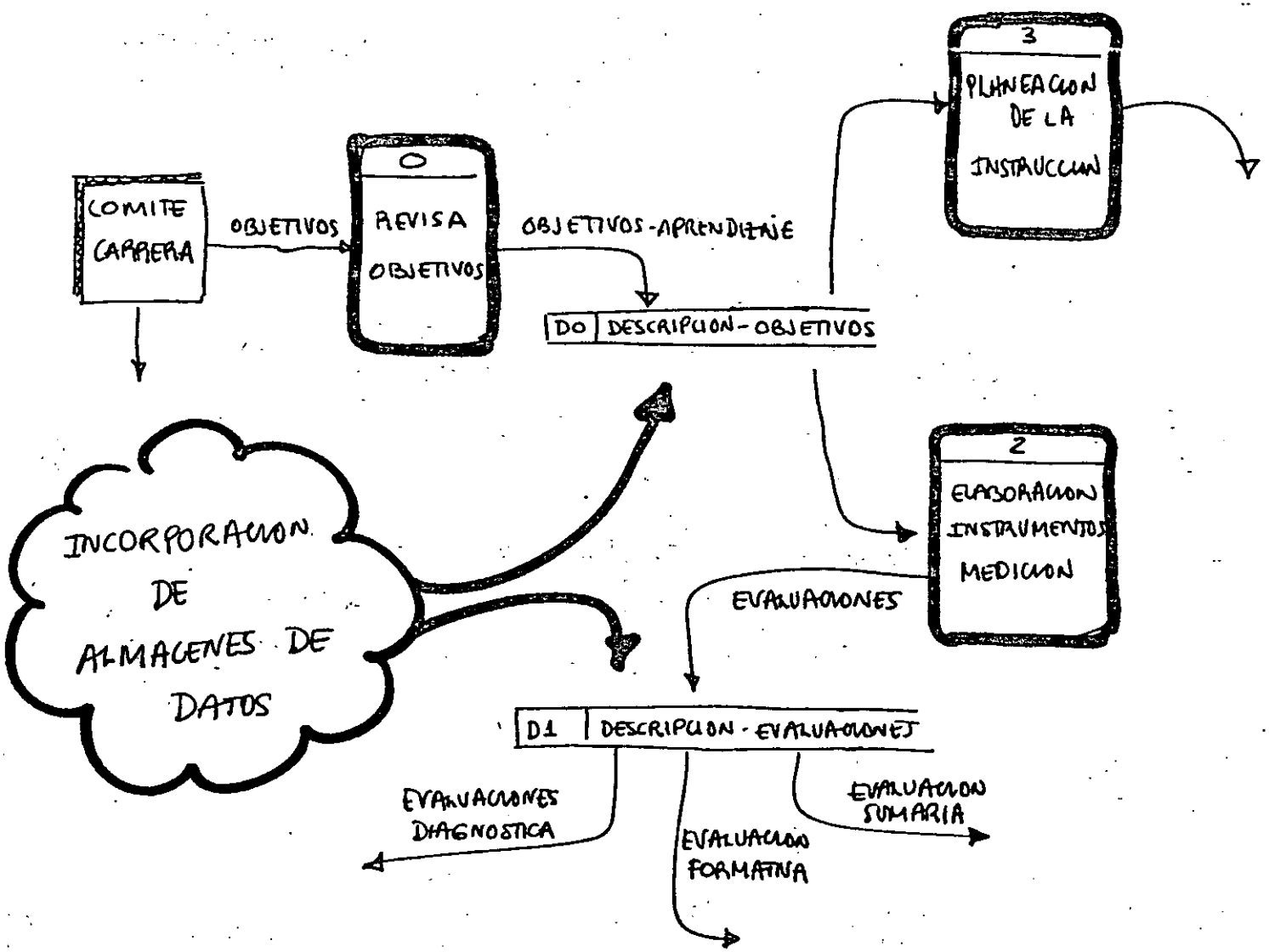
- ¿QUE CURSOS ESTAS IMPARTIENDO?
- ¿QUE ALUMNOS TIENES INSCRITOS EN CADA CURSO?
- ¿CUAL ES EL PROMEDIO DE CALIFICACIONES?
- ¿CUANTOS ALUMNOS ABANDONARON EN CADA CURSO?
- ¿QUE CAMBIOS SUFRIERON LOS OBJETIVOS?

ALUMNO:

- ¿CUAL ES MI PROMEDIO?
- ¿CUAL ES MI CALIFICACION FINAL?
- ¿DONDE PUEDO ENCONTRAR MAS INFORMACION?
- ¿QUIEN ME PUEDE ASESORAR?
- ¿CUALES SON LOS OBJETIVOS DEL CURSO?
- ¿CUALES SON LOS OBJETIVOS DE ESTA UNIDAD?
- ¿CUALES SON LOS ANTECEDENTES DEL CURSO?


RESULTADOS - SUMARIOS
COMITE CARRERA
OBJETIVOS
0
REVISA OBJETIVOS
MAESTRO
DATOS - CURSO
OBJETIVOS-DE APRENDIZAJE
3
PLANEA LA INSTRUCCION
PROFESOR
MATERIAL
6
APLICA EVALUACION SUMARIA
PROFESOR
EVALUACION DIAGNOSTICA
2
ELABORA INSTRUMENTO MEDICION
PROFESOR
EVALUACION SUMARIA
DATOS - ALUMNO
CONTESTA - DIAGNOSTICA
4
APLICA EVALUACION DIAGNOSTICA
PROFESOR
RESULTADOS - DIAGNOSTICA
EVALUACION - FORMATIVA
ALUMNO
5
APLICA INSTRUCCION
PROFESOR
CONTESTA - FORMATIVA
RESULTADOS - FORMATIVA
CONTESTA - SUMATIVA
RESULTADOS - SUMARIOS

4, 5, 6, 7)


COMITE CARRERA
OBJETIVOS
0
REVISA OBJETIVOS
OBJETIVOS-APRENDIZAJE
3
PLANEACION DE LA INSTRUCCION
D0 DESCRIPCION-OBJETIVOS
2
ELABORACION INSTRUMENTOS MEDICION
EVALUACIONES
INCORPORACION DE ALMACENES DE DATOS
D1 DESCRIPCION-EVALUACIONES
EVALUACIONES DIAGNOSTICA
EVALUACION FORMATIVA
EVALUACION SUMARIA

CONSTRUCCION DEL DICCIONARIO DE DATOS.

En los diagramas de flujos de datos, les hemos dado nombre a los flujos de datos, a los almacenes de datos y a los procesos. Pero que significan exactamente los nombres que se les han asignado, la respuesta se encuentra en el diccionario de datos (DD).

El diccionario de datos es la herramienta con la cual, documentaremos cada uno de los elementos del DFD.

Para documentar los flujos de datos y los archivos es necesario establecer que son: un dato elemental y una estructura de datos.

Dato elemental.-

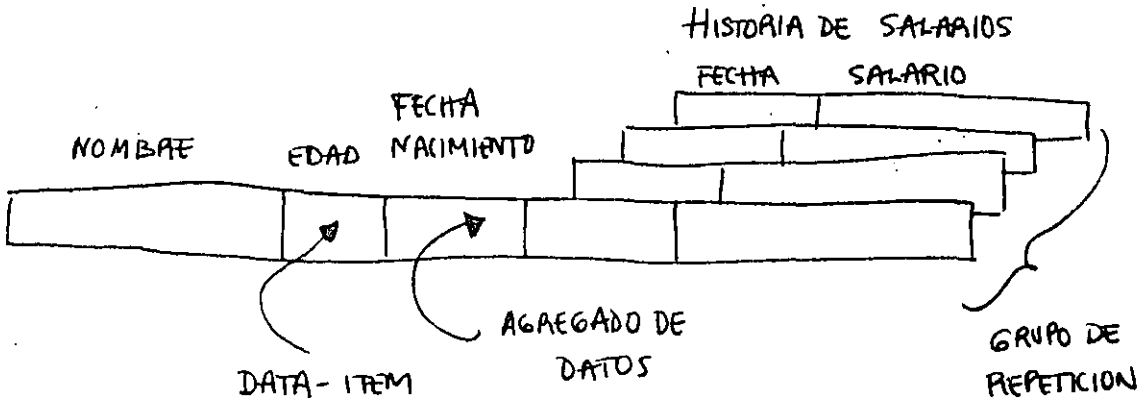
Dato elemental, es sinónimo de data-item, campo y elemento. Cualquier dato elemental, toma valores atómico o que no es posible su descomposición.
El patron de unos y ceros que lo representa debe ser considerado como un todo.

Estructura de datos.-

Es un agregado de estructuras elementos o de otro tipo de datos (agregado de datos elementales, grupos de repetición, etc.) o de una mezcla de ambos.

Ejemplo:

ENTIDAD: PERSONA ATRIBUTOS: NOMBRE
EDAD
FECHA DE NACIMIENTO
HISTORIA DE SALARIOS
FECHA
SALARIO



Convenciones para especificar las estructuras de datos:

=	es equivalente a
+	and
[]	or
{}	interación de los componentes
( )	opcional

para mostrar como serían utilizados definiremos la estructura de datos PERSONAL

```
PERSONAL  =  NOMBRE+
             EDAD+
             FECHA DE NACIMIENTO+
              HISTORIA DE SALARIOS

FECHA DE NACIMIENTO = DIA+
                      MES+
                      AÑO

HISTORIA DE SALARIOS = FECHA+
                       SALARIO

NOMBRE = DATO-ELEMENTAL
EDAD   = DATO-ELEMENTAL
DIA    = DATO-ELEMENTAL
MES    = DATO-ELEMENTAL
AÑO    = DATO-ELEMENTAL
FECHA  = DATO-ELEMENTAL
SALARIO= DATO-ELEMENTAL
```

Para documentar cada uno de los elementos del DAD se sugiere - elaborar formas para describirlas. Las siguientes son un ejemplo en las que se muestra los atributos que deben definirse en cada caso:

La forma F1 es utilizada para especificar las estructuras de - datos elementales dando para ella una descripción el tipo, sinónimo, valores discretos o continuos etc., la forma F2 es para - especificar estructuras de datos considerando una descripción, los componentes o atributos, los flujos y almacenes en donde se utiliza y los volumenes de datos que involucra; F3 es para especificar los flujos de datos especificando de donde y hacia donde fluyen los datos, una descripción del contenido del flujo, - las estructuras que fluyen y los volúmenes de datos; la F4 es - para describir los almacenes de datos dando una descripción de los flujos que llegan y salen y el contenido del almacen en término de las estructuras de datos y la forma F5 para especificar procesos dando en este caso una descripción, las entradas y salidas y un resumen del proceso.

F1

Nombre del dato elemental

descripción

tipo A AN N

sinónimos

valores discretos

valor	significado

valores continuos

rango

valores típicos

longitud

estructuras de datos donde aparece

F2

Nombre de la estructura

descripción

Componentes

flujos y almacenes donde aparece

Volumen de información

F3

flujo de datos

fuente ref. _____ descripción _____

destino ref. _____ descripción _____

descripción _____

estructuras de datos que fluyen

volumen de información

F4

Archivo Ref

descripción _____

flujos de datos que llegan

flujos de datos que salen

Contenido _____

F5

proceso Ref: ____

descripción ____

entradas	resumen lógico	salidas

EJEMPLOS:

SALARIO — Nombre del dato elemental

descripcion PERSEPCION DIARIA POR UNA JORNADA DE 8 HORAS — tipo A AN X

sinonimos

valores discretos

valor	significado

valores continuos

rango

valores típicos LOS DEL SALARIO MINIMO

longitud 7 DIGITOS

estructuras de datos donde aparece PERSONAL

PERSONAL — Nombre de la estructura

descripción DATOS SOBRE UN EMPLEADO QUE LABORA EN LA EMPRESA

Componentes

NOMBRE
EDAD
FECHA DE NACIMIENTO
HISTORIA DE SALARIOS

flujos y almacenes donde aparece

Volumen de información

400 EMPLEADOS

OBJETIVOS — flujo de datos

fuente ref. a — descripción COMITE DE CARRERA

destino ref. o — descripción REVISION DE OBJETIVOS

descripción DESCRIPCION DE LOS OBJETIVOS QUE DEBEN LOGRARSE AL FINALIZAR EL CURSO

estructuras de datos que fluyen

DESCRIP-OBJETIVO

volumen de información

40 OBJETIVOS/CURSO

DESCRIPCION-EVALUACIONES — Archivo Ref D1

descripción CONTIENE LOS TEXTOS Y LAS IDENTIFICACIONES DE CADA UNO DE LOS REACTIVOS QUE COMPONEN LAS EVALUAC.

flujos de datos que llegan

EVALUACIONES

flujos de datos que salen

EVALUACIONES-DIAGNOSTICAS

EVALUACIONES-FORMATIVAS

EVALUACIONES-SUMARIAS

Contenido

IDENTIFICADOR

TEXTO-REACTIVO

PLANEACION DE LA INSTRUCCION — proceso Ref: 3

descripción: ANALISIS DE CADA OBJETIVO CON EL PROPOSITO DE ESTABLECER LOS RECURSOS, ESTRATEGIAS, TEXTOS QUE NOS LLEVEN AL LOGRO DEL MISMO

entradas	resumen lógico	salidas
OBJETIVOS-DE APRENDIZAJE	PARA CADA OBJETIVO ESTABLECER: ANALISIS DE CONCEPTOS SECUENCIA JERARQUICA RECURSOS ESTRATEGIAS DEFINICIONES	MATERIAL

CONSTRUCCION DE MINIESPECIFICACIONES

Miniespecificaciones

Una miniespecificación es un documento en el que se da la descripción de la política que gobierna la transformación de una o mas entradas a un proceso a sus correspondientes salidas.

Las herramientas que presenta el análisis estructurado como una alternativa al texto clasico son las siguientes

1. Español estructurado
2. Tablas de decisión
3. Arboles de decisión

Español estructurado

La figura principal para describir los procesos es:

Si Condición 1
  entonces Acción-A
o bien (no se cumple condición -1)
  hacer Acción-B

Ejemplo: Sumar A con B siempre que A sea mayor que B si A es menor que B restar A de B.

Si A es menor que B
  entonces restar A de B
o bien (A no es menor que B)
  hacer suma de A con B

Existen una gran cantidad de políticas en donde es necesario especificar rangos.

Ejemplo: hasta 20 artículos no existe descuento mas de 20 tienen el 5%

Si el número de artículos es mayor que 20
  entonces dar el 5% de descuento
o bien (el número es menor o igual que 20)
  hacer no dar descuento

En la especificación de rangos los comparadores mas utilizados son:

Mayor que
Mayor o igual que
Menor que
Menor o igual que

En algunos casos es necesario establecer mas de una condición, los cuales deben satisfacerse todos o algunos de ellos para -- tomar una acción.

Ejemplo:

Clientes que nos han comprado mas de 1 000 000.00 en el año y tienen una buena historia de pagos tienen prioridad en el servicio y también clientes con mas de 20 años de - antiguedad.

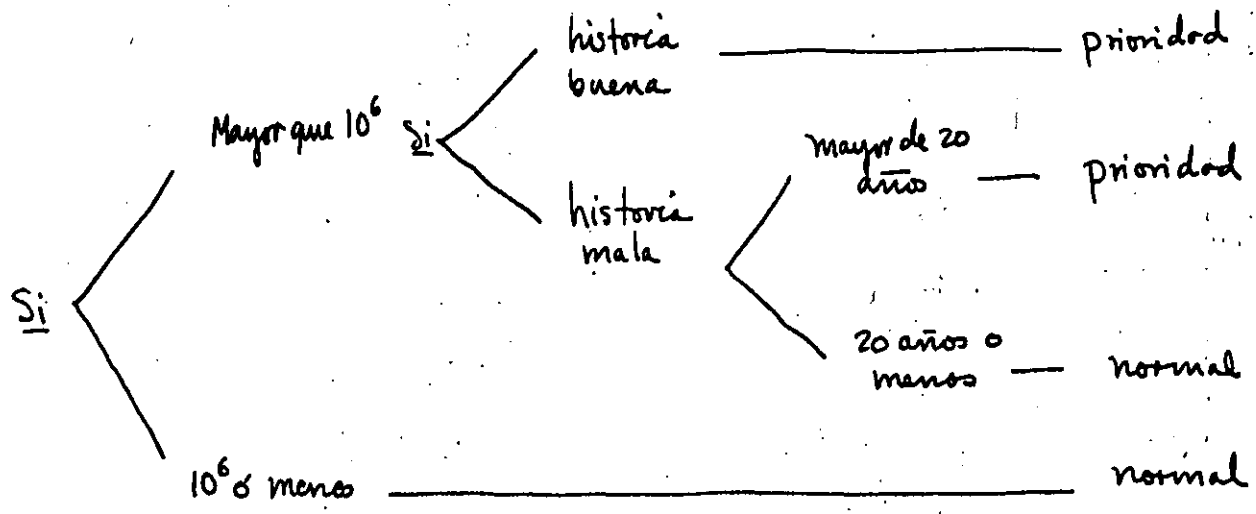
("mas de 100 000" y "buena historia de pago") o "mas de 20 años"

"mas de 100 000" y ("buena historia de pagos" o "mas de 20 años")

Arboles de decisión.

En algunos casos, resulta ser mas clara la especificación, si - construimos un arbol de decisión. Por ejemplo:

En el ejemplo anterior cualquier otra combinación implica un - tratamiento normal al cliente, todas estas posibilidades que-- dan en un arbol de decisión de la siguiente manera:



Español estructurado

```
Si    que 1000000
  entonces si buena historia
          entonces tiene prioridad
  o bien (mala historia)
          hacer  si mas de 20 años
                    entonces tiene prioridad
                 o bien (20 a menos)
                    hacer tramiento normal
o bien (  1000000)
  hacer tratamiento normal
```

Tablas de decisión

Una tabla de decisión se forma, haciendo una lista de las con-- diciones y al final una lista de las acciones.

Ejemplo:

para el ejemplo anterior tenemos

C.1	mas de 1000000	si	si	si	si	no	no	no	no
C.2	buenos pagos	si	si	no	no	sí	si	no	no
C.3	mas de 20 años	si	no	si	no	si	no	si	no
a.1	prioridad	X	X	X					
a.2	normal				X	X	X	X	X

En algunos casos, la acción está determinada por el valor de una o de dos condiciones sin importar cual es el valor de las otras. Esto hace posible reducir la tabla de decisión.

para reducir es necesario observar lo siguiente:

1) Encontrar un par de reglas para los cuales
   - la acción es la misma
   - los valores de las condiciones son los mismos excepto -- una y solo una condición que es diferente.

2) Reemplazar el por por una sola columna sustituyendo la condición diferente por el símbolo (-) que significa "no importa"

3) Repetir este criterio los veces que sea necesario

Resultado del ejemplo anterior

	1/2	3	4	5/6/7/8
C.1	SI	SI	SI	NO
C.2	SI	NO	NO	-
C.3	-	SI	NO	-
A.1	X	X		
A.2			X	X

Recomendaciones:

1) Hay que usar un arbol de decisión cuando el número de acciones es reducido y no cualquier combinación de las condiciones es posible.

2) Hay que usar una tabla cuando el número de acciones es grande y muchas combinaciones de las condiciones es posible.

Bibliografía

Awad M. Elias. Systems Analysis and Design.
Richard D. Irwing Inc. 1979

Burch, Strater, Gridnitski. Information Systems:
Theory and practice.
John Wiley and Sons 1979.

De Marco. Structured Analysis and System Specification
Prentice Hall. 1979.

Gane and Sorson. Structured Systems Analysis:
tool and techniques. Prentice Hall. 1979

Yourdon N. Edward. Classics in Software Engineering.
Yourdon Press. 1979

Yourdon N. Edward. Structured Analysis/Design Workshop.
Edition 7.2 1981.

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

METODOLOGIA PARA EL DESARRROLLO DE SISTEMAS DE INFORMACION

ARTICULOS CORRESPONDIENTE AL TEMA I Y II

EL ENFOQUE ESTRUCTURADO DE DESARROLLO DE SOFTWARE Y ANALISIS ESTRUCTURADO

ING. JORGE I. EUAN AVILA

MAYO, 1985

Software Engineering

SOFTWARE ENGINEERING	BOEHM
STRUCTURED ANALYSIS FOR REQUIREMENTS DEFINITION	ROSS & SCHOMAN
PSL/PSA: A COMPUTER-AIDED TECHNIQUE FOR STRUCTURED DOCUMENTATION AND ANALYSIS OF INFORMATION PROCESSING SYSTEMS	TEICHROEW & HERSHEY
STRUCTURED ANALYSIS AND SYSTEM SPECIFICATION	DEMARCO

I. Introduction

The annual cost of software in the U.S. is approximately 20 billion dollars. Its rate of growth is considerably greater than that of the economy in general. Compared to the cost of computer hardware, the cost of software is continuing to escalate along the lines predicted in Fig. 1 [1].* A recent SHARE study [2] indicates further that software demand over the years 1975-1985 will grow considerably faster (about 21-23 percent per year) than the growth rate in software supply at current estimated growth rates of the software labor force and individual productivity per individual, which produce a combined growth rate of about 11.5-17 percent per year over the years 1975-1985.

In addition, as we continue to automate many of the processes which control our life-style — our medical equipment, air traffic control, defense system, personal records, bank accounts — we continue to trust more and more in the reliable functioning of this proliferating mass of software. *Software engineering* is the means by which we attempt to produce all of this software in a way that is both cost-effective and reliable enough to deserve our trust. Clearly, it is a discipline which is important to establish well and to perform well.

*Another trend has been added to Fig. 1 — the growth of software maintenance — which will be discussed later.

This paper will begin with a definition of "software engineering." It will then survey the current state of the art of the discipline, and conclude with an assessment of likely future trends.

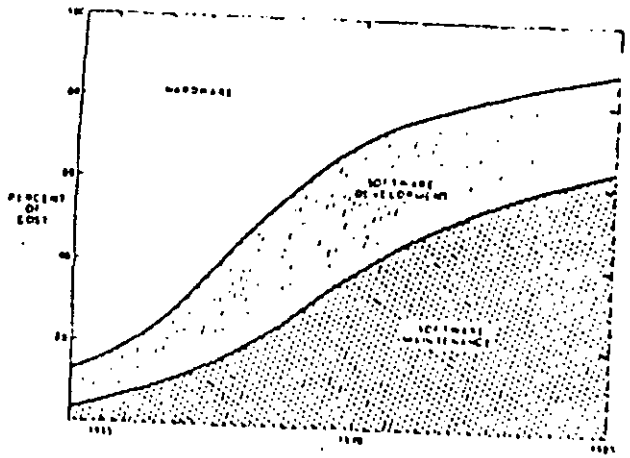


Figure 1. Hardware-software cost trends.

II. Definitions

Let us begin by defining "software engineering." We will define software to include not only computer programs, but also the associated documentation required to develop, operate, and maintain the programs. By defining software in this broader sense, we wish to emphasize the necessity of considering the generation of timely documentation as an integral portion of the software development process. We can then combine this with a definition of "engineering" to produce the following definition.

> *Software Engineering:* The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

Three main points should be made about this definition. The first concerns the necessity of considering a broad enough interpretation of the word "design" to cover the extremely important activity of software requirements engineering. The second point is that the definition should cover the entire software life cycle, thus including those activities of redesign and modification often termed "software maintenance." (Fig. 2 indicates the overall set of activities thus encompassed in the definition.) The final point is that our store of knowledge about software which can really be called "scientific knowledge" is a rather small base upon which to build an engineering discipline. But, of course, that is what makes software engineering such a fascinating challenge at this time.

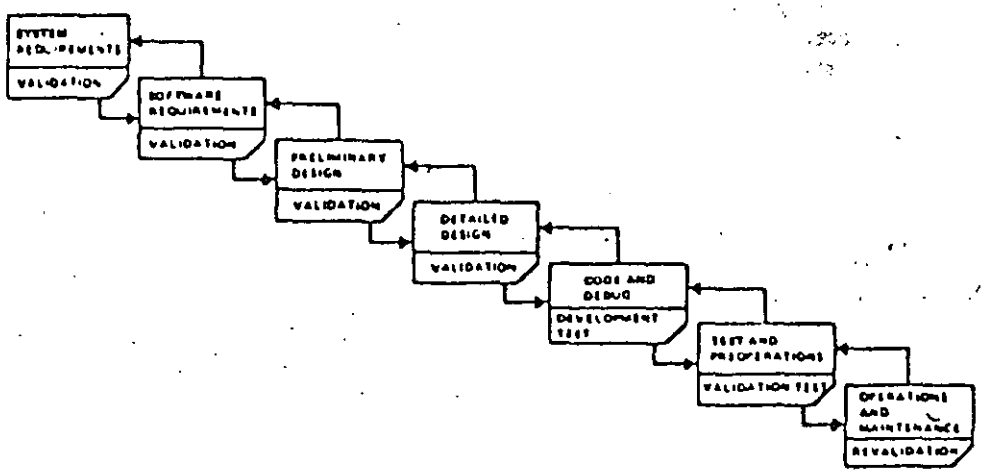


Figure 2. Software life cycle.

The remainder of this paper will discuss the state of the art of software engineering along the lines of the software life cycle depicted in Fig. 2. Section III contains a discussion of software requirements engineering, with some mention of the problem of determining overall system requirements. Section IV discusses both preliminary design and detailed design technology trends. Section V contains only a brief discussion of programming, as this topic is also covered in a companion article in this issue [3]. Section VI covers both software testing and the overall life cycle concern with software reliability. Section VII discusses the highly important but largely neglected area of software maintenance. Section VIII surveys software management concepts and techniques, and discusses the status and trends of integrated technology-management approaches to software development. Finally, Section IX concludes with an assessment of the current state of the art of software engineering with respect to the definition above.

Each section (sometimes after an introduction) contains a short summary of current practice in the area, followed by a survey of current frontier technology, and concluding with a short summary of likely trends in the area. The survey is oriented primarily toward discussing the domain of applicability of techniques (where and when they work) rather than how they work in detail. An extensive set of references is provided for readers wishing to pursue the

III. Software requirements engineering

A. Critical nature of software requirements engineering

Software requirements engineering is the discipline for developing a complete, consistent, unambiguous specification — which can serve as a basis for common agreement among all parties concerned — describing *what* the software product will do (but *not how* it will do it; this is to be done in the design specification).

The extreme importance of such a specification is only now becoming generally recognized. Its importance derives from two main characteristics: 1) it is easy to delay or avoid doing thoroughly; and 2) deficiencies in it are very difficult and expensive to correct later.

Fig. 3 shows a summary of current experience at IBM [4], GTE [5], and TRW on the relative cost of correcting software errors as a function of the phase in which they are corrected. Clearly, it pays off to invest effort in finding requirements errors early and correcting them in, say, 1 man-hour rather than waiting to find the error during operations and having to spend 100 man-hours correcting it.

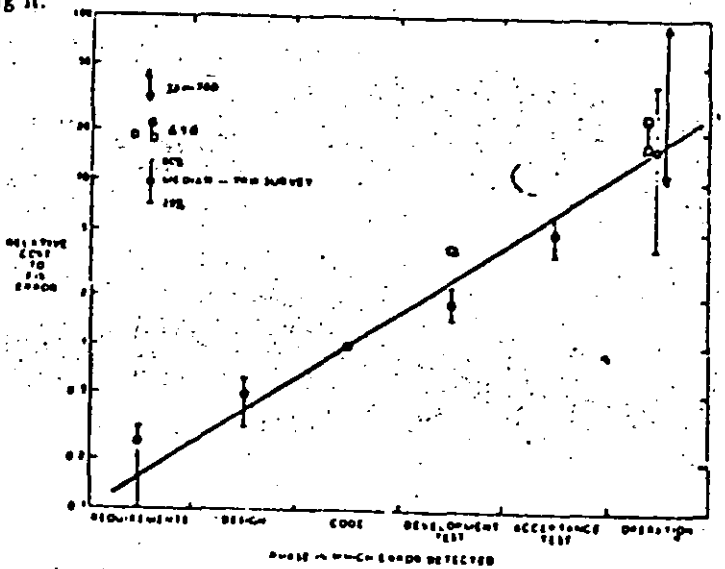


Figure 3. Software validation: the price of procrastination.

Besides the cost-to-fix problems, there are other critical problems stemming from a lack of a good requirements specification. These include [6]: 1) top-down designing is impossible, for lack of a well-specified "top"; 2) testing is impossible, because there is nothing to test against; 3) the user is frozen out, because there is no clear statement of what is being produced for him; and 4) management is not in control, as there is no clear statement of what the project team is producing.

B. Current practice

Currently, software requirements specifications (when they exist at all) are generally expressed in free-form English. They abound with ambiguous terms ("suitable," "sufficient," "real-time," "flexible") or precise-sounding terms with unspecified definitions ("optimum," "99.9 percent reliable") which are potential seeds of dissension or lawsuits once the software is produced. They have numerous errors; one recent study [7] indicated that the first independent review of a fairly good software requirements specification will find from one to four nontrivial errors per page.

The techniques used for determining software requirements are generally an ad hoc manual blend of systems analysis principles [8] and common sense. (These are the good ones; the poor ones are based on ad hoc manual blends of politics, preconceptions, and pure salesmanship.) Some formalized manual techniques have been used successfully for determining business system requirements, such as accurately defined systems (ADS), and time automated grid (TAG). The book edited by Couger and Knapp [9] has an excellent summary of such techniques.

C. Current frontier technology: Specification languages and systems

1) *ISDOS:* The pioneer system for machine-analyzable software requirements is the ISDOS system developed by Teichroew and his group at the University of Michigan [10]. It was primarily developed for business system applications, but much of the system and its concepts are applicable to other areas. It is the only system to have passed a market and operations test; several commercial, aerospace, and government organizations have paid for it and are successfully using it. The U.S. Air Force is currently using and sponsoring extensions to ISDOS under the Computer Aided Requirements Analysis (CARA) program.

ISDOS basically consists of a problem statement language (PSL) and a problem statement analyzer (PSA). PSL allows the analyst to specify his system in terms of formalized entities (INPUTS, OUTPUTS, REAL WORLD ENTITIES), classes (SETS, GROUPS), relationships (USES, UPDATES, GENERATES), and other information on timing, data volume, synonyms, attributes, etc. PSA operates on the PSL statements to produce a number of useful summaries, such as: formatted problem statements; directories and keyword indices; hierarchical structure reports; graphical summaries of flows and relationships; and statistical summaries. Some of these capabilities are actually more suited to supporting system design activities; this is often the mode in which ISDOS is used.

Many of the current limitations of ISDOS stem from its primary orientation toward business systems. It is currently difficult to express real-time performance requirements and man-machine interaction requirements, for example. Other capabilities are currently missing, such as support for configuration control, traceability to design and code, detailed consistency checking, and automatic simulation generation. Other limitations reflect deliberate, sensible design choices: the output graphics are crude, but they are produced in standard 8½ × 11 in size on any standard line printer. Much of the current work on ISDOS/CARA is oriented toward remedying such limitations, and extending the system to further support software design.

2) *SREP:* The most extensive and powerful system for software requirements specification in evidence today is that being developed under the Software Requirements Engineering Program (SREP) by TRW for the U.S. Army Ballistic Missile Defense Advanced Technology Center (BMDATC) [11, 12, 13]. Portions of this effort are derivative of ISDOS; it uses the ISDOS data management system, and is primarily organized into a language, the requirements statement language (RSL), and an analyzer, the requirements evaluation and validation system (REVS).

SREP contains a number of extensions and innovations which are needed for requirements engineering in real-time software development projects. In order to represent real-time performance requirements, the individual functional requirements can be joined into stimulus-response networks called R-Nets. In order to focus early attention on software testing and reliability, there are capabilities for designating "validation points" within the R-Nets. For early requirements validation, there are capabilities for automatic generation of functional simulators from the requirements statements. And, for adaptation to changing requirements, there are capabilities for configuration control, traceability to design, and extensive report generation and consistency checking.

Current SREP limitations again mostly reflect deliberate design decisions centered around the autonomous, highly real-time process-control problem of ballistic missile defense. Capabilities to represent large file processing and man-machine interactions are missing. Portability is a problem: although some parts run on several machines, other parts of the system run only on a TI-ASC computer with a very powerful but expensive multicolor interactive graphics terminal. However, the system has been designed with the use of compiler generators and extensibility features which should allow these limitations to be remedied.

3) *Automatic programming and other approaches:* Under the sponsorship of the Defense Advanced Research Projects Agency (DARPA), several researchers are attempting to develop "automatic programming" systems to replace the functions of those currently performed by programmers. If successful, could [illegible]

be the need to determine what software the system should produce, i.e., the software requirements. Thus, the methods, or at least the forms, of capturing software requirements are of central concern in automatic programming research.

Two main directions are being taken in this research. One, exemplified by the work of Balzer at USC-ISI [14], is to work within a general problem context, relying on only general rules of information processing (items must be defined or received before they are used, an "if" should have both a "then" and an "else," etc.) to resolve ambiguities, deficiencies, or inconsistencies in the problem statement. This approach encounters formidable problems in natural language processing and may require further restrictions to make it tractable.

The other direction, exemplified by the work of Martin at MIT [15], is to work within a particular problem area, such as inventory control, where there is enough of a general model of software requirements and acceptable terminology to make the problems of resolving ambiguities, deficiencies, and inconsistencies reasonably tractable.

This second approach has, of course, been used in the past in various forms of "programming-by-questionnaire" and application generators [1, 2]. Perhaps the most widely used are the parameterized application generators developed for use on the IBM System/3. IBM has some more ambitious efforts on requirements specification underway, notably one called the Application Software Engineering Tool [16] and one called the Information Automat [17], but further information is needed to assess their current status and directions.

Another avenue involves the formalization and specification of required properties in a software specification (reliability, maintainability, portability, etc.). Some success has been experienced here for small-to-medium systems, using a "Requirements-Properties Matrix" to help analysts infer additional requirements implied by such considerations [18].

D. Trends

In the area of requirements statement languages, we will see further efforts either to extend the ISDOS-PSL and SREP-RSL capabilities to handle further areas of application, such as man-machine interactions, or to develop language variants specific to such areas. It is still an open question as to how general such a language can be and still retain its utility. Other open questions are those of the nature, "which representation scheme is best for describing requirements in a certain area?" BMDATC is sponsoring some work here in representing general data-processing system requirements for the BMD problem, involving Petri nets, state transition diagrams, and predicate calculus [11].

A good deal more can and will be done to extend the capability of requirements statement analyzers. Some extensions are fairly straightforward consistency checking; others, involving the use of relational operators to deduce derived requirements and the detection (and perhaps generation) of missing requirements are more difficult, tending toward the automatic programming work.

Other advances will involve the use of formal requirements statements to improve subsequent parts of the software life cycle. Examples include requirements-design-code consistency checking (one initial effort is underway), the automatic generation of test cases from requirements statements, and, of course, the advances in automatic programming involving the generation of code from requirements.

Progress will not necessarily be evolutionary, though. There is always a good chance of a breakthrough: some key concept which will simplify and formalize large regions of the problem space. Even then, though, there will always remain difficult regions which will require human insight and sensitivity to come up with an acceptable set of software requirements.

Another trend involves the impact of having formal, machine-analyzable requirements (and design) specifications on our overall inventory of software code. Besides improving software reliability, this will make our software much more portable; users will not be tied so much to a particular machine configuration. It is interesting to speculate on what impact this will have on hardware vendors in the future.

IV. Software design

A. The requirements/design dilemma

Ideally, one would like to have a complete, consistent, validated, unambiguous, machine-independent specification of software requirements before proceeding to software design. However, the requirements are not really validated until it is determined that the resulting system can be built for a reasonable *cost* — and to do so requires developing one or more software *designs* (and any associated hardware designs needed).

This dilemma is complicated by the huge number of degrees of freedom available to software/hardware system designers. In the 1950's, as indicated by Table 1, the designer had only a few alternatives to choose from in selecting a central processing unit (CPU), a set of peripherals, a programming language, and an ensemble of support software. In the 1970's, with rapidly evolving mini- and microcomputers, firmware, modems, smart terminals, data management systems, etc., the designer has an enormous number of alternative design components to sort out (possibilities) and to seriously choose from (likely choices). By the 1980's, the number of possible design combinations will be formidable.

Table 1
Design Degrees of Freedom for New Data Processing Systems (Rough Estimates)

Element	Choices (1950's)	Possibilities (1970's)	Likely Choices (1970's)
CPU	5	200	100
Op-Codes	fixed	variable	variable
Peripherals (per function)	1	200	100
Programming language	1	50	5–10
Operating system	0–1	10	5
Data management system	0	100	30

The following are some of the implications for the designer. 1) It is easier for him to do an outstanding design job. 2) It is easier for him to do a terrible design job. 3) He needs more powerful analysis tools to help him sort out the alternatives. 4) He has more opportunities for designing-to-cost. 5) He has more opportunities to design and develop tunable systems. 6) He needs a more flexible requirements-tracking and hardware procurement mechanism to support the above flexibility (particularly in government systems). 7) Any rational standardization (e.g., in programming languages) will be a big help to him, in that it reduces the number of alternatives he must consider.

B. Current practice

Software design is still almost completely a manual process. There is relatively little effort devoted to design validation and risk analysis before committing to a particular software design. Most software errors are made during the design phase. As seen in Fig. 4, which summarizes several software error analyses by IBM [4, 19] and TRW [20, 21], the ratio of design to coding errors generally exceeds 60:40. (For the TRW data, an error was called a design error if and only if the resulting fix required a change in the detailed design specification.)

Most software design is still done bottom-up, by developing software components before addressing interface and integration issues. There is, however, increasing successful use of top-down design. There is little organized knowledge of what a software designer does, how he does it, or of what makes a good software designer, although some initial work along these lines has been done by Freeman [22].

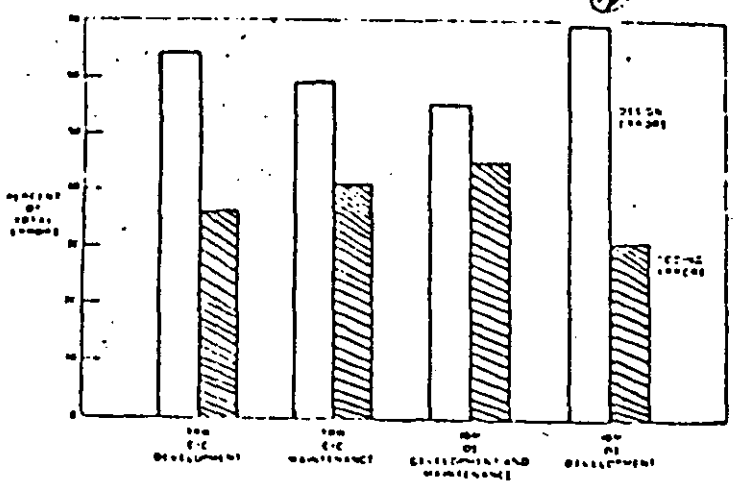


Figure 4. Most errors in large software systems are in early stages.

C. *Current frontier technology*

Relatively little is available to help the designer make the overall hardware-software tradeoff analyses and decisions to appropriately narrow the large number of design degrees of freedom available to him. At the micro level, some formalisms such as LOGOS [23] have been helpful, but at the macro level, not much is available beyond general system engineering techniques. Some help is provided via improved techniques for simulating information systems, such as the Extendable Computer System Simulator (ECSS) [24, 25], which make it possible to develop a fairly thorough functional simulation of the system for design analysis in a considerably shorter time than it takes to develop the complete design itself.

1) *Top-down design:* Most of the helpful new techniques for software design fall into the category of "top-down" approaches, where the "top" is already assumed to be a firm, fixed requirements specification and hardware architecture. Often, it is also assumed that the data structure has also been established. (These assumptions must in many cases be considered potential pitfalls in using such top-down techniques.)

What the top-down approach does well, though, is to provide a procedure for organizing and developing the control structure of a program in a way which focuses early attention on the critical issues of integration and interface definition. It begins with a top-level expression of a hierarchical control structure (often a top level "executive" routine controlling an "input," a "process," and an "output" routine) and proceeds to iteratively refine each successive lower-level component until the entire system is specified. The successive refinements, which may be considered as "levels of abstraction" or "virtual machines" [26], provide a number of advantages in improved understanding, communication, and verification of complex designs [27, 28]. In general, though, experience shows that some degree of early attention to bottom-level design issues is necessary on most projects [29].

The technology of top-down design has centered on two main issues. One involves establishing guidelines for *how to perform* successive refinements and to group functions into modules; the other involves techniques of *representing* the design of the control structure and its interaction with data.

2) *Modularization:* The techniques of structured design [30] (or composite design [31]) and the modularization guidelines of Parnas [32] provide the most detailed thinking and help in the area of module definition and refinement. Structured design establishes a number of successively stronger types of binding of functions into modules (coincidental, logical, classical, procedural, communicational, informational, and functional) and provides the guideline that a function should be grouped with those functions to which its binding is the strongest. Some designers are able to use this approach quite successfully; others find it useful for reviewing designs but not for formulating them, and others simply find it too ambiguous or complex to be of help. Further experience will be needed to determine how much of this is simply a learning curve effect. In general, Parnas' modularization criteria and guidelines are more straightforward and widely used than the levels-of-binding guidelines, although they may also be becoming more complicated as they address such issues as distribution of responsibility for erroneous inputs [33]. Along these lines, Draper Labs' Higher Order Software (HOS) methodology [34] has attempted to resolve such issues via a set of six axioms covering relations between modules and data, including responsibility for erroneous inputs. For example, Axiom 5 states, "Each module controls the rejection of invalid elements of its own, and only its own, input set."*

3) *Design representation:* Flow charts remain the main method currently used for design representation. They have a number of deficiencies, particularly in representing hierarchical control structures and data interactions. Also, their free-form nature makes it too easy to construct complicated, unstructured designs which are hard to understand and maintain. A number of representation schemes have been developed to avoid these deficiencies.

---

*Problems can arise, however, when one furnishes such a design choice with the power of an axiom. Suppose, for example, the input set contains a huge table or a master file. Is the module stuck with the job of checking it, by itself, every time?

(13)

The hierarchical input-process-output (HIPO) technique [35] represents software in a hierarchy of modules, each of which is represented by its inputs, its outputs, and a summary of the processing which connects the inputs and outputs. Advantages of the HIPO technique are its ease of use, ease of learning, easy-to-understand graphics, and disciplined structure. Some general disadvantages are the ambiguity of the control relationships (are successive lower level modules in sequence, in a loop, or in an if/else relationship?), the lack of summary information about data, the unwieldiness of the graphics on large systems, and the manual nature of the technique. Some attempts have been made to automate the representation and generation of HIPO's such as Univac's PROVAC System [36].

The structure charts used in structured design [30, 31] remedy some of these disadvantages, although they lose the advantage of representing the processes connecting the inputs with the outputs. In doing so, though, they provide a more compact summary of a module's inputs and outputs which is less unwieldy on large problems. They also provide some extra symbology to remove at least some of the sequence/loop/branch ambiguity of the control relationships.

Several other similar conventions have been developed [37, 38, 39], each with different strong points, but one main difficulty of any such manual system is the difficulty of keeping the design consistent and up-to-date, especially on large problems. Thus, a number of systems have been developed which store design information in machine-readable form. This simplifies updating (and reduces update errors) and facilitates generation of selective design summaries and simple consistency checking. Experience has shown that even a simple set of automated consistency checks can catch dozens of potential problems in a large design specification [21]. Systems of this nature that have been reported include the Newcastle TOPD system [40], TRW's DACC and DEVISE systems [21], Boeing's DECA system [41], and Univac's PROVAC [36]; several more are under development.

Another machine-processable design representation is provided by Caine, Farber, and Gordon's Program Design Language (PDL) System [42]. This system accepts constructs which have the form of hierarchical structured programs, but instead of the actual code, the designer can write some English text describing what the segment of code will do. (This representation was originally called "structured pidgin" by Mills [43].) The PDL system again makes updating much easier; it also provides a number of useful formatted summaries of the design information, although it still lacks some wished-for features to support terminology control and version control. The program-like representation makes it easy for programmers to read and write PDL, albeit less easy for nonprogrammers. Initial results in using the PDL system on projects have been quite favorable.

(14)

Once a good deal of design information is in machine-readable form, there is a fair amount of pressure from users to do more with it: to generate core and time budgets, software cost estimates, first-cut data base descriptions, etc. We should continue to see such added capabilities, and generally a further evolution toward computer-aided-design systems for software. Besides improvements in determining and representing control structures, we should see progress in the more difficult area of data structuring. Some initial attempts have been made by Hoare [44] and others to provide a data analog of the basic control structures in structured programming, but with less practical impact to date. Additionally, there will be more integration and traceability between the requirements specification, the design specification, and the code — again with significant implications regarding the improved portability of a user's software.

The proliferation of minicomputers and microcomputers will continue to complicate the designer's job. It is difficult enough to derive or use principles for partitioning software jobs on single machines; additional degrees of freedom and concurrency problems just make things so much harder. Here again, though, we should expect at least some initial guidelines for decomposing information processing jobs into separate concurrent processes.

It is still not clear, however, how much one can formalize the software design process. Surveys of software designers have indicated a wide variation in their design styles and approaches, and in their receptiveness to using formal design procedures. The key to good software design still lies in getting the best out of good people, and in structuring the job so that the less-good people can still make a positive contribution.

V. Programming

This section will be brief, because much of the material will be covered in the companion article by Wegner on "Computer Languages" [3].

A. Current practice

Many organizations are moving toward using structured code [28, 43] (hierarchical, block-oriented code with a limited number of control structures — generally SEQUENCE, IFTHENELSE, CASE, DOWHILE, and DOUNTIL — and rules for formatting and limiting module size). A great deal of terribly unstructured code is still being written, though, often in assembly language and particularly for the rapidly proliferating minicomputers and microcomputers.

B. Current frontier technology

Languages are becoming available which support structured code and additional valuable features such as data typing and type checking (e.g., Pascal [45]). Extensions such as concurrent Pascal [46] have been developed to support the programming of concurrent processes. Extensions to data typing involving more explicit binding of procedures and their data have been embodied in recent languages such as ALPHARD [47] and CLU [48]. Metacompiler and compiler writing system technology continues to improve, although much more slowly in the code generation area than in the syntax analysis area.

Automated aids include support systems for top-down structured programming such as the Program Support Library [49], Process Construction [50], TOPD [40], and COLUMBUS [51]. Another novel aid is the Code Auditor program [50] for automated standards compliance checking — which guarantees that the standards are more than just words. Good programming practices are now becoming codified into style handbooks, i.e., Kernighan and Plauger [52] and Ledgard [53].

C. Trends

It is difficult to clean up old programming languages or to introduce new ones into widespread practice. Perhaps the strongest hope in this direction is the current Department of Defense (DoD) effort to define requirements for its future higher order programming languages [54], which may eventually lead to the development and widespread use of a cleaner programming language. Another trend will be an increasing capability for automatically generating code from design specifications.

VI. Software testing and reliability

A. Current practice

Surprisingly often, software testing and reliability activities are still not considered until the code has been run the first time and found not to work. In general, the high cost of testing (still 40-50 percent of the development effort) is due to the high cost of reworking the code at this stage (see Fig. 3), and to the wasted effort resulting from the lack of an advance test plan to efficiently guide testing activities.

In addition, most testing is still a tedious manual process which is error-prone in itself. There are few effective criteria used for answering the question, "How much testing is enough?" except the usual "when the budget (or schedule) runs out." However, more and more organizations are now using disciplined test planning and some objective criteria such as "exercise every instruction" or "exercise every branch," often with the aid of automated test monitoring tools and test case planning aids. But other technologies, such as mathematical proof techniques, have barely begun to penetrate the world of production software.

B. Current frontier technology

1) *Software reliability models and phenomenology:* Initially, attempts to predict software reliability (the probability of future satisfactory operation of the software) were made by applying models derived from hardware reliability analysis and fitting them to observed software error rates [55]. These models worked at times, but often were unable to explain actual experienced error phenomena. This was primarily because of fundamental differences between software phenomenology and the hardware-oriented assumptions on which the models were based. For example, software components do not degrade due to wear or fatigue; no imperfection or variations are introduced in making additional copies of a piece of software (except possibly for a class of easy-to-check copying errors); repair of a software fault generally results in a different software configuration than previously, unlike most hardware replacement repairs.

Models are now being developed which provide explanations of the previous error histories in terms of appropriate software phenomenology. They are based on a view of a software program as a mapping from a space of inputs into a space of outputs [56], of program operation as the processing of a sequence of points in the input space, distributed according to an operational profile [57], and of testing as a sampling of points from the input space [56] (see Fig. 5). This approach encounters severe problems of scale on large programs, but can be used conceptually as a means of appropriately conditioning time-driven reliability models [58]. Still, we are a long way off from having truly reliable reliability-estimation methods for software.

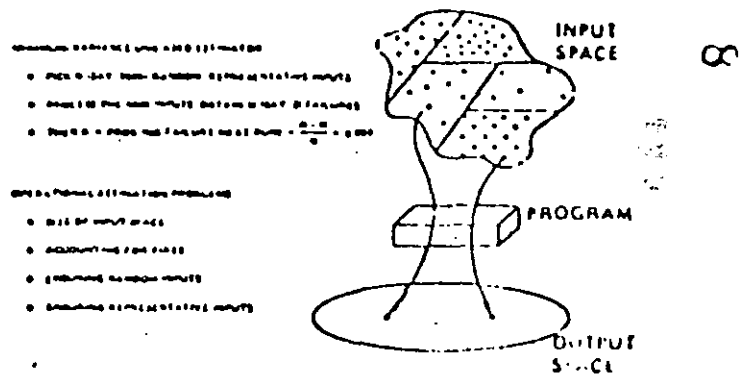


Figure 5. Input space sampling provides a basis for software reliability measurement.

2) *Software error data:* Additional insights into reliability estimation have come from analyzing the increasing data base of software errors. For example, the fact that the distributions of serious software errors are dissimilar from the distributions of minor errors [59] means that we need to define "errors" very carefully when using reliability prediction models. Further, another study [60] found that the rates of fixing serious errors and of fixing minor errors vary with management direction. ("Close out all problems quickly" generally gets minor simple errors fixed very quickly, as compared to "Get the serious problems fixed first.")

Other insights afforded by software data collection include better assessments of the relative efficacy of various software reliability techniques [4, 19, 60], identification of the requirements and design phases as key leverage points for cost savings by eliminating errors earlier (Figs. 2 and 3), and guidelines for organizing test efforts (for example, one recent analysis indicated that over half the errors were experienced when the software was handling data singularities and extreme points [60]). So far, however, the proliferation of definitions of various terms (error, design phase, logic error, validation test), still make it extremely difficult to compare error data from different sources. Some efforts to establish a unified software reliability data base and associated standards, terminology and data collection procedures are now under way at USAF Rome Air Development Center, and within the IEEE Technical Committee on Software Engineering.

3) *Automated aids:* Let us sketch the main steps of testing between the point the code has been written and the point it is pronounced acceptable for use, and describe for each step the main types of automated aids which have been found helpful. More detailed discussion of these aids can be found in the surveys by Reifer [61] and Ramamoorthy and Ho [62] which in turn have references to individual contributions to the field.

a) *Static code analysis:* Automated aids here include the usual compiler diagnostics, plus extensions involving more detailed data-type checking. Code auditors check for standards compliance, and can also perform various type-checking functions. Control flow and reachability analysis is done by structural analysis programs (flow charters have been used for some of the elementary checks here, "structurizers" can also be helpful). Other useful static analysis tools perform set-use analysis of data elements, singularity analysis, units consistency analysis, data base consistency checking, and data-versus-code consistency checking.

b) *Test case preparation:* Extensions to structural analysis programs provide assistance in choosing data values which will make the program execute along a desired path. Attempts have been made to automate the generation of such data values; they can generally succeed for simple cases, but run into difficulty in handling loops or

branching on complex calculated values (e.g., the results of numerical integration). Further, these programs only help generate the *inputs;* the tester must still calculate the expected outputs himself.

Another set of tools will automatically insert instrumentation to verify that a desired path has indeed been exercised in the test. A limited capability exists for automatically determining the minimum number of test cases required to exercise all the code. But, as yet, there is no tool which helps to determine the most appropriate sequence in which to run a series of tests.

c) *Test monitoring and output checking:* Capabilities have been developed and used for various kinds of dynamic data-type checking and assertion checking, and for timing and performance analysis. Test output post-processing aids include output comparators and exception report capabilities, and test-oriented data reduction and report generation packages.

d) *Fault isolation, debugging:* Besides the traditional tools — the core dump, the trace, the snapshot, and the breakpoint — several capabilities have been developed for interactive replay or backtracking of the program's execution. This is still a difficult area, and only a relatively few advanced concepts have proved generally useful.

e) *Retesting (once a presumed fix has been made):* Test data management systems (for the code, the input data, and the comparison output data) have been shown to be most valuable here, along with comparators to check for the differences in code, inputs, and outputs between the original and the modified program and test case. A promising experimental tool performs a comparative structure analysis of the original and modified code, and indicates which test cases need to be rerun.

f) *Integration of routines into systems:* In general, automated aids for this process are just larger scale versions of the test data management systems above. Some additional capabilities exist for interface consistency checking, e.g., on the length and form of parameter lists or data base references. Top-down development aids are also helpful in this regard.

g) *Stopping:* Some partial criteria for thoroughness of testing can and have been automatically monitored. Tools exist which keep a cumulative tally of the number or percent of the instructions or branches which have been exercised during the test program, and indicate to the tester what branch conditions must be satisfied in order to completely exercise all the code or branches. Of course, these are far from complete criteria for determining when to stop testing; the completeness question is the subject of the next section.

4) *Test sufficiency and program proving:* If a program's input space and output space are finite (where the input space includes not only all possible incoming inputs, but also all possible values in the program's data base), then one can construct a set of "black box" tests (one for each point in the input space) which can show conclusively that the program is correct (that its behavior matches its specification).

In general, though, a program's input space is infinite; for example, it must generally provide for rejecting unacceptable inputs. In this case, a finite set of black-box tests is not a sufficient demonstration of the program's correctness (since, for any input x, one must assure that the program does not wrongly treat it as a special case). Thus, the demonstration of correctness in this case involves some formal argument (e.g., a proof using induction) that the dynamic performance of the program indeed produces the static transformation of the input space indicated by the formal specification for the program. For finite portions of the input space, a successful exhaustive test of all cases can be considered as a satisfactory formal argument. Some good initial work in sorting out the conditions under which testing is equivalent to proof of a program's correctness has been done by Goodenough and Gerhart [63] and in a review of their work by Wegner [64].

5) *Symbolic execution:* An attractive intermediate step between program testing and proving is "symbolic execution," a manual or automated procedure which operates on symbolic inputs (e.g., variable names) to produce symbolic outputs. Separate cases are generated for different execution paths. If there are a finite number of such paths, symbolic execution can be used to demonstrate correctness, using a finite symbolic input space and output space. In general, though, one cannot guarantee a finite number of paths. Even so, symbolic execution can be quite valuable as an aid to either program testing or proving. Two fairly powerful automated systems for symbolic execution exist, the EFFIGY system [65] and the SELECT system [66].

6) *Program proving (program verification):* Program proving (increasingly referred to as program verification) involves expressing the program specifications as a logical proposition, expressing individual program execution statements as logical propositions, expressing program branching as an expansion into separate cases, and performing logical transformations on the propositions in a way which ends by demonstrating the equivalence of the program and its specification. Potentially infinite loops can be handled by inductive reasoning.

In general, nontrivial programs are very complicated and time-consuming to prove. In 1973, it was estimated that about one man-month of expert effort was required to prove 100 lines of code [67]. The largest program to be proved correct to date contained about 2000 statements [68]. Again, automation can help out on some of the complications. Some automated verification systems exist, notably those of Good *et al.* [69] and von Henke and Luckham [70]. In general, such systems do not work on programs in the more common languages such as Fortran or Cobol. They work in languages such as Pascal [45], which has (unlike Fortran or Cobol) an axiomatic definition [71] allowing clean expression of program statements as logical propositions. An excellent survey of program verification technology has been given by London [72].

Besides size and language limitations, there are other factors which limit the utility of program proving techniques. Computations on "real" variables involving truncation and roundoff errors are virtually impossible to analyze with adequate accuracy for most nontrivial programs. Programs with nonformalizable inputs (e.g., from a sensor where one has just a rough idea of its bias, signal-to-noise ratio, etc.) are impossible to handle. And, of course, programs can be proved to be consistent with a specification which is itself incorrect with respect to the system's proper functioning. Finally, there is no guarantee that the proof is correct or complete; in fact, many published "proofs" have subsequently been demonstrated to have holes in them [63].

It has been said and often repeated that "testing can be used to demonstrate the presence of errors but never their absence" [73]. Unfortunately, if we must define "errors" to include those incurred by the two limitations above (errors in specifications and errors in proofs), it must be admitted that "program proving can be used to demonstrate the presence of errors but never their absence."

7) *Fault-tolerance:* Programs do not have to be error-free to be reliable. If one could just detect erroneous computations as they occur and compensate for them, one could achieve reliable operation. This is the rationale behind schemes for fault-tolerant software. Unfortunately, both detection and compensation are formidable problems. Some progress has been made in the case of software detection and compensation for hardware errors; see, for example, the articles by Wulf [74] and Goldberg [75]. For software errors, Randell has formulated a concept of separately-programmed, alternate "recovery blocks" [76]. It appears attractive for parts of the error compensation activity, but it is still too early to tell how well it will handle the error detection problem, or what the price will be in program slowdown.

C. Trends

As we continue to collect and analyze more and more data on how, when, where, and why people make software errors, we will get added insights on how to avoid making such errors, how to organize our validation strategy and tactics (not only in testing but throughout the software life cycle), how to develop or evaluate new automated aids, and how to develop useful methods for predicting software reliability. Some automated aids, particularly for static code checking, and for some dynamic-type or assertion checking, will be integrated into future programming languages and compilers. We should see some added useful criteria and associated aids for test completeness, particularly along the lines of ex-

ercising "all data elements" in some appropriate way. Symbolic execution capabilities will probably make their way into automated aids for test case generation, monitoring, and perhaps retesting.

Continuing work into the theory of software testing should provide some refined concepts of test validity, reliability, and completeness, plus a better theoretical base for supporting hybrid test/proof methods of verifying programs. Program proving techniques and aids will become more powerful in the size and range of programs they handle, and hopefully easier to use and harder to misuse. But many of their basic limitations will remain, particularly those involving real variables and nonformalizable inputs.

Unfortunately, most of these helpful capabilities will be available only to people working in higher order languages. Much of the progress in test technology will be unavailable to the increasing number of people who find themselves spending more and more time testing assembly language software written for minicomputers and microcomputers with poor test support capabilities. Powerful cross-compiler capabilities on large host machines and microprogrammed diagnostic emulation capabilities [77] should provide these people some relief after a while, but a great deal of software testing will regress back to earlier generation "dark ages."

VII. Software maintenance

A. Scope of software maintenance

Software maintenance is an extremely important but highly neglected activity. Its importance is clear from Fig. 1: about 40 percent of the overall hardware-software dollar is going into software maintenance today, and this number is likely to grow to about 60 percent by 1985. It will continue to grow for a long time, as we continue to add to our inventory of code via development at a faster rate than we make code obsolete.

The figures above are only very approximate, because our only data so far are based on highly approximate definitions. It is hard to come up with an unexceptional definition of software maintenance. Here, we define it as "the process of modifying existing operational software while leaving its primary functions intact." It is useful to divide software maintenance into two categories: software *update*, which results in a changed functional specification for the software, and software *repair*, which leaves the functional specification intact. A good discussion of software repair is given in the paper by Swanson [78], who divides it into the subcategories of corrective maintenance (of processing, performance, or implementation failures), adaptive maintenance (to changes in the processing or data environment), and perfective maintenance (for enhancing performance or maintainability).

For either update or repair, three main functions are involved in software maintenance [79].

Understanding the existing software: This implies the need for good documentation, good traceability between requirements and code, and well-structured and well-formatted code.

Modifying the existing software: This implies the need for software, hardware, and data structures which are easy to expand and which minimize side effects of changes, plus easy-to-update documentation.

Revalidating the modified software: This implies the need for software structures which facilitate selective retest, and aids for making retest more thorough and efficient.

Following a short discussion of current practice in software maintenance, these three functions will be used below as a framework for discussing current frontier technology in software maintenance.

B. Current practice

As indicated in Fig. 6, probably about 70 percent of the overall cost of software is spent in software maintenance. A recent paper by Elshoff [80] indicates that the figure for General Motors is about 75 percent, and that GM is fairly typical of large business software activities. Daly [5] indicates that about 60 percent of GTE's 10-year life cycle costs for real-time software are devoted to maintenance. On two Air Force command and control software systems, the maintenance portions of the 10-year life cycle costs were about 67 and 72 percent. Often, maintenance is not done very efficiently. On one aircraft computer, software development costs were roughly $75/instruction, while maintenance costs ran as high as $4000/instruction [81].

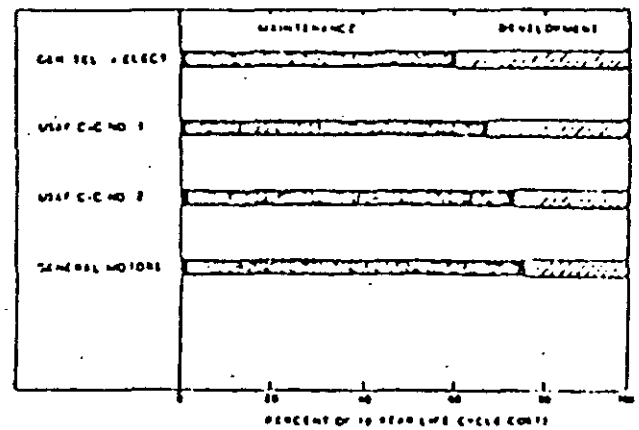


Figure 6. Software life-cycle cost breakdown.

Despite its size, software maintenance is a highly neglected activity. In general, less-qualified personnel are assigned to maintenance tasks. There are few good general principles and few studies of the process, most of them inconclusive.

Further, data processing practices are usually optimized around other criteria than maintenance efficiency. Optimizing around development cost and schedule criteria generally leads to compromises in documentation, testing, and structuring. Optimizing around hardware efficiency criteria generally leads to use of assembly language and skimping on hardware, both of which correlate strongly with increased software maintenance costs [1].

C. Current frontier technology

1) *Understanding the existing software:* Aids here have largely been discussed in previous sections: structured programming, automatic formatting, and code auditors for standards compliance checking to enhance code readability; machine-readable requirements and design languages with traceability support to and from the code. Several systems exist for automatically updating documentation by excerpting information from the revised code and comment cards.

2) *Modifying the existing software:* Some of Parnas' modularization guidelines [32] and the data abstractions of the CLU [48] and ALPHARD [47] languages make it easier to minimize the side effects of changes. There may be a maintenance price, however. In the past, some systems with highly coupled programs and associated data structures have had difficulties with data base updating. This may not be a problem with today's data dictionary capabilities, but the interactions have not yet been investigated. Other aids to modification are structured code, configuration management techniques, programming support libraries, and process construction systems.

3) *Revalidating the modified software:* Aids here were discussed earlier under testing; they include primarily test data management systems, comparator programs, and program structure analyzers with some limited capability for selective retest analysis.

4) *General aids:* On-line interactive systems help to remove one of the main bottlenecks involved in software maintenance: the long turnaround times for retesting. In addition, many of these systems are providing helpful capabilities for text editing and software module management. They will be discussed in more detail under "Management and Integrated Approaches" below. In general, a good deal more work has been done on the maintainability aspects of data bases and data structures than for program structures; a good survey of data base technology is given in a recent special issue of *ACM Computing Surveys* [82].

The increased concern with life cycle costs, particularly within the US DoD [83], will focus a good deal more attention on software maintenance. More data collection and analysis on the growth dynamics of software systems, such as the Belady-Lehman studies of OS/360 [84], will begin to point out the high-leverage areas for improvement. Explicit mechanisms for confronting maintainability issues early in the development cycle, such as the requirements-properties matrix [18] and the design inspection [4] will be refined and used more extensively. In fact, we may evolve a more general concept of software quality assurance (currently focussed largely on reliability concerns), involving such activities as independent reviews of software requirements and design specifications by experts in software maintainability. Such activities will be enhanced considerably with the advent of more powerful capabilities for analyzing machine-readable requirements and design specifications. Finally, advances in automatic programming [14, 15] should reduce or eliminate some maintenance activity, at least in some problem domains.

VIII. Software management and Integrated approaches

A. Current practice

There are more opportunities for improving software productivity and quality in the area of management than anywhere else. The difference between software project successes and failures has most often been traced to good or poor practices in software management. The biggest software management problems have generally been the following.

Poor Planning: Generally, this leads to large amounts of wasted effort and idle time because of tasks being unnecessarily performed, overdone, poorly synchronized, or poorly interfaced.

Poor Control: Even a good plan is useless when it is not kept up-to-date and used to manage the project.

Poor Resource Estimation: Without a firm idea of how much time and effort a task should take, the manager is in a poor position to exercise control.

Unsuitable Management Personnel: As a very general statement, software personnel tend to respond to problem situations as designers rather than as managers.

Poor Accountability Structure: Projects are generally organized and run with very diffuse delineation of responsibilities, thus exacerbating all the above problems.

Inappropriate Success Criteria: Minimizing development costs and schedules will generally yield a hard-to-maintain product. Emphasizing "percent coded" tends to get people coding early and to neglect such key activities as requirements and design validation, test planning, and draft user documentation.

Procrastination on Key Activities: This is especially prevalent when reinforced by inappropriate success criteria as above.

B. Current frontier technology

1) *Management guidelines:* There is no lack of useful material to guide software management. In general, it takes a book-length treatment to adequately cover the issues. A number of books on the subject are now available [85-95], but for various reasons they have not strongly influenced software management practice. Some of the books (e.g., Brooks [85] and the collections by Horowitz [86], Weinwurm [87], and Buxton, Naur, and Randell [88]) are collections of very good advice, ideas, and experiences, but are fragmentary and lacking in a consistent, integrated life cycle approach. Some of the books (e.g., Metzger [89], Shaw and Atkins [90], Hice *et al.* [91], Ridge and Johnson [92], and Gildersleeve [93]) are good on checklists and procedures but (except to some extent the latter two) are light on the human aspects of management, such as staffing, motivation, and conflict resolution. Weinberg [94] provides the most help on the human aspects, along with Brooks [85] and Aron [95], but in turn, these three books are light on checklists and procedures. (A second volume by Aron is intended to cover software group and project considerations.) None of the books have an adequate treatment of some items, largely because they are so poorly understood: chief among these items are software cost and resource estimation, and software maintenance.

In the area of software cost estimation, the paper by Wolverton [96] remains the most useful source of help. It is strongly based on the number of object instructions (modified by complexity, type of application, and novelty) as the determinant of software cost. This is a known weak spot, but not one for which an acceptable improvement has surfaced. One possible line of improvement might be along the "software physics" lines being investigated by Halstead [97] and others; some interesting initial results have been obtained here, but their utility for practical cost estimation remains to be demonstrated. A good review of the software cost estimation area is contained in [98].

2) *Management-technology decoupling:* Another difficulty of the above books is the degree to which they are decoupled from software technology. Except for the Horowitz and Aron books, they say relatively little about the use of such advanced-technology aids as formal, machine-readable requirements, top-down design approaches, structured programming, and automated aids to software testing.

Unfortunately, the management-technology decoupling works the other way, also. In the design area, for example, most treatments of top-down software design are presented as logical exercises independent of user or economic considerations. Most automated aids to software design provide little support for such management needs as configuration management, traceability to code or requirements, and resource estimation and control. Clearly, there needs to be a closer coupling between technology and management than this. Some current efforts to provide integrated management-technology approaches are presented next.

3) *Integrated approaches:* Several major integrated systems for software development are currently in operation or under development. In general, their objectives are similar: to achieve a significant boost in software development efficiency and quality through the synergism of a unified approach. Examples are the utility of having a complementary development approach (top-down, hierarchical) and set of programming standards (hierarchical, structured code); the ability to perform a software update and at the same time perform a set of timely, consistent project status updates (new version number of module, closure of software problem report, updated status logs); or simply the improvement in software system integration achieved when all participants are using the same development concept, ground rules, and support software.

The most familiar of the integrated approaches is the IBM "top-down structured programming with chief programmer teams" concept. A good short description of the concept is given by Baker [49]; an extensive treatment is available in a 15-volume series of reports done by IBM for the U.S. Army and Air Force [99]. The top-down structured approach was discussed earlier. The Chief Programmer Team centers around an individual (the Chief) who is responsible for designing, coding, and integrating the top-level control structure as well as the key components of the team's product; for managing and motivating the team personnel and personally reading and reviewing all their code; and also for performing traditional management and customer interface functions. The Chief is assisted by a Backup programmer who is prepared at anytime to take the Chief's place, a Librarian who handles job submission, configuration control, and project status accounting, and additional programmers and specialists as needed.

In general, the overall ensemble of techniques has been quite successful, but the Chief Programmer concept has had mixed results [99]. It is difficult to find individuals with enough energy and talent to perform all the above functions. If you find one, the project will do quite well; otherwise, you have concentrated most of the project risk in a single individual, without a good way of finding out whether or not he is in trouble. The Librarian and Programming Support Library concept have generally been quite useful, although to date the concept has been oriented toward a batch-processing development environment.

Another "structured" integrated approach has been developed and used at SofTech [38]. It is oriented largely around a hierarchical-decomposition design approach, guided by formalized sets of principles (modularity, abstraction, localization, hiding, uniformity, completeness, confirmability), processes (purpose, concept, mechanism, notation, usage), and goals (modularity, efficiency, reliability, understandability). Thus, it accommodates some economic considerations, although it says little about any other management considerations. It appears to work well for SofTech, but in general has not been widely assimilated elsewhere.

A more management-intensive integrated approach is the TRW software development methodology exemplified in the paper by Williams [50] and the TRW Software Development and Configuration Management Manual [100], which has been used as the basis for several recent government in-house software manuals. This approach features a coordinated set of high-level and detailed management objectives, associated automated aids — standards compliance-checkers, test thoroughness checkers, process construction aids, reporting systems for cost, schedule, core and time budgets, problem identification and closure, etc. — and unified documentation and management devices such as the Unit Development Folder. Portions of the approach are still largely manual, although additional automation is underway, e.g., via the Requirements Statement Language [13].

The SDC Software Factory [101] is a highly ambitious attempt to automate and integrate software development technology. It consists of an interface control component, the Factory Access and Control Executive (FACE), which provides users access to various tools and data bases: a project planning and monitoring system, a software development data base and module management system, a top-down development support system, a set of test tools, etc. As the system is still undergoing development and preliminary evaluation, it is too early to tell what degree of success it will have.

Another factory-type approach is the System Design Laboratory (SDL) under development at the Naval Electronics Laboratory Center [102]. It currently consists primarily of a framework within which a wide range of aids to software development can be incorporated. The initial installment contains text editors, compilers, assemblers, and microprogrammed emulators. Later additions are envisioned to include design, development, and test aids, and such management aids as progress reporting, cost reporting and user profile analysis.

SDL itself is only a part of a more ambitious integrated approach, ARPA's National Software Works (NSW) [102]. The initial objective here has been to develop a "Works Manager" which will allow a software developer at a terminal to access a wide variety of software development tools on various computers available over the ARPANET. Thus, a developer might log into the NSW, obtain his source code from one computer, text-edit it on another, and perhaps continue to hand the program to additional computers for test instrumentation, compiling, executing, and postprocessing of output data. Currently, an initial version of the Works Manager is operational, along with a few tools, but it is too early to assess the likely outcome and payoffs of the project.

C. *Trends*

In the area of management techniques, we are probably entering a consolidation period, particularly as the U.S. DoD proceeds to implement the upgrades in its standards and procedures called for in the recent DoD Directive 5000.29 [104]. The resulting government-industry efforts should produce a set of software management guidelines which are more consistent and up-to-date with today's technology than the ones currently in use. It is likely that they will also be more comprehensible and less encumbered with DoD jargon; this will make them more useful to the software field in general.

Efforts to develop integrated, semiautomated systems for software development will continue at a healthy clip. They will run into a number of challenges which will probably take a few years to work out. Some are technical, such as the lack of a good technological base for data structuring aids, and the formidable problem of integrating complex software support tools. Some are economic and managerial, such as the problems of pricing services, providing tool warranties, and controlling the evolution of the system. Others are environmental, such as the proliferation of minicomputers and microcomputers, which will strain the capability of any support system to keep up-to-date.

Even if the various integrated systems do not achieve all their goals, there will be a number of major benefits from the effort. One is of course that a larger number of support tools will become available to a larger number of people (another major channel of tools will still continue to expand, though: the independent software products marketplace). More importantly, those systems which achieve a degree of conceptual integration (not just a free-form tool box) will eliminate a great deal of the semantic confusion which currently slows down our group efforts throughout the software life cycle. Where we have learned how to talk to each other about our software problems, we tend to do pretty well.

IX. Conclusions

Let us now assess the current state of the art of tools and techniques which are being used to solve software development problems, in terms of our original definition of software engineering: the practical application of *scientific knowledge* in the design and construction of software. Table II presents a summary assessment of the extent to which current software engineering techniques are based on solid scientific principles (versus empirical heuristics). The summary assessment covers four dimensions: the extent to which existing scientific principles apply across the entire software life cycle, across the entire

range of software applications, across the range of engineering-economic analyses required for software development, and across the range of personnel available to perform software development. (29)

Table II
Applicability of Existing Scientific Principles

Dimension	Software Engineering	Hardware Engineering
Scope Across Life Cycle	Some principles for component construction and detailed design, virtually none for system design and integration, e.g., algorithms, automata theory.	Many principles applicable across life cycle, e.g., communication theory, control theory
Scope Across Application	Some principles for "systems" software, virtually none for applications software, e.g., discrete mathematical structures	Many principles applicable across entire application system, e.g., control theory application
Engineering Economics	Very few principles which apply to system economics, e.g., algorithms.	Many principles apply well to system economics, e.g., strength of materials, optimization, and control theory
Required Training	Very few principles formulated for consumption by technicians, e.g., structured code, basic math packages	Many principles formulated for consumption by technicians, e.g., handbooks for structural design, stress testing, maintainability.

For perspective, a similar summary assessment is presented in Table II for hardware engineering. It is clear from Table II that software engineering is in a very primitive state as compared to hardware engineering, with respect to its range of scientific foundations. Those scientific principles available to support software engineering address problems in an area we shall call *Area 1: detailed design and coding* of *systems software* by *experts* in a relatively *economics-independent* context. Unfortunately, the most pressing software development problems are in an area we shall call *Area 2: requirements analysis design, test, and maintenance* of *applications software* by *technicians** in an *economics-driven* context. And in Area 2, our scientific foundations are so slight that one can seriously question whether our current techniques deserve to be called "software engineering."

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

What is rather surprising, and a bit disappointing, is the reluctance of the computer science field to address itself to the more difficult and diffuse problems in Area 2, as compared with the more tractable Area 1 subjects of automata theory, parsing, computability, etc. Like most explorations into the relatively unknown, the risks of addressing Area 2 research problems in the requirements analysis, design, test and maintenance of applications software are relatively higher. But the prizes, in terms of payoff to practical software development and maintenance, are likely to be far more rewarding. In fact, as software engineering begins to solve its more difficult Area 2 problems, it will begin to lead the way toward solutions to the more difficult large-systems problems which continue to beset hardware engineering.

*For example, a recent survey of 14 installations in one large organization produced the following profile of its "average coder": 2 years college-level education, 2 years software experience, familiarity with 2 programming languages and 2 applications, and generally introverted, sloppy, inflexible, "in over his head," and undermanaged. Given the continuing increase in demand for software personnel, one should not assume that this typical profile will improve much. This has strong implications for effective software engineering technology which, like effective software, must be well-matched to the people who must use it.

References

1. B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, Vol. 19, No. 5 (May 1973), pp. 48-59.
2. T.A. Dolotta, et al., *Data Processing in 1980-85* (New York: Wiley-Interscience, 1976).
3. P. Wegner, "Computer Languages," *IEEE Transactions on Computers*, Vol. C-25, No. 12 (December 1976), pp. 1207-25.
4. M.E. Fagan, *Design and Code Inspections and Process Control in the Development of Programs*, IBM Corporation, Report No. IBM-SDD TR-21.572 (December 1974).
5. E.B. Daly, "Management of Software Development," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 3 (May 1977), pp. 230-42.
6. W.W. Royce, "Software Requirements Analysis, Sizing, and Costing," *Practical Strategies for the Development of Large Scale Software*, ed. E. Horowitz (Reading, Mass.: Addison-Wesley, 1975).
7. T.E. Bell and T.A. Thayer, "Software Requirements: Are They a Problem?" *Proceedings of the IEEE/ACM Second International Conference on Software Engineering* (October 1976), pp. 61-68.
8. E.S. Quade, ed., *Analysis for Military Decisions* (Chicago: Rand-McNally, 1964).
9. J.D. Couger and R.W. Knapp, eds., *System Analysis Techniques* (New York: Wiley & Sons, 1974).
10. D. Teichroew and H. Sayani, "Automation of System Building," *Datamation*, Vol. 17, No. 16 (August 15, 1971), pp. 25-30.
11. C.G. Davis and C.R. Vick, "The Software Development System," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 69-84.
12. M. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 60-69.
13. T.E. Bell, D.C. Bixler, and M.E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 49-60.
14. R.M. Balzer, *Imprecise Program Specification*, University of Southern California, Report No. ISI/RR-75-36 (Los Angeles: December 1975).
15. W.A. Martin and M. Bosyj, "Requirements Derivation in Automatic Programming," *Proceedings of the MRI Symposium on Computer Software Engineering* (April 1976).
16. N.P. Dooner and J.R. Lourie, *The Application Software Engineering Tool*, IBM Corporation, Research Report No. RC 5434 (Yorktown Heights, N.Y.: IBM Research Center, May 29, 1975).
17. M.L. Wilson, *The Information Automat Approach to Design and Implementation of Computer-Based Systems*, IBM Corporation (Gaithersburg, Md.: IBM Federal Systems Division, June 27, 1975).
18. B.W. Boehm, "Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design," *Proceedings of the IFIP Congress* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1974), pp. 192-97.
19. A.B. Endres, "An Analysis of Errors and Their Causes in System Programs," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 140-49.
20. T.A. Thayer, "Understanding Software Through Analysis of Empirical Data," *AFIPS Proceedings of the 1975 National Computer Conference*, Vol. 44 (Montvale, N.J.: AFIPS Press, 1975), pp. 335-41.
21. B.W. Boehm, R.L. McClean, and D.B. Urfrig, "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1 (March 1975), pp. 125-33.
22. P. Freeman, *Software Design Representation: Analysis and Improvements*, University of California, Technical Report No. 81 (Irvine, Calif.: May 1976).
23. E.L. Glaser, et al., "The LOGOS Project," *Proceedings of the IEEE Computer Conference* (1972), pp. 175-92.
24. N.R. Nielsen, *ECSS: Extendable Computer System Simulator*, Rand Corporation, Report No. RM-6132-PR/NASA (January 1970).
25. D.W. Kosy, *The ECSS II Language for Simulating Computer Systems*, Rand Corporation, Report No. R-1895-GSA (December 1975).
26. E.W. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability," *Software Engineering, Concepts and Techniques*, P. Naur, B. Randell, and J.N. Buxton, eds. (New York: Petrocelli/Charter, 1976).
27. H.D. Mills, *Mathematical Foundations for Structured Programming*, IBM Corporation, Report No. FSC 72-6012 (Gaithersburg, Md.: IBM Federal Systems Division, February 1972).

28. C.L. McGowan and J.R. Kelly, *Top-Down Structured Programming Techniques* (New York: Petrocelli/Charter, 1975).

29. B.W. Boehm, et al., "Structured Programming: A Quantitative Assessment," *Computer*, Vol. 8, No. 6 (June 1975), pp. 38-54.

30. W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2 (1974), pp. 115-39.

31. G.J. Myers, *Reliable Software Through Composite Design* (New York: Petrocelli/Charter, 1975).

32. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems Into Modules," *Communications of the ACM*, Vol. 15, No. 12 (December 1972), pp. 1053-58.

33. ———, "The Influence of Software Structure on Reliability," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 358-62.

34. M. Hamilton and S. Zeldin, "Higher Order Software — A Methodology for Defining Software," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1 (March 1976), pp. 9-32.

35. *HIPO — A Design Aid and Documentation Technique*, IBM Corporation, Manual No. GC20-1851-0 (White Plains, N.Y.: IBM Data Processing Division, October 1974).

36. J. Mortison, "Tools and Techniques for Software Development Process Visibility and Control," *Proceedings of the ACM Computer Science Conference* (February 1976).

37. I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming," *SIGPLAN Notices*, Vol. 8, No. 8 (August 1973), pp. 12-26.

38. D.T. Ross, J.B. Goodenough, and C.A. Irvine, "Software Engineering: Process, Principles, and Goals," *Computer*, Vol. 8, No. 5 (May 1975), pp. 17-27.

39. M.A. Jackson, *Principles of Program Design* (New York: Academic Press, 1975).

40. P. Henderson and R.A. Snowden, "A Tool For Structured Program Development," *Proceedings of the 1974 IFIP Congress*, (Amsterdam, The Netherlands: North-Holland Publishing Co.), pp. 204-07.

41. L.C. Carpenter and L.L. Tripp, "Software Design Validation Tool," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 395-400.

42. S.H. Caine and E.K. Gordon, "PDL: A Tool for Software Design," *AFIPS Proceedings of the 1975 National Computer Conference*, Vol. 44 (Montvale, N.J.: AFIPS Press, 1975), pp. 271-76.

43. H.D. Mills, *Structured Programming in Large Systems*, IBM Corporation (Gaithersburg, Md.: IBM Federal Systems Division, November 1970).

44. C.A.R. Hoare, "Notes on Data Structuring," *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (New York: Academic Press, 1972).

45. N. Wirth, "An Assessment of the Programming Language Pascal," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 192-98.

46. P. Brinch-Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 199-208.

47. W.A. Wulf, *ALPHARD: Toward a Language to Support Structured Programs*, Carnegie-Mellon University, Internal Report (Pittsburgh, Pa.: April 30, 1974).

48. B.H. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices*, Vol. 9, No. 4 (April 1974), pp. 50-59.

49. F.T. Baker, "Structured Programming in a Production Programming Environment," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 241-52.

50. R.D. Williams, "Managing the Development of Reliable Software," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 3-8.

51. J. Witt, "The COLUMBUS Approach," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 4 (December 1975), pp. 358-63.

52. B.W. Kernighan and P.J. Plauger, *The Elements of Programming Style* (New York: McGraw-Hill, 1974).

53. H.F. Ledgard, *Programming Proverbs* (Rochelle Park, N.J.: Hayden, 1975).

54. W.A. Whitaker, et al., *Department of Defense Requirements for High Order Computer Programming Languages: 'Tinman,'* Defense Advanced Research Projects Agency (April 1976).

55. *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability* (April-May 1973).

56. E.C. Nelson, *A Statistical Basis for Software Reliability Assessment*, TRW Systems Group, Report No. TRW-SS-73-03 (Redondo Beach, Calif.: March 1973).

58. J.D. Musa, "Theory of Software Reliability and Its Application," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 3 (September 1975), pp. 312-27.

59. R.J. Rubey, J.A. Dana, and P.W. Biche, "Quantitative Aspects of Software Validation," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 150-55.

60. T.A. Thayer, M. Lipow, and E.C. Nelson, *Software Reliability Study*, TRW Systems Group, Report to RADC, Contract No. F30602-74-C-0036 (Redondo Beach, Calif.: March 1976).

61. D.J. Reifer, "Automated Aids for Reliable Software," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 131-42.

62. C.V. Ramamoorthy and S.B.F. Ho, "Testing Large Software With Automated Software Evaluation Systems," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1 (March 1975), pp. 46-58.

63. J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 156-73.

64. P. Wegner, "Report on the 1975 International Conference on Reliable Software," in *Findings and Recommendations of the Joint Logistics Commanders' Software Reliability Work Group*, Vol. II (November 1975), pp. 45-88.

65. J.C. King, "A New Approach to Program Testing," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 228-33.

66. R.S. Boyer, B. Elspas, and K.N. Levitt, "Select — A Formal System for Testing and Debugging Programs," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 234-45.

67. J. Goldberg, ed., *Proceedings of the Symposium on High Cost of Software*, Stanford Research Institute (Stanford, Calif.: September 1973), p. 63.

68. L.C. Ragland, "A Verified Program Verifier," Ph.D. Dissertation, University of Texas, 1973.

69. D.I. Good, R.L. London, and W.W. Bledsoe, "An Interactive Program Verification System," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1 (March 1975), pp. 59-67.

70. F.W. von Henke and D.C. Luckham, "A Methodology for Verifying Programs," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 156-64.

71. C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta Informatica*, Vol. 2 (1973), pp. 335-55.

72. R.L. London, "A View of Program Verification," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 534-45.

73. E.W. Dijkstra, "Notes on Structured Programming," *Structured Programming*, O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare (New York: Academic Press, 1972).

74. W.A. Wulf, "Reliable Hardware-Software Architectures," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 233-40.

75. J. Goldberg, "New Problems in Fault-Tolerant Computing," *Proceedings of the 1975 International Symposium on Fault-Tolerant Computing* (Paris, France: June 1975), pp. 29-36.

76. B. Randell, "System Structure for Software Fault-Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 220-32.

77. R.K. McClean and B. Press, "Improved Techniques for Reliable Software Using Microprogrammed Diagnostic Emulation," *Proceedings of the IFAC Congress*, Vol. IV (August 1975).

78. E.B. Swanson, "The Dimensions of Maintenance," *Proceedings of the IEEE/ACM Second International Conference on Software Engineering* (October 1976).

79. B.W. Boehm, J.R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," *Proceedings of the IEEE/ACM Second International Conference on Software Engineering* (October 1976), pp. 592-605.

80. J.L. Elshoff, "An Analysis of Some Commercial PL/I Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2 (June 1976), pp. 113-20.

81. W.L. Trainor, "Software: From Satan to Saviour," *Proceedings of the NAECON* (May 1973).

82. E.H. Sibley, ed., *ACM Computing Surveys*, Vol. 8, No. 1 (March 1976). Special issue on data base management systems.

83. *Defense Management Journal*, Vol. II (October 1975). Special issue on software management.

84. L.A. Belady and M.M. Lehman, *The Evolution Dynamics of Large Programs*, IBM Corporation (Yorktown Heights, N.Y.: IBM Research Center, September 1975).

85. F.P. Brooks, *The Mythical Man-Month* (Reading, Mass.: Addison-Wesley, 1975).

86. E. Horowitz, ed., *Practical Strategies for Developing Large-Scale Software* (Reading, Mass.: Addison-Wesley, 1975).

87. G.F. Weinwurm, ed., *On the Management of Computer Programming* (New York: Auerbach, 1970).

88. P. Naur, B. Randell, and J.N. Buxton, eds., *Software Engineering. Concepts and Techniques. Proceedings of the NATO Conferences* (New York: Petrocelli/Charter, 1976).

89. P.J. Metzger, *Managing a Programming Project* (Englewood Cliffs, N.J.: Prentice-Hall, 1973).

90. J.C. Shaw and W. Atkins, *Managing Computer System Projects* (New York: McGraw-Hill, 1970)

91. G.F. Hice, W.S. Turner, and L.F. Cashwell, *System Development Methodology* (New York: American Elsevier, 1974).

92. W.J. Ridge and L.E. Johnson, *Effective Management of Computer Software* (Homewood, Ill.: Dow Jones-Irwin, 1973).

93. T.R. Gildersleeve, *Data Processing Project Management* (New York: Van Nostrand Reinhold, 1974).

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

95. J.D. Aron, *The Program Development Process: The Individual Programmer* (Reading, Mass.: Addison-Wesley, 1974).

96. R.W. Wolverton, "The Cost of Developing Large-Scale Software," *IEEE Transactions on Computers*, Vol. C-23, No. 6 (June 1974).

97. M.H. Halstead, "Toward a Theoretical Basis for Estimating Programming Effort," *Proceedings of the ACM Conference* (October 1975), pp. 222-24.

98. *Summary Notes, Government/Industry Software Sizing and Costing Workshop*, USAF Electronic Systems Division (October 1974).

99. B.S. Barry and J.J. Naughton, *Chief Programmer Team Operations Description*, U.S. Air Force, Report No. RADC-TR-74-300, Vol. X (of 15-volume series), pp. 1-2-1-3.

100. *Software Development and Configuration Management Manual*, TRW Systems Group, Report No. TRW-SS-73-07 (Redondo Beach, Calif.: December 1973).

101. H. Bratman and T. Court, "The Software Factory," *Computer*, Vol. 8, No. 5 (May 1975), pp. 28-37.

102. *Systems Design Laboratory: Preliminary Design Report*, Naval Electronics Laboratory Center, Preliminary Working Paper TN-3145 (March 1976).

103. W.E. Carlson and S.D. Crocker, "The Impact of Networks on the Software Marketplace," *Proceedings of EASCON* (October 1974).

104. *Management of Computer Resources in Major Defense Systems*, U.S. Department of Defense, Directive 6000.29 (April 1976).

INTRODUCTION

The next article, by Ross and Schoman, is one of three papers chosen for inclusion in this book that deal with the subject of structured analysis. With its companion papers — by Teichroew and Hershey [Paper 23] and by DeMarco [Paper 24] — the paper gives a good idea of the direction that the software field probably will be following for the next several years.

The paper addresses the problems of traditional systems analysis, and anybody who has spent any time as a systems analyst in a large EDP organization immediately will understand the problems and weaknesses of "requirements definition" that Ross and Schoman relate — clearly *not* the sort of problems upon which academicians like Dijkstra, Wirth, Knuth, and most other authors in this book have focused! To stress the importance of proper requirements definition, Ross and Schoman state that "even the best structured programming code will not help if the programmer has been told to solve the wrong problem, or, worse yet, has been given a correct description, but has not understood it."

In their paper, the authors summarize the problems associated with conventional systems analysis, and describe the steps that a "good" analysis approach should include. They advise that the analyst separate his *logical*, or *functional*, description of the system from the *physical* form that it eventually will take; this is difficult for many analysts to do, since they assume, a priori, that the physical implementation of the system will consist of a computer.

Ross and Schoman also emphasize the need to achieve a consensus among typically disparate parties: the user liaison personnel who interface with the developers, the "professional" systems analyst, and management. Since all of these people have different interests and different viewpoints, it becomes all the more important that they have a common frame of reference — a common way of *modeling* the system-to-be. For this need, Ross and Schoman propose their solution: a proprietary package, known as SADT, that was developed by the consulting firm of SofTech for which the au-

The SADT approach utilizes a top-down, partitioned, graphic model of a system. The model is presented in a logical, or abstract, fashion that allows for eventual implementation as a manual system, a computer system, or a mixture of both. This emphasis on graphic models of a system is distinctly different from that of the Teichroew and Hershey paper. It is distinctly similar to the approach suggested by DeMarco in "Structured Analysis and System Specification," the final paper in this collection. The primary difference between DeMarco and Ross/Schoman is that DeMarco and his colleagues at YOURDON inc. prefer circles, or "bubbles," whereas the SofTech group prefers rectangles.

Ross and Schoman point out that their graphic modeling approach can be tied in with an "automated documentation" approach of the sort described by Teichroew and Hershey. Indeed, this approach gradually is beginning to be adopted by large EDP organizations; but for installations that can't afford the overhead of a computerized, automated systems analysis package, Ross and Schoman neglect one important aspect of systems modeling. That is the "data dictionary," in which *all* of the data elements pertinent to the new system are defined *in the same logical top-down fashion as the rest of the model.* There also is a need to formalize mini-specifications, or "mini-specs" as DeMarco calls them; that is, the "business policy" associated with each bottom-level functional process of the system must be described in a manner far more rigorous than currently is being done.

A weakness of the Ross/Schoman paper is its lack of detail about problem solutions: More than half the paper is devoted to a description of the problems of conventional analysis, but the SADT package is described in rather sketchy detail. There are additional documents on SADT available from SofTech, but the reader still will be left with the fervent desire that Messrs. Ross and Schoman and their colleagues at SofTech eventually will sit down and put their ideas into a full-scale book.

Structured Analysis for Requirements Definition

I. The problem

The obvious assertion that "a problem unstated is a problem unsolved" seems to have escaped many builders of large computer application systems. All too often, design and implementation begin before the real needs and system functions are fully known. The results are skyrocketing costs, missed schedules, waste and duplication, disgruntled users, and an endless series of patches and repairs euphemistically called "system maintenance." Compared to other phases of system development, these symptoms reflect, by a large margin, the lack of an adequate approach to requirements definition.

Given the wide range of computer hardware now available and the emergence of software engineering as a discipline, most problems in system development are becoming less traceable to either the machinery or the programming [1]. Methods for handling the hardware and software components of systems are highly sophisticated, but address only part of the job. For example, even the best structured programming code will not help if the programmer has been told to solve the wrong problem, or, worse yet, has been given a correct description, but has not understood it. The results of requirements definition must be both complete and understandable.

In efforts to deal with these needs, the expressions "system architecture," "system design," "system analysis," and "system engineering" seem to be accepted terminology. But in truth, there is no widely practiced methodology for systems work that has the clarity and discipline of the more classical techniques used in construction and manufacturing enterprises. In manufacturing, for example, a succession of blueprints, drawings, and specifications captures all of the relevant requirements for a product. This complete problem definition and implementation plan allows the product to be made almost routinely, by "business as usual," with no surprises. In a good manufacturing operation, major troubles are avoided because even the first production run does not create an item for the first time. The item was created and the steps of forming and assembly were done mentally, in the minds of designers and engineers, long before the set of blueprints and specifications ever arrived at the production shop. That simulation is made possible only because the notations and discipline of the blueprinting methodology are so complete and so consistent with the desired item that its abstract representation contains all the information needed for its imaginary preconstruction.

Software system designers attempt to do the same of course, but being faced with greater complexity and less exacting methods, their successes form the surprises, rather than their failures!

Experience has taught us that system problems are complex and ill-defined. The complexity of large systems is an inherent fact of life with which one must cope. Faulty definition, however, is an artifact of inadequate methods. It can be eliminated by the introduction of well-thought-out techniques and means of expression. That is the subject of this paper: Systems can be manufactured, like other things, if the right approach is used. That approach must start at the beginning.

Requirements definition

Requirements definition includes, but is not limited to, the problem analysis that yields a functional specification. It is much more than that. Requirements definition must encompass everything necessary to lay the groundwork for subsequent stages in system development (Fig. 1). Within the total process, which consists largely of steps in a solution to a problem, only once is the problem itself stated and the solution justified — in requirements definition.

Requirements definition is a careful assessment of the needs that a system is to fulfill. It must say *why* a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say *what* system features will serve and satisfy this context. And it must say *how* the system is to be constructed. Thus, requirements definition must deal with three subjects.

1) *Context analysis:* The reasons *why* the system is to be created and why certain technical, operational, and economic feasibilities are the criteria which form *boundary conditions* for the system.

2) *Functional specification:* A description of *what* the system is to be, in terms of the functions it must accomplish. Since this is part of the problem statement, it must only present *boundary conditions* for considerations later to be taken up in system design.

3) *Design constraints:* A summary of conditions specifying *how* the required system is to be constructed and implemented. This does not necessarily specify which things will be in the system. Rather it identifies *boundary conditions* by which those things may later be selected or created.

Each of these subjects must be fully documented during requirements definition. But note that these are subjects, not documents. The contents of resulting documents will vary according to the needs of the development organization. In any case, they must be reference documents which justify all aspects of the required system, not design or detailed specification documents.

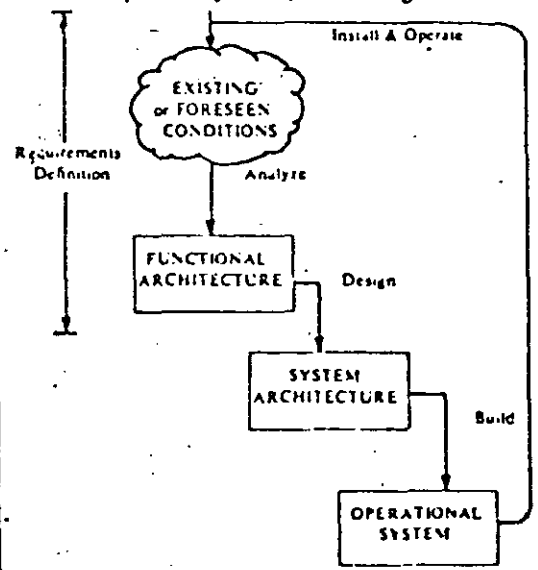


Figure 1. Simplified view of development cycle.

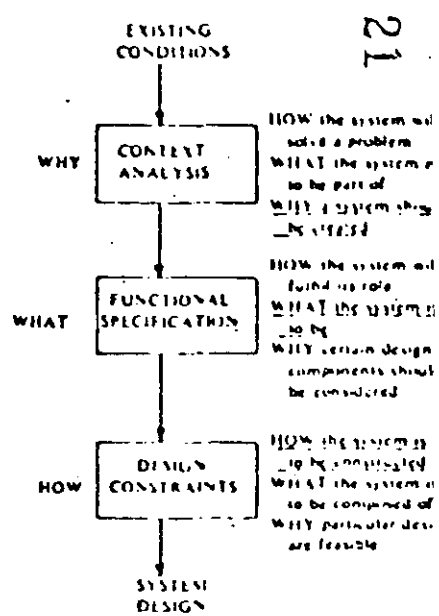


Figure 2. Each subject has a fundamental purpose.

Each of the subjects has a specific and limited role in justifying the required system. Collectively, they capture on paper, at the appropriate time, all relevant knowledge about the system problem in a complete, concise, comprehensive form. Taken together, these subjects define the whole need. Separately, they say *what* the system is to be part of, *what* the system is to be, and of *what* the system is to be composed (Fig. 2). The process known as "analysis" must apply to all three.

Descriptions of these subjects just frame the problem; they do not solve it. Details are postponed, and no binding implementation decisions are yet made. Even detailed budget and schedule commitments are put off, except those for the next phase, system design, because they depend on the results of that design. Requirements definition only (but completely) provides *boundary conditions* for the subsequent design and implementation stages. A problem well-stated is well on its way to a sound solution.

The problem revisited

The question still remains: why is requirements definition not a standard part of every system project, especially since not doing it well has such disastrously high costs [2]? Why is project start-up something one muddles through and why does it seem that requirements are never completely stated? What goes wrong?

The answer seems to be that just about everything goes wrong. Stated requirements are often excessive, incomplete, and inconsistent. Communication and documentation are roadblocks, too. Because they speak with different vocabularies, users and developers find it difficult to completely understand each other. Analysts are often drawn from the development organization, and are unable to document user requirements without simultaneously stating a design approach.

Good requirements are complete, consistent, testable, traceable, feasible, and flexible. By just stating necessary boundary conditions, they leave room for tradeoffs during system design. Thus, a good set of requirements documents can, in effect, serve as a user-developer contract. To attain these attributes, simply proposing a table of contents for requirements documentation is not enough. To do the job, one must emphasize the *means* of defining requirements, rather than prescribe the contents of documentation (which would, in any case, be impossible to do in a generic way).

Even in organizations which do stipulate some document or another, the process of defining requirements remains laborious and inconclusive. Lacking a complete definition of the job to be done, the effect of a contract is lacking, and the designers will make the missing assumptions and decisions because they must, in order to get the job done. Even when the value of requirements definition is recognized, it takes more than determination to have an effective approach. To define requirements, one must understand: 1) the nature of that which is to be described, 2) the form of the description; and 3) the process of analysis.

Many remedies to these problems have been proposed. Each project management, analysis, or specification scheme, whether "structured" or not, has its own adherents. Most do indeed offer some improvements, for any positive steps are better than none. But a significant impact has not been achieved, because each partial solution omits enough of the essential ingredients to be vulnerable.

The key to successful requirements definition lies in remembering that people define requirements. Thus, any useful discussion of requirements definition must combine: 1) a generic understanding of systems which is scientifically sound; 2) a notation and structure of documenting specific system knowledge in a rigorous, easy-to-read form; 3) a process for doing analysis which includes definition of people roles and interpersonal procedures; and 4) a way to technically manage the work, which enables allocation of requirements and postponement of design.

Academic approaches won't do. A pragmatic methodology must itself be: 1) technically feasible, i.e., consistent with the systems to be developed; 2) operationally feasible, i.e., people will use it to do the job well; and 3) economically feasible, i.e., noticeably improve the system development cycle.

This paper sketches such an approach, which has been successful in bringing order and direction to a wide range of system contexts. In even the most trying of circumstances, something can be done to address the need for requirements definition.

II. The process of requirements definition

The nature of systems

One fundamental weakness in current approaches to requirements definition is an inability to see clearly what the problem is, much less measure it, envision workable solutions, or apply any sort of assessment.

It is common practice to think of system architecture in terms of devices, languages, transmission links, and record formats. Overview charts of computer systems typically contain references to programs, files, terminals, and processors. At the appropriate time in system development, this is quite proper. But as an initial basis for system thinking, it is premature and it blocks from view precisely the key idea that is essential to successful requirements definition — the algorithmic nature of all systems. This important concept can best be envisioned by giving it a new name, the *functional architecture*, as distinct from the system architecture.

Systems consist of things that perform activities that interact with other things, each such interaction constituting a happening. A functional architecture is a rigorous layout of the activities performed by a system (temporarily disregarding who or what does them) and the things with which those activities interact. Given this, a design process will create a system architecture which implements, in good order, the functions of the functional architecture. Requirements definition is founded on showing what the functional architecture is (Fig. 3), also showing why it is what it is, and constraining how the system architecture is to realize it in more concrete form.

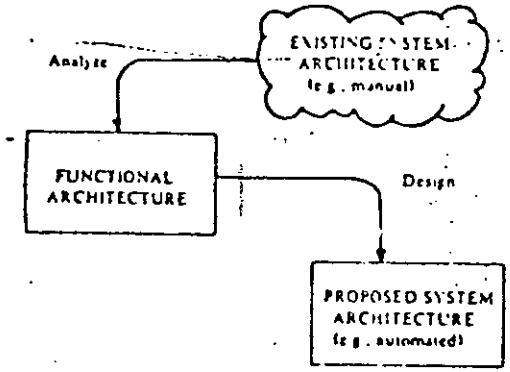


Figure 3. Functional architecture is extracted by analysis.

The concepts of functional architecture are universally applicable, to manual as well as automated systems, and are perfectly suited to the multiple needs of context analysis, functional specification, and design constraints found in requirements definition. Suppose, for example, that an operation, currently being performed manually is to be automated. The manual operation has a system architecture, composed of people, organizations, forms, procedures, and incentives, even though no computer is involved. It also has a functional architecture outlining the purposes for which the system exists. An automated system will implement the functional architecture with a different system architecture. In requirements definition, we must be able to extract the functional architecture (functional specification), and link it both to the boundary conditions for the manual operation (context analysis) and to the boundary conditions for the automated system (design constraints).

Functional architecture is founded on a generic universe of *things* and *happenings:*

objects	operations
data	activities
nouns	verbs
information	processing
substances	events
passive	active

Things and happenings are so intimately related that they can only exist together. A functional architecture always has a very strong structure making explicit these relationships. Functional architecture is, perhaps surprisingly, both abstract and precise.

Precision in functional architecture is best achieved by emphasizing only one primary structural relationship over and over — that of parts and wholes. Parts are related by interfaces to constitute wholes which, in turn, are parts of still larger wholes. It is always valid to express a functional architecture from the generic view that systems are made of components (parts and interfaces) and yet are themselves components.

So like all other architectures, the structure of functional architecture is both modular and hierarchic (even draftsmen and watchmakers use "top-down" methods). Like other architectures, it can be seen from different viewpoints (even the construction trades use distinct structural, electrical, heating, and plumbing blueprints). And like other architectures, it may not necessarily be charted as a physical system would be (even a circuit diagram does not show the actual layout of components, but every important electrical characteristic is represented).

This universal way to view any system is the key to successful requirements definition. From our experience, people do not tend to do this naturally, and not in an organized fashion. In fact, we find that the most basic problem in requirements definition is the fact that most people do not even realize that such universality exists! When it is explained, the usual reaction is, "that is just common sense." But, as has been remarked for ages, common sense is an uncommon commodity. The need is to structure such concepts within a discipline which can be learned.

The form of documentation

To adequately define requirements, one must certainly realize that functional architecture exists, can be measured, and can be evaluated. But to describe it, one needs a communication medium corresponding to the blueprints and specifications that allow manufacturing to function smoothly. In fact, the form of documentation is the key to achieving communication. "Form" includes paging, paragraphing, use of graphics, document organization, and so forth. Because the distinction between form and content is so poorly understood, many adequate system descriptions are unreadable, simply because they are so hard to follow. When Marshall McLuhan said, "The medium is the message," he was apparently ignored by most system analysts.

Analysis of functional architecture (and design of system architecture) cannot be expressed both concisely and unambiguously in natural language. But by imbedding natural language in a blueprint-like graphic framework, all necessary relationships can be shown compactly and rigorously. Well-chosen graphic components permit representation of all aspects of the architecture in an artificial language which is natural for the specific function of communicating that architecture.

The universal nature of systems being both wholes and parts can be expressed by a graphic structure which is both modular and hierarchic (Fig. 4). Because parts are constrained to be wholes, interfaces must be shown explicitly. Interface notation must allow one to distinguish input from output from control (the concept of "control" will not be defended here). Most important, the notation itself must distinguish the things with which activities interact from the things which perform the activities. And because the graphics are a framework for the descriptive abilities of natural language, one must be able to name everything.

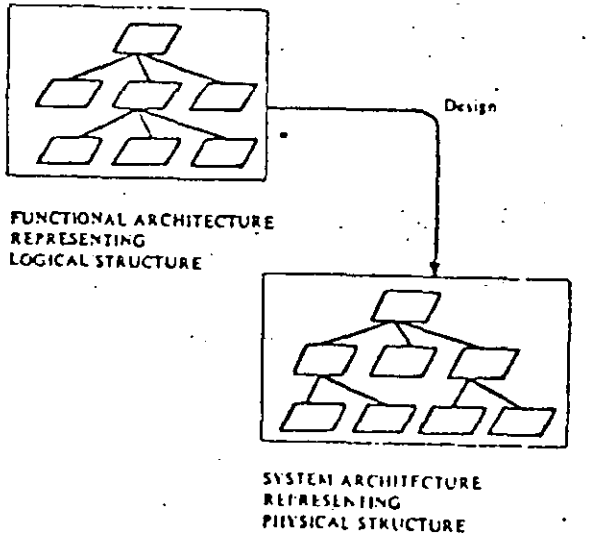


Figure 4. Physical structure is seldom identical to logical structure.

To achieve communication, the form of the diagram in which graphic symbols appear is also important. They must be bounded — to a single page or pair of facing pages — so that a reader can see at once everything which can be said about something. Each topic must be carefully delineated so a reader can grasp the whole message. A reader must be able to mentally walk through the architectural structure which is portrayed, just as blueprints enable a manufacturer to "see" the parts working together. And finally, everything must be in-

devised so that the whole set of diagrams will form a complete model of the architecture.

This appears to be a tall order, but when done elegantly, it yields documentation that is clear, complete, concise, consistent, and convincing. In addition, the hierarchic structure of the documentation can be exploited both to do and to manage the process of analysis. By reviewing the emergent documentation incrementally, for example, while requirements are being delineated, all interested parties can have a voice in directing the process. This is one of the features that results in the standardized, business-as-usual, "no surprises" approach found in manufacturing.

The analysis team

When thinking about why requirements are neither well-structured nor well-documented, one must not forget that any proposed methodology must be people-oriented. Technical matters matter very much, but it is the wishes, ideas, needs, concerns, and skills of people that determine the outcome. Technical aspects can only be addressed through the interaction of all people who have an interest in the system. One of the current difficulties in requirements definition, remember, is that system developers are often charged with documenting requirements. Their design background leads them (however well-intentioned they may be) to think of system architecture rather than functional architecture, and to define requirements in terms of solutions.

Consider a typical set of people who must actively participate in requirements definition. The *customer* is an organization with a need for a system. That customer authorizes a *commissioner* to acquire a system which will be operated by users. The commissioner, although perhaps technically oriented, probably knows less about system technology than the *developers* who will construct the system. These four parties may or may not be within one organization. For each administrative structure, there is a *management* group. Requirements definition must be understood by all these parties, answer the questions they have about the system, and serve as the basis for a development contract.

Each of these parties is a partisan whose conflicting, and often vague, desires must be amalgamated through requirements definition. There is a need at the center for trained, professional *analysts* who act as a catalyst to get the assorted information on paper and to structure from it adequate requirements documentation. The mental facility to comprehend abstraction, the ability to communicate it with personal tact, along with the ability to accept and deliver valid criticism, are all hallmarks of a professional analyst.

Analysts, many of whom may be only part-time, are not expected to supply expertise in all aspects of the problem area. As professionals, they are expected to seek out requirements from experts among the other parties concerned. To succeed, the task of analysis must be properly managed and coordi-

nated, and the requirements definition effort must embody multiple viewpoints. These viewpoints may be overlapping and, occasionally, contradictory.

Managing the analysis

Because many interests are involved, requirements definition must serve multiple purposes. Each subject — context analysis, functional specification, and design constraints — must be examined from at least three points of view: technical, operational, and economic. Technical assessment or feasibility concerns the architecture of a system. Operational assessment concerns the performance of that system in a working environment. Economic feasibility concerns the costs and impacts of system implementation and use. The point is simply that a wide variety of topics must be considered in requirements definition (Fig. 5). Without a plan for assembling the pieces into coherent requirements, it is easy for the analysis team to lose their sense of direction and overstep their responsibility.

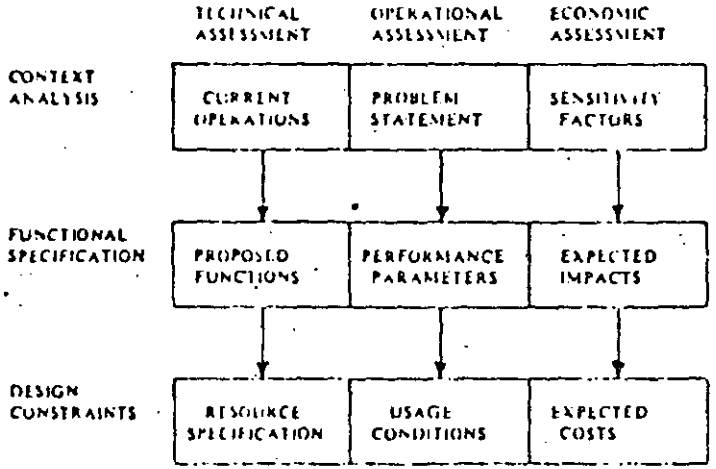


Figure 5. Multiple viewpoints of requirements definition.

Requirements definition, like all system development stages, should be an orderly progression. Each task is a logical successor to what has come before and is completed before proceeding to what comes after. Throughout, one must be able to answer the same three questions: 1) What are we doing? 2) Why are we doing it? 3) How do we proceed from here? Adequate management comes from asking these questions, iteratively, at every point in the development process. And if management is not to be a chameleon-like art, the same body of procedural knowledge should be applicable every time, although the subject at hand will differ.

It is precisely the lack of guidance about the process of analysis that makes requirements definition such a "hot item" today. People need to think about truly analyzing problems ("divide and conquer"), secure in the knowledge that postponed decisions will dovetail because the process they use enforces consistency. Analysis is an art of considering everything relevant at a given point, and nothing more. Adequate requirements will be complete, without over-specification, and will postpone certain decisions to later stages in the system development process without artifice. This is especially important in the development of those complex systems where requirements are imposed by higher level considerations (Fig. 6). Decisions must be *allocated* (i.e., postponed) because too many things interrelate in too many complicated ways for people to understand, all at once, what is and is not being said.

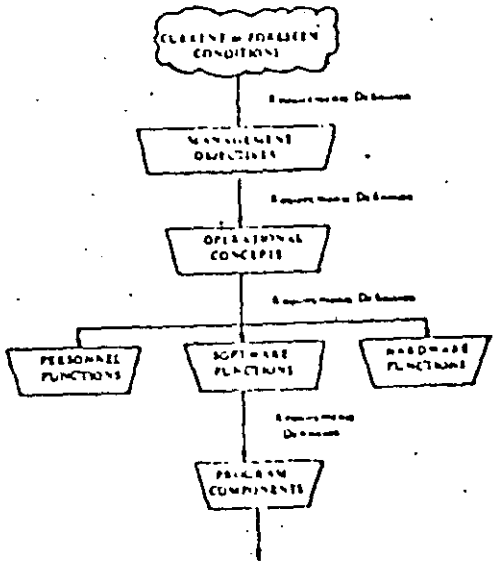


Figure 6. Analysis is repetitive in a complex environment.

Controlling a system development project is nearly impossible without reviews, walk-throughs, and configuration management. Such techniques become workable when the need for synthesis is recognized. Quite simply, system architectures and allocated requirements must be justifiable in light of previously stated requirements. One may choose, by plan or by default, not to enforce such traceability. However, validation and verification of subsequent project stages must not be precluded by ill-structured and unfathomable requirements.

Obviously, decisions on paper are the only ones that count. Knowing that an alternative was considered and rejected, and why, may often be as important as the final requirement. Full documentation becomes doubly necessary when the many parties involved are geographically separated and when staff turnover or expansion may occur before the project is completed.

The features just discussed at length — functional architecture, documentation, analysis teamwork, and the orderly process of analyzing and synthesizing multiple viewpoints — all must be integrated when prescribing a methodology for requirements definition.

III. Structured analysis

Outline of the approach

For several years, the senior author and his colleagues at SofTech have been developing, applying, and improving a general, but practical approach to handling complex system problems. The method is called Structured Analysis and Design Technique (SADT®) [3]. It has been used successfully on a wide range of problems by both SofTech and clients. This paper has presented some of the reasons why SADT works so well, when properly applied.

SADT evolved naturally from earlier work on the foundations of software engineering. It consists of both techniques for performing system analysis and design, and a process for applying these techniques in requirements definition and system development. Both features significantly increase the productivity and effectiveness of teams of people involved in a system project. Specifically, SADT provides methods for: 1) thinking in a structured way about large and complex problems; 2) working as a team with effective division and coordination of effort and roles; 3) communicating interview, analysis, and design results in clear, precise notation; 4) documenting current results and decisions in a way which provides a complete audit of history; 5) controlling accuracy, completeness and quality through frequent review and approval procedure; and 6) planning, managing, and assessing progress of the team effort. Two aspects of SADT deserve special mention: the graphic techniques and the definition of personnel roles.

Graphic techniques

The SADT graphic language provides a limited set of primitive constructs from which analysts and designers can compose orderly structures of any required size. The notation is quite simple — just boxes and arrows. *Boxes* represent parts of a whole in a precise manner. *Arrows* represent interfaces between parts. *Diagrams* represent wholes and are composed of boxes, arrows, natural language names, and certain other notations. The same graphics are applicable to both activities and data.

An SADT model is an organized sequence of diagrams, each with concise supporting text. A high-level overview diagram represents the whole subject. Each lower level diagram shows a limited amount of detail about a well-constrained topic. Further, each lower level diagram connects exactly into higher level portions of the model, thus preserving the logical relationship of each component to the total system (Fig 7).

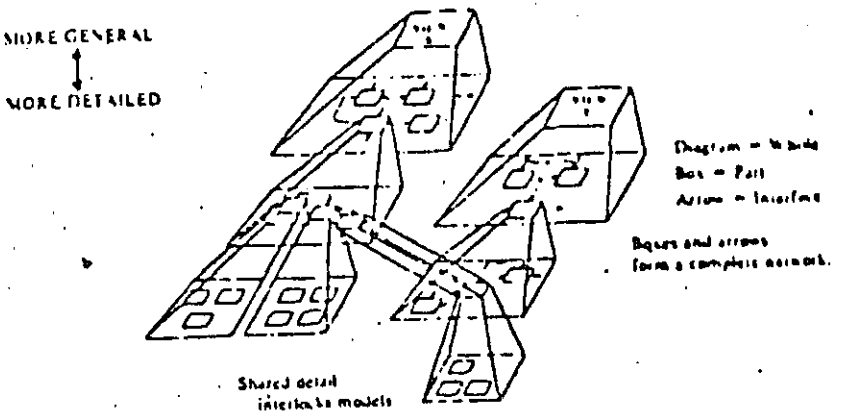


Figure 7. SADT provides practical, rigorous decomposition.

An SADT model is a graphic representation of the hierarchic structure of a system, decomposed with a firm purpose in mind. A model is structured so that it gradually exposes more and more detail. But its depth is bounded by the restriction of its vantage point and its content is bounded by its viewpoint. The priorities dictated by its purpose determine the layering of the top-down decomposition. Multiple models accommodate both multiple viewpoints and the various stages of system realization.

The arrow structure on an SADT diagram represents a constraint relationship among the boxes. It does not represent flow of control or sequence, as for example, on a flowchart for a computer program. Constraint arrows show necessary conditions imposed on a box.

Most arrows represent interfaces between boxes, whether in the same or different models. Some arrows represent noninterface interlocking between models. Together, these concepts achieve both overlapping of multiple viewpoints and the desirable attributes of good analysis and design projects (e.g., modularity, flexibility, and so forth [4]). The interface structure, particularly, passes through several levels of diagrams, creating a web that integrates all parts of the decomposition and shows the whole system's environmental interfaces with the topmost box.

Clearly, requirements definition requires cooperative teamwork from many people. This in turn demands a clear definition of the kinds of interactions which should occur between the personnel involved. SADT anticipates this need by establishing titles and functions of appropriate roles (Fig. 8). In a requirements definition effort, for example, the "authors" would be analysts, trained and experienced in SADT.

Name	Function
Authors	Personnel who study requirements and constraints, analyze system functions and represent them by models based on SADT diagrams
Commenters	Usually authors, who must review and comment in writing on the work of other authors
Readers	Personnel who read SADT diagrams for information but are not expected to make written comments
Experts	Persons from whom authors obtain specialized information about requirements and constraints by means of interviews
Technical Committee	A group of senior technical personnel assigned to review the analysts at every major level of decomposition. They either resolve technical issues or recommend a decision to the project management
Project Librarian	A person assigned the responsibility of maintaining a centralized file of all project documents, making copies, distributing, reader-service keeping records, etc.
Project Manager	The member of the project who has the final technical responsibility for carrying out the system analysis and design
Monitor (or Chief Analyst)	A person fluent in SADT who assists and advises project personnel in the use and application of SADT
Instructor	A person fluent in SADT, who trains Authors and Commenters using SADT for the first time

Figure 8. Personnel roles for SADT.

The SADT process, in which these roles interact, meets the needs of requirements definition for continuous and effective communication, for understandable and current documentation, and for regular and critical review. The process exploits the structure of an SADT model so that decisions can be seen in context and can be challenged while alternatives are still viable.

Throughout a project, draft versions of diagrams in evolving models are distributed to project members for review. Commenters make their suggestions in writing directly on copies of the diagrams. Written records of decisions and alternatives are retained as they unfold. As changes and corrections are made, all versions are entered in the project files. A project librarian provides filing, distribution, and record-keeping support, and, not so incidentally, also ensures configuration control.

This process documents all decisions and the reasons why decisions were made. When commenters and authors reach an understanding, the work is reviewed by a committee of senior technical and management personnel. During the process, incorrect or unacceptable results are usually spotted early, and oversights or errors are detected before they can cause major disruptions. Since everything is on record, future enhancement and system maintenance can reference previous analysis and design decisions.

When documentation is produced as the model evolves, the status of the project becomes highly visible. Management can study the requirements (or the design) in a "top-down" manner, beginning with an overview and continuing to any relevant level of detail. Although presentations to upper management usually follow standard summary and walk-through methods, even senior executives sometimes become direct readers, for the blueprint language of SADT is easily learned.

Implementing the approach

How the ideas discussed in this paper are employed will vary according to organization needs and the kinds of systems under consideration. The methodology which has been described is not just a theory, however, and has been applied to a wide range of complex problems from real-time communications to process control to commercial EDP to organization planning. It is, in fact, a total systems methodology, and not merely a software technique. ITT Europe, for example, has used SADT since early 1974 for analysis and design of both hardware/software systems (telephonic and telegraphic switches) and non-software people-oriented problems (project management and customer engineering). Other users exhibit similar diversity, from manufacturing (AFCAM [5]) to military training (TRAIDEX [6]). Users report that it is a communications vehicle which focuses attention on well-defined topics, that it increases management control through visibility and standardization, that it creates a systematic work breakdown structure for project teams, and that it minimizes errors through disciplined flexibility.

There is no set pattern among different organizations for the contents of requirements documentation. In each case, the needs of the users, the commissioner, and the development organization must be accommodated. Government agencies tend to have fixed standards, while other organizations encourage flexibility. In a numerical control application, almost 40 models were generated in requirements definition (SINTEF, University of Trondheim, Norway). At least one supplier of large-scale computer-based systems mandates consideration of system, hardware, software, commercial, and administrative

constraints. Among all distinct viewpoints, the only common ground lies in the vantage points of context analysis, functional specification, and design constraints. Experience has shown that use of well-structured models together with a well-defined process of analysis, when properly carried out, does provide a strong foundation for actual system design [7].

Because local needs are diverse, implementation of the approach discussed in this paper cannot be accomplished solely by the publication of policy or standards. It is very much a "learn by doing" experience in which project personnel acquire ways of understanding the generic nature of systems. One must recognize that a common sense approach to system manufacturing is not now widely appreciated. A change in the ways that people think about systems and about the systems work that they themselves perform cannot be taught or disseminated in a short period of time.

Further developments

In manufacturing enterprises, blueprinting and specification methods evolved long before design support tools. In contrast to that analogy, a number of current efforts have produced systems for automating requirements information, independent of the definition and verification methodology provided by SADT. To date, SADT applications have been successfully carried out manually, but in large projects, where many analysts are involved and frequent changes do occur, the question is not whether or how to automate but simply what to automate.

Existing computer tools which wholly or partly apply to requirements [8] are characterized by a specification (or a design) database which, once input, may be manipulated. All such attempts, however, begin with user requirements recorded in a machine-readable form. Two impediments immediately become evident. The first is that requirements stated in prose texts cannot be translated in a straightforward manner to interface with an automated problem language. The second is that no computer tool will ever perform the process of requirements definition. Defining and verifying requirements is a task done by users and analysts [9].

Given the right kind of information, however, computer tools can provide capabilities to insure consistency, traceability, and allocation of requirements. A good example of this match to SADT is PSL/PSA, a system resulting from several years' effort in the ISDOS Project at the University of Michigan [10, 11]. The PSL database can represent almost every relationship which appears on SADT diagrams, and the input process from diagrams to database can be done by a project librarian. Not only does SADT enhance human communication (between user and analyst and between analyst and designer), but the diagrams become machine-readable in a very straightforward manner.

The PSA data analyzer and report generator is useful for summarizing database contents and can provide a means of controlling revisions. If the database has been derived by SADT, enhancements to existing PSA capabilities are possible to further exploit the structure of SADT models. For example, SADT diagrams are not flow diagrams, but the interface constraints are directed. These precedence relations, systematically pursued in SADT to specify quantities (volumes and rates) and sequences, permit any desired degree of simulation of a model (whether performed mentally or otherwise).

Computer aids are created as support tools. SADT provides a total context, within which certain automated procedures can play a complementary role. The result will be a complete, systematized approach which both suits the needs of the people involved and enables automation to be used in and extended beyond requirements definition. Such comprehensive methods will enable arbitrary systems work to attain the fulfillment that blueprint techniques deliver in traditional manufacturing.

IV. Conclusion

Requirements definition is an important source of costly oversights in present-day system innovation, but something can be done about it. None of the thoughts presented here are mere speculation. All have produced real achievements. The methods described have been successfully applied to a wide range of planning, analysis and design problems involving men, machines, software, hardware, databases, communications, procedures, and finances. The significance of the methods seems to be that a well-structured approach to documenting what someone thinks about a problem or a system can materially aid both that person's thinking and his ability to convey his understanding to others. Properly channeled, the mind of man is indeed formidable. But only by considering all aspects of the task ahead can teamwork be more productive and management be more effective. Communication with nontechnical readers, an understanding of the nature and structure of systems [12], and indeed a thorough knowledge of the process of analysis itself are the essential ingredients of successful requirements definition.

Appendix

Asking the right questions

The basic difficulty in requirements definition is not one of seeing the woods instead of the trees. The real need is for a methodology which, in any given circumstance, will enable an analyst to distinguish one from the other. It is always more important to ask the right questions than it is to assert possibly wrong answers. It is said that when a famous rabbi was asked, "Rabbi, why does a rabbi always answer a question with a question?" he replied, "Is there a b[illegible]r way?" This is the famous dialectic method used by Socrates to lead his

students to understanding. Answering questions with questions leaves options open, and has the nature of breaking big questions into a top-down structure of smaller questions, easier and easier to address. The answers, when they are ultimately supplied, are each less critical and more tractable, and their relations with other small answers are well-structured.

This is the focus of requirement definition. The appropriate questions — *why, what, how* — applied systematically, will distinguish that which must be considered from that which must be postponed. A sequence of such questions, on a global, system-wide scale, will break the complexity of various aspects of the system into simpler, related questions which can be analyzed, developed, and documented. The context analysis, functional specification, and design constraints — subjects which are part of requirements definition — are merely parts of an overlapping chain of responses to the appropriate why, what, how questions (Fig. 9). In different circumstances, the subjects may differ, but the chaining of questions will remain the same. *Why* some feature is needed molds *what* it has to be, which in turn molds *how* it is to be achieved.

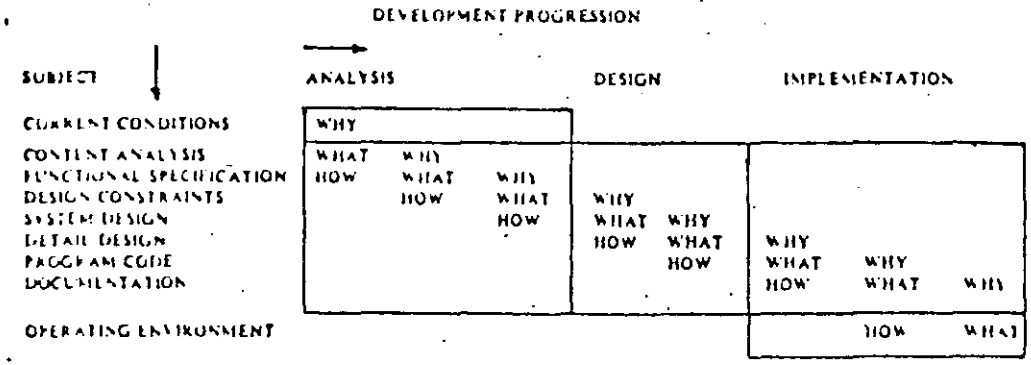


Figure 9. System development is a chain of overlapping questions, documented at each step.

These questions form an overlapping repetition of a common pattern. Each time, the various aspects of the system are partitioned, and the understanding which is developed must be documented in a form consistent with the pattern. It is not sufficient merely to break big problems into little problems by shattering them. "Decompose" is the inverse of "compose"; at every step, the parts being considered must reassemble to make the whole context within which one is working.

The English word "cleave" captures the concept exactly. It is one of those rare words that has antithetical meanings. It means both to separate and to cling to! Thus, in the orderly process of top-down decomposition which describes the desired system, multiple views must intersect in order to supply the whole context for the next stage of system development (Fig. 10).

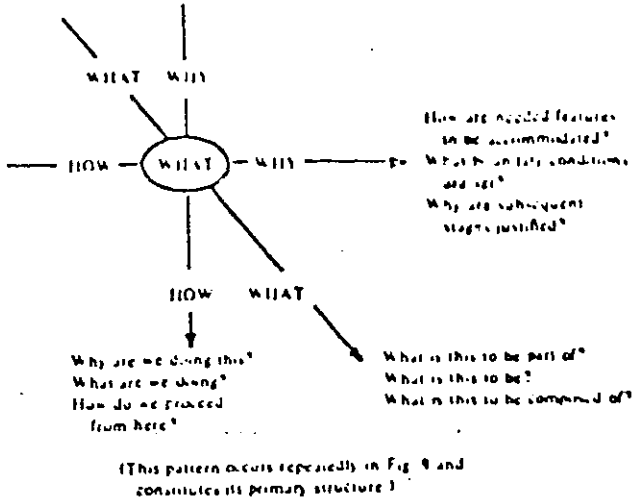


Figure 10. Right questions occur within a context and form a context as well.

And finding the right answers

Probably the most important aspect of this paper is its emphasis on a common approach to all phases of system development, starting with requirements definition. Knowing how postponed decisions will be handled in later phases (by the same methods) allows their essence to be established by boundary conditions, while details are left open (Fig. 11). The knowledge that all requirements must ultimately be implemented somehow (again by the same methods) allows their completeness and consistency to be verified. Finally, an orderly sequence of questions enforces gradual exposition of detail. All information is presented in well-structured documentation which allows first-hand participation by users and commissioners in requirements definition.

The way to achieve such coherence is to seek the right kind of answers to every set of why, what, how questions. By this is meant to establish the viewpoint, vantage point, and purpose of the immediate task before writing any document or conducting any analysis.

The initial *purpose* of the system development effort is established by context analysis. Proper cleaving of subjects and descriptions then takes two different forms, the exact natures of which are governed by the overall project purpose. One cleaving creates partial but overlapping delineations according to *viewpoint* — viewpoint makes clear what aspects are considered relevant to achieving some component of the overall purpose. The other cleaving creates rigorous functional architectures according to *vantage point* — vantage point is a level of abstraction that relates the viewpoints and component purposes which together describe a given whole subject (Fig. 12).

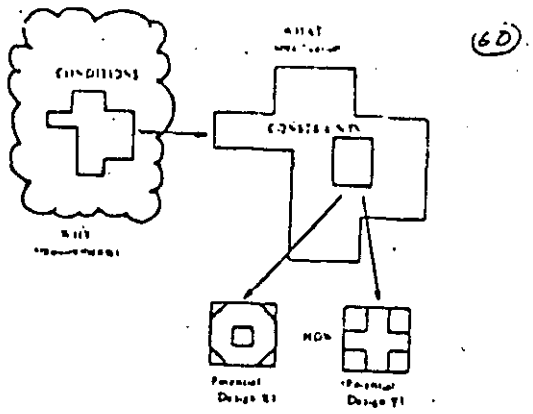


Figure 11. Postponed decisions occur within a prior framework.

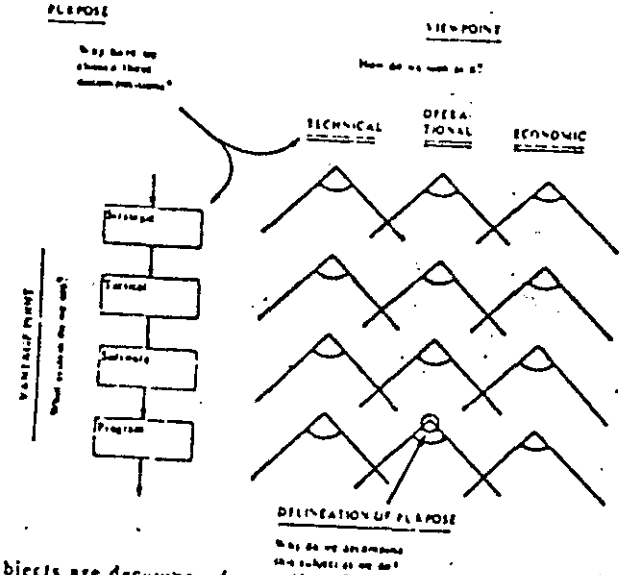


Figure 12. Subjects are decomposed according to viewpoint, vantage point, and purpose.

Depending on the system, any number of viewpoints may be important in requirements definition and may continue to be relevant as the vantage point shifts to designing, building, and finally using the system. Any single viewpoint always highlights some aspects of a subject, while other aspects will be lost from view. For example, the same system may be considered from separate viewpoints which emphasize physical characteristics, functional characteristics, operation, management, performance, maintenance, construction cost, and so forth.

Picking a vantage point always abstracts a functional architecture, while muting implementation-defined aspects of the system architecture. The placement of a vantage point within a viewpoint establishes an all-important *bounded context* — a subset of purpose which then governs the decomposition and exposition of a particular subject regarding the system.

Can give the right results

Requirements definition, beginning with context analysis, can occur whenever there is a need to define, redefine, or further delineate purpose. Context analysis treats the most important, highest level questions first, setting the conditions for further questioning each part in turn. Each subsequent question is asked within the bounded context of a prior response. Strict discipline ensures that each aspect of any question is covered by exactly one further question. Getting started is difficult, but having done so successfully, one is introduced to a "top-down" hierarchy of leading questions, properly sequenced, completely decomposed, and successively answered.

Whether explicitly stated or not, vantage points, viewpoints, and purposes guide the activities of *any* analysis team which approaches a requirements definition task. With the global understanding offered by the above discussion, analysts should realize that a place can be found for every item of information gathered. Structuring this mass of information still must be done, on paper, in a way that ensures completeness and correctness, without overspecification and confusion. That is the role of SADT.

Acknowledgment

The authors would like to thank J.W. Brackett and J.B. Goodenough of SofTech, Inc., Waltham, MA, who made several helpful suggestions incorporated into the presentation of these ideas. Many people at SofTech have, of course, contributed to the development and use of SADT.

1. B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, Vol. 19, No. 5 (May 1973), pp. 48-59.

2. P. Hirsch, "GAO Hits Wimmix Hard: FY'72 Funding Prospects Fading Fast," *Datamation*, Vol. 17, No. 7 (March 1, 1971), p. 41.

3. *An Introduction to SADT™*, SofTech, Inc., Report No. 9022-78 (Waltham, Mass.: February 1976).

4. D.T. Ross, J.B. Goodenough, and C.A. Irvine: "Software Engineering: Process, Principles, and Goals," *Computer*, Vol. 8, No. 5 (May 1975), pp 17-27.

5. Air Force Materials Laboratory, *Air Force Computer-Aided Manufacturing (AFCAM) Master Plan*, Vol. II, App. A, and Vol. III, AFSC, Wright Patterson Air Force Base, Report No. AFML-TR-74-104 (Ohio: July 1974). Available from DDC as AD 922-041L and 922-171L.

6. *TRAIDEX Needs and Implementation Study*, SofTech, Inc., Final Report (Waltham, Mass.: May 1976). Available as No. ED-129244 from ERIC Printing House on Information Resources (Stanford, Calif.).

7. B.W. Boehm, "Software Design and Structure," *Practical Strategies for Developing Large Software Systems*, ed. E. Horowitz (Reading, Mass.: Addison-Wesley, 1975), pp. 115-22.

8. R.V. Head, "Automated System Analysis," *Datamation*, Vol. 17, No. 16 (Aug. 15, 1971), pp. 22-24.

9. J.T. Rigo, "How to Prepare Functional Specifications," *Datamation*, Vol. 20, No. 5 (May 1974), pp. 78-80.

10. D. Teichroew and H. Sayani, "Automation of System Building," *Datamation*, Vol. 17, No. 16 (Aug. 15, 1971), pp. 25-30.

11. D. Teichroew and E.A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 41-48.

12. F.M. Haney, "What It Takes to Make MAC, MIS, and ABM Fly," *Datamation*, Vol. 20, No. 6 (June 1974), pp. 168-69.

INTRODUCTION

I vividly remember a *Datamation* article, written by Daniel Teichroew in 1967, predicting that within ten years programmers would be obsolete. All we had to do, he said, was make it possible for users to state *precisely* what they wanted a computer system to do for them; at that point, it should be possible to mechanically generate the code. Having been in the computer field for only a few years, I was profoundly worried. Maybe it was time to abandon data processing and become a farmer?

Those early ideas of Professor Teichroew perhaps were ahead of their time — after all, there still were a sizable number of programmers in existence in 1977! — but some of his predictions are beginning to take concrete form today. The following paper, written by Teichroew and Hershey and originally published in the January 1977 *IEEE Transactions on Software Engineering*, is the best single source of information on a system for automated analysis, known as "ISDOS" or "PSL/PSA."

The Teichroew/Hershey paper deals specifically with structured analysis, and, as such, is radically different from the earlier papers on programming and design. Throughout the 1960s, it was fashionable to blame all of our EDP problems on *programming*; structured programming and the related disciplines were deemed the ideal solution. Then, by the mid-1970s, emphasis shifted toward *design*, as people began to realize that brilliant code would not save a mediocre design. But now our emphasis has shifted even further: Without an adequate statement of the user's requirements, the best design and the best code in the world won't do the job. Indeed, without benefit of proper requirements definition, structured design and structured programming may simply help us arrive at a systems disaster faster.

So, what can be done to improve the requirements definition process? There are, of course, the methods proposed by Ross and Schoman in the previous paper, as well as those set forth by DeMarco in the paper that appears after this one. In this paper, Teichroew and Hershey concentrate on the *documentation* associated with re-

quirements definition, and on the difficulty of producing and managing manually generated documentation.

The problems that Teichroew and Hershey address certainly are familiar ones, although many systems analysts probably have come to the sad conclusion that things always were meant to be like this. They observe, for example, that much of the documentation associated with an EDP system is not formally recorded until the *end* of the project, by which time, as DeMarco points out, the final document is "of historical significance only" [Paper 24]. The documentation generated is notoriously ambiguous, verbose, and redundant, and, worst of all, it is *manually* produced, *manually* examined for possible errors and inconsistencies, and *manually* updated and revised as user requirements change.

So, the answer to all of this, in Teichroew and Hershey's opinion, is a computer program that allows the systems analyst to input the user requirements in a language that can be regarded as a subset of English; those requirements then can be changed, using facilities similar to those on any modern text-editing system, throughout the analysis phase of the project — *and throughout the entire system life cycle!* Perhaps most important, the requirements definition can be subjected to automated analysis to determine such things as contradictory definitions of data elements, missing definitions, and data elements that are generated but never used.

The concept for PSL was documented by Teichroew more than ten years ago, but it is in this 1977 paper that we begin to get some idea of the impact on the real world. Automated analysis slowly is becoming an option, with a number of large, prestigious organizations using PSL/PSA.

One would expect continued growth; ironically, though, a reported problem is *machine inefficiency!* PSL/PSA apparently consumes significant amounts of CPU time and other computer resources, although Teichroew and Hershey maintain that such tangible, visible costs probably are smaller than the intangible, hidden costs of time wasted during "manual" analysis. It appears that PSL/PSA is most successfully used in organizations that already have vast computer installations and several hundred (or thousand) EDP people. A significant drawback to widespread adoption of PSL/PSA is the fact that it is written in FORTRAN! Admittedly this has helped make it portable — but FORTRAN?

PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems

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

eventually results in an operational system. Most of the activities are data processing activities, in that they use data and information to produce other data and information. Each activity can be regarded as receiving specifications or requirements from preceding activities and producing data which are regarded as specifications or requirements by one or more succeeding activities.

Since many individuals may be involved in the system development process over considerable periods of time and these or other individuals have to maintain the system once it is operating, it is necessary to record descriptions of the system as it evolves. This is usually referred to as "documentation."

In practice, the emphasis in documentation is on describing the system in the final form so that it can be maintained. Ideally, however, each activity should be documented so that the results it produces become the specification for succeeding activities. This does not happen in practice because the communications from one activity to succeeding activities is accomplished either by having the same person carrying out the activities, by oral communication among individuals in a project, or by notes which are discarded after their initial use.

This results in projects which proceed without any real possibility for management review and control. The systems are not ready when promised, do not perform the function the users expected, and cost more than budgeted.

Most organizations, therefore, mandate that the system development process be divided into phases and that certain documentation be produced by the end of each phase so that progress can be monitored and corrections made when necessary. These attempts, however, leave much to be desired and most organizations are attempting to improve the methods by which they manage their system development [20, 6].

This paper is concerned with one approach to improving systems development. The approach is based on three premises. The first is that more effort and attention should be devoted to the front end of the process where a proposed system is being described from the user's point of view [2, 14, 3]. The second premise is that the computer should be used in the development process since systems development involves large amounts of information processing. The third premise is that a computer-aided approach to systems development must start with "documentation."

This paper describes a computer-aided technique for documentation which consists of the following:

1) The results of each of the activities in the system development process are recorded in computer processible form as they are produced.

2) A computerized data base is used to maintain all the basic data about the system.

3) The computer is used to produce hard copy documentation when required.

The part of the technique which is now operational is known as PSL/PSA. Section II is devoted to a brief description of system development as a framework in which to compare manual and computer-aided documentation methods. The Problem Statement Language (PSL) is described in Section III. The reports which can be produced by the Problem Statement Analyzer (PSA) are described in Section IV. The status of the system, results of experience to date, and planned developments are outlined in Section V.

II. Logical systems design

The computer-aided documentation system described in Sections III and IV of this paper is designed to play an integral role during the initial stages in the system development process. A generalized model of the whole system development process is given in Section II-A. The final result of the initial stage is a document which here will be called the System Definition Report. The desired contents of this document are discussed in Section II-B. The activities required to produce this document manually are described in Section II-C and the changes possible through the use of computer-aided methods are outlined in Section II-D.

A. A model of the system development process

The basic steps in the life cycle of information systems (initiation, analysis, design, construction, test, installation, operation, and termination) appeared in the earliest applications of computers to organizational problems (see for example, [17, 1, 4, 7]). The need for more formal and comprehensive procedures for carrying out the life cycle was recognized; early examples are the IBM SOP publications [5], the Philips ARDI method [8], and the SDC method [23]. In the last few years, a large number of books and papers on this subject have been published [11, 19].

Examination of these and many other publications indicate that there is no general agreement on what phases the development process should be divided into, what documentation should be produced at each phase, what it should contain, or what form it should be presented in. Each organization develops its own methods and standards.

In this section a generalized system development process will be described as it might be conducted in an organization which has a Systems Department responsible for developing, operating, and maintaining computer based information processing systems. The System Department belongs to some higher unit in the organization and itself has some subunits, each with certain func-

tions (see for example, [24]). The System Department has a system development standard procedure which includes a project management system and documentation standards.

A request for a new system is initiated by some unit in the organization or the system may be proposed by the System Department. An initial document is prepared which contains information about why a new system is needed and outlines its major functions. This document is reviewed and, if approved, a senior analyst is assigned to prepare a more detailed document. The analyst collects data by interviewing users and studying the present system. He then produces a report describing his proposed system and showing how it will satisfy the requirements. The report will also contain the implementation plan, benefit/cost analysis, and his recommendations. The report is reviewed by the various organizational units involved. If it passes this review it is then included with other requests for the resources of the System Department and given a priority. Up to this point the investment in the proposed system is relatively small.

At some point a project team is formed, a project leader and team members are assigned, and given authority to proceed with the development of the system. A steering group may also be formed. The project is assigned a schedule in accordance with the project management system and given a budget. The schedule will include one or more target dates. The final target date will be the date the system (or its first part if it is being done in parts) is to be operational. There may also be additional target dates such as beginning of system test, beginning of programming, etc.

B. Logical system design documentation

In this paper, it is assumed that the system development procedure requires that the proposed system be reviewed before a major investment is made in system construction. There will therefore be another target date at which the "logical" design of the proposed system is reviewed. On the basis of this review the decision may be to proceed with the physical design and construction, to revise the proposed system, or to terminate the project.

The review is usually based on a document prepared by the project team. Sometimes it may consist of more than one separate document; for example, in the systems development methodology used by the U.S. Department of Defense [21] for non-weapons systems, development of the life cycle is divided into phases. Two documents are produced at the end of the Definition subphase of the Development phase: a Functional Description, and a Data Requirements Document.

Examination of these and many documentation requirements show that a Systems Definition Report contains five major types of information:

1) a description of the organization and where the proposed system will fit, showing how the proposed system will improve the functioning of the organization or otherwise meet the needs which lead to the project;

2) a description of the operation of the proposed system in sufficient detail to allow the users to verify that it will in fact accomplish its objectives, and to serve as the specification for the design and construction of the proposed system if the project continuation is authorized;

3) a description of its proposed system implementation in sufficient detail to estimate the time and cost required;

4) the implementation plan in sufficient detail to estimate the cost of the proposed system and the time it will be available;

5) a benefit/cost analysis and recommendations.

In addition, the report usually also contains other miscellaneous information such as glossaries, etc.

C. Current logical system design process

During the initial stages of the project the efforts of the team are directed towards producing the Systems Definition Report. Since the major item this report contains is the description of the proposed system from the user or logical point of view, the activities required to produce the report are called the logical system design process. The project team will start with the information already available and then perform a set of activities. These may be grouped into five major categories.

1) *Data collection.* Information about the information flow in the present system, user desires for new information, potential new system organization, etc., is collected and recorded.

2) *Analysis.* The data that have been collected are summarized and analyzed. Errors, omissions, and ambiguities are identified and corrected. Redundancies are identified. The results are prepared for review by appropriate groups.

3) *Logical design.* Functions to be performed by the system are selected. Alternatives for a new system or modification of the present system are developed and examined. The "new" system is described.

4) *Evaluation.* The benefits and costs of the proposed system are determined to a suitable level of accuracy. The operational and functional feasibility of the system are examined and evaluated.

5) *Improvements.* Usually as a result of the evaluation a number of deficiencies in the proposed system will be discovered. Alternatives for improvement are identified and evaluated until further possible improvements are not judged to be worth additional effort. If major changes are made, the evaluation step may be repeated; further data collection and analysis may also be necessary.

In practice the type of activities outlined above may not be clearly distinguished and may be carried out in parallel or iteratively with increasing level of detail. Throughout the process, however it is carried out, results are recorded and documented.

It is widely accepted that documentation is a weak link in system development in general and in logical system design in particular. The representation in the documentation that is produced with present manual methods is limited to:

1) text in a natural language;

2) lists, tables, arrays, cross references;

3) graphical representation, figures, flowcharts.

Analysis of two reports showed the following number of pages for each type of information.

Form	Report A	Report B
text	90	117
lists and tables	207	165
charts and figures	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].

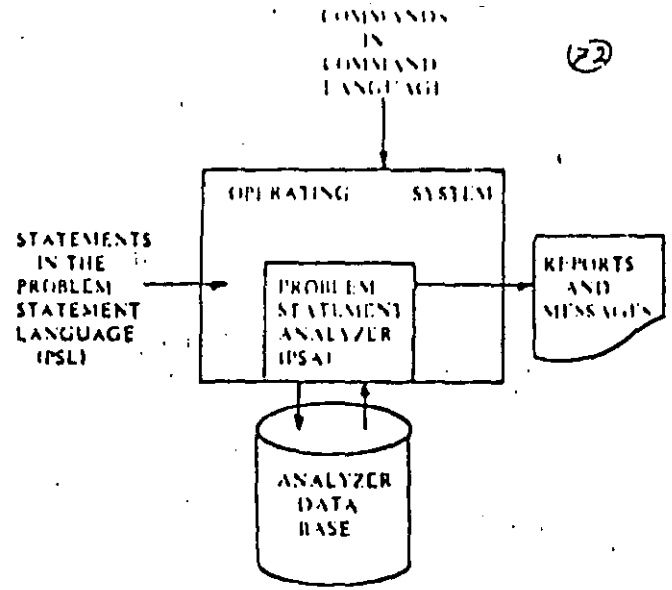


Figure 1. The problem statement analyzer.

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].

The Problem Statement Language is based first on a model of a general system, and secondly on the specialization of the model to a particular class of systems, namely information systems.

The model of a general system is relatively simple. It merely states that a system consists of things which are called OBJECTS. These objects may have PROPERTIES and each of these PROPERTIES may have PROPERTY VALUES. The objects may be connected or interrelated in various ways. These connections are called RELATIONSHIPS.

The general model is specialized for an information system by allowing the use of only a limited number of predefined objects, properties, and relationships.

B. *An overview of the problem statement language syntax and semantics*

The objective of PSL is to be able to express in syntactically analyzable form as much of the information which commonly appears in System Definition Reports as possible.

System Descriptions may be divided into eight major aspects:

1) System Input/Output Flow,
2) System Structure,
3) Data Structure,
4) Data Derivation,
5) System Size and Volume,
6) System Dynamics,
7) System Properties,
8) Project Management.

PSL contains a number of types of objects and relationships which permit these different aspects to be described.

The *System Input/Output Flow* aspect of the system deals with the interaction between the target system and its environment.

System Structure is concerned with the hierarchies among objects in a system. Structures may also be introduced to facilitate a particular design approach such as "top down." All information may initially be grouped together and called by one name at the highest level, and then successively subdivided. System structures can represent high-level hierarchies which may not actually exist in the system, as well as those that do.

The *Data Structure* aspect of system description includes all the relationships which exist among data used and/or manipulated by the system as seen by the "users" of the system.

The *Data Derivation* aspect of the system description specifies which data objects are involved in particular PROCESSES in the system. It is concerned with what information is used, updated, and/or derived, how this is done, and by which processes.

Data Derivation relationships are internal in the system, while System Input/Output Flow relationships describe the system boundaries. As with other PSL facilities System Input/Output Flow need not be used. A system can be considered as having no boundary.

The *System Size and Volume* aspect is concerned with the size of the system and those factors which influence the volume of processing which will be required.

The *System Dynamics* aspect of system description presents the manner in which the target system "behaves" over time.

All objects (of a particular type) used to describe the target system have characteristics which distinguish them from other objects of the same type. Therefore, the PROPERTIES of particular objects in the system must be described. The PROPERTIES themselves are objects and given unique names.

The *Project Management* aspect requires that, in addition to the description of the target system being designed, documentation of the project designing (or documenting) the target system be given. This involves identification of people involved and their responsibilities, schedules, etc.

IV. Reports

As information about a particular system is obtained, it is expressed in PSL and entered into a data base using the Problem Statement Analyzer. At any time standard outputs or reports may be produced on request. The various reports can be classified on the basis of the purposes which they serve.

1) *Data Base Modification Reports:* These constitute a record of changes that have been made, together with diagnostics and warnings. They constitute a record of changes for error correction and recovery.

2) *Reference Reports:* These present the information in the data base in various formats. For example, the Name List Report presents all the objects in the data base with their type and date of last change. The Formatted Problem Statement Report shows all properties and relationships for a particular object

(Fig. 2). . Dictionary Report gives only data dictionary type information.

```
Parameters: [illegible]

PROCESS                                          hourly-employee-processing;
    /* DATE OF LAST CHANGE - JUN 26, 1976, 13:36:46 */
    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.

Summary Reports: These present collections of information in summary from, or gathered from several different relationships. For example, the Data Base Summary Report provides project management information by showing the totals of various types of objects and how much has been said about them. The Structure Report shows complete or partial hierarchies. The Extended Picture Report shows the data flows in a graphical form.

Analysis Reports: These provide various types of analysis of the information in the data base. For example, the Contents Comparison Report analyzes similarity of Inputs and Outputs. The Data Process Interaction Report (Fig. 3) can be used to detect gap) the information flow, or unused data objects. The Process ain Report shows the dynamic behavior of the system (Fig. 4).

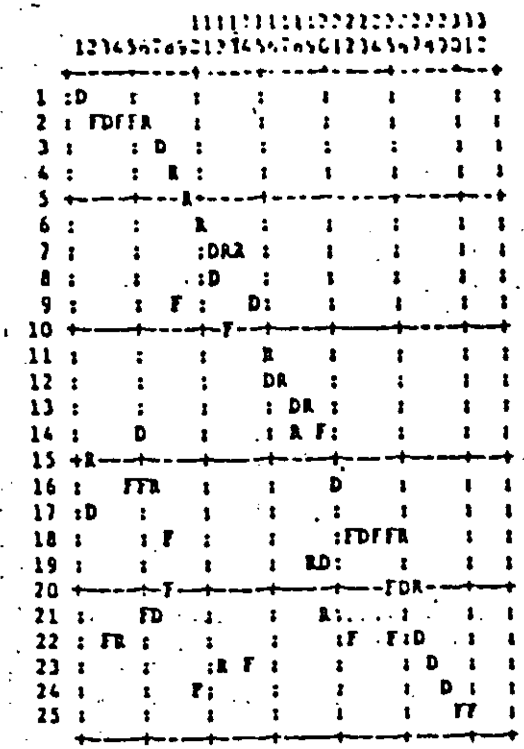


Figure 3. Example of part of a Data Process Interaction Report.

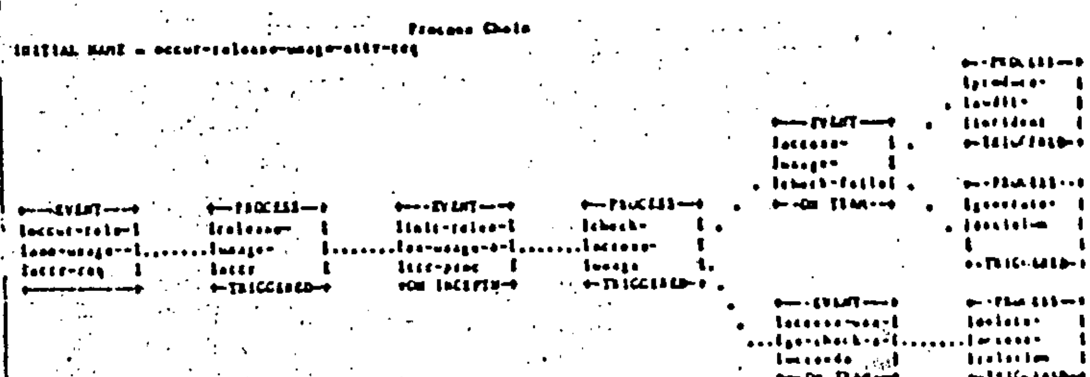


Figure 4. Example of a Process Chain Report.

After the requirements have been completed, the final documentation required by the organization can be produced semiautomatically to a prescribed format, e.g., the format required for the Functional Description and Data Requirements in [21].

(18)

V. Concluding remarks

The current status of PSL/PSA is described briefly in Section V-A. The benefits that should accrue to users of PSL/PSA are discussed in Section V-B. The information on benefits actually obtained by users is given in Section V-C. Planned extensions are outlined in Section V-D. Some conclusions reached as a result of the developments to date are given in Section V-E.

A. Current status

The PSL/PSA system described in this paper is operational on most larger computing environments which support interactive use, including IBM 370 series (OS/VS/TSO/CMS), Univac 1100 series (EXEC-8), CDC 6000/7000 series (SCOPE, TSS), Honeywell 600/6000 series (MULTICS, GCOS), AMDAHL 470/V8 (MTS), and PDP-10 (TOPS 10). Portability is achieved at a relatively high level; almost all of the system is written in ANSI Fortran.

PSL/PSA is currently being used by a number of organizations including AT&T Long Lines, Chase Manhattan Bank, Mobil Oil, British Railways, Petroleos Mexicanos, TRW Inc., the U.S. Air Force and others for documenting systems. It is also being used by academic institutions for education and research.

B. Benefit/cost analysis of computer-aided documentation

The major benefits claimed for computer-aided documentation are that the "quality" of the documentation is improved and that the cost of design, implementation, and maintenance will be reduced. The "quality" of the documentation, measured in terms of preciseness, consistency, and completeness is increased because the analysis must be more precise, the software performs checking, and the output reports can be reviewed for remaining ambiguities, inconsistencies, and omissions. While completeness can never be fully guaranteed, one important feature of the computer-aided method is that all the documentation that "exists" is the data base, and therefore the gaps and omissions are more obvious. Consequently, the organization knows what data it has, and does not have to depend on individuals who may not be available when a specific item of data about a system is needed. Any analysis performed and reports produced are up-to-date as of the time it is performed. The coordination among analysts is greatly simplified since each can work in his own area and still have the system specifications be consistent.

(21)

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.

1) The determination and documentation of requirements and functional specifications can be improved by making use of the computer for recording and analyzing the collected data and statements about the proposed system.

2) Computer-aided documentation is itself a system of which the software is only a part. If the system is to be used, adequate attention must be given to the whole methodology, including: user documentation, logistics and mechanics of use, training, methodological support, and management encouragement.

3) The basic structure of PSL and PSA is correct. A system description language should be of the relational type in which a description consists of identifying and naming objects and relationships among them. The software system should be data-base oriented, i.e., the data entry and modification procedures should be separated from the output report and analysis facilities.

4) The approach followed in the ISDOS project has succeeded in bringing PSL/PSA into operational use. The same approach can be applied to the rest of the system life cycle. A particularly important part of this approach is to concentrate first on the documentation and then on the methodology.

5) The decision to use a computer-aided documentation method is only partly influenced by the capabilities of the system. Much more important are factors relating to the organization itself and system development procedures. Therefore, even though computer-aided documentation is operational in some organizations, that does not mean that all organizations are ready to immediately adopt it as part of their system life cycle methodology.

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.)

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.

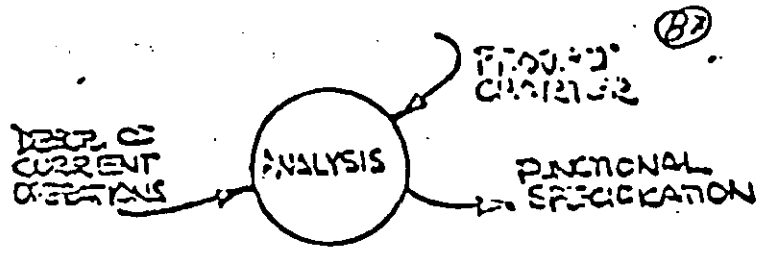
Structured Analysis and System Specification

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.

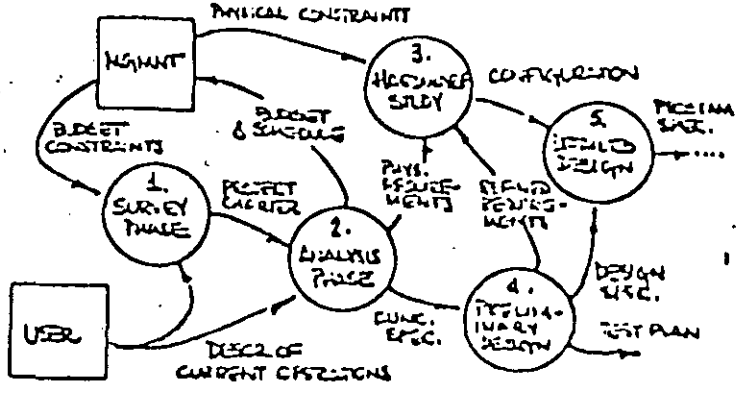
In this paper, I shall make a capsule presentation of the subject of Structured Analysis and its effect on the business of writing specifications of to-be-developed systems. I begin with a set of definitions:

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.



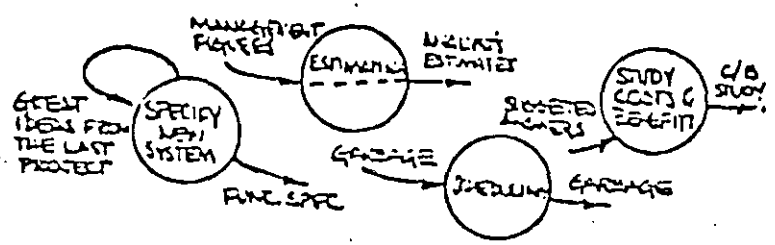
The Analysis Phase in context of the project life cycle.

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 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?

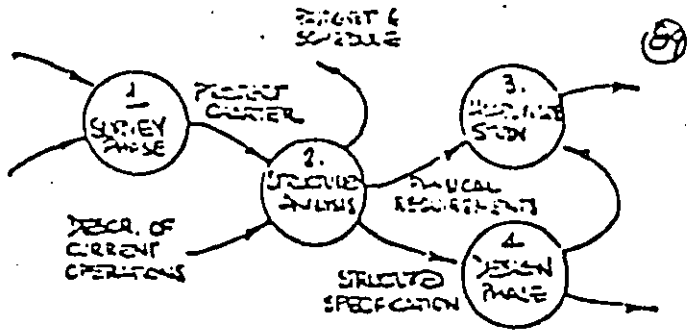


Figure 0: Structured Analysis in context of the project life cycle.

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.

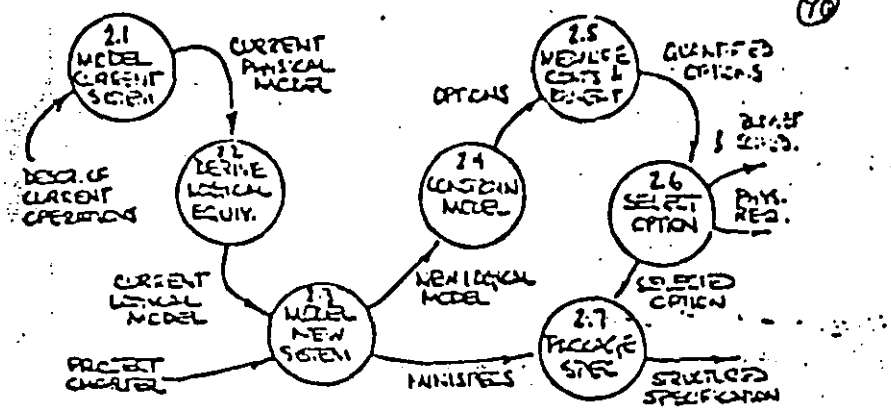


Figure 2: Details of Bubble 2, Structured Analysis.

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 Func-

tional Specifications over the past fifteen years, I have come to the conclusion that their viewpoint is, most often that of the computer. The classical specification describes what the computer does, in the order that it does it, using terms that are relevant to computers and computer people. It frequently limits itself to a discussion of processing inside the machine and data transferred in and out. Almost never does it specify anything that happens outside the man-machine boundary.

The machine's viewpoint is natural and useful for those whose concerns lie inside (the development staff), and totally foreign to those whose concerns lie outside (the user and his staff). Structured Analysis adopts a different viewpoint, that of the data. A Data Flow Diagram follows the data paths wherever they lead, through manual as well as automated procedure. A correctly drawn system model describes the outside as well as the inside of the automated portion. In fact, it describes the automated portion *in the context of the complete system.* The viewpoint of the data has two important advantages over that of any of the processors:

- The data goes everywhere, so its view is all-inclusive.
- The viewpoint of the data is common to those concerned with the inside as well as those concerned with the outside of the man-machine boundary.

The use of *limited redundancy* to create a highly maintainable product is not new. Any programmer worth his salt knows that a parameter that is likely to be changed ought to be defined in one place and one place only. What is new in Structured Analysis is the idea that the specification ought to be maintainable at all. After all, aren't we going to "freeze" it?. If we have learned anything over the past twenty years, it is that the concept of freezing specifications is one of the great pipe dreams of our profession. Change cannot be forestalled, only ignored. Ignoring change assures the building of a product that is out of date and unacceptable to the user. We may endeavor to hold off selected changes to avoid disruption of the development effort, but we can no longer tolerate being *obliged* to ignore change just because our specification is impossible to update. It is equally intolerable to accept the change, without updating the specification. The specification is our mechanism for keeping the project on target, tracking a moving goal. Failure to keep the specification up to date is like firing away at a moving target with your eyes closed. A key concept of Structured Analysis is development of a specification with little or no redundancy. This is a serious departure from the classical method that calls for specification of everything at least eleven times in eleven places. Reducing redundancy, as a by-product, makes the resultant specification considerably more concise.

6. What is a Structured Specification?

Structured Analysis is a discipline for conduct of the Analysis Phase. It includes procedures, techniques, documentation aids, logic and policy description tools, estimating heuristics, milestones, checkpoints and by-products. Some of these are important, and the rest merely convenient. What is most important is this simple idea: Structured Analysis involves building a new kind of specification, a Structured Specification, made up of Data Flow Diagrams, Data Dictionary and mini-specs.

The roles of the constituent parts of the Structured Specification are presented below:

The *Data Flow Diagrams* serve to partition the system. The system that is treated by the Data Flow Diagram may include manual as well as automated parts, but the same partitioning tool is used throughout. The purpose of the Data Flow Diagram is not to specify, but to declare. It declares component processes that make up the whole, and it declares interfaces among the components. Where the target system is large, several successive partitionings may be required. This is accomplished by lower-level Data Flow Diagrams of finer and finer detail. All the levels are combined into a leveled DFD set.

The *Data Dictionary* defines the interfaces that were declared on the Data Flow Diagrams. It does this with a notational convention that allows representation of dataflows and stores in terms of their components. (The components of a dataflow or store may be lower-level dataflows, or they may be data elements.) Before the Structured Specification can be called complete, there must be one definition in the Data Dictionary for each dataflow or data store declared on any of the Data Flow Diagrams.

The *mini-specs* define the elemental processes declared on the Data Flow Diagrams. A process is considered elemental (or "primitive") when it is not further decomposed into a lower-level Data Flow Diagram. Before the Structured Specification can be called complete, there must be one mini-spec for each primitive process declared on any of the Data Flow Diagrams.

The YOURDON Structured Analysis convention includes a set of rules and methods for writing mini-specs using Structured English, decision tables and trees and certain non-linguistic techniques for specification. For the purposes of this paper, such considerations are at the detail level — once you have partitioned to the point at which each of the mini-specs can be written in a page or less, it doesn't matter too terribly much how you write them.

7. What does it all mean?

Structured Analysis is here to stay. I estimate that its serious user community now includes more than 800 companies worldwide. Users as far away as Australia and Norway have published results of the application of Structured Analysis techniques to real-world projects [10, 11]. There are courses, texts and even a video-tape series [12] on the subject. There are automated support tools for use in Structured Analysis [13]. There are rival notations and symbologies [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 ideas have been used piecemeal for years. What is new is the emerging discipline. The advantages of this discipline are substantial. They include a much more methodical approach to specification, a more usable and maintainable product and fewer surprises when the new system is installed. These are especially attractive when compared to the advantages of the classical discipline for analysis. 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).

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

TEMA III

DISEÑO ESTRUCTURADO

ING. ALEJANDRO ACOSTA

MAYO, 1985

C o n t e n i d o

1.0 EL DISEÑO

En los últimos años, se ha prestado creciente atención a una nueva filosofía y a un grupo de técnicas para el desarrollo de Software. Técnicas como Programación Estructurada, Programación Modular, Diseño top-down, etc. se han utilizado frecuentemente con resultados satisfactorios en la mayoría de los casos.

Sin embargo, uno de los problemas que mayor confusión generan es el de terminología, como lo demuestra el uso del calificativo "Estructurado". Hay Programación Estructurada, Análisis Estructurado, Diseño Estructurado, Implantación Estructurada, Pruebas Estructuradas, etc., y, aunque se tiende a una mayor estandarización en el significado de la terminología, es necesario aclarar cuál es el significado que daremos al Diseño Estructurado.

> Diseño Estructurado es el proceso de transformar (decidir) lo que se tiene que hacer (el qué) en la manera de hacerlo (el cómo) para resolver un problema bien especificado.

Una definición más precisa:

> Diseño Estructurado es la elaboración (utilizando herramientas de modelado de sistemas) de una solución jerárquica del sistema, con los mismos componentes e interrelaciones que el problema que se intenta resolver.

Por tanto, el Diseño Estructurado es una actividad que se realiza después de decidir lo que el usuario desea, y antes de implementar estas necesidades en terminos de código.

1.1 Objetivos del Diseño Estructurado

Existen una serie de problemas comunes a la mayoría de los proyectos de desarrollo de software, los que a continuación se mencionan se deben a la falta de técnicas apropiadas para su desarrollo:

1. Difíciles de Administrar.

La principal consecuencia de este problema es que cuando se detecta un retraso o una desviación en lo presupuestado, es demasiado tarde para corregirlo. Además los soluciones convencionales aplicables a proyectos de otras áreas no son aplicables al desarrollo de Software, por ejemplo, añadir recursos humanos tiene un efecto que se puede predecir con la ley de Brooks (1):

> Adding manpower to a late Software project makes it later.

2. Poco Satisfactorio.

Comparado con otras disciplinas de Ingeniería, el desarrollo de Software es poco profesional, frecuentemente los sistemas terminados dejan pocas satisfacciones a usuarios y diseñadores y muchas frustraciones. Esto lleva a que el desarrollo de Software sea, en general, una actividad poco productiva.

3. Poco Confiable.

Es muy frecuente que una vez que el sistema se "libera", falle una y otra vez. Las fallas pueden no tener razones obvias o peor, se pueden producir resultados incorrectos que pasen inadvertidos.

4. Inflexible.

Cuando un sistema llega a trabajar se considera un milagro. Quién supone que se podrán realizar futuros cambios sin problema?

5 Difícil de Mantener.

Para modificar un sistema, se requiere:

- Entender cómo trabaja el sistema actual.
- Concebir el cambio
- Calcular las ramificaciones del cambio.

6. Ineficientes.

La idea generalizada de eficiencia está influenciada por la época en que el costo del procesador central era muy superior a los otros costos, de aquí que se considere eficientes a los programas que optimizan el uso del procesador central en lugar de aquellos que hacen uso eficiente de recursos escasos.

Con el Diseño Estructurado se pretende hacer sistemas cuyas características eviten los problemas antes mencionados (y otros no mencionados).

El objetivo del Diseño Estructurado es hacer Sistemas que sean:

- Utiles
- Mantenibles
- Modificables
- Flexibles
- Eficientes
- Generales

Todos estos elementos son componentes de un objetivo de calidad.

1.2 Filosofía General.

Podemos resumir los objetivos del Diseño Estructurado en uno solo: Producir sistemas económicamente convenientes (baratos).

Un diseño exitoso se basa en un principio conocido desde los días de Julio César

Divide y vencerás.

Veamos como este principio se aplica a los siguientes aspectos:

1. Implementación.

   El costo de implementar sistemas de cómputo será minimizado cuando las partes del problema sean:

   - suficientemente pequeñas
   - solucionables separadamente

2. Mantenimiento.

   De manera similar, el costo de mantenimiento será minimizado cuando las partes del sistema sean:

   - fácilmente relacionadas a la aplicación
   - suficientemente pequeñas
   - corregibles separadamente

3. Modificaciones.

   Finalmente, el costo de modificar un sistema será minimizado cuando sus partes sean:

   - fácilmente relacionadas al problema
   - modificables separadamente

En resumen, podemos establecer la siguiente filosofía:

> Implementación, Mantenimiento y Modificación serán generalmente minimizados cuando cada parte del sistema corresponda a exactamente una parte bien definida y pequeña del problema, y cada relación entre partes del sistema corresponda solo a relaciones entre partes del problema.

Esto significa que un buen diseño es un ejercicio consistente en dividir y organizar las partes de un sistema.

2.0 LAS IDEAS COMPLEMENTARIAS

Para alcanzar los objetivos del diseño estructurado, se requiere:

- Una herramienta de modelado para planear el sistema.
- Alguna manera para controlar la complejidad de sistemas no triviales.

2.1 El Modelo.

Un modelo es simplemente un cuerpo ordenado de hipótesis acerca de un sistema complejo; es un intento por entender algún aspecto de la infinita variedad de ellos que presenta el mundo, seleccionando a partir de percepciones y de experiencias pasadas, un cuerpo de observaciones generales aplicables al problema en cuestión.

Los modelos se presentan de varias maneras como son gráficas, ecuaciones, modelos a escala, maquetas, simuladores, etc. Los modelos son importantes en las disciplinas de ingeniería y su selección depende del problema en cuestión.

Para alcanzar los objetivos del Diseño Estructurado, requerimos de un modelo con las siguientes características:

- Gráfico.
- Divisible (top-down).
- Riguroso.
- Capaz de predecir el comportamiento del sistema.
- Que sea una consecuencia natural del Análisis Estructurado.
- Que sea una entrada natural para la Implementación Estructurada.
- Documentación básica del sistema.
- Ayuda para mantener y/o modificar el sistema.

2.2 El Control de Complejidad.

> The basic pattern of my approach will be to compose the program in minute steps, deciding each time as little as posible. As the problem analysis proceeds, so does the further refinement of my program. E. W. DIJKSTRA

La herramienta más útil para el control de complejidad en el diseño de sistemas es la Caja Negra.

Una Caja Negra es un mecanismo del cual se conoce:

- Sus entradas
- Sus salidas
- Lo que hace
- No se necesita saber cómo lo hace

Cuando los diseños se hacen con cajas negras pequeñas e independientes éstas son fácilmente entendidas, probadas, corregidas, mantenidas y modificadas.

El uso de la Caja Negra para controlar la complejidad del sistema, radica en dividir el sistema en Cajas Negras conectadas de tal forma que:

- Cada Caja Negra corresponda a una parte bien definida del problema.
- Cada Caja negra sea fácil de entender.
- Las conecciones entre Cajas Negras correspondan a conecciones del problema.
- Las conexiones entre Cajas Negras sean lo más simple posible, de tal manera que las Cajas Negras sean independientes entre sí.

2.3 Las Herramientas.

El Diseño Estructurado se apoya en el uso de dos herramientas:

- El Diagrama de Estructura.
- El Diagrama de Datos.

El Diagrama de Estructura muestra la división del sistema, su jerarquía y su organización.

El Diagrama de Datos muestra la división del sistema, sus flujos de datos y su organización.

Existen otras herramientas complementarias a las antes mencionadas como son las heurísticas de diseño, la morfología del sistema, criterios de transformación, etc., las cuales intervienen en el Diseño Estructurado; sin embargo la aplicación de estas herramientas complementarias requiere del Diagrama de Datos y del Diagrama de Estructura.

( ) La Metodología.

Es posible dividir el proceso de diseño en:

- Diseño general
- Diseño detallado

El diseño general consiste en decidir qué funciones, parámetros y relaciones son requeridas para el programa (o sistema). Diseño detallado es cómo implementar las funciones.

En términos muy generales, la metodología del Diseño Estructurado consiste en la aplicación sistemática de las herramientas de diseño de la siguiente forma:

- Hacer el Diagrama de Datos.
- Aplicando criterios de transformación, hacer el Diagrama de Estructura.
- Factorizar el Diagrama de Estructura.
- Analizar el Diagrama de Estructura utilizando criterios de:
  - Acoplamiento
  - Cohesión
  - Heurísticas de diseño
- Especificar cada módulo

3.0 EL DIAGRAMA DE ESTRUCTURA

El Diagrama de Estructura es una representación gráfica del sistema que se utiliza como herramienta para el diseño, implementación, documentación, modificación y mantenimiento del sistema. Es una herramienta, no un método.

El Diagrama de Estructura es un modelo independiente del tiempo de las relaciones jerárquicas de los módulos de un programa o sistema; es por esto que no se puede inferir de un Diagrama de Estructura, cuál es el orden en que se ejecutan los módulos.

Los elementos que forman el Diagrama de Estructura son los siguientes:

- Módulos (cuadros)
- Conexiones (flechas)
- Interfases (Nombres de datos que entran y salen de cada módulo)

Un Módulo es una secuencia de instrucciones continuas de programa confinadas por variables limítrofes, tienen un identificador.

Los Módulos tienen los siguientes atributos:

El qué:
- Entradas (lo que obtiene de su invocador)
- Salidas (lo que regresa a su invocador)
- Función (lo que hace a las entradas para producir las salidas)

El cómo:
- Mecánica (cómo hace su función)
- Datos internos (espacio privado de trabajo, solo él los referencia)

Nótese que no existe diferencia entre Módulo, Programa y Sistema.

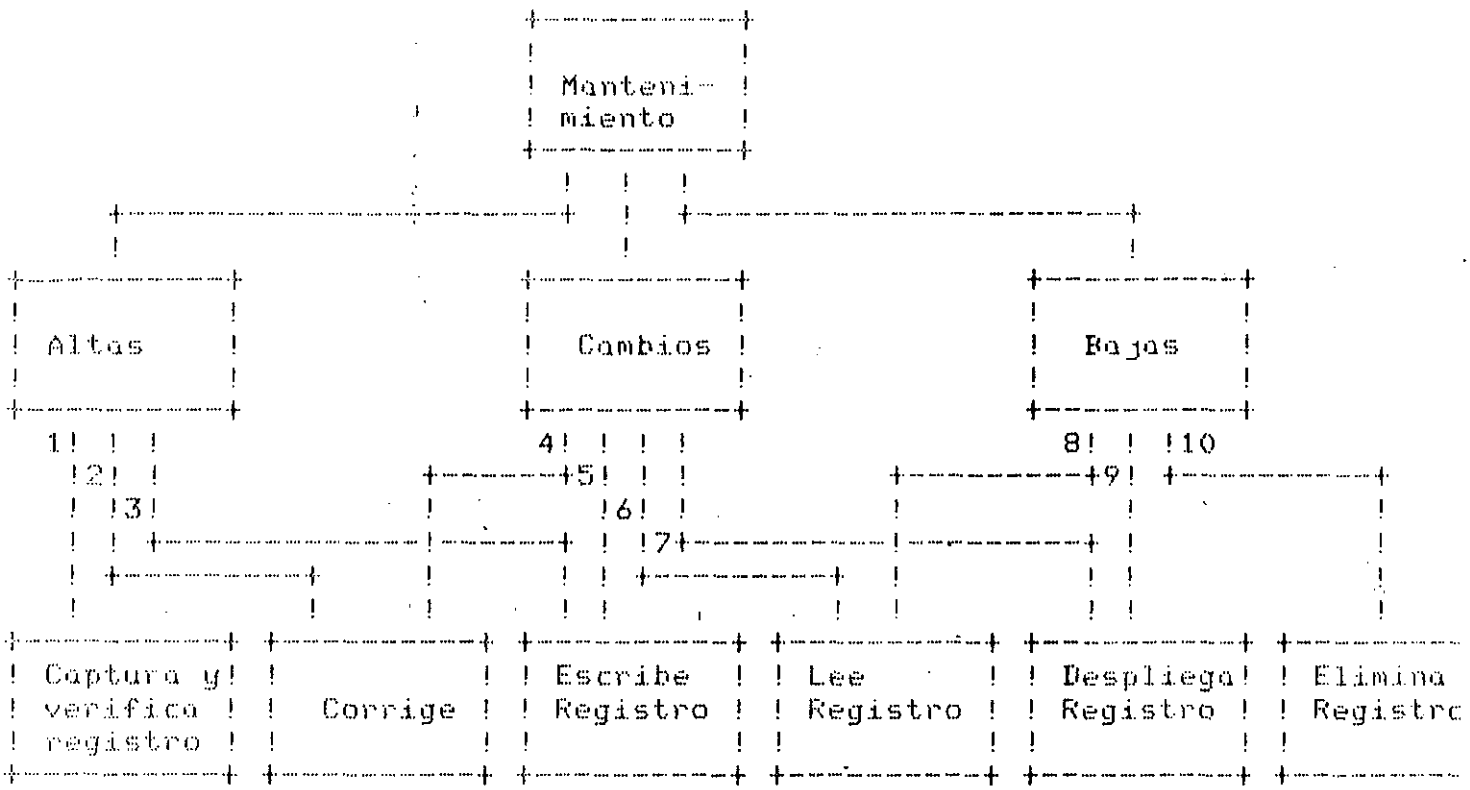
El Diagrama de Estructura muestra lo siguiente:

- La división del sistema en Módulos
- Jerarquía y organización de los Módulos
- Interfases de comunicación entre Módulos
- Nombres (funciones) de los Módulos

El Diagrama de Estructura no muestra:

- Mecanismos internos de los Módulos
- Datos internos de los Módulos

Ejemplo de Diagrama de Estructura:



	Entran (al subordinado)	Salen (del subordinado)
1		Registro
2	Registro	Registro
3	Registro	
4	Registro	Registro
5	Registro	
6		Registro
7	Registro	
8		Registro
9	Registro	
10	Registro	Resultado

4.0 ACOPLAMIENTO

Dos Módulos son totalmente independientes si cada uno puede funcionar completamente sin la presencia del otro. Esta definición implica que no hay interconecciones directas o indirectas, explicitas o implicitas, obvias u oscuras, entre los Módulos. Esto marca el punto cero en la escala de "dependencia" entre Módulos.

En general, entre más interconecciones existan entre Módulos, serán más dependientes entre sí.

El Acoplamiento es una medida de la interdependencia de un Módulo respecto a otro. Entonces, los Módulos altamente acoplados están unidos por interconecciones rigidas. Los Módulos holgadamente acoplados estan unidos por interconecciones débiles. Los Módulos no acoplados son aquellos que no tienen interconecciones.

El Acoplamiento es el criterio más importante para juzgar las bondades de un diseño.

4.1 Factores que intervienen en el Acoplamiento

Existen cuatro factores principales que pueden incrementar o decrementar el acoplamiento de módulos, son los siguientes:

1. Tipo de conexión entre módulos.

   Existen tres tipos de sistemas:

   - Conexiones mínimas.
   - Conexiones normales.
   - Conexiones patológicas.

2. Complejidad de la interfase.

   Se puede aproximar por el número de elementos pasados entre los módulos, entre más elementos haya, la interfase será más compleja.

3. Tipo de flujo de información a lo largo de la conexión.

   Los sistemas con acoplamiento de datos tienen menor acoplamiento que los de acoplamiento de control y éstos a su vez son mejores que los de acoplamiento híbrido.

4. Momento de unión de la conexión.

   Las conexiones unidas a referencias fijas al momento de ejecución, tienen menor acoplamiento que cuando la unión se efectúa al tiempo de carga o de compilación o de codificación.

El concepto de Acoplamiento invitó al desarrollo de un concepto nuevo: Deacoplamiento.

Deacoplamiento es cualquier técnica o método sistemático para volver a los módulos más independientes.

4.2 Tipos de acoplamiento

Entre menor sea el acoplamiento entre cualesquiera dos módulos, éstos serán más independientes y el diseño será mejor.

Existen cinco tipos de acoplamiento:

- Datos
- Estampado
- Control
- Area común
- Contenido

El mejor acoplamiento es el de datos y el peor el de contenido.

Si se presenta mas de un tipo de acoplamiento entre módulos, es el peor el que aplica.

4.2.1 Acoplamiento de Datos. -

Acoplamiento de datos es cuando solo los datos necesarios son comunicados entre módulos.

Este tipo de acoplamiento es el más deseable, y de hecho, cualquier sistema puede construirse de tal manera que el único acoplamiento sea de datos.

4.2.2 Acoplamiento de Estampado. -

Dos módulos presentan acoplamiento de estampado si referencian la misma estructura de datos (no global).

Una estructura de datos es un compuesto de elementos.

Este tipo de acoplamiento presenta el siguiente problema: un cambio en la estructura de datos afectará a todos los módulos que están "estampados" con la estructura;

sin embargo, usando una buena y natural estructura de datos, este tipo de acoplamiento se acerca al acoplamiento de datos.

4.2.3 Acoplamiento de Control. -

Dos módulos presentan acoplamiento de control si se comunican usando al menos un elemento de control.

Este acoplamiento es indeseable porque uno o ambos módulos no serán una caja negra.

Un peligro relacionado con el acoplamiento de control es el denominado "acoplamiento hibrido".

Acoplamiento hibrido sucede cuando se tienen dos o mas significados para el mismo dato.

4.2.4 Acoplamiento de Area Común. -

Un grupo de módulos presentan acoplamiento de área común si comparten una misma área global de datos.

Existen varios problemas con el acoplamiento de control:

1. Es más dificil reusar los módulos que utilizan áreas comunes ya que usualmente lo hacen por su nombre.

2. El mantenimiento se hace más dificil sobre todo cuando se pasan diferentes tipos de datos através del área común.

3. El uso de áreas globales dificulta la legibilidad de los programas.

4. Es difícil saber qué módulos usan qué datos.

5. En algunos lenguajes como FORTRAN, donde la posición de los datos en COMMON es importante, además del problema de acoplamiento de área común, se tiene un problema de acoplamiento de estampado.

6. Un error en cualquier módulo usando el área común puede aparecer en otro módulo (que tambien use el área común) ya que el área común no está protegida por ningún módulo.

4.2.5 Acoplamiento de Contenido. -

Este es el peor caso, ocurre cuando:

1. Un módulo altera instrucciones en otro módulo.

2. Un módulo referencia o cambia datos contenidos en otro módulo

3. Un módulo brinca a otro.

4. Dos módulos comparten las mismas literales.

Los casos 1, 2 y 3 también se conocen como Patológicos.

El problema principal es que un cambio pequeño y de apariencia inocente a uno de los módulos puede desquiciar a otro módulo en cualquier parte del sistema.

5.0 COHESION

Cohesión es una medida de la consistencia de la asociación de los elementos dentro de un módulo.

Un elemento es:

- Una instrucción
- Un grupo de instrucciones
- Una llamada a otro módulo

Es deseable tener módulos fuertes, altamente cohesivos, es decir, módulos cuyos elementos están altamente relacionados.

Claramente, cohesión y acoplamiento estan relacionados. A mayor cohesión de los módulos del sistema, existirá el menor acoplamiento.

5.1 Tipos de Cohesión.

La cohesión es una segunda medida de que tan bien dividimos el sistema. Existen los siguientes tipos:

- Funcional
- Secuencial
- Comunicacional
- De procedimiento
- Temporal
- Lógica
- Coincidental

Siendo mejor la cohesión funcional y peor la coincidental.

5.1.1 Cohesión Funcional: -

Un módulo con cohesión funcional es aquel en el que todos sus elementos contribuyen a una y solo una tarea completa; cada elemento es parte integral y es esencial para la ejecución de la función del módulo.

5.1.2 Cohesión Secuencial. -

Un módulo con cohesión secuencial es aquel en el que sus elementos están involucrados en tareas en las que datos que salen de un elemento sirven de entrada a otro elemento.

Los módulos con cohesión secuencial son fuertes, usualmente tienen buen acoplamiento. No son ideales porque contienen varias funciones que no forman una sola función completa.

5.1.3 Cohesión Comunicacional. -

Un módulo con cohesión comunicacional es aquel en el que sus elementos contribuyen a diferentes tareas, pero cada tarea se refiere a los mismos parámetros de entrada y/o salida.

Este tipo de cohesión es fuerte ya que el agrupamiento de funciones dentro de un módulo tienen alguna relación con el problema en lugar de la implementación de la solución.

Al dividir estos módulos en módulos separados, se simplifica el acoplamiento y se incrementa la cohesión.

5.1.4 Cohesión de Procedimiento. -

Un módulo con cohesión de procedimiento es aquel en el que el control fluye de un elemento al siguiente, pero, los datos no necesariamente fluyen de la misma manera.

El mayor problema con estos módulos es que manipulan resultados parciales, variables internas, banderas, switches, etc. y que tienen partes de varias funciones.

5.1.5 Cohesión Temporal. -

Los módulos con cohesión temporal son aquellos cuyos elementos están relacionados en tiempo.

Usualmente estos elementos realmente pertenecen a diferentes funciones.

5.1.6 Cohesión Lógica. -

Los módulos con cohesión lógica son aquellos en que sus elementos aparentan estar relacionados a tareas de la misma categoría general (debería llamarase cohesión ilógica).

5.1.7 Cohesión Coincidental. -

Un módulo con cohesión coincidental, tiene elementos sin relación significativa entre ellos. Usualmente ejecutan tareas diferentes y sin relación para diferentes "jefes".

6.0 ANALISIS DE TRANSFORMACION.

Análisis de transformación es un de las estrategias principales para diseñar sistemas altamente balanceados. Tambien se le conoce como Diseño de la Transformación central.

Para el Análisis de transformación se requiere del Diagrama de Datos. El resultado es un Diagrama de Estructura en el que el módulo superior trabaja con datos altamente procesados, datos lógicos.

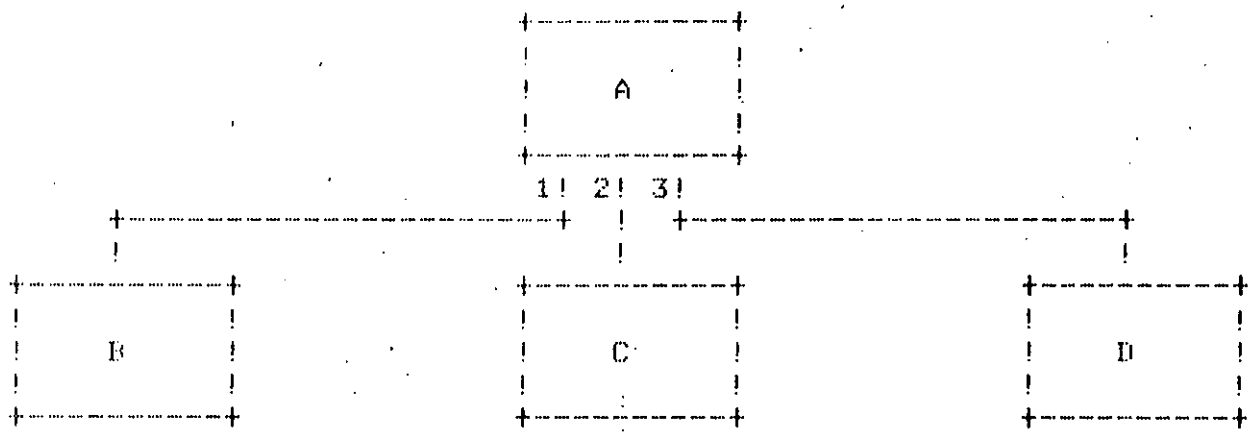
6.1 La estrategia.

1. Dibujar el diagrama de datos del problema.

2. Identificar la transformación central. Esto puede realizarse siguiendo las ramas aferentes y eferentes del diagrama hasta encontrar los puntos en que los datos son independientes de los dispositivos de entrada-salida. Los procesos entre estos puntos forman la transformación central.

3. Identificar las ramas principales de datos de entrada y salida. Determinar los puntos de mayor abstracción.

4. Diseñe la estructura a partir de la información previa con un módulo para cada rama principal de entrada y un modulo para cada rama principal de salida. En general se llegará a la estructura siguiente:



	Entran (al subordinado)	Salen (del subordinado)
1	Usualmente nada	Datos abstractos de entrada
2	Datos abstractos de entrada	Datos abstractos de salida
3	Datos abstractos de salida	Usualmente nada

5. Para cada uno de los módulos de entrada, identificar la última transformación necesaria para producir los datos en la forma en que los regresa el módulo. Después identifique la forma de la entrada justo antes de la última transformación. Para módulos de salida, identifique el primer proceso necesario para acercarse a la salida deseada con el formato deseado. Repita este paso hasta llegar a la forma original de los datos y el formato deseado de los resultados.

7.0 ANALISIS DE TRANSACCIONES

Una transacción es cualquier elemento de datos, control, señal, evento, o cambio de estado que causa, dispara o inicia alguna acción o secuencia de acciones.

En otras palabras, una transacción es una parte de datos que puede ser cualquiera de un número de tipos, cada tipo requiere diferente procesamiento.

La estrategia de análisis de transacciones, simplemente reconoce que los diagramas de datos de este tipo pueden mapearse a una estructura modular particular. El centro de transacciones de un sistema, debe ser capaz de:

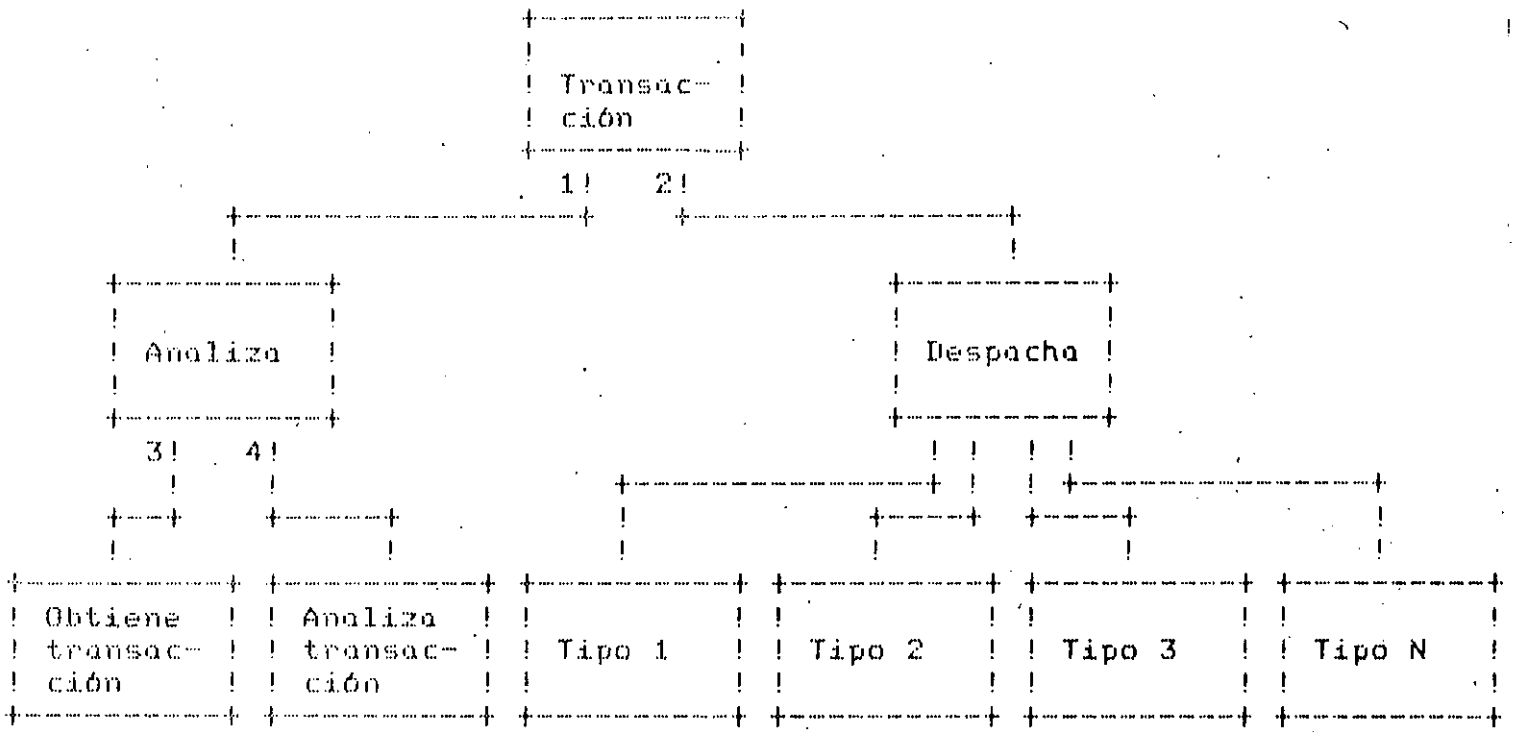
1. Obtener (responder a) las transacciones en su forma mas primitiva.

2. Analizar cada transaccion para determinar su tipo.

3. Despachar dependiendo de la transacción.

4. Completar el proceso de cada transacción.

En su forma más factorizada, el centro de transacciones puede ser modularizado de la siguiente forma:



	Entran (al subordinado)	Salen (del subordinado)
1		Transacción codificada internamente
2	Transacción codificada internament	
3		Transacción
4	Transacción	Transacción analizada y codificada

7.1 La estrategia.

1. Identificar los orígenes de transacciones.

2. Especificar la organización de transacciones apropiada (la figura anterior es un buen modelo en general).

3. Identificar la transaccion y las acciones que la definen.

4. Note situaciones poteciales en que se puedan combinar módulos.

5. Para cada transacción o grupo cohesivo de transacciones, especificar un módulo de transacciones para procesarla completamente.

6. Para cada acción en una transacción, especificar un módulo de acción subordinado al correspondiente módulo de transacción.

7. Para cada paso detallado en un módulo de acción, especificar un módulo detallado subordinado al módulo de acción que lo necesite.

8.0 CASOS PARTICULARES

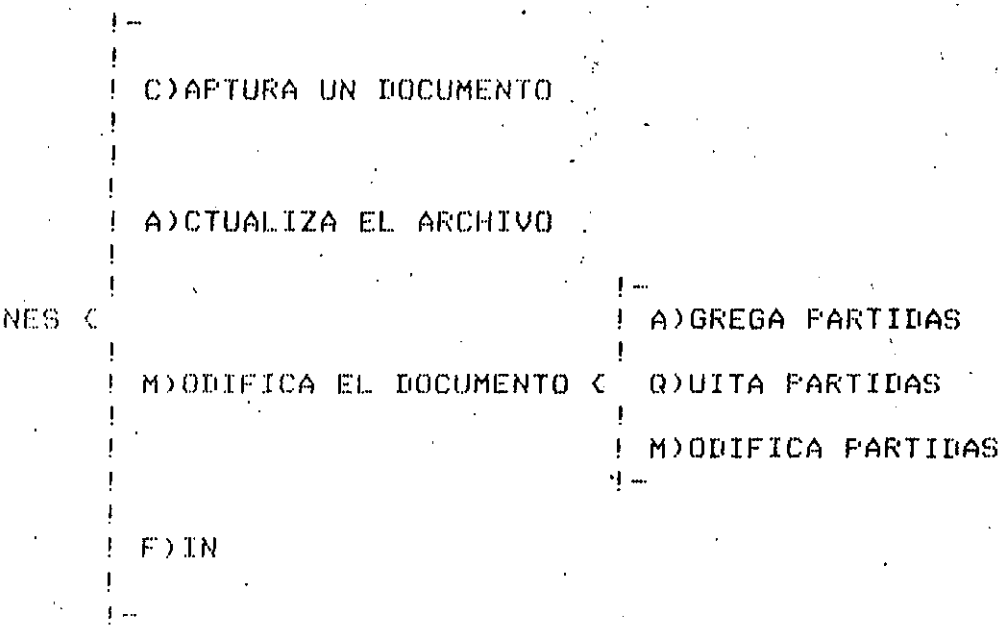
8.1 Programas interactivos

Un programa interactivo tiene las siguientes características:

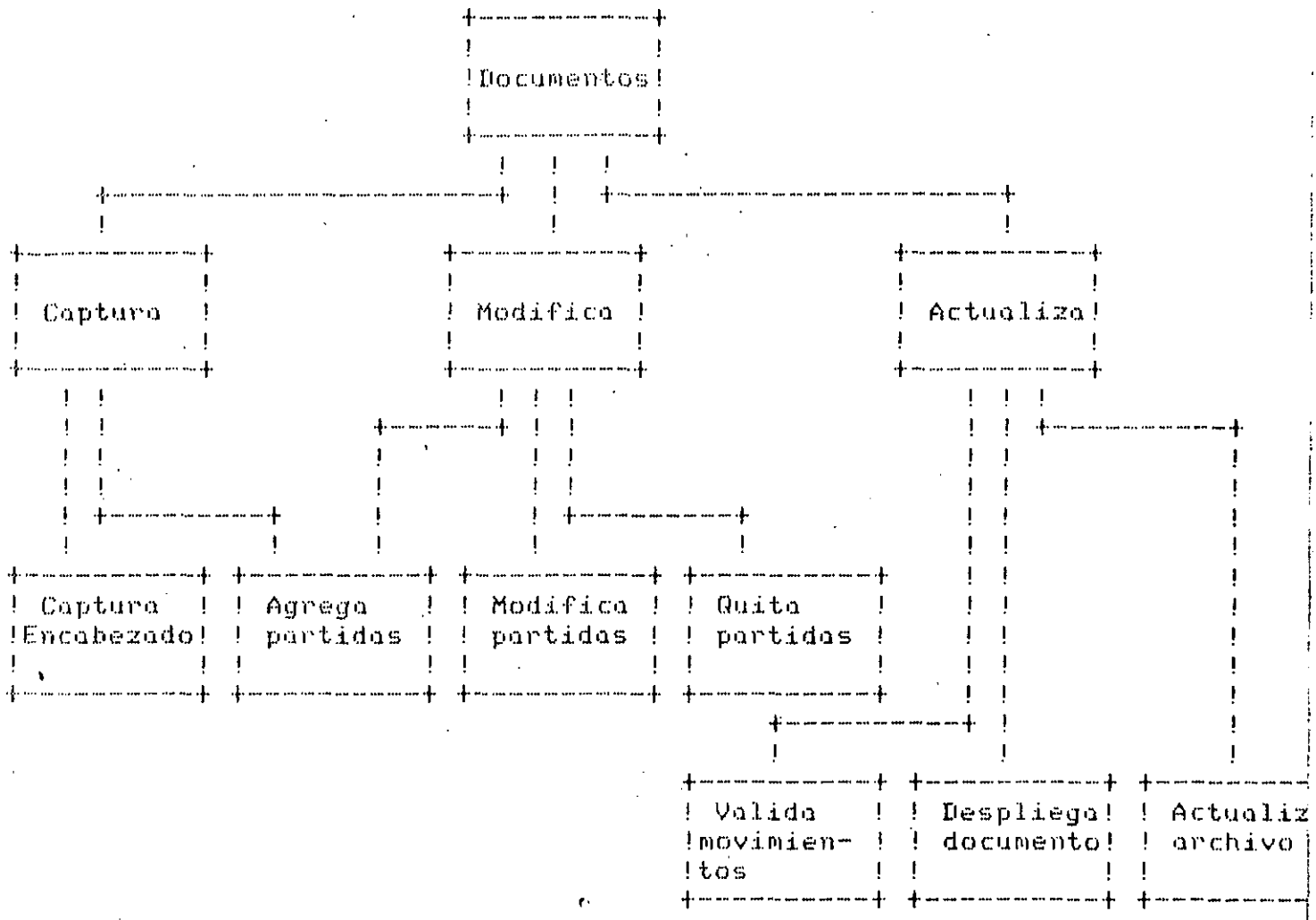
1. Conversacional.
2. Estructurado por tipo función.
3. Cada transacción es validada (consultando archivos) y actualizada instantaneamente.

Estos programas por naturaleza presentan una estructura arborea. Por tanto, la estrategia más apropiada para diseñarlos es a partir del árbol de funciones.

Ejemplo de árbol de funciones:



Ejemplo de estructura a partir del árbol de funciones.



8.2 Programas Batch.

Un programa batch tiene las siguientes características:

1. Controlado mediante comandos. Si el comando es incorrecto, se requiere realimentarlo hasta que sea correcto.

2. No es posible hacer correcciones a los errores de validación al momento de ejecutar el programa.

3. Tipicamente se utilizan en procesos periódicos.

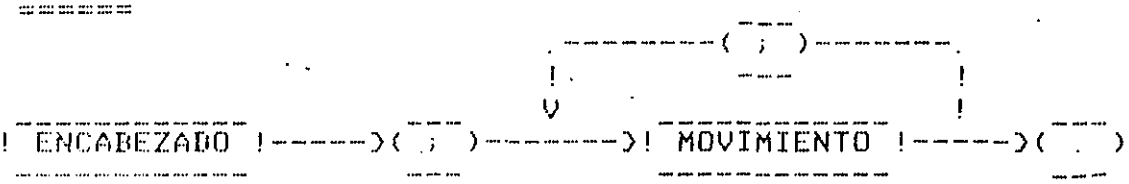
La manera más adecuada para diseñar estos programas es a partir de una definición sintáctica formal de los comandos y de la información que se va a procesar.

Una de las definiciones sintácticas más convenientes es la notación de Bäckus Naur (BNF) la cual se puede representar gráficamente con los diagramas de ferrocarril (estilo PASCAL).

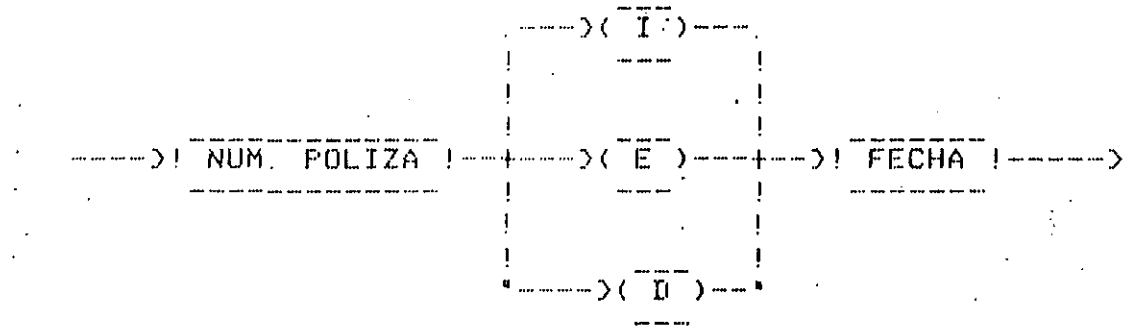
30

Ejemplo de diagrama de sintaxis:

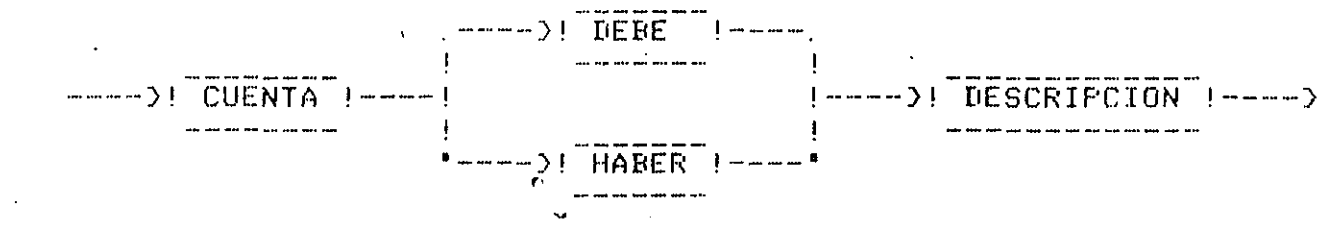
POLIZA:



ENCABEZADO:



MOVIMIENTO:



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

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

DISEÑO ESTRUCTURADO
(COMPLEMENTO)

MAYO, 1985

Palacio de Minería Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

NOTAS ELABORADAS POR : SALVADOR GOMEZ SAAB

* TEORIA ***

1.- DISEÑO ESTRUCTURADO Y SU IMPORTANCIA.

1.1.- QUE ES EL DISEÑO ESTRUCTURADO.
1.2.- ADQUISICION DE SISTEMAS DE COSTO MINIMO.
1.2.1.- FILOSOFIA GENERAL.
1.2.2.- IDEAS COMPLEMENTARIAS.
(MODELO Y CONTROL DE LA COMPLEJIDAD)

2.- ACOPLAMIENTO.

2.1.- QUE ES EL ACOPLAMIENTO.
2.2.- FACTORES QUE AFECTAN EL ACOPLAMIENTO.
2.3.- TIPOS DE ACOPLAMIENTO.
2.4.- DESACOPLAMIENTO.
2.5.- CUANTIFICACION DEL ACOPLAMIENTO.

3.- COHESION.

3.1.- QUE ES LA COHESION.
3.2.- NIVELES DE COHESION.
3.3.- CUANTIFICACION DE LA COHESION.

4.- PLAN DE DIAGNOSTICO.

4.1.- CARACTERISTICAS DE UN PROGRAMA.

++++ 1.- DISEÑO ESTRUCTURADO Y SU IMPORTANCIA ++++

*** 1.1- QUE ES EL DISEÑO ESTRUCTURADO ?***

DISEÑO ESTRUCTURADO ES EL ARTE DE DISEÑAR LOS COMPONENTES DE UN SISTEMA Y LA INTERRELACION ENTRE ESOS COMPONENTES EN LA MEJOR MANERA POSIBLE O

DISEÑO ESTRUCTURADO ES EL PROCESO DE DECIDIR CUALES COMPONENTES INTERCONECTADOS Y DE QUE MANERA , RESUELVEN ALGUN PROBLEMA BIEN DEFINIDO.

EL DISEÑO ESTRUCTURADO UNICAMENTE CONSOLIDA , FORMALIZA Y HACE VISIBLES ACTIVIDADES DE DISEÑO Y DECISIONES LAS CUALES OCURREN INEVITABLEMENTE E INVISIBLEMENTE EN EL CURSO DE CADA PROYECTO DE DESARROLLO DE SISTEMAS.

DISEÑO SIGNIFICA PLANIFICAR LA FORMA Y METODO DE ENCONTRAR UNA SOLUCION AL PROBLEMA. ES EL PROCESO EL QUE DETERMINA LAS PRINCIPALES CARACTERISTICAS DEL SISTEMA FINAL Y ESTABLECE COTAS SUPERIORES EN RENDIMIENTO Y CALIDAD QUE LA MEJOR IMPLEMENTACION PUEDE ADQUIRIR , Y PUEDE QUIZAS DETERMINAR EN CIERTO GRADO EL COSTO FINAL COMO VEREMOS ADELANTE Y QUE ES UNO DE LOS MOTIVOS DE ESTE ESCRITO.

PARA PODER DISEÑAR SISTEMAS DE COMPUTO NECESITAMOS PRIMERO QUE NADA , DEFINIR CLARAMENTE OBJETIVOS TECNICOS PARA PROGRAMAS COMO SISTEMAS .

DESAFORTUNADAMENTE , MUCHOS DISEÑADORES ESTAN EN UN NIVEL DONDE SI UN SISTEMA PARECE TRABAJAR, ES UN "BUEN PROGRAMA" (ESTO ES , YA SEA PASAR UNA MODESTA PRUEBA CON TIEMPOS LIMITES TOLERABLES O BIEN, SI HACE LO QUE DEBE , ENTONCES ES MAS QUE SUFICIENTE Y ESTE ES UN CRITERIO MUY PRACTICADO EN MUCHAS PARTES). AUN HOY MUCHOS DISEÑADORES CONFUNDEN COMO SI UNA SOLUCION, CUALQUIERA, FUESE LA SOLUCION.

UNA DE LAS METAS DEL DISEÑO ES "LA EFICIENCIA".

EN GENERAL QUEREMOS QUE SISTEMAS Y PROGRAMAS SEAN EFICIENTES EN EL SENTIDO QUE UTILICEN APROPIADAMENTE LOS RECURSOS DEL SISTEMA.

UNA DE LAS RAZONES IMPORTANTES POR LAS CUALES LOS SISTEMAS BUSCAN EFICIENCIA ES PORQUE LOS COSTOS DE SOFTWARE Y HARDWARE HAN VENIDO CAMBIANDO EN LOS ULTIMOS AÑOS Y VENDRAN HACIENDOLO AUN MAS.

CONFIABILIDAD ES OTRA MEDIDA DE CALIDAD , ESTE CONCEPTO FUE ENTENDIDO Y AHORA ES ESENCIAL VER QUE MIENTRAS LA CONFIABILIDAD DEL SOFTWARE PUEDE SER VISTO COMO UN PROBLEMA DE CORRECCION DE ERRORES, TAMBIEN PUEDE SER ---- Y MAS PRODUCTIVAMENTE QUIZAS ---- VISTO COMO UN PROBLEMA DE DISEÑO.

ESTE ULTIMO ENFOQUE HA CRECIDO EN POPULARIDAD.

CERCANO A LA CONFIABILIDAD EN SUS EFECTOS ESTA LA MANTENIBILIDAD.

DE HECHO PODEMOS EXPRESAR LA CONFIABILIDAD COMO :

" EL TIEMPO ENTRE FALLAS " ----------> (T E F)

Y LA MENTENIBILIDAD COMO :

" EL TIEMPO QUE PASA ENTRE REPARACIONES " ----> (T E R)

Y AHORA CON ESTO PODEMOS DEFINIR :
" DISPONIBILIDAD DEL SISTEMA " -----> (D S) = TEF/(TEF+TER)

SABEMOS QUE UN SISTEMA TENDRA A LO LARGO DE SU VIDA UN NUMERO DE FALLAS QUE PERMANECERAN (FALLAS RESIDUALES).

MIENTRAS QUE NOSOTROS ESPERAMOS QUE DESAPAREZCAN CON EL TIEMPO , LA DIFICULTAD DE CORREGIR , ANALIZAR Y REDISEÑAR AUMENTA DEBIDO A UNA VARIEDAD DE EFECTOS.

LA MANTENIBILIDAD AFECTA LA VIABILIDAD DEL SISTEMA EN UNA FORMA RELATIVAMENTE CONSTANTE.

LA MODIFICABILIDAD AFECTA EL COSTO DE CONSERVAR EL SISTEMA VIABLE EN LA FASE DE CAMBIAR LOS REQUERIMIENTOS.

LA FLEXIBILIDAD REPRESENTA LA FACILIDAD DE HACER VARIACIONES EN UN SISTEMA.

EN TERMINOS SIMPLES ES SUFICIENTE ASEVERAR QUE NUESTRO PRINCIPAL OBJETIVO SON LOS "SISTEMAS DE COSTO MINIMO".

ESTO ES , ESTAMOS INTERESADOS EN SISTEMAS QUE SON FACILES DE DESARROLLAR , OPERAR , MANTENER Y MODIFICAR .

++++ 1.2.- ADQUISICION D. SISTEMAS DEL COSTO MINIMO ++++

*** 1.2.1.- FILOSOFIA GENERAL. ***

SUGERIMOS QUE LOS COSTOS DE IMPLEMENTACION DE UN SISTEMA SI COMPUTO SERAN MINIMIZADOS CUANDO LAS PARTES DEL PROBLEMA ESTEN :

A) MANEJABLEMENTE PEQUE?AS
B) SEPARABLEMENTE RESOLVIBLES

DE MANERA SIMILAR ARGUIMOS QUE LOS COSTOS DE MANTENIMIENTO SON MINIMIZADOS CUANDO LAS PARTES DEL SISTEMA CUMPLEN :

A) LAS PARTES DEL SISTEMA SE RELACIONEN FACILMENTE CON LA APLICACION
B) MANEJABLEMENTE PEQUE?AS
C) CORRECTAMENTE SEPARABLES

UN EJEMPLO SERIA, SI TENEMOS UN ERROR EN ALGUN PROGRAMA, ES MUY IMPORTANTE QUE ESTE TENGA PARTES MANEJABLEMENTE PEQUE?AS O MODULOS, SERIA MAS DIFICIL ENCONTRAR UN ERROR Y COMPONERLO EN UN MODULO O PROGRAMA DE 1000 LINEAS QUE EN UNO DE 20, EN TODO CASO SI NO SE ENCUENTRA EN ESTE ULTIMO, SE DESECHA EL MODULO Y SE ESCRIBE OTRO

FINALMENTE SUGERIMOS QUE EL COSTO DE LA MODIFICACION DE UN SISTEMA SERA MINIMIZADO CUANDO SUS PARTES SEAN :

A) FACILMENTE RELACIONADAS AL PROBLEMA
B) SEPARABLEMENTE MODIFICABLES

EN RESUMEN , PODEMOS ESTABLECER LA SIGUIENTE FILOSOFIA: IMPLEMENTACION , MANTENIMIENTO Y MODIFICACION GENERALMENTE SERAN MINIMIZADOS CUANDO CADA PIEZA DEL SISTEMA CORRESPONDA EXACTAMENTE A UNA PEQUE?A Y BIEN DEFINIDA PIEZA DEL PROBLEMA Y CADA RELACION ENTRE LAS PIEZAS DE UN SISTEMA CORRESPONDA UNICAMENTE A UNA RELACION ENTRE LAS PIEZAS DEL PROBLEMA.

ESTO SIGNIFICA QUE UN BUEN DISE?O ES UN EJERCICIO EN PARTICIONAR Y ORGANIZAR LAS PIEZAS DE UN SISTEMA.

DE AQUI SURGE LA SIGUIENTE CUESTION :

DONDE Y COMO SE DEBE DIVIDIR EL PROBLEMA ?

EL DISE?O ESTRUCTURADO RESPONDE ESTAS CUESTIONES CON 2 PRINCIPIOS BASICOS :

A) PARTES ALTAMENTE RELACIONADAS DEL PROBLEMA, DEBERIAN ESTAR EN LA MISMA PARTE DEL SISTEMA
B) PARTES NO RELACIONADAS ENTRE ELLAS DEBEN ESTAR POR SEPARADO

*** 1.2.2.- IDEAS COMPLEMENTARIAS . ***

PARA ALCANZAR LOS OBJETIVOS DEL DISEÑO ESTRUCTURADO REQUERIMOS BASICAMENTE DE UNA HERRAMIENTA DE MODELADO PARA DISEÑAR EL SISTEMA Y OTRA HERRAMIENTA PARA CONTROLAR LA COMPLEJIDAD DE SISTEMAS NO TRIVIALES

EL MODELO :

UN MODELO PARA DISEÑAR UN SISTEMA CONSISTE DE HIPOTESIS ACERCA DEL MISMO. ESTO VIENE A SER UN ESFUERZO POR ENTENDER ALGUNOS ASPECTOS DE LOS MUCHOS QUE PUEDE PRESENTAR.

MUCHAS DE ESTAS HIPOTESIS SE ELABORAN A PARTIR DE PERCEPCIONES Y EXPERIENCIAS PASADAS APLICABLES AL PROBLEMA EN CUESTION.

SE PUEDE DECIR QUE PARA ALCANZAR LOS OBJETIVOS DE UN DISEÑO ESTRUCTURADO SOLICITAMOS DE UN MODELO LO SIGUIENTE :

QUE SEA :
1) GRAFICO
2) DIVISIBLE
3) RIGUROSO
4) CAPAZ DE PREDECIR EL COMPORTAMIENTO DEL SISTEMA
5) QUE SEA CONSECUENCIA DE UN ANALISIS ESTRUCTURADO
6) DOCUMENTACION BASICA DEL SISTEMA
7) AYUDA PARA MANTENER Y/O MODIFICAR EL SISTEMA

CONTROL DE LA COMPLEJIDAD :

LA HERRAMIENTA MAS UTIL PARA PODER CONTROLAR LA COMPLEJIDAD EN EL DISEÑO DE SISTEMAS ES LA CAJA NEGRA.

ESTA CAJA NEGRA ES UN MECANISMO DEL CUAL SE CONOCE :

A) SUS ENTRADAS Y SALIDAS
B) LO QUE HACE Y QUE ADEMAS :
C) NO SE NECESITE SABER COMO LO HACE

CUANDO EN LOS DISEÑOS SE UTILIZAN CAJAS NEGRAS PEQUEÑAS E INDEPENDIENTES ESTOS SERAN MEJOR ENTENDIDOS , PROBADOS , MANTENIDOS , CORREGIDOS O MODIFICADOS.

LA UTILIZACION DE ESTO ES PARA CONTROLAR LA COMPLEJIDAD DEL SISTEMA Y RADICA EN DIVIDIR EL MISMO EN CAJAS NEGRAS CONECTADAS DE TAL FORMA QUE :

- CADA CAJA NEGRA SEA FACIL DE ENTENDER.
- LAS CONEXIONES ENTRE CAJAS CORRESPONDAN A CONEXIONES DEL PROBLEMA
- ESTAS CONEXIONES ENTRE CAJAS NEGRAS SEAN LO MAS SIMPLE POSIBLE, DE TAL MANERA QUE LAS CAJAS NEGRAS SEAN INDEPENDIENTES ENTRE SI

++++ 2.- ACOPLAMIENTO. ++++

*** 2.1.- QUE ES EL ACOPLAMIENTO. ***

MUCHOS ASPECTOS DE LA MODULARIDAD PUEDEN SER ENTENDIDOS EXAMINANDOLOS EN RELACION A OTRO.

DOS MODULOS SON TOTALMENTE INDEPENDIENTES SI CADA UNO PUEDE FUNCIONAR COMPLETAMENTE SIN LA PRESENCIA DEL OTRO . ESTA DEFINICION IMPLICA QUE NO HAY INTERCONEXIONES ENTRE LOS MODULOS

EN GENERAL ENTRE MAS INTERCONEXIONES HAYA , PARECE QUE HABRA MENOS INDEPENDENCIA.

MIENTRAS MAS SEPAMOS DE UN MODULO PARA ENTENDER OTRO ENTONCES MAS CERCANAMENTE ESTARAN INTERCONECTADOS ESTOS.

DESAFORTUNADAMENTE, LA FRASE "CONOCIMIENTO REQUERIDO PARA ENTENDER UN MODULO" NO ES MUY OBJETIVO ; NECESITAMOS UN METODO OPERACIONAL DE APROXIMAR EL GRADO DE INTERCONEXION.

UN SIMPLE CONTEO DEL NUMERO Y VARIEDAD DE INTERCONEXIONES ENTRE MODULOS ES INSUFICIENTE PARA PODER CARACTERIZAR COMPLETAMENTE LA INFLUENCIA DE LAS INTERCONEXIONES EN LA MODULARIDAD DEL SISTEMA.

LA MEDIDA QUE BUSCAMOS ES CONOCIDA COMO ACOPLAMIENTO.

ES UNA MEDIDA DE ESTRECHEZ EN LA INTERCONEXION.

MODULOS "ALTAMENTE ACOPLADOS" SON UNIDOS POR INTERCONEXIONES MUY FUERTES ; MODULOS "POCO ACOPLADOS" ESTAN UNIDOS POR UNA DEBIL INTERCONEXION.

MODULOS "DESACOPLADOS" NO TIENEN INTERCONEXIONES Y SON INDEPENDIENTES.

DIREMOS QUE EL ACOPLAMIENTO COMO UN CONCEPTO ABSTRACTO CONSISTE EN QUE EL GRADO DE INTERDEPENDENCIA ENTRE MODULOS PUEDE SER OPERACIONALIZADO COMO LA PROBABILIDAD DE QUE EN LA CODIFICACION , MODIFICACION O CORRECCION DE UN MODULO, UN PROGRAMADOR TENDRA QUE TOMAR EN CUENTA ALGO ACERCA DE OTRO MODULO .

CLARAMENTE EL COSTO TOTAL DE LOS SISTEMAS SERA FUERTEMENTE INFLUENCIADO POR EL GRADO DE ACOPLAMIENTO ENTRE MODULOS .

PODEMOS DECIR QUE EL ACOPLAMIENTO ES EL CRITERIO MAS IMPORTANTE PARA JUZGAR LAS BONDADES DE UN DISEÑO.

*** 2.2.- FACTORES QUE AFECTAN EL ACOPLAMIENTO . ***

CUATRO ASPECTOS PRINCIPALMENTE PUEDEN INCREMENTAR O DECREMENTAR EL ACOPLAMIENTO INTERMODULAR . EN ORDEN DE ESTIMAR LA MAGNITUD DE SU EFECTO EN EL ACOPLAMIENTO , ESTOS SON :

1) TIPO DE CONEXION ENTRE MODULOS .

SISTEMAS MINIMAMENTE CONECTADOS TIENEN EL MAS BAJO ACOPLAMIENTO.

2) COMPLEJIDAD DE LA INTERFASE .

ESTO ES APROXIMADAMENTE IGUAL AL NUMERO DE COSAS DIFERENTES QUE SON PASADAS (NO LA CANTIDAD DE DATOS) - MIENTRAS MAS PARAMETROS , MAYOR EL ACOPLAMIENTO .

3) TIPO DE INFORMACION QUE FLUYE A LO LARGO DE LA CONEXION.

4) TIEMPO DE UNION EN LA CONEXION.

VEAMOSLOS UNO POR UNO.

1) TIPO DE CONEXION ENTRE MODULOS :

UNA CONEXION EN UN PROGRAMA ES UNA REFERENCIA POR UN ELEMENTO AL NOMBRE , IDENTIFICADOR ETC DE OTRO ELEMENTO.

UNA CONEXION INTERMODULAR OCURRE CUANDO EL ELEMENTO REFERENCIADO ESTA EN UN MODULO DIFERENTE QUE EL QUE LO LLAMO O REFERENCIO.

CLARAMENTE QUEREMOS MINIMIZAR LA COMPLEJIDAD DE LOS MODULOS DE UN SISTEMA, EN PARTE MINIMIZANDO EL NUMERO DE (Y VARIEDAD) DE INTERFASES POR MODULO.

YA SABEMOS QUE CADA MODULO AL MENOS DEBE TENER UNA INTERFASE QUE LO DISTINGA Y QUE LO ATE AL SISTEMA.

SOLAMENTE CONTROL Y DATOS PUEDEN SER PASADOS ENTRE MODULOS EN UN SISTEMA DE PROGRAMACION. UNA INTERFASE PUEDE SERVIR PARA TRANSMITIR DATOS A UN MODULO (DE ENTRADA O DE SALIDA)

PUEDE SER UN NOMBRE POR EL CUAL EL CONTROL SEA RECIBIDO O TRANSMITIDO .

UN SISTEMA ES LLAMADO MINIMAMENTE CONEXO SI TODAS SUS CONEXIONES ESTAN RESTRINGIDAS TOTALMENTE A PARAMETROS CONDICIONADOS A TRANSFERIR CONTROL O RECIBIRLO.

DECIMOS QUE UN MODULO ES NORMALMENTE CONEXO SI NINGUNA DE LAS SIGUIENTES CONDICIONES SE CUMPLE :

A) SI HAY MAS DE UN PUNTO DE ENTRADA A UN SOLO MODULO.

B) SI EL CONTROL REGRESA A OTRA INSTRUCCION QUE NO SEA LA QUE SIGA AL MODULO QUE LO ACTIVO

C) SI EL CONTROL ES TRANSFERIDO POR OTRA RAZON QUE LA NORMAL A TRANSFERIRLO .

EL USO MULTIPLE DE PUNTOS DE ENTRADA A UN MODULO GARANTIZARA QUE HABRA MAS DEL MINIMO NUMERO DE INTERCONEXIONES AL SISTEMA.

SI UN SISTEMA NO ES MINIMA NI NORMALMENTE CONEXO ENTONCES ALGUNO DE SUS MODULOS DEBE TENER CONEXIONES A LAS QUE LLAMAMOS PATOLOGICAS.

CON ESTO QUIERO DECIR QUE AL MENOS UN MODULO HACE TRANSFERENCIA DE CONTROL INCONDICIONAL A OTROS MODULOS O HACE REFERENCIA EXPLICITA A DATOS FUERA DE SUS COTAS O LIMITES.

TODAS ESTAS SITUACIONES INCREMENTAN EL ACOPLAMIENTO DEL

SISTEMA INCREMENTANDO LA CANTIDAD DE LO QUE DEBEMOS SABER DEL "MUNDO DE AFUERA" PARA PODER ENTENDER COMO TRABAJA NUESTRO MODULO SE DEB. TRATAR YA POR ULTIMO DE MINIMIZAR EL ACOPLAMIENTO.

2) COMPLEJIDAD DE LA INTERFASE .

LA SEGUNDA DIMENSION DEL ACOPLAMIENTO ES LA COMPLEJIDAD MIENTRAS MAS COMPLEJA SEA UNA CONEXION , MAYOR ES EL A ACOPLAMIENTO.

POR COMPLEJIDAD NOS REFERIMOS A COMPLEJIDAD EN TERMINOS HUMANOS.

HAY VARIAS MANERAS DE APROXIMAR LA COMPLEJIDAD DE UNA INTERFASE AUNQUE NINGUNA ES PERFECTA . POR ESTO QUIERO DECIR, ENTRE MAS ELEMENTOS SE PASEN LOS MODULOS MAS COMPLEJA SERA LA INTERFASE.

3) FLUJO DE INFORMACION.

OTRO ASPECTO IMPORTANTE DEL ACOPLAMIENTO TIENE QUE VER CON EL TIPO DE INFORMACION QUE SE TRANSMITE ENTRE EL SUPERORDINADO Y EL SUBORDINADO.

LAS CLASES DE INFORMACION QUE DISTINGUIMOS SON : DATOS CONTROL Y UNA ESPECIE HIBRIDA DE DATOS Y CONTROL.

LA INFORMACION QUE CONSTITUYEN LOS DATOS ES AQUELLA QUE ES OPERADA , MANIPULADA O CAMBIADA POR UN PEDAZO DE PROGRAMA.

INFORMACION DE CONTROL (AUN CUANDO SEA REPRESENTADA POR DATOS) ES AQUELLA LA CUAL GOBIERNA COMO LAS OPERACIONES O MANIPULACIONES U OTRA INFORMACION SE LLEVARA A CABO.

SE PUEDE MOSTRAR QUE LA COMUNICACION DE DATOS SOLA ES NECESARIA PARA EL FUNCIONAMIENTO DE SISTEMAS MODULARES.

LA COMUNICACION DE CONTROL NO ES INDISPENSABLE , AUNQUE ES HALLADA EN CASI TODOS LOS SISTEMAS .

EL ACOPLAMIENTO ES MINIMIZADO CUANDO SOLO FLUYEN DATOS TANTO DE ENTRADA COMO DE SALIDA EN LOS MODULOS.

EL ACOPLAMIENTO POR CONTROL CUBRE TODAS LAS FORMAS DE CONEXION QUE COMUNICAN ELEMENTOS DE CONTROL.

UN CONTROL INDIRECTO SE LLAMARA COORDINACION . COORDINACION INVOLUCRA UN MODULO DENTRO DE OTRO.

CUANDO UN MODULO MODIFICA EL CONTENIDO DE OTRO TENEMOS UN ACOPLAMIENTO HIBRIDO.

EL GRADO DE INTERDEPENDENCIA ASOCIADO CON UN ACOPLAMIENTO HIBRIDO ES CLARAMENTE MUY FUERTE, YA QUE CADA FUNCION DEL MODULO A MODIFICAR PUEDE SER CAMBIADA. RESUMIENDO LO ANTERIOR PODEMOS ESTABLECER QUE LO SISTEMAS CON ACOPLAMIENTOS DE DATOS TIENEN MENOR ACOPLAMIENTO QUE LOS DE ACOPLAMIENTO DE CONTROL Y ESTOS A SU VEZ MENOR QUE LOS HIBRIDOS.

4) TIEMPO DE LA UNION DE CONEXIONES INTERMODULARES .

"BINDING" , PEGANDO , MEZCLANDO O UNIENDO ES UN TERMINO COMUNMENTE USADO EN EL CAMPO DE PROCESAMIENTO DE DATOS PARA REFERIRSE AL PROCESO DE RESOLVER O REPARAR LOS VALORES DE IDENTIFICADORES DENTRO DE UN SISTEMA .

LA UNION DE VARIABLES PUEDE TOMAR PARTE EN CUALQUIER NIVEL O PERIODO DE TIEMPO EN LA EVOLUCION DEL SISTEMA .

LA HISTORIA DE TIEMPO DE UN SISTEMA PUEDE SER PENSADO COMO UNA LINEA QUE SE EXTIENDE DEL MOMENTO DE ESCRIBIR EL PROGRAMA , HASTA SU EJECUCION . ESTA LINEA PUEDE SER SUBDIVIDIDA A MAYOR O MENOR GRADO DE REFINAMIENTO POR DIFERENTES COMBINACIONES DE SISTEMAS DE COMPUTO/LENGUAJE/COMPILADORES/OPERATIVOS .

POR LO TANTO EL PEGADO O UNION PUEDE LLEVARSE A CABO CUANDO EL PROGRAMADOR ESCRIBE UN PEDAZO DE CODIGO , CUANDO UN MODULO ES COMPILADO O ENSAMBLADO , CUANDO SU IMAGEN DE CODIGO ES CARGADA A MEMORIA Y FINALMENTE CUANDO EL SISTEMA EMPIEZA A CORRER.

LAS CONEXIONES UNIDAS A REFERENCIAS FIJAS AL MOMENTO DE EJECUCION TIENEN MENOR ACOPLAMIENTO QUE CUANDO LA UNION SE EFECTUA AL TIEMPO DE CARGA , COMPILACION O CODIFICACION .

PODEMOS DECIR QUE ENTRE MENOR SEA EL ACOPLAMIENTO ENTRE MODULOS , MAS INDEPENDIENTES SERAN ESTOS Y MEJOR SERA EL DISEÑO DEL SISTEMA.

EXISTEN DIFERENTES TIPOS DE ACOPLAMIENTO ENTRE ELLOS TENEMOS :

-- DE DATOS O POR REFERENCIA.
-- ESTAMPADO.
-- POR CONTROL.
-- COMUN.
-- POR CONTENIDO.

EMPEZAREMOS POR EL ULTIMO :

ACOPLAMIENTO POR CONTENIDO :
.....

DOS MODULOS ESTAN ACOPLADOS POR CONTENIDO SI UN MODULO HACE UNA REFERENCIA DIRECTA A LOS CONTENIDOS DE OTRO MODULO.

CON REFERENCIA DIRECTA A LOS CONTENIDOS DEL OTRO INTENDEMOS :

I) UN MODULO ALTERA EL CONTENIDO DE OTRO.
II) UN MODULO CAMBIA DATOS EN OTRO MODULO.
III) UN MODULO CAMBIA DE INSTRUCCION SALTANDO A OTRA DE OTRO MODULO.
IV) SI DOS MODULOS COMPARTEN LAS MISMAS LITERALES.

LOS PRIMEROS TRES CASOS SON CONOCIDOS COMO PATOLOGICOS Y EL PROBLEMA PRINCIPAL ES QUE UN CAMBIO PEQUEÑO Y DE APARIENCIA INOCENTE A UNO DE LOS MODULOS PUEDE DESQUICIAR A OTRO MODULO EN CUALQUIER PARTE DEL SISTEMA.

ES IMPORTANTE HACER NOTAR QUE ESTE TIPO DE ACOPLAMIENTO ES POSIBLE EVITARLO EN LA MAYORIA DE LOS LENGUAJES DE ALTO NIVEL.

MODULOS QUE PRESENTAN ESTE ACOPLAMIENTO SON EXAGERADAMENTE DEPENDIENTES Y MUY DIFICILES DE MODIFICAR.

ACOPLAMIENTO COMUN :
.....

UN GRUPO DE MODULOS ESTAN COMUNMENTE ACOPLADOS SI REFERENCIAN O HACEN USO (POR LECTURA O ESCRITURA) DATOS HETEROGENEOS GLOBALES COMPARTIDOS O BIEN VARIABLES , ARREGLOS ETC.

ESTE TIPO DE ACOPLAMIENTO INDUCE POCA MODIFICABILIDAD Y CONFIABILIDAD POR MUCHAS RAZONES, ENTRE OTRAS LOS DATOS GLOBALES INHIBEN LEGIBILIDAD AL PROGRAMA Y CREA DEPENDENCIAS ENTRE TODOS LOS MODULOS USANDO DATOS GLOBALES TALES QUE CADA UNO DE ESTOS MODULOS DEBE CUIDAR EL USO O QUE REPRESENTA PARA EL ESTE DATO GLOBAL.

HE AQUI ALGUNAS DE LAS CONSECUENCIAS :

1) ES MAS DIFICIL REVISAR LOS LOS MODULOS QUE UTILIZAN AREAS COMUNES YA QUE USUALMENTE LO HACEN POR SU NOMBRE.
2) EL MANTENIMIENTO ES MAS DIFICIL SOBRE TODO SI EL TIPO DE DA

TOS ES DIFERENTE O VARIADO.

7) EL USO DE ESTAS REGIONES QUITA LEGIBILIDAD A LOS PROGRAMAS Y ES DIFICIL SABER QUE MODULOS USAN QUE DATOS.

8) EN LENGUAJES COMO FORTRAN DONDE LA POSICION DE LOS DATOS EN COMUN ES IMPORTANTE, ADEMAS DEL PROBLEMA DE AREA COMUN SE TIENE UN ACOPLAMIENTO DE ESTAMPADO EL CUAL DISCUTIREMOS ENSEGUIDA.

9) UN ERROR EN CUALQUIER MODULO PROPAGA LA FALLA A OTROS YA QUE ESTA REGION NO ESTA PROTEGIDA POR NINGUN MODULO.

PODEMOS AGREGAR QUE UN AMBIENTE O MEDIO COMUN PUEDE SER UNA REGION COMPARTIDA DE COMUNICACION , UN ARCHIVO CONCEPTUAL EN CUALQUIER MEDIO DE ALMACENAMIENTO , UN ARCHIVO O DISPOSITIVO FISICO , UNA AREA COMUN DE BASE DE DATOS ETC.

ACOPLAMIENTO POR CONTROL:
.....

DOS MODULOS ESTAN ACOPLADOS POR CONTROL SI UN MODULO PASA ELEMENTOS DE CONTROL A OTRO , COMO BANDERAS DE CONTROL , NOMBRES DE MODULOS Y ETIQUETAS.LOS MODULOS DE ESTE TIPO SON DEPENDIENTES YA QUE CADA UNO CONOCE ALGO ACERCA DE LA LOGICA INTERNA DEL OTRO.

ACOPLAMIENTO POR ESTAMPADO :
.....

ESTE TIPO DE ACOPLAMIENTO SE PRESENTA CUANDO DOS O MAS MAS MODULOS REFERENCIAN LA MISMA HETEROGENEA ESTRUCTURA DE DATOS , TENIENDO CUIDADO DE QUE ESTOS DATOS NO SEAN GLOBALES.

ESTO IMPLICA QUE ESTA ESTRUCTURA ES PASADA COMO PARAMETRO.

UN EJEMPLO DE ESTO SERIAN LOS TIPOS ESTRUCTURADOS EN EL LENGUAJE PASCAL. UNA ESTRUCTURA DE DATOS ES UN COMPUESTO DE ELEMENTOS Y CUALQUIER CAMBIO A ESTA ESTRUCTURA AFECTARA A TODOS LOS MODULOS QUE ESTEN ESTAMPADOS CON LA ESTRUCTURA.

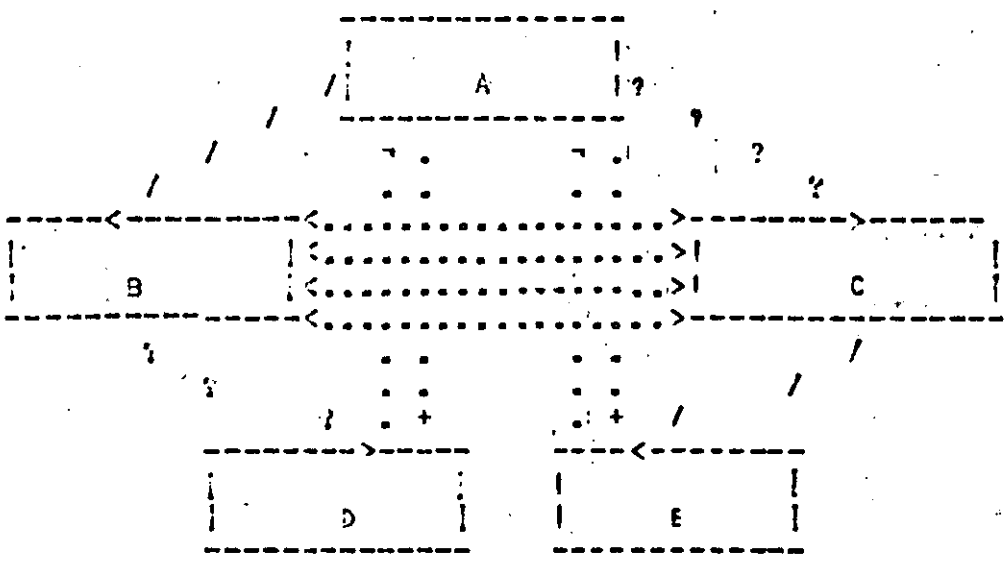
ACOPLAMIENTO DE DATOS:
.....

ESTE TIPO DE ACOPLAMIENTO ES EL MAS SENCILLO Y DESEABLE
DOS MODULOS ESTAN ACOPLADOS POR DATOS , SI UNO LLAMA AL OTRO Y ELLOS NO ESTAN ACOPLADOS POR LOS TIPOS ANTERIORES ADEMAS QUE SUS ENTRADAS Y SALIDAS DEL QUE LLAMA COMO DEL ACTIVADO SONI PASADOS COMO ARGUMENTOS HOMOGENEOS.

LAS MEJORES FORMAS DE ACOPLAMIENTO SON ESTAS DOS ULTIMAS Y LOS PROGRAMAS QUE ADQUIEREN ESTOS TIPOS DE ACOPLAMIENTO SON FACILES DE ENTENDER Y MODIFICAR Y QUE TODAS SUS INTERCONEXIONES SON EXPLICITAS Y FACILES DE IDENTIFICAR. PODEMOS DECIR DE OTRA MANERA :

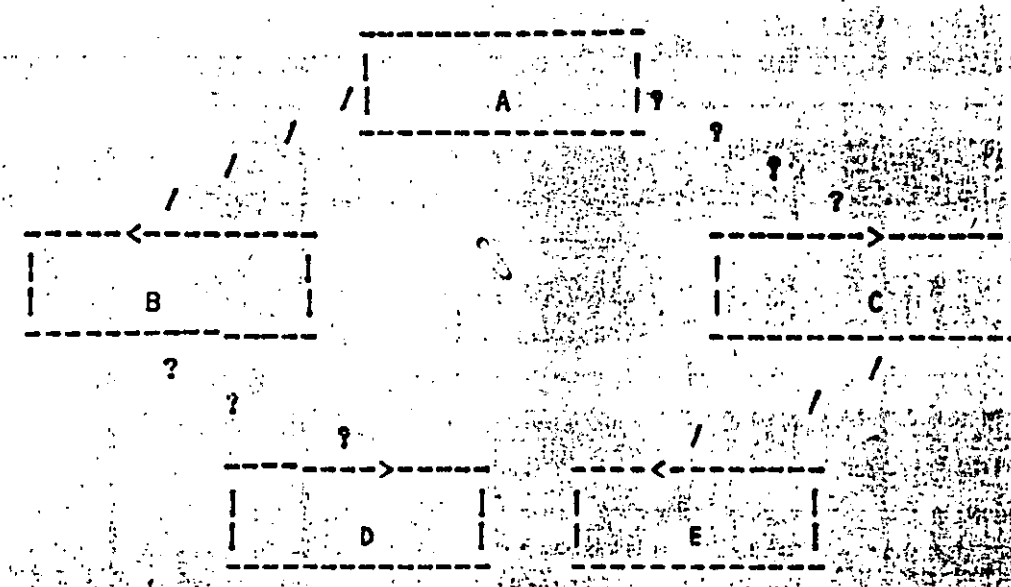
SE PRODUCE CUANDO UNICAMENTE DATOS SON PASADOS DE UN MODULO A OTRO O VICEVERSA Y DE HECHO CUALQUIER SISTEMA PUEDE DISEÑARSE DE TAL MANERA QUE SOLO CONTENGA ESTE TIPO DE ACOPLAMIENTO EN SUS MODULOS.

EFECTO DE ACOPLAMIENTO POR DISPOSITIVO COMUN



.... ACOPLAMIENTO POR AMBIENTE COMUN
(LA ZONA CENTRAL ES EL AREA COMUN A LOS MODULOS)
--- RELACION DE CONTROL

SISTEMA SIN ACOPLAMIENTO POR DISPOSITIVO COMUN



--- RELACION DE CONTROL

*** 2.4.- DESACOPLAMIENTO . ***

ES UN CONCEPTO CONTRARIO AL ACOPLAMIENTO Y ES EN SI UNA TECNICA O METODO SISTEMATICO QUE INVITA A ELABORAR MODULOS MAS INDEPENDIENTES . ES CLARO VER QUE CADA FORMA DE ACOPLAMIENTO INVITA A UNA FORMA DE DESACOPLAMIENTO .

EL ACOPLAMIENTO CAUSADO POR CUALESQUIERA DE LOS ANTERIORES TIPOS PUEDE SER DESACOPLADO CAMBIANDO APROPIADAMENTE LOS PARAMETROS ETC.

EL DESACOPLAMIENTO EN OCASIONES ES DIFICIL (PUEDE DECIRSE QUE EN LA MAYORIA DE LOS CASOS) Y ES NECESARIO CONOCER A FONDO LAS INTERRELACIONES ENTRE MODULOS PARA PODER HACERLO .

UN ESENCIAL METODO DE DESACOPLAMIENTO ES REDUCIR LOS EFECTOS DEL MEDIO COMUN EN EL ACOPLAMIENTO .

AQUI SE HA INTRODUCIDO UNO DE LOS MAS IMPORTANTES CRITERIOS PARA JUZGAR LA BONDAD DE UN DISEÑO : ACOPLAMIENTO .

EL ACOPLAMIENTO SERA CUANTIFICADO POR LA TABLA SIGUIENTE (ESTA TABLA PUEDE SER MODIFICADA EN SUS VALORES AUNQUE ESTOS VALORES DEBERAN SEGUIR DE MENOR A MAYOR PARA CONCORDAR CON TODO LO ANTERIORMENTE ESTABLECIDO).

*** 2.5.- CUANTIFICACION DEL ACOPLAMIENTO . ***

TIPO DE ACOPLAMIENTO	I	VALOR PARA EL ACOPLAMIENTO
DE DATOS	I	.1
ESTAMPADO	I	.3
POR CONTROL	I	.5
COMUN	I	.7
POR CONTENIDO	I	.9

LA CUANTIFICACION DE ESTE ACOPLAMIENTO SIRVE PARA LO SIGUIENTE :

1) REFLEJA QUE ALGUN TIPO DE ANALISIS AL PROGRAMA O SISTEMA DEBE SER REALIZADO .

2) FACILITA EL ENTENDIMIENTO O DISCUSION ACERCA DE LAS PROPIEDADES QUE EL SISTEMA TIENE .

3) IDENTIFICA LAS RELACIONES ENTRE MODULOS Y NOS DICE A QUE TIPO DE ACOPLAMIENTO DEBIERAMOS DE LLEVAR NUESTRO SISTEMA .

4) NOS PREDICE UNA MEDIDA EN BASE A EXPERIENCIA PROPIA ACERCA DE SU RENDIMIENTO , CONFIABILIDAD Y MANTENIBILIDAD .

DISCUTIREMOS A CONTINUACION UN CONCEPTO LLAMADO COHESION QUE JUNTO CON EL ACOPLAMIENTO FORMAN A MI MUY PARTICULAR MODO DE VER LA TEORIA CENTRAL DEL DISEÑO ESTRUCTURADO .

++++ 3.- COHESION .++++

*** 5.1.- QUE ES LA COHESION ? ***

LA MANERA DE ESCOGER LOS MODULOS EN UN SISTEMA ES POR FUERZA NO ARBITRARIA , Y ESTA ELECCION VA A AFECTAR EN RELACION DIRECTA LA COMPLEJIDAD DE NUESTRO SISTEMA FINAL .

UNA DE LAS COSAS MAS IMPORTANTES ES ADAPTAR NUESTRO SISTEMA AL PROBLEMA A RESOLVER .

SUPONGAMOS QUE ENCONTRAMOS UNA MEDIDA PARA VER LA EFECTIVIDAD DE UN SISTEMA . POR MEDIO DE ESTA MEDIDA VEREMOS QUE EL MAS EFECTIVO SISTEMA MODULAR ES AQUEL DONDE LA RELACION DE SUS ELEMENTOS NO TENGA NADA O CASI NADA QUE VER CON OTROS ELEMENTOS EN OTRO MODULO AJENO A ESTE .

EN POCAS PALABRAS, UN SISTEMA MODULAR ES ACEPTABLE SI SU ACOPLAMIENTO ES LO MAS DEBIL POSIBLE (ENTRE OTRAS MEDIDAS) .

ESTE ULTIMO PUNTO DEBE QUEDAR CLARO .

LO QUE DEBEMOS CONSIDERAR ES LA COHESION DE CADA MODULO POR SEPARADO, ESTO ES : QUE TAN FUERTE ES LA RELACION DE LOS ELEMENTOS UNO CON OTRO EN EL MISMO MODULO .

UN CONCEPTO QUE QUEDARA ENTENDIDO MAS ADELANTE ES EL QUE LA COHESION Y EL ACOPLAMIENTO ESTAN ESTRECHAMENTE LIGADOS EN RAZON INVERSAMENTE PROPORCIONAL, CON ESTO QUIERO DECIR, MIENTRAS MAS GRANDE LA COHESION EN LO MODULOS DE UN SISTEMA, MENOR SERA EL ACOPLAMIENTO ENTRE ESTOS .

LA COHESION Y EL ACOPLAMIENTO SON HERRAMIENTAS IMPORTANTISIMAS EN EL DISEÑO DE ESTRUCTURAS MODULARES, PERO LA COHESION EMERGE DE UNA PRACTICA INTENSIVA , COMO LA MAS IMPORTANTE .

LA COHESION REPRESENTA UN REFINAMIENTO OPERACIONAL SOBRE CONCEPTOS DE RELACION FUNCIONAL . MUCHA GENTE HA TRATADO DE ENSEÑAR O LLEVAR A LA PRACTICA MODULOS ALTAMENTE FUNCIONALES SIN ENFRENTAR EL PROBLEMA FUNDAMENTAL DE COMO RECONOCER ESTOS MODULOS FUNCIONALES.

LA COHESION PUEDE SER PUESTA EN PRACTICA CON LA IDEA DE DE PRINCIPIO ASOCIATIVO.

EL DECIDIR PONER CIERTOS ELEMENTOS DE PROCESO EN UN MODULO , EN EFECTO INVOCA UN PRINCIPIO DE CIERTAS PROPIEDADES O CARACTERISTICAS QUE RELACIONAN A LOS ELEMEINTOS QUE LAS POSSEN.

PARA DISEÑAR SISTEMAS MODULARES , DEBEMOS SER CAPACES DE DETERMINAR LA COHESION DE MODULOS QUE AUN NO EXISTEN .

LA COHESION DE UN MODULO PUEDE SER CONCEPTUALIZADA COMO EL CEMENTO QUE MANTIENE JUNTOS LOS ELEMENTOS DE PROCESO DE UN MODULO .

CLARAMENTE LA COHESION Y EL ACOPLAMIENTO ESTAN INTERRELACIONADOS. MIENTRAS MAYOR SEA LA COHESION DE LOS MODULOS INDIVIDUALES DEL UN SISTEMA , MENOR SERA EL ACOPLAMIENTO ENTRE LOS MODULOS.

*** 3.2.- NIVELES DE COHESION . ***

EL INTENTO DE JUNTAR ELEMENTOS EN UN MISMO MODULO O EL PORQUE DE PONERLOS ALLI RESULTA EN UNA ESTRATIFICACION DE LA COHESION.

EN UN PRINCIPIO FUERON SOLO TRES LOS NIVELES PERO CON EL PASO DEL TIEMPO Y DEBIDO AL EFECTO DE LOS MODULOS HOY EN DIA SE SE TIENEN MAS NIVELES .

SON 7 LOS NIVELES DE COHESION QUE SE DISTINGUEN POR SIETE PRINCIPIOS ASOCIATIVOS DIFERENTES.

LOS NIVELES SON LOS SIGUIENTES Y VAN DE MAYOR A MENOR GRADO DE COHESION .

- ASOCIACION FUNCIONAL .
- ASOCIACION SECUENCIAL.
- ASOCIACION COMUNICACIONAL.
- ASOCIACION DE PROCEDIMIENTOS.
- ASOCIACION TEMPORAL.
- ASOCIACION LOGICA.
- ASOCIACION COINCIDENTAL.

ES NECESARIO ENTENDER QUE ESTOS NIVELES NO CONSTITUYEN UNA ESCALA LINEAL , AUNQUE SE HA VISTO EN LA PRACTICA QUE LOS PRIMEROS TRES NIVELES REDUNDAN EN UN DISEÑO EFECTIVO EN CASI TODOS LOS ASPECTOS AL CONTRARIO DE LOS TRES ULTIMOS .

DAREMOS UNA EXPLICACION DE ESTOS NIVELES EMPEZANDO CON LOS DE COHESION MAS BAJA.(AL REVES DE COMO SE PRESENTARON ARRIBA).

- COHESION COINCIDENTAL .

ESTE TIPO DE COHESION OCURRE CUANDO LOS ELEMENTOS DE UN MODULO NO TIENEN NADA QUE VER LOS UNOS CON LOS OTROS. AFORTUNADAMENTE SUCEDE POCAS VECES Y EN LA MAYORIA DE LO CASOS RESULTA CUANDO UN PROGRAMADOR DECIDE JUNTAR PROPOSICIONES DE UN PROGRAMA YA ESCRITO EN UN MODULO , ES DECIR , SI VARIAS PROPOSICIONES SE REPITEN, LAS PONE EN UN MODULO .

ESTO EN PRINCIPIO PUEDE SALVAR MEMORIA Y PUEDE NO SER UN GRAN PROBLEMA , AUNQUE EL MANTENIMIENTO Y MODIFICACION EN OCASIONES SUELE PRESENTAR CASOS MUY DESAGRADABLES .

- COHESION LOGICA .

LOS ELEMENTOS DE UN MODULO SON LOGICAMENTE ASOCIADOS SI UNO PUEDE PENSAR QUE ELLOS CAEN EN LA MISMA CLASE LOGICA DE REALIZAR FUNCIONES RELACIONADAS .

UN EJEMPLO SERIA HACER UN MODULO DONDE TODAS LAS FUNCIONES DE ENTRADA SE REALIZARAN, O UNO DONDE TODAS LAS OPERACIONES ARITMETICAS SE REALIZEN ETC.

ESTE TIPO DE COHESION ES AUN MAS FUERTE QUE LA COINCIDENTAL , PRESENTA MENOS PROBLEMAS POR ASOCIAR LOS ELEMENTOS DE ESTA MANERA, PERO NO REALIZA UNA FUNCION SINO VARIAS EN FUNCION DE LO QUE

SE REQUIERA. LAS DESVENTAJAS POTENCIALES SE VERAN AL PASAR AL SIGUIENTE NIVEL DE COHESION.

@ COHESION TEMPORAL .

ES PARECIDA A LA COHESION LOGICA PERO ES AUN MAS FUERTE QUE ELLA YA QUE SUS ELEMENTOS ESTAN RELACIONADOS AL TIEMPO.

ESTO ES , SUPONGAMOS QUE HAY PARTES DEL SISTEMA QUE SE ENCARGAN DE ABRIR ARCHIVOS, CERRARLOS, PONER CONTADORES EN CERO ETC.

ESTOS MODULOS PUEDEN SER PUESTOS EN UNO SOLO Y MANDAR LOS EJECUTAR TODOS A UN TIEMPO, ESTOS REALIZARAN SUS ACTIVIDADES EN UN MOMENTO DADO DENTRO DE LA EJECUCION DEL SISTEMA Y POR TIEMPO DE TERMINADO.

SI UNO ENTRA A MODIFICAR ESTOS MODULOS EL PROBLEMA PUEDE SER GRANDE YA QUE PUEDE HABER EXCEPCIONES QUE MANEJAR DENTRO DEL MODULO Y CREAR INTERDEPENDENCIAS MUY GRANDES QUE SE REFLEJARAN EN LA PROPAGACION DE ERRORES DEL SISTEMA SI ESTO FALLA.

DICHO DICHO, EL PROBLEMA REALMENTE OCURRE CUANDO EL PROGRAMADOR DE MANTENIMIENTO DESEA CAMBIAR UNA FUNCION DEL MODULO SIN QUERER DESTRUIR CUALQUIER OTRA , Y SI EL PROCESAMIENTO FUNCIONAL DE LOS ELEMENTOS HAN SIDO MEZCLADOS EN EL MODULO, ESTA TAREA SERA DIFICIL DE RESOLVER.

@ COHESION DE PROCEDIMIENTOS .

EN UN PRINCIPIO AL TRATAR DE MEDIR LA COHESION DE LOS MODULOS SE NOTO QUE CUANDO UN DISEÑADOR TRATABA O USABA DIAGRAMAS DE FLUJO PARA ELABORAR UN PROCESO Y DISEÑARLO MODULARMENTE , LOS RESULTADOS ERAN ALTAMENTE VARIABLES Y DE BAJA COHESION.

ESTOS ELEMENTOS MODULARMENTE ASOCIADOS FORMAN PARTE COMUN DE UNA UNIDAD MODULAR ; SON COMBINADOS EN UN MODULO DE COHESION DE PROCEDIMIENTOS YA QUE SON ENCONTRADOS EN EL MISMO PROCEDIMIENTO.

UNA UNIDAD COMUN DE PROCEDIMIENTOS PUEDE SER UN LOOP O PROCESO DE DECISION O UNA SECUENCIA LINEAL DE PASOS. ESTA ULTIMA RELACION , UNA SIMPLE SUCESION DE PASOS, ES DEBIL Y A VECES CAE EN UNA COHESION TEMPORAL.

UN MODULO TEMPORALMENTE COHESIVO PUEDE INCLUIR VARIOS PASOS LOS CUALES PUEDEN SER EJECUTADOS EN UN DETERMINADO TIEMPO, PERO NO NECESARIAMENTE EN UNA SECUENCIA PARTICULAR. LA INICIALIZACION ES UN EJEMPLO OBVIO . LA DISTINCION AQUI ES QUE LOS MODULOS QUE ADQUIEREN UNA COHESION DE PROCEDIMIENTOS SON AQUELLOS EN QUE SUS ELEMENTOS PERTENECEN A UNA ITERACION , DECISION U OPERACION SECUENCIAL.

ESTE TIPO DE MODULOS TIENE SUS PROBLEMAS Y EL PUNTO ES QUE UN MODULO CON COHESION DE PROCEDIMIENTOS SOLO CONTIENE PARTES O FRAGMENTOS DE VARIAS FUNCIONES.

@ COHESION COMUNICACIONAL .

NINGUNO DE LOS NIVELES DE COHESION ANTERIORMENTE DESCRITOS ESTA LIGADO FUERTEMENTE A LA ESTRUCTURA DE UN PROBLEMA ESPECIFICO.

ESTE TIPO DE COHESION ES EL GRADO MAS BAJO DONDE ENCONTRAMOS UNA RELACION DE LOS ELEMENTOS A PROCESAR DENTRO DEL MODULO QUE SEA DEPENDIENTE A UN PROBLEMA .

QUE ESTOS ELEMENTOS SEAN COMUNICACIONALMENTE ASOCIADOS QUIERE DECIR QUE TODOS ESTOS ELEMENTOS OPERAN BAJO LOS MISMOS DATOS DE ENTRADA O PRODUCEN LOS MISMOS DATOS DE SALIDA .

UNA ASOCIACION COMUNICACIONAL ES COMUN EN APLICACIONES COMERCIALES O DE NEGOCIOS.

GENERALMENTE ES EL RESULTADO DE PENSAR EN TERMINO DE TODAS LAS COSAS QUE SE PUEDEN HACER CON ALGUNOS DATOS UNA VEZ QUE ESTOS SE HAN OBTENIDO O GENERADO O POR OTRA PARTE EN TERMINOS DE LO QUE SE DEBE HACER PARA GENERAR O DAR COMO RESULTADO ALGO, COMO POR EJEMPLO UNA LINEA DETALLADA DE UN REPORTE ETC.

EJEMPLOS TIPICOS SERIAN : UN MODULO QUE IMPRIMA UN ARCHIVO DE TRANSACCIONES O UN MODULO QUE ACEPTE DATOS DE DIFERENTES FUENTES TRANSFORMANDOLAS Y ENSAMBLANDOLAS EN UNA LINEA DEL REPORTE.

⊖ COHESION SECUENCIAL .

ES UNA ASOCIACION SECUENCIAL DONDE LOS RESULTADOS O DATOS DE SALIDA DE UN ELEMENTO DENTRO DEL MODULO SIRVEN DE ENTRADA A OTRO DENTRO DEL MISMO MODULO.

EN TERMINOS DE UN DIAGRAMA DE FLUJO DE PROGRAMA , LA COHESION SECUENCIAL COMBINA UNA CADENA LINEAL DE TRANSFORMACIONES SUCESIVAS DE DATOS.

LA DEBILIDAD POTENCIAL DE UN MODULO SECUENCIAL ES SIMILAR A UNO DE LOS PROBLEMAS DE LOS MODULOS COINCIDENTALES , LOGICOS, TEMPORALES, DE PROCEDIMIENTOS Y COMUNICACIONALES . AL TRATAR DE MODIFICAR EL CODIGO PARA UNA FUNCION QUE SE ENCUENTRA EN TODA O UNA PARTE DEL MODULO , EL PROGRAMADOR PUEDE ENCONTRAR QUE ESTA INADVERTIDAMENTE MODIFICANDO O QUE DEBE CONSIDERAR CODIGO PARA OTRA FUNCION QUE SE ENCUENTRA EN EL MISMO MODULO.

SIMILARMENTE SI ENCONTRAMOS QUE CADA MODULO CONTIENE PARTE DE UNA FUNCION LOS ARGUMENTOS DEL ACOPLAMIENTO SE APLICAN :

"PARA ENTENDER LO QUE HACE UN MODULO , DEBEMOS ENTENDER LO QUE EL OTRO HACE Y EL SEGUNDO PUEDE CONTENER OTROS ELEMENTOS DE PROCESO QUE NADA TIENEN QUE HACER CON LAS FUNCIONES QUE REALIZA EL PRIMERO".

⊕ COHESION FUNCIONAL .

EN UN MODULO COMPLETAMENTE FUNCIONAL , CADA ELEMENTO DE PROCESO ES UNA PARTE INTEGRAL DE, Y ES ESENCIAL A LA REALIZACION DE UNA SOLA FUNCION.

POR LO TANTO UN MODULO QUE ES PURAMENTE FUNCIONAL NO CONTIENE EXTRAÑOS ELEMENTOS RELACIONADOS SOLO POR PRINCIPIOS SECUENCIALES O DEBILES . ES IMPORTANTE DECIR QUE LOS MAS CLAROS Y FACILES EJEMPLOS DE ENTENDER COMO ASOCIACION FUNCIONAL SON LOS QUE PROVIENEN DE LAS MATEMATICAS . LA ELABORACION DE UN MODULO QUE REALICE UNA RAIZ CUADRADA ES ALTAMENTE COHESIVA Y PROBABLEMENTE COMPLETAMENTE FUNCIONAL.

IDENTIFICAMOS ESTA CLASE DE MODULOS FUNCIONALES COMPARANDO SU COHESION FUNCIONAL CON LAS DEMAS . ESTA IDENTIFICACION SE ANTOJA DESCUBRIR POR UN METODO DE ELIMINACION ANTE LAS DEMAS.

LAS SIGUIENTES LINEAS NOS AYUDARAN A DISTINGUIR LOS MODULOS NO FUNCIONALES.

* SI LA UNICA MANERA RAZONABLE DE DESCRIBIR LA OPERACION

DE UN MODULO ES UNA PROPOSICION COMPUESTA O UNA PROPOSICION QUE CONTIENE UNO O MAS VERBOS , ENTONCES EL MODULO ES PROBABLEMENTE MENOS FUNCIONAL . PUDO SER SECUENCIAL , COMUNICACIONAL O LOGICO EN TERMINOS DE COHESION.

* SI LA DESCRIPCION DE LA PROPOSICION CONTIENE PALABRAS OBLIGADAS COMO : CUANDO , HASTA , PRIMERA , ENTONCES , DESPUES , PROXIMO , PRIMERO , ETC. , ENTONCES EL MODULO ES PROBABLEMENTE TEMPORAL O DE PROCEDIMIENTOS O A VECES ES INDICATIVO DE COHESION SECUENCIAL.

* SI LA DESCRIPCION DE LA PROPOSICION NO CONTIENE UN OBJETIVO ESPECIFICO EL MODULO ES PROBABLEMENTE LOGICAMENTE COHESIVO.

* PALABRAS TALES COMO "INICIALIZA","LIMPIA",ETC IMPLICAN UNA COHESION TEMPORAL.

ES IMPORTANTE RECORDAR QUE HAY MUCHAS FORMAS DE DESCUBRIR LA TAREA DE UN MODULO Y ALGUNAS DE LAS DESCRIPCIONES PUEDEN HACER QUE EL MODULO PAREZCA FUNCIONAL AUNQUE NO LO SEA (Y VICEVERSA).

*** 3.3.- CUANTIFICACION DE LA COHESION .***

DE IGUAL MANERA QUE SE HIZO CON EL ACOPLAMIENTO, CUANTIFICAREMOS AHORA LA COHESION, RECORDANDO QUE LA COHESION ES UNA MEDIDA DE RELACION ENTRE LOS ELEMENTOS DE UN MISMO MODULO .

LA SIGUIENTE TABLA NOS DA UNA IDEA DE LO FUERTE DE ESTA RELACION.

TIPO DE COHESION	I	VALOR DE LA COHESION
.9	I	FUNCIONAL
.8	I	SECUENCIAL
.7	I	COMUNICACIONAL
.5	I	DE PROCEDIMIENTO
.3	I	TEMPORAL
.2	I	LOGICA
.1	I	COINCIDENTAL

PODEMOS DECIR :

"LA COHESION DE UN MODULO ES APROXIMADAMENTE EL MAYOR NIVEL DE COHESION APLICABLE A TODOS LOS ELEMENTOS DE PROCESO DEL MODULO."

LAS CUANTIFICACIONES OBTENIDAS PARA EL ACOPLAMIENTO Y COHESION NOS SERAN DE UTILIDAD MAS ADELANTE PARA DESARROLLAR UN MODELO QUE JUSTIFIQUE EL COMPORTAMIENTO DE UN SISTEMA O PROGRAMA .

*** 4.1.- CARACTERISTICAS DE UN PROGRAMA . ***

DESPUES DE TODO LO ANTERIORMENTE DESCRITO PODEMOS LLEGAR A DERIVAR SITUACIONES NOCIVAS PARA LA CREACION DE NUESTROS PROGRAMAS , ASI COMO SI ESTE YA EXISTE QUE COSAS PUEDEN LLEGAR A FALLAR MAS RAPIDAMENTE Y COMO EVITARLO. INCLUSO CON ESTO, PODEMOS DECIDIR O LLEGAR A LA CONCLUSION DE QUE PARTES DE NUESTRO SISTEMA O PROGRAMA ESTAN CONFUSAS O MAL PROGRAMADAS.

ANTES DE ENTRAR A UN DIAGNOSTICO OBTENIDO POR EL ANALISIS DE NUESTRO PROGRAMA O PROGRAMAS , ES NECESARIO RECALCAR QUE EL UN VEREDICTO SOBRE LA BONDAD O MALA REALIZACION DE UN PROGRAMA SE PUEDE LLEVAR A CABO Y DE HECHO LA LLEVAREMOS A CABO EN BASE A LA INFORMACION QUE NOS PROPORCIONE TAN SOLO SU SIMPLE LISTADO , ESTO SE REALIZARA CON LA AYUDA DE LA TEORIA YA EXPUESTA SOBRE COHESION Y ACOPLAMIENTO.

LA IDEA DE LLEVAR ESTO A CABO SE DERIVA DEL SIGUIENTE RAZONAMIENTO : PODEMOS DEDUCIR QUE BASICAMENTE EL TIPO DE INFORMACION QUE SE PUEDE OBTENER DE UN PROGRAMA QUEDARIA DIVIDIDO EN DOS PARTES .

ESTAS PARTES VENDRIAN A SER CIERTO TIPO DE CARACTERISTICAS QUE NOS ENTREGAN INFORMACION AL TIEMPO DE LA EJECUCION DE UN PROGRAMA O PARTES DE QUE SE EJECUTE ESTE Y NOS BRINDE SUS RESULTADOS.

ESTE TIPO DE CARACTERISTICAS LAS DENOMINAREMOS POR ANALOGIA A SU STATUS: CARACTERISTICAS ESTATICAS Y CARACTERISTICAS DINAMICAS.

LAS CARACTERISTICAS DINAMICAS NOS PROVEEN DE INFORMACION QUE SE PUEDE OBTENER POR MEDIO DEL ANALISIS DE LOS RESULTADOS PRODUCIDOS O BIEN DE LOS QUE SE ESTAN PRODUCIENDO SIN LLEGAR AL TERMINAR AUN LA EJECUCION DEL PROGRAMA.

ESTE TIPO DE INFORMACION PUEDE SER EXTRAIDO POR MEDIO DE MONITOREOS AL PROGRAMA , VACIADOS DE MEMORIA , SISTEMAS DE INFORMACION , BASES DE DATOS , ETC.

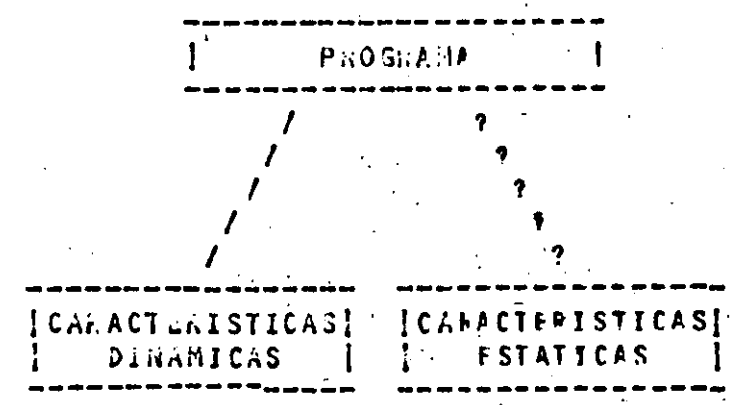
LAS CARACTERISTICAS ESTATICAS NOS PROVEEN DE INFORMACION QUE ES POSIBLE EXTRAER SIN LA NECESIDAD DE LLEGAR AL EJECUTAR EL PROGRAMA.

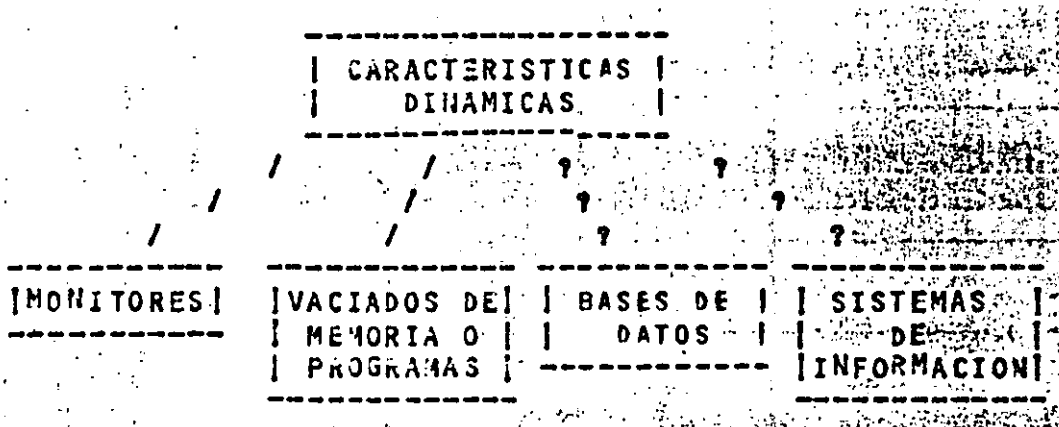
LA MANERA DE OBTENER ESTA INFORMACION SERIA UTILIZANDO ANALIZADORES DE CODIGO , LA DOCUMENTACION PROVISTA PARA EL SISTEMA , EL LISTADO DEL PROGRAMA MISMO , ETC.

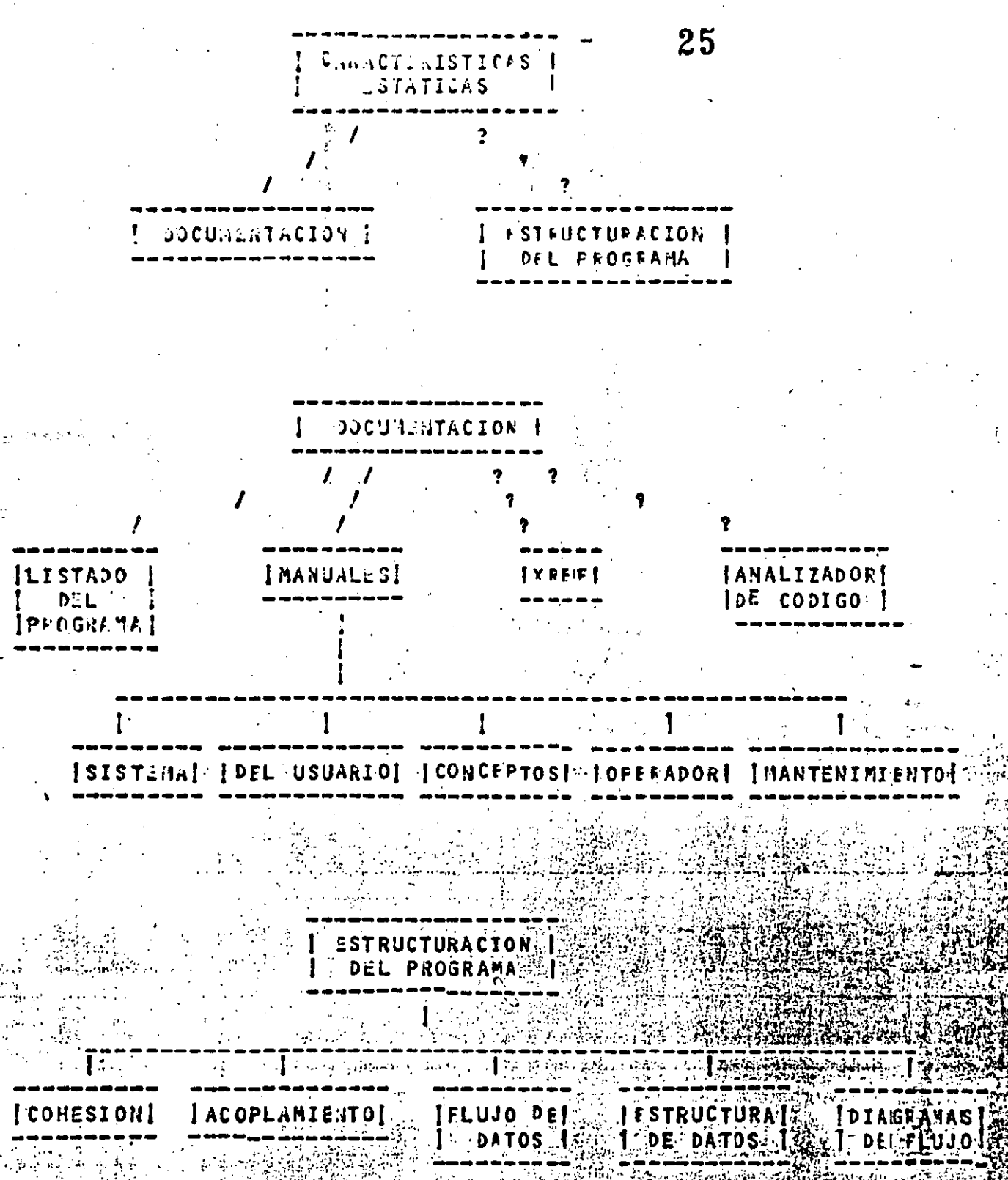
SI BIEN ESTE SERA UN MODELO TEORICO DE ANALISIS Y DIAGNOSTICO CUYA MAYORIA DE CONCEPTOS PUEDEN APLICARSE A UN SINNUMERO DE LENGUAJES DE PROGRAMACION , SERA NECESARIO LIMITARLO A PROGRAMAS DE ALTO NIVEL , EN ESPECIAL ESCOGERE EL LENGUAJE PASCAL POR LAS SIGUIENTES VENTAJAS : SU COMPILADOR ES ACEPTABLEMENTE PEQUEÑO POR LA SENCILLA Y BIEN DEFINIDA SINTAXIS , POR SER UN LENGUAJE ESTRUCTURADO QUE PERMITE DE UNA MANERA CLARA LA DEFINICION DE MODULOS Y OTRO TIPO DE ESTRUCTURAS QUE EJEMPLIFICAN CLARAMENTE LO DISCUTIDO POR LA

TEORIA DE COHESION Y ACOPLAMIENTO , POR SER FACIL DE ENTENDER A PRIMERA VISTA SU PROGRAMACION EN COMPARACION CON OTROS LENGUAJES , POR LA FACILIDAD DE SU PROGRAMACION , Y PORQUE TIPIFICA CLARAMENTE LA MAYORIA DE LOS CASOS EN LOS QUE UNO SE PUEDE ENCONTRAR .

CARACTERISTICAS DE UN PROGRAMA .






CARACTERISTICAS ESTATICAS
DOCUMENTACION
ESTRUCTURACION DEL PROGRAMA
DOCUMENTACION
LISTADO DEL PROGRAMA
MANUALES
XREF
ANALIZADOR DE CODIGO
SISTEMA
DEL USUARIO
CONCEPTOS
OPERADOR
MANTENIMIENTO
ESTRUCTURACION DEL PROGRAMA
COHESION
ACOPLAMIENTO
FLUJO DE DATOS
ESTRUCTURA DE DATOS
DIAGRAMAS DE FLUJO

COMO MUCHOS DE LOS LENGUAJES DE PROGRAMACION LA ESTRUCTURA DE PASCAL ESTA DIVIDIDO EN DOS PARTES FUNDAMENTALES.

UNA ES LA DE DECLARACIONES Y LA OTRA LA DE EXPRESIONES EJECUTABLES.

ESTO ES SIMILAR A LO SIGUIENTE Y QUE CON SEGURIDAD APARECE EN LA MAYORIA DE TEXTOS EN CUANTO AL ESTUDIO DE PASCAL SE REFIERE.

EL PROGRAMA ESTA DIVIDIDO EN 2 PARTES LLAMADAS ENCABEZADO Y CUERPO , TAMBIEN LLAMADO A VECES BLOQUE.

EL ENCABEZADO LE DA AL PROGRAMA UN NOMBRE Y LISTA SUS PARAMETROS. ESTOS SON ARCHIVOS O VARIABLES Y REPRESENTAN LOS ARGUMENTOS Y RESULTADOS DE LA OPERACION.

EL BLOQUE CONSISTE DE 6 SECCIONES , DE LAS CUALES TODAS PUEDEN SER VACIAS CON EXCEPCION DE LA ULTIMA .

EN EL ORDEN REQUERIDO POR EL COMPILADOR SON :

1) DECLARACIÓN DE ETIQUETAS.
2) DEFINICION DE CONSTANTES.
3) DEFINICION DE TIPOS.
4) DECLARACION DE VARIABLES.
5) DECLARACION DE FUNCIONES O PROCEDIMIENTOS (MODULOS).
6) EXPRESIONES EJECUTABLES.

UNA VEZ SABIENDO QUE TIPO DE DECLARACIONES PODEMOS ENCONTRARNOS , PODEMOS EMPEZAR DE UNA MANERA UN POCO GLOBAL O GENERAL A DECIR QUE COSAS HACEN DAÑO A NUESTRO PROGRAMA.

ES NECESARIO RECALCAR QUE LO QUE SE TRATARA DE MEDIR O VALORAR POR MEDIO DEL MODELO A ELABORAR ES PRINCIPALMENTE EL ACOPLAMIENTO AUNQUE HAY MUCHO DE SUBJETIVO EN ALGUNAS COSAS.

ESTA SUBJETIVIDAD NOS IMPIDE EN CIERTOS CASOS NO PODER AFIRMAR CATEGORICAMENTE . ESTO MISMO IMPIDE QUE DE NO TRATEMOS EN PRIMERA INSTANCIA DE HACER UN ANALISIS SEMEJANTE A LA COHESION .

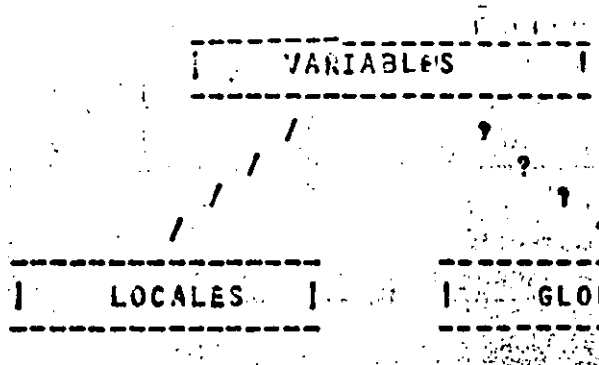
POR LO TANTO Y SINTIENDO NO TAN INTUITIVAMENTE , YA QUE ALGUNOS CASOS SON CLAROS , TRATAREMOS DE MEDIR EL ACOPLAMIENTO.

INTERNEMONOS DE UNA VEZ Y UN POCO SUPERFICIAL PARA EMPEZAR , A ANALIZAR NUESTRO LISTADO DEL PROGRAMA , HERRAMIENTA SUFICIENTE EN ESTE ESTUDIO .

YA VIMOS TODAS LAS PARTES EN QUE ESTA COMPUESTO Y LA SERIE DE DECLARACIONES QUE TENEMOS DISPONIBLES EN EL PROGRAMA.

ALGO MUY UTILIZADO POR LOS PROGRAMADORES PARA HACER OPERACIONES O PARA GUARDAR RESULTADOS O TRANSMITIR INFORMACION , SON LAS LLAMADAS VARIABLES . LAS VARIABLES PUEDEN SER LOCALES A CIERTA PARTE DEL PROGRAMA O BIEN PUEDEN SER VISTAS DESDE CUALQUIER PARTE DE ESTE .

A ESTO SE LE DENOMINA VARIABLE LOCAL O GLOBAL SEGUN EL CASO .



ESTAS DOS CATEGORIAS EN QUE CAEN LAS VARIABLES TIENEN SUS VENTAJAS Y DESVENTAJAS , COMO TODO .

LAS VARIABLES LOCALES SOLO PUEDEN UTILIZARSE DENTRO DEL BLOQUE EN QUE HAN SIDO DECLARADAS .

LAS VARIABLES GLOBALES COMO SU NOMBRE LO INDICA, PUEDEN SER UTILIZADAS EN CUALQUIER PARTE DEL PROGRAMA .

PUES BIEN , SI ESTAS VARIABLES HAN SIDO DECLARADAS, AL TIEMPO DE EJECUCION SIEMPRE SE LES ASIGNARA UN ESPACIO EN MEMORIA DONDE GUARDARAN ALGUN RESULTADO O INFORMACION INDEPENDIENTEMENTE DE QUE HAYAN SIDO UTILIZADAS O NO .

POR LO TANTO UNA PRIMERA SITUACION QUE ESPERAMOS O DEBIERAMOS ESPERAR ES QUE SI APARECEN DECLARADAS, AL MENOS SEAN UTILIZADAS DENTRO DE LAGUNA PARTE DEL PROGRAMA, YA QUE SI NO SOLO DESPERDICIARAN ESPACIO DE MEMORIA , ADEMAS DE QUITARLE CLARIDAD AL PROGRAMA.

ESTAS VARIABLES YA SEAN LOCALES O GLOBALES CAEN EN AL

GUNA DE ESTAS CATEGORIAS : ESCALARES O ESTRUCTURADAS .

DENTRO DE LA CATEGORIA ESCALAR ESTAN LOS LLAMADOS TIPOS ESTANDAR QUE TODOS CONOCEMOS Y QUE SON : EL REAL , EL ENTERO , EL BOOLEANO Y EL TIPO CHAR O CARACTER .

HAY OTROS TIPOS ESCALARES QUE EL PROGRAMADOR PUEDE DEFINIR Y SU DEFINICION NOS INDICA UN CONJUNTO ORDENADO DE VALORES A LOS CUALES EL IDENTIFICADOR EL CUAL DENOTA LOS VALORES .

EJEMPLO DE ESTO SERIA :

```
TYPE COLOR = ( BLANCO , AMARILLO , VERDE , NEGRO);
```

UN TIPO PUEDE SER DEFINIDO COMO UN SUBRANGO DE OTRO YA DEFINIDO . LA DEFINICION DE SUBRANGO NOS INDICA EL MENOR Y MAYOR VALOR CONSTANTE .

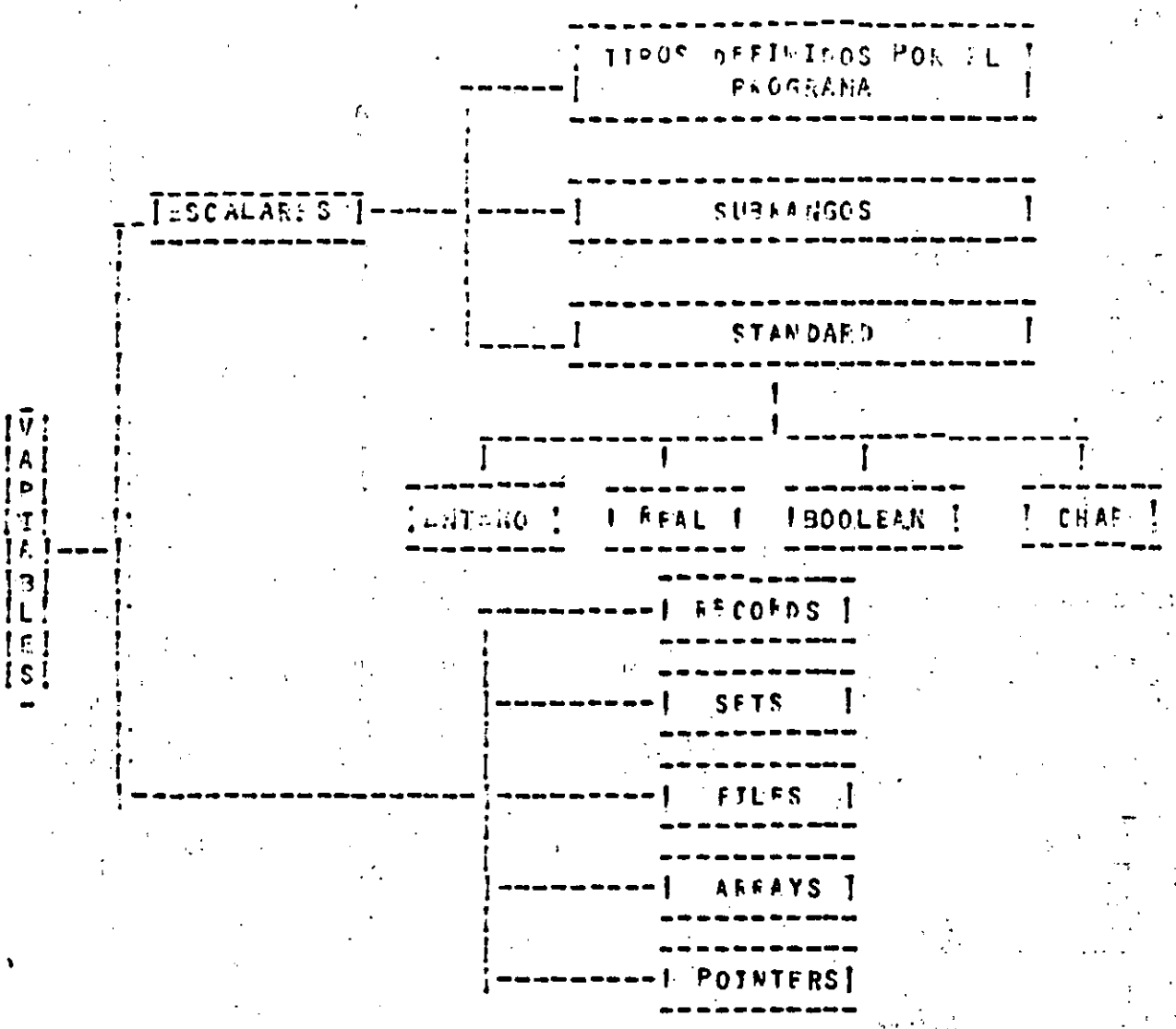
EJEMPLO :

```
TYPE COLOR = ( BLANCO , AMARILLO , VERDE , NEGRO );
SCOLOR = AMARILLO .. NEGRO +SUBRANGO DE COLOR!
```

ESTOS TIPOS ESCALARES Y LOS SUBRANGOS SON TIPOS NO ESTRUCTURADOS . LOS DEMAS TIPOS EN PASCAL SON TIPOS ESTRUCTURADOS.

ESTOS TIPOS ESTRUCTURADOS SON COMPOSICIONES DE OTROS TIPOS.

EJEMPLO DE ESTO TENEMOS A LOS ARREGLOS , A LOS SETS , A LOS ARCHIVOS , A LOS RECORDS Y A LOS POINTERS .



SIGUIENDO CON LO QUE SERIA CONVENIENTE HACER, ALGO CONVENIENTE SERIA QUE AL DECLARAR ARREGLOS TRATEMOS DE INICIALIZARLOS HASTA QUE EN VERDAD SEA NECESARIO YA QUE ESTOS OCUPAN TANTO AREA DE MEMORIA PRINCIPAL COMO SECUNDARIA.

LAS ETIQUETAS HASTA DONDE SEA POSIBLE NO HAY QUE DECLARARLAS (UN CASO EN EL QUE SON UTILES ES, CUANDO SON DE ACCION, ES DECIR, EN CASO DE QUE ALGUNA INSTRUCCION COMO ESCRITURA O LECTURA SE REALICE INSATISFACTORIAMENTE HAY QUE TOMAR ALGUNA ACCION CORRECTIVA O PREVENTIVA).

HAY QUE TRATAR DE EVITAR LOS SALTOS O GO TO'S POR MEDIO DE LA UTILIZACION DE OTRAS INSTRUCCIONES MAS ESTRUCTURADAS COMO SON LAS DECLARACIONES IF-THEN-ELSE, WHILE, DO-UNTIL, O REPEAT-UNTIL ETC., YA QUE AL HACER SALTOS SE PIERDE CONTINUIDAD Y ES FACIL ROMPER EL CONTROL DE ALGUNA INSTRUCCION, CON ESTO TAMBIEN SE OCASIONA QUE MUCHAS DE LAS VECES SE PIERDA UNO EN EL PROGRAMA DIFICULTANDO LA MAYORIA DE LAS VECES UNA FUTURA CORRECCION O CAMBIO.

DE LOS MODULOS, SUBRUTINAS O PROCEDIMIENTOS SE PUEDE HA

DLAN Y ESCRIBIR MUCHO , PARA EMPEZAR ES CONVENIENTE QUE SE DECLAREN TANTO VARIABLES COMO ARREGLOS LOCALES (AUNQUE A VECES NO ES POSIBLE HACERLO) Y DE SER POSIBLE NO UTILIZAR LAS GLOBALES YA QUE AL HACER EL ASIGNACIONES CON ESTAS, SE CAMBIAN LOS VALORES QUE SE TENIAN ORIGINALMENTE.

LO ES BUENO SALIRSE DE ESTAS SUBRUTINAS POR MEDIO DE GO TO'S .

TRATAR DE EVITAR LAS CONEXIONES EXCESIVAS ENTRE MODULOS , ENTRE MENOS Y SIMPLES SEAN LAS CONEXIONES MAS FACIL DE ENTENDER SE VUELVE EL MODULO .

MINIMIZANDO ESTAS CONEXIONES ENTRE MODULOS TAMBIEN SE MINIMIZAN LAS TRAYECTORIAS A LO LARGO DE LAS CUALES LOS ERRORES Y LOS CAMBIOS SE PUEDEN PROPAGAR EN OTRAS PARTES DEL SISTEMA.

UNA TECNICA DE LA QUE HABLABAMOS ERA LA DE EVITAR USAR DATOS COMUNES (VARIABLES O ARREGLOS GLOBALES, ETC. O MODULOS SIN SU PROPIO CONJUNTO DE DECLARACIONES) QUE PUEDE RESULTAR EN UN ENORME NUMERO DE CONEXIONES ENTRE LOS MODULOS DE UN PROGRAMA.

LA COMPLEJIDAD DE UN PROGRAMA SE AFECTA NO SOLO POR EL NUMERO DE CONEXIONES SINO TAMBIEN POR EL GRADO AL CUAL CADA CONEXION ASOCIA 2 MODULOS, HACIENDOLOS INTERDEPENDIENTES EN LUGAR DE HACERLOS INDEPENDIENTES .

EL ACOPLAMIENTO ES LA MEDIDA DE ESTRECHEZ DE ASOCIACION ESTABLECIDA POR UNA CONEXION DE UN MODULO A OTRO.

UN ACOPLAMIENTO FUERTE COMPLICA UN PROGRAMA YA QUE UN PROCEDIMIENTO O MODULO ES MAS DIFICIL DE ENTENDER , CAMBIAR O MODIFICAR SI ESTA ESTRECHAMENTE LIGADO O INTERRELACIONADO CON OTRO U OTROS MODULOS .

LA COMPLEJIDAD PUEDE SER REDUCIDA DISEÑANDO EL PROGRAMA CON EL ACOPLAMIENTO MAS DEBIL POSIBLE ENTRE MODULOS.

UN PROGRAMA BIEN ESTRUCTURADO ES AQUEL EN EL CUAL LA COMUNICACION SE HACE VIA PARAMETROS EN INTERFASES DEFINIDAS,YA QUE SE REQUIERE MENOS ESFUERZO PARA SU ENTENDIMIENTO QUE UNO QUE HACE USO EXTENSIVO DE VARIABLES GLOBALES O COMPARTIDAS.

LOS MECANISMOS DE TRANSMISION VARIAN DE LENGUAJE A LENGUAJE.

LOS MECANISMOS MAS COMUNES DE TRANSMISION SON :

- POR REFERENCIA
- POR VALOR
- POR NOMBRE
- POR RESULTADO
- POR VALOR/RESULTADO

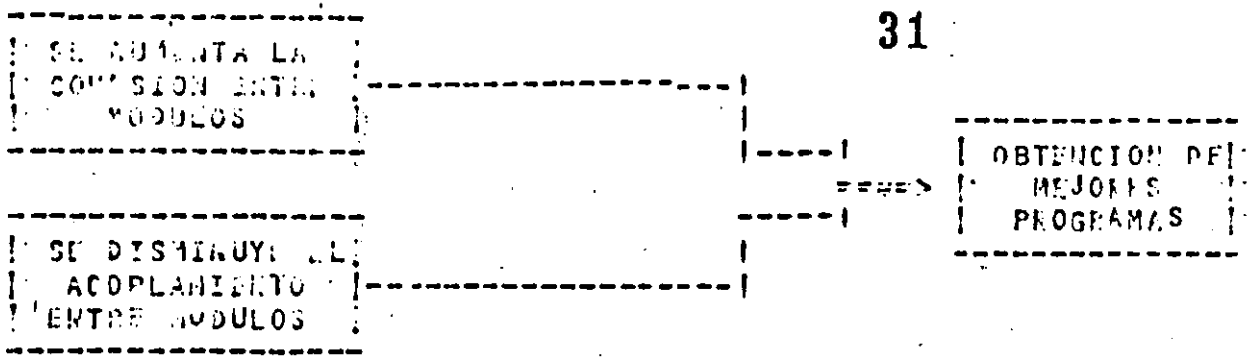
ESTOS MECANISMOS SON DE INTERES YA QUE PUEDEN RESTRINGIR LOS MEDIOS POR LOS CUALES LOS DATOS SON TRANSMITIDOS ENTRE MODULOS Y ESTOS MECANISMOS PUEDEN AFECTAR LOS RESULTADOS REGRESADOS POR UN MODULO .

EL ACOPLAMIENTO ES REDUCIDO CUANDO LAS RELACIONES ENTRE LOS ELEMENTOS QUE NO ESTAN DENTRO DEL MISMO MODULO SON MINIMIZADOS.

HAY 2 MANERAS DE OBTENER ESTO COMO YA SABEMOS :

MINIMIZANDO LA RELACION ENTRE MODULOS Y MAXIMIZANDO LAS RELACIONES ENTRE LOS ELEMENTOS DEL MISMO MODULO.

ESTE SEGUNDO METODO NOS LLEVO A DISCUTIR OTRA PARTE IMPORTANTE QUE FUE LA COHESION.


SE AUMENTA LA
COMUNICACION ENTRE
MODULOS
SE DISMINUYE EL
ACOPLAMIENTO
ENTRE MODULOS
OBTENCION DE
MEJORES
PROGRAMAS

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

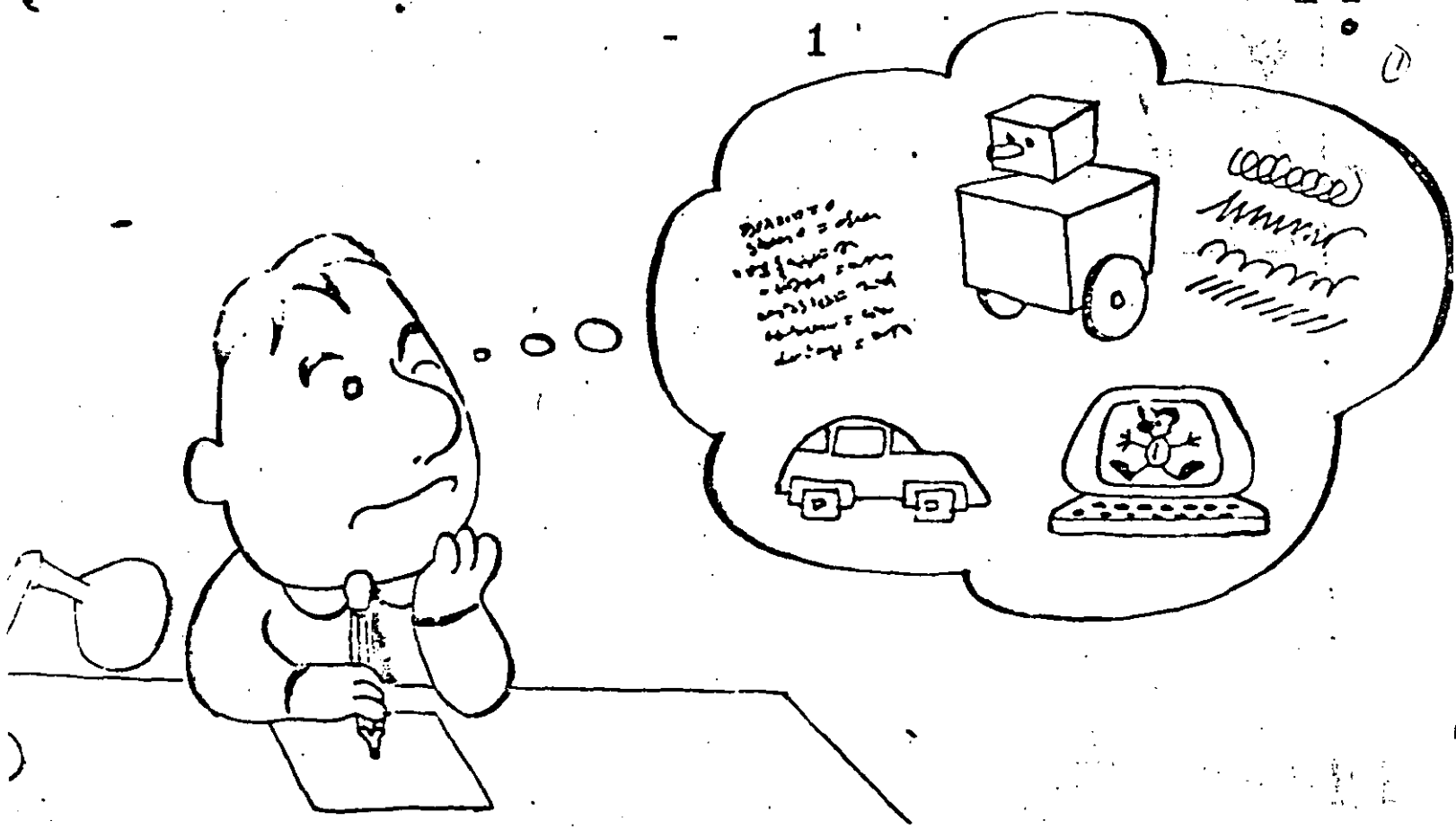
METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

ANEXOS AL TEMA:

DISEÑO ESTRUCTURADO

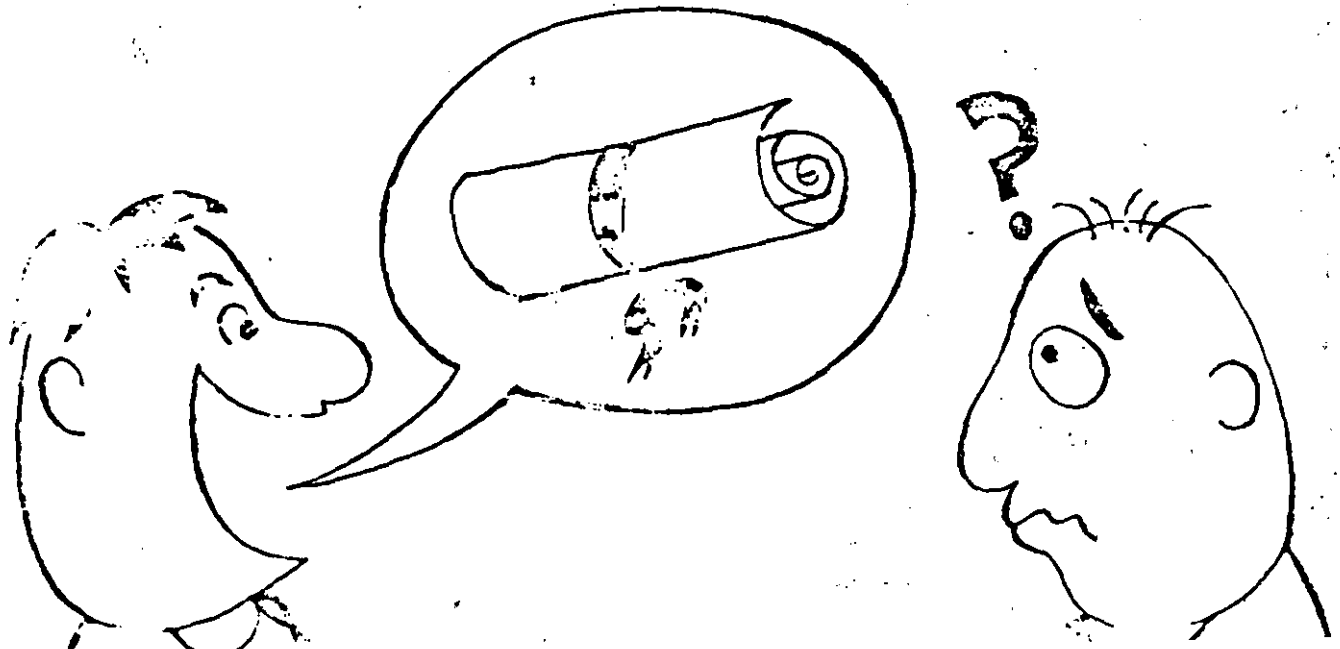
ING. ALEJANDRO ACOSTA

MAYO, 1985



DISEÑO es la comunicación

de una IDEA

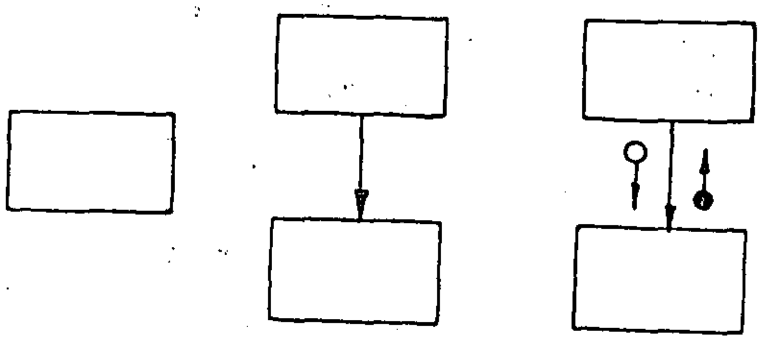


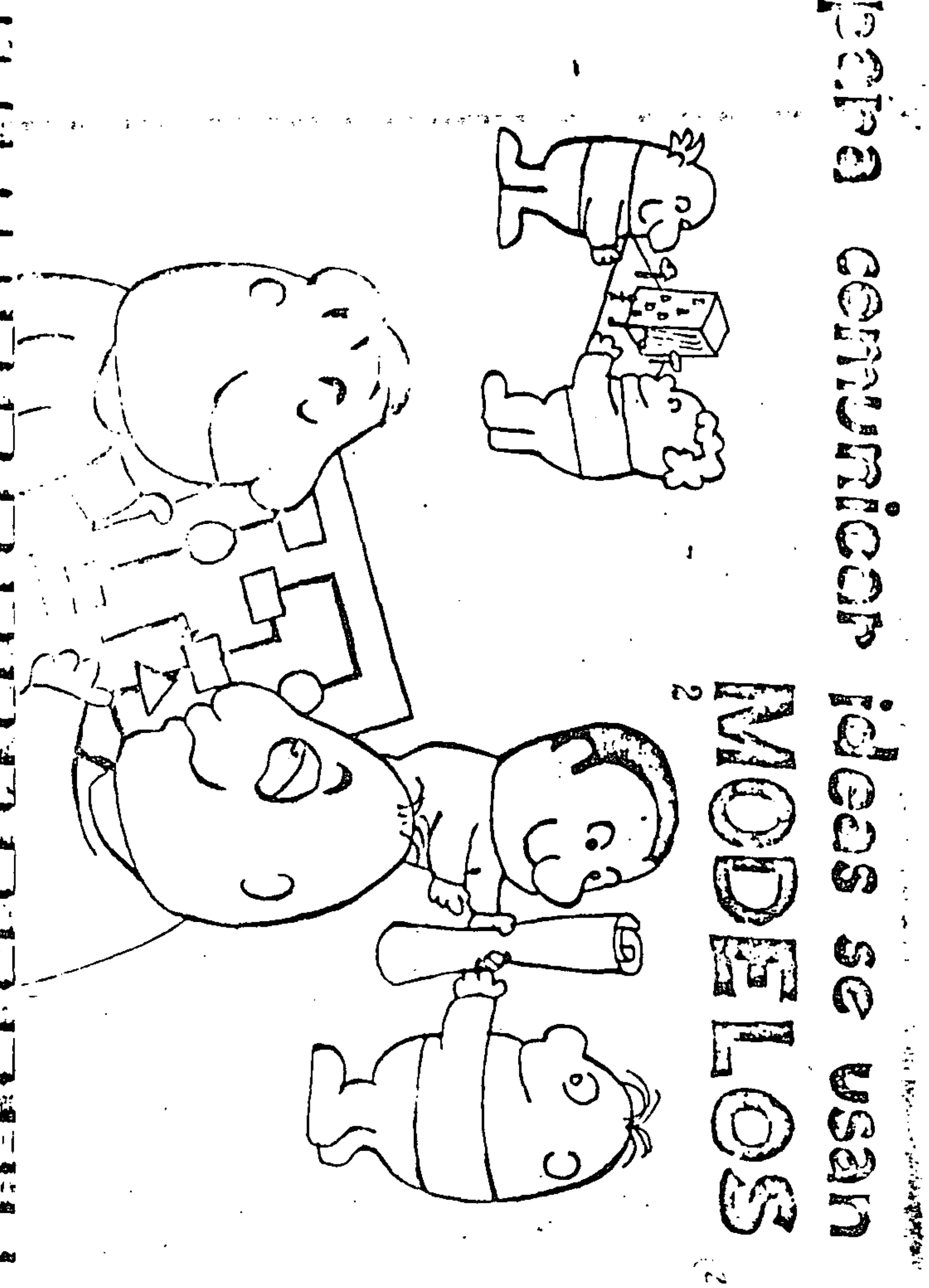
10.1 The Structure Chart

The Structure Chart represents the computer's view of the organization of and interaction between system components.

Structure Chart notation is used to represent:

MODULES | MODULE CALLS | MODULE COMMUNICATIONS





Examples of modules:

In PL/I

PROCEDURE

In COBOL

PROGRAM, SUBPROGRAM, SECTION, PARAGRAPH

In FORTRAN

SUBROUTINE, FUNCTION

etc., etc.

Definition of a Module

A module has four basic attributes:

What:

1. INPUT (what it gets from its invoker(s))

   OUTPUT (what it gives back to its invoker(s))

2. FUNCTION (*what* it does to its input to produce its output)

How:

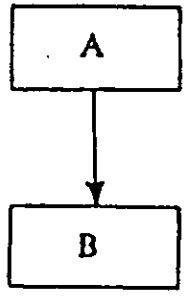
3. MECHANICS (*how* it does its function)

4. INTERNAL DATA (its own private workspace — that it alone refers to)

In addition, it

- has a name, by which it can be referred to as a whole
- can use or be used by other modules, eg. by a CALL
- usually sits in one place
- may have other vendor-specific or language specific characteristics

Representation of module calls



BOSS (caller)

SUBORDINATE (called)

- This means:

  A calls (invokes) B

  B does its thing

  B returns control to A (immediately after calling point)

- In other words, ↓ shows a normal subroutine call.

  (The direction of the arrow shows who calls whom.)

- A may contain

  1 'CALL B' statement

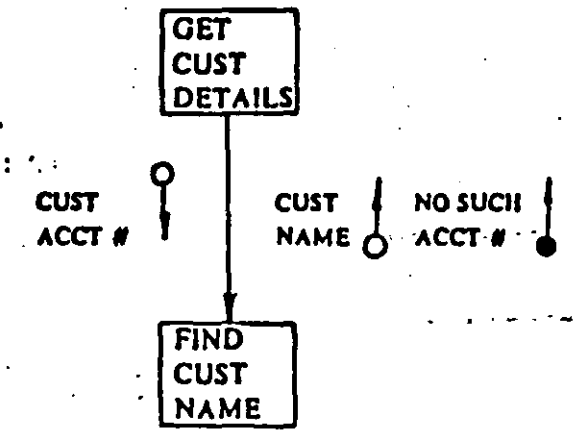
  10 'CALL B' statements

  23 'CALL B' statements

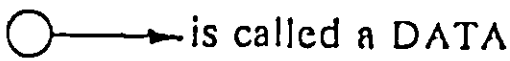
  although there is only one ↓ shown.

- At run time, A may not actually call B at all!

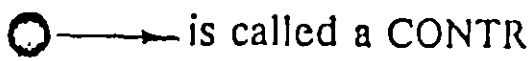
Module communication



- Now, not only does GET CUST DETAILS call FIND CUST NAME, but...
  - → GET CUST DETAILS sends *data* CUST ACCT # to FIND CUST NAME
  - → FIND CUST NAME returns *data* CUST NAME to GET CUST DETAILS
  - → FIND CUST NAME returns *control information* NO SUCH ACCT # to GET CUST DETAILS

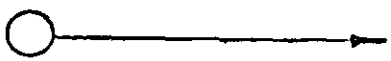


is called a DATA COUPLE



is called a CONTROL COUPLE

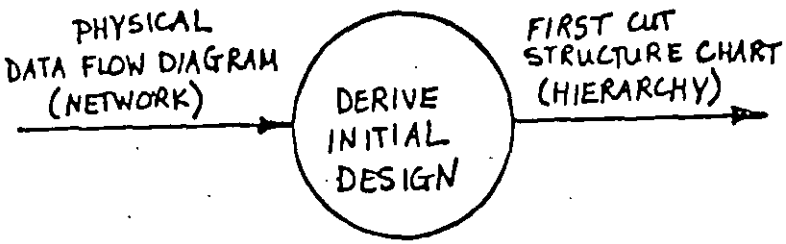
The arrow shows



SENDER RECEIVER

of the information

11.1 Turning Networks into Hierarchies



The derivation strategies are a set of techniques that enable us to generate a reasonably good design *quickly*.

The techniques are:

- Transform Analysis
- Transaction Analysis
- Top-Down Factoring

What Kind of Hierarchy Do We Want?

We want our systems to be

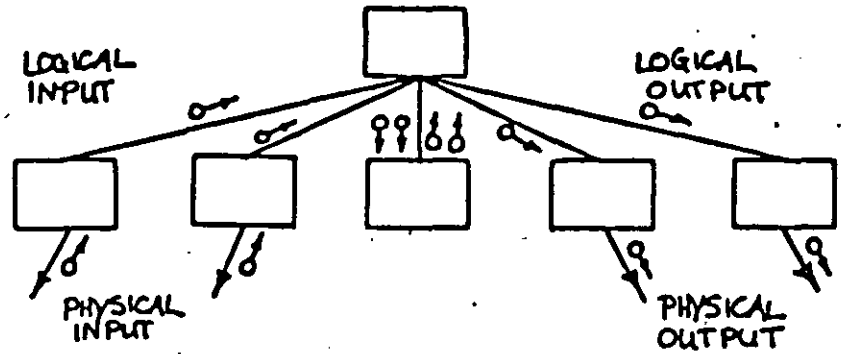
- easy to understand
- easy to maintain

Therefore, we want a hierarchy that:

1. is easy to understand
2. minimizes the effect of a given change on all modules
3. forces each module to assume its fair share of vulnerability to change.

And that preserves the simplicity of the Data Flow Diagram Model produced during analysis.

11.2 The Balanced Hierarchy

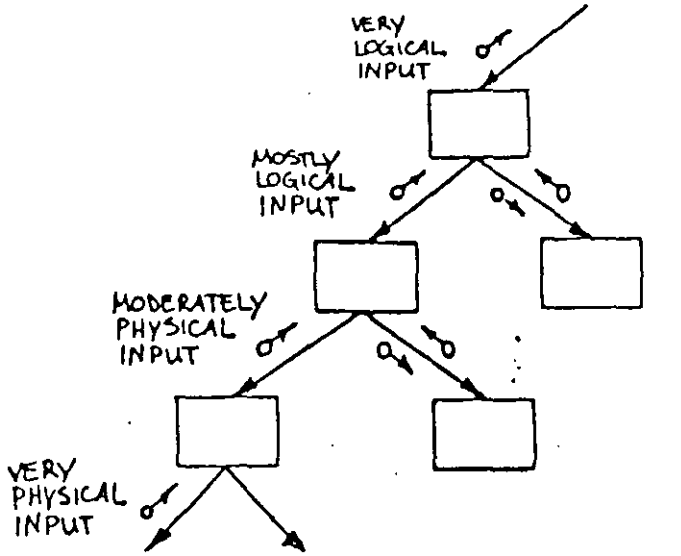


In a balanced hierarchy, the top module supervises the transformation of logical input data into logical output data.

To do this, a balanced executive module may call upon several kinds of subordinate hierarchies:

- afferent subhierarchies
- central transform subhierarchies
- efferent subhierarchies

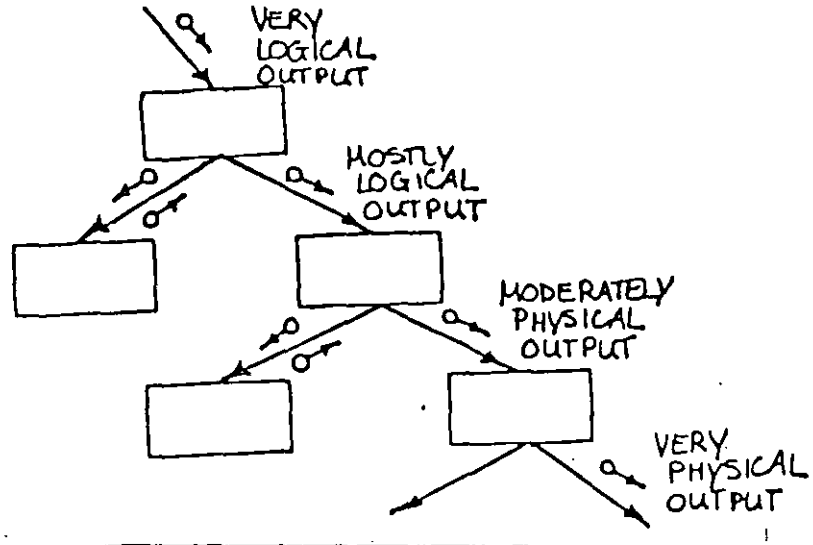
Afferent Subhierarchies



The job of an afferent subhierarchy is to refine physical input data into logical input data

The data primarily flows "uphill," *and becomes more logical with each upward step.*

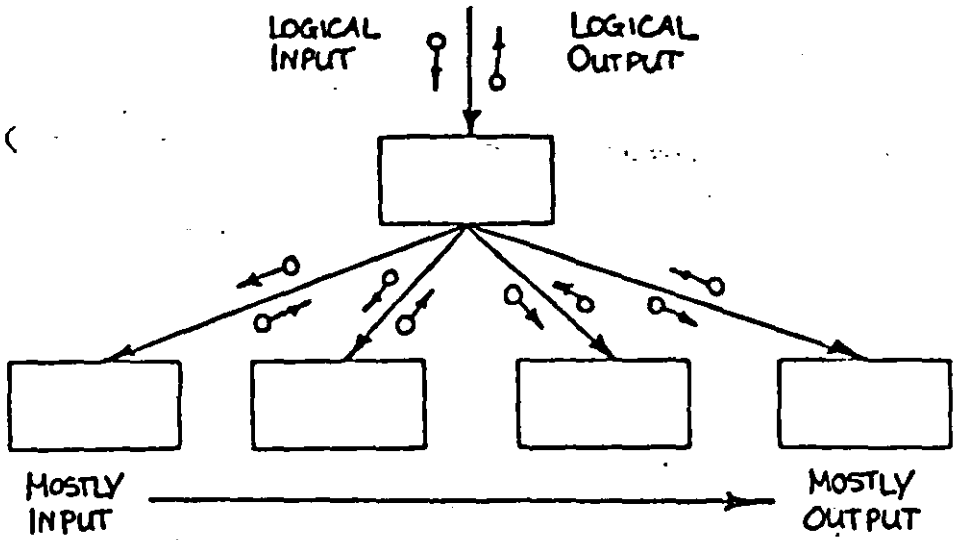
Efferent Subhierarchies



Efferent subhierarchies package logical output data into physical output data.

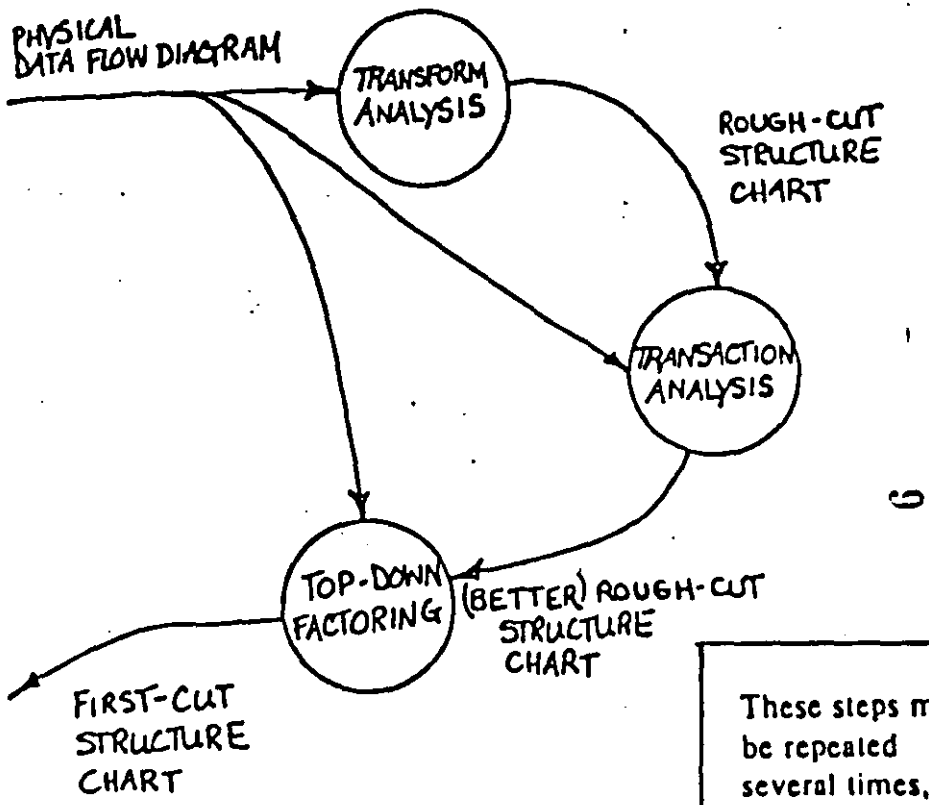
Here, the data flows "downhill," and becomes more and more implementation-dependent as it goes.

Central Transform Subhierarchies



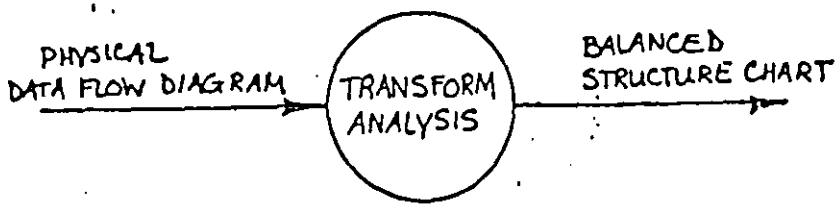
The central transform performs the functions after which the whole system is named: it tranforms logical input data into logical output data.

11.3 The Derivation Procedure



These steps may be repeated several times, until we get a satisfying first-cu design.

11.4 Transform Analysis



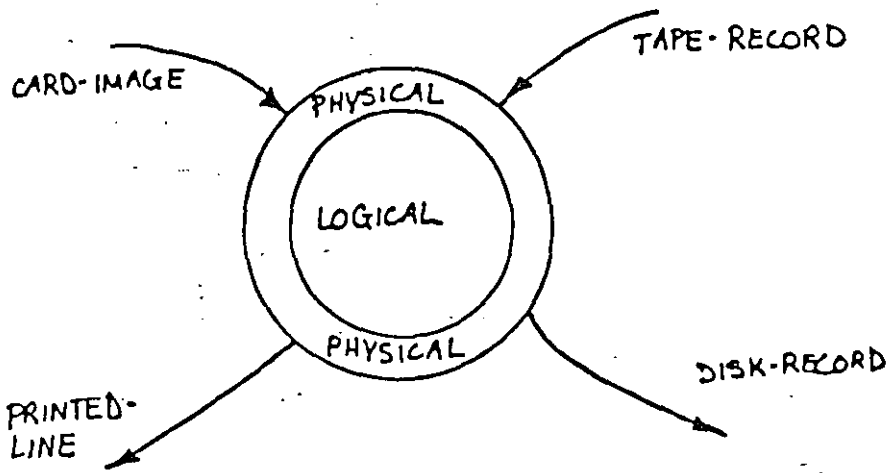
Transform Analysis is a technique for establishing a well-balanced hierarchy based upon a physical Data Flow Diagram.

The idea behind Transform Analysis is to manipulate the Data Flow Diagram such that the logical processes end up as central transform modules, and the physical processes become modules in the afferent and efferent subhierarchies.

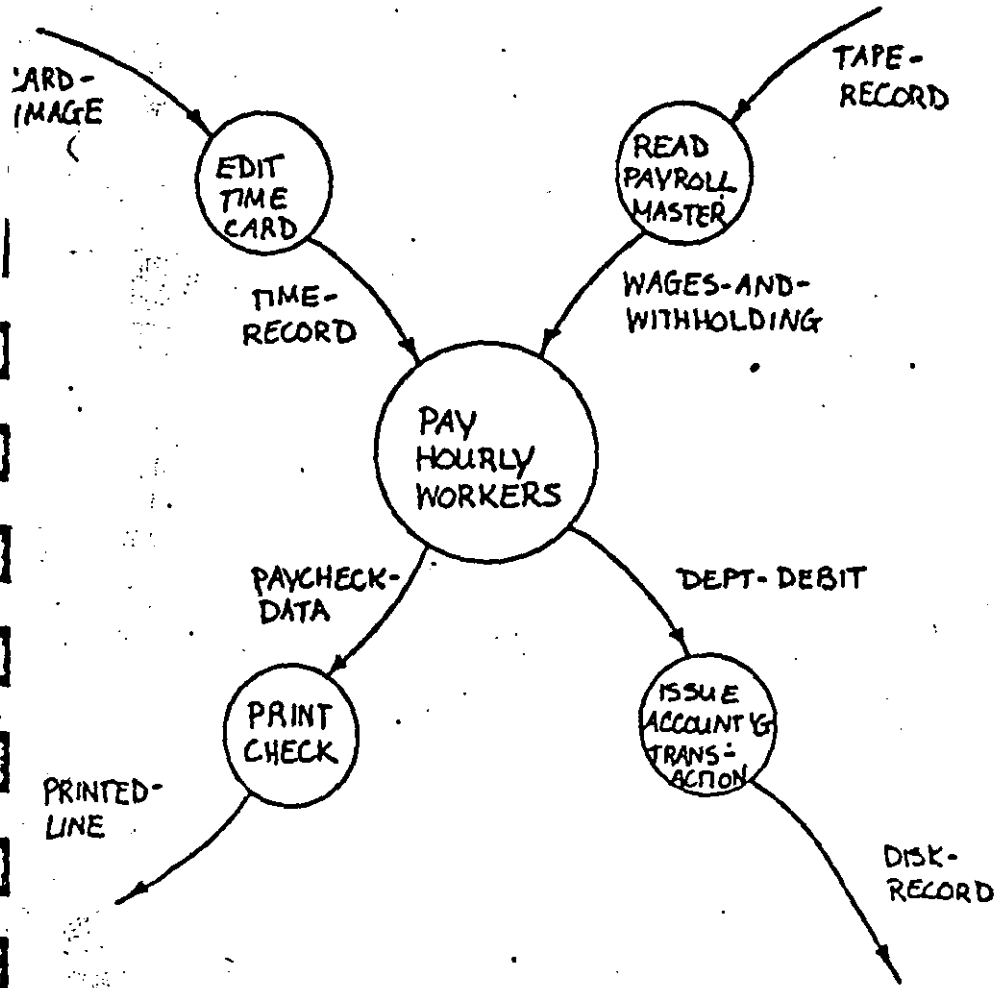
The Physical Data Flow Diagram

Transform Analysis assumes that the physical Data Flow Diagram consists of two parts:

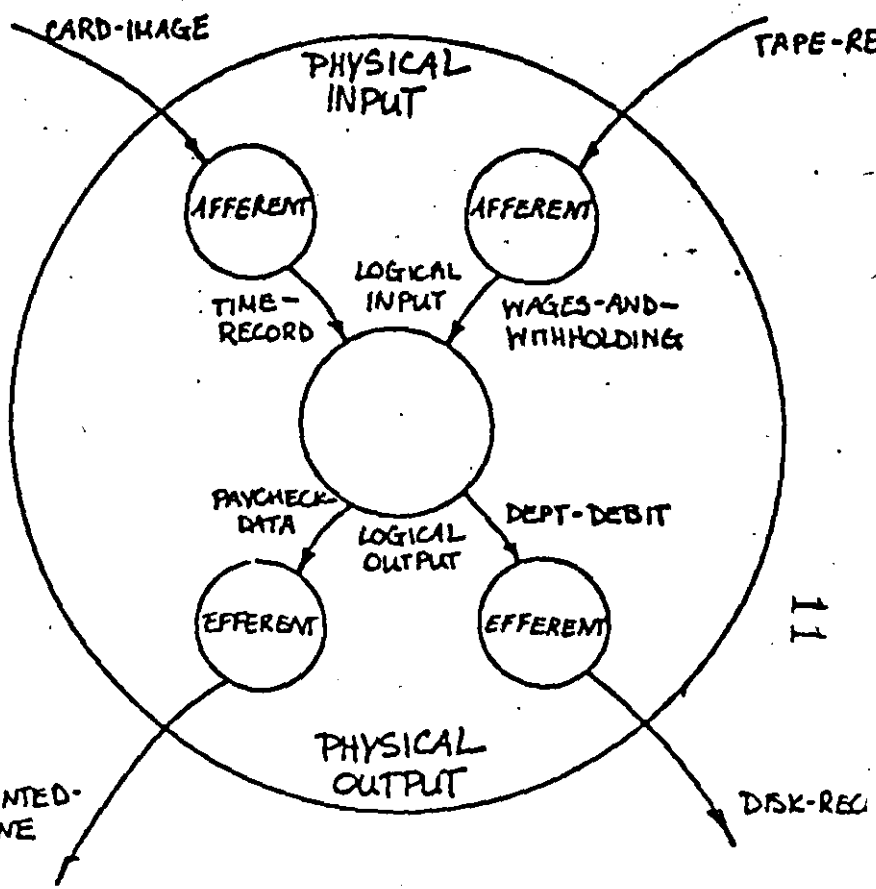
- a logical "core" from the tailored new logical model.
- a surrounding "ring" of physical processes added to form interfaces with the outside world and the data stores.



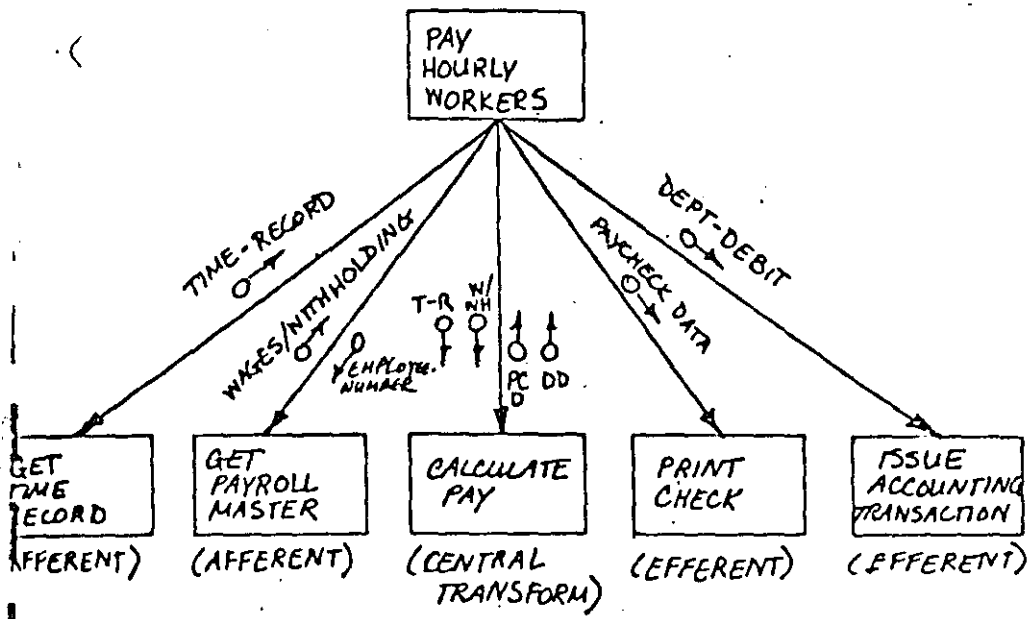
An Example of Transform Analysis



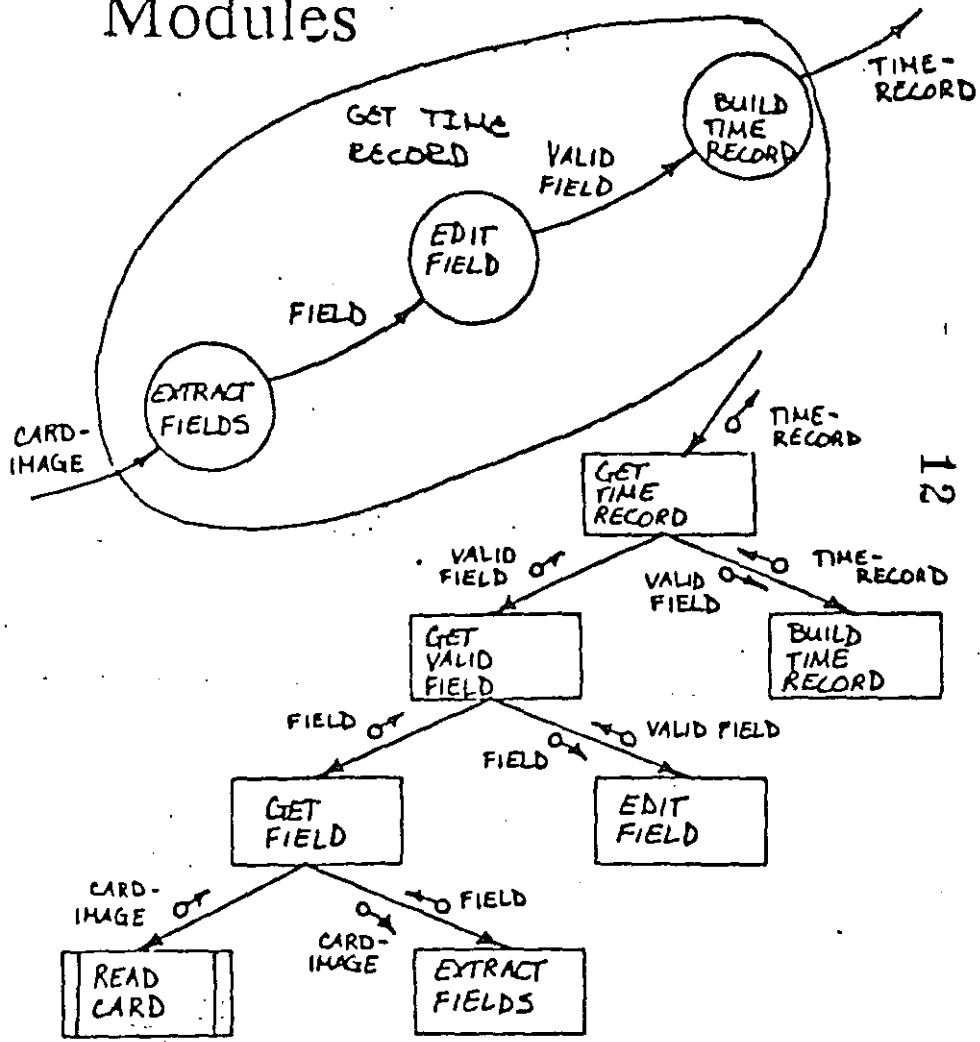
Identifying the Central Transform



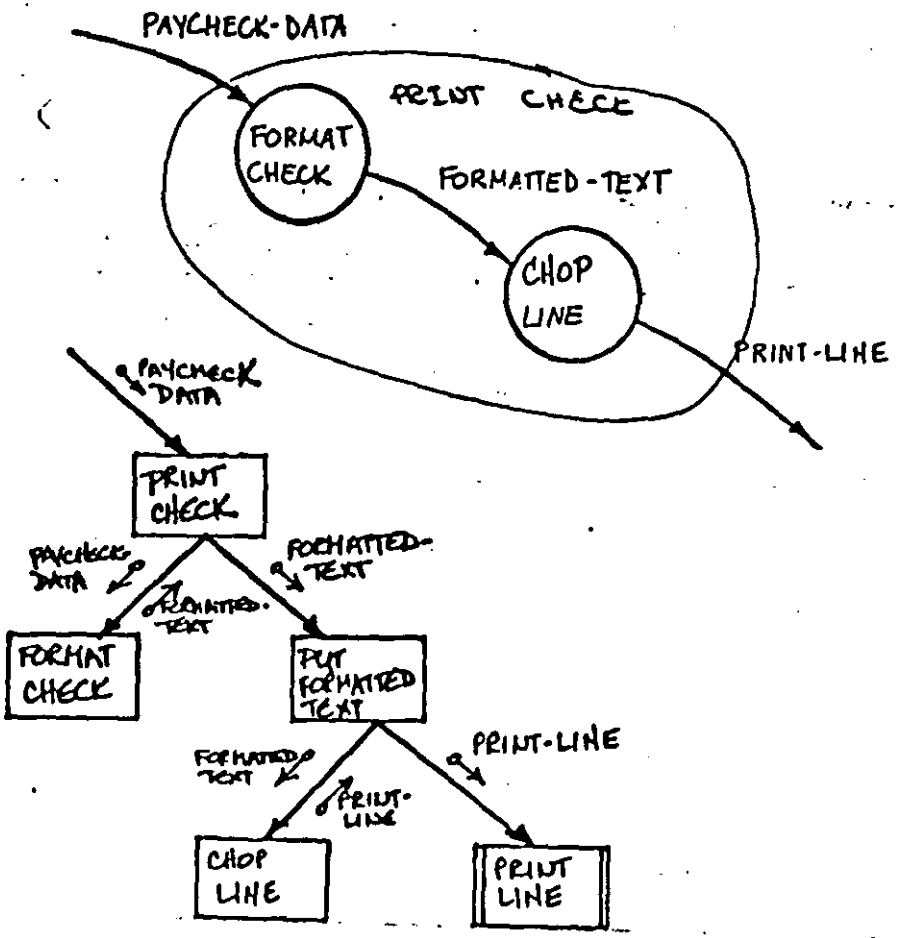
Deriving the Vice-Presidential Modules



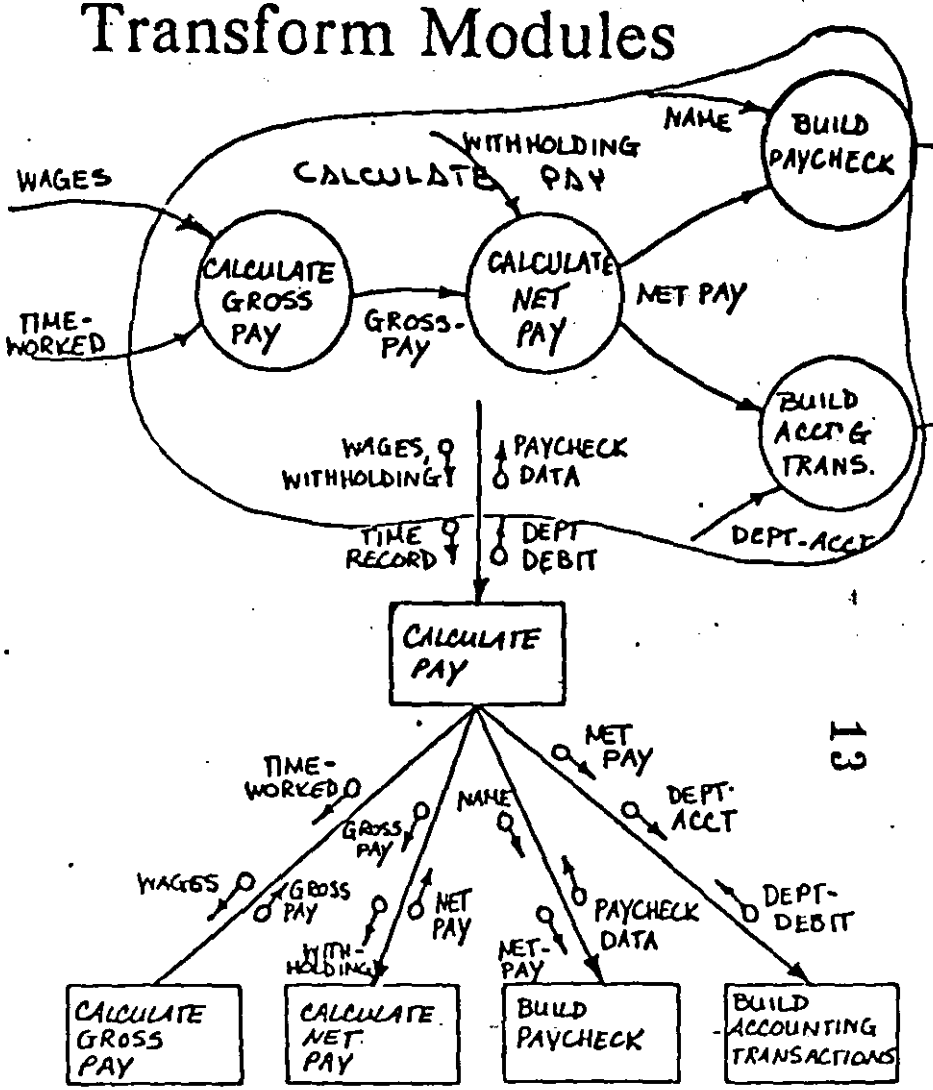
Deriving the Afferent Modules



Deriving the Efferent Modules



Deriving the Central Transform Modules



The Transform Analysis Procedure

1. Locate the points where the afferent and efferent dataflows are *most logical.*
2. Mark the logical core, or central transform.
3. Mark each afferent and efferent stream of processes in the physical ring.
4. Elevate the central transform and hire a boss for it.
5. Allocate Vice-Presidential modules for:
   - each afferent stream
   - the central transform
   - each efferent stream
6. Allocate lower-level modules for:
   - the bubbles dangling on each afferent and efferent stream.
   - the bubbles within the central transform.

Notes on Transform Analysis

- Don't worry about not choosing *exactly* the correct central transform. The refinement techniques will take care of any problems.
- Transform analysis really does not help us to derive the central transform modules. Here, we rely on the two remaining derivative strategies.

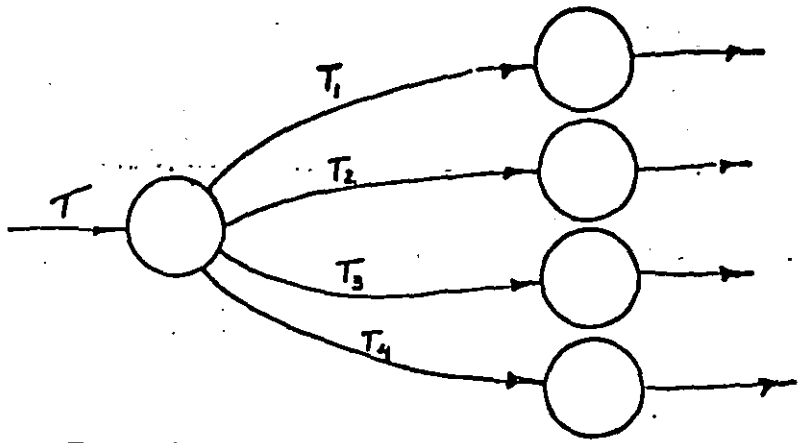
11.5 Transaction Analysis

For our purposes, a **transaction** is any dataflow that:

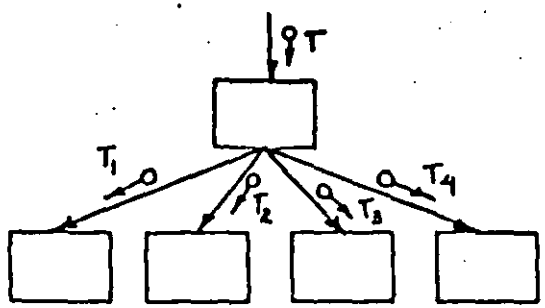
- comes in different flavors
- contains an identifying data element to tell us what flavor it is
- wants to have different actions taken depending on what flavor it is.

The Transaction Analysis Procedure

When you see this on a DFD:



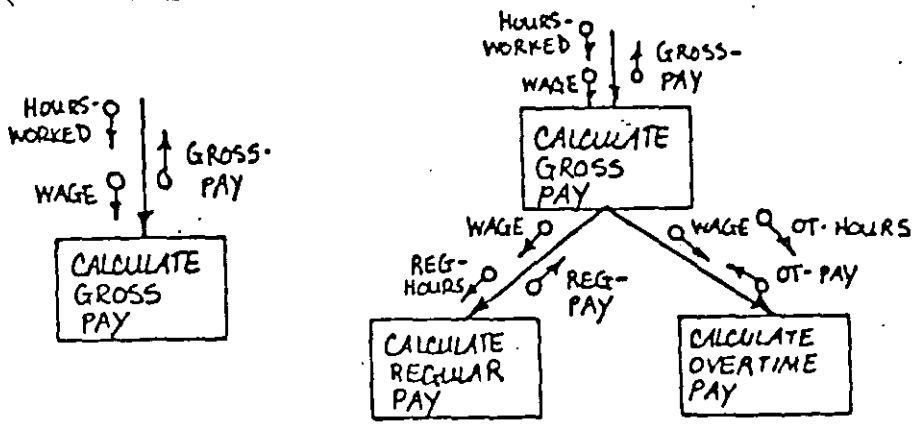
Do this on the structure chart:



Note: Transaction centers can occur in the afferent and efferent subhierarchies as well as the central transform.

11.6 Top-Down Factoring

Factoring simply means dividing a module into a manager module and the subordinates it calls upon to do work.



Factoring is a vestige of Structured Design's revered ancestors:

- Top-Down Design
- Step-Wise Refinement

Factor Forever (Almost)

During the early stages of design, we should factor the structure chart until every module is either a manager or a worker:

- A manager makes decisions, calls subordinates, and hands data to its boss.
- A worker performs calculations, moves text, or does other "processing."

16

This much factoring will lead to ridiculously small modules. We'll take care of that later. For now, we *want* that kind of detail, because it helps us to identify reusable modules.

(6)

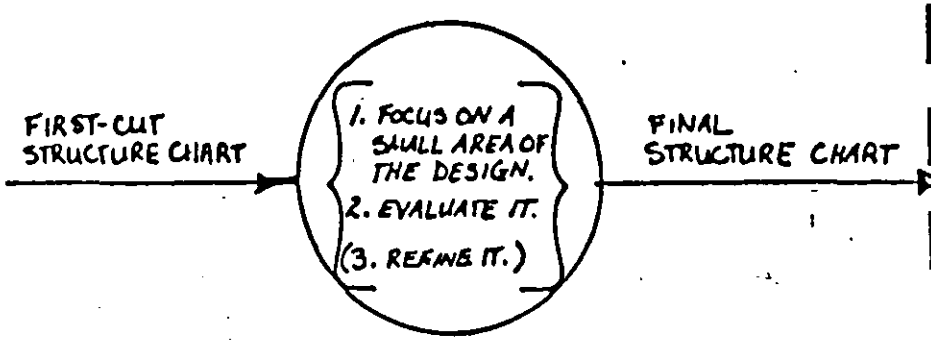
11.7 Evaluation of Overall Quality

Transform and transaction analysis get the design off to a great start. We can see this is best by comparing the first cut design we get from using derivative strategies to designs started with other techniques.

Other techniques like:

- input process output design
- mainline design
- blind intuition

12.1 Goals for Evaluation and Refinement



17

Let's restate our overall goals:

1. To make the automated portion of the system as understandable as possible.
2. To minimize the number of modules affected by a given change.
3. To minimize the area of the structure chart affected by a given change.
4. To establish modules that assume their fair share of vulnerability to change.

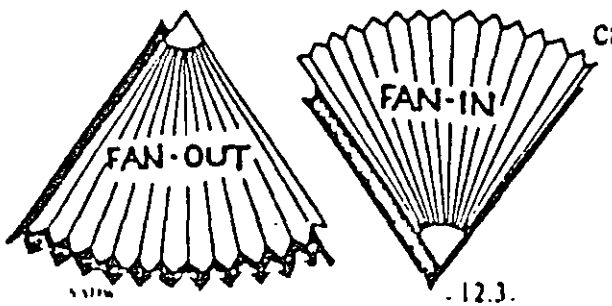
(17)

12.2 Module Complexity

> We advocate factoring systems into very simple modules because we want each module to be as understandable and reusable as possible.

There are several evaluation criteria that help us to determine if we have factored sufficiently:

- Module-Size: How many lines of pseudocode?
- Fan-Out: How many modules does it call?
- Fan-In: How many modules call it?



Module Size

> Like minispecs, modules typically should not exceed one page of pseudocode.

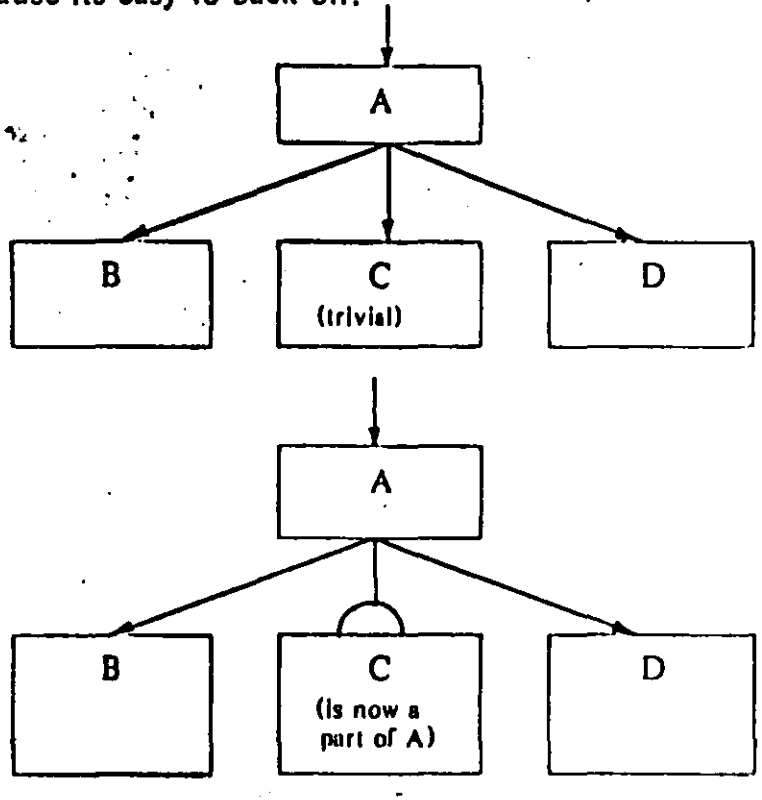
The key concept here is that we are *much* less able to understand a module if we have to turn pages to see it all.

Quite often, we choose to divide a module into pieces much smaller than a page if we can factor out a commonly-called function.

Module Size

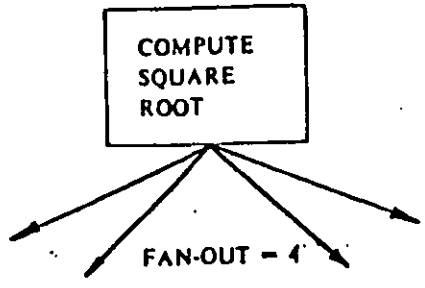
IT IS BETTER TO FACTOR TOO FAR THAN NOT FAR ENOUGH

because its easy to back off:



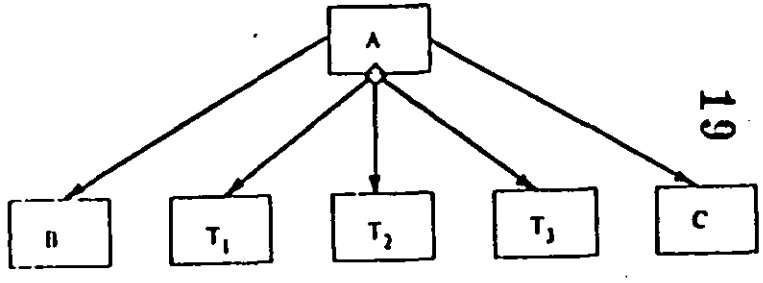
Fan-Out

Fan-Out is the number of direct subordinates to a module.



Fan-Out shouldn't be more than 7 ± 2

Note: Transaction centers count as one

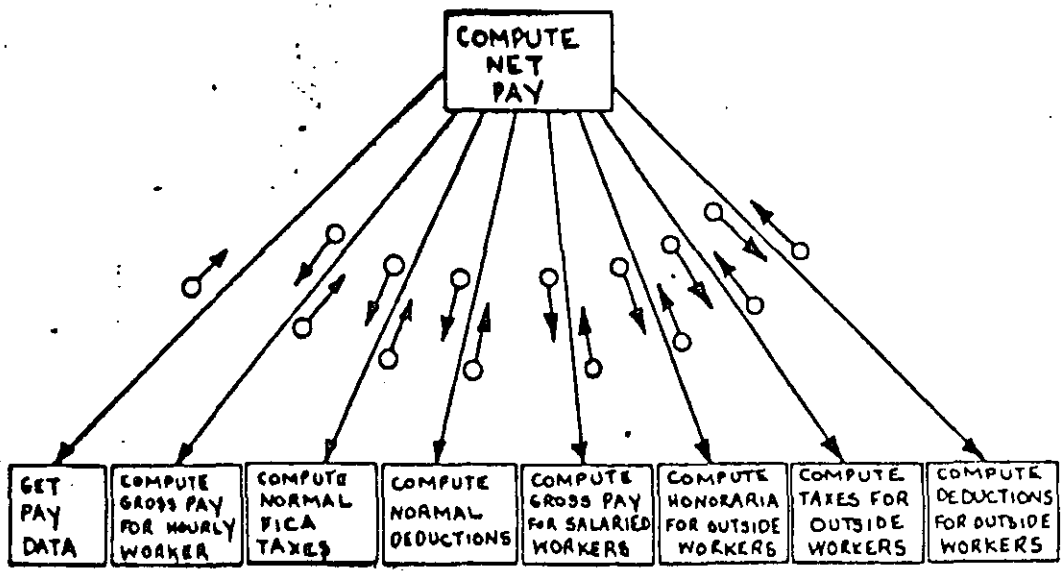


FAN-OUT OF A = 3

Fan-Out

A module with high fan-out is usually a pain to code

— *and worse to maintain!*



Pancakes are a symptom of missing intermediate levels.

What is coupling?

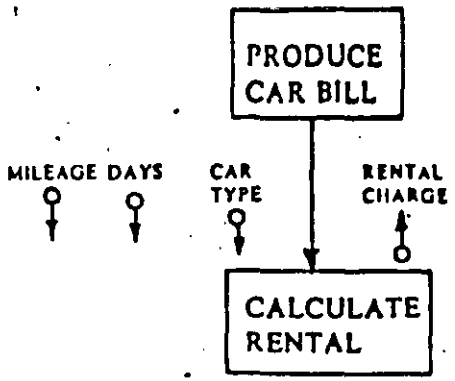
Coupling is the measure of the interdependence among the modules of a hierarchy.

> Just as we partitioned our Data Flow Diagrams to minimize interfaces, we shall factor our structure charts to minimize coupling.

Why? Because having lots of coupling:

1. makes it more difficult to understand one module without looking at others.
2. increases the number of modules that will be affected by a change to a well-traveled data item.
3. increases the probability that an individual, highly-coupled module will be affected by a given change.

Data Coupling



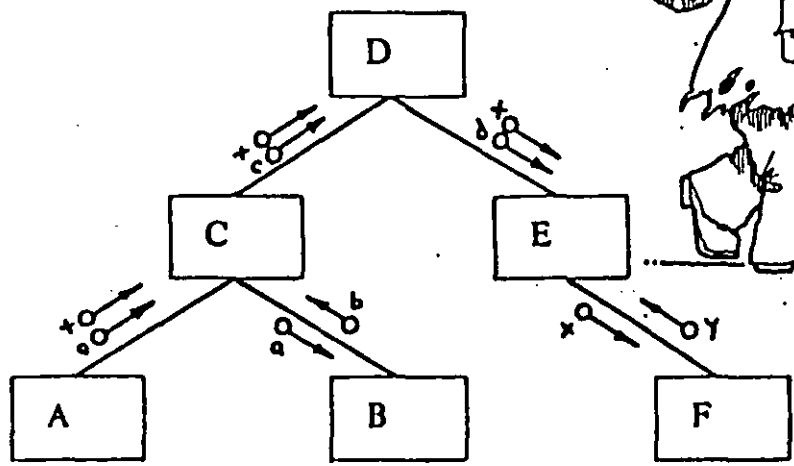
Data coupling is merely the necessary data communication between modules.

- Although it's unavoidable, keep it to a minimum — that is, keep your interfaces narrow.

DATA
STAMP
CONTROL
COMMON
CONTENT

Data Coupling

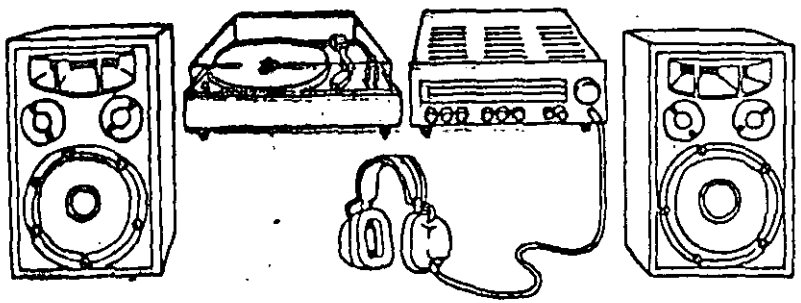
- Also, beware of "tramp data," which shuffles aimlessly around the system unused and unwanted by most of the modules it passes through



It's likely that modules C, D and E do not use x, since its name does not change as it passes through them.

What is coupling?

Coupling is the measure of the **interdependence** among the modules of a **hierarchy.**



VS.

12.4.1 Types of coupling

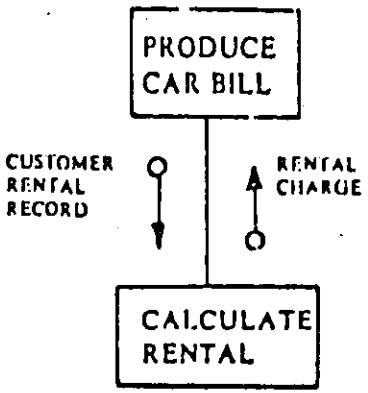
1	DATA	BEST
2	STAMP	
3	CONTROL	
4	COMMON	
5	CONTENT	WORST

Breadth of Coupling

22

The less coupling there is between any two modules, the more independent those modules are and the better the design is.

Stamp Coupling



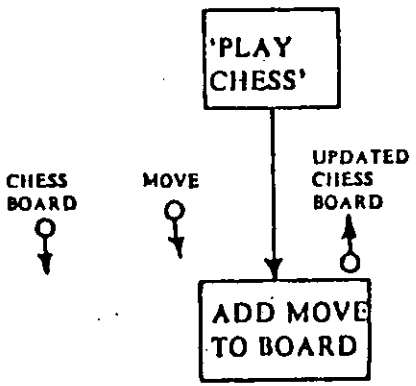
Two modules are stamp coupled if they refer to the same (non-global) data structure.

(A data structure is a composite of data elements, e.g. an array, a record, etc.)

Stamp Coupling

- An isolated change in the structure of CUSTOMER RECORD will affect all modules that are stamp coupled via CUSTOMER RECORD

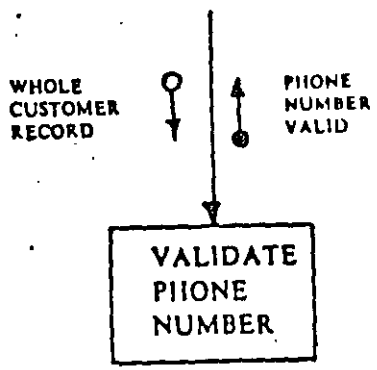
- However, using a good, *natural* data structure is the way to overcome high *data* coupling, e.g.



Stamp Coupling

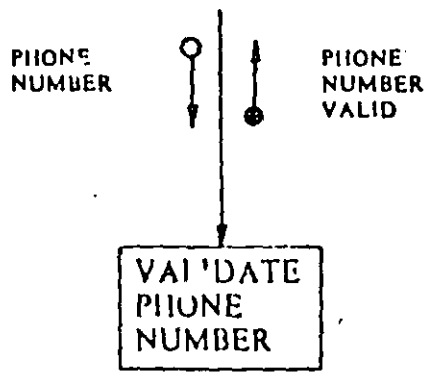
A warning about data structures:

Danger arises when a module is passed a whole data structure but only needs a part of it, e.g.



Stamp Coupling

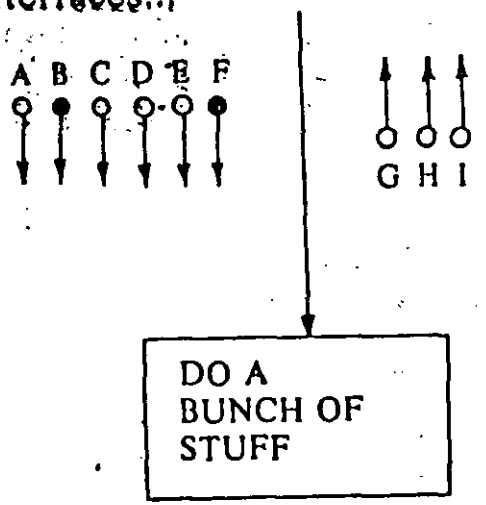
- A change in CUSTOMER RECORD format may affect VALIDATE PHONE NUMBER even if the change has nothing to do with phone numbers
- Also VALIDATE PHONE NUMBER needs data it does nothing with, making it a not very useful module
- So, starve your modules



Stamp Coupling

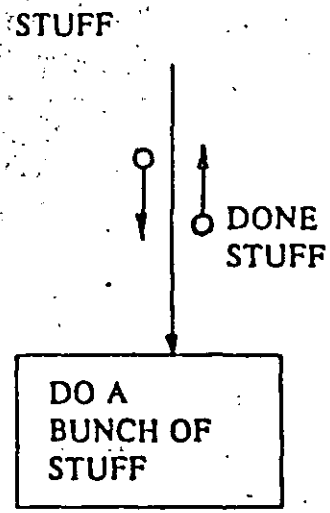
Another danger related to stamp coupling is called *bundling*.

Sometimes people with messy module interfaces...



...will bundle together unrelated data and/or control into an artificial data structure, e.g....

Stamp Coupling

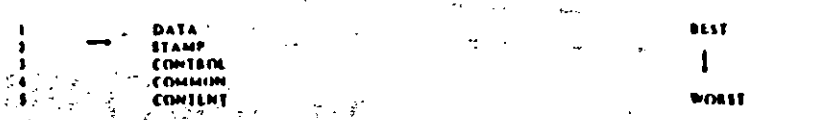


STUFF = A+B+C+D+E

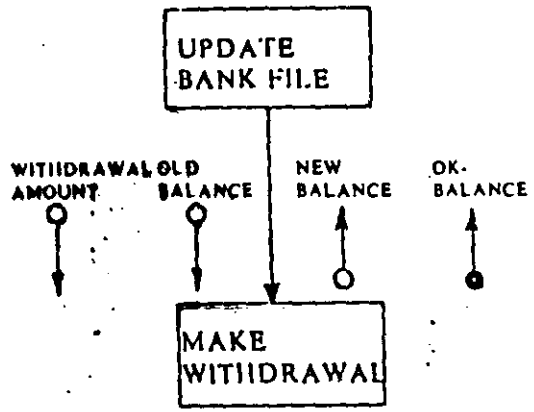
DONE
STUFF = G+H+I

This designer thinks he has improved his coupling.

He may be fooling himself but he's *not* fooling the GODS OF SYSTEM DESIGN!



Control coupling

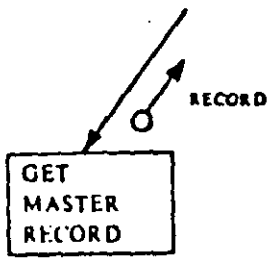


Two modules are control coupled if they communicate using at least one control couple.

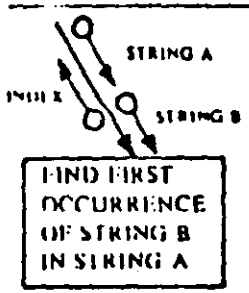
- This is undesirable since one or both modules will not be a black box.

Hybrid Coupling

Hybrid coupling is the "shoe-horning" of two or more meanings into the same item of data.



RECORD – MASTER FILE RECORD
HEX 'FF' IF END-OF-FILE



INDEX – POSITION OF STRING B
0 IF STRING B NOT FOUND

Common Coupling

> Two (or more) modules are common coupled if they share data that was not passed along a normal module call (e.g. data held in a common area).

Common coupling can happen at two points during system development:

- **Design-Time:** if the designer *deliberately decides* to allow modules to share data in common
- **Coding-Time:** if the programming language makes common data a regrettable necessity.

Common Coupling at Coding-Time

The causes:

- COBOL- Data Division
- FORTRAN - Common Blocks
- PL/I - Global Data, External Storage

The Consequences:

- When a fault causes a shared data item to be clobbered, the "usual suspects" include just about every module in the system. This takes the joy out of 2 A.M. phone calls.
- Maintenance programmers may take advantage of the global accessability of data to add undesigned, undocumented, and sometimes unpredictable new features to the system.

The Solutions:

- Don't code with common data if you don't have to.
- At least, *design* without it, even if you must code with it.
- Teach structured design to your maintenance staff.

Common Coupling at Design-Time

The damage here is often fatal.

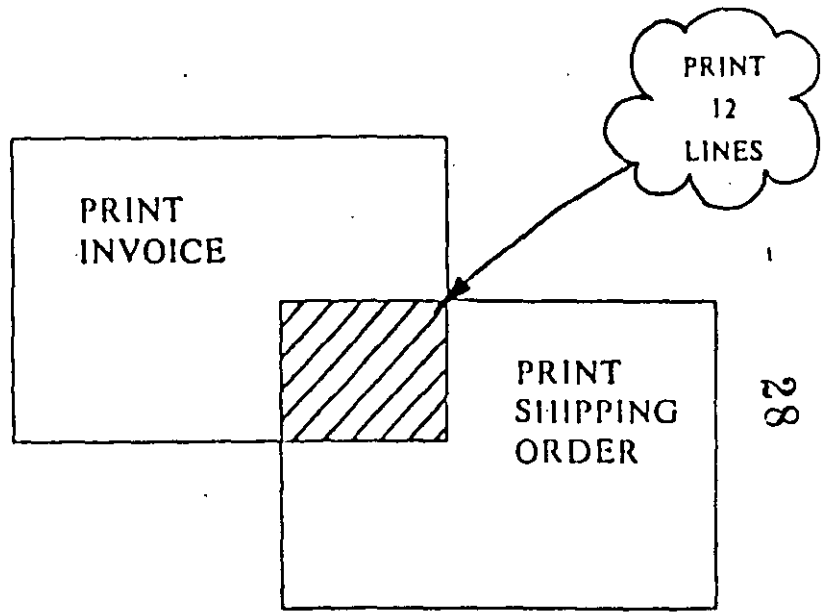
Once the designer realizes that "all the data is right there in the Data Division," he'll be tempted not to draw the couples on the structure chart.

> You cannot evaluate and refine a design without knowing the communication among modules.

The Solution: Don't design unrestricted access to global data into your system. Ever.

Content Coupling

- SHARED CODE
- SHARED DATA



28

Content Coupling

This is CLOSE ENCOUPLING OF THE WORST KIND!

It occurs when:

1. One module alters a statement in another module.
2. One module refers to (or changes!) data contained inside another module.
3. One module branches into another.
4. Two modules share the same literals.

1, 2 and 3 are usually called "pathological," because they're very sick indeed.

4 is more subtle but also dangerous.

Types of Internal Cohesion

There are six levels of internal cohesion, but they fall into three main types:

- *Data-Oriented Cohesion* is typical of structure charts derived from Data Flow Diagrams.
- *Time-Oriented Cohesion* is commonly found on structure charts derived from flowcharts, or drawn by someone who concentrates on the sequence of execution.
- *Suite-Oriented Cohesion* can show up anywhere. It happens when the designer clusters several modules together into a "suite of functions."

Levels of Internal Cohesion

Best

Data-Oriented Levels: Very Good

- Sequential
- Communicational

Time-Oriented Levels: Mediocre

- Procedural
- Temporal

Suite-Oriented Levels: Truly Bad

- Logical
- Coincidental

Worst

Sequential Cohesion

Object of this exercise:

To paint our car a different color

CLEAN CAR BODY ①	FILL IN HOLES IN CAR BODY ②
SAND CAR BODY ③	④ ?

Sequential Cohesion

A sequentially cohesive module is one whose elements are involved in tasks where output data from one task serves as input data to the next.

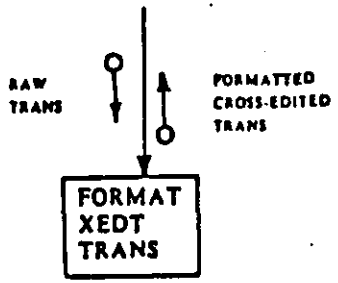
module format-xedit-trans

uses raw-trans

format raw-trans
cross-edit "result"

returns formatted cross-edited trans

endmodule



Communicational Cohesion

FIND TITLE OF BOOK	FIND PRICE OF BOOK
FIND CODE # OF BOOK	?

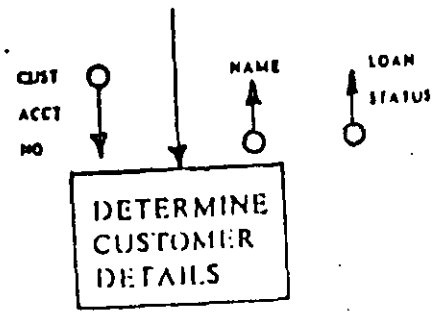
Communicational Cohesion

A communicationally cohesive module is one whose elements contribute to different tasks, but each task refers to the same input or output parameters.

```
module determine-customer-details
    uses cust-acct-no

    determine customer name
    determine customer loan status

    return name, loan-status
endmodule
```



Communicational Cohesion

- Communicational cohesion is fairly strong because the grouping of functions within one module has some relationship to the problem rather than the implementation of the solution.

- But it may cause trouble, e.g.

  we may want a customer's name without finding his loan status, or

  because of a temptation to share code.

- Splitting a communicationally cohesive module into separate modules simplifies coupling and improves cohesion.

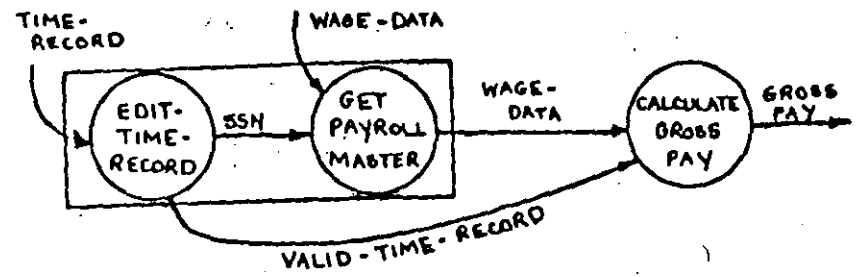
DATA-ORIENTED COHESION

Sequential and Communicational Cohesion

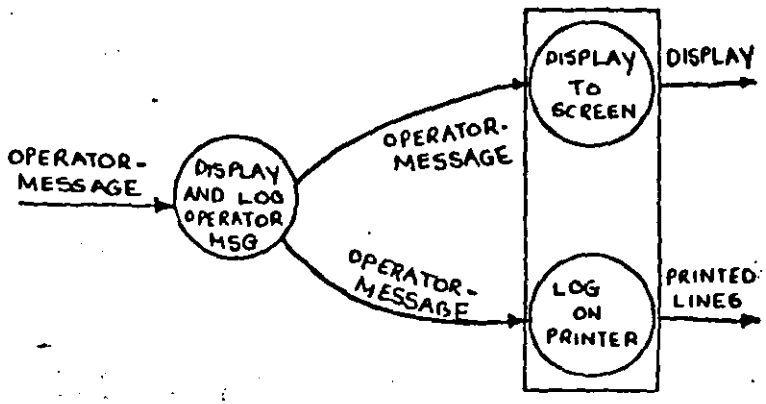
- Both types of modules are strongly connected because their elements are grouped from a data rather than a processor point of view.

Data-Oriented Cohesion

SEQUENTIAL



COMMUNICATIONAL



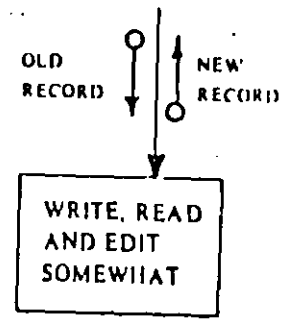
Procedural Cohesion

CLEAN UTENSILS FROM PREVIOUS MEAL (1)	PREPARE TURKEY FOR ROASTING (2)
MAKE PHONE CALL (3)	TAKE SHOWER (4)
CHOP VEGETABLES (5)	? (6)

Procedural Cohesion

- A procedurally cohesive module is one in which control flows from one element to the next.
- However, data does not necessarily flow from one element to the next — that's sequential cohesion.

```
module write-read-and-edit-somewhat
    uses old-record
    write old-record
    read old-record
    edit some of the fields in the new record
    return new-record
endmodule
```



Temporal Cohesion

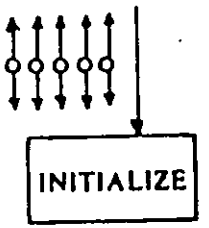
MOON	STARS
OWL	?

PUT OUT MILK BOTTLES	PUT OUT CAT
MAKE BED	?

Temporal Cohesion

A temporally cohesive module is one whose elements are related in time.

- Usually, however, these elements really "belong" to different functions.



TIME-ORIENTED COHESION

Temporal and Procedural Cohesion

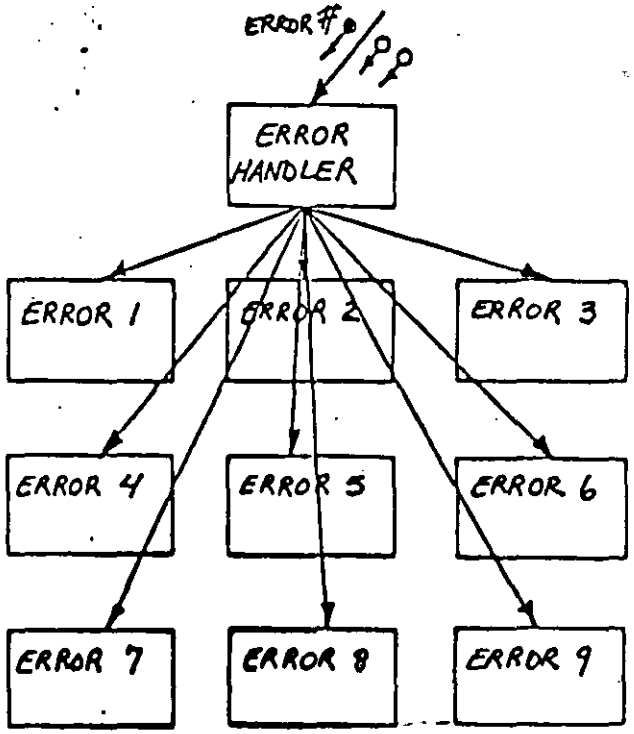
- Both types of modules can be created by taking a scissors to a flowchart.
- Such modules contain elements which, though only loosely related to one another, are often closely related to elements in other modules.
- So coupling is poor at worst, variable at best.

Logical Cohesion

GO BY CAR	GO BY TRAIN
GO BY BOAT	?

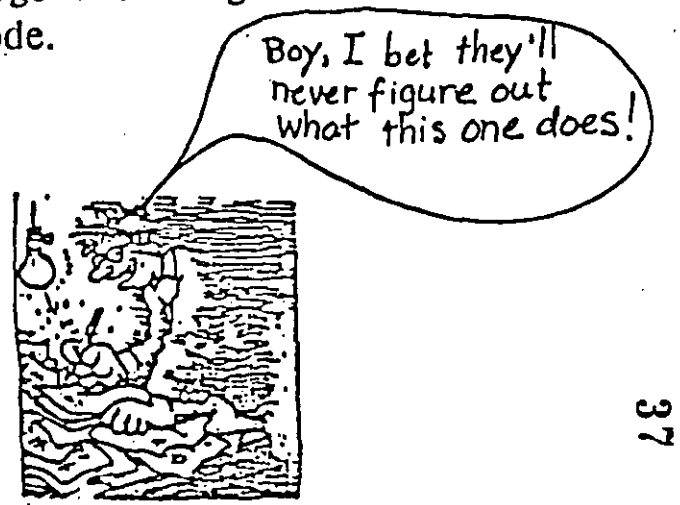
Logical Cohesion

A logically cohesive module is one whose elements *seem* to be involved in tasks of the same general category.



Logical Cohesion

- A "clever" programmer might set flags and switches which allow the module to meander sneakily through large intertwingled areas of shared code.



- It should really be called *ILLOGICAL COHESION*

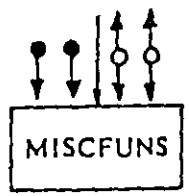
Coincidental Cohesion

PAINT HOUSE	DRINK COFFEE
PLAY BALL	?

Coincidental Cohesion

A coincidentally cohesive module has elements with no meaningful relationship to one another.

```
module miscfuns
    uses funcflag, operator-message
    updates matnum, acc
    if funcflag = 1
    then set matnum to 0 /* matnum is a matrix */.
        set acc to 1
    elself funcflag = 2 or funcflag = 3
    then rewind tape-B
        if funcflag = 2
        then print headings
        endif
    else display operator-message
    endif
endmodule
```



SUITE-ORIENTED COHESION

Coincidental and Logical Cohesion

- Both types of modules have an identity crisis

  because

  the boss always has to send down a completely artificial flag telling the module what to do today.

- Such modules violate the idea of independent black boxes and worsen coupling.

7.0 ANALISIS DE TRANSACCIONES

Una transacción es cualquier elemento de datos, control, señal, evento, o cambio de estado que causa, dispara o inicia alguna acción o secuencia de acciones.

En otras palabras, una transacción es una parte de datos que puede ser cualquiera de un número de tipos, cada tipo requiere diferente procesamiento.

La estrategia de análisis de transacciones, simplemente reconoce que los diagramas de datos de este tipo pueden mapearse a una estructura modular particular. El centro de transacciones de un sistema, debe ser capaz de:

1. Obtener (responder a) las transacciones en su forma mas primitiva.

2. Analizar cada transaccion para determinar su tipo.

3. Despachar dependiendo de la transacción.

4. Completar el proceso de cada transacción.

En su forma más factorizada, el centro de transacciones puede ser modularizado de la siguiente forma:

7.1 La estrategia.

1. Identificar los origenes de transacciones.

2. Especificar la organización de transacciones apropiada (la figura anterior es un buen modelo en general).

3. Identificar la transaccion y las acciones que la definen.

4. Note situaciones potenciales en que se pueden combinar módulos.

5. Para cada transacción o grupo cohesivo de transacciones, especificar un módulo de transacciones para procesarla completamente.

6. Para cada acción en una transacción, especificar un módulo de acción subordinado al correspondiente módulo de transacción.

7. Para cada paso detallado en un módulo de acción, especificar un módulo detallado subordinado al módulo de acción que lo necesite.

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

ARTICULOS CORRESPONDIENTES AL TEMA III DISEÑO ESTRUCTURADO

ING. ALEJANDRO ACOSTA COLORADO

MAYO, 1985

Module connection analysis - A tool for scheduling software debugging activities	Frederick M. Haney
Order and Discipline: Benefits of structured Techniques	David S. Iwahashi
Structured Systems Analysis: A Technique to Define Business Requirements	Kathleen S. Mendes
HIPO and integrated program design	J.F. Stay
Conversion of unstructured flow diagrams to structured form.	M.H. Williams and H.L. Ossher
The second Structured Revolution	Edward Yourdon
Composite design facilities of six Programming languages	G. J. Myers
A summary of progress toward proving program correctness	T.A. Linden
Improved software development through projet management	Robert B. Fireworker and Leonard J. Bogner, Jr.
Structured design	W.P. Stevens, G.L. Myers, and L.L. Constantine
Is this Really Necessary? A First Look at Design Techniques	Gregg Williams

Module connection analysis—A tool for scheduling software debugging activities

by FREDERICK M. HANEY

Xerox Corporation
El Segundo, California

INTRODUCTION

The largest challenge facing software engineers today is to find ways to deliver large systems on schedule. Past experience obviously indicates that this is not a well-understood problem. The development costs and schedules for many large systems have exceeded the most conservative, contingency-laden estimates that anyone dared to make. Why has this happened? There must be a plethora of explanations and excuses, but I think H. R. J. Grosch identified the common denominator in his article, "Why MAC, MIS and ABM will never fly."[1] Grosch's observation is essentially that for some large systems the problem to be solved and the system designed to solve it are in such constant flux that stability is never achieved. Even for some systems that are flying today, it is obvious that they came precariously close to this unstable, "critical mass" state.

It is my feeling that our most significant problem has been gross underestimation of the effort required to *change* (either for purposes of debugging or adding function) a large, complex system. Most existing systems spent several years in a state of gradual, painfully slow transition toward a releasable product. This transition was only partially anticipated and almost entirely unstructured; it was a time for putting out fires with little expectation about where the next one would occur.

The difficulties of stabilizing large systems are universal enough that our experience has resulted in several improved methods for estimating projects. Rules-of-thumb like "10 lines of code per man day" once sounded like extremely conservative allowance for the complexities of system integration and testing. J. D. Aron[2] has described a relatively elaborate technique for estimating total effort for large projects. Aron's technique is based on the estimated amount of code for a project and empirically observed distributions of various kinds of effort such as design, coding, module testing, etc. More recently Belady and Lehman described a mathematical model for the "meta-dynamics of systems in growth."[3] These schemes provide useful insights into the difficulties of designing and implementing large systems.

Even with these improved estimation techniques, however, we still face the threat of long periods of unstructured post-integration putting out of fires. We may know better how long this "final" debugging will take, but we are still at a loss to predict what resources will be required or what specific activities will take place. If we predict an 18 month period for "final testing," will management buy it? How can we peer into this hazy contingency portion of a schedule and predict in greater detail where bugs will occur, who will be needed to fix them, elapsed time between internal releases, etc.? Belady and Lehman suggest the need for a "micro-model" for system activities; i.e., a model based on internal, structural aspects of a system. This is essentially the objective of this paper. In the following sections, we will develop a very simple, but useful, technique for modeling the "stabilization" of a large system as a function of its internal structure.

The concrete result described in this paper is a simple matrix formula which serves as a useful *model* for the "rippling" effect of changes in a system. The real emphasis is on the use of the formula as a model; i.e., as an aid to understanding. The formula can certainly be used to obtain numeric estimates for specific systems, but its greater value is that it helps to *explain*, in terms of system structure and complexity, why the process of changing a system is generally more involved than our intuition leads us to believe.

The technique described here, called *Module Connection Analysis*, is based on the idea that every module pair (may be replaced by subsystem, component, or any other classification) of a system has a finite (possibly 0)

probability that a change in one module will necessitate a change in any other module. By interpreting these probabilities and applying elementary matrix algebra, we can derive formulae for estimating the total number of "changes" required to stabilize a system and the staging of internal releases. The total number of changes, by module, is given by

$$A \times (I-P)^{-1},$$

where A is a row vector representing the initial changes per module, P is a matrix such that Pij is the probability that a change in module i necessitates a change in module j, and I is the $n \times n$ identity matrix. The number of changes required for each "internal release" is given by AP^k, $K=0, 1, \ldots$, or by

$$A \times (I-P)^{-1} \times Uk, \qquad k=1, 2, \ldots n,$$
$$Uk = (0, \ldots, \underset{\uparrow}{1}, \ldots 0)$$
$$k \text{ th element}$$

depending upon the release strategy. The derivations of these formulae are presented in the following section.

Module connection analysis is useful primarily as a tool for augmenting a designer's quantitative understanding of his problem. It produces quantitative estimates of the effects of module interconnections, an area in which intuitive judgment is generally inadequate.

THEORY OF MODULE CONNECTIONS

As a basis for our analysis, we postulate several characteristics of a system:

- A system is hierarchical in structure. It may consist of subsystems, which contain components, which contain modules or it may be completely general having n different levels of composition where an object at any level is composed of objects at the next lower level.
- At any level of the hierarchy, there may be some interdependence between any two parts of the system.
- If we view a system as a collection of modules (or, whatever object resides at the lowest hierarchical level), then the various interdependencies are manifested in terms of dependencies between all pairs of modules.

By dependence here, we mean that a change in one module may necessitate a change in the other. The fundamental axiom of module connection analysis is that intermodule connections are the essential culprit in elongated schedules. That a change in one module creates the necessity for changes in other modules, and these changes create others, and so on. Later, we will see that perfectly harmless-looking assumptions lead easily to sums like hundreds of changes required as a result of a single initial change. (The notions of hierarchy, interconnection, etc., used here are described at length in Reference 4.)

If we assume that a system consists of n "modules," then there are n^2 pairwise relationships of the form—

Pij = Probability that a change in module i necessitates a change in module j.

In the following, the letter "P" denotes the $n \times n$ matrix with elements pij. Furthermore, with each module i, there is associated a number Ai of changes that must be made in module i upon integration with the system. (Ai is approximately the number of bugs that show up in module i when it is integrated with the system.) If we let A denote a row vector with elements Ai, then we have the following:

A = total changes, by module, required at integration time, or at *internal release* 0.

AP = total changes required, by module, as a result of changes made in release 0, or total changes for *internal release* 1.

(Internal release $n+1$ is, roughly, a version of the system containing fixes for all first-order problems in internal release n.)

Now we observe that the i,jth element of P^2 is

$$\sum_{k=1}^{n} Pik\ Pkj,$$

which represents the sum of probabilities that a change in module i is propagated to module k and then to module j. Hence, the i,jth element of P^2 is the "two-step" probability that a change in module i propagates to module j. Or, AP^2 is the number of changes required in *internal release* 2.

The generalization is now obvious. The number of changes required in internal release k is given by AP^k and the total number of changes, T, is given by

$$T = A(I+P+P^2+P^3+\cdots).$$

Now we are interested to know whether or not the matrix power series in P converges; clearly, if it does not our system will never stabilize. To establish con-

vergence of the power series, we appeal to matrix algebra (see Reference 5, for example) which tells us that the above series converges whenever the eigenvalues of P are less than 1 in absolute value. If this is the case, then the series converges and

$T = A(I-P)^{-1}$, where I is the $n \times n$ identity matrix.

We now have an extremely simple way to estimate the total number of changes required to stabilize a system as a linear function of a set of initial changes, A. Moreover, the number of changes at each release is given by the elements of AI, AP, AP^2, etc.

ESTIMATING TOTAL DEBUGGING EFFORT FOR A SYSTEM

The above theory suggests a simple procedure for estimating the total number of changes required to stabilize a system. The procedure is as follows:

(1) For each module pair, i, j, estimate the probability that a change in module i will force a change in module j. These estimates constitute the probability matrix P.
(2) From the vector A by estimating for each module i the number of "zero-order" changes, or changes required at integration time.
(3) Compute the total number of changes, by module:

$$T = A(I-P)^{-1}.$$

(4) Sum the elements of the column vector T to obtain the total number of changes, N.
(5) Make a simple extrapolation to "total time" based on past experience and knowledge of the environment. If past experience suggests a "fix" rate of d per week, then the total number of weeks required is N/d.

Hence if we have some estimate for the initial correctness (or "bugginess") of a system and for the inter-module connectivity (the probabilities), then we can easily obtain an estimate for the *total* number of changes that will be required to debug the system. The formula is a simple one in matrix notation, but the fact that we are dealing with matrices probably explains the failure of our intuition in understanding debugging problems.

In the following sections, we will show how the above formula can be used to aid our understanding of other aspects of the debugging process.

STAGING INTERNAL RELEASES

There are various strategies for tracking down bugs in a complex system. The most obvious are: (1) fix all bugs in one selected module and chase down *all* side effects, or, (2) fix all "first-order" bugs in each module, then fix all "second-order" bugs, and so on. The module connection model can aid in predicting release intervals for either approach.

For strategy (1) (one module at a time), the number of changes required to stabilize module i, given Ai initial changes, is given by

$$(p, \ldots, Ai, \ldots, 0)(I-P)^{-1}$$

The product is a row vector with elements corresponding to the number of changes that must be made in each module as a result of the original changes. The total number of changes required to stabilize this one release is given by

$$Ai \sum_{k=1}^{n} Xik,$$

where the Xik are elements of $(I-P)^{-1}$. This strategy, then results in n internal releases where the time for release i is

$$Ai(\max_k Xik) \times (\text{time required per change})$$

and the total debug time after integration is

$$\sum_i (Ai \max_k Xik) \times (\text{time required per change})$$

With the second debugging strategy (make all "first-order" changes, then all "second-order" changes, etc.), the number of changes in the kth release is given by AP^k. That is, AP^k is a row vector with elements corresponding to the number of changes in each module for release k. The time required for release k is approximately

$$\max (AP^k) \times \text{time required per change.}$$

To determine the total number of releases for this strategy, we must examine A, AP, AP^2, until the number of changes AP^s in release s is small enough that the system is releasable. The total time for this strategy, then, is

$$\sum_{k=0}^{s} \max AP^k \times \text{time required per change.}$$

It is worth noting that both of the debug strategies described above evidence a "critical path" effect. The total time in each case is a sum of maximum times for each release. This effect corresponds to the well-known fact that debugging is generally a highly sequential

process with only minor possibilities for making many fixes in parallel. This fact, coupled with the "amplification" of changes caused by rippling effects, certainly accounts for a large portion of many schedule slips.

REFINING THE INITIAL ESTIMATES

Module connection analysis is proposed as a tool for aiding designers and implementors. More than anything else, it is a rationale for making detailed quantitative estimates for what is generally called "contingency." Now, we must ask, "As a project progresses, how can we take advantage of actual experience to refine the initial estimates?" The module connection model is based on two objects: A, the vector of initial changes; and P, the matrix of connection probabilities between the modules. Both A and P can be revised simply as live data become available.

As each module i is integrated into the system, the number Ai of initial changes becomes apparent.

Using updated values for the vector A, we can recompute the expected total number of changes and the revised release strategy.

The elements, Pij, of the matrix P can be revised periodically if sufficient data is kept on changes, their causes, and their after-effects. One simple way to do this is to keep a record for each module as follows:

Module i		
description of change	caused by which module?	other modules affected

After a relatively large sample of data is available, the above forms can be used to revise P as follows:

$$Pij = \frac{\text{number of changes in } j \text{ caused by } i}{\text{total changes made to } i}$$

The revised matrix P can be used to revise earlier estimates for total effort and release strategies.

AN EXAMPLE OF MODULE CONNECTION ANALYSIS

The following example is based on the Xerox Universal Timesharing System. Eighteen actual subsystems are used as "modules." Estimates for connection probabilities and initial changes are made in the same way that they would be made for a new system, except that some experience and "feel" for the system were used to obtain realistic numbers. (Thanks to G. E. Bryan, Xerox Corporation, for helping to construct this example.)

.2	.1	0	0	0	.1	0	.1	0	.1	.1	.1	0	0	0	.1	0	0
0	.2	0	0	.1	.1	.1	0	0	0	0	0	.1	.1	.1	0	.1	0
0	0	.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	.1	0	.2	0	.1	.1	.1	0	0	0	0	0	0	.1	0	.1	0
.1	0	0	0	.4	.1	.1	.1	0	0	0	0	0	0	0	0	.1	0
.1	0	0	0	0	.8	.1	0	0	.1	0	0	0	.1	0	0	.1	0
.1	0	0	.1	.2	.1	.3	.1	0	.1	0	0	0	.1	0	.1	.1	0
.1	.1	0	.1	.2	0	.1	.4	0	.1	0	0	0	.1	0	0	0	.1
0	0	0	0	0	0	0	0	.1	0	0	0	0	0	0	0	0	0
.1	0	0	0	0	.1	.1	.1	0	.4	.2	.1	.2	.1	.1	.1	.1	.1
.1	0	0	.1	0	0	0	0	0	.2	.3	.1	0	0	0	0	0	0
.2	0	0	0	0	.1	0	0	0	0	.2	.3	0	0	.1	.1	0	0
.1	.1	0	0	0	.1	.1	.1	0	.2	.1	0	.3	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	.2	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	.2	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.2	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.2	0
0	0	0	0	.1	0	.1	0	0	.1	0	0	0	0	0	0	0	.3

Figure 1—Probability connection matrix, P

The 18×18 probability connection matrix for this example is given in Figure 1. The matrix is relatively sparse; moreover, most of the nonzero elements have a value of .1. Most the larger elements lie on the diagonal

INITIAL AND FINAL CHANGES

Module	Initial Changes	Total Required Changes
1	2	241.817
2	8	100.716
3	4	4.44444
4	6	98.1284
5	28	248.835
6	12	230.976
7	8	228.951
8	28	257.467
9	4	4.44444
10	8	318.754
11	40	238.609
12	12	131.311
13	16	128.318
14	12	157.108
15	12	96.1138
16	28	150.104
17	28	188.295
18	40	139.460
TOTALS	296	2963.85

Figure 2

corresponding to the fact that the subsystems are relatively large so that the probability of ripple within a subsystem is relatively large.

The total number of changes required in each module are given in Figure 2. It is interesting to note which modules require the most changes and to observe that six modules account for 50 percent of the changes.

Figure 3 illustrates the one-release-per-module debug strategy. That is, we repair one module and all side effects, then another module, and so on.

This strategy is rather erratic since the time between releases, which is determined by the maximum number of fixes in one module, ranges from 4 to 95 indiscriminately. If we adopt this strategy, we may want to select the

ONE RELEASE PER MODULE

Release	Maximum Changes in One Module
1	4.41764
2	11.8619
3	4.44444
4	8.84029
5	67.8994
6	24.7185
7	20.3720
8	85.8099
9	4.44444
10	35.2976
11	95.2147
12	22.5608
13	39.7013
14	15.0000
15	15.0000
16	35.0000
17	35.0000
18	66.5554
"CRITICAL PATH" TOTAL	592.138

Figure 3

worst module first and continue using the worst module at each step. We will see, however, that this strategy is far from optimal because it does not take maximum advantage of opportunities to make fixes in parallel.

A more effective release strategy is illustrated in Figure 4. This strategy assumes all first-order changes in release 1, all second order changes in release 2, etc.

ure 4 shows, for each release, the maximum number changes in one module and the total number of changes. The reader who has worked on a large system will, no doubt, recognize the painfully slow convergence pattern. In this case, the system is assumed to be ready for external release when the "maximum changes per module" becomes less than one.

If we assume the "critical path" changes are made at

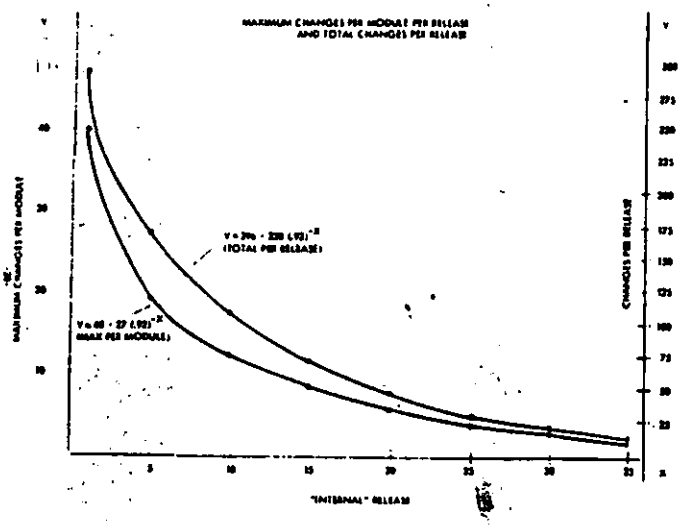


Figure 4—"Internal" release

an average rate of about 1 per day, then Figure 4 is fairly representative of experience with the first release of UTS. The total number of changes on the "critical path" is 338, so that approximately 15 months would

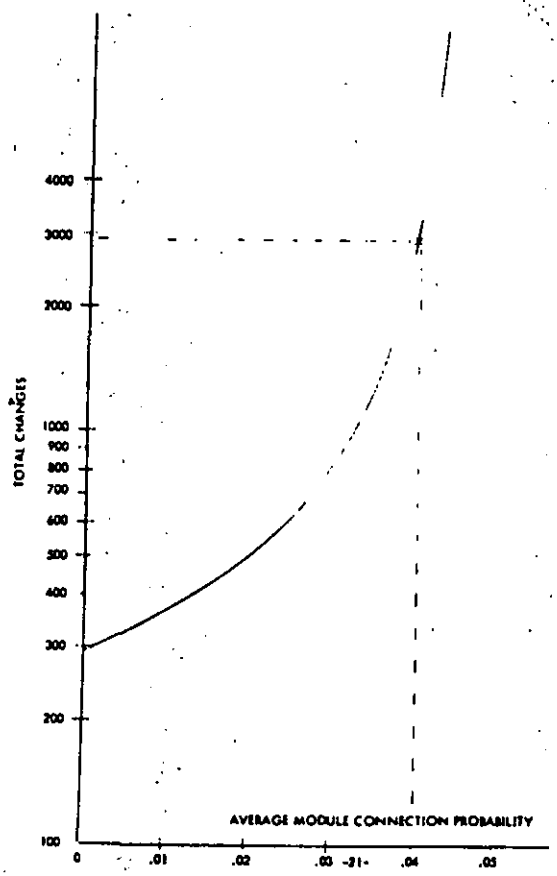


Figure 5—Total changes as a function of "average connection probability"

be required to stabilize the system for the first external release.

To conclude this example, let us take a brief look at the relationship between "total changes" and the probability of intermodule connection. The probabilities in the connection matrix above have an average value of approximately .04. What is the result if we assume the same relative distribution of probabilities in the matrix, but reduce the average by dividing each element by a constant?

Figure 5 shows the total number of changes as a function of "average probability of module connection" under the above assumption. This curve shows that our example is precariously close to "critical mass" and that any small improvement in the connection probabilities results in significant payoff.

OTHER APPLICATIONS OF MODULE CONNECTION ANALYSIS

The value of module connection analysis is its simplicity. The computations can be performed easily by a small (less than 50 lines) program written in APL, BASIC, or whatever language is available. Used on-line, the technique is useful for experimenting with various design approaches, implementation strategies, etc. Three examples of this use of the model are described below:

Estimating new work

If the designers, or managers, of a system have kept detailed records of the module-module changes in the system (as described above), then the matrix P is a reliable estimator of the "ripple factor" for the system. It can be used to predict, and stage, the effort to stabilize the system after any set of changes. If we postulate a major improvement release of the system, then we can assume, for example, that the new program code falls into two categories: (1) independent code particular to a new function and, (2) code that necessitates changes in an existing module. By estimating the number of changes, bi, to each module i, we can estimate the total number of changes to restabilize the system:

$$\text{Total changes} = (b_1, b_2, \ldots, b_n)(I-P)^{-1}.$$

The previously described computations can be used to estimate release intervals and total time for the improvement release.

To be more realistic, it may be useful in the above computation to use $bi+ei$ as the estimated changes in the module, where ei represents the number of changes required in module i by previous activity.

Evaluating design approaches

The best time to guarantee success of a system development effort is in the early design stages when architecture of the system is still variable. There is much to be gained by selecting an appropriate "decomposition" (see Reference 4). of the system into subsystems, components, etc. During this stage of a project, module connection analysis is a useful tool for evaluating various decompositions, interfacing techniques, etc. It is a simple, *quantitative* way of estimating the *modularity* of a system, the ever-present objective that no one knows exactly how to achieve. By fixing some of his assumptions about intermodule connections, a designer can experiment with various system organizations to determine which are the least likely to achieve "critical mass."

Evaluating implementation approaches

The reader who performs some simple experiments with the formulas described here is likely to be very surprised at the results. Even an extremely sparse connection matrix with very low probabilities can result [examine $(I-P)^{-1}$] in very large "ripple factors." It is also interesting to experiment with small perturbations in the connection matrix and observe the profound effect they can have on the "ripple factor." One becomes convinced more than ever before that it is necessary to minimize connections between modules, localize changes, and simplify the process of making changes.

The most impressive gains come from minimizing the probabilities of intermodule propagation of changes. A reduction of the average probability by as little as 5 or 10 percent can cause a significant reduction in the "ripple factor." Additional improvement can result from improvements in techniques for making changes. The total debug time is essentially linear with respect to the time required to make a change, but the multiplier (total number of changes) can be so large that any reduction in the time-per-change results in enormous savings.

Module connection techniques are extremely useful in estimating the value of various implementation techniques and strategies. How are the module connection probabilities changed if we use a high-level implementation language? How much easier will it be to

make changes? How much will we save, if any, by doing elaborate environment simulation and testing of each module before it is integrated with the system? Module connection analysis is a valuable augmentation of intuition in these areas and can be useful for generating cost justifications for approaches that result in significant savings.

CONCLUSION

The objective of this paper has been to describe a simple model for the effect of "rippling changes" in a large system. The model can be used to estimate the number of changes and a release strategy for stabilizing a system given any set of initial changes. The model can be criticized for being simplistic, yet it seems to describe the *essence* of the problem of stabilizing a system. It is clear, to the author at least, that experimentation with the module connection model could have prevented a significant portion of the schedule delay that occurred for many large systems.

REFERENCES

1 H R J GROSCH
Why MAC, MIS, and ABM won't fly
Datamation 17 Nov 1 1971 pp 71-72

2 J D ARON
Estimating resources for large programming systems
Software Engineering Techniques J N Buxton and B Randell (eds) April 1970

3 L A BELADY M M LEHMAN
Programming system dynamics or the meta-dynamics of systems in maintenance and growth
Research Report IBM Thomas J Watson Research Center Yorktown Heights New York July 1971

4 C ALEXANDER
Notes on the synthesis of form
Cambridge Mass Harvard University Press 1964

5 M MARCUS
Basic theorems in matrix theory
National Bureau of Standards Applied Mathematics Series #57 U S Government Printing Office January 1960

Structured Systems Analysis: A Technique to Define Business Requirements

Kathleen S. Mendes **Exxon Corporation**

Structured Systems Analysis (SSA) is one of an integrated set of techniques that has resulted from Exxon's research in software methods. The author discusses SSA's role in system development, describes its graphical language, and explains how users and computer analysts use it to model business operations and determine requirements for computer applications. She then describes diverse business and technical projects that illustrate SSA's efficacy. The author suggests that SSA — originally developed for use in the initial stages of system development — has become a general-purpose modeling technique, which has helped to further establish formal methods for system development. *Ed.*

Growth in the development and use of computer systems has been rapid over the past twenty years. Hardware costs, for a given level of performance, have decreased. Communications technology, an important factor in the advance of computers and information processing, has expanded the capabilities for sharing and coordinating information from many sources. However, software technology — the methods and languages used to develop computer systems — has not shown comparable improvement. As a result, the hardware revolution has driven the demand for computer systems upward by providing increasingly attractive opportunities, but software capability has constrained the ability of system developers to meet this demand.

These hardware and software trends motivated Exxon's computer scientists, in the mid-1970s, to try to improve software methods. Their objective was to devise systems analysis and design techniques that would enhance both the quality of computer applications and the ability of development staffs to deliver systems.

Structured Systems Analysis (SSA) is a member of an integrated set of techniques that resulted from Exxon's research and development in software methods. SSA is a business modeling and communication technique used jointly by users and computer systems analysts to describe business environments and to formulate requirements for computer applications. It gives analysts, for the first time, both a definition of the products to be generated at each stage of a project, and a basis for project planning. Moreover, its application extends beyond the system development environment. It has been used effectively in business improvement studies that do not involve computerized solutions.

This article describes Exxon's SSA, illustrates its usage, and discusses its benefits. It also suggests that SSA, originally developed as an analysis technique for use during the first stages of system building, has evolved into a general purpose modeling technique for describing a business.

Background and Development

In the past, due to the absence of formal analysis and design techniques, analysts learned through apprenticeship. System developers worked with more experienced people until they developed their own approaches. The first formalization of the system building process introduced the project management disciplines of other engineering and technical areas into computer application projects. These disciplines regarded system development as taking place in four general stages: *Analysis*, in which a business is described and the requirements for a computer system are formulated; *Design*, in which the hardware and software specified in Analysis are configured; *Construction*, in which the software modules are implemented and the hardware is installed; and *Maintenance*, which encompasses a broad range of activities related to keeping the system operational. However, they failed to provide a methodology that could accomplish the objectives of each of the stages of system development.

During the late 1960s, software engineering pioneers, such as Edsgar Dijkstra and Michael Jackson, began fundamental research on formal approaches for system development. Much of the research focused on program design and implementation, key ac-

The utility and applicability of structured techniques must be recognized before they can be successfully implemented.

ORDER AND DISCIPLINE: BENEFITS OF STRUCTURED TECHNIQUES

by David S. Iwahashi

In the last few years, many articles have been published expounding the virtues of structured programming concepts and techniques. These techniques reportedly have improved software manageability, productivity, quality, and maintainability. However, in spite of these claims, large software development efforts are still plagued with the classic programming problems:

- Inaccurate, overly optimistic percentage-complete estimates. Programmers have been characterized as notoriously unreliable in providing accurate completion estimates, as in the "95% complete" syndrome.
- Uneven productivity throughout the development effort. Disarray, overtime, and waived standards increase as the development approaches the due date.
- Excessively complex systems design. The result is fragile software that is difficult to maintain and modify.
- Inadequate documentation and a lack of appropriate standards and insight into the needs of the user.
- Inaccurate instruction count estimates. Deriving cost estimates entirely from software instruction count estimates is not a reliable technique; cost overruns may frequently result.

Theories and textbook methodology are often inadequate or unrealistic. Software systems are developed in unique environments with diverse characteristics. Furthermore, the human element is a major variable, with a complexity that is accentuated in large-scale software efforts. The art of effective large-scale software development requires insight, imagination, and creativity.

As the chief programmer on a large-scale effort, I attempted to maintain a perspective on problems and appropriate solutions. A technically sound system was paramount. Fundamental software and management control techniques were considered simultaneously in order to facilitate comprehension between programmers and nonprogrammers. The basic techniques and objectives were structured design to simplify system complexity and understanding, status and control techniques to provide progress visibility and realism, checkout milestones to impose steady productivity, and standards and guidelines to enhance code and documentation quality.

Here, we will discuss how several structured programming concepts were incorporated into a large-scale scientific software development effort (15 programmers, 100K lines of code, 200 man-months). This scientific software development effort consisted of two separate but dependent programs—the display program and the batch program.

The display program consists of approximately 19 separate displays that provide the user with the capability to generate and update controls for a real-time system; to specify data sets, schedule and optimize data; and to generate and update control files and list files. This program can read 12 different types of control and reference files and write nine. The approximate program size is 77 overlays and suboverlays, 790 subroutines, 60K total FORTRAN source instructions, 11K total machine language source instructions, and 117K total cards.

The batch program has the capability to generate, update, and list nine reference files in a batch mode; these reference files are used by the display program. The approximate size of the program is 42 overlays and suboverlays, 366 subroutines, 20K total FORTRAN source instructions, 4K total machine language source instructions, and 42K total cards. (Core restriction requirement for both programs produced numerous overlays and suboverlays.)

The programs were developed on Control Data 6000 series equipment. Initial requirements had been established during a nine-month study effort. Program development required approximately seven months for requirements generation, preliminary design, and detail design; about 11 months were needed for code, checkout, and testing.

STAFF DIVIDED IN THREE PARTS

The software development staff was organized into programming, software integration, and system engineering groups. This organizational structure had been in use for approximately nine years, producing similar special-purpose scientific software. These software products and development efforts met requirements, were delivered relatively on schedule (a few months late at worst), and required extensive programmer overtime (averaging 11% at worst), especially during the final phase of checkout. The efforts also occasionally overran cost estimates slightly (+10% at worst), and were relatively error free.

Several structured programming techniques were implemented on these previous efforts, including top-down code development, utilization of intermediate programming language (IPL), code indentation, GOTO-less programs, and code reviews.

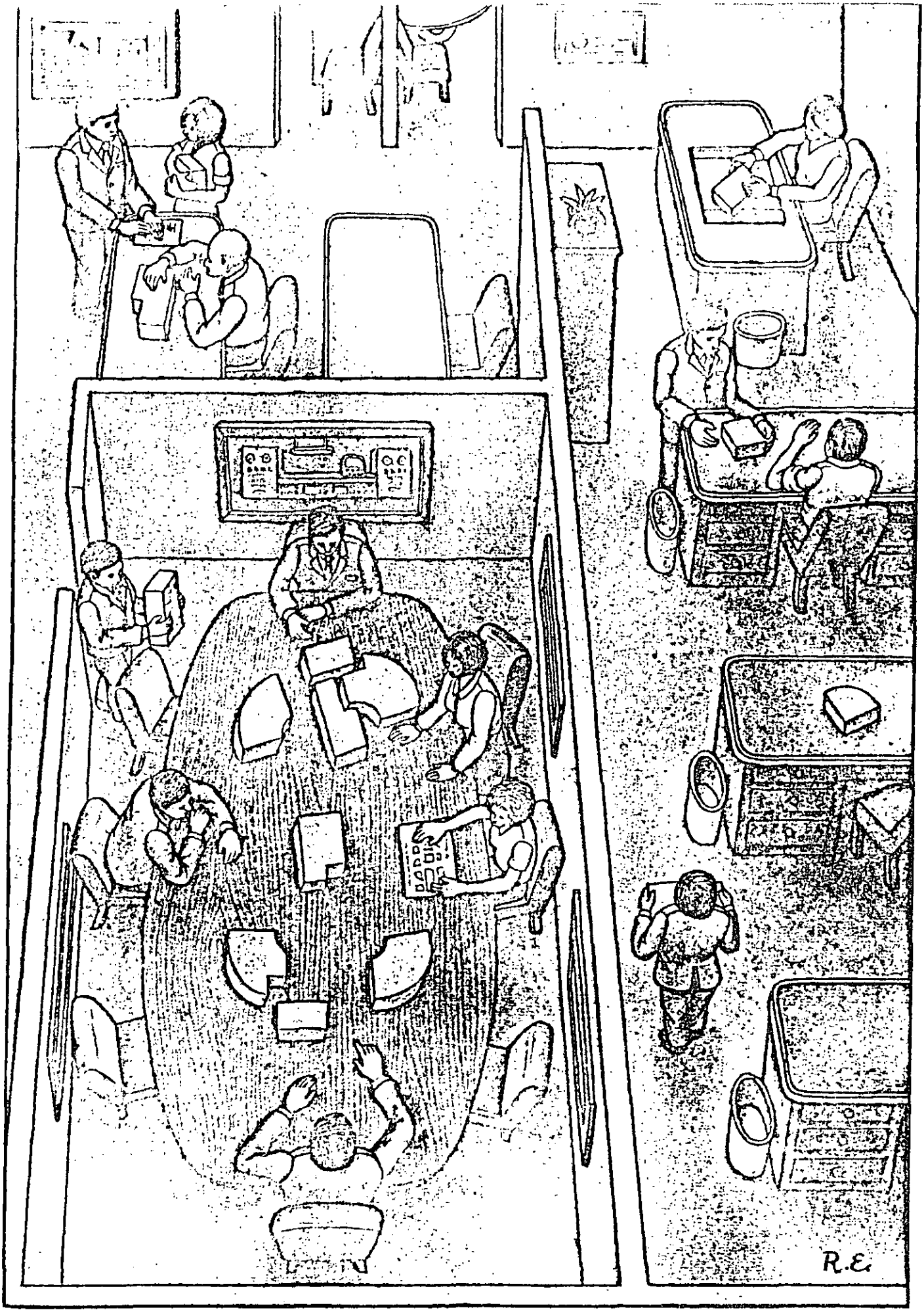
These earlier structured efforts experienced some software development problems. The indented code was difficult to update and maintain, reviews tended to be cursory, and extensive use of GOTO-less code took longer to execute and was difficult to follow. As a result, the maintenance programmers complained loudly about structured programming.

The organization, staffing, and responsibility on this software development effort were as follows:

Chief Programmer. The chief programmer performs a specific set of functions, and is not a position in name only. The chief programmer need not be a superprogrammer. On this effort, it was more important that the chief programmer be a software system designer, understand and appreciate software development problems, and have some imagination and insight for effective task assignments and control. The day-to-day programmer interface and guidance required for software design reviews and technical decisions indicate that the chief programmer should come from the software pits.

The chief programmer had overall responsibility for the entire development. A thorough and comprehensive understanding of all technical requirements was mandatory. The chief programmer had extensive, detailed experience (10 years) with the computer system, customer, and development procedures.

It is my opinion that the chief programmer can effectively direct, control, and offer technical guidance to only six to eight team members performing unique


R.E.

tasks. If there are more subordinates, some capable assistants are required. On this effort, software integration personnel provided the necessary assistance through technical guidance on the complex algorithms, intermediate program checkout and verification, and documentation reviews.

Assistant Chief Programmer. The assistant aids the chief programmer and handles lower-level design decisions. An assistant is necessary on large software development efforts to review and critique technical decisions, follow up in problem areas, assist with external communication and interfaces, train and brief new personnel, and minimize stopgap programming. The assistant also gets valuable training and experience, which insures the project against the loss of the chief programmer.

Unfortunately, this development effort did not have an assistant because programming personnel were unavailable. As a result, the programmers and software integration personnel were required to work harder.

Programmers. The programmers had responsibility for detailed software design, design documentation, development of code, and program checkout. Up to 15 programmers were required to complete this development effort. Programmer experience ranged from one to 19 years and every programmer had some previous experience on the applicable computer equipment. Individual programmers differed greatly in ability and experience. The number of years of software experience does not measure a programmer's applicable experience, true ability, or overall professionalism. It is necessary to correlate an individual programmer's experience, capability, and track record with the complexity of the specific task.

Task complexity ranged from straightforward data manipulation (75% of resultant code) to complex mathematical scheduling and optimization algorithms. The hierarchical diagram and structured design provide a means of better understanding complexity and help provide a basis for appropriate personnel assignments. Attempts were made to assign tasks according to programmer experience, expertise, track record, and individual preference. Although no generalization should be derived, the following are interesting observations:

- The less experienced programmers tended to do checkout superficially and have more discrepancies.
- Competent programmers successfully completed tasks regardless of complexity.
- The less productive programmers tended to govern on-time product delivery.

All major tasks were sized into manageable subtasks; major subtasks were further subdivided so that each component required approximately three to six months' effort. Manageable subtasks made it easier to accommodate schedule slippages and personnel attrition with reassignments to supplement development progress.

Librarian. The librarian's function was to keep track of program versions and files, and verify program updates throughout the development effort. This was an important responsibility and a key reason the development made steady progress. The most recent program version and listing were always available for checkout; no time was lost due to improper updates being used.

System Programmer. This programmer was responsible for computer system problems and anomalies. Since this effort was developed on an established computer and operating system, the support was on a problem-solving basis only. On new large-scale developmental computer systems, it would be advisable to have a resident system programmer available to handle the day-to-day problems encountered during software development.

Software Integration Group. This group was responsible for the coordination of the development effort, algorithm support, intermediate program checkout and user's manuals. The integration group provided a useful interface buffer between the programming group and the system engineering group. The complexity of the mathematical scheduling and optimization algorithms necessitated one member of the integration group be assigned to follow, monitor, checkout, and verify each algorithm.

System Engineering Group. This group was responsible for the software requirements, formal software testing, and customer interface.

SOFTWARE REQUIREMENTS DOCUMENT

Nine months of earlier engineering effort culminated in a study report that identified existing system limitations and proposed improvements, a design approach, and design requirements. This report contained numerous separate engineering desires and concepts, as well as ambiguities and uncertainties.

The software requirements document was written by the system engineering group. Even though the study report provided initial requirements, additional effort had to be expended to rewrite the requirements to provide logical and functional specifications rather than detailed algorithm and implementation design specifics. Software development efforts frequently begin with a feasibility analysis. A structured top-down approach to these studies and reports could significantly enhance a structured implementation.

Preliminary Design. The overall problem and the computer system and resources must be totally understood before a feasible software solution can be designed. The initial design was in hierarchical form and indicated how the problem could be solved in an orderly, well-structured manner. The initial diagrams were completed by the chief programmer in a one-month period, prior to establishing the programming team. These diagrams provided program organization in a top-down, stepwise refinement manner, as well as verification that all requirements were included (diagram contained study report and requirement document paragraph numbers for cross-reference); an outline for preliminary and detail design documentation; identification of subprogram size and efforts for personnel assignments; a clear delineation of responsibility; new personnel indoctrination/briefing information; identification of areas of concentration; identification of common utility and library routines; minimization of intraprogram communication (key factor in simplifying programming tasks by reducing interfaces).

The initial hierarchical diagrams were of sufficient detail (10 to 12 levels for each function) to show that the problem could be solved, and that all requirements were incorporated. These diagrams were not superficial; rather, they had sufficient depth to identify the complexity and magnitude of the subprograms. The key design features were to establish communication and control high within the hierarchy, standardize and minimize communication between hierarchical components, and design manageable components which could be independently verified.

Good, detailed hierarchical diagrams simplify subsequent assignments of tasks and responsibilities, as well as simplify control. Hierarchical diagrams cannot replace flow diagrams. Flow diagrams identify sequences, controls, and data flow, and are useful in describing the details of complex algorithms.

The hierarchical diagrams were the entire basis of the 450-page preliminary design documentation, and took about three months to develop. The chief programmer developed the overall docu-

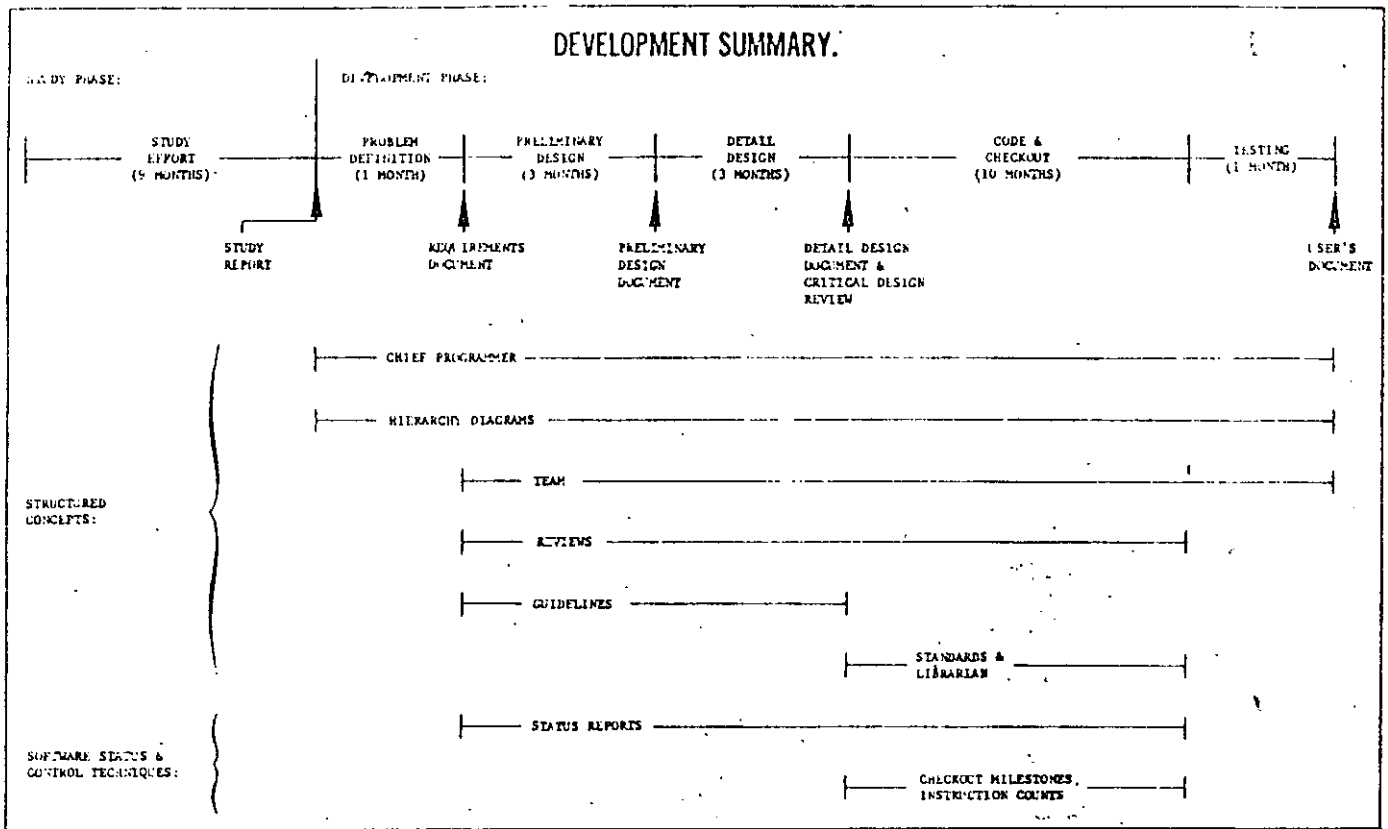


Fig. 1.

mentation outline and preliminary design guidelines. The basic documentation guideline was that the preliminary design should be of sufficient depth to verify a technical understanding of the problem and to indicate a viable solution. An estimate of the number of pages in each subsection was provided to scope the effort expected of each programmer.

Further, each programmer was required to submit an outline of his subsection. These initial outlines tended to lack organizational structure and consistency, and had to be modified. The chief programmer and integration group reviewed and proofread each programmer's initial documentation. In most cases, the initial documentation required extensive reworking to delete unnecessary detailed processing flow and to provide adequate depth for complex functions.

Previous experience with software design documentation indicated written material tended to lack organization, evaded complex problems, or provided repetitive detail about trivial matters. The documentation outlines and guidelines were an attempt to solve these shortcomings. For future efforts, I would also consider having each major section introduction, summary, and/or conclusion written and reviewed prior to supportive text. This could further scope documentation effort and minimize extensive reworking.

DETAIL DESIGN DOCUMENT

The detail design document of about 1,125 pages took about three months to develop. The preliminary design provided the basis for the detail design which described how the delivered software would be built. The chief programmer developed the overall documentation outline and detail design guidelines. The basic guideline was that the detail design should be a programmer-oriented document that would provide detailed information on how functions were performed. It was to be of sufficient detail to code from, yet general enough to reflect resultant code with a minimum documentation impact. Specific guidelines were identification of all inter-subprogram communication buffers and storage, subprogram overlays, suboverlays, and major routines, subprogram buffers and arrays whose size or structure is dependent on data external to the subprogram, subprogram defaults, and subprogram error conditions and corrective action.

As with the preliminary design, each programmer was required to submit an outline of his subsection. Again, these outlines proved informative but often had to be redone to reflect a hierarchical structure. Each programmer's initial documentation was proofread and returned for reworking. As with the preliminary design effort, I recommend each major section introduction, summary and/or conclusion be written and reviewed prior to the supportive text.

Code and Checkout. Software is frequently evaluated and remembered by its worst attribute (e.g., slow execution, difficult to modify, or awkward to use). Attempts were made to avoid these attributes during the code and checkout phase. Standards, reviews, checkout procedures, and software aids were utilized to enhance development and to produce quality software. The code and checkout phase took about 10 months.

Coding Standards. The basic coding standard was to develop software that was readable, sequential, and maintainable. It is difficult to derive all-encompassing standards due to the numerous peculiar situations and circumstances for which software is developed. Personal judgment is ultimately required to evaluate the detailed software development standards; there is no substitute for good judgment. Mandatory standards were provided regarding routine description, input/output identification, error condition descriptions, and comments. Programming conventions and guidelines were also provided regarding overlay

Don't expect immediate success and miracles because structured techniques are used. 13

names, file utilization, variable utilization, variable names, variable assignments, and routine length. The standards, conventions, and guidelines attempted to achieve maintainable code without restricting the experienced programmer's judgment or habits.

Basically, these standards were adhered to; total adherence would require several people full time to monitor each line of code. The effectiveness of these standards will be realized in the future, when program maintenance and modifications are required.

Implementation Reviews. Following successful completion of the customer critical design review, implementation (code and checkout) began. Each programmer's implementation approach and initial coding efforts were reviewed by the chief programmer. Reviews were performed on detailed hierarchy and flow diagrams, and then on listings. These reviews were beneficial, necessary, and a key factor to the success of this software development effort.

It was particularly interesting (and amazing) to review and critique the initial implementation phase. Reviews indicated difficulty in implementing the details of the software design in a manner which was well structured, independent, and easy to check out. That is, the initial software implementation tended to impose additional complexities which could be avoided with a more straightforward approach. Some of the more common faults were:

- Lack of program structure and organization—implementation did not have clear, definable subsets which could be totally verified at intermediate points. Implementation increased program complexity and required extra checkout.
- Misuse of computer resources—one-word disk reads and writes.
- Lack of implementation flexibility—fragile software which was extremely data-dependent.
- Use and misuse of data statements—same data statement used in numerous routines, or buffer sizes not specified in data statements for easy identification and update.
- Redundant code—required extra checkout (vs one subroutine).
- Limited understanding and appreciation for the amount of time it takes to process large volumes of data—data was processed one item at a time.

These are merely personal observations made during the implementation reviews; it is not obvious whether these faults arise from a lack of understanding of the computer and its resources, beginning too close to the task, or not understanding the entire task. Team reviews were not held throughout the development effort; the size and complexity of each programmer's task was such that it would be difficult for other team members to understand and offer constructive detailed criticisms.

Team reviews during the design and especially during the initial implementation phase could alleviate obvious shortcomings and common faults, and could be educational for subsequent development efforts. I'm not convinced team reviews during the entire coding phase are beneficial and economical on all large-scale software development efforts. If code reviews are held, they should be performed by a code review board (vs entire team); the board approach appears to be necessary if programming requirements impose mandatory coding standards.

TEAM CODE CRITIQUE

Upon completion of the software development effort, each programmer was given representative listings of code generated by all other programmers on the team. These listings were reviewed and critiqued in a team review meeting. The intent of this review was to expose the development programmers to software maintenance and enhance their judgment of coding techniques.

The codes received a variety of comments. Some were described as well structured, maintainable, and easy to follow, with a few singled out as easier to read and follow due to the linearity of the task. A number received descriptive processing flow comments. Programmers pointed out restrictions or error conditions, occasional lack or misuse of data statements and external storage resources, and indicated where programming notes should have been more frequent and descriptive.

The effectiveness of these reviews will be realized in the future, when team members develop new software.

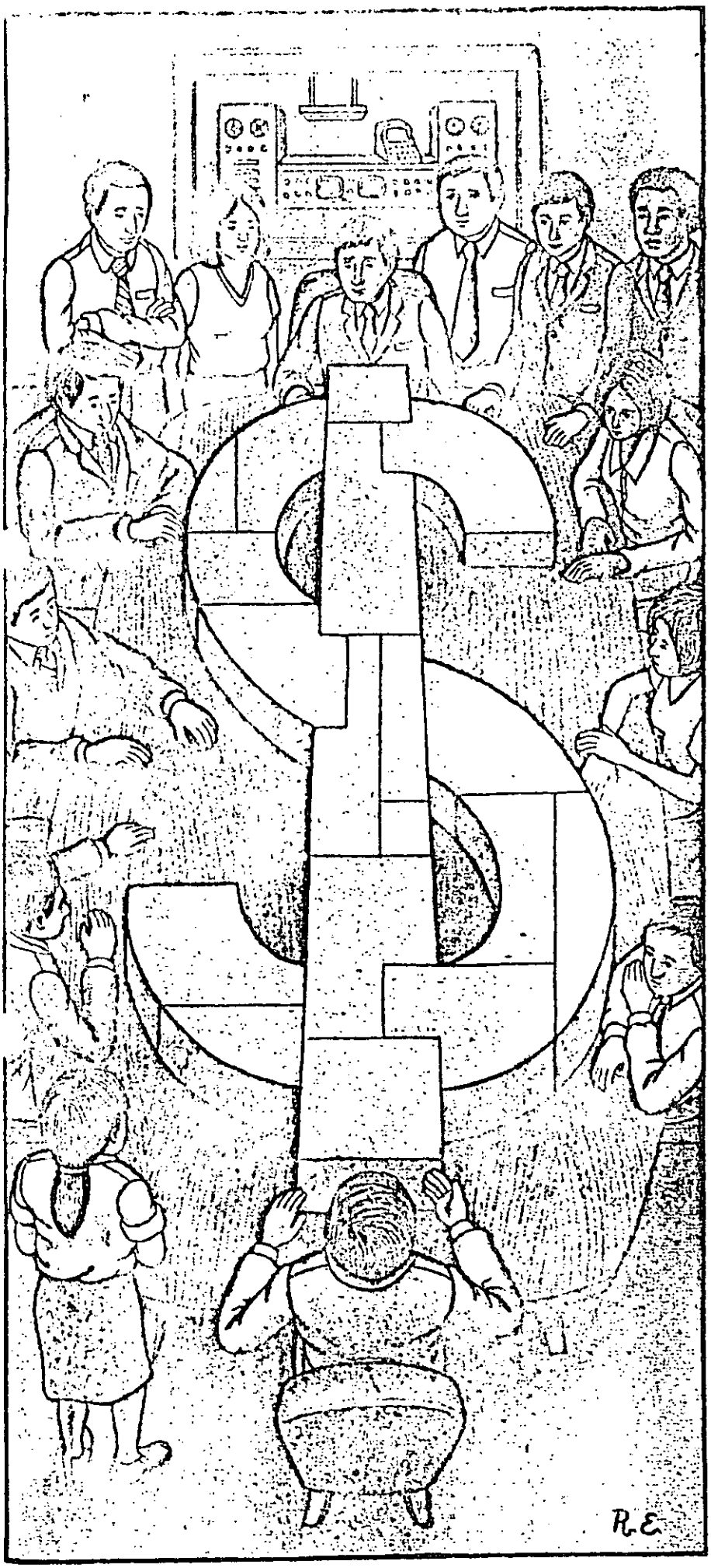
Top-Down Coding/Checkout. Code and checkout progressed in a top-down manner. Initially, the program executive was coded and totally checked out; "stubs" for all overlays were incorporated, and utility/library routines completed. Main overlays and suboverlay "stubs" were coded and checked out by the responsible programmer. Subsequent code and checkout sequences were left to the discretion of the individual programmer; emphasis was placed on completing the more complex and interoverlay/suboverlay functions first.

Software Aids. A useful set of software development aids and utility routines were factors in the overall success of the programs. The following are noteworthy:

- Display processing simulation program. This batch program saved significant checkout time by enabling the programmer to simulate execution of display subprograms without actually operating the display.
- Display program central memory, external storage, and internal file dump features. These online features provided the capability to dump pertinent data if errors were encountered during display checkout without subsequent diagnostic trace updates.
- Library containing frequently utilized functions such as data bit manipulations and disk access routines.
- General purpose file program used to create checkout files.
- Central file book. This book contained all pertinent file formats and specifications; modification to files were coordinated through one individual.
- Interactive computer terminals. Terminals significantly improved daily computer turnaround.

Testing. The system engineering group was responsible for the final formal program testing and demonstration. This test was deficient in that it did not demonstrate all the program's capabilities, or a realistic operational scenario; however, the test did identify several program discrepancies. The number of program discrepancies did not appear to be significantly less than on previous development efforts. Strict quantitative comparison as to the number of discrepancies is not the single most important software development evaluation criterion. Evaluation criteria should realistically include such things as requirement satisfaction, design flexibility, code maintainability, and fault isolation simplicity.

Software development visibility and realistic operational applications could be better demonstrated by distributing formal testing throughout the entire development phase. These incremental tests could include combinations of top-down, diagnostic, and subprogram component tests. (Diagnostic tests exercise program limits, error messages, conflicts, input anomalies, and abnormal operational sequences.) The formality of these incremental tests must be traded off with development costs and schedules. Effective software testing requires an understanding of the requirements, operations, and software design.



14

SOFTWARE CONTROL TECHNIQUES

Software control is a topic of much concern, especially on the development of large-scale computer programs. Effective software control requires a very detailed understanding of the effort, a design which may be partitioned into separate independent subprograms, and frequent periodic monitoring. Theoretical control techniques provide basic software management guidelines; the identification of applicable control techniques requires a range of programming and implementation experience.

This experience provides the insight required to develop effective software control techniques which fit the needs of the development effort. A single control technique is not adequate to effectively monitor and control a large-scale software development effort. Effective and realistic software development statusing and control require a combination of several independent techniques and objective evaluations. The following techniques were used during the code and checkout phase of this effort.

Completion Date Estimates. The hierarchy diagrams were used to identify tasks to be estimated. The responsible programmer for each task was requested to make an estimate as to the number of instructions and how long it would take to code and check out the task. Estimates were required for each task, subtask, and subroutine. Weight factors were also estimated to provide a subfunction's relative magnitude and complexity. The intent of having the responsible programmer estimate the effort was to force a more detailed analysis of the task and, hopefully, to instill a personal commitment to the estimate.

With allowance for the fact that programmers tend to be optimistic, the estimates were reviewed and a work schedule generated (see Fig. 2). Each week, the responsible programmer estimated the percent complete on each subtask. The chief programmer evaluated the reported percentages and estimated overall status of the task; the number of weeks ahead or behind schedule was recorded.

Checkout Milestones. Intermediate checkout milestones were established for each task. These consisted of 10 to 15 specific items or functions which could be verified with printer outputs, dumps, or on the displays (e.g., subroutine "A" complete, subtask "1" file generation complete, task "A" vector generation subfunction complete). These milestone dates were relatively evenly spaced throughout the code/checkout phase, and were consistent with the completion date

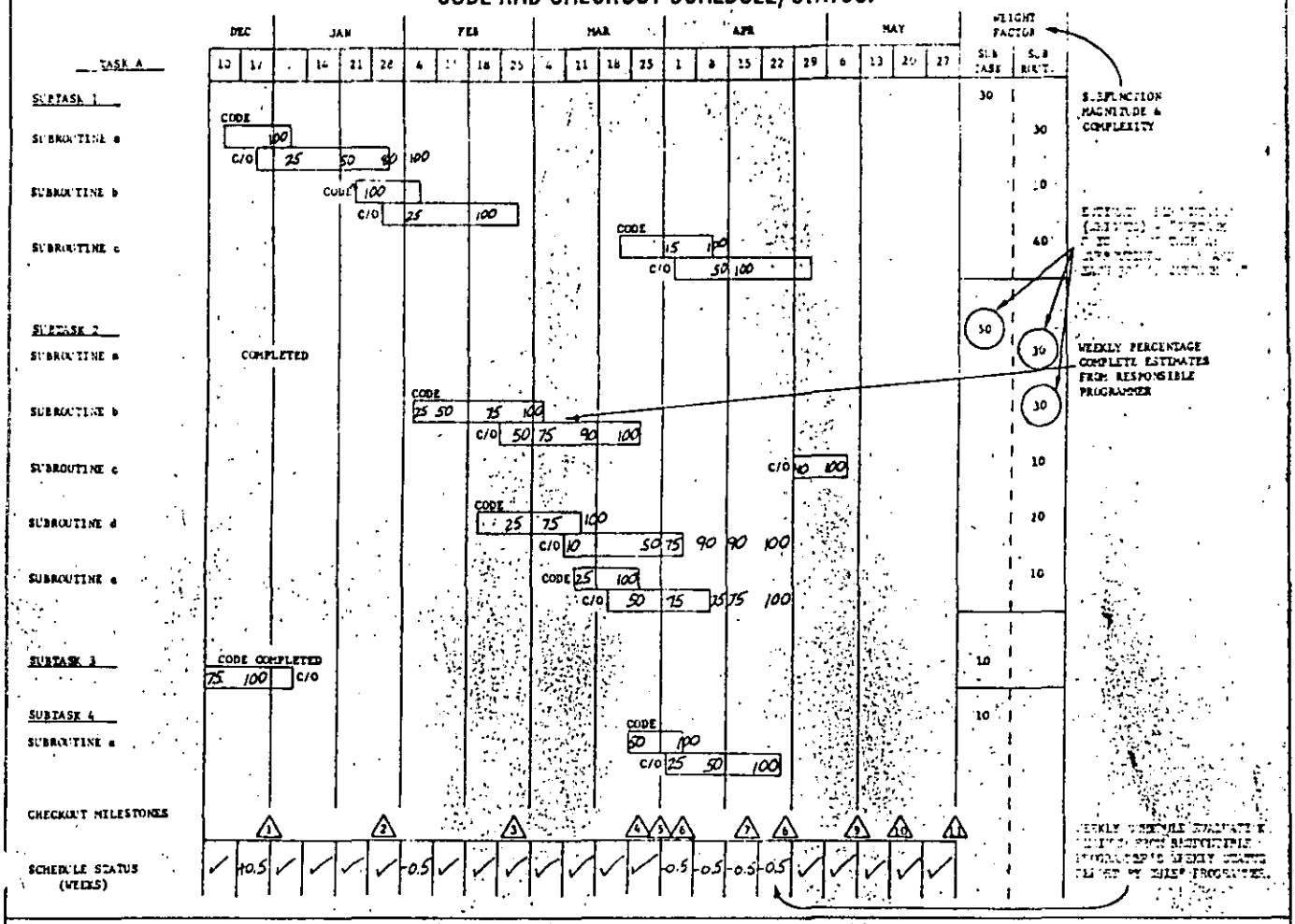


Fig. 2.

estimates. The objective of these milestones was to impose smooth productivity throughout the development phase, and to more clearly indicate schedule problem areas. These milestones were defined such that the item was a GO or NO GO situation; this alleviated subjective evaluation of the task as percentages complete.

Time-Line Charts. Time-line charts were used to indicate dates when "stubs," functions, and subfunctions were to be incorporated. These charts were simply bubble charts with associated dates and estimated critical paths. The dates on the time-line charts were consistent with the completion date estimates and checkout milestones.

Weekly Status Reports. Weekly status reports were required from each programmer; these contained an estimate of the code and checkout completed, status of checkout milestones, and written text regarding progress and problems encountered during the week.

These weekly reports helped to isolate problems and slippages so corrective action could be taken. Theoretically, the programmer updates the "estimated completion dates" as the programming effort progresses; this doesn't work too well with an overall fixed completion date. All schedules were developed with about a one-month margin; tasks which began to slip were monitored more closely. Minor slips (one to two weeks) were tolerable, but positive attempts were made to assist the programmer through job prioritization (keypunch, and computer turnaround), data analysis assistance (software integration personnel), and in a very few cases, overtime was allowed (less than 0.01%).

The programmers seemed to respond to this active concern for their tasks. Major slips required task partitioning and assignment of subtasks to other programmers; the design allowed this partitioning with minimum overall impact. Two major slippages were noted early in the development schedule, when task reassignments could easily be implemented.

A discrepancy log identified problems encountered during checkout by the engineering and integration groups. The intent of this log was to keep track of discrepancies and assure that they were being solved and not "forgotten." This log contained the date the discrepancies were encountered, description, and responsible programmer(s). The weekly number of open discrepancies per programmer seemed to vary significantly (from two to 25); the number appeared to be inversely related to the individual programmer's expertise.

MAKING COUNT ESTIMATES

Historically, accurate instruction count estimation has been a difficult task on large-scale software development efforts. Estimates and actual final instruction counts tend to differ by orders of magnitude. This instruction count difference is probably the area of greatest variance in large-scale software development efforts, that is, the area most likely to be wrong by the largest magnitude. The procedure for estimating instruction counts must be more accurate *if future software development efforts are to be realistically costed/priced according to these estimates.*

The estimation procedure is further complicated since a given task can be coded by different programmers and may result in an order-of-magnitude difference in the final instruction count. For this software effort, an "instruction" is defined as an executable or declarative source statement (comment or continuation cards are not "instructions").

Monthly Instruction Productivity. Source instruction count totals were obtained for each programmer and task on a monthly basis. These counts gave some indication as to how the overall effort was progressing and the productivity of the individual programmer.

During the peak coding months,

individual programmer productivity ranged from 150 to 2,000 instructions a month, with a nominal range of 400 to 1,000. The average productivity for the code and checkout phase (10 months) was approximately 490 instructions a month.

"Total cards" (instructions and comments) is occasionally a useful productivity figure. The "total card" average productivity for the code and checkout phase was approximately 890 cards a month. Programmer productivity varied from 680 to 1,360 cards a month.

Instruction Estimation Technique. Originally, each task was separated into subtasks and functions (as derived from the hierarchical diagrams). These functions were then evaluated against comparable functions in existing programs for which actual instruction counts were available. Estimates were made for program overhead and for functions which had no comparisons. These counts were summed to determine original program instruction estimates, but the estimates were as much as 100% off. Throughout the coding phase, the instruction count estimate was updated, based on actual monthly instruction counts and programmer estimates of percent code completed (see Figs. 3 and 4).

The display program code was reported 100% complete during the eighth month, yet the actual instruction count increased about 10% over the next two months.

The structured concepts, control techniques, and personnel led directly to the success of this software development effort. The top-down design and hierarchy diagrams provided the foundation from which the software was developed, managed, and controlled. A more structured top-down approach to feasibility-study report documentation and testing could enhance all aspects of a software development effort. The chief programmer and team concept was successful because of each individual's ability and willingness to meet the demands of the task. Documentation guidelines and reviews enhanced the quality and usefulness of the software documents. Additional guideline refinements and intermediate reviews appear necessary to minimize extensive documentation rework.

In-depth team reviews on large systems do not appear to be totally effective due to size-comprehension limitation; constructive detailed criticism appear inversely related to program size. The status and control techniques offered credence to the development progress and imposed steady productivity. These techniques provided the visibility and confidence required to manage the development effort and to achieve an on-time product delivery.

Don't expect immediate success and miracles because structured techniques and concepts are used on a software development effort. The value of structured techniques is to impose some order and discipline; ultimately, it is the individuals who govern the final outcome. Don't implement the concepts and techniques for the sake of being "structured"; one must recognize their utility and applicability before they can be successfully implemented. Software development and control techniques are evolving. The concepts and techniques incorporated in this effort fit its specific personalities and needs. ✪

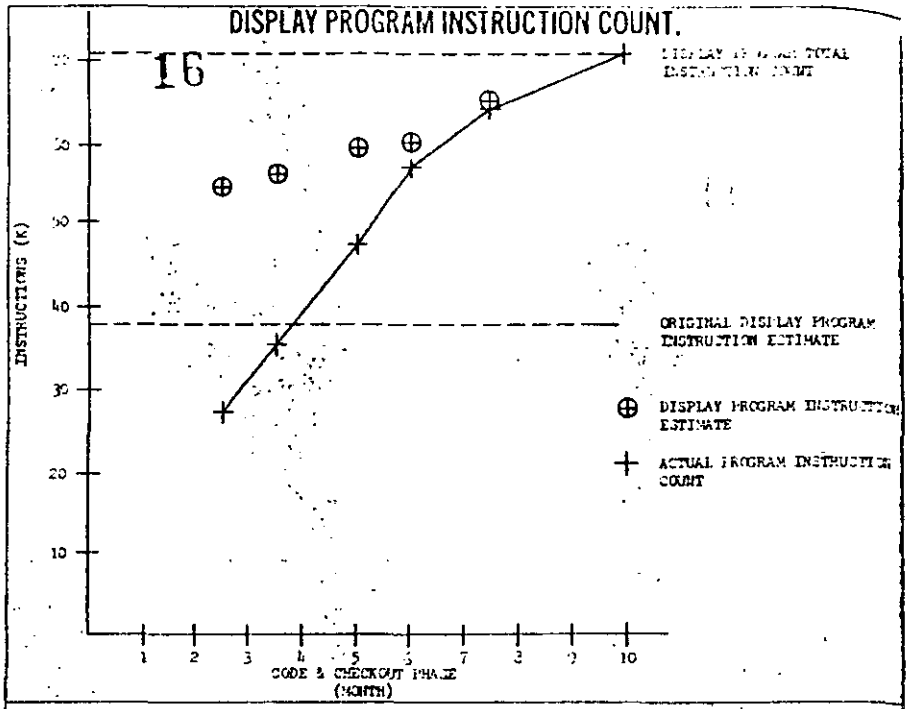


Fig. 3

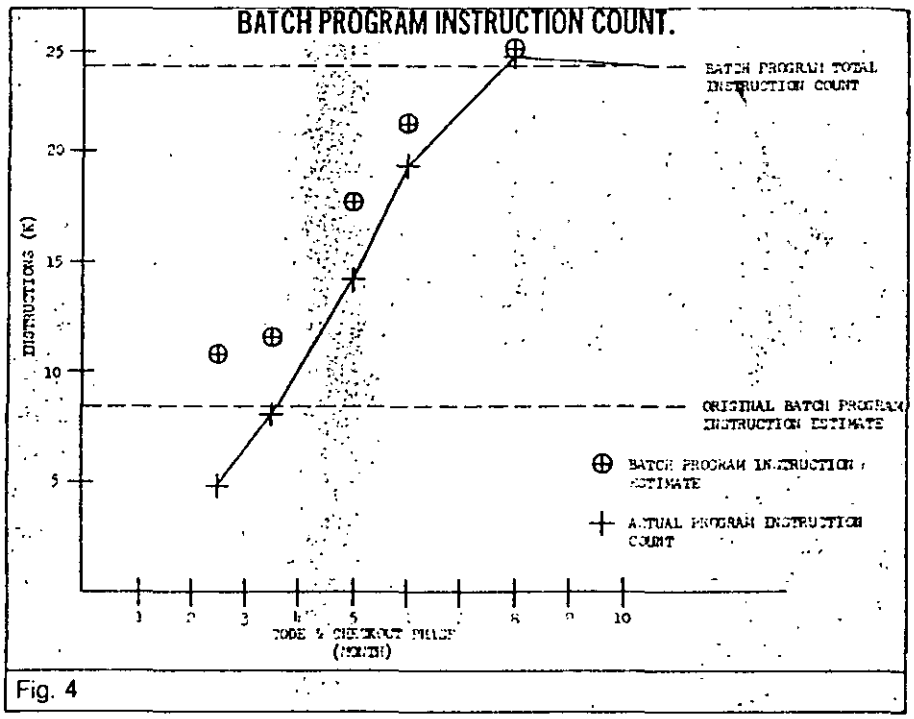


Fig. 4

DAVID S. IWAHASHI

Mr. Iwahashi is a staff engineer in software design and development at Lockheed Missiles and Space Co., Inc., in Sunnyvale, Calif. Since joining LMSC in 1966, his projects have included systems analysis and design, developing and maintaining programming, and software standards and requirements.

Kathleen S. Mendes is Senior Project Analyst in the Communications and Computer Sciences Department of the Exxon Corporation. Ms. Mendes holds the B.A. degree in mathematics from Manhattanville College and the M.B.A. degree from the Graduate School of Business of Rutgers University. In her present position, she has participated in the research, development, and instruction of Exxon's Structured Systems Analysis Technique, and has contributed to the establishment of a long-term strategic plan for software technology. Ms. Mendes has been a part-time faculty member of the Management Sciences Department of Kean College, and has published numerous Exxon proprietary documents.

tivities in the Construction stage of application building. In the mid-1970s, Exxon began to apply the results of this work to its own environment. Its main purpose was to establish an integrated set of system development techniques. The initial effort led to the introduction of PST (Program Structure Technology) to Exxon's worldwide community of system developers. PST, adapted from Michael Jackson's "Program Design Technique," could provide a precise description of a program and effectively communicate the program's design and behavior.[1] Its graphical notation was a key factor in the widespread usage of PST, and it became a well-established standard within two years.

In the mid-1970s, empirical evidence began to confirm the important role Analysis plays in the building of high quality computer applications. The results of one study indicated that more errors are introduced into a system as a result of failures in Analysis than as a result of failures in any other system building phase.[2] The conclusion of another study was that Analysis errors are more costly to correct and have a greater impact on the effectiveness of the system than errors introduced during either Design or Construction. Significantly, the effects of Analysis errors do not cease with the implementation of the system, but carry over to Maintenance.[3]

Therefore, any methods that improve the quality of Analysis increase the effectiveness of the system development process. Such methods, by contributing to more efficient use of time and by reducing the need to gather additional information as the project progresses, also increase the productivity of systems analysts — usually the most highly paid members of the data processing staff. In the late 1970s, Exxon's computer scientists focused their efforts on developing an Analysis methodology. The purpose of this methodology was to provide a disciplined approach for Analysis, as PST had done for program design and implementation.

Concurrently, application analysts, who were using PST, began to experiment with the flexibility of the notation. They began to use the PST diagramming style during the Analysis stage of system development to specify requirements. Based on the success of this experimental work, Exxon's computer scientists decided to use this approach to formalize system requirements analysis. Thus, the PST notation was brought forward from Construction to Analysis, and it was generalized to provide the basis for the present SSA technique.

Having established a commonality of approach and notation, SSA development proceeded in an evolutionary manner. Existing theory and concepts were drawn upon as needed. They were then adapted and integrated into the technique. At major checkpoints in its development, SSA was tested on pilot projects. This testing contributed significantly to the quality and usability of the present technique.

SSA and Other Requirements Analysis Techniques

The development of a number of requirements analysis and specification techniques during the past several years indicates the importance of these techniques to the data processing industry. Some of the more well-known specification approaches are Softech's SADT ("Structured Analysis and Design Technique"), Yourdon's "Structured Analysis Technique", Sarson and Gane's "Structured Systems Analysis", and the University of Michigan's ISDOS ("Information System Development and Optimization System").[4]

The fundamental difference between SSA and these methods is one of orientation. SSA approaches Analysis from the viewpoint of the business. The other techniques approach it from the perspective of data processing. SSA can represent completely in a clear, concise model all aspects of the business operation. Other methods are incapable of providing such complete business models.

Another significant difference between SSA and other structured techniques is the graphical notation. The SSA graphical approach consists of several diagram types, while the techniques of Softech, Yourdon, and Sarson and Gane rely upon a single style

of diagram. Analysts using SSA have indicated the advantages of its diagrammatic approach. It gives more insight into the business by providing different views, each of which contributes to an integrated business model. Each business entity is described by a notation that is appropriate for that entity's level of decomposition and for its role in the model analysis to follow.

SSA: A Description

SSA is an analysis technique used to understand the business operations of planned computer applications. Its purposes are to model, document, and communicate to users the requirements of computer applications, and to specify the capabilities to be delivered by the recommended system. It consists of a *graphical language* — a precise notation and set of diagrams which collectively are used to model a business — and a process — a step-by-step method for building and verifying the model. These components provide a formal, self-documenting approach to the Analysis stage of system development. The graphical language supports information gathering and analysis. The process assists in modeling the operation as it currently exists. The model is then modified to include the new functions provided by the computer application. The revised model and the supporting text replace the traditional specification document.

SSA has been used at Exxon for two years. During this time it has been effective in a wide range of applications that have varied in size, complexity, and orientation. It has been scaled to suit the needs of individual projects, and has been used effectively in a diversity of environments — including both business information systems and technical computing applications.

SSA's Graphical Language

The SSA graphical language provides a versatile vocabulary for model building and analysis. As shown in Figure 1, the style of

Figure 1 SSA Graphical Language

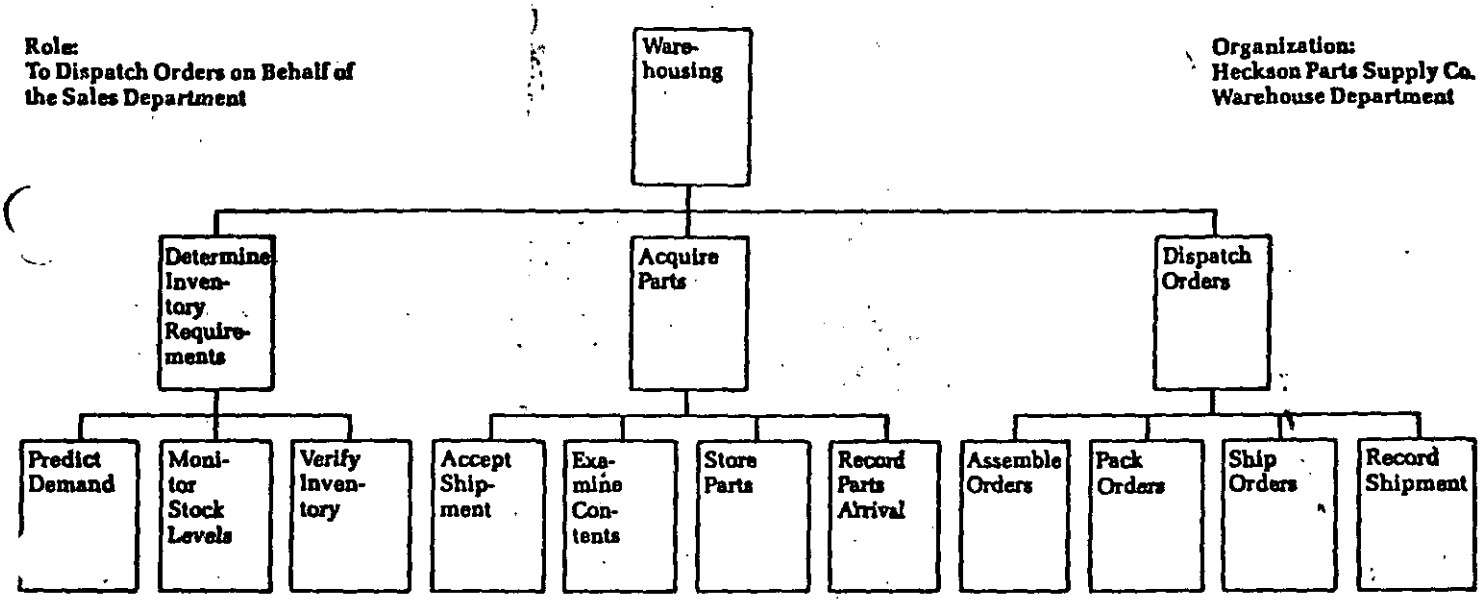
		Business Entities	SSA Diagrams
Business Operation	Is Described in Terms of	Business Functions	Global Business Model
		Responsibilities	Function Matrix
		Information Flows	Information Flow Diagram
		Business Activities	Detail Business Activity Model
		Business Information	Data Structure Diagram
		Business Terms	Glossary

the model consists of several diagram types: hierarchies, matrices, and network flows. Figure 1 also illustrates the relationship between the SSA model diagrams and the business entities they describe.

A business model can be represented with the following diagrams and definitions:

- *A Global Model* to describe the overall *logical relationships* among the *business functions* in the organization under study.
- *A Function Matrix* to define the *responsibilities* for each function identified in the Global Model.
- *An Information Flow Diagram* to represent the *flow* of business information.
- *A Detail Activity Model* to describe how *activities*, which correspond to the lowest-level functions in the Global Model, are carried out.
- *A Data Structure Diagram* to describe the logical structure of *business information* as viewed by the user.

Figure 2 Global Business Model



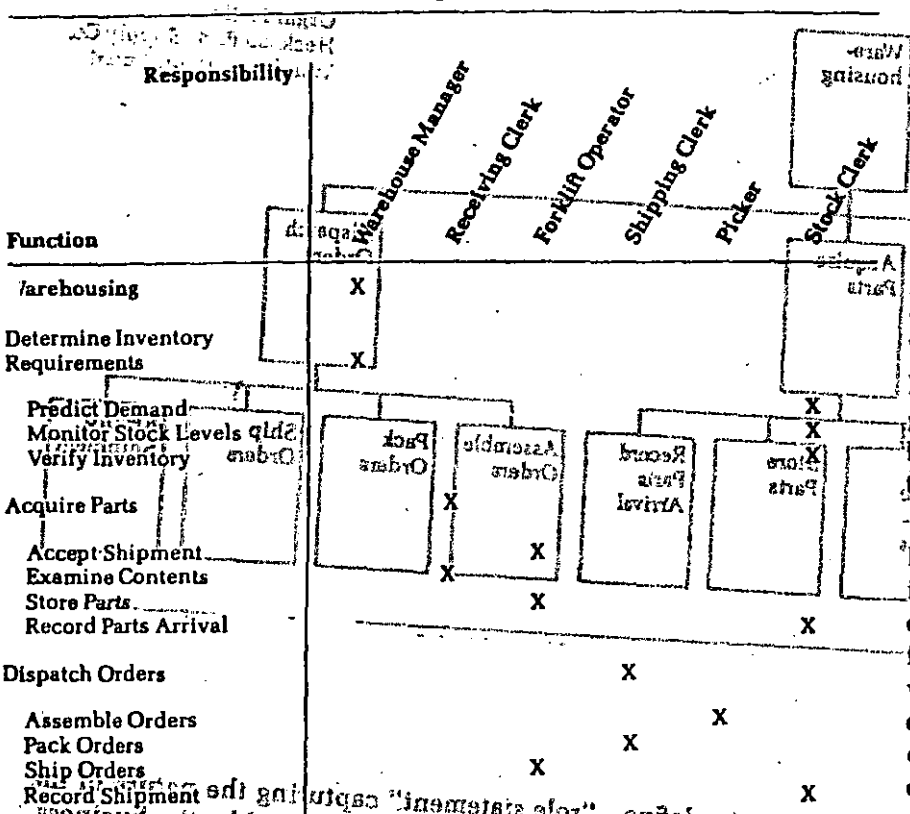
—*A Glossary of Business Terms* to define terminology common to the business operation.

The remainder of this section describes SSA's graphical language. Subsequent sections explain how to build model diagrams and how to use them to identify and specify solutions. Each diagram is discussed in the recommended sequence of construction.

Global Model. The Global Model of a business is a functional decomposition (a breakdown of business *functions* by purpose) described in a hierarchy of three to five levels. (Figure 2 shows a Global Model of a warehousing business.) In SSA, a *business* is defined as a logical set of functions which exists to provide a product or service (e.g., Warehousing). A *function* is a group of logically related activities (decisions or tasks) required to manage the resources of the business (e.g., Assemble Orders). In addition, a Global Model contains a single-sentence "role statement" capturing the nature of the product or service supplied by the business, the client or market it supports, and its economic commitment. ~~The Global Model concept was developed at Exxon by integrating Constantine and Yourdon's and Myers's functional decomposition techniques with concepts derived from IBM's "Business Systems Planning Methodology."[5]~~

Function Matrix. A Function Matrix maps the logical organization of the Global Model to the physical organization chart. (Figure 3 shows a sample Function Matrix which corresponds to the Warehousing Global Model in Figure 2.) All of the business functions are listed down the left side of the figure. Across the top are the *responsibilities*, i.e., the job descriptions, organizational units, or personnel accountable for the performance of the function. An "X" at the intersection indicates which functions are performed by each group.

Figure 3 Function Matrix
Warehousing



scribe the relationships among the parent functions of the subtrees. These relationships are referred to as high-level flows.

The notation used in Flow Diagrams represents information or material flows; business functions; information stores (passive repositories of information or material which can be either automated or manual); sources or destinations of information or material outside of the business operation; and sources or destinations of information or material outside of the network but inside of the business operation. The basic format of Flow Diagrams derives from the "Data Flow Diagram" concept developed by Yourdon and DeMarco. However, their usage and notation are modified in SSA.

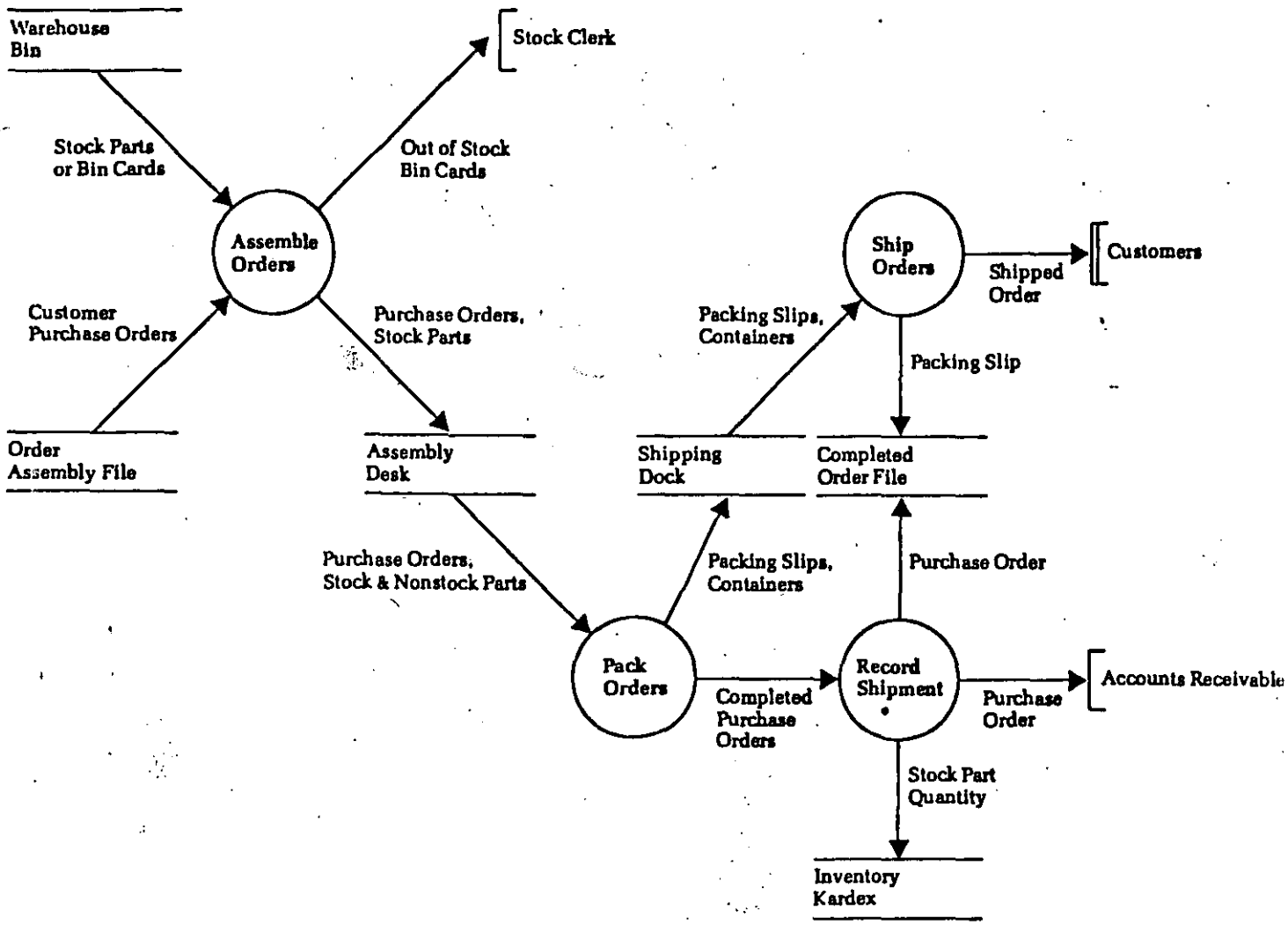
Detail Activity Model. A Detail Model specifies the relationships between activities and conditions under which activities are performed. An activity is a well-defined unit of work — a task or a decision (e.g., notify stock clerk if part is out of stock). A Detail Model is drawn for each of the lowest-level functions on the Global Model. Figure 5 is an example of a Detail Model for the function, Assemble Orders. It is a hierarchical breakdown by function, which is annotated to describe relationships among activities. Its symbols represent such relationships or conditions as hierarchic membership (with an implied or unspecified sequence); repetition; exclusive alternatives; inclusive alternatives; explicit sequence; parallel activities; release or critical timing condition; and termination activity. The original concept of the annotated hierarchy was derived from Michael Jackson's Program Design Technique. Exxon extended Jackson's notation to the Detail Model's description of business processes.

Information Flow Diagram. A Flow Diagram is a network representation showing the flow of business information. A flow is the transfer of information or material between business functions. The sources or destinations of these flows can exist within or outside of the business. At a minimum, one Flow Diagram is drawn for the lowest-level functions of each Global Model subtree. Figure 4 is an example of an Information Flow Diagram for the Dispatch Orders subtree of the Warehousing Global Model. It describes where information is used, what it is used for, and how the functions interact. Flow Diagrams can be annotated with statistics on volumes, resource usage, and critical timing. Optionally, Flow Diagrams may be drawn to de-

Data Structure Diagram. A Data Structure Diagram is prepared for each unit of business information that is identified on the Flow Diagrams. Business information, as defined in SSA, includes organized data aggregates such as documents, files, products, and intangible forms of information (e.g., order

22

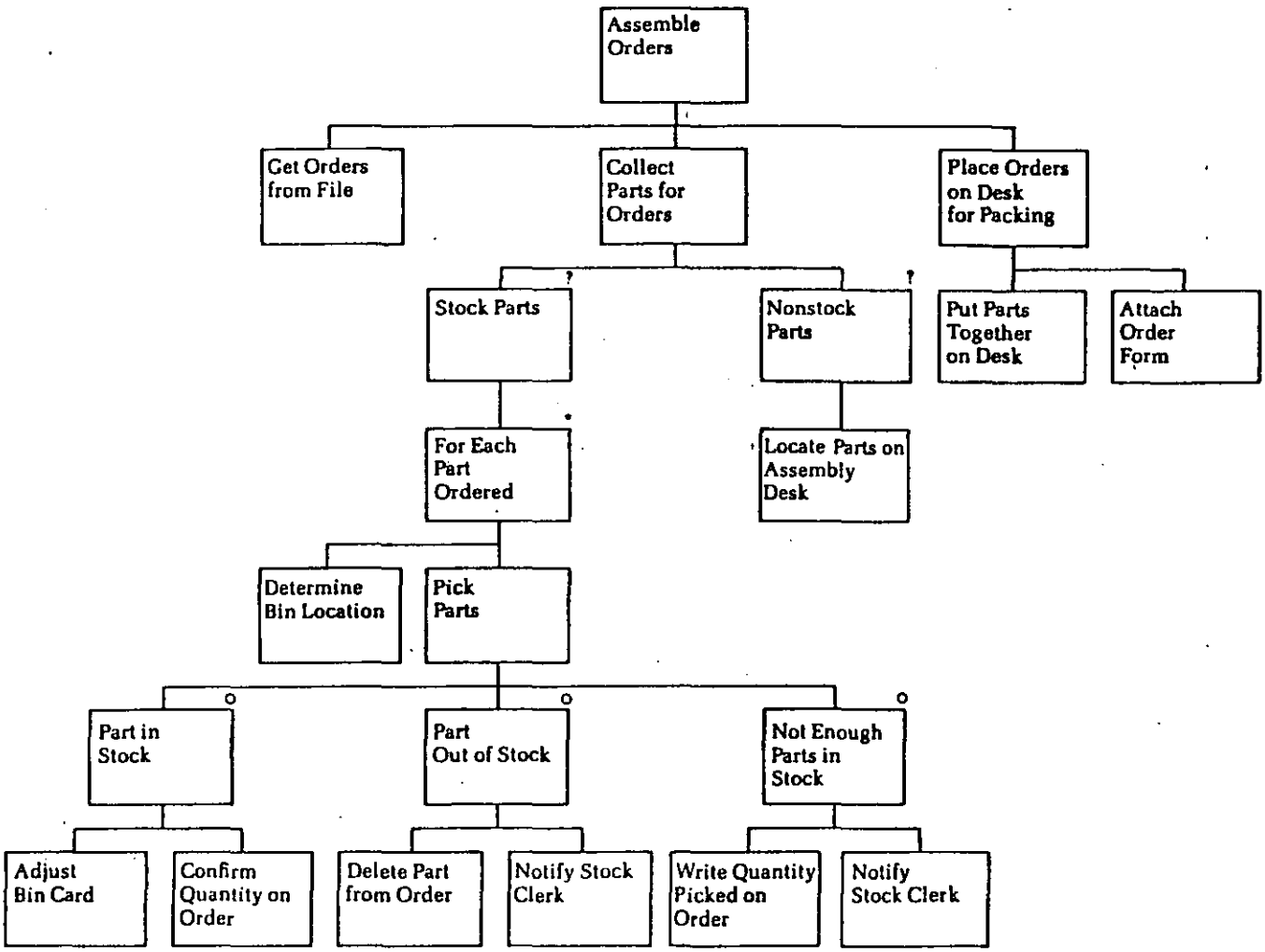
Figure 4 **Information Flow Diagram**
Dispatch Orders



Key:

- → — Information or material flows
- ○ — Business functions
- ═ — Information stores
-] — Sources or destinations of information or material outside of the business operation
-] — Sources or destinations of information outside of the network but inside the business operation

Figure 5 **Detail Business Activity Model**



Key:

blank — Hierarchic membership with an implied or unspecified sequence

* — Repetition

o — Exclusive alternatives

? — Inclusive alternatives

> — Explicit sequence

| | — Parallel activities

— Release or critical timing condition

↓ — Termination activity

forms, credit criteria). Figure 6 shows an example of a Data Structure Diagram of an order form as viewed by the clerk who takes orders. It is used to gain a deeper understanding of the information required to support the tasks and decisions involved in operating the business. A Data Structure Diagram is a hierarchical data decomposition, which may be annotated with statistics on data volumes and frequencies. It describes the "parent-child" relationship of the data by indicating hierarchic membership, repeated occurrence, mutually exclusive alternatives, and inclusive alternatives. It also includes a symbol for a dimension operator which is used in special matrixlike cases. Its concept and notation originated from Michael Jackson's Program Design Technique. As used in SSA, it has been generalized and expanded.

Glossary of Business Terms: A Glossary is compiled throughout the model building process. It is used to eliminate multiple definitions and ambiguities, and it serves as a basis for discussion and clear communication between users and analysts.

The Process of SSA

The SSA process is a step-by-step approach that guides the analyst and user through systems analysis. It has both the flexibility needed to develop a structure of business operations and the formalism needed to build a model. It also provides criteria for evaluating the model's correctness.

Figure 7 is a schematic representation of the SSA process. It shows the sequence of building the model diagrams. This sequence uses a top-down approach that starts with the most general (the Global Business Model) and proceeds to the most detailed (the Data Model). The figure also shows the recycling loops needed to refine the diagrammatic representations. During information gathering, the business model is the medium of communication between the user and analyst. The model is completed when it is confirmed that it fits the users' logical views of the business. Once verified, the model becomes the basis for identifying problems and formulating solutions.

SSA moves business and systems analysis from an intuitive to a teachable, structured, and concrete approach. This advance is the result of a well-defined process that assists the analyst in *building*, *verifying*, and *analyzing* a business model.

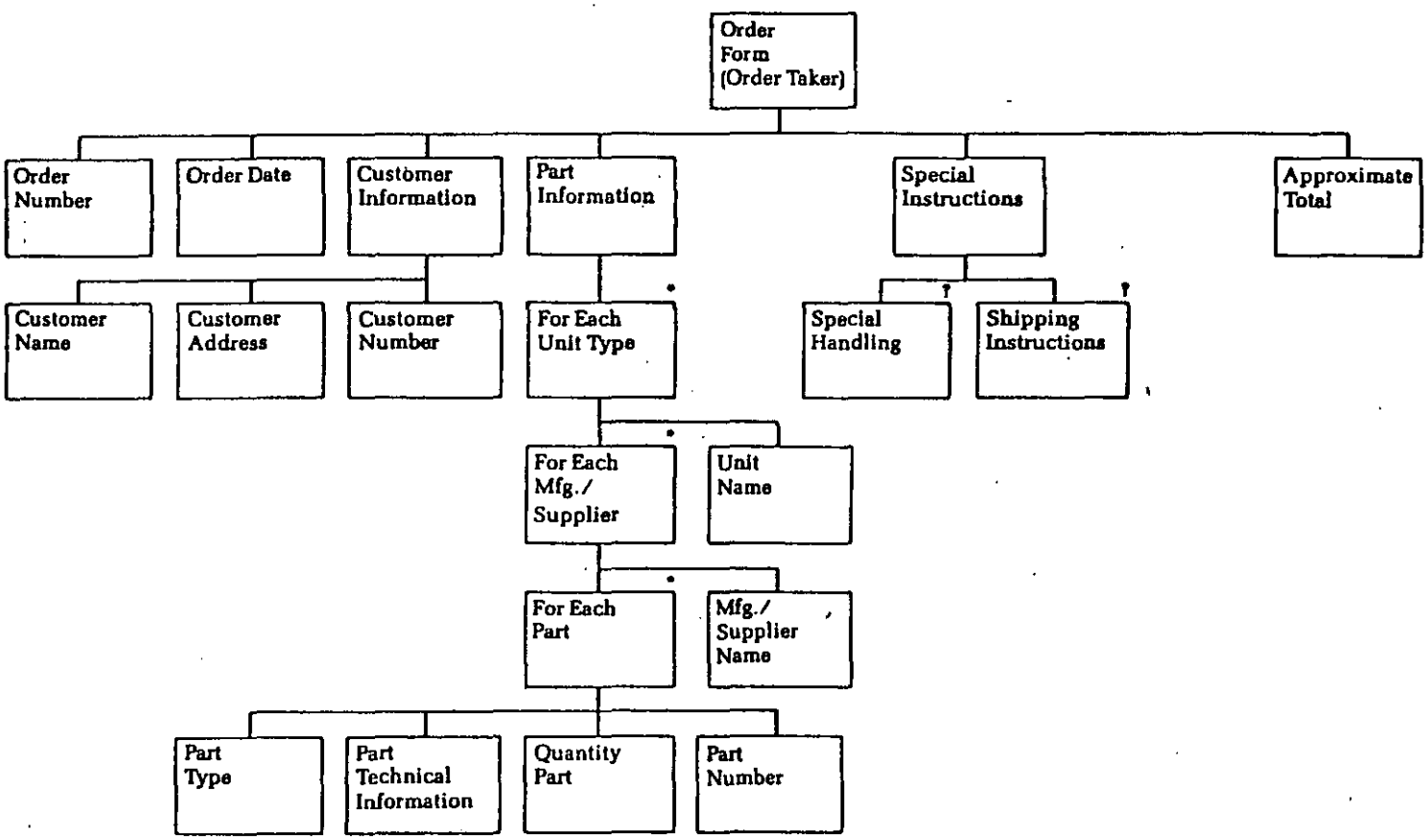
Model Building. Building an SSA business model is a cooperative effort between analyst and user. The diagrams are constructed during fact-gathering interviews, and serve to describe the current business in the project reports. The analyst uses SSA guidelines to elicit the relevant information, record key features, and eliminate irrelevant detail. The interviewing structure fosters consistency in communications between analysts and users. The diagram building approach establishes the set of information to be gathered and thereby provides a framework for each interview. Preestablished questions concerning business objectives, problems, future plans, and business environment forecasts can be drawn upon as needed.

SSA provides procedures to determine the sequence for building diagrams. Model diagrams generally are constructed in the order indicated in Figure 7. However, flexibility exists at the lower levels. For example, in a data-driven business, the Data Model is built before the Detail Model. Nevertheless, the relationships between diagrams always remain intact. Depending upon the size and objectives of the project, SSA can be used to produce a scaled description of the business. This description includes, at minimum, the Global Model and Information Flow Diagrams.

The rules and syntax for using the SSA notation are the grammatical foundation of the SSA graphical language. For instance, these rules provide that special symbols may not appear on the Global Model; a role statement must be a simple sentence; functions and activities must be described in verb-object form; and error conditions should be excluded from Flow Diagrams.

SSA also provides guidelines for decom-

Figure 6 **Data Structure Diagram**



Key:

blank — Hierarchic membership

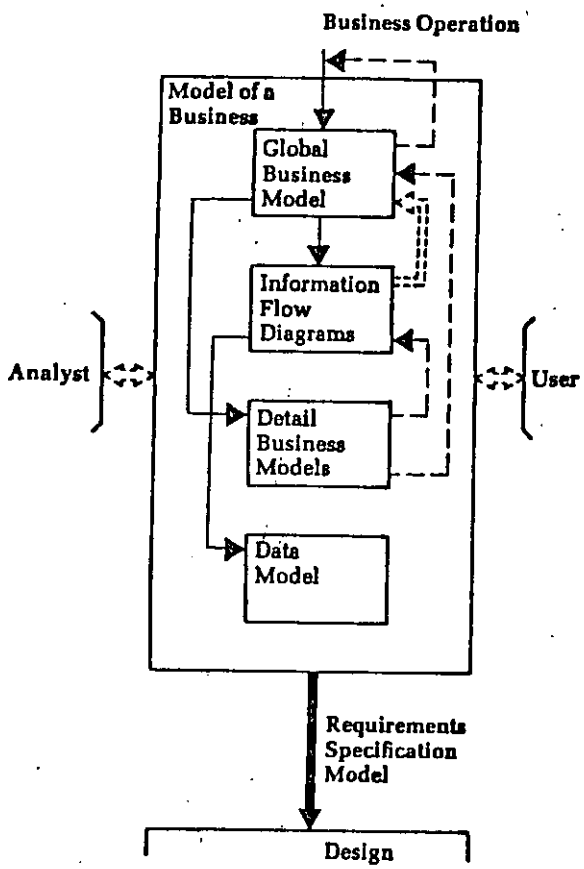
• — Repeated occurrence

o — Mutually exclusive alternatives

? — Inclusive alternatives

& — Dimension operator — used in special matrixlike cases

Figure 7 SSA Process



posing functions and activities. For example, the following rules determine the functions on the first level of the Global Model hierarchy:

— The first function must cover the requirements aspects of the business; that is, it must show how resource planning and acquisition are carried out (e.g., Determine Inventory Requirements in Figure 2).

— At least one supporting function must be identified that describes the business's provisions for maintaining its service or product (e.g., Acquire Parts in Figure 2).

— The last function must be a disposition function which describes the business's termination of responsibility for a product or service (e.g., Dispatch Orders in Figure 2).

Similar rules apply in the breakdown of the Detail Model. Without criteria such as the above, functional decomposition can be difficult, requiring subjective judgments and experience.

Model Verification. Building the model diagrams is an iterative process. As such, criteria are applied to the model to determine if and how the model can be improved. Once the criteria have been satisfied, the model is ready for confirmation by all users. Closure is established when all levels of users have verified it.

Criteria to determine whether the model has been decomposed to the appropriate level are applied to the model diagrams. For example, Flow Diagrams that have multipath connections (two functions that pass information back and forth) suggest that further decomposition of the Global Model is required. Or, if a multipurpose function (one with multiple flows in or out and having more than one information transformation) can be identified, then the model requires a further breakdown.

Criteria are also applied in order to test for the correctness of the graphical representation. An analyst can evaluate the model diagram, for instance, by seeing whether or not each function is labeled descriptively. The function should not be defined in terms of its output flow, as described in the Flow Diagram, but should be named to reflect its *purpose*. The analyst also can ask: Have laws of completeness and logical consistency been met? For example, does every information store have an input and output? Are the specified inputs of a function sufficient to produce the indicated output? Is all flow-related information appropriately represented on both Flow Diagrams and Detail Models?

Adherence to the SSA model building and verification process yields a technically cor-

rect, self-documenting description of the business operation. Moreover, an analyst's use of SSA leads to models that are the equivalent of those produced by others who use the technique.

Model Analysis. After the SSA model of the business has been verified by users, it becomes the basis for *diagnosing business problems, identifying computerization opportunities,* and *specifying the requirements model.*

Diagnosing Business Problems
The first step of this analysis addresses those aspects of the business that have been cited as problems by users. In proceeding, it is important to evaluate the organizational and control structure of the business. This analysis can be made by comparing the Global Model and Function Matrix with the organization chart. The comparison shows if the business is organized functionally, if responsibility and control are clearly allocated for each function, and if the geographic distribution of the business makes sense in terms of functional interaction.

It is also important to estimate work loads and peak loads. One can apply simple formulas, using the statistics collected on the Flow Diagrams, to determine if functions are under or overworked; to uncover the source of backlogs; or to trace the cause of high error rates. The Detail Models can be used to find ways to reduce demands on people. Eliminating unnecessary tasks or transferring certain tasks to the computer, for instance, could accomplish this reduction. The Flow Diagrams and Data Structure Diagrams can indicate ways to increase the capacity of staff, such as improving documents, forms, and human-computer interfaces.

Finally, it is important to establish data requirements. The model diagrams can be used to determine the content and structure of the information required for each business function. By contrasting the data actually used by a function (as represented in the Detail Model) with that passed to it (as described in Flow Diagrams and Data Structure Diagrams), one can identify redundant, inconsistent, incomplete, and improperly structured data.

Identifying Computerization Opportunities
It is often necessary, in solving business problems, to determine where the use of computers is feasible and cost-effective. In making this evaluation, SSA applies the concepts of Keen and Scott Morton's "Framework for Information Systems" to the analysis of the SSA Detail Model diagrams.[9] By applying this framework to the Detail Model, the structured activities that are candidates for automation can be readily identified. Moreover, the Detail Model can show where it is possible to implement Decision Support Systems. Thus, the Detail Model is a convenient medium for communicating to the user how the computerized procedures will support the business function.

Specifying the Requirements Model
Generally, the analyst proposes several alternative business solutions and documents them in separate SSA models. After the user has chosen one, the Requirements Specification Model must be prepared in detail. This model is the final product of Analysis. It represents how the business will operate when the new manual or computer procedures are implemented.

The Requirements Specification Model includes organizational requirements that are depicted in the Global Model and Function Matrix; information requirements that are described in the Flow Diagrams and Data Structure Diagrams; and procedural requirements that are detailed in the Flow Diagrams and Detail Models. Thus, it provides a comprehensive set of specifications for use during Design.

A Review of SSA Usage

SSA was originally developed as a formal technique for analysts to model both the business environment and the requirements of planned computer systems. However, as users gained experience with SSA, its potential applications increased. It has been prov-

Special recognition must be given to C. M. B. Anderson (Imperial Oil, Toronto, Canada), one of the primary developers of SSA. His enthusiasm, technical expertise, and consulting expertise were critical to the successful dissemination of SSA within Exxon. I would also like to acknowledge the contributions of C. Galpin (Exxon Corp., Florham Park, New Jersey). Through his SSA teaching and consulting experience, he made significant refinements and constructive extensions to the technique.

gresses, since SSA establishes a comprehensive information base during Analysis.

— SSA provides a basis for identifying opportunities for common systems. Its modeling capabilities enable it to show the potential for generic (or multiuse) systems. SSA models of several business operations can be easily compared or contrasted, and common functions can be readily identified. It is then possible to estimate the degree of tailoring needed to apply the common system to different uses. SSA is a convenient medium for gaining user commitment to the requirements of an application that may serve several user groups.

— SSA is an important tool in project management. Its systematic process provides natural checkpoints for measuring progress. Its notation offers a means of ensuring continuity of effort and consistency in communication even when there is staff turnover. Estimates of the duration, cost, and staffing needs of the project can be obtained earlier in the system building process than would be possible without SSA.

— SSA can adapt to the way in which a system is implemented, even though its specifications are independent of it. SSA is as effective in quickly establishing a set of threshold requirements for an evolutionary or prototyped system as it is in dealing with more traditional system building approaches.

The objective of Exxon's ongoing research efforts has been to develop a compatible set of formal methodologies for building systems — from Analysis to Maintenance. SSA and PST, consistent in their approach and notation, have established a foundation upon which further research will build.

References

1
See M. Jackson, *Principles of Program Design* (London: Academic Press, 1975).

2
See B. Boehm, "Quantitative Assessment," *Datamation*, May 1973, pp. 49–59.

3
M. L. Shooman and M. I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," *International Conference on Reliable Software Proceedings*, pp. 347–357.

4
See:
D. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, January 1977;
"SADT — Structured Analysis and Design Technique Overview" (Waltham, MA: Softech Inc., Form Nos. 9569–4, 956905, 1976);
L. Constantine and E. Yourdon, *Structured Design* (Englewood Cliffs, NJ: Prentice-Hall, 1979);
G. Myers, *Composite/Structured Design* (New York: Van Nostrand Reinhold, 1978);
C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques* (Englewood Cliffs, NJ: Prentice-Hall, 1979);
D. Teichroew and E. Hershey III, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, January 1977;
"Problem Statement Language Reference Summary" (Ann Arbor, MI: University of Michigan, ISDOS Project Team, Reference No. 79A51–0174–4, 1979). See also additional ISDOS documentation.

5
See:
Constantine and Yourdon (1979);
Myers (1978);
"Information Systems Planning Guide" (White Plains, NY: International Business Machines Corp., 1978).

6
See T. DeMarco, *Structured Analysis and System Specification* (New York: Yourdon Inc., 1978).

7
See Jackson (1975).

8
Ibid.

9
See P. G. W. Keen and M. S. Scott Morton, *Decision Support Systems: An Organizational Perspective* (Reading, MA: Addison-Wesley, 1978), pp. 79–98.

en effective, for instance, in business improvement projects that do not necessarily involve computerized solutions. Since its introduction two years ago, SSA has gained rapid acceptance by users and analysts throughout the worldwide Exxon organization. Its acceptance is attributable to three main factors: business and systems analysts have found its notation and process usable; it can be applied to diverse business environments; and the technique can be adapted to projects of varying sizes and levels of complexity. The following studies illustrate the strengths and versatility of SSA as both a business and a systems analysis technique.

Central Purchasing System. In this application, SSA was used to specify the requirements for a large-scale purchasing system of an internationally based Exxon affiliate. Analysts used the SSA technique to interview approximately 120 users from diverse business functions. The technique helped to structure the process of gathering information and to improve communications between analysts and users. Its value was evident in the high quality of the requirements specification produced.

Refinery Blending System. In this study, SSA was applied in a technical computing environment. It was used to describe a process control system which monitored a refinery's blending operation. This case illustrates the power and adaptability of the SSA modeling notation to scientific and engineering applications.

Planning Process Study. In this project, SSA was used as a business analysis technique. It served as a documentation tool to describe the planning operation of the enterprise. Its analytical capabilities enabled it both to help identify ways of improving the planning process and to determine if the current operations could be extended to support strategic planning.

Financial Modeling System. In this case, a multipurpose financial modeling system that served many different business functions was evaluated. SSA was used to describe the business environment supported by the system and to foster communication among users. It resulted in greater understanding by each user of other users' needs, and in agreements on definitions and calculations to be used in the system.

Conclusion

SSA has been used widely in diverse business environments. Based on this experience, it is evident that the technique is valuable in several respects.

— It improves the quality of analysis and of requirements definition. SSA's systematic approach to analysis formalizes what was a very unstructured activity. It enhances the skills of experienced analysts and fosters the involvement of even the most reluctant business users during requirements analysis. Moreover, it enables analysts to prepare Requirements Specification Models that are superior to traditional narrative specifications. The resulting systems are of higher quality than before; they are more capable of representing and adapting to changing business needs.

— SSA facilitates communication between analysts and users. Their interviews are a cooperative effort to construct an SSA model of the business. The analyst does not play the role of interrogator in these interviews; rather, the participants play equal roles. In addition to improving communication during the information-gathering stage, SSA helps to describe to the users the proposed system's effect on the organization.

— SSA increases the productivity of analysts and users, enabling them to use their time more effectively and efficiently. Analysts can learn more about the business in less time. Also, requirements can be documented concurrently with data collection and analysis; SSA does not treat this task as a separate activity. Finally, there is less need to gather additional information as the project pro-

...ssed as a procedure of hierarchical functional design by ... programming projects can be analyzed successfully on pro... and module levels. It is shown that program design is ... more efficient by applying Hierarchy plus Input-Process-Output (HIPO) techniques at each level to form an integrated view of all levels.

HIPO and integrated program design

by J. F. Stay

By the mid-1970s, programming appears to be reaching a stage of refinement and cost-effectiveness such that regular business management and control methods can be applied to it. Top-down development, structured programming, chief programmer teams, structured walk throughs, Hierarchy plus Input-Process-Output (HIPO), and structured design have taken us a long way toward transforming "a private art into a public practice."[1] As a result, a body of programming knowledge and methods that are teachable and practicable has been building. This paper discusses the integration of several programming methods by means of an example.

hierarchical functional design

Most of the change in system development has been directed toward the programming effort. Although programming errors are the direct cause of many rework costs, perhaps one third of the rework ultimately can be traced back to errors in the analysis and design phases of a project.[2] Since maintenance can account for as much as seventy precent of all programming costs, more emphasis must be placed on the quality of analysis and design. Structured design and HIPO are useful techniques for organizing the application design process. This article describes how these two methods can be integrated to create a hierarchical functional design. This integrated method allows an application system to be specified from the highest functional level of a conceptual design to the lowest detailed level in a coded routine, using a single method and format. Both the concepts and the techniques of hierarchical functional design can be employed effectively throughout a development cycle in the following phases:

- Requirement definition.
- System analysis.
- System design.
- Program design.
- Detailed module design.
- System and program documentation.

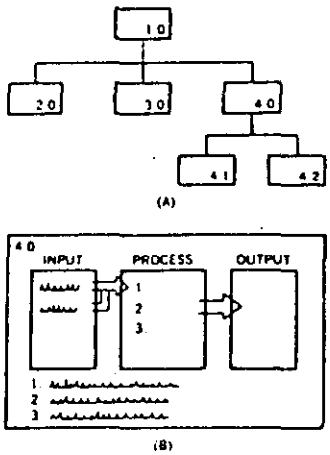


Figure 1 HIPO: Hierarchy plus Input Process Output (A) Schematic diagram of a hierarchy chart (B) Schematic diagram of an input-process-output chart

This paper presents a basis for the thought processes involved in designing a system through the use of these techniques. Although this paper does not explain the design techniques in detail, the references provide practical help.

Two techniques for achieving functional design are the following:

- Hierarchy plus Input-Process-Output (HIPO)
- Structured design

HIPO, a technique for use in the top-down design of systems, was developed originally as a documentation tool. HIPO charts continue to serve as the final programming documentation. HIPO consists of two basic components: a hierarchy chart, which shows how each function is divided into subfunctions; and input-process-output charts, which express each function in the hierarchy in terms of its input and output. These two types of charts are illustrated in Figure 1.

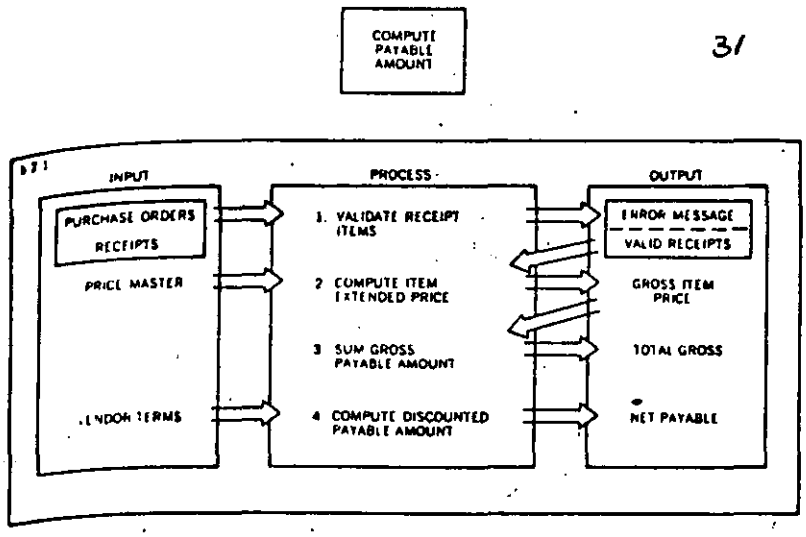
example

The HIPO design process is an iterative top-down activity in which it is essential that the hierarchy chart and the input-process-output charts be developed concurrently, so as to create a functional breakdown. The example of COMPUTE PAYABLE AMOUNT is followed through its development process as part of an accounts payable system.

The first step is to describe a given function as a series of steps in terms of their inputs and outputs. The input-process-output chart for the example of COMPUTE PAYABLE AMOUNT is shown in Figure 2.

Having completed the input-process-chart, it is possible to move to the next level of the hierarchy. The COMPUTE PAYABLE AMOUNT hierarchy now appears as shown in Figure 3. It is now possible to develop an input-process-output chart for each of the boxes at the level shown in Figure 3. If additional definitions are required, the recommended approach is to make each line on the input-process-output chart a box on the next level of the hierarchy. This process causes the developer to focus on the level of function that is being defined

Figure : HIPO chart for COMPUTE PAYABLE AMOUNT function 32



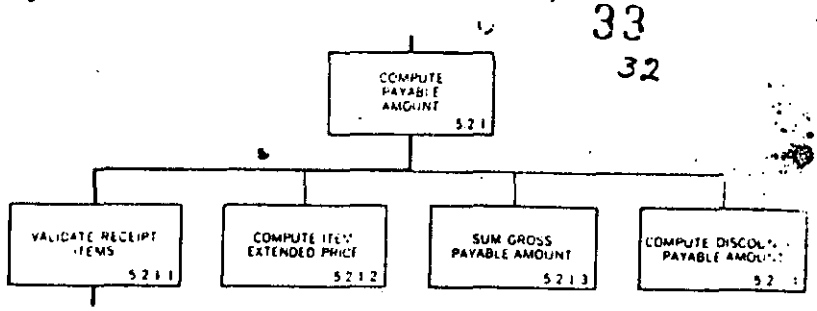
Structured design

The hierarchy plus input-process-output charting is that part of the hierarchical functional design process by which a problem description is made. The other component is structured design. Structured design[3] is a set of techniques for converting from a problem description to a functional, modular program structure. Myers[4] uses the term composite design in reference to the "structure attribute of a program, in terms of module, data, and task structure and module interfaces." This goes beyond the concept of modular design and addresses how a program module is designed, the proper scope of function of a module, and the appropriate communication between modules.

Two concepts of structured design are *module strength* (relationships within a module) and *module coupling* (relationship between modules). The way in which functions are grouped within modules determines the strength of the modules. A module may consist of a group of related functions, such as all editing functions, functions grouped according to the procedure of the problem, or all functions related to a data set.

Functional strength is the grouping of all steps to perform a single function. Although any application may contain modules that have some or all of these strengths, the objective is to produce modules that have functional strength. Functional strength does not apply only to the lowest modular level. A module may call other modules to perform subordinate functions, but if the upper-level module performs a single function and performs it completely, the module probably has functional strength. For example, the module COMPUTE PAYABLE AMOUNT might consist of the following statements (in pseudo-code):

Figure 3 COMPUTE PAYABLE AMOUNT function hierarchy




```
COMPUTE-PAY-AMT (RECEIPT, PAYABLE, RETURN-CODE);
      DO WHILE MORE-ITEMS; GET PRICE-MASTER;
         CALL VALIDATE-ITEM (RECEIPT-ITEM);
         CALL EXTEND-PRICE (RECEIPT-ITEM, PRICE);
         CALL SUM-AMOUNT (PRICE, TOTAL-PRICE);
         CALL COMPUTE-DISCOUNTED (TOTAL-PRICE,
         DISCOUNT-RATE, PAYABLE);
      END DO;
END;
```

This module performs very few functions by itself. However, it transforms one input (RECEIPT) into one output (PAYABLE) completely; at the same time it does no unrelated processing. Therefore, this module is said to have functional strength.

Interactions between modules, termed *module coupling*, may be as varied as interactions within a module. The extreme of module coupling exists when one module directly modifies an instruction in another module.

The preferred relationship between modules is *data coupling*. In data coupling, each module simply passes application data, usually as parameters to the next lower-level module. Use of artificial switches and indicators is avoided. When a calling module passes switches, indicators, or other control information to another module, these items must only communicate the status of the calling program. The calling program should not assume that it knows what the called program will do, based on this control information. A serious problem in program maintenance results when a simple change to a module changes the meaning of an item of control that has an unsuspected effect on the logic flow of one or more other modules.

This paper does not treat structured design in depth but only with sufficient detail to carry the concepts of functional strength and data coupling into earlier stages of the design process. The reader may find References 3 and 4 to be of valuable assistance in module design.

Hierarchical functional design addresses mental processes that may be applied to the application analysis and design tasks. Hierarchical functional design applies the design concepts of functional strength and data coupling to the functional decomposition and graphic techniques of HIPO to provide a single methodology that allows an application design to develop in an orderly manner from a clear statement of the requirement to an intelligible, well constructed set of application functions. A system that is designed through the use of hierarchical functional design is implemented in a top-down manner. Modules should have a single entry point and a single exit point; they should be small; and they should use the SEQUENCE, IF-THEN-ELSE, and DO-WHILE concepts of structured programming. Hierarchical functional design can be used in all phases of the development cycle, and thereby provide a visable system that is suitable for a design walk through. The chief programmer team concept is also supported, since functional breakdown with clearly defined interfaces allows modules to be delegated to developers or to other teams, with the common understanding that is required for programs to integrate properly.

Hierarchical functional design employs the following three design concepts:

- A functional design in which the computer solution is structured in terms of the user's function.
- An iterative process in which each level of design is validated against the level above it.
- Conceptual levels of design, in which each level emphasizes a particular aspect of the problem solution.

functional design

A computer system can be viewed as a single function that can be divided (or decomposed) into a hierarchy of sets of successively lower-level functions until the elemental functions are described. An understanding of the meaning of the term "function" is necessary for further discussion. *Function* can be defined as an action upon an object, or, for our purposes, the transformation of some input data to some return data.[3] A statement of function describes what is done rather than how it is done. Since a function is also singular, it is defined with a simple declarative statement that consists of only one verb and one object. Both the verb and the object may be conditional.

A function should also have the characteristics that are defined for structured design. A function should be completely defined in one place, and relationships among functions should be primarily data relationships. Thus the concepts of structured design are valid for designing systems as well as modules. Functional

design, then, consists of stating what is to be done in terms of data in and out. A high-level functional statement is reduced to a set of more detailed low-level statements, in a verb-object format. The set of lower-level statements must equal the function of the higher-level statement. In the accounts payable example that is used in this paper, the high-level function is COMPUTE PAYABLE AMOUNT. This function is stated in terms of its input data (purchase receipt) and its output (net payable amount). In this case, the output may simply be passed to another function. The function COMPUTE PAYABLE AMOUNT is then reduced to the following four functional statements:

VALIDATE RECEIPT ITEMS
COMPUTE ITEM EXTENDED PRICE
SUM GROSS PAYABLE AMOUNT
COMPUTE DISCOUNTED PAYABLE AMOUNT

Each of these statements can then be expressed in terms of its input and output data. This set is an explicit statement of the steps required to perform the function COMPUTE PAYABLE AMOUNT.

This is only one example of the way in which a function may be subdivided. The structuring of subfunctions requires analytical skill and imagination, and each analyst may define the subcomponents of a function slightly differently. It is important, however, that the definition of a function determine the functions that are subordinate to it. The function COMPUTE PAYABLE AMOUNT could not legitimately have a subordinate function that, for example, updates the inventory balance.

Iterative process

The design of an application should be an orderly growth process from inception to implementation. During development, frequent reviews should be conducted so that a given design always meets its objective. Design has usually been done at least twice before a system is complete and running. First, a functional design has been made to provide an understanding between the user and the programming department. Then a logic design has been made, from which programming could proceed. With hierarchical functional design, the function is the logic, and redundant effort may thus be avoided.

Hierarchical functional design is an evolving, top-down process. The first step is a translation of a statement of need into a functional statement of system objectives. As information about a required system is gathered, that information is organized according to the functional structure. The first statement should contain display screen formats, report layouts, perhaps a two-level hierarchy chart, and a single level of HIPO charts. The analyst should walk through these charts with the customer, to verify

fy that this level of design conforms with the requirement. As the process of design moves to areas such as file access methods, record layout, and message traffic, definitions in successively lower levels the hierarchy may cause upper levels to change. This is typical of the program development process and is the reason for continuing discussion with the customer. This is the iterative process—the refining of the higher levels of design as the more detailed levels are developed. As the iteration of detail progresses downward, the impact on the top-level design should become minimal. Because of the successive iterations of assessing the upward impact of design decisions, the result is a stable, intelligible, and maintainable design.

Levels of the design hierarchy

As an application is divided into functions and each function in turn is subdivided, the hierarchy proceeds toward greater detail. All levels of the system are described as functions, and can be grouped into three categories, proceeding from the broadest to the finest level of detail, as follows:

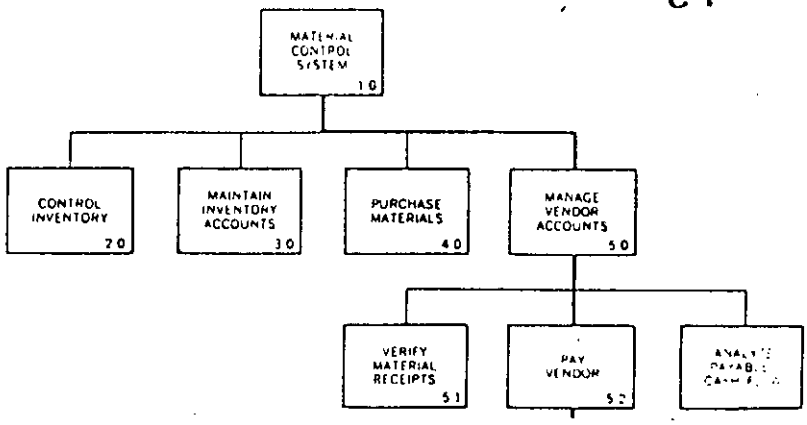
- System
- Program
- Module

These three levels are conceptual, and are not a physical part of the design. Each conceptual level may represent multiple levels on the hierarchy chart, and any given box on a chart may be both the bottom of one conceptual level and the top of the next level.

system level

The system level of the hierarchy contains the major component parts of the application, and is the view that a department manager might have of the application. A system might contain multiple system levels: Accounts payable is a component of a material control system, and in turn, the subsystems of accounts payable itself would be contained within the system level. This level of hierarchy is started by structuring the original statement of requirement for an application. The analysis phase of a project may result in a system-level hierarchy such as that in the example in Figure 4. Each box in Figure 4 can be stated in functional terms, and can be represented by a HIPO chart. Although the terms of the user may differ somewhat from those on the chart, the functional statement should be used because it is more explicit than the common term. For example, the term ACCOUNTS PAYABLE may be simply a subset of the general ledger, or it may be a complete system for managing the payment of vendor accounts. The functional term MANAGE VENDOR ACCOUNTS makes the objective of this application more clear.

Figure 4 System level of the accounts payable application 36 37



The system level normally does not include the representation of any executable computer instructions, but rather it provides a conceptual view of the application. Input and output are defined in terms of forms, files, and reports, which are a user's view of the data. If it appears that the user manager's view of the structure of an application does not provide the basis for program design, it should be mentioned that one of the primary objectives of functional design is to have a single view of the application that represents both the user's requirement and the program design. Although functions defined at the top level may be grouped differently for program design, they should all still exist with the same basic relationships in the computer implementation. This is the key to building systems that can be readily maintained and enhanced.

program level

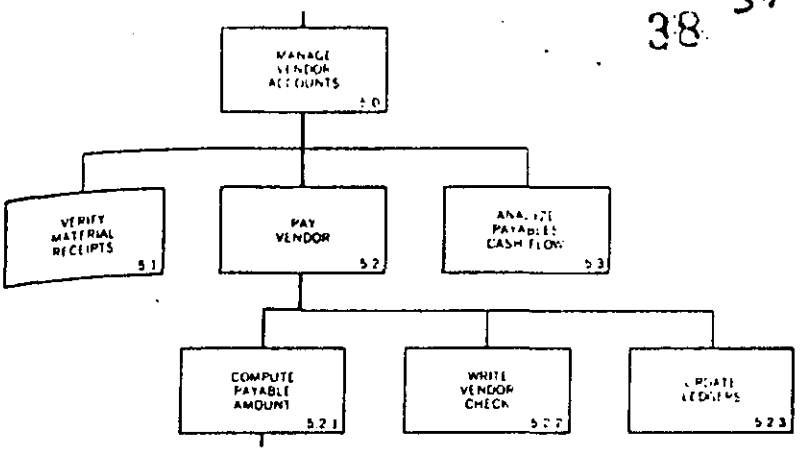
The program level of the hierarchy shows the highest level of segmenting of the computer system. The program level may also be characterized as the end-user level, and represents the level of tasks initiated by a terminal operator in an interactive application, or batch programs in a batch application.

In the accounts payable application, consider the boxes below MANAGE VENDOR ACCOUNTS as tasks or programs. An example program level (one of the boxes in the accounts payable application) is shown in Figure 5. The program or task level is a result of the general design phase of a development project. Input and output for this level are usually defined as records or groups of records, messages, and report lines. Since this level normally represents executable computer instructions, it is recommended that program and module names be assigned to the boxes.

module level

Detailed program design occurs at the module level. Since design is an iterative process, detailed module design may expose

Figure 5 Program level of the accounts payable application



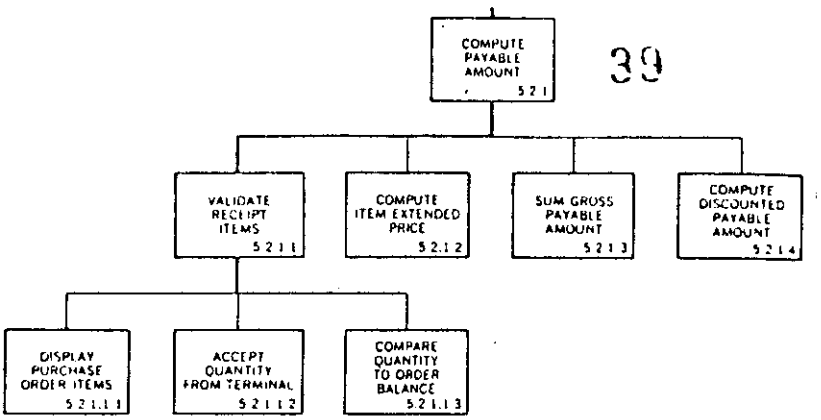
flaws or required restructuring of the more general design. It is essential that the upper-level HIPO charts be revised and revalidated before continuing with the design process. Design modification may be required when a low-level change causes the higher-level design to do something different from that which the user requires.

The module level represents an executable segment of program code that is usually compiled as a unit. This unit of code is typically called an "object module" to distinguish it from a "load module," which may be created by linking several functional modules together. At the module level of the hierarchy, the design is sufficiently detailed that program code can be written directly from the design. In the accounts payable application, two levels of modules are shown in Figure 6 below COMPUTE PAYABLE AMOUNT, which can be related to the program level in Figure 5.

The module level is the product of the specification phase of development. Input and output at this level are fields of data or parameters from or to other modules. The module level should represent a functional statement that can be completely grasped within a normal attention span. When translated into executable code, a module should usually contain fewer than fifty lines of structured high-level language statements. The HIPO chart at the module level may contain structured English (pseudo-code) statements to explain complex logic. In addition, where necessary, the extended description section of the HIPO chart may provide implementation notes as discussed in Reference 5.

The purpose of these conceptual levels is to reflect the objectives of users of the documents. The system level must be stated in terms that are relevant to user management. The module level is organized in such a way as to allow the programmer to write

Figure 6 Module level of the accounts payable application



code. The program level provides the vehicle of communication between the system level and the module level by giving detail to the user and a higher-level view to the programmer.

Hierarchical functional design in a virtual system

The discussion thus far has considered the structure of an application as an aid in the design of an intelligible, maintainable system. A current major concern in system design is that of providing efficient performance of interactive applications in a virtual system environment. Performance tuning in a virtual system is a complex science that involves relationships among hardware, system software, and application design.[6, 7] Optimizing the performance of the application programs alone does not result in an efficient system. A conscious effort of tuning all the components that affect performance is required because programmers, for example, often attempt to write efficient—sometimes complex—code without regard for the way in which the modules may ultimately affect performance.

When viewing the structured design of an application, one can see readily that functional decomposition does not reflect the performance requirements of a transaction-driven application. Hunter,[5] for example, makes clear the issue of the compounded effect of excessive paging. He defines the working set as the twenty percent of the code that does eighty percent of the work, and advises that the "working set should become the focus of application tuning." The design process, if correctly executed, can produce modules each of which requires less than a single 4K byte page of storage. On that basis, the task of application tuning becomes an effort of identifying the most active modules, rewriting those modules for efficiency (if necessary), combining modules into logically related pages, and fixing active pages in main storage.

performance tuning may, in extreme circumstances, require the ...ical modification of a few critical modules. Such modifica... may include changing a CALL (transfer of control) to a ...py (compile-time inclusion) or even the integration and re-...structuring of modules. It must be clearly understood that only a very few modules ever seriously affect the performance of most systems. Thus, if an application is structured, the necessary tuning can be done consciously with proper control. With this perspective, even extreme coding techniques may be justified for those few modules that must be efficient to avoid performance degradation.

Concluding remarks

The improvement of the system development process requires innovation in two areas: the development of a discipline that involves a set of structured techniques, and an understanding of the theories on which that discipline is based.

Significant progress has been made in developing a discipline that, in time, should make program design and implementation an engineering skill. Structured programming provides basic building blocks for code development, much as electronic circuit development can be based on a set of predesigned, basic electronic components. HIPO and structured design are first steps in bringing that type of discipline to the design stage of application development.

It has been the intention of this article to address the following mental processes:

- Identification of function.
- Functional decomposition.
- Iterative design.
- Module relationship.
- Delayed performance optimization.

By applying the concepts of hierarchical functional design to the disciplines of HIPO and structured design throughout the analysis and design process, several of the following possible benefits are typically realized:

- User understanding and agreement on functional content are made easier.
- Missing or inconsistent information is identified early.
- Functions are discrete and are therefore more easily documented and, if necessary, modified.
- Documentation is accomplished with a single effort rather than multiple efforts at different stages of development.

40

- Module interfaces are simple and therefore reduce the probability of logic errors.
- The resultant design supports structured, top-down coding.
- Maintenance and enhancement are more transferable because the system can be easily understood at all levels.

Since these processes are ways of thinking about the design activity, it is often difficult to measure objectively the effect of using this knowledge. By applying these principles to a development project, however, one becomes aware of their value.

CITED REFERENCES

1. H. D. Mills and F. T. Baker, "Chief programmer teams," *Datamation* 19, No. 12 (December 1973).
2. P. Moody and R. Perry, "Application development cycle problems," *Proceedings of Guide 40*, Miami Beach, May 18–23 (1975).
3. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal* 13, No. 2, 115–139 (1974).
4. G. J. Myers, *Reliable Software Through Composite Design*, Mason/Charter Publishers, New York, New York (1975).
5. John J. Hunter, "Rethinking application programs, key to VS success," *Computerworld*, March 25, 1975.
6. J. G. Rogers, "Structured programming for virtual storage systems," *IBM Systems Journal* 14, No. 4, 385–406 (1975).
7. H. A. Anderson, Jr., M. Reiser, and G. L. Galati, "Tuning a virtual storage system," *IBM Systems Journal* 14, No. 3, 246–263 (1975).

GENERAL REFERENCES

1. *HIPO—A Design Aid and Documentation Technique*, Order No. GC20-1851, IBM Corporation, Data Processing Division, White Plains, New York 10504.

2. *Structured Programming Independent Study Program*, Order No. SR20-7149, IBM Corporation, Data Processing Division, White Plains, New York 10504.

3. J. D. Aron, *The Program Development Process, Part I, The Individual Programmer*, Addison-Wesley Publishing Company, Reading, Massachusetts, 31–36 (1974).

Reprint Order No. G321-5031.

The *Second* Structured Revolution

Edward Yourdon
YOURDON inc.
1133 Avenue of the Americas
New York, New York 10036

March 1979

1. INTRODUCTION

After nearly ten years of discussions at computer conferences and in trade journals, nearly everyone in the data processing field has at least *heard* of such technologies as structured programming, structured design, structured analysis, top-down implementation and structured walkthroughs. Indeed, many programmers and analysts will tell you confidently that they *understand* the structured techniques, and that they are faithfully *using* the techniques.

Alas, it just isn't so. The structured techniques are *not* as widely understood or used as one might imagine from reading the popular literature. A recent survey in *Datamation*, for example, indicated that only a small fraction of the major EDP organizations in southern California were making consistent use of structured programming and design.

Even worse, many large organizations are finding that the large sum of money they spent attempting to convert their staffs to a "structured" approach has been wasted. After all the fanfare of the "structured revolution" — and after spending a considerable amount of time, energy and money on training, textbooks and guest lectures by the prophets of structured programming — many organizations are finding that their people are still developing the same expensive, bug-ridden, slipshod, unmaintainable software that they were developing before structured programming was introduced.

Why? Is it because our programmers and analysts are too stupid to learn new techniques? Is it because our programming languages and operating systems don't support the new techniques? Or is it just that the Forces of Evil are determined to prevent us from ever being able to develop quality software?

No! The real problem is *management.* It's my opinion that management is to blame for the failure of the structured revolution. And only management can ensure that the *second* revolution — a revolution that is now beginning in several EDP organizations — will be successful.

The purpose of this paper is to explore the failure of the first revolution in more detail — so that organizations that are just beginning to think about structured programming can avoid those failures. Then we will present a battle plan for the second revolution, a plan for implementing structured techniques economically and successfully.

2. AN OVERVIEW OF THE STRUCTURED TECHNIQUES

Let's begin by summarizing the structured techniques themselves. Obviously, a one-paragraph summary won't explain all of the technical details of each technique; you may find it helpful to refer to some of the standard reference texts on the subject for more details.

Structured analysis is a collection of graphical tools that help the systems analyst document the *functional specifications* of a system. Instead of the classical *narrative* specification — which, like a Victorian novel, tends to be monolithic, verbose, redundant, ambiguous and boring — structured analysis allows the analyst to portray the user's system as an abstract, partitioned, top-down "model" of the system-to-be. The primary tools of structured analysis are data flow diagrams, data dictionaries, data structure diagrams and structured English.

Structured design is a collection of guidelines and strategies that help the designer select the models and the module interfaces that will most economically implement a software system that has already been well-specified. Structured design consists of documentation tech-

niques such as *HIPO* and *structure charts*. It also consists of "evaluation criteria," notably *coupling* and *cohesion*, which help the designer distinguish between good designs and bad designs. And structured design also includes "cookbook strategies" (e.g., transform analysis and data-structure analysis) that help the designer systematically generate good designs for common types of problems.

Structured programming is a coding discipline based on the concept (first published in the mid-1960's) that all program logic can be built from combinations of:

- "sequential" instructions — e.g., MOVE, ADD, SHIFT, etc.
- IF-THEN-ELSE
- DO-WHILE

Often referred to as GOTOless programming, structured programming probably the oldest of the structured techniques and is now taught in many universities.

Top-down implementation is a strategy for building software systems that have been specified and designed. In sharp contrast to the classical "bottom-up" approach (in which modules are tested first, followed by program testing, subsystem testing and system testing), the top-down approach calls for implementing high-level "executive" modules *first*, with the lower-level modules implemented as "stubs" or "dummy modules" (e.g., modules which return constant outputs or which exit without doing any processing). The top-down approach has the advantage of allowing the EDP personnel to demonstrate a "skeleton" version of the system to the users at an early stage, so that the users can see if they are getting the system they really want.

Walkthroughs are generally thought of as a peer-group process for reviewing the products of structured analysis, structured design and structured programming. Managers, "big bosses" and other outsiders are generally not invited to walkthroughs since their presence often results in a review of the producer rather than the product. The primary purpose of the walkthrough is to ensure the quality and correctness of the product (whether it be a functional specification, a design or a page of COBOL coding); secondarily, the walkthrough approach serves as an excellent training device, and it makes the project less vulnerable if a key technician leaves in the middle of the project.

3. WHY DID THE FIRST REVOLUTION FAIL?

Before we can explore successful strategies for introducing structured software development techniques, we need to explore in more detail the reasons why they weren't introduced successfully the first time around. The following reasons seem to be most important.

Inadequate selling of the techniques

In many companies, most people in the EDP organization remain blissfully unaware of all new technological developments: they're quite happy to keep developing systems the way they have been for the past umpteen years. So when the resident technical hot-shot comes back from an ACM conference with some new ideas about structured programming, he's likely to be hooted down by his colleagues. When an aggressive young project manager returns from a GUIDE conference with some interesting case studies about successful uses of structured analysis, his colleagues will mumble something about, "Well, that's fine for XYZ company, but it won't work here!"

Indeed, it's even worse in some organizations, where the practice of "sending out the scouts" is used to examine new technologies. The "scout" often turns out to be the worst of all salesmen in terms of convincing his organization to begin using something that he's learned about in a conference or a symposium — indeed, he's often so brash, so intellectually superior and so downright arrogant that the only way his organization can stand him is to send him away regularly to such meetings!

Even if the scout is reasonably diplomatic and articulate, he may have great trouble selling the structured techniques to his colleagues simply because, by virtue of working within the organization, his colleagues view him as not expert enough to make them change their way of doing things.

Inadequate training

In the organizations where "structure," as it's often known, does gain some degree of acceptance, massive training programs are often begun — but as my distinguished colleague Gerald Weinberg points out, it was often done with a "sheep dip" approach: large numbers of programmers and analysts are herded into an auditorium, where they are, in a sense, dipped in a bath of structured snake-oil. That is, an outside "expert" (who may be little more than a salesman for his firm's products or services) is asked to give a short half-day presentation on "Everything You're Ever Likely to Need to Know About All That Structured Stuff" to the assembled group — whereupon they are herded back to their desks and told that their development schedules have just been cut in half because the expert has said that "structure" doubles programming productivity!

Even when the EDP organization provides thorough, comprehensive training, there's no guarantee that the students have learned what they were taught. And, more important, there is no guarantee that they will *use* what they learned. As a teacher, I have had far too many situations where a student came to me at the end of a course, shook my hand and said, "Well, that was sure a nice course — now I can get back to drawing flowcharts in my maintenance shop again!"

Inadequate management support and follow-through

Ultimately, the problems mentioned above have to be described as "management" problems: *management* has got to be involved in the "selling" of structured systems development techniques, and *management* has got to be aware of the need for proper training.

But even more important than the training and the consulting (both of which can be accomplished with a great deal of fanfare and a great expenditure of money) is the *follow-through*. There are a number of large EDP organizations that did a reasonably good selling job when the structured techniques first became popular in the mid-1970's, and that also provided a substantial amount of high-quality training. But when it came time to put the techniques to work on *real* projects, things began to fizzle. It's not hard to imagine the kinds of situations that develop:

- A project manager learns about the structured techniques *after* his project team has finished its analysis and half of its design — so he decides that he might as well ignore *all* of the structured techniques, even though top-down implementation and walkthroughs would be eminently practical.

- A project manager decides that the project he is about to undertake is far too sensitive and far too critical to risk using "new-fangled" techniques.
- Conversely, a project manager decides that his project is too small and too simple to warrant using such "high-powered" development techniques.
- The project manager begins using the new techniques but then runs into "mid-project panic": because of the learning curve associated with the introduction of *any* new set of ideas, the project teams thrashes around and spends a considerable amount of its time arguing about esoteric theory — so the project manager, in a moment of panic, decides to abandon the structured techniques and return to the more familiar methods of developing software

4. HOW TO DO IT RIGHT

If these are the problems, then how can an EDP organization introduce the structured techniques — or, for that matter, any other new technology — successfully?

To a large extent, the answer is: *approach it the same way you would approach a "real" EDP project.* Just as a "normal" project requires distinct activities of analysis, design, implementation and testing, so the introduction of structured development techniques requires the same activities.

An analogy might help illustrate the point. A competent EDP professional would never dream of walking into a user organization and thoroughly disrupting its way of doing business with the introduction of a new computer system — not without a great deal of planning, analysis and design. And since the EDP organization is (when viewed from the outside) a *business,* it should be approached with the same caution that one would use when approaching a user's business.

What's needed, then, is the following:

- A dignified "selling" effort to convince the EDP staff that the structured techniques are, if nothing else, at least worth investigating.
- A formal analysis activity.
- A formal design activity.
- A formal implementation activity.
- A formal testing activity.

Each of these activities is discussed in more detail below.

5. THE SELLING OF STRUCTURED X

Don't be misled into thinking that "selling" is unnecessary or unimportant. The larger the organization, the more likely it is that there will be pockets of resistance (or downright reactionary ignorance!). If your local hardware vendor has already done a selling job for you, that's fine — but if not, make sure that the key people in your organization are aware of the virtues of the structured techniques, and what their benefits are likely to be.

In most organizations, you'll find that you have to approach several distinct *levels* of people as you do your selling job. The boundaries between these levels are somewhat fuzzy, but basically they are as follows:

- Top-level EDP management and, on occasion, managers *above* the EDP organization. This might include people like the VP of Finance, or the Director of MIS or the Manager of Computers & System Development, etc.
- Middle-level managers — e.g., first-level team leaders, second-level project leaders, etc.
- Technicians — i.e., the programmers and analysts who will be most directly involved in the day-to-day *use* of the structured techniques.

Top management, in my experience, is usually not involved in the day-to-day crises of software development. If a project is behind schedule, *they* won't be spending their nights and weekends in the computer room with the harried crew of programmers. So they may not feel the same sense of urgency about the introduction of new development techniques that the first-level managers do.

Not only that, they're probably totally uninterested in — and unaware of — the technical details of structured analysis, structured design and structured programming. Indeed, one of the most frustrating experiences I've ever had is trying desperately to explain such technical issues in language simple enough and "jargonless" enough that an insurance vice president could understand it, only to have him say, "Gee, that sounds awfully simple — in fact, it sounds like plain old common sense. Are you sure our people haven't been doing this structured stuff all along?"

So skip the technical details. Concentrate instead on arguments of *economics* — i.e., how much money will structured programming save the organization? How will maintenance costs be reduced? How many fewer of those long-haired, unwashed programmers will we have to hire this year if the new techniques are introduced? After all, if it's a high-level manager that you're talking to, the economic — that is to say, the *business* — ramifications of the structured techniques are what concern him the most anyway.

Middle management is where you'll have the most trouble selling the techniques, *particularly* if you try to sell the techniques in a negative way! It's very easy for a veteran project manager to get the impression that the introduction of structured analysis or design or programming is an implied criticism of the way he's done his job for the past fifteen years. And if he gets that impression, he'll be able to come up with a hundred different reasons for why "It'll never work here!"

When I use the term "middle-level manager" I have the image of someone who has come up through the ranks — that is, someone who was once a programmer or systems analyst. That can be both good and bad. Such a manager ought to be able to understand the technical aspects of structured development techniques, *but* there's always the danger that he will try to think about the structured techniques in the context of RPG and the IBM 1401 — and that's when you'll start getting complaints like, "That stuff sounds too inefficient — why I remember that a subroutine call used to take 6.573 microseconds . . . "

There's one other thing about the middle-level manager: for the most part, he *is* involved in day-to-day crises of software development. More to the point, he's plagued by pressures of deadlines. He may agree with you that the new techniques can reduce maintenance

costs — but right now he's faced with a deadline that he can just barely meet if he applies all of his conventional techniques. And he's not allowed to add any people to the project; and management will have a coronary if he suggests slipping the deadline; and besides, all the money in the training budget has been allocated to learning the mysteries of relational data base systems.

What can be done in a situation like this? Realistically, the answer is — not much. In the long run, the support and the budget and the extra people and the extended deadlines have got to come from higher levels of management. It's possible, of course, for the middle-level manager to gamble: he can try to pick a medium-sized project that doesn't have all the cards stacked against it — i.e., one that can always be rescued by what my colleague Tom DeMarco calls "the time-honored tradition of unpaid overtime." If you're a middle-level manager, though, and you decide to do something like this, you should be very aware that it's a gamble. For the first several weeks (or even months), when everyone expects your people to be busily coding and they're doing nothing but arguing about data flow diagrams and functional cohesion, about the only thing you'll be able to say is, "Trust me . . . it will all work out. Trust me."

Technicians also have to be sold — though my experience is that a great number of them are already convinced. Many of them have complained for years that "management" (that great nemesis, second only to users as a source of frustration) never has time to do the job right the first time around, but always has time to do it twice.

Of course, there will be senior technicians — the grizzled veterans who have been programming since 1952 — who will have many of the same reactions as the middle-level project managers: if it didn't work on the IBM 650, it certainly won't work now! And there will be moderately experienced technicians — with 2 to 5 years of experience — who will accept the structured techniques intellectually, but whose brains have been so badly warped by years of terrible programming languages (e.g., COBOL, FORTRAN, PL/I, RPG . . .) that they'll never *really* specify, design or code a structured program.

Indeed, it may well be the trainees — those with virgin minds — who are our only salvation. Obviously, trainees are (or should be) humble enough to do what they're told — and they can be trained to do it right from the very beginning. *If* — and this is a very large if!! — they are trained properly. It's surprising to see how many organizations are still teaching "basic" courses in programming, design and analysis that are riddled with ideas and techniques that are grossly obsolete — only to follow this up with "advanced" courses that preach all of the modern structured techniques. The student spends the first half of such advanced courses being asked to *unlearn* a substantial amount of material that he learned in the basic course — something that's obviously unpleasant and difficult to do.

One last suggestion about the selling of structured X: *do it top-down!* If top management doesn't support the new development techniques, there's not much point training the technicians. Except in places like Iran, grass roots revolutions don't seem to be in vogue these days.

6. THE ANALYSIS ACTIVITY

Perhaps the most important aspect of introducing structured development techniques is a formal *analysis* of the way the EDP organization develops software. In other words, the analysis activity should be attempting to answer the following questions:

- Do we have a problem in our EDP organization today?
- If so, what is the nature of the problem? Is it political? Is it hardware-oriented? Is it associated with the maintenance of systems developed 15 years ago?
- How serious is the problem? Or, to put it in a more positive light, how much money could be saved by introducing the structured techniques?
- How long would it take to convert the organization to the structured techniques? What is the "return on investment"?
- What are the risks? Is it possible that the structured techniques could destroy the organization? Will all of the newly trained programmers immediately quit and get better-paying jobs elsewhere?

In effect, this amounts to a "feasibility study" and a detailed examination of the way the EDP organization is presently doing business. Obviously, if the organization has no problems (either real or perceived), there's not much incentive for introducing the new techniques. *If* there is a problem, and *if* it's perceived as a serious problem, and *if* it appears to be related to the methods for developing software — *then* there may be a good argument for introducing the new techniques. But the argument should be made with the same careful analysis that one would use to convince a user organization to install a new-fangled computer system.

7. THE DESIGN ACTIVITY

After the analysis has taken place, the next step is *design.* Basically, this involves developing a *plan* for implementing the techniques — as opposed to a "shotgun" approach of training everyone in sight, issuing "structured edicts" that will probably be ignored anyway, etc. The design activity should address the following kinds of questions:

- Which of the structured techniques should be implemented first?
- Which parts of the organization should be exposed first?
- What kind of training is appropriate? For whom?
- What kind of pilot projects should be used to experiment with the new techniques?
- How can we manage and control the process of converting the organization to the new techniques?
- How can we measure the impact that the structured techniques will have on the productivity of the EDP organization?
- How can we strike a compromise between the deadline pressures of today and the long-range benefits of the structured techniques?

Obviously, these are not simple questions — and as you can imagine, there are no "pat" answers for any of them. However, my own experience with a number of organizations lead me to offer the following suggestions:

1. *Informal walkthroughs are a good way to begin.* In addition to the "obvious" benefit of reducing errors and increasing the quality of specifications, designs and code, the introduction of walkthroughs can help ensure some *consistency*

in the use of structured techniques by your staff. This is particularly important when a group of programmers and analysts all begin using structured programming, design or analysis at the same time — there is a great danger that each technician will interpret differently the techniques that he has learned from a textbook, a video training course or a classroom training experience.

2. *Don't begin with structured coding — start with analysis or design.* There's an obvious reason for this suggestion: with good analysis and design, you can tolerate mediocre code; on the other hand, brilliant code can't save a bad design or a fuzzy specification. There's a more important reason for making this suggestion, though: assuming that it takes one to two years (or more) to introduce the new techniques, there's some danger that the EDP organization may "run out of steam" after six months and stop introducing new techniques. If such is the case (and it's happened in a number of organizations, either because of economic considerations, a change in management or other subtle political reasons), then it would be sad to find that the only thing that had been accomplished was an improvement in coding techniques.

3. *Don't introduce too many new techniques at once.* Attempting to get a large group of people to adopt a dozen new techniques at once is a sure-fire way of guaranteeing that none of the techniques will be introduced successfully. Begin with just one or two new techniques — e.g., structured analysis and walkthroughs. Then move on to structured design and top-down implementation; then perhaps structured programming . . .

4. *Choose one or more medium-sized "safe" pilot projects.* The notion of a pilot project is familiar to most EDP organizations — that is, a formal experiment to see if a new technology really works. In our experience, the best pilot projects have been at least six person-months long; they have been projects that are "real" and visible to the organization. But at the same time, they have not been projects whose failure would bankrupt the organization! Indeed, the very best kind of pilot project, in many cases, is a "conversion" — i.e., rewriting an old system that was about to collapse anyway. Among other things, such a pilot project offers a basis for comparison (admittedly a somewhat biased comparison, but better than no comparison at all) between the old way of doing things and the new way of doing things.

5. *Groom some internal "structured gurus" for ongoing consulting assistance.* Another advantage of the pilot projects mentioned above is that it accomplishes *this* suggestion. It provides the organization with a group of people who have really *done* structured analysis, design and/or programming, as opposed to people who have done nothing more than just read about it in a book. This, in my opinion, is essential to the success of installing the new techniques in a large organization.

6. *Establish a group with responsibility for coordinating the introduction of the new techniques.* This can be the training department (though they often have zero credibility in the EDP organization), or the standards department (who, unfortunately, often have *negative* creditibility in the organization) or a special task force reporting to an appropriately high level in the EDP hierarchy. But the main point is obvious: The techniques won't be introduced by themselves — and, left to their own devices, each programmer or analyst will be-

gin practicing his own interpretation of that subset of the structured techniques that he wants to introduce. And nothing more!

8. IMPLEMENTATION

By implementation, I mean the process of actually training the staff, doing the "grunt work" of rewriting appropriate standards and doing the messy managerial/political work of enforcing the new methods of analysis, design and coding. Most of the actual work will be obvious, once the analysis and design activities have been completed (note that the same thing is true in a "real" EDP project).

One word of caution is in order though: the implementation process may go on forever. This is partly because of the turnover and staff growth that the EDP organization generally faces. It may also happen for another reason: if most of the programmers and analysts are engaged in pure maintenance work, there may be little opportunity (or motivation) to train them in structured analysis or structured design. Thus, if the new techniques are introduced solely to technicians working on *new* development projects, it may be five to ten years before it has filtered through the entire organization.

There is one other point to consider, particularly if the introduction of the new techniques is going to stretch over a period of five years, and that is: *the techniques themselves will continue to expand and change over the next several years.* We can certainly expect that structured analysis and structured design will be refined, enlarged and improved over the next several years — so the kind of training done in 1984 will most likely be different from the kind of training done in 1979 or 1980.

9. THE TESTING ACTIVITY

The activity most often forgotten or ignored by EDP organizations installing the structured techniques is *testing*. The basic objective of the testing activity is to continually monitor the organization to find out:

- whether the technicians have actually learned the techniques,
- whether they are using the techniques in a uniform fashion,
- how much improvement there has been.

This implies, then, that there needs to be a "quality assurance" group or an auditing group continually inspecting the methods used by the organization.

10. SUMMARY

Despite some of the fanfare and publicity in the EDP journals, the structured techniques are not "magic." Indeed, one could argue that they are nothing more than "common sense" — but the crucially important point is that the structured techniques represent a *standardized* common sense. And besides, as Will Rogers once said, "Common sense isn't common."

Whether they represent common sense or black magic, there is increasing evidence to confirm that structured analysis, design and programming can *substantially* increase productivity, maintainability and reliability of EDP systems. Certainly, any EDP organization whose existence depends heavily on the quality of its software is going to find its existence seriously threatened in the next decade if it does not adopt modern development techniques.

But the question, as we have seen in this paper, is *how* to introduce the new techniques. A small organization of geniuses can — perhaps! — implement the structured techniques on an *ad hoc* basis. But a large organization, with hundreds of programmers, can literally put itself out of business if it doesn't have a *plan* for implementing the techniques.

A summary of progress toward proving program correctness

by T. A. LINDEN

National Security Agency
Ft. George G. Meade, Maryland

INTRODUCTION

Interest in proving the correctness of programs has grown explosively within the last two or three years. There are now over a hundred people pursuing research on this general topic; most of them are relative newcomers to the field. At least three reasons can be cited for this rapid growth:

(1) The inability to design and implement software systems which can be guaranteed correct is severely restricting computer applications in many important areas.
(2) Debugging and maintaining large computer programs is now well recognized as one of the most serious and costly problems facing the computer industry.
(3) A large number of mathematicians, especially logicians, are interested in applications where their talents can be used.

This paper summarizes recent progress in developing rigorous techniques for proving that programs satisfy formally defined specifications. Until recently proofs of correctness were limited to toy programs. They are still limited to small programs, but it is now conceivable to attempt to prove the correctness of small critical modules of a large program. This paper is designed to give a sufficient introduction to current research so that a software engineer can evaluate whether a proof of correctness might be applicable to some of his problems sometime in the future.

THE NATURE OF CORRECTNESS PROOFS

Given formal specifications for a program and given the text of a program in some formally defined language, it is then a well-defined mathematical question to ask whether the program text is correct with respect to those specifications. The mathematics necessary for this was originally worked out primarily by Floyd[1] and Manna.[2]

It must be made clear that a proof of correctness is radically different from the usual process of testing a program. Testing can and often does prove a program is incorrect, but no reasonable amount of testing can ever prove that a nontrivial program will be correct over all allowable inputs.

Example

The approach to proving programs correct which was developed and popularized by Floyd is still the basis for most current proofs of correctness. It is generally known as the method of inductive assertions. Let us begin with a simple example of the basic idea. Consider the flowchart in Figure 1 for exponentiation to a positive integral power by repeated multiplication. For simplicity, assume all values are integers. I have put assertions or specifications for correctness on the input and output of the program. We want to prove that if X and Y are inputs with $Y>0$, then the output Z will satisfy $Z=X^Y$. This assertion at the output is the specification for correctness of the program. The assertion at the input defines the input conditions (if any) for which the program is to produce output satisfying the output assertion. Note that the proof will use symbolic techniques to establish that the program is correct for all allowable inputs.

The proof technique works as follows: Somewhere within each loop we must add an assertion that adequately characterizes an invariant property of the loop. This has been done for the single loop flowchart of Figure 1. It is now possible to break this flowchart into tree-like sections such that each section begins and ends with assertions and no section contains a loop. This is

...n in Figure 2 if one disregards the dashed-line boxes. We want to show that if execution of a section begins in a state with the assertion at its head true, then when the execution leaves that section, the assertion at the exit must also be true. By taking an assertion at the end of each of these sections and using the semantics of the program statement above it, one can generate an assertion which should have held before that statement if the assertions after it are to be guaranteed true. Working up the trees one then generates all the assertions in dashed-line boxes in Figure 2. Each section will then preserve truth from its first to its last assertions if the first assertion implies the assertion that was generated in the dashed-line box at the top. One thus gets the logical theorems or verification conditions given below each section. With a little thought it can now be seen that if these theorems can all be proven and if the program halts, then it will halt with the correct output values. In this case the theorems are obviously true. Halting can be proven by other techniques.

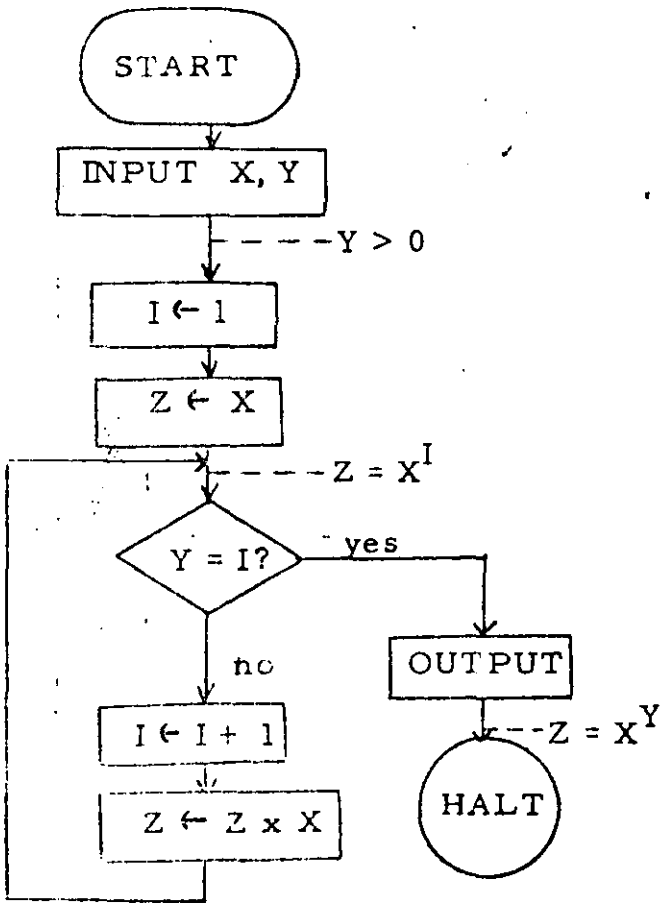


Figure 1—Exponentiation program

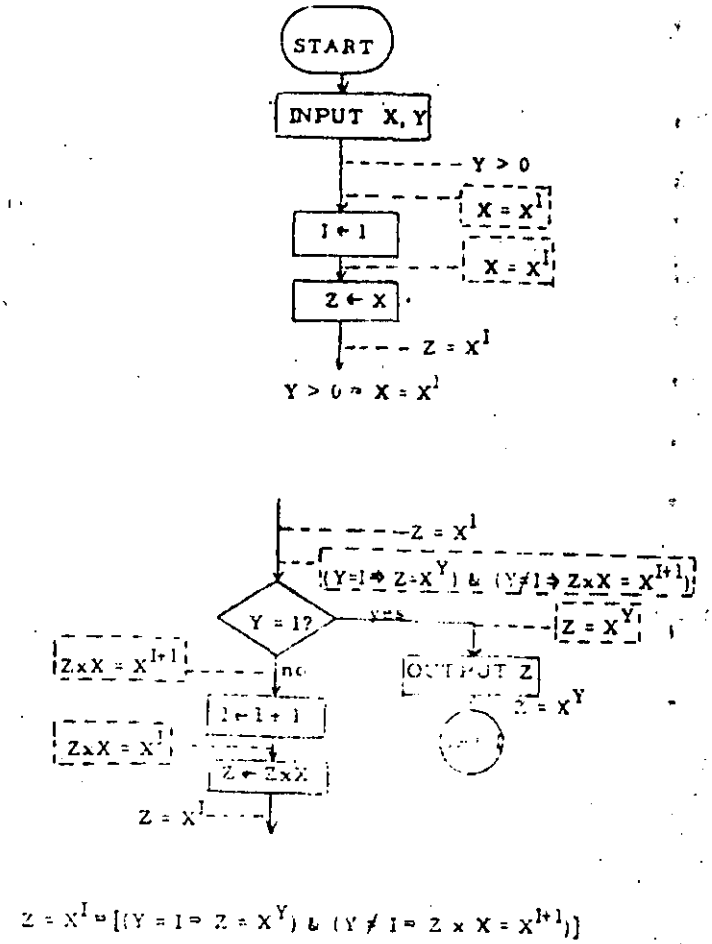


Figure 2—Sectioned flowchart

The careful reader will note that the input assumption $Y>0$ is not really needed for the proof of either of these theorems. This is because that assumption is really only needed to prove that the program terminates.

Inherent difficulties

This process for proving the correctness of programs is subject to many variations both to handle programming constructs which do not occur in this example and to try to make the proof of correctness more efficient. Full treatments with many examples are available in a recent survey by Elspas, et al.,[3] and in Manna's forthcoming textbook.[4] Some further general comments about the nature of the problem will be made here. Analogous comments could be made about most of the other approaches to proving correctness.

Programs can only be said to be correct with respect to formal specifications or input-output assertions.

There is no formal way to guarantee that these specifications adequately express what someone really wants the program to do.

Given a program with specifications on the input and output, there is probably no automatic way to generate all the additional assertions which must be added to make the proof work. For a human to add these assertions requires a thorough understanding of the program. The programmer should be able to supply these assertions if he is able to formalize his intuitive understanding of the program.

Given a program with assertions in each loop and given an adequate definition of the semantics of the programming language, it is fairly routine to generate the theorems or verification conditions. Several existing computer programs that do this are described below.

The real problem in proving correctness lies in the fact that even for simple programs, the theorems that are generated become quite long. This length makes proving the theorems very difficult for a human or for current automatic theorem provers.

Formalizing the programmer's intuition of correctness

It may not be apparent, but the process of proving correctness is just a formalization into rigorous logical terms of the informal and sometimes sloppy reasoning that a programmer uses in constructing and checking his program. The programmer has some idea of what he expects to be true at each stage of his program (the assertions), he knows how the programming language semantics will transform a stage (generating the assertions in dashed-line boxes of Figure 2), and he convinces himself that the transformations will give the desired result (the proof of the theorem). In this sense proving program correctness is just a way to put into formal language everything one should understand in reading and informally checking a program for correctness. In fact, there is no clear division between the idea of reading code to check it for correctness and the idea of proving it correct by more rigorous means; the difference is one of degree of formality.

One question that should be addressed in this context regards the fact that both the correctness and the halting problems for arbitrary programs are known to be undecidable in the mathematical sense. However, this question of mathematical undecidability should not arise for any program for which there are valid intuitive reasons for the program to be correct.

Confidence in correctness

I hope I have made the point that logical proof of correctness techniques are radically different from testing techniques which are based on executing the program on selected input data in a specific environment. However, I do not want to imply that in a practical situation a proof or anything else can lead to absolute certitude of correctness. In fact a proof by itself does not necessarily lead to a higher level of confidence than might be achieved by extensive testing of a program. From a practical viewpoint there are a number of things that could still be wrong after a proof if one is not careful: what is proven may not be what one thought was proven, the proof may be incorrect, or assumptions about either the execution environment or the problem domain may not be valid. However, a proof does give a quite different and independent view of program correctness, and if it is done well, it should be able to provide a very high level of confidence in correctness. In particular, to the extent that a proof is valid, there should no longer be any doubt about what might happen after allowable but unexpected input values.

MANUAL PROOFS

The basic ideas in the last section have been known for some time. This section describes the practical progress which has been made with manual proofs in the last few years.

The size of programs which can be proven by hand depends on the level of formality that is used. In 1967 McCarthy and Painter[5] manually proved the correctness of a compiler for very elementary arithmetic expressions. It was a formal proof based on formal definitions of the syntax and semantics of the simple languages involved.

Rigorous but informal proofs

A more informal approach to proofs is now popular. This approach is rigorous, but uses a level of formality like that in a typical mathematics text. Arguments are based on an intuitive definition of the semantics of the programming language without a complete axiomatization. Using these techniques a variety of realistic, efficient programs to do sorting, merging, and searching have been proven correct. The proof of a twenty line sort program might require about three pages. It would now be a reasonable exercise for advanced graduate students.

Proofs of significantly more complex programs have also been published. London[6,7] has done proofs of a pair of LISP compilers. The larger compiler is about 160 lines of highly recursive code. It complies almost the

full LISP language—enough so it can compile itself. It is a generally unused compiler. It was written for teaching purposes, but it is not just a toy program. Another complex program has been proven correct by Jones.[9] The program is a PL-1 coding of a slightly simplified version of Earley's recognizer algorithm. It is about 200 lines of code. Probably the largest program that has been proven correct is in the work on computer interval arithmetic by Good and London.[9] There they proved the correctness of over 400 lines of Algol code. The largest individual procedure was in the 150-200 line category. A listing of many other significant programs which have been proven correct can be found in London's recent paper.[10]

If a complex 200 line program can now be proven correct by one man in a couple of months, one can begin to think about breaking larger programs into modules and getting a proof of correctness within a few man years of effort. Clearly there are programs for which a guarantee of the correctness of the running program would be worth not man years but many man decades of effort. We had better take a closer look at the feasibility of such an undertaking and what the proof of correctness would really accomplish.

Environment problems

In most existing proofs of program correctness, what has been proven correct is either the algorithm or a high level language representation of the algorithm. With today's computers what happens when the program actually runs on a physical computer would still be anybody's guess. It would be a significant additional chore to verify that the environment for the running program satisfies all the assumptions that were made about it in the proof. Problems with round off errors, overflow, and so forth can be handled in proofs. Good and London,[9] Hoare,[11] and others have described techniques for proving properties of programs in the context of computer arithmetic, but this can make the proof much more complex. Furthermore, to assure correctness of the running program one would have to be sure that all assumptions about the semantics of the programming language were actually valid in the implementation. The compiler and other system software would have to be certified. Finally, this could all be for naught considering the possibility of hardware failure as it exists in today's machines.

Thus, proving the correctness of a source language program is only one aspect of the whole problem of guaranteeing the correctness of a running program. Nevertheless, eliminating all errors from the source language program would certainly go a long way toward improving the probability that the program will run according to specifications.

Errors in the proof

An informal proof of correctness typically is much longer than the program text itself—often five to ten times as long. Thus the proof itself is subject to error just like any other extremely detailed and complex task done by humans. There is the possibility that an informal proof is just as wrong as the program. However, a proof does not have any loops and the meaning of a statement is fixed and not dependent on the current internal state of the computer. To read and check a proof is a straightforward and potentially automatable operation. The same can hardly be said for programs. Despite its potential fallibility, an informal proof would dramatically improve the probability that a program is correct. There is evidence from London's work[7] that a proof of correctness will find program bugs that have been overlooked in the code.

Less rigorous proofs

A person proving a program correct by manual techniques must first achieve a very thorough understanding of all details of the program. This clearly limits manual proof techniques to programs simple enough to be totally comprehended by the program provers. It also means that clarity and simplicity is very important in the program design if the program is to be proven correct. There is another school of thought which places primary emphasis on techniques for obtaining clarity and structure in the program design. Dijkstra[12,13] has long been the primary advocate of this approach. By appropriately structuring the program and by using what is apparently a much less formal approach to proofs, Dijkstra claims to have proven the correctness of his THE operating system.[14] Mills[15] advocates a similar approach with the program being sufficiently structured so an informal proof can be as short as the program text itself.

It is probably true that more practical results can be obtained with less rigorous approaches to proofs, especially in the near future. It is even debatable whether the more rigorous proofs give more assurance of correctness, but the formality does make it more feasible to automate the proof process. Whether or not one feels that the rigorous hand proofs of correctness will have much practical value, they are providing experience with different proof techniques that should

be very valuable in attempting to automate the proof process.

AUTOMATING PROOFS OF CORRECTNESS

In proving program correctness the logical statement that has to be proven usually is very long; however, the *proof* is seldom mathematically deep and much of it is likely to be quite simple. In the example given previously the theorems to be proven were almost trivial. It would seem that some sort of automatic theorem proving should be able to be applied in proving program correctness. This has been tried. So far the results have not been very exciting from a practical viewpoint.

Computer-generated proofs

Fully automatic theorem provers based on the resolution principle generally can prove correctness for very small programs—not much larger than the exponentiation program above. However, Slagle and Norton[6] report that they have obtained fully automatic proofs of the verification conditions for Hoare's sophisticated little program FIND which finds the nth largest element of an array. In 1969 London completed a program verifier that automatically generated the verification conditions and then used a special theorem prover based on a natural deduction principle to automatically prove them. This system successfully proved programs to do a simple exchange sort, to test whether a number is prime, and similar integer manipulation programs. The data types were limited to integer variables and one dimensional arrays. Others have experimented with other data types and proof procedures. At the time of this writing I believe that there is no automatic theorem prover which has proven correctness for a program significantly larger than those mentioned.

Automatic theorem provers still cannot handle the length and complexity of the theorems that result from larger programs. Another problem lies in the fact that some semantics of the programming language and additional facts about the application area of the program have to be supplied to the theorem prover as axioms. Automatic theorem provers have difficulty in selecting the right axioms when they are given a large number of them. Even in the minor successes that have been achieved, a somewhat tailor-made set of axioms or rules of inference have been used.

Computer-aided proofs

There are now several efforts directed toward providing computer assistance for proving correctness. This takes the form of systems to generate verification conditions and to do *proof* checking, formula simplification and editing, and semiautomatic or interactive theorem proving. Unfortunately at this time almost any automation of the *proof* process forces one into more detailed formalisms and reduces the size of the program that can be proven. This is because the logical size of the *proof* steps that can be taken in a partially automated *proof* system is still quite small. Presumably this is a *temporary* phenomenon. It seems reasonable to expect that we will *soon* see computer-aided verification systems which make use of some automatic theorem proving and can be used to prove correctness of programs somewhat larger than those that have been proven by hand.

Igarashi, London, and Luckham[9] are developing a system for proving programs written in PASCAL. The verification condition generator handles almost all the constructs of that language except for many of the data structures. Their approach is based on the work of Hoare.[3,5]

Elspas, Green, Levitt, and Waldinger[7] are developing a proof-of-correctness system based on the problem-solving language QA4.[8] It will use the goal-oriented, heuristic approach to theorem proving which is characteristic of that language.

Good and Ragland[2] have designed a simple language NUCLEUS with the idea that a verification system and a compiler for the language could be proven correct. Both the verification system and the compiler would be written in NUCLEUS and the proofs of correctness would be based on a formal definition of the language. The intent is that the language would then be able to be used to obtain other certified system software.

These three systems give a general idea of the current work going on. A proof-checking system will be described in the next section. Several other interesting systems have been implemented and basic information about them is readily available in London's recent paper.[10]

Long-range outlook

Proofs of correctness are currently far behind testing techniques in terms of the size and complexity of the programs that can be handled adequately. It is very much an open question whether automated proof techniques will ever be feasible as a commonly used alter-

native to testing a program. Many arguments pro and con are too subjective for adequate consideration here; however, a few comments are in order before one uses the rate of progress in the past as a basis for extrapolating into the future.

Proofs are based on sophisticated symbolic manipulations, and we are still at an early stage of gathering information about ways to automate them. Existing proof systems have been aimed mostly at testing the feasibility of techniques. Few if any have involved more than a couple man years of effort—many have been conceived on a scale appropriate for a Ph.D. dissertation. If and when a cost-effective system for proving correctness becomes feasible, it will certainly require a much larger implementation effort.

Proofs may be practical only in cases where a very high level of confidence is desired in specified aspects of program behavior. With computer-aided proofs one could hope to eliminate most of the sources of error that might remain after a manual proof. As exemplified by the work of Good and Ragland,[23] the verification system itself as well as compilers and other system software should be able to be certified. If the basic hardware/software is implemented with a system such as LOGOS[24] for computer-aided design of computer systems, then there should be a reasonable guarantee that the implemented computer system meets design specifications. With sufficient error-checking and redundancy, it should thus be possible to virtually eliminate the danger of either design or hardware malfunction errors. By the end of this decade these techniques may make it possible to obtain virtual certitude about a program's behavior in a running environment. There are many applications in areas such as real-time control, financial transactions, and computer privacy for which one would like to be able to achieve such a level of confidence.

SOME THEORETICAL FRONTIERS

Proofs of program correctness involve one in a seemingly exorbitant amount of formalism and detail. Some of this is inherent in the nature of the problem and will have to be handled by automation; however, the formalisms themselves often seem awkward. The long formulas and excessive detail may result partially because we have not yet found the best techniques and notation. Active theoretical research is developing many new techniques that could be used in proving correctness. Research in this area, usually called the mathematical theory of computation, has been active since McCarthy's[25,26] early papers on the subject. I feel that practical applications for proofs of correctness will develop slowly unless new techniques for proving correctness can significantly reduce the awkwardness of the formalisms required. This section will describe some of the current ideas being investigated. The topics chosen are those which seemed more directly related to techniques for facilitating proofs of correctness.

Induction techniques for loops and recursion

Proving correctness of programs would be comparatively simple if programs had no loops or recursion. However, some form of iteration or recursion is central to programming, and techniques for dealing with it effectively in proofs have been a subject of intensive study. All the techniques use some form of induction either explicitly or implicitly. The method of inductive assertions described previously handles loops in flowcharts by the addition of enough extra assertions to break every loop and then appeals to induction on the number of commands executed. For theoretical purposes it is often easier and more general to work with recursively defined functions rather than flowcharts. Almost ten years ago McCarthy proposed what he called Recursion Induction[26] for this situation. Manna et al. have extended the inductive assertion method to cover recursive,[27] parallel,[28] and non-deterministic[29] programs. Several other induction principles have been proposed by Burstall,[30] Park,[31] Morris,[32] and Scott.[33] A development and comparison of the various induction principles has been done recently by Manna, Ness, and Vuilleman.[34]

Formalizing the semantics of programming languages

The process of constructing the verification conditions or logical formulation of correctness is dependent on the meaning or semantics of the programming language. One can also take the opposite approach—proving correctness is a formal way of knowing whether a higher level meaning is true of the program. Thus the meaning or semantics of any program in a language is implicitly defined by a formal standard for deciding whether the program satisfies a specification. There is a very close interrelation between techniques for formalizing the semantics of a programming language and proofs of program correctness. Floyd's early work on assigning meanings to programs[1] has been developed especially by Manna[2] and Ashcroft.[35] Burstall[36] gives an alternative way to formulate program semantics in first-order logic. Ashcroft[37] has recently summarized this work and described its relevance.

Hoare,[11,20] Igarashi,[28] de Bakker,[39] and others have worked to develop axiomatic characterizations of the semantics of particular programming languages and constructs. The Vienna Definition Language[40] uses an abstract machine approach to defining semantics, and Allen[41] describes a way of obtaining an axiomatic definition from an abstract machine definition. The axiomatic definition is generally more useful in proofs. Scott and Strachey have developed another approach to defining semantics[42] which is described below.

Work on defining the semantics of programming languages is very active with many different approaches being tried. Those described above are only the ones more closely related to proofs. If any of these ideas can greatly simplify the expression or manipulation of properties of programs, they should have a similar simplifying impact on proofs of correctness.

Formal notation for specifications

Formal correctness only has meaning with respect to an independent, formal specification of what the program is supposed to do. For some programs such specifications can be given fairly easily. For example, consider a routine SORT which takes a vector X of arbitrary length n as an argument and produces a vector Y as its result. With appropriate conventions, the desired ordering on Y is specified by:

$$(\forall i, j)[1 \leq i < j \leq n \Rightarrow Y(i) \leq Y(j)]$$

One also needs a specification about the relation between X and Y. With the property $\mathrm{PERM}(x)$ meaning "x is a permutation" and using $\circ$ for functional composition, the following will do:

$$(\exists P)[\mathrm{PERM}(P) \& Y = X \circ P]$$

Note that the specification allows for any one of many possible algorithms to be chosen—presumably on the basis of efficiency. Yet from an external point of view the specification is complete. If SORT is to be used as part of a larger program, the specifications contain all one may want to know about it.

We can usually define correctness in this way for numeric, mathematical, and other simple programs typically found in program libraries. In fact the causality is largely the other way around: it is worth putting a program in a library to the extent that there is a good way of precisely defining the effects of the program without getting into all the details of its algorithmic implementation.

It would be useful to have a good way of writing formal specifications for a much wider range of computational processes. Parnas has been working on such techniques for formally specifying software modules.[43] His approach does handle error messages, and all side effects have to be carefully formalized.

From a proof of correctness point of view the formalism must have convenient deductive techniques as well as expressive power. First-order predicate calculus has the best deductive techniques, but without extensive definitions and axioms, its expressive power is very poor. For the SORT program above we assumed a definition of permutation, and still the specifications are more obscure than one might desire. For many programs the attempt to define their external effects with the formalism of a fairly standard predicate calculus can lead to extremely long and complex expressions. In particular, proof techniques associated with iteration and recursion have often been awkward when expressed in formal logic. One reason is that recursion and iteration lead to partial functions, that is, functions that may not be defined at all points. There has been a need for the logic that handles undefined values and can be easily used to prove properties of partial functions. Despite many efforts there has been no really successful, agreed-upon logical calculus that dealt with undefined values in a clean and natural way. Some recent work by Scott offers a possible solution to this and other problems.

The work of Scott, Strachey, and Milner

In 1964 Strachey[44] outlined an approach to defining the semantics of a programming language by mapping programs into a mathematical structure built up from a rather small number of precisely specified basic concepts. The approach eliminated any need for an abstract evaluating mechanism. Unfortunately, the idea required some mathematical objects (such as self-referential functions) for which there was no firm mathematical foundation.

In 1969 Scott started to work on the underlying mathematical problems. The main breakthrough led to the first matmheatical model of the λ-calculus.[45,46] The work involved the breaking of new ground in both lattice theory and topology. Function spaces are considered as lattices by using the "is consistent with and less defined than" relation on partial functions for the lattice partial ordering. It is then possible to define a logic with a fairly natural induction scheme which seems to have great generality and ease of expression for proving properties of recursively defined functions.

Scott's techniques allow for the construction of a universe of computable mathematical functions which

is sufficiently general so that it should be possible to define the meaning of any program by associating with it a specific function in this universe.[42,47] The semantics of a program are thus defined mathematically in terms of a limited number of basic mathematical concepts and not in terms of the result of a calculation on a machine. The semantical function that makes the association is defined recursively on the syntax of the program. The mathematical universe is sufficiently general so that the semantical function itself exists within the universe.[43]

The practicality of this approach has yet to be determined, but it seems to hold out the hope of a much less cumbersome way to formalize semantics. This mathematical approach to semantics may enable one to abstract from the arbitrary choices a great amount of extraneous detail that is typical of program implementations. The trick, of course, is to abstract from the right detail without losing important properties of the program.

Milner[48] has implemented a mechanical proof checker for a logic of computable functions based on some of the work of Scott. The implementation includes extensive simplification mechanisms and an interactive goal setting structure. Milner and Weyhrauch have used the logic to formalize semantics,[49,50] to prove simple program correctness,[50] and to give a mechanical proof of compiler correctness based on formally defined semantics.[51] The proof checker is still limited to proving properties of rather small programs; however, expressing formal properties of programs does seem to be simplified. The expression simplification mechanisms have also been useful.

The nature of this and other active theoretical research indicates that there may soon be techniques which will significantly simplify the problem of proving program correctness.

INTEGRATING PROOFS WITH PROGRAM DESIGN

Proving program correctness has usually been done after a program is written. An alternate approach is to integrate the proof with the program design. This approach provides some hope that proofs might eventually help to organize and simplify the program production process. A proof of correctness will greatly increase the amount of formalism that must be dealt with. However, if a proof can be integrated into the design and writing stages, it should eliminate most of the need for debugging and may alleviate the problems of documentation and maintenance. Floyd[52] has envisioned an automated verification system such that a programmer can interact with it in real time as he is writing his program.

Hoare's proof of correctness for his program FIND[53] was done in a top down way with the program and the proof evolving simultaneously. Jones in his proof of Earley's recognizer algorithm[8] exemplified a process he calls the formal development of correct algorithms. It is the longest published example of how a proof might discipline program design.

Throughout the development of the algorithm Jones uses a special formal notation related to the Vienna Definition Language, he does not introduce an ordinary programming language until the very end. With this notation he was able to give a formal, non-procedural specification for a recognizer in about half a page. He then develops the algorithm by stages while at each stage extending a proof that the partially developed algorithm will meet the specifications. At each stage the proof depends on formally expressed assumptions about the undeveloped part of the algorithm.

At the present time the amount of formalism required for the proof tends to overwhelm the program design effort. Nevertheless, this approach appears to make proofs of correctness somewhat more practical in an actual programming environment.

Automatic program synthesis

Rather than writing both specifications and a program, one might want to let the computer create the program and thus be responsible for its correctness. One technique for automatic program synthesis is closely related to techniques for proving correctness. One proves that there is an output satisfying the specifications and then extracts a program from this proof.[54,55] By using induction in the proof, it is possible to construct programs with loops. Manna and Waldinger have given several examples of this.[56]

While automatic program synthesis would be more useful than proving correctness, automatic synthesis requires a much more difficult proof. Since techniques for generating the required proofs are the major unsolved problem in this whole area, this form of automatic program synthesis is a more long-range goal than proofs of correctness.

CONCLUSION

Work on proving properties of programs has progressed to the point where one can argue whether there will soon be useful results. It is mostly a matter of what one means by "useful".

The software engineer who is worried about large programming projects will find current proof techniques hopelessly inadequate for all the large scale problems that are the center of his concern. Even for small modules he will probably find that test methods are more cost-effective than rigorous proofs. One should be able to obtain very great confidence in the correctness of a moderate-sized program if the level of talent and resources that would be necessary for a rigorous proof were devoted to reading and testing the program. Considering the time it normally takes for research results to work their way into practical applications, I would expect that it will be at least three or four years before this situation changes significantly.

Within the next three or four years, less rigorous techniques for structuring, understanding, and checking a program may become widely used. More rigorous proof techniques could be useful on small critical modules where adequate confidence cannot be achieved by other means. In this case it may be worth the additional cost of a proof to obtain an independent evaluation of correctness.

While most work on proving correctness has been for programs written in higher level languages, the most useful early applications may occur either for algorithms at the hardware or microcode level or for the calling structure at the highest level in the design of a large program. In both cases there is a high priority on correctness, and one would like to be assured of correctness long before testing becomes possible.

If current research on simplifying and automating the proof process can significantly reduce the difficulty of proving correctness, then in a few years proofs may be commonly used on small critical modules. Gradually the proof techniques could then be extended to larger programs so that they can be more useful in implementing very reliable systems. It is unlikely that proof techniques will be cost-effective for routine programs within this decade, but the potential is there for eventually revolutionizing the software marketplace.

REFERENCES

1 R W FLOYD
Assigning meanings to programs
Proceedings of a Symposium in Applied Mathematics Vol 19 Mathematical Aspects of Computer Science American Mathematical Society 1967 pp 19-32

2 Z MANNA
The correctness of programs
Journal of Computer and System Sciences Vol 3 No 2 May 1969 pp 119-127

3 B ELSPAS K N LEVITT R J WALDINGER A WAKSMAN
An assessment of techniques for proving program correctness
Computing Surveys Vol 4 No 2 June 1972

4 Z MANNA
Introduction to the mathematical theory of computation
McGraw Hill Book Co Inc to be published

5 J McCARTHY J PAINTER
Correctness of a compiler for arithmetic expressions
Proceedings of a Symposium in Applied Mathematics Vol 19 Mathematical Aspects of Computer Science American Mathematical Society 1967 pp 33-41

6 R L LONDON
Correctness of a compiler for a LISP subset
Proceedings of an ACM Conference on Proving Assertions about Programs
SIGPLAN Notices Vol 7 No 1 and SIGACT News No 14 Jan 1972 pp 121-127

7 R LONDON
Correctness of two compilers for a LISP subset
Artificial Intelligence Memo 151 Stanford Univ Oct 1971

8 C B JONES
Formal development of correct algorithms: An example based on Earley's recognizer
Proceedings of an ACM Conference on Proving Assertions about Programs
SIGPLAN Notices Vol 7 No 1 and SIGACT News No 14 Jan 1972 pp 150-169

9 D I GOOD R L LONDON
Computer interval arithmetic: Definition and proof of correct implementation
Journal of the ACM Vol 17 No 4 Oct 1970 pp 603-612

10 R L LONDON
The current state of proving programs correct
Proceedings of the ACM Annual Conf ACM 1972

11 C A R HOARE
An axiomatic basis of computer programming
Communications of the ACM Vol 12 No 10 Oct 1969 pp 576-583

12 E W DIJKSTRA
Notes on structured programming
Technische Hogeschool Eindhoven August 1969

13 E W DIJKSTRA
A constructive approach to the problem of program correctness
BIT Vol 8 1968 pp 174-186

14 E W DIJKSTRA
The structure of the "THE" multiprogramming system
Communications of the ACM Vol 11 No 5 May 1968 pp 341-346

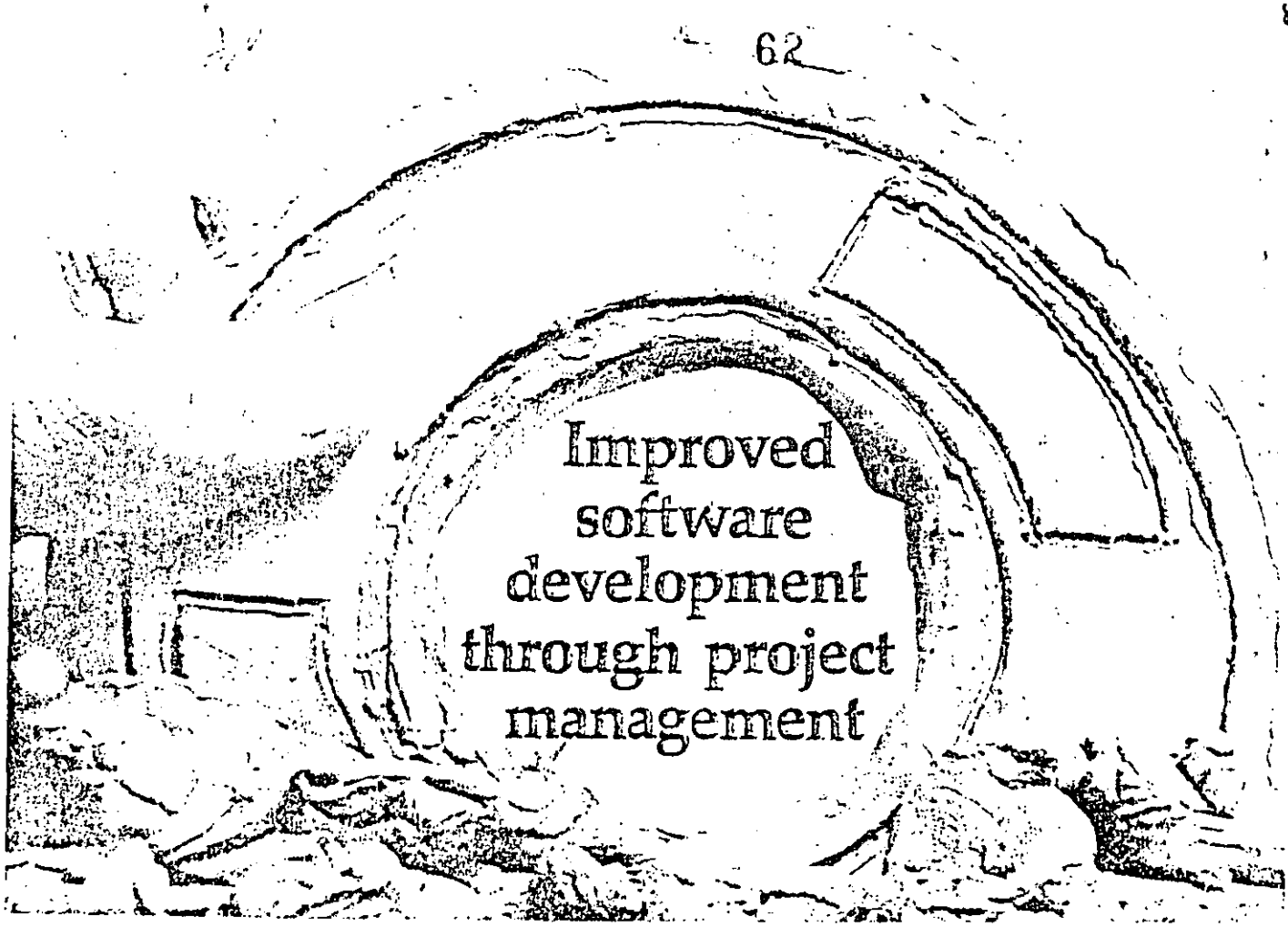
15 H D MILLS
The complexity of programs
Proc of SIGPLAN Symposium on Computer Program Test Methods Prentice-Hall to appear

16 J R SLAGLE L M NORTON
Experiments with an automatic prover having partial ordering rules
Heuristics Laboratory, National Institutes of Health 1971

17 C A R HOARE
Algorithm 65, find
Communications of the ACM Vol 4 No 7 July 1961 p 321

18 J C KING
A program verifier
PhD Thesis Carnegie-Mellon University Sept 1969

19 S IGARASHI R L LONDON D LUCKHAM
Private communication



by Robert B. Fireworker and Leonard J. Bogner, Jr.

There is a need for improvement in state-of-the-art project management and programming techniques which aim to improve software development.

Customarily, programming projects begin with the user's problem statement and needs analysis. This, in turn, is delivered to the software development group. Promises are made: Timely delivery (commitments, dates), accuracy, stated cost, system reliability and enhanced system functions. At this point, user/DP communication become secondary.

The typical product delivered:

- Functions—are they worth the cost and are they actually useful?
- Late installation—project slippage, poor estimates and/or change control.
- Resources—greater cost than anticipated.
- Questionable (interpretative) reliability.

A system containing these "features" must be further customized to meet the user's actual needs. Yes, an effective system may have been developed, with little help from original estimates; but how efficient and productive was the development process?

These items are symptomatic of software development problems.

The problems are poor planning (estimating, scheduling, control) and poor communication.

"We are still in the unfortunate condition that software development is not a science, it is a craft; and our knowledge is of the meager and unsatisfactory kind."—M. I. Bernstein "Hardware is Easy: It's Software That's Hard," DATAMATION.

Not to dissent the creative, artistic atmosphere of systems design, standardization must exist in terms of planning, documentation and review—in order to relieve ambiguity. Emphasis must not only be placed on what the user thinks he or she needs, but a determination should be made of present needs as well as needs for the future.

The elements of successful project management—planning, estimating and scheduling, tracking and control of change—will be the crux here. Modern programming technologies, as effective and efficient development tools, will be handled after that.

Software development cycle

The standard which evolved as the structure for development projects is the project life cycle (or software development cycle). Based on the premise that software development has birth (inception), maturity (development) and death (completion), the life cycle serves as a framework for communication and cooperation. It is utilized to monitor activities, highlight critical tasks and to record progress.

"The programming development cycle is simply a series of orderly, interrelated activities leading to the successful completion of a set of programs."—Phillip W. Metzger, *Managing a Programming Project*, Prentice-Hall.

Standardization is evident through documentation of project phases, milestones and estimates. Milestones

are defined "end-products" which result from each phase. Estimates are judgments based on past experience and/or research. Ideally, estimates should be refined phase by phase. They should not be specific pinpoints of time and/or cost, but present an acceptable range.

By using the system life cycle as a tracking device, where estimates are compared to reality, project reviews should be conducted periodically. The number of reviews depends on the size and complexity of the project, experience of staff and events which occur. It is a good idea to plan a major milestone review after the completion of the conceptual design. This review should include all parties involved with emphasis given to user feedback concerning design direction.

Where are we now, and where are we going? As the project grows it becomes people intensive (See Chart A). If the design is off-course at the outset, it will result in timeframe slippage and increased costs further down the road.

Where the life cycle approach moves away from convention is in its flexible nature to accomplish its objectives.

Differences in user application are seen in phase titles, definitions, number and subdivisions. This flexibility allows the option of a broadly defined life cycle or a narrow, thorough breakdown. Chart B depicts various user interpretations of the life cycle. Although approaches to this technique differ, the objectives are the same—to document the project plan by task and timeframe.

The life cycle can be broken into five fundamental stages (See Chart A):

Planning
Design
Development
Testing
Implementation

Planning

Expectations are set during the initial stage of software development.

Productivity will be judged on how the appropriate expectations are met. Extensive time, up front, given to a well-thought-out plan is invaluable to the success of a project. Metzger recommends 33 percent time allocation for planning process, while Dr. Raymond Winters (IBM) recommends 45 percent given up front. Coordination and utilization of all parties concerned during this process will stimulate commitment and lessen ambiguity.

Long term planning is the responsibility of upper management (both user and development) and is characterized by a few people involved and broad topic discussion. The planning process is continuous throughout the life cycle. As development progresses, more people provide planning input as the specifics of the system are discussed. Project management must be aware of and monitor short term plans and their relationship to long term goals.

User interaction will determine the scope of the project. Through the use of problem analysis, requirements study and feasibility techniques, project management will form a baseline. The manager must investigate the project environment: Past, present and future.

Past—systems and applications.

Present—resources (people and money), politics.

Future—corporate strategy, expansion plans.

Following this interaction, objectives are set, responsibilities assigned, schedules (resources and time) are developed.

Change control must be addressed —as user's needs change he or she must be aware of its effect on the scope of the project. If change is imperative, the plan may have to be revised and both the user and development management must sign off on the revision. Additional or shifting responsibilities will be decided at this time.

Measurement techniques also will be designated. The most useful technique will serve multiple functions: Planning, scheduling, controlling, communicating and simulating.

"What has emerged over the years is a technique that employs the common planning and scheduling procedures of PERT and CPM."—C. W. Burrill and L. W. Ellsworth: *Modern Project Management* (unedited manuscript, used with permission of the publisher, Burrill-Ellsworth Assoc., Inc.).

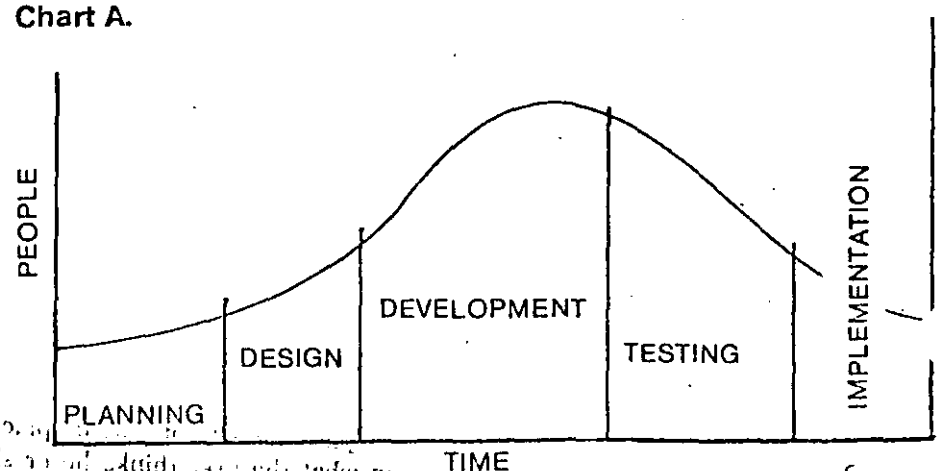
Networks (PERT/CPM) list the activities and their sequence within the life cycle. They provide a simple procedure for establishing a time schedule and monitor the critical activities involved. Through constant comparison of estimates to actual, project control is exercised. A documented network provides a communicative project plan. Networks also may be used for simulation as assessments of change and its impact may be observed affecting time and cost factors.

Top management's goals are overseeing long term planning.

Project management must also acknowledge:

- Temptation to design isolated from planning.
- Initial emphasis on priorities—not dates.
- Do not assume a "perfect world." Provide for contingencies such as illness, vacation and turnover.

Chart A.



Design

Once objectives are set and resources allocated, the next stage is to specify how the program system is to work.

The "conceptual design document" is the major outcome of the design phase. It serves as a blueprint for the design specifications (solutions to the problem) and serves as a starting point for the programmer.

The conceptual design document provides narrative portraying the overall concept. High-level explanations are given to these areas:

- Programming standards—covering flowcharting, data naming, interfacing. Prohibitions also should be stated.
- Program design—the actual structure of the system, as an overview of the hierarchy and not a breakdown of unit specifics.
- File design—accompanies the program design, defining system files to be utilized and accessed by program modules.
- Data flow—a narrative description of how the system will perform its tasks. This will be presented to non-technical management.

If possible, the conceptual design should define optional approaches and tradeoffs associated with each approach. Management will observe the long term and short term benefits of each and determine a course of action.

A major milestone review is to be conducted with the objective of confirming the design direction in terms of the user's expectations. This review will determine if modifications are to be made and to approve the continuation of the project.

Development

This is the most people intensive phase of the life cycle. It is the heart of the project because development is where the user's expectations will be met through products developed.

Development is people intensive: The final product must meet the user's expectations.

Because of the technical complexity of major programming projects, segregating the development functions into four groups is advised.

Four typical groups and their functions are as follows:

Programming group

Since the programmers are the focal point of the project, this section centers on their function.

From the conceptual design document, the programmers will develop a detailed design. The most current approach utilized is top-down design (discussed later). The programmer's job is to design module, in detail, concurring to the conventions of the conceptual design.

Next, program code must be developed. The detailed design may have to be changed but these changes should remain within the domain of the programmer as long as they do not affect the conceptual design.

Each module must be thoroughly tested prior to integration within the design hierarchy. Test drivers (programs written to simulate environmental conditions) are a useful testing aid. "It is management's responsibility to provide a good test environment—predictable computer time and smooth interfacing with the computer facility," says Metzger.

As modules are constructed and tested they must be documented. Programmers are responsible for this documentation. Changes to the original detailed design should be inserted within the document.

Once modules are tested and documented, they must be integrated within the design hierarchy. Observation of the new modules' effects on others and the system as a whole is the responsibility of programming management.

Analysis and design group

The analysts and designers remain active during the programming phase and serve the following functions:

- Change control—investigate, recommend, document.
- Data control—integrity of the system files.
- Review detailed design—compare to user expectations.
- User documentation—installation, operation, and maintenance.

Test group

This group prepares for and performs tests (see following information on testing) which are not concerned with programming but with overall systems tests. The separation of these functions allows programmers to concentrate only on code.

Staff support

Areas taken for granted are provided by this unit. Concerns include controlling computer time, supplying keypunch services, coordinating terminals and handling special fire-fighting assignments.

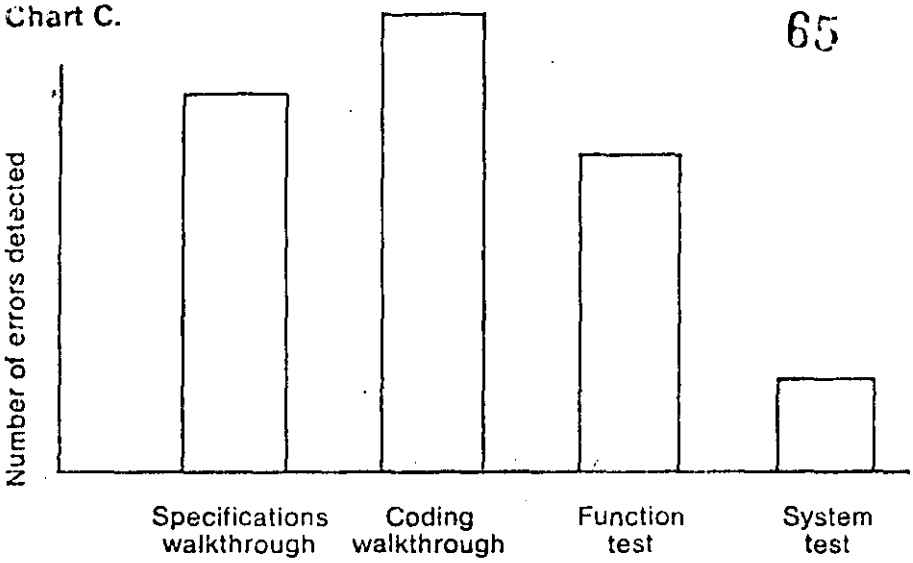
Testing

The main objective of the testing phase is to condition the programmers' products to an all-inclusive set of tests neither designed nor executed by the programmers and

Chart B.

Metzger	Montgomery Ward	Manufacturers Hanover
Definition	Planning	Feasibility
Design	Initial Investigation	Functional Analysis
Programming	Preliminary Study	Design
System Test	Systems Planning Study	Implementation
Acceptance Test	Development	
Installation & Operation	Systems Requirements	
	Systems Specifications	
	Technical Requirements	
	Implementation Planning	
	Programming	
	User Training	
	Systems Test	
	Implementation	
	Conversion	
	Post-Implementation Review	

Chart C.



run in as nearly a live atmosphere as possible. The secondary objective is to provide initial training to the user.

Responsibility for meeting these objectives belongs to the test group.

Upon receipt of the developed product and working with the aid of staff support, the tester must be prepared with specific and documented test procedures, scheduled computer time, library facilities (simulated and live data) and people ready to tear the system apart. The developed system should experience a thorough beating, for if it cannot handle almost any situation imaginable the system may not be accepted.

Observers are welcome, from both the user and programming side, to offer feedback and recommendations. It must be understood that this is a "test" and not a "demonstration" of the finalized system.

If it is determined that a program change must be made, a regression test should be performed. This is to discover the effect of the change on portions tested previously.

Documentation must contain listings of final test runs, programs and corresponding (narrative) documentation.

User training must involve those who maintain and operate the system. Users are to be provided with operating manuals and formal operational training. If a firm does not provide a regular education program, the test group should prepare formal classroom and/or on-the-job training.

Accompanying training methods, users who also will maintain the system must be provided with the detailed design document and the detailed code. Trouble-shooting manuals, listing peculiarities of the system, are also advised.

Implementation

Successful implementation is the result of careful planning during the previous phases. Very few objects reach this stage without being implemented. This is the time that user expectations are to be met, change is formally created within the organization and information processing begins. The user has the ultimate responsibility to verify the system's functional acceptability.

Criteria for system acceptance include:

- Integrity of the system's business functions, user manuals and procedures.
- Demonstration of the user operations group's ability to run the system in a live environment.
- A full scale live environment test using all resources available, executing all procedures and options —with a decision on the worthiness of the programs.

State-of-the-art programming techniques

Traditionally programming has been characterized by its flair for running into the same problems. To name a few: Inability to easily integrate modules, redundancies in functions, superfluous code and inability to detect errors up front. Modern programming techniques relieve the severity of these occurrences.

In this section, some of these techniques are discussed along with observations from three program development projects which utilized them. These projects are: Hartford Insurance Group; Commercial Auto Ratemaker System; Manufacturers Hanover Trust Co. (MHT); Wholesale Demand Deposit Accounting System; and Standard Oil of California, Chevron Program Development System.

Structured environment: Design and programming

Structured design is a collection of practices and procedures, chosen to complement one another, along with rules for applying them. "At MHT, a design methodology also includes management techniques, documentation procedures, tools to aid the designer, standards for specification that serve as the input to the design process (i.e., functional specifications), and the implementation process (i.e., design specifications)."—A. Block and K. Hamilton, "Programmer Productivity in a Structured Environment," INFOSYSTEMS.

Chief programmer is in charge of the program design and for reviewing code, integration and testing.

Advantages of this methodology, as seen through MHT, include:

- Consistency—variance is detected with higher certainty and at an earlier time.
- Modularity—one module for one function, ease of change control and maintenance.
- Documentation—more time and thought given to design than the traditional approach, resulting in greater design integrity.

"Structured programming is based on a mathematically proven structured theorem which states that any program can be written using only the three control logic structures." —"Installation Management/Improved Programming Technologies," IBM.

The chief programmer is responsible for the overall program design; writing mainline routines, critical code, OS interfaces; data definitions; defining modules and assigning to subordinates; and for specifying interfaces between modules. He or she also is responsible for reviewing code, overseeing all integration and testing, and reporting project status to management.

Advantages (Standard Oil) are:

- Technical expertise at management level, reviewing and optimizing code.
- Expert handling critical code and delegating simpler routines to less experience (level of competence).

HIPO diagrams

...erarchy plus Input-Process-Output (HIPO) diagrams are a documentation technique utilized during the design stage prior to the actual coding. "HIPO reduces the amount and ambiguity of the prose required to document function and provides a systematic means of identifying all the functions to be performed and the modules to perform them."—"Installation Manag nt," IBM.

Major milestone review sizes up progress on the project: Is this what the user wants?

HIPO renders a top-down description of program functions. First is an overview, followed by the details of the particular function and its necessary inputs and resultant outputs.

Typically, a HIPO package contains:

- A hierarchy chart that identifies all overview and detail diagrams within the top-down structure.
- Overview and detailed graphic descriptions: Input—files, records, fields, control blocks; process—interactions; output—files, records, control blocks.

Structured walkthroughs

This is an objective check on a program's overall logic and completeness by someone who has not been immersed in its details. "The structured walkthrough is designed to detect and remove errors as early as possible (See Chart C) during the cycle in a problem solving and nonfault finding atmosphere where everyone involved—especially the developer—is eager to find any errors in the work product being reviewed."—"Installation Management," IBM.

These structures are:

- Sequence of two or more operations (MOVE, ADD, . . .).
- Conditional branching to one of two operations and return (IF THEN ELSE).
- Repetition of an operation while a condition is true (DO WHILE).

This method is characterized by the absence of GO TOs and have only one entry and one exit. Since arbitrary branching is not utilized, modules of code are easy to read. Flexibility is allowed through extensions to logic as long as the code retains its one-entry/one-exit property.

Advantages of structured programming, noted by the Hartford Insurance Group, are:

- Compact, accurate, easily followed code.
- Consistently fewer lines of code.
- Use of comments became almost unnecessary.
- Programmers found it easier to locate bugs in their code during tests (See Chart C).

Top-down design and program development

Top-down strategy is a hierarchical approach to program design and development. Similar to the overall systems plan, the initial design is to first design and code high-level modules followed by determining the low-level units which will be needed. "Testing of high-level modules is through the use of test subroutines and procedure calls."—"Improved Productivity in Implementing Information Systems," INFO 79 speech by Harold Feinlieb of National CSS, Inc., and Kevin Tweedy, Standard Oil of California.

Advantages to this method, as experienced by Standard Oil, are:

- Eliminates unnecessary code.
- Ease of integration.
- Effective management control through visible high level products.

Chief programmer teams

Here a small group of programmers (3-5) are under the direction of a senior level or chief programmer. The team represents an opportunity to improve both the manageability and productivity of the group through organizational techniques. These techniques include: Specialized programming jobs, expert technical leadership, stress on training and career development, defined relationships among specialists and effective work effort with a developing, and always visible, project.

The structured walkthrough is a review session during which the developer invites peers to observe his or her logic, step-by-step, and to offer constructive criticism. This technique has evolved not only as an error detection device but also as a communications tool, through discussion of personal approaches to program logic.

Advantages as viewed by the Hartford Group are illustrated in Chart C. ■

About the authors

Fireworker is professor of quantitative analysis in the graduate school of business, St. John's University, Jamaica, NY. He also directs the university's computer information systems academic program. With an ongoing consulting practice, Fireworker has done work for 3M Co., Lever Brothers Co., IBM, Consolidated Edison, Purolator and other large firms. He holds a Ph.D. in operations research and an MS in computer science from New York University. Bogner is a systems analyst with Manufacturers Hanover Trust Co. and is in charge of information and administrative services. He earned his MBA at St. John's University.—Ed.

References

Books

Burrill, C. W., and Ellsworth, I. W.: Modern Project Management, *(unedited manuscript, used with permission of the publisher, Burrill-Ellsworth Assoc., Inc.) obtained at IBM Systems Science Institute.*

Metzger, P. W.: Managing a Programming Project, *Prentice-Hall, Englewood Cliffs, NJ, 1973.*

Technical Publications

IBM, Installation Management Series: *"Managing Systems Development—Montgomery Ward," November, 1973; "Improved Pro-*

Continued on page 35

67 86

Considerations and techniques are proposed that reduce the complexity of programs by dividing them into functional modules. This can make it possible to create complex systems from simple, independent, reusable modules. Debugging and modifying programs, reconfiguring I/O devices, and managing large programming projects can all be greatly simplified. And, as the module library grows, increasingly sophisticated programs can be implemented using less and less new code.

Structured design

by W. P. Stevens, G. J. Myers, and L. L. Constantine

Structured design is a set of proposed general program design considerations and techniques for making coding, debugging, and modification easier, faster, and less expensive by reducing complexity.[1] The major ideas are the result of nearly ten years of research by Mr. Constantine.[2] His results are presented here, but the authors do not intend to present the theory and derivation of the results in this paper. These ideas have been called *composite design* by Mr. Myers.[3-5] The authors believe these program *design* techniques are compatible with, and enhance, the *documentation* techniques of HIPO[6] and the *coding* techniques of structured programming.[7]

These cost-saving techniques always need to be balanced with other constraints on the system. But the ability to produce simple, changeable programs will become increasingly important as the cost of the programmer's time continues to rise.

General considerations of structured design

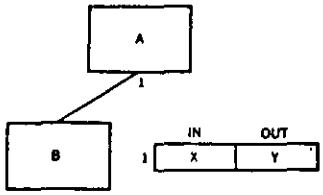
Simplicity is the primary measurement recommended for evaluating alternative designs relative to reduced debugging and modification time. Simplicity can be enhanced by dividing the system into separate pieces in such a way that pieces can be considered, implemented, fixed, and changed with minimal consideration or effect on the other pieces of the system. Observability (the ability to easily perceive how and why actions occur) is another use-

ful consideration that can help in designing programs that can be changed easily. Consideration of the effect of reasonable changes is also valuable for evaluating alternative designs.

Mr. Constantine has observed that programs that were the easiest to implement and change were those composed of simple, independent modules. The reason for this is that problem solving is faster and easier when the problem can be subdivided into pieces which can be considered separately. Problem solving is hardest when all aspects of the problem must be considered simultaneously.

The term *module* is used to refer to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of variable names. Examples of modules are PL/I procedures, FORTRAN mainlines and subprograms, and, in general, subroutines of all types. Considerations are always with relation to the program statements *as coded*, since it is the programmer's ability to understand and change the *source* program that is under consideration.

Figure 1 A structure chart



While conceptually it is useful to discuss dividing whole programs into smaller pieces, the techniques presented here are for designing simple, independent modules originally. It turns out to be difficult to divide an existing program into separate pieces without increasing the complexity because of the amount of overlapped code and other interrelationships that usually exist.

Graphical notation is a useful tool for structured design. Figure 1 illustrates a notation called a *structure chart*,[8] in which:

1. There are two modules, A and B.
2. Module A *invokes* module B. B is *subordinate* to A.
3. B receives an input parameter X (its name in module A) and returns a parameter Y (its name in module A). (It is useful to distinguish which calling parameters represent data passed *to* the called program and which are for data to be *returned* to the caller.)

Coupling and communication

To evaluate alternatives for dividing programs into modules, it becomes useful to examine and evaluate types of "connections" between modules. A connection is a reference to some label or address defined (or also defined) elsewhere.

The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other

Table 1 Contributing factors

	Interface complexity	*Type of connection*	*Type of communication*
low	simple, obvious	to module by name	data
COUPLING			control
high	complicated, obscure	to internal elements	hybrid

modules. Minimizing connections between modules also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous "ripple" effects, where changes in one part cause errors in another, necessitating additional changes elsewhere, giving rise to new errors, etc. The widely used technique of using common data areas (or global variables or modules without their own distinct set of variable names) can result in an enormous number of connections between the modules of a program. The complexity of a system is affected not only by the number of connections but by the degree to which each connection couples (associates) two modules, making them interdependent rather than independent. Coupling is the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules.

The degree of coupling established by a particular connection is a function of several factors, and thus it is difficult to establish a simple index of coupling. Coupling depends (1) on how complicated the connection is, (2) on whether the connection refers to the module itself or something inside it, and (3) on what is being sent or received.

Coupling increases with increasing complexity or obscurity of the interface. Coupling is lower when the connection is to the normal module interface than when the connection is to an internal component. Coupling is lower with data connections than with control connections, which are in turn lower than hybrid connections (modification of one module's code by another module). The contribution of all these factors is summarized in Table 1.

Interface complexity

When two or more modules interface with the same area of storage, data region, or device, they share a common environment. Examples of common environments are:

- A set of data elements with the EXTERNAL attribute that is

copied into PL/I modules via an INCLUDE statement or that is found listed in each of a number of modules.
- Data elements defined in COMMON statements in FORTRAN modules.
- A centrally located "control block" or set of control blocks.
- A common overlay region of memory.
- Global variable names defined over an entire program or section.

The most important structural characteristic of a common environment is that it couples every module sharing it to every other such module without regard to their functional relationship or its absence. For example, only the two modules XVECTOR and VELOC might actually make use of data element X in an "included" common environment of PL/I, yet changing the length of X impacts *every* module making any use of the common environment, and thus necessitates recompilation.

Every element in the common environment, whether used by particular modules or not, constitutes a separate path along which errors and changes can propagate. Each element in the common environment adds to the complexity of the total system to be comprehended by an amount representing all possible pairs of modules sharing that environment. Changes to, and new uses of, the common area potentially impact all modules in unpredictable ways. Data references may become unplanned, uncontrolled, and even unknown.

A module interfacing with a common environment for some of its input or output data is, on the average, more difficult to use in varying contexts or from a variety of places or in different programs than is a module with communication restricted to parameters in calling sequences. It is somewhat clumsier to establish a new and unique data context on each call of a module when data passage is via a common environment. Without analysis of the entire set of sharing modules or careful saving and restoration of values, a new use is likely to interfere with other uses of the common environment and propagate errors into other modules. As to future growth of a given system, once the commitment is made to communication via a common environment, any new module will have to be plugged into the common environment, compounding the total complexity even more. On this point, Belady and Lehman,[9] observe that "a well-structured system, one in which communication is via passed parameters through defined interfaces, is likely to be more growable and require less effort to maintain than one making extensive use of global or shared variables."

The impact of common environments on system complexity may be quantified. Among M objects there are $M\ (M-1)$ or-

dered pairs of objects. (Ordered pairs are of interest because A and B sharing a common environment complicates both, A being coupled to B and B being coupled to A.) Thus a common environment of N elements shared by M modules results in $NM(M-1)$ first order (one level) relationships or paths along which changes and errors can propagate. This means 150 such paths in a FORTRAN program of only three modules sharing the COMMON area with just 25 variables in it.

It is possible to minimize these disadvantages of common environments by limiting access to the smallest possible subset of modules. If the total set of potentially shared elements is subdivided into groups, all of which are *required* by some subset of modules, then both the size of each common environment and the scope of modules among which it is shared is reduced. Using "named" rather than "blank" COMMON in FORTRAN is one means of accomplishing this end.

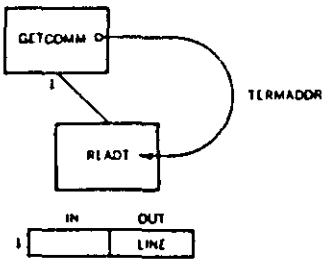
The complexity of an interface is a matter of how much information is needed to state or to understand the connection. Thus, obvious relationships result in lower coupling than obscure or inferred ones. The more syntactic units (such as parameters) in the statement of a connection, the higher the coupling. Thus, extraneous elements irrelevant to the programmer's and the modules' immediate task increase coupling unnecessarily.

type of connection

Connections that address or refer to a module as a whole by its name (leaving its contents unknown and irrelevant) yield lower coupling than connections referring to the internal elements of another module. In the latter case, as for example the use of a variable by direct reference from within some other module, the entire content of that module may have to be taken into account to correct an error or make a change so that it does not make an impact in some unexpected way. Modules that can be used easily without knowing anything about their insides make for simpler systems.

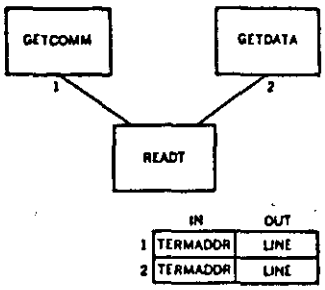
Consider the case depicted in Figure 2. GETCOMM is a module whose function is getting the next command from a terminal. In performing this function, GETCOMM calls the module READT, whose function is to read a line from the terminal. READT requires the address of the terminal. It gets this via an externally declared data element in GETCOMM, called TERMADDR. READT passes the line back to GETCOMM as an argument called LINE. Note the arrow extending from *inside* GETCOMM to *inside* READT. An arrow of this type is the notation for references to internal data elements of another module.

Figure 2 Module connections



Now, suppose we wish to add a module called GETDATA, whose function is to get the next data line (i.e., not a command) from a

Figure 3 Improved module connections

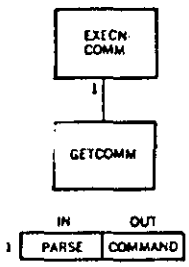


(possibly) different terminal. It would be desirable to use module READT as a subroutine of GETDATA. But if GETDATA modifies TERMADDR in GETCOMM before calling READT, it will cause GETCOMM to fail since it will "get" from the wrong terminal. Even if GETDATA restores TERMADDR after use, the error can still occur if GETDATA and GETCOMM can ever be invoked "simultaneously" in a multiprogramming environment. READT would have been more usable if TERMADDR had been made an input argument to READT instead of an externally declared data item as shown in Figure 3. This simple example shows how references to internal elements of other modules can have an adverse effect on program modification, both in terms of cost and potential bugs.

type of communication

Modules must at least pass data or they cannot functionally be a part of a single system. Thus connections that pass data are a necessary minimum. (Not so the communication of control. In principle, the presence or absence of requisite input data is sufficient to define the circumstances under which a module should be activated, that is, receive control. Thus the explicit passing of control by one module to another constitutes an additional, theoretically inessential form of coupling. In practice, systems that are *purely* data-coupled require special language and operating system support but have numerous attractions, not the least of which is they can be fundamentally simpler than any equivalent system with control coupling.[10])

Figure 4 Control-coupled modules



Beyond the practical, innocuous, minimum control coupling of normal subroutine calls is the practice of passing an "element of control" such as a switch, flag, or signal from one module to another. Such a connection affects the execution of another module and not merely the data it performs its task upon by involving one module in the internal processing of some other module. Control arguments are an additional complication to the essential data arguments required for performance of some task, and an alternative structure that eliminates the complication always exists.

Consider the modules in Figure 4 that are control-coupled by the switch PARSE through which EXECNCOMM instructs GETCOMM whether to return a parsed or unparsed command. Separating the two distinct functions of GETCOMM results in a structure that is simpler as shown in Figure 5.

The new EXECNCOMM is no more complicated; where once it set a switch and called, now it has two alternate calls. The sum of GETPCOMM and GETUCOMM is (functionally) less complicated than GETCOMM was (by the amount of the switch testing). And the two small modules are likely to be easier to comprehend than the one large one. Admittedly, the immediate gains here

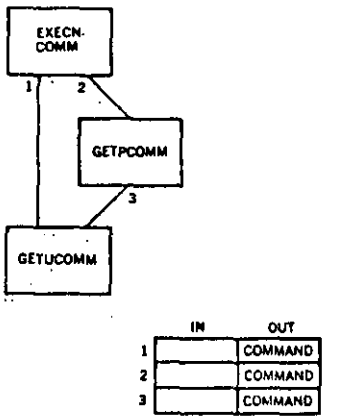
may appear marginal, but they rise with time and the number of alternatives in the switch and the number of levels over which it is passed. Control coupling, where a called module "tells" its caller what to do, is a more severe form of coupling.

Modification of one module's code by another module may be thought of as a hybrid of data and control elements since the code is dealt with as data by the modifying module, while it acts as control to the modified module. The target module is very dependent in its behavior on the modifying module, and the latter is intimately involved in the other's internal functioning.

Cohesiveness

Coupling is reduced when the relationships among elements *not* in the same module are minimized. There are two ways of achieving this—minimizing the relationships among modules and maximizing relationships among elements in the same module. In practice, both ways are used.

Figure 5 Simplified coupling



The second method is the subject of this section. "Element" in this sense means any form of a "piece" of the module, such as a statement, a segment, or a "subfunction". Binding is the measure of the cohesiveness of a module. The objective here is to reduce coupling by striving for high binding. The scale of cohesiveness, from lowest to highest, follows:

1. Coincidental.
2. Logical.
3. Temporal.
4. Communicational.
5. Sequential.
6. Functional.

The scale is not linear. Functional binding is much stronger than all the rest, and the first two are much weaker than all the rest. Also, higher-level binding classifications often include all the characteristics of one or more classifications below it *plus* additional relationships. The binding between two elements is the highest classification that applies. We will define each type of binding, give an example, and try to indicate why it is found at its particular position on the scale.

coincidental binding

When there is no meaningful relationship among the elements in a module, we have coincidental binding. Coincidental binding might result from either of the following situations: (1) An existing program is "modularized" by splitting it apart into modules. (2) Modules are created to consolidate "duplicate coding" in other modules.

As an example of the difficulty that can result from coincidental binding, suppose the following sequence of instructions appeared several times in a module or in several modules and was put into a separate module called X:

```
A = B + C
GET CARD
PUT OUTPUT
IF B = 4, THEN E = 0
```

Module X would probably be coincidentally bound since these four instructions have no apparent relationships among one another. Suppose in the future we have a need in one of the modules originally containing these instructions to say GET TAPERECORD instead of GET CARD. We now have a problem. If we modify the instruction in module X, it is unusable to all of the other callers of X. It may even be difficult to *find* all of the other callers of X in order to make any other compatible change.

It is only fair to admit that, independent of a module's cohesiveness, there are instances when any module can be modified in such a fashion to make it unusable to all its callers. However, the *probability* of this happening is very high if the module is coincidentally bound.

logical binding

Logical binding, next on the scale, implies some logical relationship between the elements of a module. Examples are a module that performs all input and output operations for the program or a module that edits all data.

The logically bound, EDIT ALL DATA module is often implemented as follows. Assume the data elements to be edited are master file records, updates, deletions, and additions. Parameters passed to the module would include the data and a special parameter indicating the type of data. The first instruction in the module is probably a four-way branch, going to four sections of code—edit master record, edit update record, edit addition record, and edit deletion record.

Often, these four functions are also intertwined in some way in the module. If the deletion record changes and requires a change to the edit deletion record function, we will have a problem if this function is intertwined with the other three. If the edits are truly independent, then the system could be simplified by putting each edit in a separate module and eliminating the need to decide which edit to do for each execution. In short, logical binding usually results in tricky or shared code, which is difficult to modify, and in the passing of unnecessary parameters.

temporal binding

Temporal binding is the same as logical binding, except the elements are also related in time. That is, the temporally bound elements are executed in the same time period.

The best examples of modules in this class are the traditional "initialization", "termination", "housekeeping", and "clean-up" modules. Elements in an initialization module are logically bound because initialization represents a logical class of functions. In addition, these elements are related in time (i.e., at initialization time).

Modules with temporal binding tend to exhibit the disadvantages of logically bound modules. However, temporally bound modules are higher on the scale since they tend to be simpler for the reason that *all* of the elements are executable at one time (i.e., no parameters and logic to determine which element to execute).

communicational binding

A module with communicational binding has elements that are related by a reference to the same set of input and/or output data. For example, "print and punch the output file" is communicationally bound. Communicational binding is higher on the scale than temporal binding since the elements in a module with communicational binding have the stronger "bond" of referring to the same data.

sequential binding

When the output data from an element is the input for the next element, the module is sequentially bound. Sequential binding can result from flowcharting the problem to be solved and then defining modules to represent one or more blocks in the flowchart. For example, "read next transaction and update master file" is sequentially bound.

Sequential binding, although high on the scale because of a close relationship to the problem structure, is still far from the maximum—functional binding. The reason is that the procedural processes in a program are usually distinct from the *functions* in a program. Hence, a sequentially bound module can contain several functions or just part of a function. This usually results in higher coupling and modules that are less likely to be usable from other parts of the system.

functional binding

Functional binding is the strongest type of binding. In a functionally bound module, all of the elements are related to the performance of a single function.

A question that often arises at this point is what is a function? In mathematics, $Y = F(X)$ is read "Y is a function F of X." The function F defines a transformation or mapping of the independent (or input) variable X into the dependent (or return) variable Y. Hence, a function describes a transformation from some

input data to some return data. In terms of programming, we broaden this definition to allow functions with no input data and functions with no return data.

In practice, the above definition does not clearly describe a functionally bound module. One hint is that if the elements of the module all contribute to accomplishing a single goal, then it is probably functionally bound. Examples of functionally bound modules are "Compute Square Root" (input and return parameters) "Obtain Random Number" (no input parameter), and "Write Record to Output File" (no return parameter).

A useful technique in determining whether a module is functionally bound is writing a sentence describing the function (purpose) of the module, and then examining the sentence. The following tests can be made:

1. If the sentence *has* to be a compound sentence, contain a comma, or contain more than one verb, the module is probably performing more than one function; therefore, it probably has sequential or communicational binding.

2. If the sentence contains words relating to time, such as "first", "next", "then", "after", "when", "start", etc., then the module probably has sequential or temporal binding.

3. If the predicate of the sentence doesn't contain a single specific object following the verb, the module is probably logically bound. For example, Edit All Data has logical binding; Edit Source Statement may have functional binding.

4. Words such as "initialize", "clean-up", etc. imply temporal binding.

Functionally bound modules *can* always be described by way of their elements using a compound sentence. But if the above language is unavoidable while still completely describing the module's function, then the module is probably not functionally bound.

One unresolved problem is deciding how far to divide functionally bound subfunctions. The division has probably gone far enough if each module contains no subset of elements that could be useful alone, and if each module is small enough that its entire implementation can be grasped all at once, i.e., seldom longer than one or two pages of source code.

Observe that a module can include more than one type of binding. The binding between two elements is the highest that can be

applied. The binding of a module is lowered by every element pair that does not exhibit functional binding.

78

Predictable modules

A predictable, or well-behaved, module is one that, when given the identical inputs, operates identically each time it is called. Also, a well-behaved module operates independently of its environment.

To show that dependable (free from errors) modules can still be unpredictable, consider an oscillator module that returns zero and one alternately and dependably when it is called. It might be used to facilitate double buffering. Should it have multiple users, each would be required to call it an even number of times before relinquishing control. Should any of the users have an error that prevented an even number of calls, all other users will fail. The operation of the module given the same inputs is not constant, resulting in the module not being predictable even though error-free. Modules that keep track of their own state are usually not predictable, even when error-free.

This characteristic of predictability that can be designed into modules is what we might loosely call "black-boxness." That is, the user can understand what the module does and use it without knowing what is inside it. Module "black-boxness" can even be enhanced by merely adding comments that make the module's function and use clear. Also, a descriptive name and a well-defined and visible interface enhances a module's usability and thus makes it more of a black box.

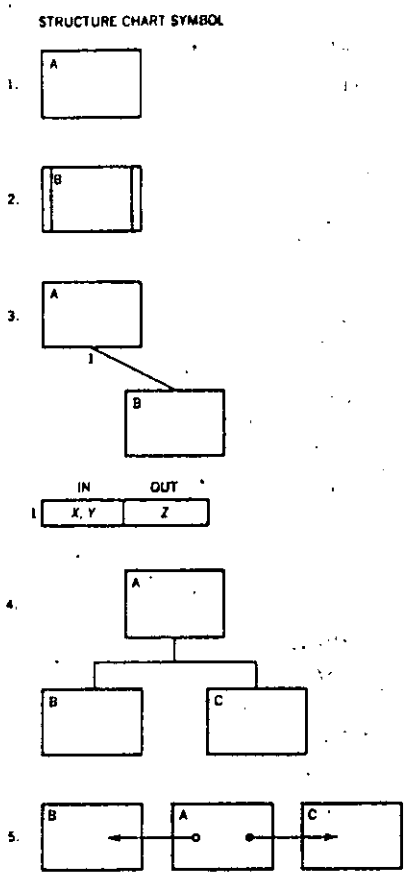
Tradeoffs to structured design

The overhead involved in writing many simple modules is in the execution time and memory space used by a particular language to effect the call. The designer should realize the adverse effect on maintenance and debugging that may result from striving just for minimum execution time and/or memory. He should also remember that programmer cost, is, or is rapidly becoming, the major cost of a programming system and that much of the maintenance will be in the future when the trend will be even more prominent. However, depending on the actual overhead of the language being used, it is very possible that a structured design can result in less execution and/or memory overhead rather than more due to the following considerations:

For memory overhead

1. Optional (error) modules may never be called into memory.

Figure 6 Definitions of symbols used in structure charts



THE MORE COMPREHENSIVE "PROPOSED STANDARD GRAPHICS FOR PROGRAM STRUCTURE," PREFERRED BY MR. CONSTANTINE AND WIDELY USED OVER THE PAST SIX YEARS BY HIS CLASSES AND CLIENTS, USES SEPARATE ARROWS FOR EACH CONNECTION. SUCH AS FOR THE CALLS FROM A TO B AND FROM A TO C, TO REFLECT STRUCTURAL PROPERTIES OF THE PROGRAM. THE CHARTING SHOWN HERE WAS ADOPTED FOR COMPATIBILITY WITH THE HIERARCHY CHART OF HIPO.

2. Structured design reduces duplicate code and the coding necessary for implementing control switches, thus reducing the amount of programmer-generated code.
3. Overlay structuring can be based on actual operating characteristics obtained by running and observing the program.
4. Having many single-function modules allows more flexible, and precise, grouping, possibly resulting in less memory needed at any one time under overlay or virtual storage constraints.

For execution overhead

1. Some modules may only execute a few times.
2. Optional (error) functions may never be called, resulting in zero overhead.
3. Code for control switches is reduced or eliminated, reducing the total amount of code to be executed.

4. Heavily used linkage can be recompiled and calls replaced by branches.
5. "Includes" or "performs" can be used in place of calls. (However, the complexity of the system will increase by at least the extra consideration necessary to prevent duplicating data names and by the difficulty of creating the equivalent of call parameters for a well-defined interface.)
6. One way to get fast execution is to determine which parts of the system will be most used so all optimizing time can be spent on those parts. Implementing an initially structured design allows the testing of a working program for those critical modules (and yields a working program prior to any time spent optimizing). Those modules can then be optimized separately and reintegrated without introducing multitudes of errors into the rest of the program.

Structured design techniques

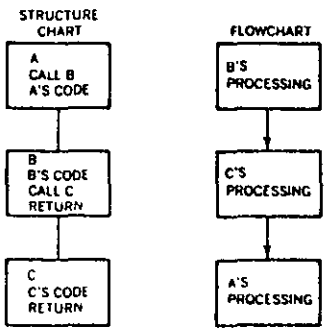
It is possible to divide the design process into general program design and detailed design as follows. General program design is deciding *what* functions are needed for the program (or programming system). Detailed design is *how* to implement the functions. The considerations above and techniques that follow result in an identification of the functions, calling parameters, and the call relationships for a structure of functionally bound, simply connected modules. The information thus generated makes it easier for each module to then be separately designed, implemented, and tested.

structure charts

The objective of general program design is to determine what functions, calling parameters, and call relationships are needed. Since flowcharts depict *when* (in what order and under what conditions) blocks are executed, flowcharts unnecessarily complicate the general program design phase. A more useful notation is the structure chart, as described earlier and as shown in Figure 6.

To contrast a structure chart and a flowchart, consider the following for the same three modules in Figure 7—A which calls B which calls C (coding has been added to the structure chart to enable the proper flowchart to be determined; B's code will be executed first, then C's, then A's). To design A's interfaces properly, it is necessary to know that A is responsible for invoking B, but this is hard to determine from the flowchart. In addition, the structure chart can show the module connections and calling parameters that are central to the consideration and techniques being presented here.

Figure 7 Structure chart compared to flowchart



The other major difference that drastically simplifies the nota-

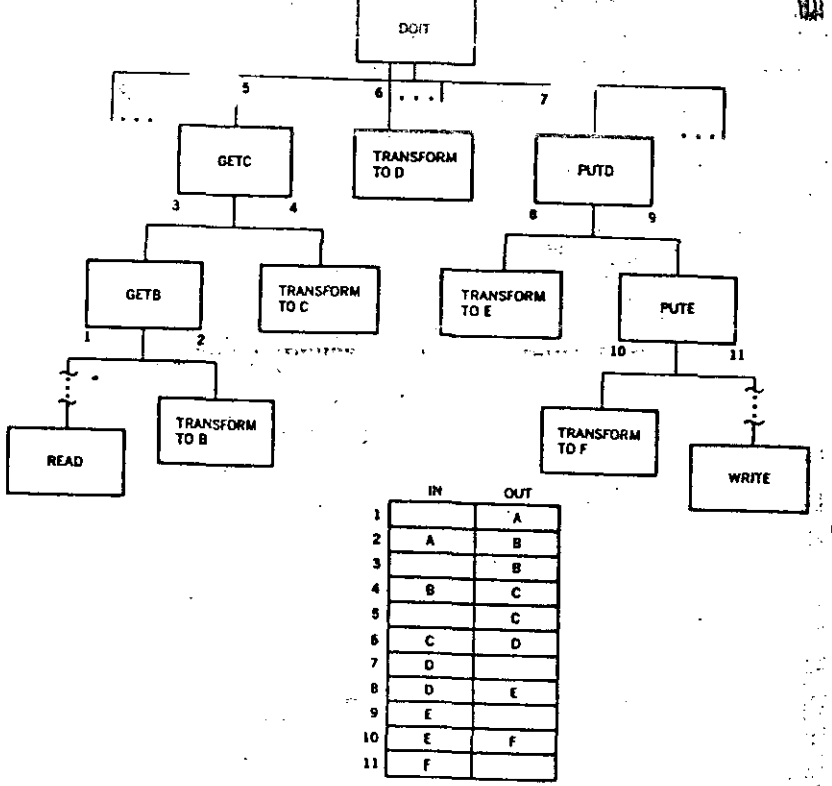
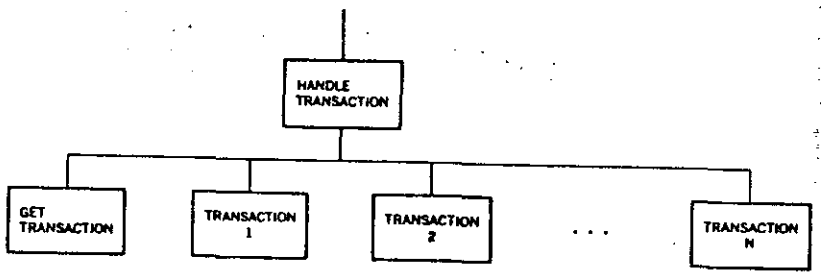
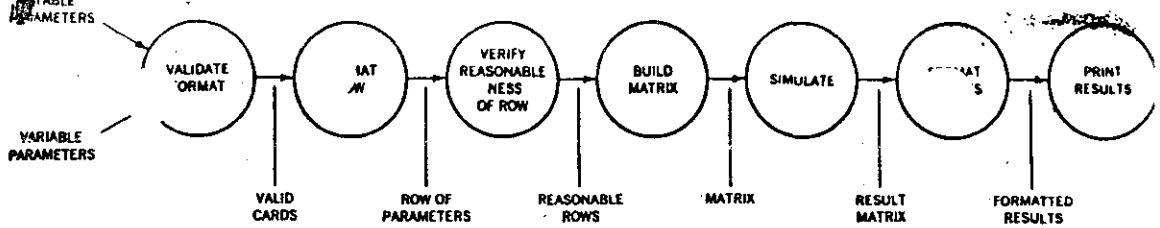


Figure 9 Transaction structure



tion and analysis during general program design is the absence in structure charts of the decision block. Conditional calls can be so noted, but "decision designing" can be deferred until detailed module design. This is an example of where the *design* process is made simpler by having to consider only part of the design problem. Structure charts are also small enough to be worked on all at once by the designers, helping to prevent suboptimizing parts of the program at the expense of the entire problem.

A shortcut for arriving at simple structures is to know the general form of the result. Mr. Constantine observed that programs of the general structure in Figure 8 resulted in the lowest-cost



implementations. It implements the input-process-output type of program, which applies to most programs, even if the "input" or "output" is to secondary storage or to memory.

In practice, the sink leg is often shorter than the source one. Also, source modules may produce output (e.g., error messages) and sink modules may request input (e.g., execution-time format commands.)

Another structure useful for implementing parts of a design is the transaction structure depicted in Figure 9. A "transaction" here is any event, record, or input, etc. for which various actions should result. For example, a command processor has this structure. The structure may occur alone or as one or more of the source (or even sink) modules of an input-process-output structure. Analysis of the transaction modules follows that of a transform module, which is explained later.

designing the structure

The following procedure can be used to arrive at the input-process-output general structure shown previously.

Step One. The first step is to sketch (or mentally consider) a functional picture of the problem. As an example, consider a simulation system. The rough structure of this problem is shown in Figure 10.

Step Two. Identify the external conceptual streams of data. An *external* stream of data is one that is external to the system. A *conceptual* stream of data is a stream of related data that is independent of any physical I/O device. For instance, we may have several conceptual streams coming from one I/O device or one stream coming from several I/O devices. In our simulation system, the external conceptual streams are the input parameters, and the formatted simulation the result.

Step Three. Identify the *major* external conceptual stream of data (both input and output) in the problem. Then, using the diagram of the problem structure, determine, for this stream, the points of "highest abstraction" as in Figure 11.

Figure 11 Determining points of highest abstraction

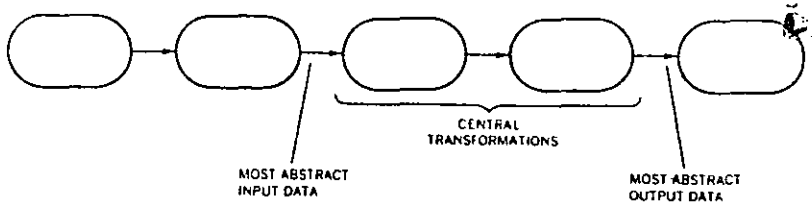
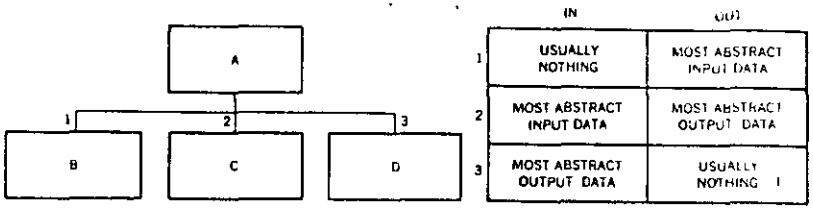


Figure 12 The top level



The "point of highest abstraction" for an input stream of data is the point in the problem structure where that data is farthest removed from its physical input form yet can still be viewed as coming in. Hence, in the simulation system, the most abstract form of the input transaction stream might be the built matrix. Similarly, identify the point where the data stream can first be viewed as going out – in the example, possibly the result matrix.

Admittedly, this is a subjective step. However, experience has shown that designers trained in the technique seldom differ by more than one or two blocks in their answers to the above.

Step Four. Design the structure in Figure 12 from the previous information with a source module for each conceptual input stream which exists at the point of most abstract input data; do sink modules similarly. Often only single source and sink branches are necessary. The parameters passed are dependent on the problem, but the general pattern is shown in Figure 12.

Describe the function of each module with a short, concise, and specific phrase. Describe what transformations occur when that module is called, not how the module is implemented. Evaluate the phrase relative to functional binding.

When mo A is called, the program or system executes. Hence, the ction of module A is equivalent to the problem being solved. If the problem is "write a FORTRAN compiler," then the function of module A is "compile

Module B's function involves obtaining the major stream of data. An example of a "typical module B" is "get next valid source statement in Polish form."

Module C's purpose is to transform the major input stream into the major output stream. Its function should be a nonprocedural description of this transformation. Examples are "convert Polish form statement to machine language statement" or "using keyword list, search abstract file for matching abstracts."

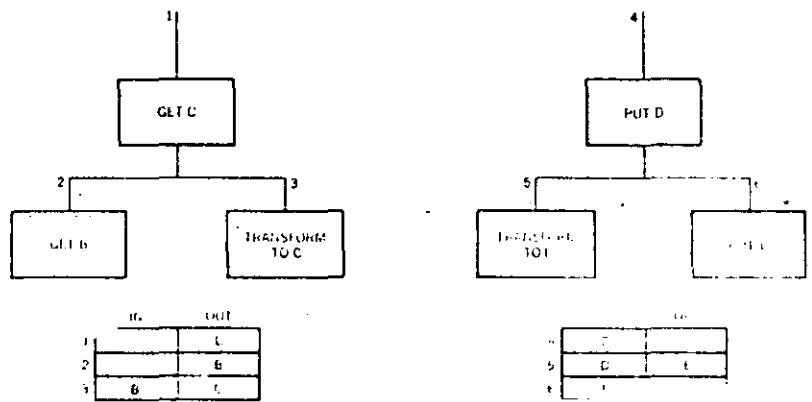
Module D's purpose is disposing of the major output stream. Examples are "produce report" or "display results of simulation."

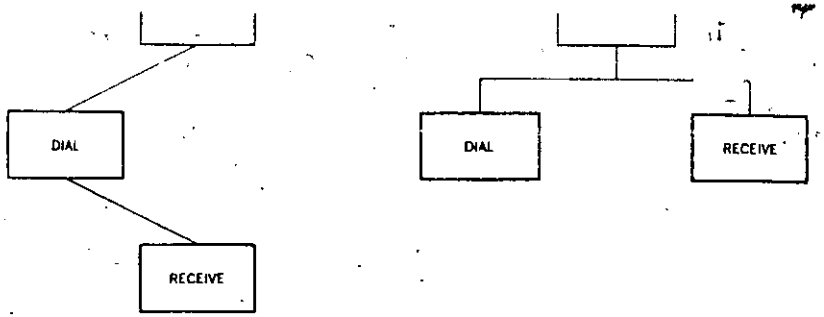
Step Five. For each source module, identify the last transformation necessary to produce the form being returned by that module. Then identify the form of the input just prior to the last transformation. For sink modules, identify the first process necessary to get closer to the desired output and the resulting output form. This results in the portions of the structure shown in Figure 13.

Repeat Step Five on the new source and sink modules until the original source and final sink modules are reached. The modules may be analyzed in any order, but each module should be done completely before doing any of its subordinates. There are, unfortunately, no detailed guidelines available for dividing the transform modules. Use binding and coupling considerations, size (about one page of source), and usefulness (are there subfunctions that could be useful elsewhere now or in the future) as guidelines on how far to divide.

During this phase, err on the side of dividing too finely. It is always easy to recombine later in the design, but duplicate func-

Figure 13 Lower levels





tions may not be identified if the dividing is too conservative at this point.

Design guidelines

The following concepts are useful for achieving simple designs and for improving the "first-pass" structures.

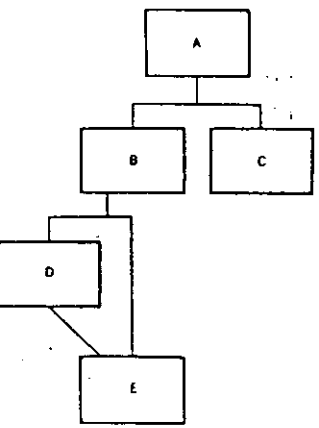
match program to problem

One of the most useful techniques for reducing the effect of changes on the program is to make the structure of the design match the structure of the problem, that is, form should follow function. For example, consider a module that dials a telephone and a module that receives data. If receiving immediately follows dialing, one might arrive at design A as shown in Figure 14. Consider, however, whether receiving is part of dialing. Since it is not (usually), have DIAL's caller invoke RECEIVE as in design B.

If, in this example, design A were used, consider the effect of a new requirement to transmit immediately after dialing. The DIAL module receives first and cannot be used, or a switch must be passed, or another DIAL module has to be added.

To the extent that the design structure does match the problem structure, changes to single parts of the problem result in changes to single modules.

Figure 15 Scope of control



scopes of effect and control

The *scope of control* of a module is that module plus all modules that are ultimately subordinate to that module. In the example of Figure 15, the scope of control of B is B, D, and E. The *scope of effect* of a decision is the set of all modules that contain some code whose execution is based upon the outcome of the decision. The system is simpler when the scope of effect of a decision is in the scope of control of the module containing the decision. The following example illustrates why.

flag to A or the decision will have to be repeated in A. The former approach results in added coding to implement the flag, and the latter results in some of B's function (decision) in module A. Duplicates of decision X result in difficulties coordinating changes to both copies whenever decision X must be changed.

The scope of effect can be brought within the scope of control either by moving the decision element "up" in the structure, or by taking those modules that are in the scope of effect but not in the scope of control and moving them so that they fall within the scope of control.

Size can be used as a signal to look for *potential* problems. Look carefully at modules with less than five or more than 100 executable source statements. Modules with a small number of statements may not perform an entire function, hence, may not have functional binding. Very small modules can be eliminated by placing their statements in the calling modules. Large modules may include more than one function. A second problem with large modules is understandability and readability. There is evidence to the fact that a group of about 30 statements is the upper limit of what can be mastered on the first reading of a module listing.[11]

Often, part of a module's function is to notify its caller when it cannot perform its function. This is accomplished with a return error parameter (preferably binary only). A module that handles streams of data must be able to signal end-of-file (EOF), preferably also with a binary parameter. These parameters should not, however, tell the caller what to do about the error or EOF. Nevertheless, the system can be made simpler if modules can be designed without the need for error flags.

Similarly, many modules require some initialization to be done. An initialize module will suffer from low binding but sometimes is the simplest solution. It may, however, be possible to eliminate the need for initializing without compromising "black-boxness" (the same inputs *always* produce the same outputs). For example, a read module that detects a return error of file-not-opened from the access method and recovers by opening the file and rereading eliminates the need for initialization without maintaining an internal state.

Eliminate duplicate functions but not duplicate code. When a function changes, it is a great advantage to only have to change it in one place. But if a module's need for its own copy of a random collection of code changes slightly, it will not be necessary to change several other modules as well.

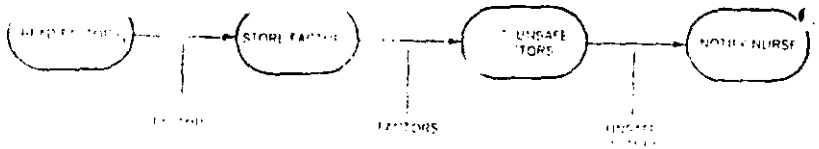
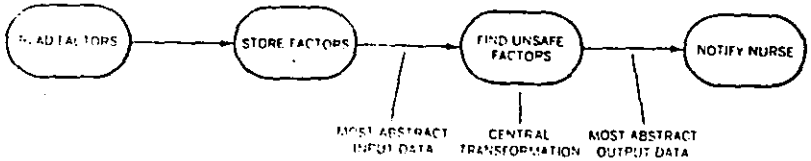


Figure 17 Points of highest abstraction



If a module seems almost, but not quite, useful from a second place in the system, try to identify and isolate the useful subfunction. The remainder of the module might be incorporated in its original caller.

Check modules that have many callers or that call many other modules. While not always a problem, it may indicate missing levels or modules.

Isolate specifications

Isolate all dependencies on a particular data-type, record-layout, index-structure, etc. in one or a minimum of modules. This minimizes the recoding necessary should that particular specification change.

reduce parameters

Look for ways to reduce the number of parameters passed between modules. Count every item passed as a separate parameter for this objective (independent of how it will be implemented). Do not pass whole records from module to module, but pass only the field or fields necessary for each module to accomplish its function. Otherwise, all modules will have to change if one field expands, rather than only those which directly used that field. Passing only the data being processed by the program system with necessary error and EOF parameters is the ultimate objective. Check binary switches for indications of scope-of-effect/scope-of-control inversions.

Have the designers work together and with the complete structure chart. If branches of the chart are worked on separately, comm modules may be missed and incompatibilities result from gn decisions made while only considering one branch.

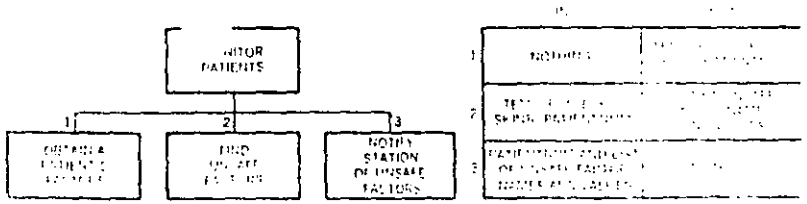
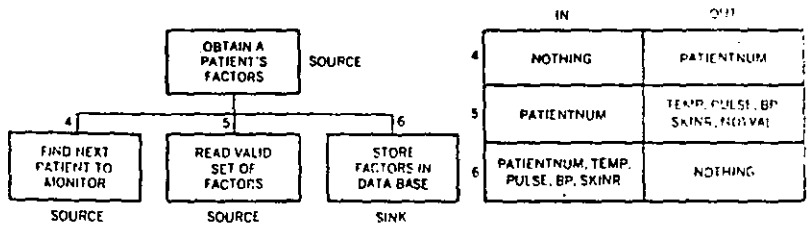


Figure 19 Structure of next level



An example

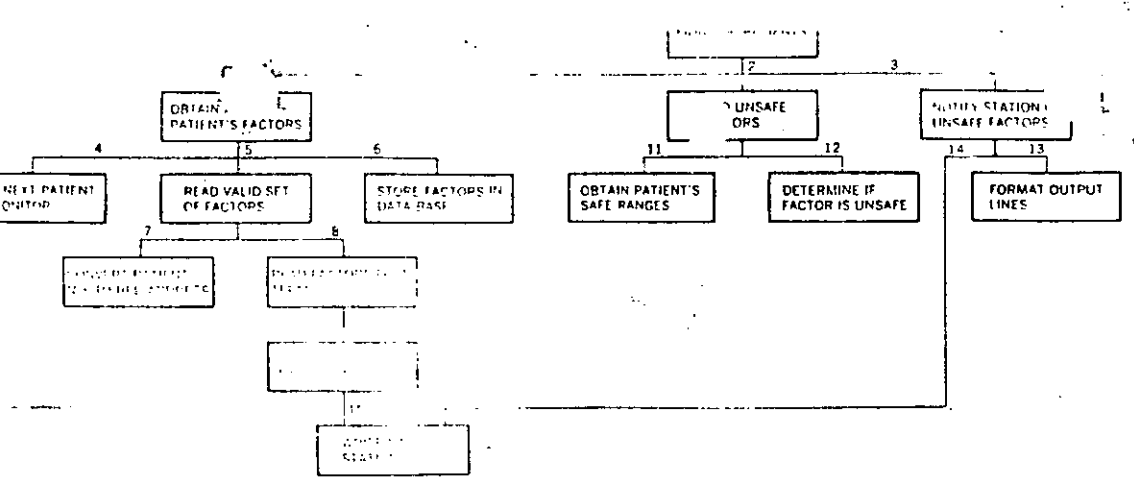
The following example illustrates the use of structured design:

A patient-monitoring program is required for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a data base. For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range is 98 to 99.5 degrees Fahrenheit). If a factor falls outside of a patient's safe range, or if an analog device fails, the nurse's station is notified.

In a real-life case, the problem statement would contain much more detail. However, this one is of sufficient detail to allow us to design the structure of the program.

The first step is to outline the structure of the problem as shown in Figure 16. In the second step, we identify the external conceptual streams of data. In this case, two streams are present, factors from the analog device and warnings to the nurse. These also represent the major input and output streams.

Figure 17 indicates the point of highest abstraction of the input stream, which is the point at which a patient's factors are in form to store in the data base. The point of highest abstractio the output stream is a list of unsafe factors (if any). We can now begin to design the program's structure as in Figure 18.



IN	OUT
	TEMP, PULSE, BP, SKINR, PATIENTNUM
TEMP, PULSE, BP, SKINR, PATIENTNUM	LIST OF UNSAFE FACTOR NAMES AND VALUES
PATIENTNUM & LIST OF UNSAFE FACTOR NAMES & VALUES	
	PATIENTNUM
PATIENTNUM	TEMP, PULSE, BP, SKINR, NOTVAL
PATIENTNUM, TEMP, PULSE, BP, SKINR	
PATIENTNUM	BEDNUM
BEDNUM	TEMP, PULSE, BP, SKINR, NOTVAL
BEDNUM	
LINE	
PATIENTNUM	TEMPR, PULSER, BPR, SKINRR
FACTOR, RANGE	UNSAFE
LIST OF UNSAFE FACTOR NAMES AND VALUES	LIST OF LINES

In analyzing the module "OBTAIN A PATIENT'S FACTORS," we can deduce from the problem statement that this function has three parts: (1) Determine which patient to monitor next (based on their specified periodic intervals). (2) Read the analog device. (3) Record the factors in the data base. Hence, we arrive at the structure in Figure 19. (NOTVAL is set if a valid set of factors was not available.)

Further analysis of "READ VALID SET OF FACTORS", "FIND UNSAFE FACTORS" and "NOTIFY STATION OF UNSAFE FACTORS" yields the results shown in the complete structure chart in Figure 20.

Note that the module "READ FACTORS FROM TERMINAL" contains a decision asking "did we successfully read from the terminal?" If the read was not successful, we have to notify the nurse's station and then find the next patient to process as depicted in Figure 21.

Modules in the scope of effect of this decision are marked with an X. Note that the scope of effect is *not* a subset of the scope

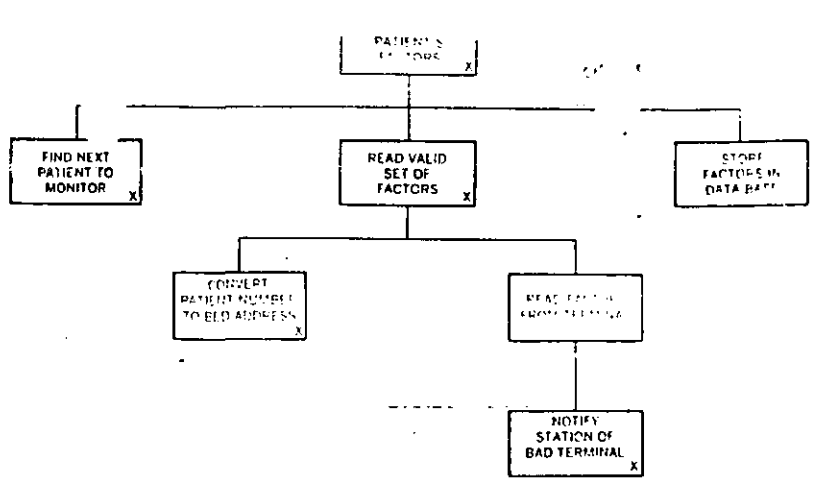
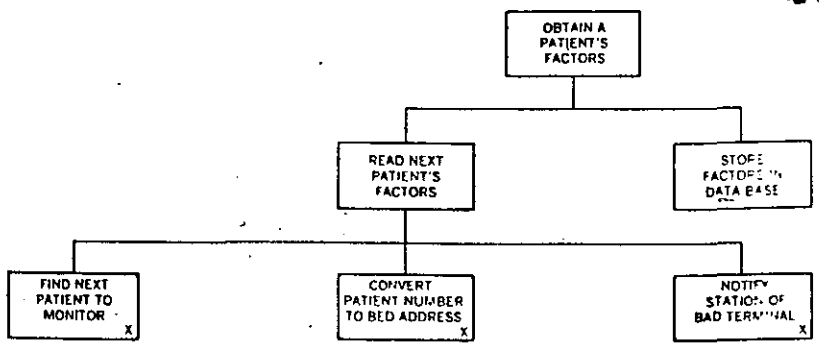


Figure 22 Scope of effect within scope of control



of control. To correct this problem, we have to take two steps. First, we will move the decision up to "READ VALID SET OF FACTORS." We do this by merging "READ FACTORS FROM TERMINAL" into its calling module. We now make "FIND NEXT PATIENT TO MONITOR" a subordinate of "READ VALID SET OF FACTORS." Hence, we have the structure in Figure 22. Thus, by slightly altering the structure and the function of a few modules, we have completely eliminated the problem.

Concluding remarks

The HIPO Hierarchy chart is being used as an aid during general systems design. The considerations and techniques presented here are useful for evaluating alternatives for those portions of the system that will be programmed on a computer. The charting technique used here depicts more details about the interfaces than the HIPO Hierarchy chart. This facilitates consideration during general program design of each individual connection and

archy chart would still show all the functions in separate blocks.) The output of the general gram design is the input for the detailed module design. The HIPO input-process-output chart is useful for describing and designing each module.

Structured design considerations could be used to review program designs in a walk-through environment.[12] These concepts are also useful for evaluating alternative ways to comply with the requirement of structured programming for one-page segments.[2]

Structured design reduces the effort needed to fix and modify programs. If all programs were written in a form where there was one module, for example, which retrieved a record from the master file given the key, then changing operating systems, file access techniques, file blocking, or I/O devices would be greatly simplified. And if *all* programs in the installation retrieved from a given file with the same module, then one properly rewritten module would have *all* the installation's programs working with the new constraints for that file.

However, there are other advantages. Original errors are reduced when the problem at hand is simpler. Each module is self-contained and to some extent may be programmed independently of the others in location, programmer, time, and language. Modules can be tested before all programming is done by supplying simple "stub" modules that merely return preformatted results rather than calculating them. Modules critical to memory or execution overhead can be optimized separately and reintegrated with little or no impact. An entry or return trace-module becomes very feasible, yielding a very useful debugging tool.

Independent of all the advantages previously mentioned, structured design would *still* be valuable to solve the following problem alone. Programming can be considered as an art where each programmer usually starts with a blank canvas – techniques, yes, but still a blank canvas. Previous coding is often not used because previous modules usually contain, for example, *at least* GET and EDIT. If the EDIT is not the one needed, the GET will have to be recoded also.

Programming can be brought closer to a science where current work is built on the results of earlier work. Once a module is written to get a record from the master file given a key, it can be used by all users of the file and need not be rewritten into each

Structured d n concepts are not new. The whole assembly-line idea is one of isolating simple functions in a way that still produces a complete, complex result. Circuits are designed by connecting isolatable; functional stages together, not by designing one big, interrelated circuit. Page numbering is being increasingly sectionalized (e.g., 4–101) to minimize the "connections" between written sections, so that expanding one section does not require renumbering other sections. Automobile manufacturers, who have the most to gain from shared system elements, finally abandoned even the coupling of the windshield wipers to the engine vacuum due to effects of the engine load on the performance of the wiping function. Most other industries know well the advantage of isolating functions.

It is becoming increasingly important to the data-processing industry to be able to produce more programming systems and produce them with fewer errors, at a faster rate, and in a way that modifications can be accomplished easily and quickly. Structured design considerations can help achieve this goal.

CITED REFERENCES AND FOOTNOTES

1. This method has not been submitted to any formal IBM test. Potential users should evaluate its usefulness in their own environment prior to implementation.
2. L. L. Constantine, *Fundamentals of Program Design*, in preparation for publication by Prentice-Hall, Englewood Cliffs, New Jersey.
3. G. J. Myers, *Composite Design: The Design of Modular Programs*, Technical Report TR00.2406, IBM, Poughkeepsie, New York (January 29, 1973).
4. G. J. Myers, "Characteristics of composite design," *Datamation* 19, No. 9, 100–102 (September 1973).
5. G. J. Myers, *Reliable Software through Composite Design*, to be published Fall of 1974 by Mason and Lipscomb Publishers, New York, New York.
6. HIPO – Hierarchical Input-Process-Output documentation technique. Audio education package, Form No. SR20-9413, available through any IBM Branch Office.
7. F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal* 11, No. 1, 56–73 (1972).
8. The use of the HIPO Hierarchy charting format is further illustrated in Figure 6, and its use in this paper was initiated by R. Ballow of the IBM Programming Productivity Techniques Department.
9. L. A. Belady and M. M. Lehman, *Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth*", RC 3546, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1971).
10. L. L. Constantine, "Control of sequence and parallelism in modular programs," *AFIPS Conference Proceedings, Spring Joint Computer Conference* 32, 409 (1968).
11. G. M. Weinberg, *PL/I Programming: A Manual of Style*, McGraw-Hill, New York, New York (1970).
12. *Improved Programming Technologies: Management Overview*, IBM Corporation, Data Processing Division, White Plains, New York (August).

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

TEMA IV

PROGRAMACION ESTRUCTURADA

FIS. RAYMUNDO HUGO RANGEL GUTIERREZ

MAYO, 1985

4.- PROGRAMACION ESTRUCTURADA

4.1 Origenes y objetivos de la programación estructurada

4.2 El proceso de refinamiento a pasos

4.3 El seudocódigo

4.4 Arboles de decisión

4.5 Diagramas estructurados

4.6 Documentación de programas

4.7 Estructuras de control en FORTRAN IV.

4.1 ORIGENES Y OBJETIVOS DE LA PROGRAMACION ESTRUCTURADA

Dos artículos de Dijkstra marcan el inicio de la programación estructurada, estos son:

- Programming considered as a human activity (Dijkstra - 1965)

- GO TO statement considered harmful (Dijkstra 1968)

En estos dos artículos ya Dijkstra habla de algunos conceptos e ideas que han llegado a ser relevantes en la programación estructurada tales como:

- Argumentos contra GO TO
- La noción del diseño de arriba-a-abajo
- El énfasis en la calidad de programas
- Argumentos contra programas que se modifican a si mismos.
- Otros

La programación estructurada surge como una necesidad de - reducir la complejidad de los grandes programas.
Con frecuencia se observa que los programas, debido a su - complejidad, no

1.- satisfacen las necesidades del usuario
2.- no se producen a tiempo
3.- cuestan mas de lo estimado
4.- contienen errores y
5.- son difíciles de darles mantenimiento

Es por ello que la programación estructurada se propone lograr los siguientes objetivos:

1.- Que los programas satisfagan los requerimientos especificados.
2.- Programas que se les pueda dar mantenimiento
3.- Minimizar el número de errores que ocurren durante el desarrollo.
4.- Operación resistente a errores.

5.- Programas transportables

6.- Programas cuya lógica sea legible

7.- Minimizar costos

4.2 EL PROCESO DE REFINAMIENTO A PASOS

Nuestra mayor herramienta en el desarrollo de programas es nuestra capacidad de abstracción.
Básicamente esto consiste (cuando resolvemos un problema) en analizar cuidadosamente el problema dado hasta lograr una representación mental de los elementos escenciales del mismo.
Cada elemento debe tener un propósito bien definido. Este proceso puede aplicarse a su vez a cada uno de los elementos del problema original, es decir, consideramos a cada elemento como un nuevo problema (subproblema). El proceso puede repetirse para los elementos del subproblema y asi sucesivamente.
Inicialmente (en la resolución del problema) estamos concentrados en que funciones (o elementos) descomponer el problema, y a medida que procedemos a descomponerlos sistemáticamente llegamos a concentrarnos en como realizar cada función.
Esta forma de proceder va de lo general a lo particular.
El proceso descrito anteriormente se conoce como proceso de refinamiento a pasos y es central a la programación estructurada.
El proceso de refinamiento a pasos no es directo, ni trivial, como pudiera parecer a primera vista.
Este proceso es escencialmente un proceso de ensayo y error.

Es evidente que necesitamos una notación como herramienta para ir concretando la solución del proeblema, ya que a medida que se refina sistemáticamente la solución del problema, nuestra limitada capacidad de retención será desbordada por la gran cantidad de detalles involucrados. Una herramienta muy utilizada para este propósito es el seudcódigo o lenguaje de diseño de programas. El seudocódigo debe utilizarse como una extensión natural del proceso de abstracción.

4.3 EL SEUDOCODIGO

El seudocódigo o seudolenguaje es un lenguaje intermedio -- entre el lenguaje nativo del programador y el lenguaje de - programación en que se intenta implantar la solución. El_ seudocódigo le permite al programador pensar en la lógica y y expresar esa lógica en una forma semiformal sin tener que adentrarse en los detalles particulares de un lenguaje de - programación.

Básicamente el seudocódigo difiere en 2 aspectos de un lenguaje de programación.

1.- No existen restricciones sintácticas para el uso del -- seudocódigo. Solo las estructuras de control básicas_ y el sangrado para mejorar la claridad del alcance de - dichas estructuras, son las únicas convenciones aceptadas para el uso del seudocódigo.

2.- Cualquier operación se puede expresar a cualquier nivel de detalle, por ejemplo:

```
incrementar    contador
ctdor ←— ctdor + 1
```

El seudocódigo permite al programador tratar el problema a_ diversos niveles de abstracción, ya que esta es la herra--- mienta mediante la cual expresamos la solución de un proble ma durante el proceso de refinamiento a pasos, esto es, el - seudocódigo es un lenguaje de diseño de programas (PDL).

El seudocódigo es una forma conveniente para documentar los estados de desarrollo del programa, lo cual permite a otros programadores revisar la función del programa antes de imple mentarlo en un lenguaje de programación, además de facilitar una valoración del estado de desarrollo en cualquier estado_ del proceso de diseño de programa.

Debido a que el seudocódigo describe detalladamente el -- programa fuente completo, puede mantenerse como parte de la documentación del programa. Siendo así, el seudocó-- digo debe actualizarse cada vez que haya cambios en el -- programa fuente.

Existen pocas guías generales que hacen del seudocódigo - efectivo como una herramienta de diseño de programas. Estas son:

1.- Haga del seudocódigo una extensión del proceso del -- pensamiento.

2.- Sangre el código para resaltar la estructura de la -- lógica.

3.- De nombres a los datos de tal manera que reflejen su_ intención.

4.- Mantenga la lógica simple

5.- Utilice las estructuras de control básico permitidos en el proyecto.

Hay cuatro clases de estructuras básicas con las cuales - es posible especificar un procedimiento en seudocódigo.

a) Secuencia
b) Decisión
c) Iterativas
d) Salida

a) Secuencia

Concatenación

b) Decisión

1.- si (predicado) luego
seudocódigo
fin

2.- si (predicado) luego
    seudocódigo-A
  obien
    seudocódigo-B
  fin

3.- si (predicado-A) luego
    seudocódigo-A
  osi (predicado-B) luego
    seudocódigo-B
  osi (predicado-C) luego
    .
    .
    .
  obien
    seudocódigo-X luego
  fin

4.- Casar (selector) con
  (caso 1)
    seudocódigo-A
  (caso 2)
    seudocódigo-B
    .
    .
    .
  (obien)
    seudocódigo-X
  fin

c) Iteración

1- Entanto (predicado) luego
    seudocódigo
  fin

2- Repetir
    seudocódigo
  Hasta (predicado)

3- Desde (vc ←valor-1 hasta valor-2) repetir
    seudocódigo
  fin

d) Salida
  1- escape
  2- ciclo

4.4. ARBOLES DE DECISION

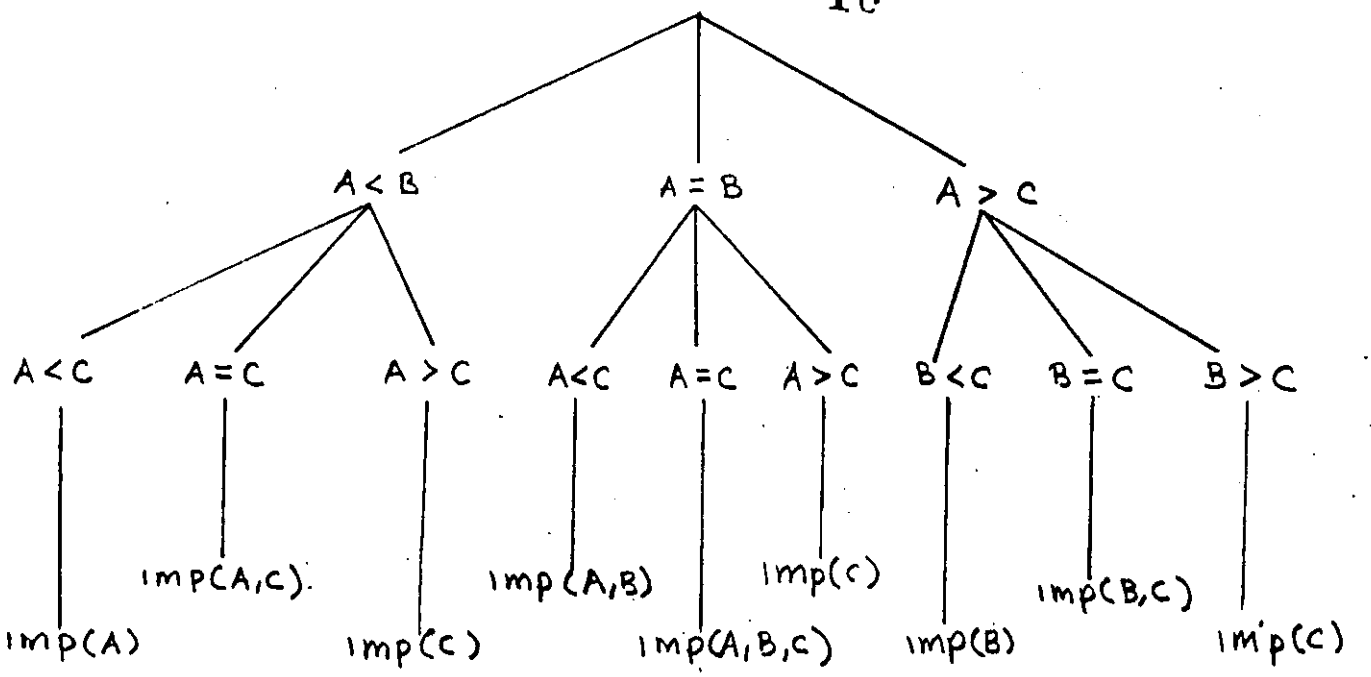
Los árboles de decisión son herramientas útiles que sirven para esclarecer, de manera sistemática, la lógica de segmentos de programa en los que aparecen condicionales anidados. El ejemplo siguiente ilustra el uso de esta herramienta.

Suponga que se quiere imprimir el menor valor de las 3 variables A, B y C de la siguiente manera:
Si una de las 3 variables es menor a las otras dos se imprime. Si dos de ellas son iguales y menores a la tercera se imprimen y si las 3 son iguales se imprimen.

Comparando primero A y B para todos los casos posibles tenemos:

A<B A=B A>B

para el primer caso se sabe que A es menor que B y por consiguiente enseguida se debe comparar A con C para todos los casos posibles. Para el segundo caso se sabe que A es igual que B y por consiguiente enseguida se debe comparar A o B con C para todos los casos posibles. Para el tercer caso se sabe que A es mayor que B y por consiguiente enseguida se debe comparar B on C para todos los casos posibles. Estas consideraciones pueden expresarse de forma gráfica de la manera siguiente:



Las líneas indican la relación jerárquica que existe entre las condiciones. Observe que las acciones que se toman una vez dada una condición o condiciones aparecen en el extremo inferior del árbol.

El mapeo de la lógica del árbol de decisiones a seudocódigo es directa.

```
si (A<B) luego
    si(A<C) luego
        imp(A)
    osi (A=C) luego
        impc(A,C)
    obien
        imp(C)
    fin
osi (A=B) luego
    si(A<C) luego
        imp(A,B)
    osi(A=C) luego
        imp(A,B,C)
    obien
        imp(c)
    fin
obien
    si(B<C) luego
        imp(B)
    osi(B=C) luego
        imp(B,C)
    obien
        imp(C)
    fin
fin
```

La correspondencia entre el árbol de decisión y el seudo-código es evidente.

a

TABLAS DE DECISION

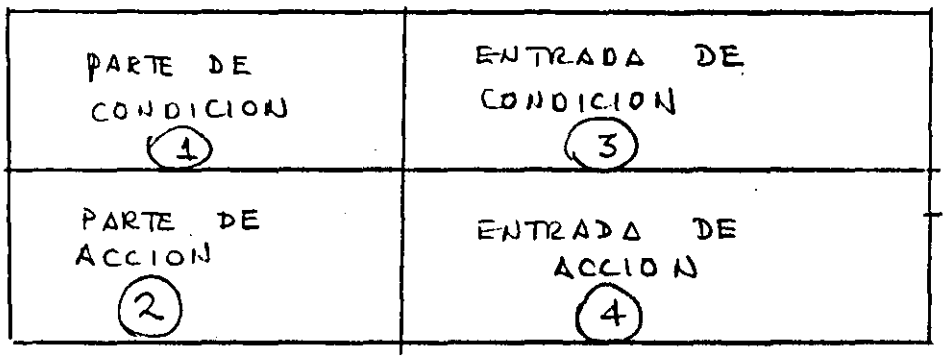
Una tabla de decisión, al igual que un arbol, es una - herramienta que permite escalarece la lógica de condiciona-- les complejos.

A diferencia de un arbol, una tabla se deriva de una - manera mucho mas sistemática, ademas de tener un formato mu- cho mas compacto.

Existen 3 tipos de trablas de decisión

a) Entrada limitada
b) Entrada extendida
c) Entrada mixta

El siguiente diagrama muestra el formato de una tabla_ de decisión:



1. En el cuadrante superior izquierdo esta la parte de condición. Esta área debe contener (en forma de pregunta) todas aquellas condiciones examinadas para un problema da do.

2. En el cuadrante inferior izquierdo esta la parte de acción. Esta área debe contener en forma de narrativa simple todas las acciones posibles resultantes de las condiciones listadas arriba.

3. En el cuadrante superior derecho esta la entrada de condición. En esta área todas las preguntas hechas en la parte de condición deben responderse y todas las combinaciones posibles de estas respuestas deben desarrollarse. Si una respuesta no se indica, puede asumirse que la condición no fue sometida a prueba en esa combinación particular.

4. En el cuadrante inferior derecho esta la entrada de acción. Las acciones apropiadas resultantes de las diversas condiciones de las respuestas a las condiciones de arriba se indican aquí. Una o mas acciones pueden indicarse para cada combinación de respuestas de condición.

   Las diversas combinaciones de respuestas a condiciones mostradas en la entrada de condición de la tabla y sus acciones resultantes se llaman reglas. A cada una se le da un número para propósitos de identificación.

   Otro elemento que requiere una tabla como un medio de distinguirla de otra tabla es un nombre.

Tablas de entrada limitada

Como un ejemplo simple de una tabla de entrada limitada, considerese el siguiente problema:

Se quiere escribir un procedimiento para una persona que indique en forma explicita que acciones tomar cuando sale de su cada en la mañana a su trabajo de acuerdo a las condiciones del tiempo.

Supóngase que las condiciones del tiempo son dia normal, llueve y hace frio y las acciones correspondientes son vestimenta normal, llevar paraguas y llevar sweater.

La tabla de decisión correspondiente queda como sigue:

¿LLUEVE?	S	S	N	N	
¿HACE FRIO?	S	N	S	N	
LLEVAR PARAGUAS	X	X			
HACE FRIO	X		X		
VESTIMENTA NORMAL			X		

Observese que en la entrada de condición sólo se tienen - 2 valores: S(si) y N(no) y en la entrada de acción una X que indica la acción que se toma de acuerdo a la condición o combinación de condiciones que se indica en la parte de condición. Así por ejemplo, la columna R1 (regla 1) se lee como sigue:

si ¿llueve y hace frío? luego

llevar paraguas y sweater

La columna R2 (regla 2)

si llueve y no hace frío luego

llevar paraguas

y así sucesivamente.

Tablas de entrada extendida

Una tabla de entrada extendida tiene en su entrada de condición mas de 2 valores, es decir, no esta limitada a SIS y NOS. Considérese el siguiente ejemplo:

Supóngase que en una línea aérea se tienen 3 clases de pasaje: primera, turista y alterna. Si un pasajero solicita la -- clase turista y hay disponible se le otorga, sino se le asigna la clase alterna. Si la clase alterna se agotó se pone en lista de espera. Si el cliente solicita clase primera se le -- otorga si la hay, en caso contrario se le asigna la clase alterna. Si no hay tampoco clase alterna se le pone en lista de es--

pera.

La tabla correspondiente de decisión, queda como sigue:

¿QUE PETICION?.	PP	PP	PP	PT	PT	PT	
¿QUE CLASE ?	CP	CT	CA	CP	CT	CA	
SE DA TURISTA					X		
SE DA PRIMERA	X						
SE DA ALTERNA			X				
PONER EN LISTA DE ESPERA		X		X		X	

PP = PETICION DE PRIMERA
PT = PETICION DE TURISTA
CP = CLASE PRIMERA
CT = CLASE TURISTA
CA = CLASE ALTERNA

Observese que en la entrada de condición se tienen 2 valores para la condición ¿que petición? (PP y PT) y para la condición ¿que clase? se tienen 3 valores (CP, CT y CA). La columna R1 (regla 1) se lee como sigue:

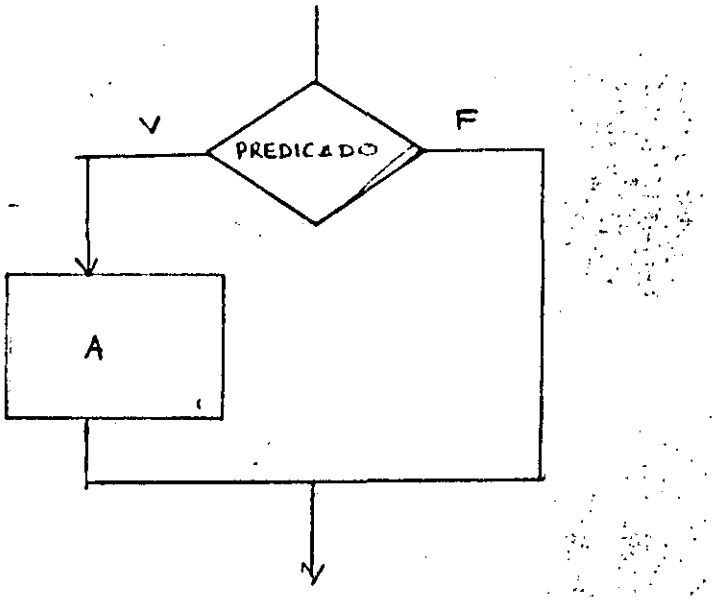
Si la petición es de primera y
la clase primera esta disponble luego
asignar clase primera

Tabla de entrada mixta

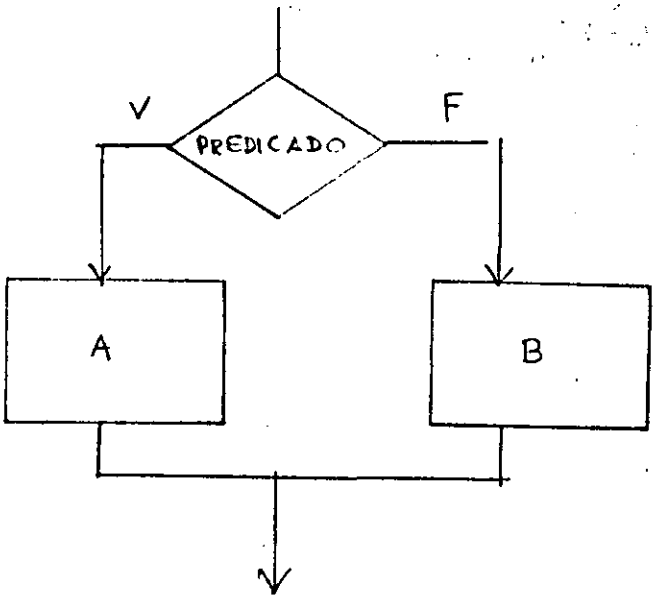
Una tabla de entrada mixta puede tener SIS y NOS y otros valores en la entrada de condición. Sí por ejemplo en la tabla anterior en la primera condición ¿qué petición? se podría poner SIS y NOS en su correspondiente entrada de condición, ya

que esta entrada tiene 2 valores PP y PT.

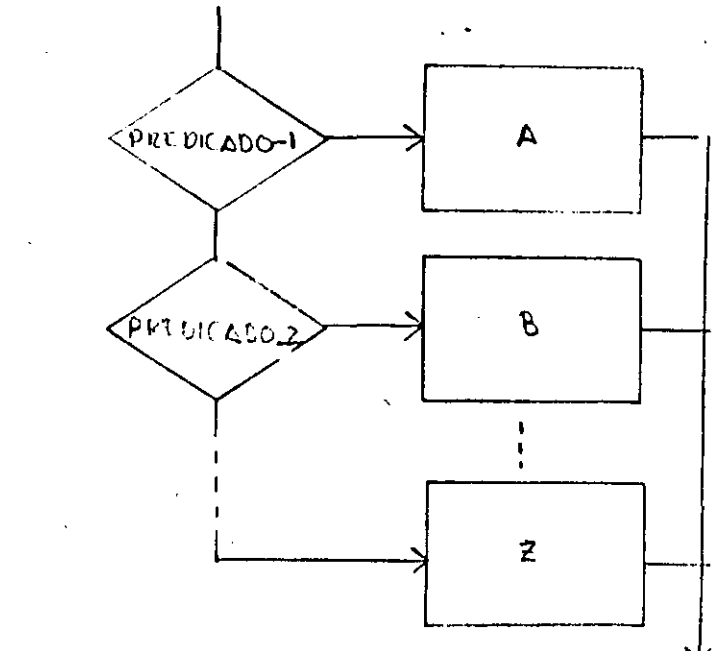
DESICION



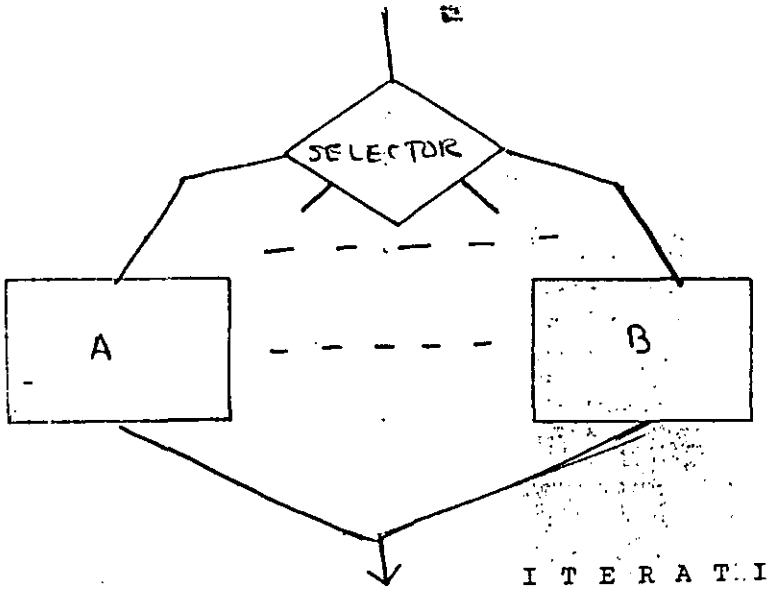

```
si (predicado) luego
      seudocódigo-A
fin
```




```
si (predicado) luego
      seudocódigo-A
o bien
      seudocódigo-B
fin
```

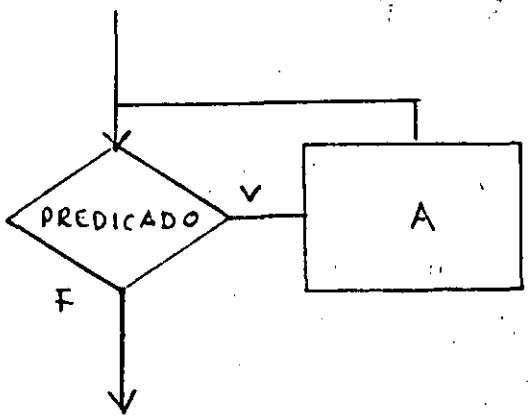



```
si (predicado) luego
      seudocódigo-A
as' (predicado) luego
      seudocódigo-B
.
.
.
o bien
      seudocódigo-Z
fin
```

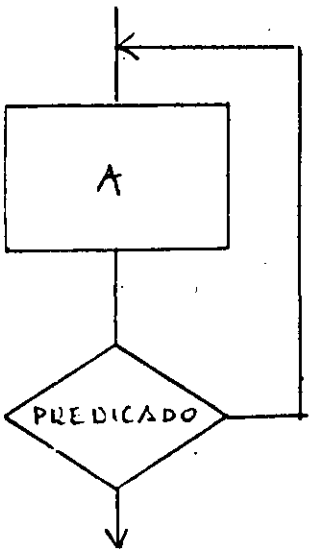



```
casar (selector) con
(caso1)
        seudocódigo-A
        .
        .
        .
(o bien)
        seudocódigo-Z
fin
```

I T E R A T I V A S



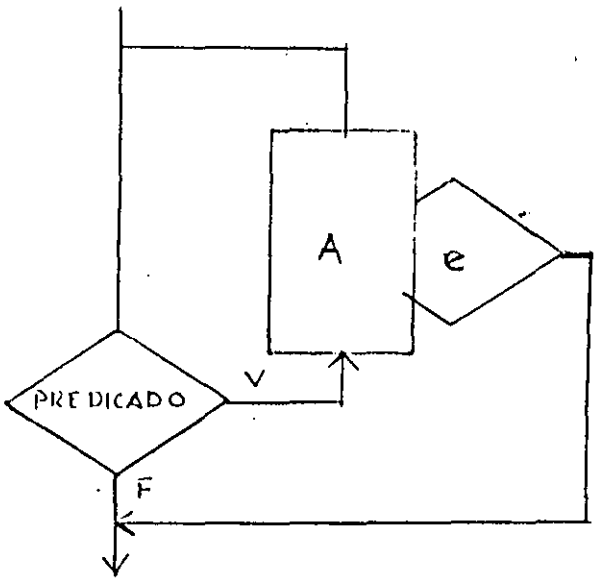

```
entanto (predicado) repetir
        seudocódigo-A
fin
```




```
repetir
      seudocódigo-A
hasta(predicado)
```

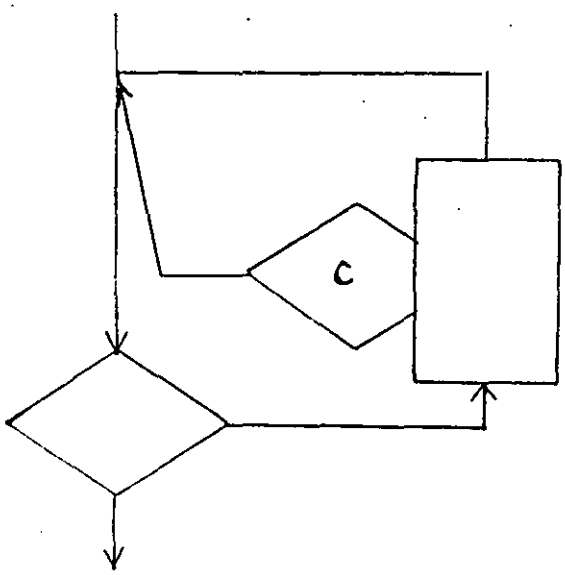
```
desde(ve←valor-1 hasta valor-2)
        repetir
        seudocódigo-A
fin
```

S A L I D A



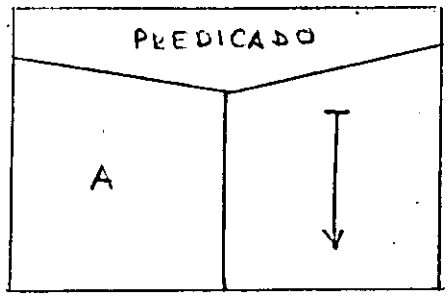
escape

ciclo

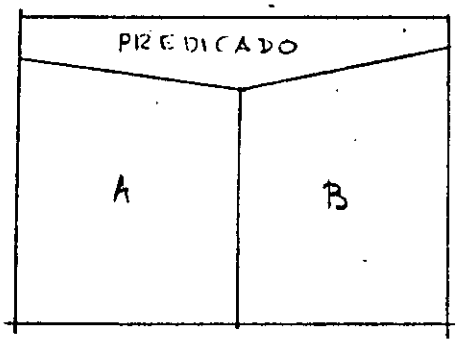


DIAGRAMAS ESTRUCTURADOS

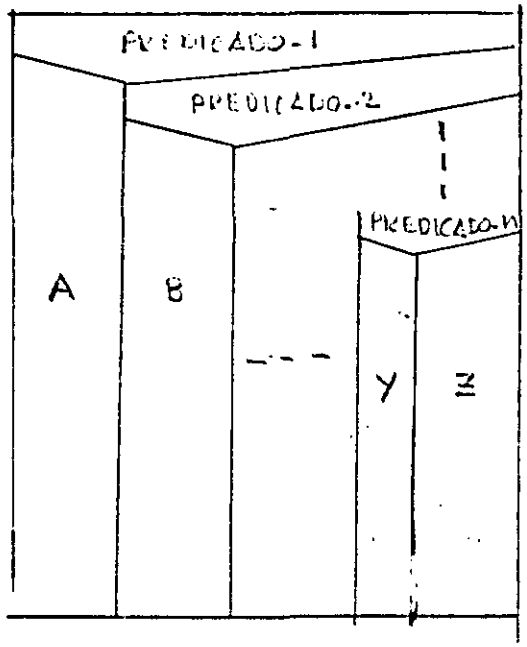
DESICION



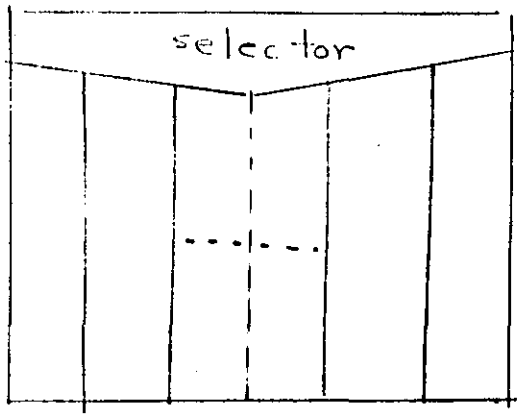

```
si (predicado) luego
        seudocódigo-A
fin
```




```
si (predicado) luego
        seudocódigo-A
o bien
        seudocódigo-B
fin
```

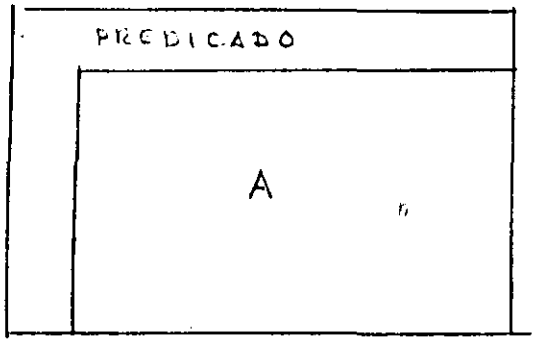



```
si (predicado) luego
        seudocódigo-A
osi (predicado) luego
        seudocódigo-B
.               .
.               .
.               .
o bien
        seudocódigo-Z
fin
```

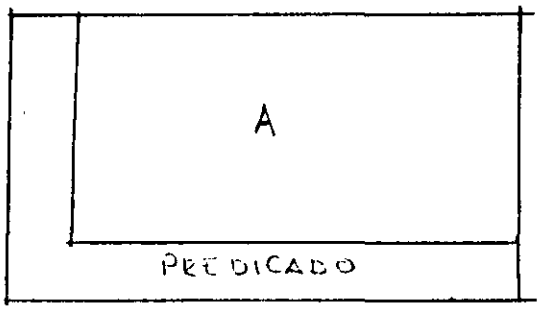



```
casar (selector) con
(caso1)
        seudocódigo-A
    .       .
    .       .
    .       .
(o bien)
        seudocódigo-Z
fin
```

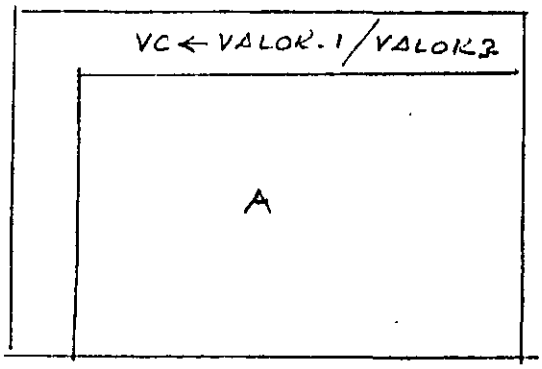
I T E R A T I V A S




```
entanto (predicado) repetir
        seudocódigo-A
fin
```

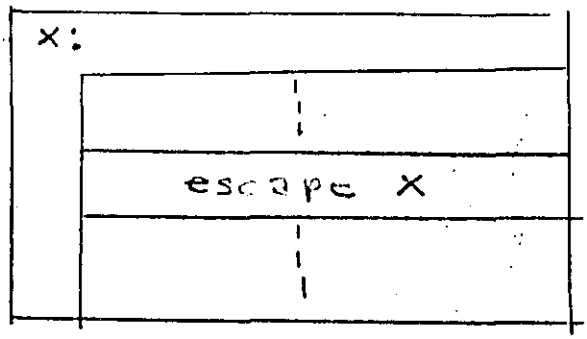



```
repetir
     seudocódigo-A
hasta(predicado)
```

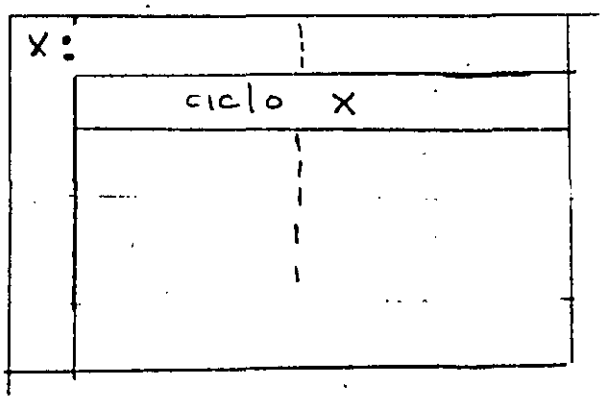



```
desde(ve←valor-1 hasta valor-1)
        repetir
        seudocódigo-A
fin
```

S A L I D A



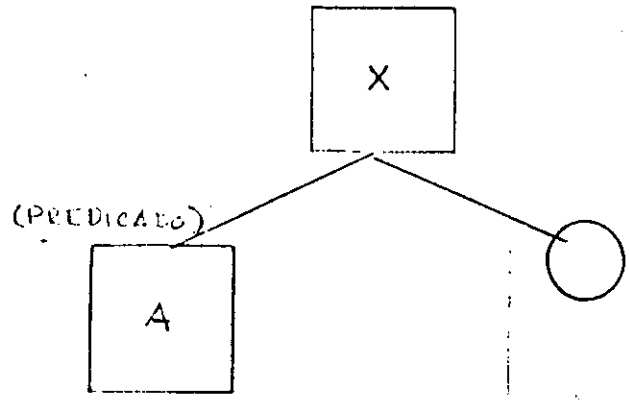
escape



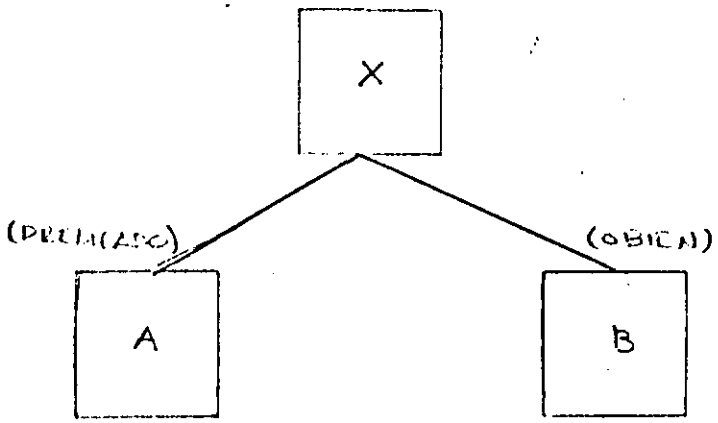
ciclo

D I A G R A M A S E S T R U C T U R A D O S

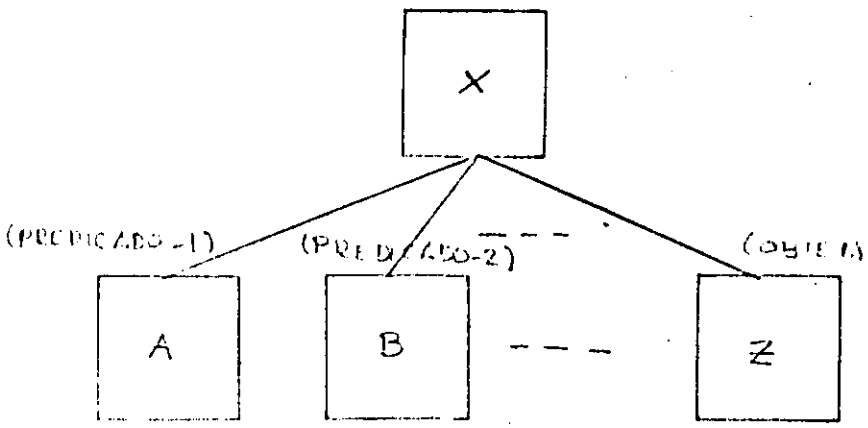
DESICION



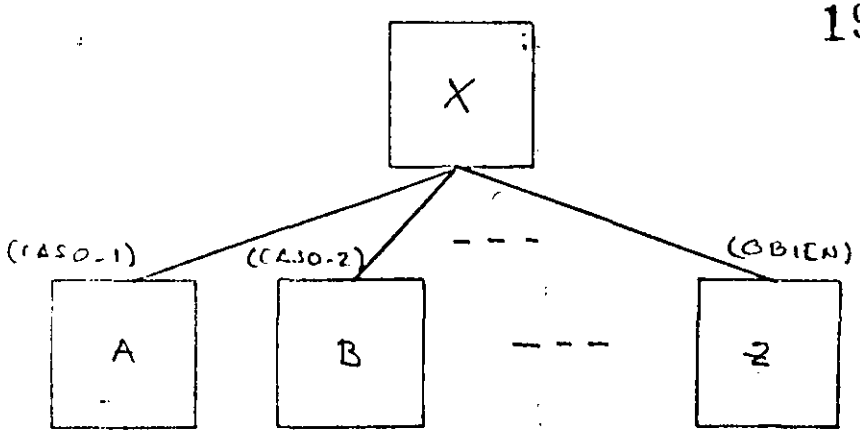

```
si (predicado)luego
        seudocódigo-A
fin
```




```
si (predicado) luego
        seudocódigo-A
o bien
        seudocódigo-B
fin
```



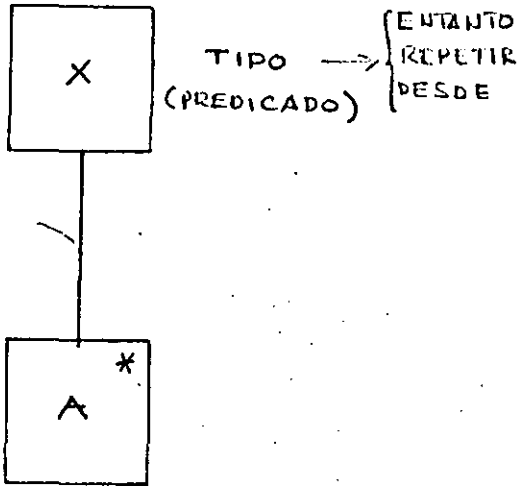

```
si (predicado) luego
        seudocódigo-A
as' (predicado) luego
        seudocódigo-B
.               .
.               .
.               .
o bien
        seudocódigo-Z
fin
```




```
casar (selector) con
(caso1)
        seudocódigo-A
   .          .
   .          .
   .          .
(o bien)
        seudocódigo-Z
fin
```

I T E R A T I V A S

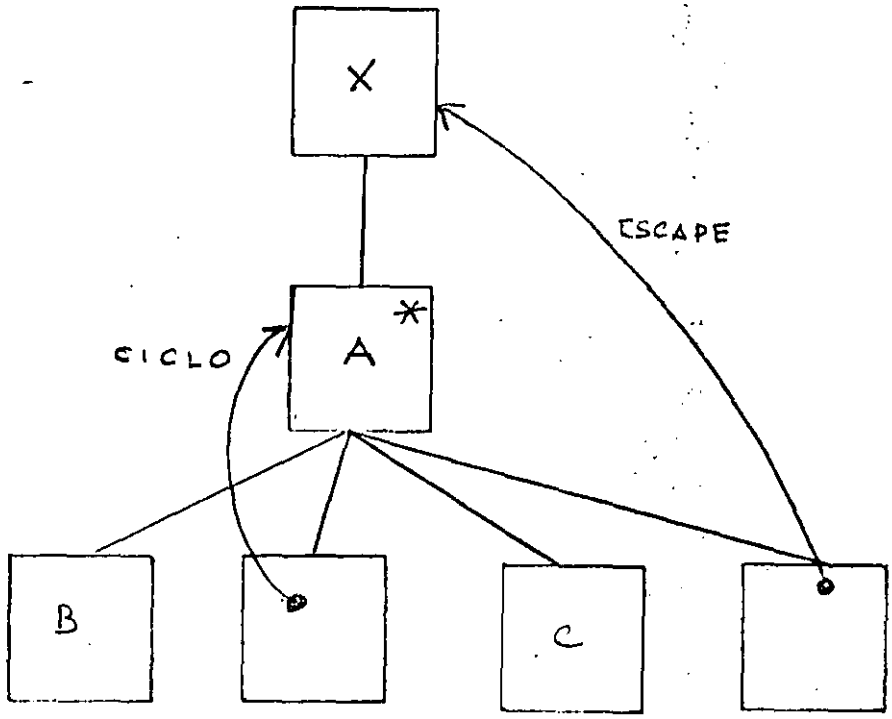
```
entanto (predicado) repetir
        seudocódigo-A
fin
```




```
repetir
    seudocódigo-A
hasta(predicado)
```

```
desde(ve← valor-1 hasta valor-2)
        repetir
        seudocódigo-A
fin
```

S A L I D A



escape

ciclo

4.6 DOCUMENTACION DE PROGRAMAS

No es suficiente presentar un programa como una secuencia de instrucciones en un orden dado, ni tampoco es suficiente usar identación para destacar visualmente el alcance de las estructuras de control. Es necesario documentar los programas ya que en muchas ocasiones unas personas codifican un programa y otras le dan mantenimiento.

Hay 2 tipos de documentación de programas:

a) Documentación interna
b) Documentación externa

La documentación interna consiste en comentarios inmersos en el mismo código ejecutable.

A continuación se dan algunas guías para la documentación interna de un programa.

1) Toda variable debe declararse. El nombre de la variable debe reflejar su propósito. Así, por ejemplo, la asignación

```
x:=2.0c*y**2;
```

no nos dice gran cosa, sin embargo si a las variables que intervienen en la asignación se les da un nombre para que reflejen su propósito.

```
circun:=2.0*PI*RADIO**2;
```

ahora es mas obvia la información que nos da;

El cálculo de la longitud de la circunferencia de un círculo.

2) Todo programa, segmento de programa o subprograma que tenga un propósito muy específico debe comenzar con comentario que describa en forma clara y consisa ese propósito.

3) No hacer comentarios inútiles, así por ejemplo

```
(*se incremnta contador*)
 Contador:=contador + 1;
```

en este caso el comentario sale sobrando.

La documentación externa se refiere a toda la información que debe estar asentada en un manual de programación. Esta información consiste en lo siguiente

a) Propósito del programa o subprograma
b) Parámetros de entrada y salida
c) Explicar a grandes rasgos la mecánica del programa
d) Listado del programa
e) Advertancias sobre posibles errores fatales; división por cero, etc.
f) Programas o subprogramas que llamen al programa -- (o subprograma) que se documenta.
g) Programas o subprogramas que llama el programa - - (o subprograma) que se documenta.

4.7 ESTRUCTURAS DE CONTROL EN FORTRAN IV

DECISION

```
      IF(.NOT. (PREDICADO)) GO TO A
                CODIGO-A
A     CONTINUE
```

```
      IF(.NOT.(PREDICADO)) GO TO A
                CODIGO-A
                GO TO B
A     CONTINUE
                CODIGO-B
B     CONTINUE
```

```
      IF(.NOT.(PREDICADO-1)) GO TO A
                CODIGO-A
                GO TO X

A     IF(.NOT.(PREDICADO-2)) GO TO B
                CODIGO-B
                GO TO X

B     IF(.NOT.(PREDICADO-3)) GO TO C
                CODIGO-C
                GO TO X
                  .
                  .
                  .
Y     CONTINUE
                CODIGO Y
X     CONTINUE
```

ITERATIVAS

```
A       IF(.NOT.(PREDICADO)) GO TO B
                CODIGO-A
                GO TO A
B       CONTINUE

A       CONTINUE
                CODIGO-A
        IF(.NOT.(PREDICADO)) GO TO A
```

SALIDA

```
        GO TO A
```

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

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

A N E X O S

ARTICULOS CORRESPONDIENTES A PROGRAMACION ESTRUCTURADA

FIS. RAYMUNDO HUGO RANGEL GUTIERREZ

MAYO, 1985

Programming Considered as a Human Activity

Introduction

By way of introduction, I should like to start this talk with a story and a quotation.

The story is about the physicist Ludwig Boltzmann, who was willing to reach his goals by lengthy computations. Once somebody complained about the ugliness of his methods, upon which complaint Boltzmann defended his way of working by stating that "elegance was the concern of tailors and shoemakers," implying that he refused to be troubled by it.

In contrast I should like to quote another famous nineteenth century scientist, George Boole. In the middle of his book, *An Investigation of the Laws of Thought*, in a chapter titled "Of the Conditions of a Perfect Method," he writes: "I do not here speak of that perfection only which consists in power, but of that also which is founded in the conception of what is fit and beautiful. It is probable that a careful analysis of this question would conduct us to some such conclusion as the following, *viz.*, that a perfect method should not only be an efficient one, as respects the accomplishment of the objects for which it is designed, but should in all its parts and processes manifest a certain unity and harmony." A difference in attitude one can hardly fail to notice.

Our unconscious association of elegance with luxury may be one of the origins of the not unusual tacit assumption that it costs to be elegant. To show that it also pays to be elegant is one of my prime purposes. It will give us a clearer understanding of the true nature of the quality of programs and the way in which they are expressed, *viz.*, the programming language. From this insight we shall try to derive some clues as to which programming language features are most desirable. Finally, we hope to convince you that the different aims are less conflicting with one another than they might appear to be at first sight.

On the quality of the results

Even under the assumption of flawlessly working machines we should ask ourselves the questions: "When an automatic computer produces results, why do we trust them, if we do so?" and after that: "What measures can we take to increase our confidence that the results produced are indeed the results intended?"

How urgent the first question is might be illustrated by a simple, be it somewhat simplified, example. Suppose that a mathematician interested in number theory has at his disposal a machine with a program to factorize numbers. This process may end in one of two ways: either it gives a factorization of the number given or it answers that the number given is prime. Suppose now that our mathematician wishes to subject to this process a, say, 20 decimal number, while he has strong reasons to suppose that it is a prime number. If the machine confirms this expectation, he will be happy; if it finds a factorization, the mathematician may be disappointed because his intuition has fooled him again, but, when doubtful, he can take a desk machine and can multiply the factors produced in order to check whether the product reproduces the original number. The situation is drastically changed, however, if he expects the number given to be nonprime: if the machine now produces factors he finds his expectations confirmed and moreover he can check the result by multiplying. If, however, the machine comes back with the answer that the number given is, contrary to his expectations and warmest wishes, alas, a prime number, why on earth should he believe this?

Our example shows that even in completely discrete problems the computation of a result is not a well-defined job, well-defined in the sense that one can say: "I have done it," without paying attention to the convincing power of the result, *viz.*, to its *quality*.

The programmer's situation is closely analogous to that of the pure mathematician, who develops a theory and proves results. For a long time pure mathematicians have thought — and some of them still think — that a theorem can be proved completely, that the question whether a supposed proof for a theorem is sufficient or not, admits an absolute answer "yes" or "no." But this is an illusion, for as soon as one thinks that one has proved something, one has still the duty to prove that the first proof was flawless, and so on, ad infinitum!

One can never guarantee that a proof is correct; the best one can say "I have not discovered any mistakes." We sometimes flatter ourselves with the idea of giving watertight proofs, but in fact we do nothing but make the correctness of our conclusions plausible. So extremely plausible, that the analogy may serve as a great source of inspiration.

In spite of all its deficiencies, mathematical reasoning presents an outstanding model of how to grasp extremely complicated structures with a brain of limited capacity. And it seems worthwhile to investigate to what extent these proven methods can be transplanted to the art of computer usage. In the design of programming languages one can let oneself be guided primarily by considering "what the machine can do." Considering, however, that the programming language is the bridge between the user and the machine — that it can, in fact, be regarded as his tool — it seems just as important to take into consideration "what Man can think." It is in this vein that we shall continue our investigations.

On the structure of convincing programs

The technique of mastering complexity has been known since ancient times: *Divide et impera* (Divide and rule). The analogy between proof construction and program construction is, again, striking. In both cases the available starting points are given (axioms and existing theory versus primitives and available library programs); in both cases the goal is given (the theorem to be proved versus the desired performance); in both cases the complexity is tackled by division into parts (lemmas versus subprograms and procedures).

I assume the programmer's genius matches the difficulty of his problem and assume that he has arrived at a suitable subdivision of the task. He then proceeds in the usual manner in the following stages:

- he makes the complete specifications of the individual parts
- he satisfies himself that the total problem is solved provided he had at his disposal program parts meeting the various specifications
- he constructs the individual parts, satisfying the specifications, but independent of one another and the further context in which they will be used.

Obviously, the construction of such an individual part may again be a task of such a complexity, that inside this part of the job, a further subdivision is required.

Some people might think the dissection technique just sketched a rather indirect and tortuous way of reaching one's goals. My own feelings are perhaps best described by saying that I am perfectly aware that there is no Royal Road to Mathematics, in other words, that I have only a very small head and must live with it. I, therefore, see the dissection technique as one of the rather basic patterns of human understanding and think it worthwhile to try to create circumstances in which it can be most fruitfully applied.

The assumption that the programmer had made a suitable subdivision finds its reflection in the possibility to perform the first two stages: the specification of the parts and the verification that they together do the job. Here elegance, accuracy, clarity and a thorough understanding of the problem at hand are prerequisite. But the whole dissection technique relies on something less outspoken, *viz.*, on what I should like to call "The principle of non-interference." In the second stage above it is assumed that the correct working of the whole can be established by taking, of the parts, into account their exterior specification only, and not the particulars of their interior construction. In the third stage the principle of non-interference pops up again: here it is assumed that the individual parts can be conceived and constructed independently from one another.

This is perhaps the moment to mention that, provided I interpret the signs of current attitudes towards the problems of language definition correctly, in some more formalistic approaches the soundness of the dissection technique is made subject to doubt. Their promoters argue as follows: whenever you give of a mechanism such a two-stage definition, first, what it should do, *viz.*, its specifications, and secondly, how it works, you have, at best, said the same thing twice, but in all probability you have contradicted yourself. And statistically speaking, I am sorry to say, this last remark is a strong point. The only clean way towards language definition, they argue, is by just defining the mechanisms, because what they then will do will follow from this. My question: "How does this follow?" is wisely left unanswered and I am afraid that their neglect of the subtle, but sometimes formidable difference between the concepts *defined* and *known* will make their efforts an intellectual exercise leading into another blind alley.

After this excursion we return to programming itself. Everybody familiar with ALGOL 60 will agree that its procedure concept satisfies to a fair degree our requirements of non-interference, both in its static properties, e.g., in the freedom in the choice of local identifiers, as in its dynamic properties, e.g., the possibility to call a procedure, directly or indirectly, from within itself.

Another striking example of increase of clarity through non-interference, guaranteed by structure, is presented by all programming languages in which algebraic expressions are allowed. Evaluation of such expressions with a sequential machine having an arithmetic unit of limited complexity will imply the use of temporary store for the intermediate results. Their anonymity in the source language guarantees the impossibility that one of them will inadvertently be destroyed before it is used, as would have been possible if the computational process were described in a von Neumann type machine code.

A comparison of some alternatives

A broad comparison between a von Neumann type machine code — well known for its lack of clarity — and different types of algorithmic languages may not be out of order.

In all cases the execution of a program consists of a repeated confrontation of two information streams, the one (say *the program*) constant in time, the other (say *the data*) varying. For many years it has been thought one of the essential virtues of the von Neumann type code that a program could modify its own instructions. In the meantime we have discovered that exactly this facility is to a great extent responsible for the lack of clarity in machine code programs. Simultaneously its indispensability has been questioned: all algebraic compilers I know produce an object program that remains constant during its entire execution phase.

This observation brings us to consider the status of the variable information. Let us first confine our attention to programming languages without assignment statements and without goto statements. Provided that the spectrum of admissible function values is sufficiently broad and the concept of the conditional expression is among the available primitives, one can write the output of every program as the value of a big (recursive) function. For a sequential machine this can be translated into a constant object program, in which at run time a stack is used to keep track of the current hierarchy of calls and the values of the actual parameters supplied at these calls.

Despite its elegance a serious objection can be made against such a programming language. Here the information in the stack can be viewed as objects with nested lifetimes and with a constant value during their entire lifetime. Nowhere (except in the implicit increase of the order counter which embodies the progress of time) is the value of an already existing named object replaced by another value. As a result the only way to store a newly formed result is by putting it on top of the stack; we have no way of expressing that an earlier value now becomes obsolete and the latter's lifetime will be prolonged, although void of interest. Summing up: it is elegant but inadequate. A second objection — which is probably a direct consequence of the first one — is that such programs become after a certain, quickly attained degree of nesting, terribly hard to read.

The usual remedy is the combined introduction of the goto statement and the assignment statement. The goto statement enables us with a backward jump to repeat a piece of program, while the assignment statement can create the necessary difference in status between the successive repetitions.

But I have reasons to ask, whether the goto statement as a remedy is not worse than the defect it aimed to cure. For instance, two programming department managers from different countries and different backgrounds — the one mainly scientific, the other mainly commercial — have communicated to me, independently of each other and on their own initiative, their observation that the quality of their programmers was inversely proportional to the density of goto statements in their programs. This has been an incentive to try to do away with the goto statement.

The idea is, that what we know as *transfer of control*, i.e., replacement of the order counter value, is an operation usually implied as part of more powerful notions: I mention the transition to the next statement, the procedure call and return, the conditional clauses and the for statement; and it is the question whether the programmer is not rather led astray by giving him separate control over it.

I have done various programming experiments and compared the ALGOL text with the text I got in modified versions of ALGOL 60 in which the goto statement was abolished and the for statement — being pompous and overelaborate — was replaced by a primitive repetition clause. The latter versions were more difficult to make: we are so familiar with the jump order that it requires some effort to forget it! In all cases tried, however, the program without the goto statements turned out to be shorter and more lucid.

The origin of the increase in clarity is quite understandable. As is well known there exists no algorithm to decide whether a given program ends or not. In other words, each programmer who wants to produce a flawless program must at least convince himself by inspection that his program will indeed terminate. In a program in which unrestricted use of the goto statement has been made, this analysis may be very hard on account of the great variety of ways in which the program may fail to stop. After the abolishment of the goto statement there are only two ways in which a program may fail to stop: either by infinite recursion, i.e., through the procedure mechanism, or by the repetition clause. This simplifies the inspection greatly.

The notion of repetition, so fundamental in programming, has a further consequence. It is not unusual that inside a sequence of statements to be repeated one or more subexpressions occur, which do not change their value during the repetition. If such a sequence is to be repeated many times, it would be a regrettable waste of time if the machine had to recompute these same values over and over again. One way out of this is to delegate to the now optimizing translator the discovery of such constant sub-expressions in order that it can take the computation of their values outside the loop. Without an optimizing translator the obvious solution is to invite the programmer to be somewhat more explicit and he can do so by introducing as many additional variables as there are constant sub-expressions within the repetition and by assigning the values to them before entering the repetition. I should like to stress that both ways of writing the program are equally misleading. In the first case the translator is faced with the unnecessary puzzle to discover the constancy; in the second case we have introduced a variable, the only function of which is to denote a constant value. This last observation shows the way out of the

difficulty: besides variables the programmer would be served by *local constants*, i.e., identifiable quantities with a finite lifetime, during which they will have a constant value, that has been defined at the moment of introduction of the quantity. Such quantities are not new: the formal parameters of procedures already display this property. The above is a plea to recognize that the concept of the *local constant* has its own right of existence. If I am well informed, this has already been recognized in CPL, the programming language designed in a joint effort around the Mathematical Laboratory of the University of Cambridge, England.

The double gain of clarity

I have discussed at length that the convincing power of the results is greatly dependent on the clarity of the program, on the degree in which it reflects the structure of the process to be performed. For those who feel themselves mostly concerned with efficiency as measured in the cruder units of storage space and machine time, I should like to point out that increase of efficiency always comes down to exploitation of structure and for them I should like to stress that all structural properties mentioned can be used to increase the efficiency of an implementation. I shall review them briefly.

The lifetime relation satisfied by the local quantities of procedures allows us to allocate them in a stack, thus making very efficient use of available store; the anonymity of the intermediate results enables us to minimize storage references dynamically with the aid of an automatically controlled set of push down accumulators; the constancy of program text under execution is of great help in machines with different storage levels and reduces the complexity of advanced control considerably; the repetition clause eases the dynamic detection of endless looping and finally, the local constant is a successful candidate for a write-slow-read-fast store, when available.

Conclusion

When I became acquainted with the notion of algorithmic languages I never challenged the then prevailing opinion that the problems of language design and implementation were mostly a question of compromises: every new convenience for the user had to be paid for by the implementation, either in the form of increased trouble during translation, or during execution or during both. Well, we are most certainly not living in Heaven and I am not going to deny the possibility of a conflict between convenience and efficiency, but now I do protest when this conflict is presented as a complete summing up of the situation. I am of the opinion that it is worthwhile to investigate to what extent the needs of Man and Machine go hand in hand and to see what techniques we can devise for the benefit of all of us. I trust that this investigation will bear fruits and if this talk made some of you share this fervent hope, it has achieved its aim.

Go To Statement Considered Harmful

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to

shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of go to statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (if *B* then *A*), alternative clauses (if *B* then *A1* else *A2*), choice clauses as introduced by C.A.R. Hoare (case [i] of (*A1*, *A2*, . . . , *An*)), or conditional expressions as introduced by J. McCarthy (*B1* → *E1*, *B2* → *E2*, . . . , *Bn* → *En*), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, while *B* repeat *A* or repeat *A* until *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is — and this seems to be inherent to sequential processes — that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well-chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate-system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the go to statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the go to statement is far from new. I remember having read the explicit recommendation to restrict the use of the go to statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C.A.R. Hoare. In [1, Sec. 3.2.1.] Wirth and

Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than go to statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the go to statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

References

1. N. Wirth and C.A.R. Hoare, "A Contribution to the Development of ALGOL," *Communications of the ACM*, Vol. 9, No. 6 (June 1966), pp. 413-32.
2. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71.

Structured Programming

Introduction

This working document reports on experience and insights gained in programming experiments performed by the author in the last year. The leading question was if it was conceivable to increase our programming ability by an order of magnitude and what techniques (mental, organizational or mechanical) could be applied in the process of program composition to produce this increase. The programming experiments were undertaken to shed light upon these questions.

Program size

My real concern is with intrinsically large programs. By "intrinsically large" I mean programs that are large due to the complexity of their task, in contrast to programs that have exploded (by inadequacy of the equipment, unhappy decisions, poor understanding of the problem, etc.). The fact that, for practical reasons, my experiments had thus far to be carried out with rather small programs did present a serious difficulty; I have tried to overcome this by treating problems of size explicitly and by trying to find their consequences as much as possible by analysis, inspection and reflection rather than by (as yet too expensive) experiments.

In doing so I found a number of subgoals that, apparently, we have to learn to achieve (if we don't already know how to do that).

If a large program is a composition of N "program components," the confidence level of the individual components must be exceptionally high if N is very large. If the individual components can be made with the probability "p" of being correct, the probability that the whole program functions properly will not exceed

$$P = p^N$$

for large N, p must be practically equal to one if P is to differ significantly from zero. Combining subsets into larger components from which then the whole program is composed, presents no remedy:

$$p^{N/2} * p^{N/2} \text{ still equals } p^N !$$

As a consequence, the problem of program correctness (confidence level) was one of my primary concerns.

The effort — be it intellectual or experimental — needed to demonstrate the correctness of a program in a sufficiently convincing manner may (measured in some loose sense) not grow more rapidly than in proportion to the program length (measured in an equally loose sense). If, for instance, the labour involved in verifying the correct composition of a whole program out of N program components (each of them individually assumed to be correct) still grows exponentially with N, we had better admit defeat.

Any large program will exist during its life-time in a multitude of different versions, so that in composing a large program we are not so much concerned with a single program, but with a whole family of related programs, containing alternative programs for the same job and/or similar programs for similar jobs. A program therefore should be conceived and understood as a member of a family; it should be so structured out of components that various members of this family, sharing components, do not only share the correctness demonstration of the shared components but also of the shared substructure.

Program correctness

An assertion of program correctness is an assertion about the net effects of the computations that may be evoked by this program. Investigating how such assertions can be justified, I came to the following conclusions:

1. The number of different inputs, i.e. the number of different computations for which the assertions claim to hold is so fantastically high that demonstration of correctness by sampling is completely out of the question. *Program testing can be used to show the presence of bugs, but never to show their absence!*

   Therefore, proof of program correctness should depend only upon the program text.

2. A number of people have shown that program correctness can be proved. Highly formal correctness proofs have been given; also correctness proofs have been given for "normal programs," i.e. programs not written with a proof procedure in mind. As is to be expected (and nobody is to be blamed for that) the circulating examples are concerned with rather small programs and, unless measures are taken, the amount of labour involved in proving might well (will) explode with program size.

3. Therefore, I have not focused my attention on the question "how do we prove the correctness of a given program?" but on the questions "for what program structures can we give correctness proofs without undue labour, even if the programs get large?" and, as a sequel, "how do we make, for a given task, such a well-structured program?" My willingness to confine my attention to such "well-structured programs" (as a subset of the set of all possible programs) is based on my belief that we can find such a well-structured subset satisfying our programming needs, i.e. that for each programmable task this subset contains enough realistic programs.

4. This, what I call "constructive approach to the problem of program correctness," can be taken a step further. It is not restricted to general considerations as to what program structures are attractive from the point of view of provability, in a number of specific, very difficult programming tasks I have finally succeeded in constructing a program by analyzing how a proof could be given that a class of computations would satisfy certain requirements; from the requirements of the proof the program followed.

The relation between program and computation

Investigating how assertions about the possible computations (evolving in time) can be made on account of the static program text, I have concluded that adherence to rigid sequencing disciplines is essential, so as to allow step-wise abstraction from the possibly different routings. In particular: when programs for a sequential computer are expressed as a linear sequence of basic symbols of a programming language, sequencing should be controlled by alternative, conditional and repetitive clauses and procedure calls, rather than by statements transferring control to labelled points.

The need for step-wise abstraction from local sequencing is perhaps most convincingly shown by the following demonstration:

Let us consider a "stretched" program of the form

$$S_1; S_2; \ldots; S_N \qquad (1)$$

and let us introduce the measuring convention that when the net effect of the execution of each individual statement S_i has been given, it takes N steps of reasoning to establish the correctness of program (1), i.e. to establish that the cumulative net effect of the N actions in succession satisfies the requirements imposed upon the computations evoked by program (1).

For a statement of the form

$$\text{if } B \text{ then } S_1 \text{ else } S_2 \qquad (2)$$

where, again, the net effect of the execution of the constituent statements S_1 and S_2 has been given; we introduce the measuring convention that it takes 2 steps of reasoning to establish the net effect of program (2), viz. one for the case B and one for the case not B.

Consider now a program of the form

$$\begin{array}{l} \text{if } B_1 \text{ then } S_{11} \text{ else } S_{12}; \\ \text{if } B_2 \text{ then } S_{21} \text{ else } S_{22}; \\ \qquad \ldots \\ \text{if } B_N \text{ then } S_{N1} \text{ else } S_{N2} \end{array} \qquad (3)$$

According to the measuring convention it takes 2 steps per alternative statement to understand it, i.e. to establish that the net effect of

$$\text{if } B_i \text{ then } S_{i1} \text{ else } S_{i2}$$

is equivalent to that of the execution of an abstract statement S_i. Having N such alternative statements, it takes us 2N steps to reduce program (3) to one of the form of program (1); to understand the latter form of the program takes us another N steps, giving 3N steps in toto.

If we had refused to introduce the abstract statements S_i but had tried to understand program (3) directly in terms of executions of the statements S_{ij}, each such computation would be the cumulative effect of N such statement executions and would as such require N steps to understand it. Trying to understand the algorithm in terms of the S_{ij} implies that we have to distinguish between 2^N different routings through the program and this would lead to $N*2^N$ steps of reasoning!

I trust that the above calculation convincingly demonstrates the need for the introduction of the abstract statements S_i. An aspect of my constructive approach is not to reduce a given program (3) to an abstract program (1), but to start with the latter.

Abstract data structures

Understanding a program composed from a modest number of abstract statements again becomes an exploding task if the definition of the net effect of the constituent statements is sufficiently unwieldy. This can be overcome by the introduction of suitable abstract data structures. The situation is greatly analogous to the way in which we can understand an ALGOL program operating on integers without having to bother about the number representation of the implementation used. The only difference is that now the programmer must invent his own concepts (analogous to the "ready-made" integer) and his own operations upon them (analogous to the "ready-made" arithmetic operations).

In the refinement of an abstract program (i.e. composed from abstract statements operating on abstract data structures) we observe the phenomenon of "joint refinement." For abstract data structures of a given type a certain representation is chosen in terms of new (perhaps still rather abstract) data structures. The immediate consequence of this design decision is that the abstract statements operating upon the original abstract data structure have to be redefined in terms of algorithmic refinements operating upon the new data structures in terms of which it was decided to represent the original abstract data structure. Such a joint refinement of data structure and associated statements should be an isolated unit of the program text: it embodies the immediate consequences of an (independent) design decision and is as such the natural unit of interchange for program modification. It is an example of what I have grown into calling "a pearl."

Programs as necklaces strung from pearls

I have grown to regard a program as an ordered set of pearls, a "necklace." The top pearl describes the program in its most abstract form, in all lower pearls one or more concepts used above are explained (refined) in terms of concepts to be explained (refined) in pearls below it, while the bottom pearl eventually explains what still has to be explained in terms of a standard interface (=machine). The pearl seems to be a natural program module.

As each pearl embodies a specific design decision (or, as the case may be, a specific aspect of the original problem statement) it is the natural unit of interchange in program modification (or, as the case may be, program adaptation to a change in problem statement).

Pearls and necklace give a clear status to an "incomplete program," consisting of the top half of a necklace; it can be regarded as a complete program to be executed by a suitable machine (of which the bottom half of the necklace gives a feasible implementation). As such, the correctness of the upper half of the necklace can be established regardless of the choice of the bottom half.

Between two successive pearls we can make a "cut," which is a manual for a machine provided by the part of the necklace below the cut and used by the program represented by the part of the necklace above the cut. This manual serves as an interface between the two parts of the necklace. We feel this form of interface more helpful than regarding data representation as an interface between operations, in particular more helpful towards ensuring the combinatorial freedom required for program adaptation.

The combinatorial freedom just mentioned seems to be the only way in which we can make a program as part of a family or "in many (potential) versions" without the labour involved increasing proportional to the number of members of the family. The family becomes the set of those selections from a given collection of pearls that can be strung into a fitting necklace.

Concluding remarks

Pearls in a necklace have a strict logical order, say "from top to bottom." I would like to stress that this order may be radically different from the order (in time) in which they are designed.

Pearls have emerged as program modules when I tried to map upon each other as completely as possible, the numerous members of a class of related programs. The abstraction process involved in this mapping turns out (not, amazingly, as an afterthought!) to be the same as the one that can be used to reduce the amount of intellectual labour involved in correctness proofs. This is very encouraging.

As I said before, the programming experiments have been carried out with relatively small programs. Although, personally, I firmly believe that they show the way towards more reliable composition of really large programs, I should like to stress that as yet I have *no* experimental evidence for this. The experimental evidence gained so far shows an increasing ability to compose programs of the size I tried. Although I tried to do it, I feel that I have given but little recognition to the requirements of program development such as is needed when one wishes to employ a large crowd; I have no experience with the Chinese Army approach, nor am I convinced of its virtues.

The Translation of 'go to' Programs to 'while' Programs

1. GENERAL DISCUSSION

1.1. Introduction

The first class of programs we consider are simple *flowchart programs* constructed from assignment statements (that assign terms to variables) and test statements (that test quantifier-free formulas) operating on a "state vector" $\bar{x}$. The flowchart program begins with a unique start statement of the form

where $\bar{x}_{input}$ is a subvector of $\bar{x}$, indicating the variables that have to be given values at the beginning of the computation. It ends with a unique halt statement of the form

where $\bar{x}_{output}$ is a subvector of $\bar{x}$, indicating the variables whose values will be the desired result of the computation.

We make no assumptions about the domain of individuals, or about the operations and predicates used in the statements. Thus our flowchart programs are really flowchart schemas (see, for example, Luckham, Park and Paterson [1]) and all the results can be stated in terms of such schemas.

Let P_1 be any flowchart program of the form shown in Figure 1. Note that, for example, the statement $\bar{x} \leftarrow e(\bar{x})$ stands for any sequence of assignment statements whose net effect is the replacement of vector $\bar{x}$ by a new vector $e(\bar{x})$. Similarly, the test $p(\bar{x})$, for example, stands for any quantifier-free formula with variables from $\bar{x}$. The flowchart program P_1 will be used as an example throughout the paper.

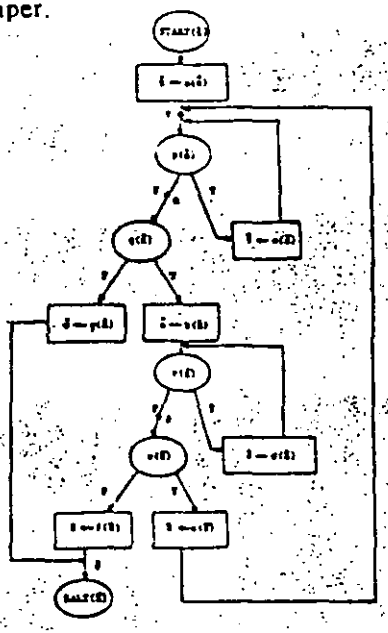


Figure 1. The flowchart program P_1.

In order to write such flowchart programs in a conventional programming language, *goto* statements are required. There has recently been much discussion (see, for example, Dijkstra [2]) about whether the use of *goto* statements makes programs difficult to understand, and whether the use of *while* or *for* statements is preferable. In addition, it is quite possible that simpler proof methods of the validity and equivalence of programs may be found for programs without *goto* statements (see, for example, Stark [3]). It is clearly relevant to this discussion to consider whether the abolition of *goto* statements is really possible.

Therefore the second class of programs we consider are *while programs*, i.e., Algol-like programs consisting only of *while* statements of the form *while* (quantifier-free formula) *do* (statement), in addition to conditional, assignment and block* statements. As before, each program starts with a unique start statement, START($\bar{x}_{input}$), and ends with a unique halt statement, HALT($\bar{x}_{output}$).

*A block statement is denoted by any sequence of statements enclosed by square brackets

Since both classes of programs use the same kind of start and halt statements, we can define the equivalence of two programs independently of the classes to which they belong. Two programs (with the same input subvectors $\bar{x}_{input}$ and the same output subvectors $\bar{x}_{output}$) are said to be *equivalent* if for each assignment of values to $\bar{x}_{input}$ either both programs do not terminate or both terminate with the same values in $\bar{x}_{output}$.

1.2. Translation to while programs by adding variables

1.2.1. Extending the state vector $\bar{x}$

We can show that by allowing extra variables which keep crucial past values of some of the variables in $\bar{x}$, one can effectively translate every flowchart program into an equivalent while program (ALGORITHM 1). The importance of this result is that original "topology" of the program is preserved, and the new program is of the same order of efficiency as the original program. However, we shall not enter into any discussion as to whether the new program is superior to the original one or not.

This result, considered in terms of schemas, can be contrasted with those of Paterson and Hewitt [4] (see also Strong [5]). They showed that although it is not possible to translate all recursive schemas into flowchart schemas, it is possible to do this for "linear" recursive schemas, by adding extra variables. However, as they point out, the flowchart schemas produced are less efficient than the original recursive schemas.

As an example, ALGORITHM 1 will give the following while program which is equivalent to the flowchart program P_1 (Figure 1):

```
START(x̄);
x̄ ← a(x̄);
[while p(x̄) do x̄ ← e(x̄);
 ȳ ← x̄;
 if q(x̄) then [x̄ ← b(x̄); while r(x̄) do x̄ ← d(x̄)];
while q(ȳ) ∧ s(x̄) do
   [x̄ ← c(x̄);
    [while p(x̄) do x̄ ← e(x̄);
     ȳ ← x̄;
     if q(x̄) then
        [x̄ ← b(x̄); while r(x̄) do x̄ ← d(x̄)]]];
if q(ȳ) then x̄ ← f(x̄) else x̄ ← g(x̄);
HALT(x̄).
```

If the test $q(\bar{x})$ uses only a subvector of $\bar{x}$, then the algorithm will indicate that the vector of extra variables $\bar{y}$ need only be of the same length as this subvector.

Note that on each cycle of the main while statement, the state vector $\bar{x}$ is at point β, while $\bar{y}$ holds the preceding values of $\bar{x}$ at point α.

Note also that the two subprograms enclosed in broken lines are identical. This is typical of the programs produced by the algorithm. One might use this fact to make the programs more readable by using "subroutines" for the repeated subprograms.

Because of space limitations we cannot present ALGORITHM I in this paper. The detailed algorithm can be found in the preliminary report of this paper (CS 188, Computer Science Dept., Stanford University).

1.2.2. *Adding boolean variables*

The translation of flowchart programs into while programs by the addition of boolean variables is not a new idea. Böhm and Jacopini [6] and Cooper [7] (see also Bruno and Steiglitz [8]) have shown that every flowchart program can be effectively translated into an equivalent while program (with one while statement) by introducing new boolean variables into the program, new predicates to test these variables, together with assignments to set them *true* or *false*. The boolean variables essentially simulate a program counter, and the while program simply interprets the original program. On each repetition of the while statement, the next operation of the original program is performed, and the "program counter" is updated. As noted by Cooper and Bruno and Steiglitz themselves, this transformation is undesirable since it changes the "topology" (loop-structure) of the program, giving a program that is less easy to understand. For example, if a while program is written as a flowchart program and then transformed back to an equivalent while program by their method, the resulting while program will not resemble the original.

We give an algorithm (ALGORITHM II) for transforming flowchart programs to equivalent while programs by adding extra boolean variables, which is an improvement on the above method. It preserves the "topology" of the original program and in particular it does not alter while-like structure that may already exist in the original program.

For the flowchart program P_1 (Figure 1), for example ALGORITHM II will produce the following while program.

```
START(x̄);
x̄ ← a(x̄);
t ← true;
while t do
    [while p(x̄) do x ← e(x̄);
    if q(x̄) then [x̄ ← b(x̄);
        while r(x̄) do x̄ ← d(x̄);
        if s(x̄) then x̄ ← e(x̄)
            else [x̄ ← f(x̄); t ← false]]
        else [x̄ ← g(x̄); t ← false]];
HALT(x̄).
```

Note that each repetition of the main while statement starts from point γ and proceeds either back to γ or to δ. In the latter case, t is made *false* and we subsequently exit from the while statement.

1.3. Translation to while programs without adding variables

It is natural at this point to consider whether every flowchart program can be translated into an equivalent while program without adding extra variables (i.e., using only the original state vector $\bar{x}$). We show that this cannot be done in general, and in fact there is a flowchart program of the form of Figure 1 which is an appropriate counter-example.

A similar negative result has been demonstrated by Knuth and Floyd [9] and Scott [private communication]. However, the notion of equivalence considered by those authors is more restrictive in that it requires equivalence of computation sequences (i.e., the sequence of assignment and test statements in order of execution) and not just the equivalence of final results of computation as we do. Thus, since our notion of equivalence is weaker, our negative result is stronger.

2. ALGORITHM II: TRANSLATION BY ADDING BOOLEAN VARIABLES

The second algorithm, ALGORITHM II, translates flowchart programs to equivalent while programs by adding boolean variables. It makes use of the fact that every flowchart program (without the start and halt statements) can be decomposed into blocks where a block is any piece of flowchart program with only one exit (but possibly many entrances). This is obvious since in particular the whole body of the given flowchart program can be considered as such a block. The aim, whenever possible, is to get blocks containing at most one top-level test statement (i.e., test statement not contained in inner blocks) since such blocks can be represented as a piece of while program without adding boolean variables. In particular, if a while program is expressed as a flowchart program, this latter program can always be decomposed into such simple blocks, and the algorithm will give us back the original while program.

For any given flowchart program we construct the equivalent while program by induction on the structure of the blocks.

For each entrance b_i to block B we consider that part B_i of the block reachable from b_i. We then recursively construct an equivalent piece of while program $\tilde{B}_i(\bar{x}, \bar{t})^*$ as follows. There are two cases to consider:

Case 1: (a) B_i contains at most one top-level test statement.
or (b) B_i contains no top-level loops.

*$\bar{t}$ is a (possibly empty) vector of additional boolean variables introduced by the translation.

In both cases $\bar{B}_i(x, t)$ is the obvious piece of while program requiring at most one top-level while statement (and no extra boolean variables).

Case 2: B_i contains two or more top-level test statements and at least one top-level loop.

In this case we choose a set of points on top-level arcs of B_i (called "cut-set" points) such that each top-level loop contains at least one such point. One point on the exit arc of the block is also included in this set. We shall translate B_i into a piece of while program $\bar{B}_i(\bar{x}, \bar{t})$ with one top-level while statement in such a way that each iteration of the while statement follows the execution of B_i from one cut-set point to the next. In this case, $\bar{B}_i(\bar{x}, \bar{t})$ includes boolean variables introduced to keep track of the current cut-set point. Note that n boolean variables $t_1, t_2, \ldots, t_n$ are sufficient to distinguish between k cut-set points, $2^{n-1} < k \leq 2^n$.

Example: We shall illustrate the method using again the flowchart program P_1 (Figure 1). We decompose P_1 into blocks as shown in Figure 2. Blocks B_1 and B_2 are of type 1 and can each be written as a single while statement. Block B_3 is of type 2 with a single top-level loop. Thus it is sufficient to choose points α and β as the cut-set points. To distinguish between α and β we need one boolean variable, t say. We can therefore generate the while program given on the preceding page which is equivalent to the flowchart program P_1.

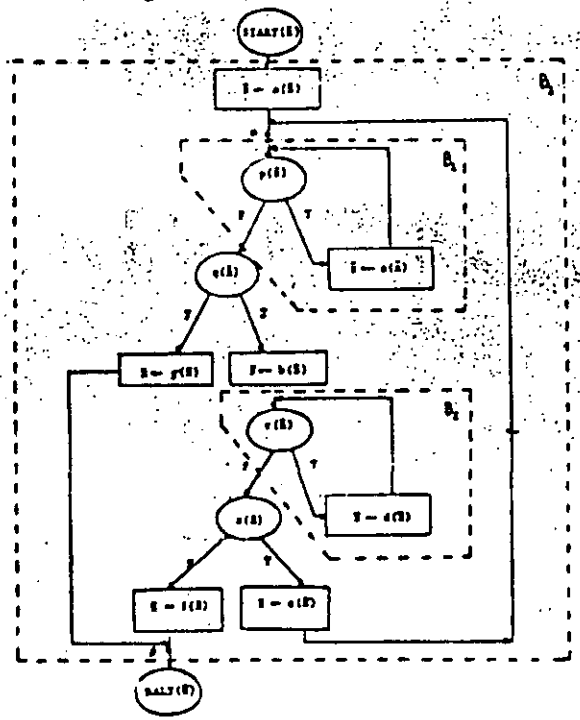
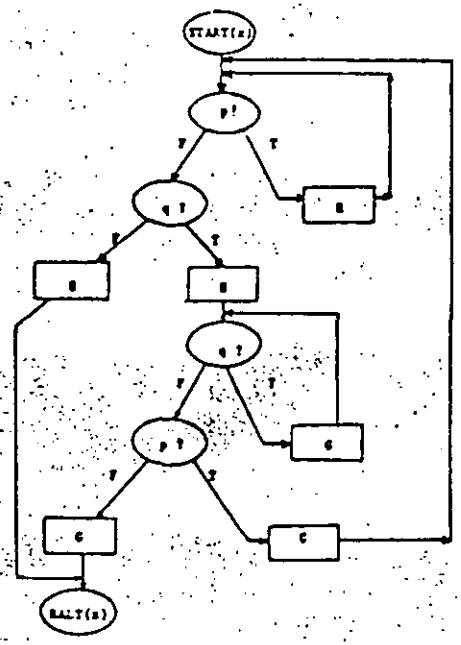


Figure 2. The flowchart program P_1 (for ALGORITHM II).

3. THE NEGATIVE RESULT

We consider the flowchart program P_2 (Figure 3) which has the structure of Figure 1. The domain D is the set of all pairs of strings such that the first string, called "head," is any *finite* string over letters $\{g, h\}$, and the second string, called "tail," is any *infinite*[*] string over letters $\{\alpha, \beta, \gamma\}$, with at most one occurrence of γ.



Where test p means "is 'α' the leftmost letter in tail"; test q means "is 'β' the leftmost letter in tail"; operation G means "erase the leftmost letter in tail and add 'g' on the right of head"; and operation H means "erase the leftmost letter in tail and add 'h' on the right of head."

Figure 3. The flowchart program P_2.

During a computation of P_2, the only changes in the value of the program variable are deletion of leftmost letters from the tail and adding letters g or h to the right of the head. The tests in the program simply look at the tail, and therefore the computation is determined by the tail of the initial value. Thus, since the program terminates if and only if both tests p and q are false, it implies that P_2 terminates if and only if the tail of the initial value contains γ. Another important feature of any computation of P_2 is that whenever the leftmost letter of the tail is α, the next but one operation must be operation H. Similarly, whenever the leftmost letter is β, the next but one operation must be G.

[*] Note that the domain is non-enumerable. However, we can in fact restrict ourselves to the enumerable domain of ultimately periodic strings, i.e., infinite strings which eventually repeat some finite substring indefinitely.

Let us assume that we have a while program P'_2 equivalent to P_2 which also has one variable and the same domain D. Although we could allow the assignment statements of P'_2 to use any terms obtained by compositions of the operations G and H, we assume without loss of generality that each assignment statement in P'_2 consists of a single operation G or H. The tests in the conditional and while statements may only use quantifier-free formulas obtained from tests p and q, and operations G and H. Since we use only one variable, it follows that the sequences of values describing corresponding computations of P'_2 and P_2 are identical. Note also that since there is a bound on the depth of terms in the quantifier-free formulas, there is a bound, M say, on the number of leftmost letters in the tail that can affect the decision of any test in P'_2. Finally, without loss of generality we shall make the restriction that there is no redundant while statement in P'_2; i.e., there is no while statement with a uniform bound on the number of its iterations in terminating computations.

Since P'_2 must contain some (non-redundant) while statement, let W be any while statement in P'_2 which is not contained or followed by another while statement. The point in P'_2 immediately after W we shall denote by A.

Lemma: For all n ($n \geq 0$) there exist strings $a, c \in \{\alpha, \beta\}^*$ and $d \in \{\alpha, \beta\}^\infty$ ($|c| = n$)[1] such that for all strings $b \in \{\alpha, \beta\}^*$ the computation of P'_2 starting with tail $abc\gamma d$ passes A with some tail $\underline{ab}c\gamma d$, where $\underline{ab}$ is some rightmost substring of ab (possibly empty).

From this lemma we immediately obtain the following corollary.

Corollary. For every n, $n \geq 0$, there exists a finite computation of P'_2 which passes through A with more than n operations still to be performed.

But this contradicts the fact that since there is no while statement following A, the number of operations that P'_2 can perform after A is bounded.

Proof of Lemma: By induction on n.

Base step. Choose any computation starting with tail $a'a''b'\gamma d'$ ($a', a'', b' \in \{\alpha, \beta\}^*$, $d' \in \{\alpha, \beta\}^\infty$ and $|a''| = M$) that enters W with tail $a''b'\gamma d'$. (Such computation exists by non-redundancy of W.)

Since at most M leftmost letters of the tail can effect the decision of any test, on entering W the main test can only look at a''. Therefore the test will be true for any tail starting with a''.

[1] i.e., a and c are finite strings (possibly empty) over $\{\alpha, \beta\}$, d is an infinite string over $\{\alpha, \beta\}$ and the length of c is n.

In particular, the computation starting with tail $a'a''b\gamma a''d'$, *for any* b $\{\alpha, \beta\}^*$, also enters W at the same point, i.e., with tail $a''b\gamma a''d'$. Since the computation is finite, it must subsequently pass point A, but (noting that the test in W must be false when passing A) it cannot pass A with tail $a''d'$.

Hence, with $a = a'a''$, $d = a''d'$, for all strings b in $\{\alpha, \beta\}^*$, the computation starting with $ab\gamma d$ must pass A with some tail $\underline{ab}\gamma d$ where $\underline{ab}$ is some rightmost substring of ab.

Induction step. Assume we have strings $a, c \in \{\alpha, \beta\}^*$ and $d \in \{\alpha, \beta\}^\infty$, $|c| = n$, such that for all strings b in $\{\alpha, \beta\}^*$ the computation starting with tail $abc\gamma d$ passes A with some tail $\underline{ab}c\gamma d$ where $\underline{ab}$ is some rightmost substring of ab.

We find a string $c' \in \{\alpha, \beta\}^*$, $|c'| = n+1$, such that for all strings b' in $\{\alpha, \beta\}^*$ the computation starting with tail $ab'c'\gamma d$ passes A with some tail $\underline{ab'}c'\gamma d$ where ab' is some rightmost substring of $\underline{ab'}$.

There are three cases to consider:

(i) For all nonempty strings b, the corresponding substring $\underline{ab}$ is nonempty. In this case *we take* c' *to be* ac.[1]

For any string b' in $\{\alpha, \beta\}^*$ the computation starting with tail $ab'ac\gamma d$, passes A with tail $\underline{ab'}ac\gamma d$, where $\underline{ab'}$ is a rightmost substring of ab'.

(ii) For some nonempty string $b = b''\alpha$ ($b'' \in \{\alpha, \beta\}^*$), the substring $\underline{ab}$ is empty, i.e., there exists computation S starting with $ab''ac\gamma d$ that passes A with $c\gamma d$. In this case *we take* c' *to be* βc.

By earlier remarks about P_2 and P'_2, it follows that the next operation in S after passing A must be H.

Now, for *any string* b' *in* $\{\alpha, \beta\}^*$ the computation starting with tail $ab'\beta c\gamma d$ must pass A with some tail $\underline{ab'\beta}c\gamma d$ where $\underline{ab'\beta}$ is some rightmost substring of $ab'\beta$.

$\underline{ab'\beta}$ cannot be empty because this would mean that this computation passes A with the same tail $c\gamma d$ as for S but in this case the next operation to be performed is G. This is impossible, since the course of computation from A must be determined by the tail at this point.

[1] We could equally well take c' to be βc and consider computations starting with tail $ab'\beta c\gamma d$.

Hence, the computation must pass A with some tail $\underline{ab'}\beta c\gamma d$ (i.e., $\underline{ab'}c'\gamma d$) where ab' is a rightmost substring of ab.

(ii) For some nonempty string $b = b''\beta$ ($b'' \in \{\alpha, \beta\}^*$), the substring $\underline{ab}$ is empty. In this case we take c' to be ac.

We proceed as in case (ii) with α and β interchanged and G and H interchanged.

Acknowledgement

We are indebted to David Cooper for stimulating discussions and mainly for his idea of using cut-set points which we have adopted in ALGORITHM II. We are also grateful to Donald Knuth for his critical reading of the manuscript and subsequent helpful suggestions.

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

References

1. D.C. Luckham, D.M.R. Park, and M.S. Paterson, "On Formalized Computer Programs," *Journal of Computer and System Sciences*, Vol. 4, No. 3 (June 1970), pp. 220-49.
2. E.W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3 (March 1968), pp. 147-48.
3. R. Stark, "A Language for Algorithms," *Computer Journal*, Vol. 14, No. 1 (February 1971), pp. 40-44.
4. M.S. Paterson and C.E. Hewitt, "Comparative Schematology," unpublished memo.
5. H.R. Strong, "Translating Recursion Equations into Flowcharts," *Journal of Computer and System Sciences*, Vol. 5, No. 3 (June 1971), pp. 254-85.
6. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71.
7. D.C. Cooper, "Böhm and Jacopini's Reduction of Flowcharts," *Communications of the ACM*, Vol. 10, No. 8 (August 1967), pp. 463-73.
8. J. Bruno and K. Steiglitz, "The Expression of Algorithms by Charts," *Journal of the ACM*, Vol. 19, No. 3 (July 1972), pp. 517-25.
9. D.E. Knuth and R.W. Floyd, "Notes on Avoiding 'go to' Statements," *Information Processing Letters*, Vol. 1, No. 1 (February 1971), pp. 23-31; see also Stanford University Computer Science Technical Report, Vol. CS148 (Stanford, Calif., January 1970).

(4)

WULF 8

A Case Against the GOTO

Introduction

It has been suggested that the use of the goto construct is undesirable, is bad programming practice, and that at least one measure of the "quality" of a program is inversely related to the number of goto statements contained in it. The rationale behind this suggestion is that it is possible to use the goto in ways which obscure the logical structure of a program, thus making it difficult to understand, modify, debug, and/or prove its correctness. It is quite clear that not all uses of the goto are obscure, but the hypothesis is that these situations fall into one of a small number of cases and therefore explicit and inherently well-structured language constructs may be introduced to handle them. Although the suggestion to ban the goto appears to have been a part of the computing folklore for several years, to this author's knowledge the suggestion was first made in print by Professor E.W. Dijkstra in a letter to the editor of the *Communications of the ACM* in 1968 [1].

In this paper we shall examine the rationale for the elimination of the goto in programming languages, and some of the theoretical and practical implications of its (total) elimination.

Rationale

At one level, the rationale for eliminating the goto has already been given in the introduction. Namely, it is possible to use the goto in a manner which obscures the logical structure of a program to a point where it becomes virtually impossible to understand [1, 3, 4]. It is *not* claimed that *every* use of the goto obscures the logical structure of a program; it is only claimed that it is *possible* to use the goto to fabricate a "rat's nest" of control flow which has the undesirable properties mentioned above. Hence this argument addresses the *use* of the goto rather than the goto itself.

As the basis for a proposal to totally eliminate the goto this argument is somewhat weak. It might reasonably be argued that the undesirable consequences of unrestricted branching may be eliminated by enforcing restrictions on the *use* of the goto rather than eliminating the construct. However, it will be seen that any rational set of restrictions is equivalent to eliminating the construct if an adequate set of other control primitives is provided. The strong reasons for eliminating the goto arise in the context of more positive proposals for a programming methodology which makes the goto unnecessary. It is not the purpose of this paper to explicate these methodologies (variously called "structured programming," "constructive programming," "stepwise refinement," etc.); however, since the major justification for eliminating the goto lies in this work, a few words are in order.

It is, perhaps, pedantic to observe that the present practice of building large programming systems is a mess. Most, if not all, of the major operating systems, compilers, information systems, etc. developed in the last decade have been delivered late, have performed below expectation (at least initially), and have been filled with "bugs." This situation is intolerable, and has prompted several researchers [2, 3, 4, 5, 6, 7, 8, 9] to consider whether a programming methodology might be developed to correct this situation. This work has proceeded from two premises:

1. Dijkstra speaks of our "human inability to do much" (at one time) to point up the necessity of decomposing large systems into smaller, more "human size" chunks. This observation is hardly startling, and in fact, most programming languages include features (modules, subroutines, and macros, for example) to aid in the mechanical aspects of this decomposition. However, the further observation that the particular decomposition chosen makes a significant difference to the understandability, modifiability, etc., of a program and that there is an *a priori* methodology for choosing a "good" decomposition is less expected.

2. Dijkstra has also said that debugging can show the presence of errors, but never their absence. Thus ultimately we will have to be able to prove the correctness of the programs we construct (rather than "debug" them) since their sheer size prohibits exhaustive testing. Although some progress has been made on the automatic proof of the correctness of programs (c.f., [10, 11, 12, 23, 24]), this approach appears to be far from a practical reality. The methodology proposed by Dijkstra (and others) proceeds so that the construction of a program guides a (comparatively) simple and intuitive proof of its correctness.

The methodology of "constructive programming" is quite simple and, in this context, best described by an (partial) example. Let us consider the problem of producing a KWIC* index. Construction of the program proceeds in a series of steps in which each step is a refinement of some portion of a previous step. We start with a single statement of the function to be performed:

Step 1: PRINTKWIC

We may think of this as being an instruction in a language (or machine) in which the notion of generating a KWIC index is primitive. Since this operation is not primitive in most practical languages, we proceed to define it:

Step 2: PRINTKWIC: generate and save all interesting circular shifts
alphabetize the saved lines
print alphabetized lines

Again, we may think of each of these lines as being an instruction in an appropriate language; and again, since they are not primitive in most existing languages, we must define them; for example:

Step 3a: generate and save all interesting circular shifts:

```
for each line in the input do
   begin
   generate and save all interesting
      shifts of "this line"
   end
```

etc.

*For those who may not be familiar with a KWIC (key word in context) index, the following description is adequate for this paper.

A KWIC system accepts a set of *lines*. Each *line* is an ordered set of *words* and each *word* is an ordered set of *characters*. A *word* may be one of a set of *uninteresting* words ("a," "the," "of," etc.), otherwise it is a key word. Any line may be circularly shifted by removing its first word and placing it at the end of the line. The KWIC index system generates an ordered (alphabetically by the first word) listing of all circular shifts of the input lines such that no line in the output begins with an uninteresting word.

The construction of the program proceeds by small steps* in this way until ultimately each operation is expressed in the available primitive operations of the target language. We shall not carry out the details since the objective of this paper is not to be a tutorial on this methodology. However, note that the methodology achieves the goals set out for it. Since the context is small at each step it is relatively easy to understand what is going on; indeed, it is easy to prove that the program will work correctly if the primitives from which it is constructed are correct. Moreover, proving the correctness of the primitives used at step i is a small set of proofs (of the same kind) at step $i+1$. (In the terminology of this methodology, step i is an *abstraction* from its implementation in step $i+1$.)

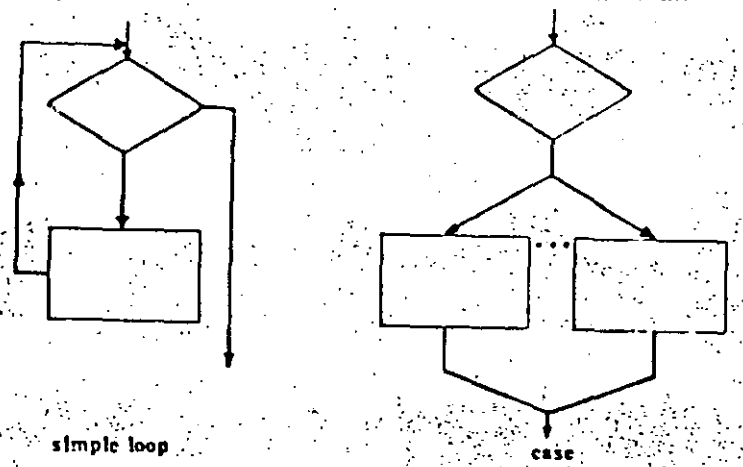
Now, the constructive programming methodology relates to eliminating the goto in the following way. It is crucial to the constructive philosophy that it should be possible to define the behavior of each primitive action at the ith step independent of the context in which it occurs. If this were not so, it would not be possible to prove the correctness of these primitives at the $i+1$st step without reference to their context in the ith step. In particular, this suggests (using flow chart terminology) that it should be possible to represent each primitive at the ith step by a (sub) flow chart with a single entry and a single exit path. Since this must be true at each step of the construction, the final flow chart of a program constructed in this way must consist of a set of totally nested (sub) flow charts. Such a flow chart can be constructed without an explicit goto if conditional and looping constructs are available.

Consider, now, programs which can be built from only simple conditional and loop constructs. To do this we will use a flow chart representation because of the explicit way in which it manifests control. We assume two basic flow chart elements, a "process" box and a "binary decision" box:

These boxes are connected by directed line segments in the usual way. We are interested in two special "goto-less" constructions fabricated from these primitives: a simple loop and an n-way conditional, or "case," construct. We consider these forms "goto-less" since they contain single entry and exit points and hence might reasonably be provided in a language by explicit syntactic constructs. (The loop considered here obviously does not correspond to all vari-

*A more complete explication of the methodology would concern itself with the nature and order of the decisions made at each step as well as the fact that they are small. See [22] for an analysis of two alternative decompositions of a KWIC system similar to the one defined here.

ants of initialization, test before or after the loop body, etc. These variants would not change the arguments to follow and have been omitted.)

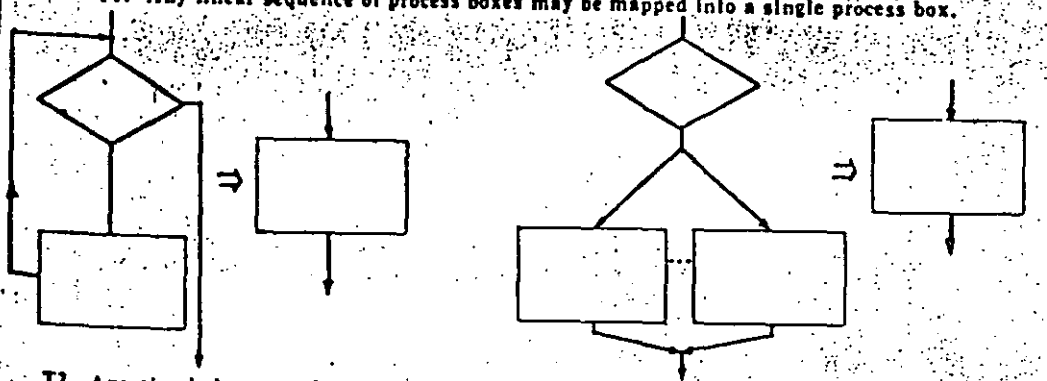


simple loop

case

Consider the following three transformations (T1, T2, T3) defined on arbitrary flow charts:

T1. Any linear sequence of process boxes may be mapped into a single process box.



T2. Any simple loop may be mapped into a process box.

T3. Any n-way "case" construct may be mapped into a process box.

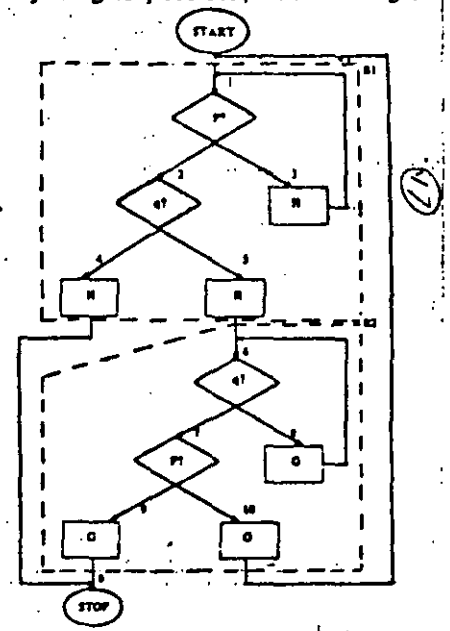
Any graph (flow chart) which may be derived by a sequence of these transformations we shall call a "reduced" form of the original. We shall say that a graph which may be reduced to a single node by some sequence of transformations is "goto-less" (independent of whether actual goto statements are used in its encoding) and that the sequence of transformations defines a set of nested "control environments." The sequence of transformations applied in order to reduce a graph to a single node may be used as a guide to both understanding and proving the correctness of the program [2, 4, 6, 7, 19].

The property of being "goto-less" in the sense defined above is a necessary condition for the program to have been designed by the constructive methodology. Moreover, the property depends only upon the topology of the program and not on the primitives from which it is synthesized; in particular, a goto statement might have been used. However, not only can such programs be constructed without a goto if conditionals and loops are available, but *any* use of a goto which is not equivalent to one of these will destroy the requisite topology. Hence any set of restrictions (on the *use* of the goto) which is intended to achieve this topology is equivalent to eliminating the goto.

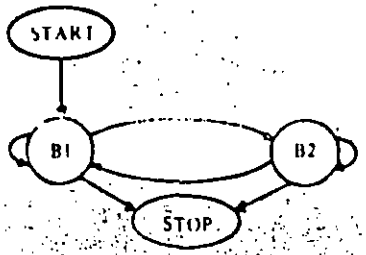
The theoretical possibility of eliminating the GOTO

It is possible to express the evaluation of an arbitrary computable function in a notation which does not have an explicit goto. This is not particularly surprising since: (1) several formal systems of computability theory, e.g., recursive functions, do not use the concept; (2) (pure) LISP does not use it; and (3) van Wijngaarden [13], in defining the semantics of Algol, eliminated labels and goto's by systematic substitution of procedures. However, this does not say that an algorithm for the evaluation of these functions is especially convenient or transparent in goto-less form. Alan Perlis has referred to similar situations as the "Turing Tarpit" in which everything is possible, but nothing is easy.

Knuth and Floyd [14] and Ashcroft and Manna [15] have shown that given an arbitrary flow chart it is *not* possible to construct another flow chart (using the same primitives and no additional variables) which performs the same algorithm and uses only simple conditional and loop constructs; of course other algorithms exist that compute the same function and which can be expressed with only simple conditionals and loops. The example given in Ashcroft and Manna of an algorithm which cannot be written in goto-less form without adding additional variables is:



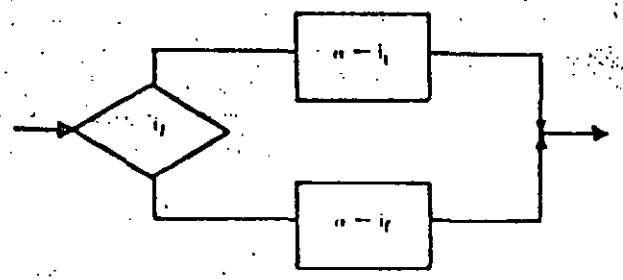
By enclosing some of the regions of the flow chart in dotted lines and labeling them (B1 and B2) as shown on the previous page, and further abstracting from the details of the process and decision structure, the abstract structure of this example is:



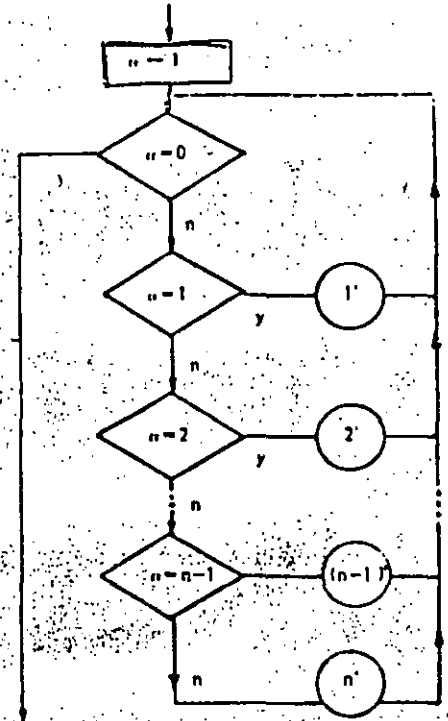
The reader is referred to [15] for a proof that such programs cannot be constructed from simple looping and conditional constructs unless an additional variable is added. Intuitively, however, it should be clear from the abstraction of the example that neither B1 nor B2 is inherently nested within the other. Moreover, the existence of multiple exit paths from B1 and B2 make it impossible to impose a superior (simple) loop (which inherently has a single exit path) to control the iteration between them unless some mechanism for path selection (e.g., an additional variable) is introduced.

In [21] Böhm and Jacopini show that an arbitrary flow chart program may be translated into an equivalent one with a single "while statement" by introducing new boolean variables, predicates to test them, and assignment statements to set them. A variant of this scheme involving the addition of a single integer variable, call it "α," which serves as a "program counter," is given below.

Suppose some flow chart program contains a set of process boxes assigned arbitrary integer labels $i_1, i_2, \ldots, i_n$, and decision boxes assigned arbitrary integer labels $i_{n+1}, i_{n+2}, \ldots, i_m$. (By convention assume the STOP box is assigned the label zero, and the entry box is assigned the label one.) For each process box, i_j, create a new box, i_j', identical to the former except for the addition of the assignment "$\alpha \leftarrow i_k$" where i_k is the label of the successor of i_j in the original program. For each decision box, i_j, create the macro box, i_j':

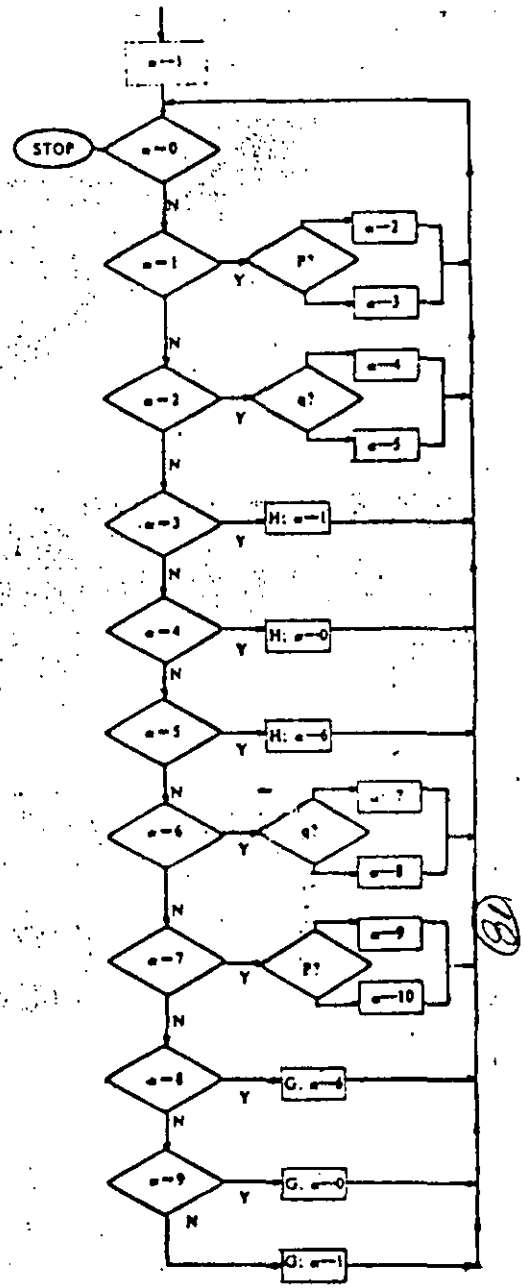


where i_t and i_f are the labels of the successors of the true and false branches of the decision box, i_j, in the original program. Now create the following flow chart:



Thus, for example, the Ashcroft and Manna example given earlier (the labels are given on the earlier diagram) becomes (see right):

Constructions such as the one given at right are undesirable not only because of their inefficiency, but because they destroy the topology (loop structure) and locality of the original program and thus make it extremely difficult to understand. Nevertheless, the construction serves to illustrate the point that adding (at least one) control variable is an effective device for eliminating the goto. Ashcroft and Manna have given algorithms for translating arbitrary programs into goto-less form (with additional variables) which preserve the efficiency and topology of the original program.



The practical possibility of eliminating the GOTO

As discussed in the previous section, it is theoretically possible to eliminate the goto. Moreover, there can be little quarrel with the *objectives* of the constructive programming methodology. A consequence of the particular methodology presented above is that it produces goto-less programs; thus the goto is unnecessary in programs produced according to this methodology. A key, perhaps the key, issue, then, is whether it is *practical* to remove the goto. In particular there is an appropriate suspicion among practicing programmers* that coding without the goto is both inconvenient and inefficient. In this section we shall investigate these two issues, for, if it is inconvenient or grossly inefficient to program without the goto then the practicality of the methodology is in question.

Convenience:

Programming without the goto is *not* (necessarily) inconvenient. The author is one of the designers, implementors, and users of a "systems implementation language," Bliss [16, 17, 18]; Bliss does not have goto. The language has been in active use for three years; we have thus gained considerable practical experience programming without the goto. This experience spans many people and includes several compilers, a conversational programming system (APL), an operating system, as well as numerous applications programs.

The inescapable conclusion from the Bliss experience is that the purported inconvenience of programming without a goto is a myth! Programmers familiar with languages in which the goto is present go through a rather brief and painless adaptation period: Once past this adaptation period they find that the lack of a goto is not a handicap; on the contrary, the invariant reaction is that the enforced discipline of programming without a goto structures and simplifies the task.

Bliss is not, however, a simple goto-less language; that is, it contains more than simple while-do and if-then-else (or case) constructs. There are natural forms of control flow that occur in real programs which, if not explicitly provided for in the language, either require a goto so that the programmer may synthesize them, or else will cause the programmer to contort himself to mold them into a goto-less form (e.g., in terms of the construction in the previous section). Contortion obscures and is therefore antipathetic with the constructive philosophy; hence the approach in Bliss has been to provide explicit forms of these natural constructs which are also inherently well-structured. In [19] the author analyzes the forms of control flow which are not easily realized in a simple goto-less language and uses this analysis to motivate the facilities in Bliss. Here we shall merely list some of the results of that analysis as they manifest themselves in Bliss (and might manifest themselves in any goto-less language):

1. A collection of "conventional" control structures: Many of the inconveniences of a simple goto-less language are eliminated by simply providing a fairly large collection of more-or-less "conventional" control structures. In particular, for example, Bliss includes: control "scopes" (blocks and compounds), conditionals (both if-then-else and case forms), several looping constructs (including while-do, do-while, and stepping forms), potentially recursive procedures, and co-routines.

2. Expression Language: As noted in an earlier section, one mechanism for expressing algorithms in goto-less form is through the introduction of at least one additional variable. The value of this variable serves to encode the state of the computation and direct subsequent flow. This is a common programming practice used even in languages in which the goto is present (e.g., the FORTRAN "computed goto"). Bliss is an "expression language" in the sense that every construct, including those which manifest control, is an expression and computes a value. The value of an expression (e.g., a block or loop) forms a natural and convenient implicit state variable.

3. Escape Mechanism: Analysis of real programs strongly suggests that one of the most common "good" uses of a goto is to prematurely terminate execution of a control environment — for example, to exit from the middle of a loop before the usual termination condition is satisfied. To accommodate this form of control Bliss* allows any expression (control environment) to be labeled; an expression of the form "leave <label> with <expression>" may be executed within the scope of this labeled environment. When a leave expression is executed two things happen: (1) control immediately passes to the end of the control environment (expression) named in the leave, and (2) the value of the named environment is set to that of the <expression> following the with. Note that the leave expression is a restricted form of forward branch just as the various forms of loop constructs are restricted backward jumps. In both cases the constructs are less general, and less dangerous, than the general goto.

*Including this author when he first read Dijkstra's letter in 1968.

*A somewhat different form of the Bliss escape is described in [19]; the form described in [19] has been replaced by that described above.

In summary, then, our experience with Bliss supports the notion that programming without the goto is no less convenient than with it. This conclusion rests heavily on the assumption that the goto was not merely removed from some existing language, but that a coherent selection of well-structured constructs were assembled as the basis of the control component of the new language. It would be unreasonable to expect that merely removing the goto from an existing language, say FORTRAN or PL/I, would result in a convenient notation. On the other hand, it is *not* unreasonable to expect that a relatively small set of additions to an existing language, especially the better structured ones such as Algol or PL/I, could reintroduce the requisite convenience. While not a unique set of solutions, the control mechanisms in Bliss are one model on which such a set of additions might be based.

Efficiency:

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity. One of these sins is the construction of a "rat's nest" of control flow which exploits a few common instruction sequences. This is precisely the form of programming which must be eliminated if we are ever to build correct, understandable, and modifiable systems.

There *are* applications (e.g., "real time" processing) and there *are* (a few) portions of every program where efficiency is crucial. This is a real issue. However, the appropriate mechanism for achieving this efficiency is a highly optimizing compiler, not incomprehensible source code. In this context it is worth noting another benefit of removing the goto — a benefit which the author did not fully appreciate until the Bliss compiler was designed — namely, that of global optimization. The presence of goto in a block-structured language with dynamic storage allocation forces runtime overhead for jumping out of blocks and procedures and may imply a distributed overhead to support the possibility of such jumps. Eliminating the goto removes both of these forms of overhead. More important, however, is that: (1) the scope of a control environment is statically defined, and (2) all control appears as one of small set of explicit control constructs. A consequence of (1) is that the Fortran-H compiler [20], for example, expends a considerable amount of effort in order to achieve roughly the same picture of overall control as that implicit in the text of a Bliss program. The consequence of (2) is that the compiler need only deal with a small number of well-defined control forms; thus failure to optimize a peculiarly constructed variant of a common control structure is impossible. Since flow analysis is pre-requisite to global optimization, this benefit of eliminating the goto must not be underestimated.

Summary

One goal of our profession must be to produce large programs of predictable reliability. To do this requires a methodology of program construction. Whatever the precise shape of this methodology, whether the one sketched earlier or not, one property of that methodology must be to isolate (sub) components of a program in such a way that the proof of the correctness of an abstraction from these components can be made independent of both their implementation and the context in which they occur. In particular this implies that unrestricted branching between components cannot be allowed.

Whether or not a language contains a goto and whether or not a programmer uses a goto in some context is related, in part, to the variety and extent of the other control features of the language. If the language fails to provide important control constructs, then the goto is a crutch from which the programmer may synthesize them. The danger in the goto is that the programmer may do this in obscure ways. The advantage in eliminating the goto is that these same control structures will appear in regular and well-defined ways. In the latter case, both the human and the compiler will do a better job of interpreting them.

References

1. E.W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3 (March 1968), pp. 147-48.
2. _____, "A Constructive Approach to the Problem of Program Correctness," *BIT*, Vol. 8, No. 3 (1968), pp. 174-86.
3. _____, "Structured Programming," *Software Engineering, Concepts and Techniques. Proceedings of the NATO Conferences*, eds. P. Naur, B. Randell, and J.N. Buxton (New York: Petrocelli/Charter, 1976), pp. 222-26.
4. _____, *Notes on Structured Programming*, 2nd ed., Technische Hogeschool Eindhoven, Report No. EWD-248, 70-WSK-0349 (Eindhoven, The Netherlands: April 1970); see also *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (New York: Academic Press, 1972).
5. P. Naur, "Proof of Algorithms by General Snapshots," *BIT*, Vol. 6, No. 4 (1966), pp. 310-16.
6. _____, "Programming by Action Clusters," *BIT*, Vol. 9, No. 3 (1969), pp. 250-58.
7. C.A.R. Hoare, "Proof of a Program: FIND," *Communications of the ACM*, Vol. 14, No. 1 (January 1971), pp. 39-45.
8. N. Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM*, Vol. 14, No. 4 (April 1971), 221-27.
9. D.L. Parnas, "Information Distribution Aspects of Design Methodology," *Proceedings of the 1971 IFIP Congress* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1971), pp. 339-44; see also Carnegie-Mellon University, Computer Science Technical Report (Pittsburgh: 1971).
10. J. King, "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University, 1969.
11. Z. Manna, "Termination of Algorithms," Ph.D. Thesis, Carnegie-Mellon University, 1968.
12. _____, "The Correctness of Programs," *Journal of Computer & System Sciences*, Vol. 3 (May 1969), pp. 119-27.
13. A. van Wijngaarden, "Recursive Definition of Syntax and Semantics," *Formal Language Description Languages*, ed. T.B. Steel (Amsterdam, The Netherlands: North-Holland Publishing Co., 1966).
14. D. Knuth and R.W. Floyd, "Notes on Avoiding 'go to' Statements," *Information Processing Letters*, Vol. 1, No. 1 (February 1971), pp. 23-31, 177; see also Stanford University, Computer Science Technical Report, Vol. CS148 (Stanford, Calif.: January 1970).
15. E. Ashcroft and Z. Manna, "The Translation of 'go to' Programs to 'while' Programs," *Proceedings of the 1971 IFIP Congress*, Vol. I (Amsterdam, The Netherlands: North-Holland Publishing Co., 1972), pp. 250-55; see also Stanford University, AI Memo AIM-138, STAN CS-71-88 (Stanford, Calif.: January 1971).
16. W.A. Wulf, et al., *BLISS/11 Reference Manual*, Carnegie-Mellon University, Computer Science Department Report No. AD-739964 (Pittsburgh: March 1972).
17. W.A. Wulf, D.B. Russell, and A.N. Habermann, "BLISS: A Language for Systems Programming," *Communications of the ACM*, Vol. 14, No. 12 (December 1971), pp. 780-90.
18. W.A. Wulf, et al., "Reflections on a Systems Programming Language," *Proceedings of the SIGPLAN Symposium on Systems Implementation Languages* (October 1971).
19. W.A. Wulf, "Programming without the GOTO," *Proceedings of the 1971 IFIP Congress*, Vol. 1 (Amsterdam, The Netherlands: North-Holland Publishing Co., 1972), pp. 408-13.
20. E.S. Lowry and C.W. Medlock, "Object Code Optimization," *Communications of the ACM*, Vol. 12, No. 1 (January 1969), pp. 13-22.
21. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71.
22. D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12 (December 1971), pp. 1053-58.
23. Z. Manna, S. Ness, and J. Vuillemin, "Inductive Methods for Proving Properties of Programs," *Communications of the ACM*, Vol. 16, No. 8 (August 1973), pp. 491-502.
24. R. Burstall, "An Algebraic Description of Programs with Assertions, Verification and Simulation," *Proceedings of the ACM Conference on Proving Assertions About Programs, SIGPLAN Notices*, Vol. 7, No. 1 (January 1972), pp. 7-14.

A Case for the GOTO

Introduction

It is with some trepidation that I undertake to defend the goto statement, a construct, which while ancient and much used has been shown to be theoretically unnecessary [1] and in recent years has come under so much attack [2]. In my opinion, there have been far too many goto statements in most programs, but to say this is not to say that goto should be eliminated from our programming languages. This paper contains a plea for the retention of goto in both current and future languages. Let us first examine the context in which the controversy occurs.

A wise philosopher once pointed out to a lazy king that there is no royal road to geometry. After discovering, in the late fifties, that programming was *the* computer problem, a search was made during the sixties for the royal road to programming. Various paths were tried including comprehensive operating systems, higher level languages, project management techniques, time sharing, virtual memory, programmer education, and applications packages. While each of these is useful, they have not solved the programming problem. Confronted with this unresolved problem and with few good ideas on the horizon, some people are now hoping that the royal road will be found through style, and that banishment of the goto statement will solve all. The existence of this controversy and the seriousness assigned to it by otherwise very sensible people are symptoms of a malaise in the computing community. We have few promising new ideas at hand. I also suspect that the controversy reflects something rather deep in human nature, the notion that language is magic and the mere utterance of certain words is dangerous or defiling. Is it an accident that "goto" has four letters?

Having indicated my belief that this controversy is not quite as momentous as some have made out, it is appropriate to point out some beneficial aspects. First, interest has been focused on programming style and while style is not everything it does have a great deal of importance. Second, the popularity of the no goto rule is, in large part, due to the fact that it is a simple rule which does improve the code produced by most programmers. As we shall see, this is not sufficient grounds for banishing the construct from our languages, although it may well justify teaching alternative methods of programming to beginners or restricting its use on a project. Perhaps the most beneficial aspect of the controversy will be to encourage the use of block structure languages and to discourage use of our most popular languages, COBOL and FORTRAN as they are not well suited to programming without the goto.

The principal motivation behind eliminating the goto statement is the hope that the resulting programs will not look like a bowl of spaghetti. Instead they will be simple, elegant and easy to read, both for the programmer who is engaged in composition and checkout as well as the poor soul attempting future modification. By avoiding goto one guarantees that a program will consist entirely of one-in-one-out control structures. It is easy to read a properly indented program without goto statements, which is written in a block structure language. The possible predecessors of every statement are obvious and, with the exception of loops, all predecessors are higher on the page. (I assume nobody writes inner procedures longer than a page anymore?) Why then should we retain the goto statement in our current and future programming languages?

Theoretical considerations

It has been demonstrated that there are flow charts which cannot be converted to a procedural notation without goto statements [3, 4]. It turns out though that this result is not really an argument for the retention of goto as there are means by which a procedure can be *rewritten* in a systematic manner to eliminate all instances of goto. An almost trivial method is to introduce a new variable which can be thought of as the instruction counter along with tests and sets of this counter. The method is fully described by Böhm and Jacopini [1]. The results of this procedure when applied to a large program with many instances of goto would usually be a program which is less readable than the original program with goto statements. However, nobody seems to be advocating using such unconsidered methods.

The real issue is that theoretical work has suggested a number of techniques that can be used to rewrite programs, eliminating the instances of goto. These include replication of code (node splitting), the introduction of new variables and tests as well as the introduction of procedures. Any of these techniques, when used with discretion, can increase the readability of code. The question is whether there are any instances when the application of such methods decreases clarity or produces some other undesirable effect. Whether or not to retain the goto does not seem to be a theoretical issue. It is rather a matter of taste, style and the practical considerations of day to day computer use.

Alternatives to GOTO

With respect to current languages which are in wide use such as COBOL and FORTRAN, there is the practical consideration that the goto statement is necessary. Even where a language is reasonably well-suited to programming without the goto, the elimination of this construct may be at once too loose and too restrictive. PL/I provides some interesting examples here. One exits from an Algol procedure when the flow of control reaches the end bracket. PL/I provides an additional mechanism, an explicit RETURN statement. Consider the table look up in Fig. 1 which is similar to an example of Floyd and Knuth [4]. The problem is to find an instance of X in the vector A and if there is no instance of X in A, then make a new entry in A of the argument X. In either case the index of the entry is returned. A count of the number of matches associated with each entry in A is also maintained in an associated vector, B. In this example there are no goto statements but the two RETURN statements cause an exit from both the procedure and the iterative DO. Thus the procedure has control structures which have more than one exit and one-in-one-out control structures were a principal reason for avoiding goto. Should the PL/I programmer add a rule forbidding RETURN? The procedure could then be rewritten as in Fig. 2. This involves the introduction of a new variable, SWITCH, and a new test. If one assumes that the introduction of gratuitous identifiers and tests is undesirable perhaps RETURN is a desirable construct even though it can result in multiple exit control structures. It is my feeling that procedures with several RETURN statements are easy to read and modify because they follow the top to bottom pattern and maintain the obvious predecessor characteristic, while avoiding the introduction of new variables. RETURN is therefore preferable to the alternative of introducing new variables and tests.

However, RETURN is a very specialized statement. It only permits an exit from one level of one type of control, the procedure. One could generalize the construct to apply to multiple levels of control and to DO groups or BEGIN blocks as well as procedures. This is exactly the flavor of the BLISS leave [6] construct. Lacking such language, the PL/I user must content himself with goto. But is this a bad thing? The good programmer, who understands the potential complexity which results from excessive use of goto, will attempt to recast such an algorithm. Failing to find an elegant restatement, he will insert the label and its associated goto out of the desired control structure. The label stands there as a warning to the reader of the routine that this is a procedure with more than the usual complexity. Note also that the label point catches the eye. It is immediately apparent when looking at this statement that it has an unusual predecessor. The careful reader will want to consult a cross reference listing to determine the potential flow of control. Note that the BLISS leave construct is somewhat less than ideal here. In BLISS when one examines the code which follows a bracket terminating a level of control, its potential predecessors are not immediately apparent. One must look upward on the page for its associated label, which indicates a potential unusual predecessor and then find the leave. It is my feeling that unusual exits from levels of control should be avoided. The multiple level case is especially ugly. Where such constructs are necessary, it should be made completely obvious to all. Statements such as the BLISS leave encourage unusual exits from multiple levels of control. One should not cover up the fact that there is an awkward bit of logic by the introduction of a new control construct.

```
LOOK_UP:
    PROC (X);
        DO I = 1 TO A_TOP;
            IF A(I) = X  THEN
                DO;
                    B(I) = B(I) + 1;
                    RETURN (I);
                END;
        END;
        A(I) = X;
        B(I) = 1;
        A_TOP = A_TOP + 1;
        RETURN (I);
END;
```

Figure 1

```
LOOK_UP2:
    PROC (X);
        SWITCH = 1;
        DO I = 1 TO A_TOP WHILE (SWITCH = 1);
            IF A(I) = X THEN
                SWITCH = 0;
        END;
IF SWITCH = 0 THEN
        B(I) = B(I) + 1;
    ELSE
        DO;
            A(I) = X;
            B(I) = 1;
            A_TOP = A_TOP + 1;
        END;
    RETURN(I);
END;
```

Figure 2

Another interesting PL/I control construct is the ON unit. This is a named block which is automatically invoked on certain events such as overflow, but it can also be invoked explicitly by a statement of the form:

SIGNAL CONDITION (name);

The name is established dynamically and need not be declared in the scope of the SIGNAL. This facility often eliminates the need to pass special error return parameters or test a return code which indicates abnormal termination of a lower level procedure. After completion of an ON unit activated by SIGNAL, control is passed back to the statement following the SIGNAL. This is usually not useful. One wants to terminate the signaling block and the only way to do this in PL/I is with a goto out of the ON unit. Is SIGNAL permissible under the no goto criteria? Elimination of goto seems too restrictive here as SIGNAL is a useful facility which can eliminate much messy programming detail. However, the natural consequence of using the SIGNAL statement is to terminate an ON unit with a goto. Perhaps it is best to admit that there is no very good alternative to a goto statement in this situation.

GOTO as a basic building block

The lack of a case statement in PL/I is a clear deficiency. The resourceful programmer will construct one out of a goto. This does not make up for the lack of a case statement, but it does point up an interesting and highly legitimate use of goto. One can use it as a primitive to construct more advanced and elegant control structures. Imaginative programmers will, from time to time, develop new control constructs as Hoare invented the case statement [5]. Those that are worthwhile will be informally defined and implemented with a macro preprocessor. The better ones will appear in experimental compilers and eventually the best will find their way into the standard languages. Such inventions are often very hard to implement with macro preprocessors for existing languages without use of the goto construct. There is still room for the incorporation of unusual control mechanisms into existing block structure languages. Decision tables are a prime example. One way of handling decision tables is to have a preprocessor convert them to source language statements. If a convenient translation process introduces goto statements, this is not important as the basic documentation is at the decision table level. The source language is treated as an internal language. The ease of translation is more important than the introduction of goto statements.

Another related reason for retaining goto even in our newest languages is that it is often possible to use a language as the target to which one translates a secondary source language. If the secondary language has goto or even a different set of control constructs, then translation could be very difficult without a goto in the target language. In other words source languages and their associated compilers are useful building blocks for the development of special constructs or languages and elimination of goto decreases the range of usefulness of a language.

GOTO as an escape

Part of the reason for retaining goto is that the world is not always a very elegant place and sometimes a goto is a useful, if ugly, tool to handle an awkward situation. Algorithms are often messy. Sometimes this may be due to inherent complexity. I suspect, however, that most of the time it is because not enough time or intelligence has been applied. Where time or intelligence are lacking, a goto may do the job. Every program will not be published. Many may be used only once. I tend to sympathize with the programmer who fixes up a one time program at 3:00 a.m. with a goto. Of course, there is always the danger that the programmer will lapse into bad habits but I am willing to take that chance. Perhaps it is an opportunity, for when the intelligent supervisor reads the code of those under him, he can focus on any goto statements. A programmer should be able to justify each use of goto.

I have avoided discussing performance, which like death and taxes, none of us can avoid forever. Suppose a procedure runs too slowly or takes up too much space. A rewrite of the procedure or restructuring of the data may be in order. But if that fails one may be driven to a rewrite in assembly language. There is an intermediate alternative which may solve the problem without resort to an assembler. The programmer who writes structured programs uses certain techniques such as the introduction of procedures and the repetition of code which can result in the loss of time and space. Given the idiosyncracies of many compilers, a little reorganization of code and a few goto statements inserted by a clever programmer can often improve performance. This is not a practice which I recommend for those starting a project, even where it is known to have stringent performance requirements. One should give up a structured program in a higher level language only after performance bottlenecks have been clearly identified and then only give up what is absolutely necessary. My guess is that very few such situations will exist but when they do, a slightly contorted procedure in a higher level language may be an attractive alternative to one written in assembly language. The villain here is the compiler which produces bad code in some situations. Would elimination (as opposed to avoidance) of goto significantly ease the task of compiler writers and thus help us to get better object code? It is difficult to do justice to this problem as there are so many different compiling techniques and some would be helped and some would not. My feeling is that elimination of the goto would not dramatically ease the problems of compiler writers. Even in compilers which do extensive control flow analysis, a small percentage of implementation effort is devoted to that task. A more interesting subject for compiler writers is the identification of those optimizations which improve the performance of programs written with none or very few goto statements. Viewed in this light the

existence of well-structured programs imposes an additional obligation and more work on compiler writers. This is work which they should eagerly accept so that programmers will not have to make the trade-off between a well-structured program and one that performs well. More work is required in this area.

Varieties of programming style

The goto issue is part of the larger topic of overall programming style. One of my worries is that we will become the prisoners of one currently fashionable "classical" style. Perhaps other rules of style are better. For example we might say that only a goto which was directed forward was elegant. Perhaps it is useful to restrict ourselves to standard type labels such as "PROC_EXIT." Vagaries of style or fashion need not disturb students who should be taught in a rather constrained way which is established by the teacher. Also, those working on large projects will have to conform to standards. However, experienced programmers and language designers of taste and imagination will want to experiment and they should be encouraged to do so. APL provides an interesting example of a diverse style. A computed goto, in the form of a right pointing arrow exists in APL, but other than function invocation there are no control constructs such as IF THEN ELSE or an iteration statement. Surprisingly one does not get a maze of goto statements in a well-written APL function, for the powerful array operators can be used in situations where loops occur in other languages. Sequential execution of statements thus becomes the general rule and few right pointing arrows are required. Whether an algorithm written in APL is clearer than the same algorithm written in a block structure language seems to be a matter on which intelligent people of taste will disagree.

Elegance in programming involves more than avoiding goto, and beyond the goto controversy there are a great many other important issues of style. There is the question about the clarity of array operations in APL and PL/I, as well as structure operations in COBOL and PL/I. To what extent are implicit conversions a subsumption of extraneous detail and in what instances do they produce surprising results? There are many questions about optimal size and complexity with respect to expressions, nesting of IF and iteration statements as well as the size and complexity of procedures. To what extent do declarations properly subsume detail and to what extent do they leave the meaning of a statement unclear unless one is simultaneously examining the declaration? Under what circumstances, if any, should functions have side effects or should iteration replace recursion? To what extent can we eliminate assignment? These and other questions are subtle but important stylistic problems which we are likely to pass over if we concentrate too heavily on the relatively simple and unimportant issue of goto.

References

1. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71.
2. E.W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3 (March 1968), pp. 147-48.
3. E. Ashcroft and Z. Manna, "The Translation of 'go to' Programs to 'while' Programs," *Proceedings of the 1971 IFIP Congress*, Vol. I (Amsterdam, The Netherlands: North-Holland Publishing Co., 1972), pp. 250-55; see also Stanford University, AI Memo, AIM-138, STAN CS-71-88 (Stanford, Calif.: January 1971).
4. D.E. Knuth and R.W. Floyd, "Notes on Avoiding 'go to' Statements," *Information Processing Letters*, Vol. 1, No. 1 (February 1971), pp. 23-31; see also Stanford University, Computer Science Technical Report, Vol. CS148, (Stanford, Calif., January 1970).
5. N. Wirth and C.A.R. Hoare, "A Contribution to the Development of ALGOL," *Communications of the ACM*, Vol. 9, No. 6 (June 1966), pp. 413-32.
6. W.A. Wulf, D.B. Russell, and A.N. Habermann, "BLISS: A Language for Systems Programming," *Communications of the ACM*, Vol. 14, No. 12 (December 1971), pp. 780-90.

Conclusion

goto should be retained in both current and future languages because it is useful in a limited number of situations. Programmers should work hard to produce well-structured programs with one-in-one-out control structures which have no goto statements. Where this is not possible, we should not think that elegance is achieved with a magic language formula. It is far better to admit the awkwardness and use the goto. Furthermore, goto is a useful means to synthesize more complex control structures and increases the usefulness of a language as a target to which other languages can be translated. Viewed in the light of practical programming as an ultimate escape, goto can also be justified if not encouraged. Finally our wisdom has not yet reached the point where future languages should eliminate the goto. If future work indicates that by avoiding goto we can gain some important advantage such as routine proofs that programs are correct, then the decision to retain the goto construct should be reconsidered. But until then, it is wise to retain it.

On the Composition of Well-Structured Programs

Introduction

In the first decade of computers, say up to the early sixties, computers were quite limited in their power. The task of the programmer was to formulate algorithms in the specific order codes of these machines so that they were utilized as effectively as possible. Primarily because of their limitations, this task was achieved by collecting sets of clever techniques and startling tricks, and by finding applications for them as frequently as possible. Examples of such techniques were the programmed self-modification of parts of the program, such as, for instance, the conversion of conditional jumps into dummy instructions and vice versa, or the sharing of store for functionally independent, but never simultaneously used auxiliary variables.

Tricks were necessary at this time, simply because machines were built with limitations imposed by a technology in its early development stage, and because even problems that would be termed "simple" nowadays could not be handled in a straightforward way. It was the programmers' very task to push computers to their limits by whatever means available. We should recall that the absence of index registers (and indirect addressing), for example, made automatic code modification a mere necessity (see also [1, 2]).

The essence of programming was understood to be the *optimization of the efficiency* of particular machines executing particular algorithms. As computers grew more powerful, the problems posed to the programmers grew proportionally, and as a result, the growing power of hardware did not ease, but rather increased the burden. The elimination of deficiencies, errors and blunders — called debugging — became the overwhelming problem.

Understandably, the remedy was sought in the development and use of better tools in the form of programming languages. The amount of resistance and prejudices which the farsighted originators of FORTRAN had to overcome to gain acceptance of their product is a memorable indication of the degree to which programmers were preoccupied with efficiency, and to which trickology had already become an addiction. However, once these adversities and fears had been overcome, FORTRAN had a tremendous impact — an impact that is still felt today. ALGOL 60 followed several years later; it went beyond FORTRAN in several significant respects, but essentially shared the same purpose and intention. In particular, it extended to the level of statements what FORTRAN had introduced on the level of (arithmetic) expressions: *structure*. But ALGOL 60 was not very successful when measured by its frequency of use in technical and commercial applications. There are many reasons for this, one being that it appeared on the scene when the relevance of structure had not yet been widely recognized, and its restrictiveness against the use of clever tricks was considered to be a handicap and a deficiency. The law of the "Wild West of Programming" was still held in too high esteem! The same inertia that kept many assembly code programmers from advancing to use FORTRAN is now the principal obstacle against moving from a "FORTRAN style" to a structured style.

As the power of computers on the one side, and the complexity and size of the programmer's task on the other continued to grow with a speed unmatched by any other technological venture, it was gradually recognized that the true challenge does not consist in pushing computers to their limits by saving bits and microseconds, but in being capable of organizing large and complex programs, and assuring that they specify a process that for all admitted inputs produces the desired results. In short, it became clear that any amount of efficiency is worthless if we cannot provide *reliability* [4]. But how can this reliability be provided? Here structure enters the scene as the one essential tool for mastering complexity, the effective means of converting a seemingly senseless mass of bits or characters into meaningful and intelligible information. We must recognize the strong and undeniable influence that our language exerts on our ways of thinking, and in fact defines and delimits the abstract space in which we can formulate — give form to — our thoughts.

But now the term *structured programming* has been coined, and it seems finally to be achieving what the term "structured language" was unable to suggest. It was first used by E.W. Dijkstra [3], and has spread with various interpretations and connotations since then. It is the expression of a conviction

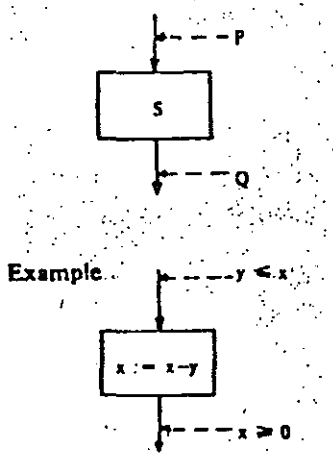
that the programmers' knowledge must not consist of a bag of tricks and trade secrets, but of a general intellectual ability to tackle problems systematically, and that particular techniques should be replaced (or augmented) by a method. At its heart lies an *attitude* rather than a recipe: the admission of the limitations of our minds. The recognition of these limitations can be used to our advantage, if we carefully restrict ourselves to writing programs which we can manage intellectually, where we fully understand the totality of their implications.

1. Intellectual manageability of programs

Our most important mental tool for coping with complexity is *abstraction*. Therefore, a complex problem should not be regarded immediately in terms of computer instructions, bits, and "logical words," but rather in terms and entities natural to the problem itself, abstracted in some suitable sense. In this process, an abstract program emerges, performing specific operations on abstract data, and formulated in some suitable notation — quite possibly natural language. The operations are then considered as the constituents of the program which are further subjected to decomposition to the next "lower" level of abstraction. This process of *refinement* continues until a level is reached that can be understood by a computer, be it a high-level programming language, FORTRAN, or some machine code [5, 6].

For the intellectual manageability, it is crucial that the constituent operations at each level of abstraction are connected according to sufficiently simple, well understood *program schemas*, and that each operation is described as a piece of program with *one starting point* and a *single terminating point*. This allows defining states of the computation (P, Q), i.e., relations among the involved variables, and attaching them to the starting and terminating points of each operation (S). It is immaterial, at this point, whether these states are defined by rigorous mathematical formulas (i.e., by predicates of logical calculus) or by sufficiently clear and informative sentences, or by a combination of both. The important point is that the programmer has the means to gain clarity about the interface conditions between the individual building blocks out of which he composes his program [7].

(1)



An example may clarify the issues at this point. The reader should be aware that any example that is sufficiently short to fit onto a single page cannot be much more than a metaphor, probably unconvincing to habitual skeptics. The important thing is to abstract from the example and to imagine the same method being applied to large programming problems.

Example 1: Sequential merging

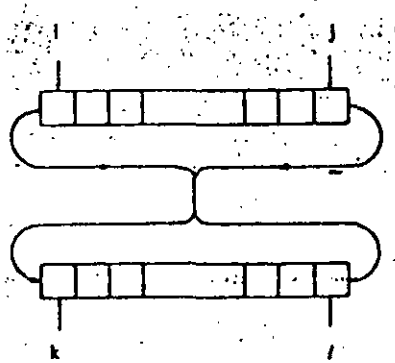
Given a set of $n = 2^N$ integer variables $a_1 \cdots a_n$, find a recipe to permute their values such that $a_1 \leq a_2 \leq \cdots \leq a_n$ using the principle of sequential merging. Thus, we are to sort under the assumption of strictly sequential access. Briefly told, we shall use the following algorithm:

1) Pick individual components $a_i^{(1)}$ and merge them into ordered pairs, depositing them in a variable $a^{(2)}$.

2) Pick the ordered pairs from $a^{(2)}$ and merge them pairwise into ordered quadruples, depositing them in a variable $a^{(3)}$.

3) Continue this game, each time doubling the size of the merged subsequences, until a single sequence of length $n = 2^N$ is generated.

(2)



At the outset, we notice that two variables $a^{(1)}$ and $a^{(2)}$ suffice, if the items are alternately shuttled between them. We shall introduce a single array variable A with $2n$ components, such that $a^{(1)}$ is represented by $A[1] \cdots A[n]$ and $a^{(2)}$ is represented by $A[n+1] \cdots A[2n]$. Each of these two conceptually independent parts has two points of sequential access, or read/write heads. These are to be denoted by pairs of index variables i, j and k, l respectively. We may now visualize the sort process as a repeated transfer under merging of tuples *up* and *down* the array A.

The first version of our program is evidently a repetition of the merge shuttle of p-tuples, where each time around p is doubled and the direction of the shuttle is changed. As a consequence, we need two variables, one to denote the tuple size, one to denote the direction. We will call them p and *up*. Note that each repetitive operation must contain a change of its (control) variables within the loop, an initialization in front of the loop, and a termination condition. We easily convince ourselves of the correctness of the following program:

```
up := true; p := 1;                                          (3)
repeat  1: "initialize indices i, j, k, and l";
        2: "merge p-tuples from i- and
            j-sequences into k- and
            l-sequences";
           up := ¬up; p := 2*p
until p = n
```

Statement-1 is easily expressed in terms of simple assignments depending on the direction of the merge pass:

```
1:  if up then                                               (4)
      begin i := 1; j := n;
            k := n+1; l := 2*n
      end
    else
      begin k := 1; l := n;
            i := n+1; j := 2*n
      end
```

Statement-2 describes the repeated merging of p-tuples; we shall control the repetition by counting the number m of items merged. The sources are designated by the indices i and j; the destination alternates between indices k and l. Instead of introducing a new variable standing alternately for k and l, we use the simple solution of interchanging k and l after each p-tuple merge, and letting k denote the destination index at all times. Clearly, the increment of k has then to alternate between the values $+1$ and -1; to denote the increment, we introduce the auxiliary variable h. We can easily convince ourselves that the following refinement is correct:

```
2:  begin m := n; h := 1;                                    (5)
      repeat m := m-2*p;
        3: "merge one p-tuple from
            each of i and j to k, in-
            crement k after each
            move by h"; h := -h;
        4: "exchange k and l"
      until m = 0
    end
```

Whereas statement-4 is easily expressed as a sequence of simple assignments, statement-3 involves more careful planning. It describes the actual merge operation, i.e., the repeated comparison of the two incoming items, the selection of the lesser one, and the stepping up of the corresponding index. In order to keep track of the number of items taken from the two sources, we introduce the two counter variables q and r. It must be noted that the merge always exhausts only one of the two sources, and leaves the other one nonempty. Therefore, the leftover tail must subsequently be copied onto the output sequence. These deliberations quickly lead to the following description of statement-3:

```
3:  begin q := p; r := p;                                    (6)
      repeat {select the smaller item}
        if A[i] < A[j] then
          begin A[k] := A[i];
                k := k+h; i := i+1;
                q := q-1
          end
        else
          begin A[k] := A[j];
                k := k+h; j := j-1;
                r := r-1
          end
      until (q = 0) ∨ (r = 0);
      5: "copy tail of i-sequence";
      6: "copy tail of j-sequence"
    end
```

The manner in which the tail copying operations are stated demands that they be designed to have no effect, if initially their counter is zero. Use of a repetitive construct testing for termination *before* the first execution of the controlled statement is therefore mandatory.

```
5:  while q ≠ 0 do                                           (7)
      begin A[k] := A[i];
            k := k+h;
            i := i+1; q := q-1
      end

6:  while r ≠ 0 do
      begin A[k] := A[j];
            k := k+h;
            j := j-1; r := r-1
      end
```

This concludes the development and presentation of this program, if a computer is available to accept statements of this form, i.e., if a suitable compiler is available.

In passing, I should like to stress that we should not be led to infer that actual program conception proceeds in such a well organized, straightforward, "top-down" manner. Later refinement steps may often show that earlier decisions are inappropriate and must be reconsidered. But this neat, *nested factorization* of a program serves admirably well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal, and even formal proofs of correctness. The emerging modularity is particularly wel-

come if programs have to be adjusted to changed or extended specifications. This is a most essential advantage, since in practice few programs remain constant for a long time. The reader is urged to rediscover this advantage by generalizing this merge-sort program by allowing n to be any integer greater than 1.

Example 2: Squares and palindromes

List all integers between 1 and N whose squares have a decimal representation which is a palindrome. (A palindrome is a sequence of characters that reads the same from both ends.)

The problem consists in finding sequences of digits that satisfy two conditions: they must be palindromes, and they must represent squares. Consequently, there are two ways to proceed: either generate all palindromes (with $\log N^2$ digits) and select those which represent squares, or generate all squares and then select those whose representations are palindromes. We shall pursue the second method, because squares are simpler to generate (with conventional programming facilities), and because for a given N there are fewer squares than palindromes. The first program draft then consists of essentially a single repetitive statement.

```
n := 0;                                                    (8)
repeat n := n+1; generate square;
  if decimal representation of square
       is a palindrome
    then write n
until n = N
```

The next step is the decomposition of the complicated, verbally described statements into simpler parts. Obviously, before testing for the palindrome property, the decimal representation of the square must have been computed. As an interface between the individual parts we introduce auxiliary variables. They represent the result of one step and function as the argument of the successive step.

$d[1] \cdots d[L]$	an array of decimal digits
L	the number of digits computed
p	a Boolean variable

(note that $L = entier(2\ log\ N) + 1$)

The refined version of (8) becomes

```
n := 0;                                                    (9)
repeat n := n+1; s := n*n;
  d := decimal representation of s;
  p := d is a palindrome;
  if p then write (n)
until n = N
```

and we can proceed to specify the three component statements in even greater detail. The computation of a decimal representation is naturally formulated as the repeated computation of individual digits starting "at the right."

```
L := 0;                                                   (10)
repeat L := L+1;
  separate the rightmost digit of s,
    call it d[L]
until s = 0
```

The separation of the least significant digit is now easily expressed in terms of elementary arithmetic operations as shown in (12). Hence, the next task is the decomposition of the palindrome property p of d. It is plain that it also consists of the repeated, sequential comparison of corresponding digits. We start by picking the first and the last digits, and then proceed inwards. Let i and j be the indices of the compared digits.

```
i := 1; j := L;                                           (11)
repeat compare the digits;
  i := i+1; j := j-1
until (i ≥ j) or digits are unequal
```

A last refinement leads to a complete solution entirely expressed in terms of a conventional programming language with adequate structuring facilities.

```
n := 0;                                                   (12)
repeat n := n+1; s := n*n; L := 0;
  repeat L := L+1;
    r := s div 10; d[L] := s - 10*r;
    s := r
  until s = 0;
  i := 1; j := L;
  repeat p := d[i] = d[j];
    i := i+1; j := j-1
  until (i ≥ j) or ¬p;
  if p then write (n)
until n = N
```

This ends the presentation of Example 2.

2. Simplicity of composition schemes

In order to achieve intellectual manageability, the elementary composition schemes must be simple. We have encountered most of the truly fundamental ones in this second example. They encompass *sequencing*, *conditioning*, and *repetition* of constituent statements. I should like to elaborate on what is meant by simplicity of composition scheme. To this end, let us select as example the repetitive scheme expressed as

```
while B do S                                              (13)
```

It specifies the repeated execution of the constituent statement S, while — at the outset of each repetition — condition B is satisfied. The simplicity consists in the ease with which we can infer properties about the while statement from known properties of the constituent statement. In particular, assume that we know that S leaves a property P on its variables unchanged or *invariant* whenever B is true initially; this may be expressed formally as

$$P \wedge B \ \{S\} \ P \tag{14a}$$

according to the notation introduced by Hoare [8]. Then we may infer that the while statement also leaves P invariant, regardless of the number of times S was repeated. Since the repetition process terminates only after condition B has become false, we may infer that in addition to P, also $\neg B$ holds after the execution of the while statement. This inference may be expressed formally as

$$P \ \{\textbf{while } B \textbf{ do } S\} \ P \wedge \neg B \tag{14b}$$

This formula contains the essence of the entire while-construct. It teaches us to look for an invariant property P, and to consider the result of the repetition to be the logical combination of P and the negation of the continuation condition B. A similar pattern of inference governs the repeat-construct used in the preceding examples. Assuming that we can prove

$$Q \vee (P \wedge \neg B) \ \{S\} \ P \tag{15a}$$

about S, then we may conclude that

$$Q \ \{\textbf{repeat } S \textbf{ until } B\} \ P \wedge B \tag{15b}$$

holds for the repeat-construct.

There remains the question, whether all programs can be expressed in terms of hierarchical nestings of the few elementary composition schemes mentioned. Although in principle this is possible, the question is rather, whether they can be expressed conveniently, and whether they *should* be expressed in such a manner. The answer must necessarily be subjective, a matter of taste, but I tend to answer affirmatively. At least an attempt should be made to stick to elementary schemes before using more elaborate ones. Yet, the temptation to rescind this rule is real, and the chance to succumb is particularly great in languages offering a facility like the goto statement, which allows the instantaneous invention of any form of composition, and which is the key to any kind of structural irregularity.

The following short example illustrates a typical situation, and the issues involved.

Example 3: Selecting distinct numbers

Given is a sequence of (not necessarily different) numbers $r_0, r_1, r_2, \cdots$. Select the first n distinct numbers and assign them sequentially to an array variable a with n elements, skipping any number r_i that had already occurred. (The sequence r may, for instance, be obtained from a pseudo-random number generator, and we can rest assured that the sequence r contains at least n different numbers.)

An obvious formulation of a program performing this task is the following:

```
for i := 1 to n do                                          (16)
begin L: get(r);
  for j := 1 to i-1 do
    if a[j] = r then goto L;
  a[i] := r
end
```

It cannot be denied that this "obvious" solution has been suggested by the tradition of expressing a repeated action by a for statement (or a DO loop). The task of computing a value for a is decomposed into n identical steps of computing a single number $a[i]$ for $i = 1 \cdots n$. Another influence leading to this formulation is the tacit assumption that the probability of two elements of the sequence being equal is reasonably small. Hence, the case of a candidate r being equal to some $a[j]$ is considered as the exception: it leads to a break in the orderly course of operations and is expressed by a jump. The elimination of this break is the subject of our further deliberations.

Of course, the goto-statement may be easily — almost mechanically — replaced in a transcription process leading to the following goto-less version.

```
for i := 1 to n do                                          (17)
begin
  repeat get(r); ok := true;
      j := 1;
    while (j<i) ∧ ok do
      begin ok := a[j] ≠ r; j := j+1
      end
  until ok;
  a[i] := r
end
```

The transcription consists of the replacement of the for statement with a fixed termination condition depending on the running index j by a more flexible while statement allowing for more complicated, composite termination (or rather continuation) conditions. But this solution appears quite unattractive. It is admittedly less transparent than the program using a jump, in spite of the fact that the most frequently heard objection to the use of jumps is that they

obscure the program. The other objection is that the goto-less version (17) requires more comparisons and tests, and hence is less efficient.

The crux of the matter is that well-structured programs should not be obtained merely through the formalistic process of eliminating goto statements from programs that were conceived with that facility in mind, but that they must emerge from a proper design process. Two alternative solutions are presented here as illustrations.

In the first case, we abandon the notion that the program must necessarily be based on the statement

```
for i := 1 to n do                                   (18)
  a[i] := the next suitable number
```

and consider the basic iteration step to consist of the generation of the next element of the sequence r, followed by the test for its acceptability.

```
i := 1;                                              (19)
while i ≤ n do
begin generate next r;
  assign it to a[i];
  check whether all a[j] are different from a[i];
  if so, proceed by incrementing i
end
```

This form makes it obvious that we are in trouble, if the sequence r should be such that i cannot be incremented any longer. Written in terms of our programming language, (19) becomes

```
i := 1;                                              (20)
while i ≤ n do
begin get(r);
  a[i] := r; j := 1;
  while a[j] ≠ r do j := j+1;
  if i = j then i := i+1
end
```

The second approach to this problem retains the basic concept of the solution as shown in (18). From there, its development is characterized by the following two snapshots:

```
for i := 1 to n do                                   (21)
  repeat generate the next r;
    check its acceptability
  until acceptable
```

```
for i := 1 to n do                                   (22)
  repeat get(r);
    a[i] := r; j := 1;
    while a[j] ≠ r do j := j+1;
  until i = j
```

In contrast to (20), this solution consists of *three* nested repetitions instead of only two, and therefore seems inferior at first sight. In fact, however, solution (22) turns out to be even more economical. The reason is that in (20) the test for continuation $i \leq n$ is actually unnecessary whenever $i \neq j$, since $i \neq j$ in this case implies $i < j$, and because i has not been altered since the last evaluation of $i \leq n$. Of course, program (22) is considerably more efficient than the original form with a jump (16).

This terminates our consideration of Example 3.

The question remains open, of course, whether jumps can *always* be avoided without disadvantage. I shall not venture to answer this question, particularly because the term "disadvantage" is sufficiently vague to admit many interpretations. But there is evidence of the existence of some characteristic and reasonably frequent situations which are expressed only with difficulty in terms of the language construct introduced above. A particular case is the *loop with exit(s) in the middle*. Lately it has led language designers to introduce specific constructs mirroring this case [12]. It turns out, however, that it is most difficult to find a satisfactory and linguistically suggestive formulation, and that sometimes solutions are invented that seem to merely replace the symbol goto by another word, such as exit or jump. For example, the construct

```
loop S1;                                             (23)
  exit if P;
    S2
end
```

with the parametric statements $S1$, $S2$, and the termination condition P might be adopted to express the program

```
L1: begin S1;                                        (24)
      if P then goto L2;
      S2; goto L1
L2: end
```

in a more concise and goto-free form.

Expressing (24) in terms of the basic repetitive statement forms does, indeed, often lead to undesirable complications, such as unnecessary reevaluation of conditions, or duplication of parts of the program, as is shown by the two proposals (25) and (26).

```
repeat S1;                                           (25)
  if ¬P then S2
until P
```

```
S1;                                                  (26)
while ¬P do
  begin S2; S1 end
```

Loop structures

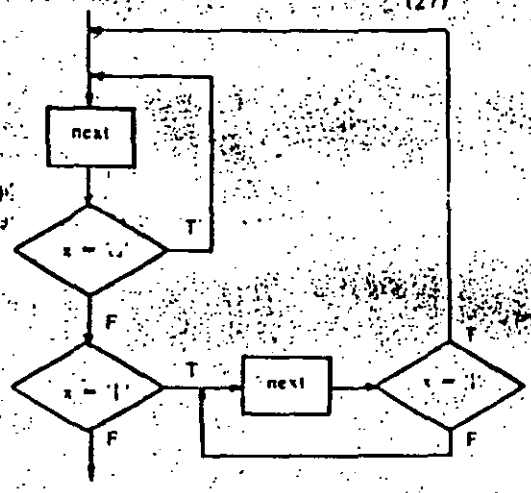
The following, and last two, examples of problems are added to show that often the need for an *exit in the middle* construct is based on a preconceived notion rather than on a real necessity, and that sometimes an even better solution is found when sticking to the fundamental constructs.

Example 4: A scanner

The task is to construct a piece of program which, each time it is activated, scans an input sequence of characters, delivering as a result the next character, but skipping over blanks and over so-called comments. A comment is defined as any sequence of characters starting with a left bracket and ending with a right bracket.

This scanner could typically occur as part of a compiler. A common solution is indicated by the following flowchart (*next* denotes the operation of reading the next character and assigning it to the result variable *x*).

(27)



This program clearly exhibits the loop structure with exit in the middle, satisfying the one-entry-one-exit prerequisite. Instead of proposing a suggestive form for this construct in sequential language, however, let me tackle the posed problem in a different manner. Recognizing the main purpose of the program as being the reading of the next character, with the additional request for skipping over blanks and comments, I propose a first version as follows:

```
next;                                                    (28)
while x in ['␣', '['] do
  "skip blanks and comments"
```

The correctness of this program is easily established, assuming that the statement in quotes performs what it says, and nothing more. The definition of the while statement guarantees that the resulting value of *x* is neither a blank nor a comment, no matter in what way blanks and comments are skipped.

The refinement of the statement in quotes is guided by the fact that upon its initiation *x* is either a blank or an opening bracket.

```
if x = '␣' then next else                                (29)
  "skip comment"
```

where the last statement is expressed, with obvious reasoning, as

```
begin repeat next until x = ']';                         (30)
  next
end
```

Only knowledge about the expected *frequencies of occurrence* of individual characters can be a reason to choose another form of this program on the grounds of efficiency. For the sake of argument, let us assume that short sequences of blanks are particularly frequent and that, on the other hand, immediately adjacent comments are extremely rare. This leads us to an equally correct alternative form of (29), namely

```
"skip consecutive blanks, if any";                       (31)
"skip comment, if any"
```

The first of the two statements is readily expressed as

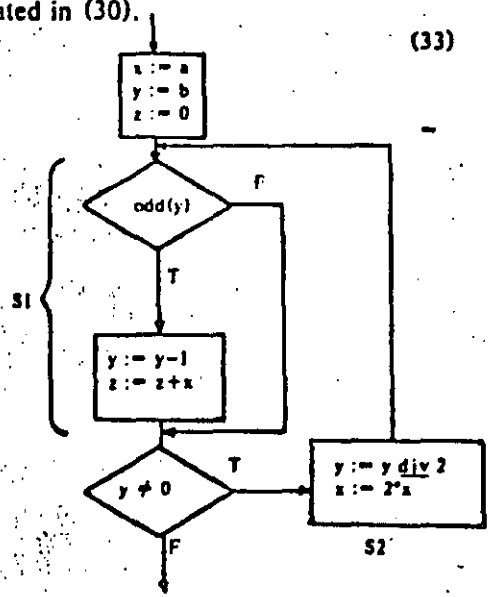
```
while x = '␣' do next                                    (32)
```

whereas the second is already elaborated in (30).

(33)

Example 5: Integer multiplication

Assume that we are to design a program to multiply two non-negative integers *a* and *b* with the use of addition, doubling, and halving only. Let the result be represented by a variable *z*. A well-known and efficient method is shown by the flowchart at right:



This program, once again, clearly exhibits the loop structure with exit in the middle, and therefore cannot be expressed as a single while statement. It is usually squeezed into the simple loop form by displacing the loop termination test, positioning it in front of statement *S1*. The program then obtains the well-known form

```
x := a; y := b; z := 0;                                   (34)
while y ≠ 0 do
  begin if odd(y) then
    begin y := y−1; z := z+x
    end;
    y := y div 2; x := 2•x
  end
```

This clearly does not change the effect of the program, because if $y = 0$ at entry to *S1*, then *S1* has no effect and, in particular, leaves y unchanged; and if $y \neq 0$, then the only additional effect incurred by the modified version is on the auxiliary variables x and y in the case of $y = 1$. But this additional effect is quite undesirable, not so much because of the additional, superfluous, and useless computation, but because this operation may be harmful by causing overflow of the arithmetic unit. Should we therefore resort to the exit-in-the-middle version?

A different solution was shown to me by E.W. Dijkstra. He proposed to tackle the problem at its roots, instead of trying to remedy a preconceived proposal. The most obvious multiplication algorithm under the stated constraints is the following:

```
x := a; y := b; z := 0;                                   (35)
while y ≠ 0 do
  begin {y > 0 and x•y+z = a•b}
    y := y−1; z := z+x
  end
```

Before we start out trying to improve this version, we observe that at the outset of each repetition two conditions are satisfied.

1. $y > 0$ follows from the fact that y is a non-negative integer and not equal to zero.
2. $x \cdot y + z = a \cdot b$ is invariant under the two repeated assignments. (To verify this claim, substitute $y-1$ for y and $z+x$ for z; this yields $x \cdot (y-1) + (z+x) = x \cdot y + z = a \cdot b$, i.e., the original equation.) At entry the equation is satisfied, since $z = 0$, $x = a$, $y = b$.

Note that the invariant equation combined with the negation of the continuation condition yields $(y = 0)$ and $(x \cdot y + z = a \cdot b)$, i.e., the desired result $z = a \cdot b$.

If we now insert any statement at the place of the invariant which leaves the product $x \cdot y$ unchanged, the result of the program will evidently remain the same. Such a statement is, e.g., the pair of assignments

```
y := y div 2; x := 2•x                                    (36)
```

under the condition that y is even. But if a relation is invariant over a statement, it remains so regardless of how often the statement is executed. This suggests the following, quite evidently correct, efficient, and elegant solution. It contains no exit-in-the-middle loop.

```
x := a; y := b; z := 0;                                   (37)
while y ≠ 0 do
  begin {y > 0 and x•y+z = a•b}
    while even(y) do
      begin y := y div 2; x := 2•x
      end;
    y := y−1; z := z+x
  end
```

So much for examples, whose purpose was to sketch and elucidate the basic ideas behind the methods of structured programming and stepwise refinement.

Conclusions

Skeptics will, of course, doubt that these methods represent any progress over the techniques of the old days — in fact, that they are *methods* at all. I can merely say that in my own experience, the new approach has improved my attitudes and abilities towards programming very considerably, and the experiences of others confirm this impression [10, 11]. A systematic, orderly, and transparent approach is mandatory in any sizable project nowadays, not only to make it work properly, but also to keep the programming cost within reasonable bounds. It is the very fact that computation has become very cheap in contrast with salaries of programmers, that squeezing the machines to yield their utmost in speed has become much less important than reliability, correctness, and organizational clarity. It is not only more urgent, but also much more costly to correct an efficient, but erroneous program, than to speed up a relatively slow, but correct program. In the past, the debugging phase has taken a ridiculously large percentage of the development cost in most large projects. The aim now is to eliminate the necessity of debugging by creating bug-free products in the first place. Doesn't this bring to mind the medical slogan "prevention is better than healing"!

The criticism has been voiced that the method of structured programming is in essence nothing more than programming by painstakingly avoiding the use of jumps (goto statements). One may, indeed, come to this conclusion by looking at the entire issue in the reverse direction. But in fact, the method of stepwise decomposition and refinement of the programming task automatically leads to goto-free programs; the absence of jumps is not the initial aim, but the final outcome of the exercise. The claim that structured programming was invented by proving that all programs can be formulated without goto statements is therefore based on a fundamental misunderstanding.

The question of whether jumps enter the picture or not is basically a matter of the level of decomposition or refinement to which the programming process is carried. Ultimately — that is in machine code — there can be no doubt about the presence of jump instructions. The moral of the story is that jumps must not be used in the initial conception of a general algorithmic strategy, and in fact should be delayed as long as possible. With today's state of technology, the introduction of jump instructions can be left to compilers of languages that offer adequate, judiciously chosen, disciplined structuring facilities.

One of the essential facilities for this purpose, besides conditional and repetitive statements, is the *recursive procedure*. In many cases it emerges as the natural formulation of a solution, such as, for instance, in most cases of backtracking algorithms. Hardly anywhere else can a natural, concise, and often self-explanatory solution be made more obscure and mystifying than by replacing its recursive formulation by one in terms of repetition and — well — jumps. This process should definitely be left to a compiler, as it concerns what is called *coding* rather than programming (code = system of symbols used in ciphers, secret messages, etc. [Webster]). Modern programming systems, however, offer efficient implementations of recursion, and thereby make "programming around recursion" a largely unnecessary exercise.

Whereas a teacher should not and must not pay attention to "percent issues" as to efficiency while explaining and exemplifying methods of composing well-structured programs, a professional programmer may well be forced to do so. He may sometimes find a dogma of sticking exclusively to a restricted set of program structuring schemas too much of a straight-jacket, and the temptation to break out too powerful. This will be the case as long as compilers are insufficiently sophisticated to take full advantage of disciplined structuring. Naturally, there will always be situations where a compiler is either denied the full information needed for successful code optimization, or where it would be unable to infer the necessary conditions. It is therefore entirely possible that in the future a more interactive mode of operation between compiler and programmer will emerge, at least for the very sophisticated professional. The purpose of this interaction would not, however, be the development of an algorithm or the debugging of a program, but rather its *improvement under invariance of correctness*.

The foregoing discussion also implies an answer to the question of whether structured programming in an unstructured language (such as FORTRAN) is possible. It is not. What is possible, however, is structured programming in a "higher level" language and subsequent hand-translation into the unstructured language. The corollary is that whereas this approach may be practicable with the almost superhuman discipline of a compiler, it is highly unsuited for *teaching* programming. Recognizing that there may be valid economic reasons for learning *coding* in, say, FORTRAN, the use of an unstructured language to teach *programming* — as the art of systematically developing algorithms — can no longer be defended in the context of computer science education. The lack of an adequate modern tool on the available computing facility is the only remaining excuse.

The last remark concerns an aspect of "structured programming" that has not been illuminated by the foregoing examples: structuring considerations of program and data are often closely related. Hence, it is only natural to subject also the specification of data to a process of stepwise refinement. Moreover, this process is naturally carried out simultaneously with the refinement of the program. A language must, therefore, not only offer program structuring facilities, but an adequate set of systematic data structuring facilities as well. An example of this direction of language development is the programming language PASCAL [12, 13]. The importance of this aspect of programming is particularly evident, as we recognize the data as the ultimate object of our interest: they represent the arguments and results of all computing processes. Only structure enables the programmer to recognize meaning in the computed information.

Acknowledgment

The author is grateful to P.J. Denning for kindly posing the problem treated in Example 3.

1. E.W. Dijkstra, "Some Meditations on Advanced Programming," *Proceedings of the 1962 IFIP Congress* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1963), pp. 535-38.

2. _____, "The Humble Programmer," *Communications of the ACM*, Vol. 15, No. 10 (October 1972), pp. 859-66.

3. _____, "Notes on Structured Programming," *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (New York: Academic Press, 1972).

4. P. Naur, B. Randell, and J.N. Buxton, eds., *Software Engineering, Concepts and Techniques, Proceedings of the NATO Conferences* (New York: Petrocelli/Charter, 1976).

5. N. Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM*, Vol. 14, No. 4 (April 1971), pp. 221-27.

6. _____, *Systematic Programming* (Englewood Cliffs, N.J.: Prentice-Hall, 1973).

7. P. Naur, "Proof of Algorithms by General Snapshots," *BIT*, Vol. 6, No. 4 (1966), pp. 310-16.

8. C.A.R. Hoare, "An Axiomatic Approach to Computer Programming," *Communications of the ACM*, Vol. 12, No. 10 (October 1969), pp. 576-80, 583.

9. W.A. Wulf, "Programming Without the GOTO," *Proceedings of the 1971 IFIP Congress*, Vol. 1 (Amsterdam, The Netherlands: North-Holland Publishing Co., 1972), pp. 408-13.

10. F.T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, Vol. 11, No. 1 (January 1972), pp. 56-73.

11. U. Ammann, "The Method of Structured Programming Applied to the Development of a Compiler," *Proceedings of the 1973 International Computing Symposium* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1974), pp. 93-100.

12. N. Wirth, "The Programming Language Pascal," *Acta Informatica*, Vol. 1, No. 1 (1971), pp. 35-63.

13. K. Jensen and N. Wirth, "PASCAL — User Manual and Report," *Lecture Notes in Computer Science*, Vol. 18 (New York: Springer-Verlag, 1974).

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

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

PROGRAMACION ESTRUCTURADA

FIS. RAYMUNDO H. RANGEL

MAYO, 1985

SECUENCIA

CONCATENACION

DECISION

A) DECISION CON SALIDA MULTIPLE

```
SI ( P_1 ) LUEGO
     SEUDOCODIGO-1
OSI ( P-2 ) LUEGO
     SEUDOCODIGO-2
      :
OSI (P-N) LUEGO
     SEUDOCODIGO-N
OBIEN
     SEUDOCODIGO-X
FINSI
```

B) DECISION CON 2 SALIDAS

```
SI ( P ) LUEGO
    SEUDOCODIGO-1
OBIEN
    SEUDOCODIGO-2
FINSI
```

C) DECISION CON UNA SALIDA

```
SI (P) LUEGO
    SEUDOCODIGO
FINSI
```

D) DECISION CON MULTIPLE SALIDA (ALTERNA)

```
SELECCIONAR (CASO) DE
(CASO 1)
        SEUDOCODIGO-1
(CASO 2)
        SEUDOCODIGO-2
          .
          .
          .
(CASO n)
        SEUDOCODIGO-n
ORIEN
        SEUDOCODIGO-X
FIN SELECCIONAR
```

ITERATIVAS

A)

```
ENTANTO (P) REPETIR
    SEUDOCODIGO
FIN ENTANTO
```

B)

```
REPETIR
    SEUDOCODIGO
HASTA (P)
```

- PROGRAMA ESTRUCTURADO PROPIO

ES CUALQUIER PROGRAMA DERIVADO MEDIANTE UN PROCESO DE REFINAMIENTO A PASOS QUE SE FORMULA AL USAR LAS 3 ESTRUCTURAS BASICAS DE CONTROL : SECUENCIA, SELECCION E ITERATIVAS

UN PUNTO DE ENTRADA UN PUNTO DE SALIDA

- PROGRAMA ESTRUCTURADO BIEN FORMADO

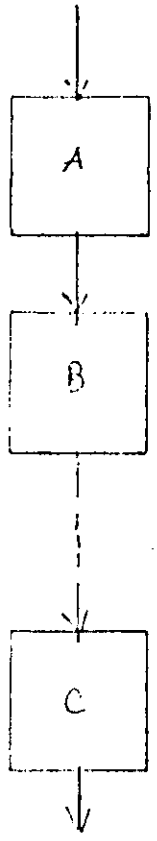
ES UN PROGRAMA DERIVADO MEDIANTE EL PROCESO DE REFINAMIENTO A PASOS QUE SE FORMULA, ADEMAS DE LAS ESTRUCTURAS BASICAS DE CONTROL DE SECUENCIA, SELECCION E ITERATIVAS, LAS ESTRUCTURAS DE CONTROL DE SALIDA.

UN PUNTO DE ENTRADA Y VARIOS PUNTOS DE SALIDA

DIAGRAMAS DE ESTRUCTURA

DIAGRAMAS ESTRUCTURADOS DE FLUJO

SECUENCIA

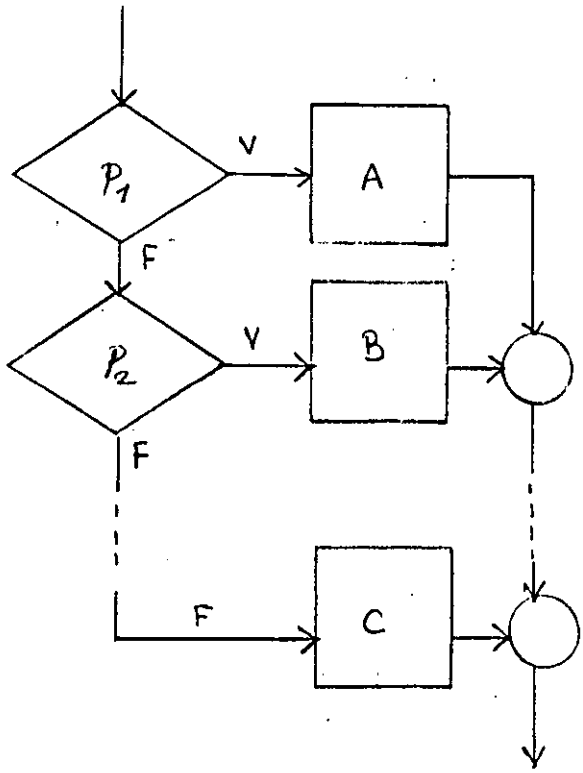


CODIGO A

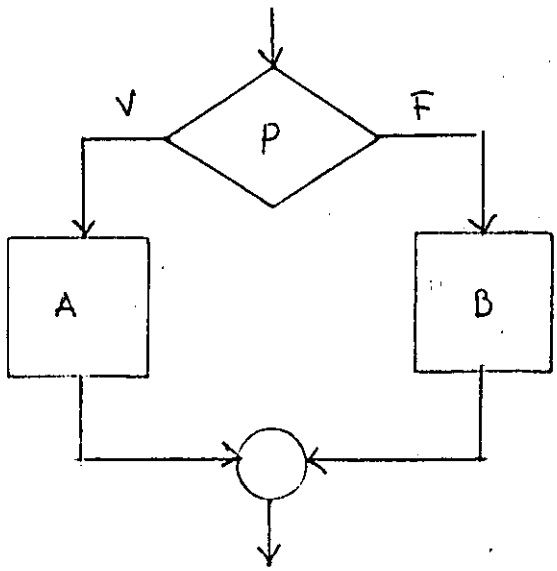
CODIGO B

CODIGO C

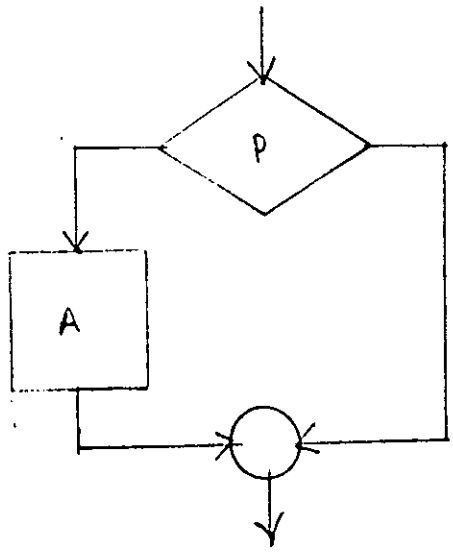
SELECCION



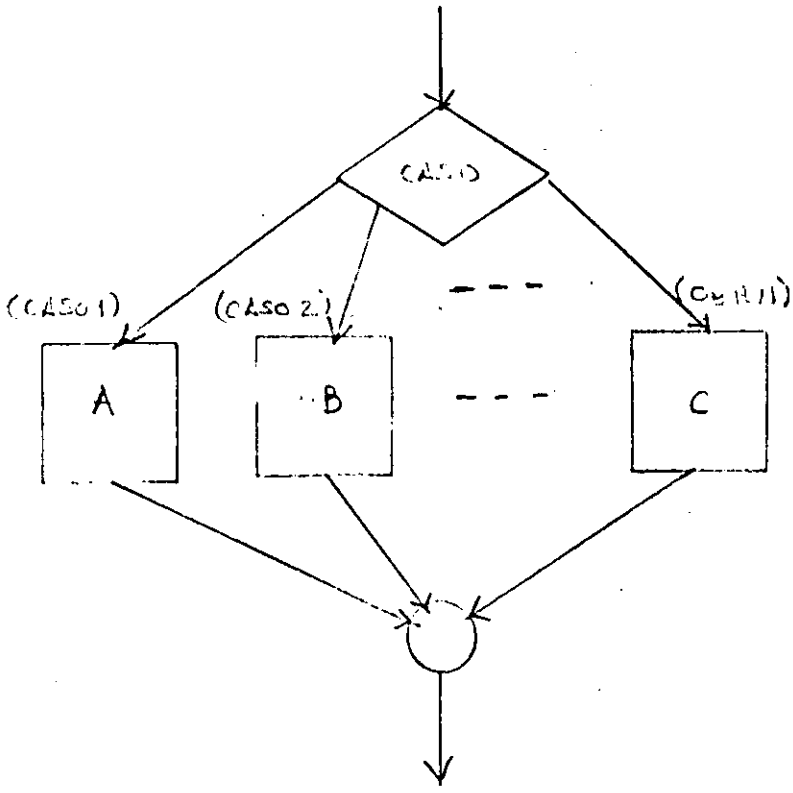

```
SI (P1) LUEGO
    CODIGO A
OSI (P2) LUEGO
    CODIGO B
    :
OBIEN
    CODIGO C
FINSI
```




```
SI (P) LUEGO
    CODIGO A
OBIEN
    CODIGO B
FINSI
```

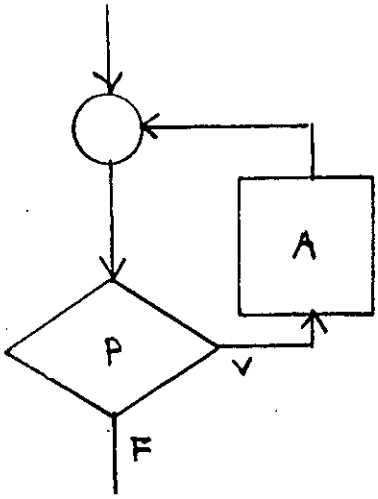


SI (P) LUEGO
    CODIGO A
FINSI



SELECCIONAR (CASO) DE
(CASO 1)
        CODIGO A
(CASO 2)
        CODIGO B
        :
OBIEN
        CODIGO C
FINSELECCIONAR

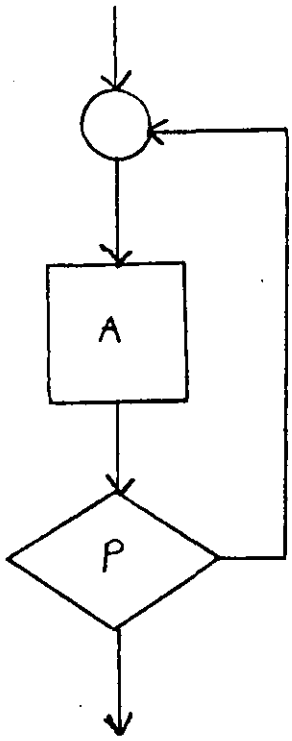
REPETICION



ENTANTO (P) REPETIR

CODIGO A

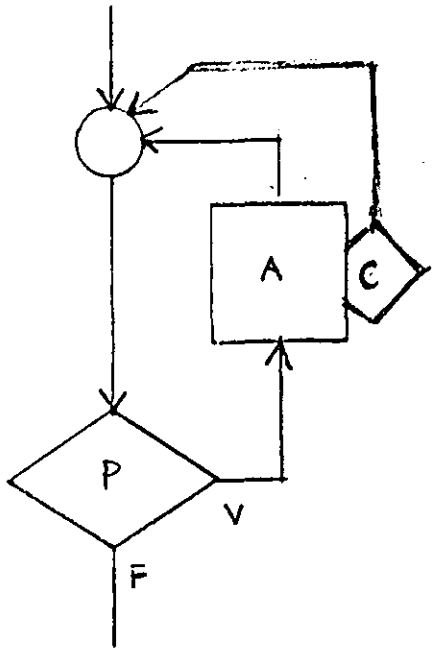
FIN ENTANTO



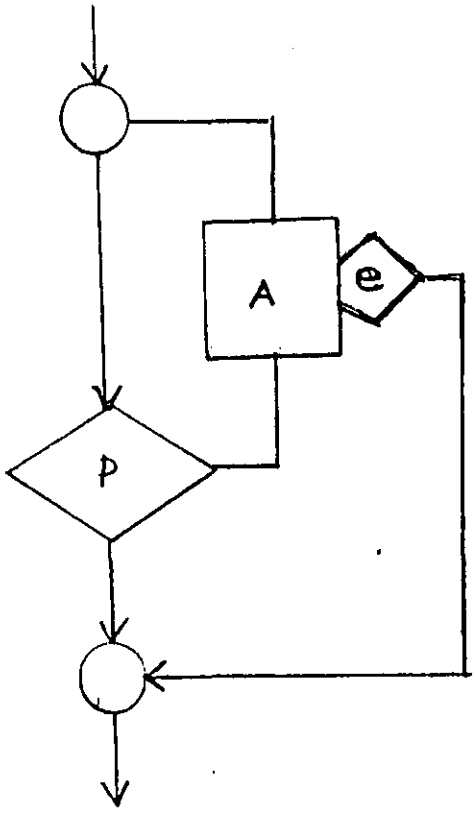
HASTA

CODIGO A

REPETIR (P)



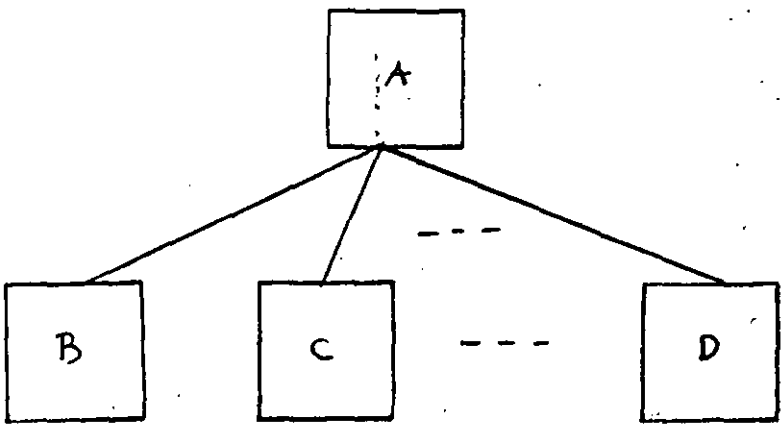
CICLO



ESCAPE

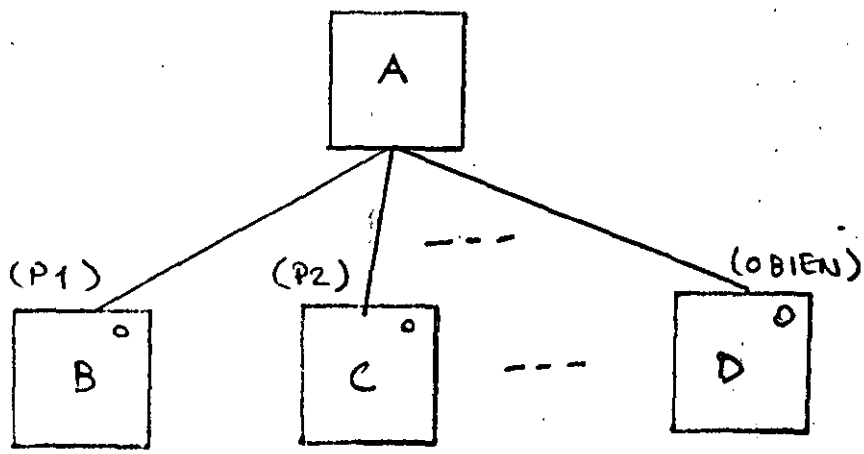
LOGICA ESQUEMATICA

SECUENCIA

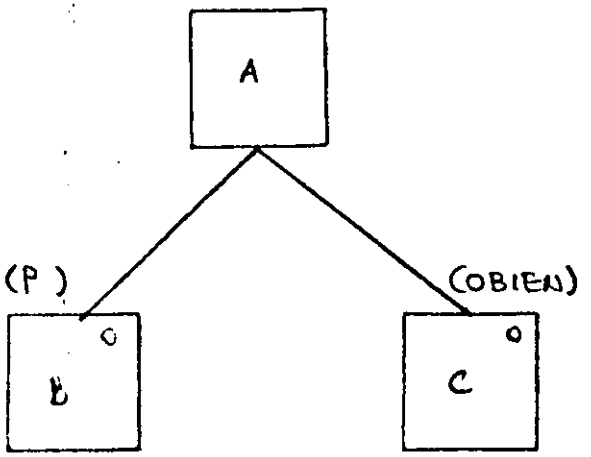


A CODIGO B
CODIGO C
⋮
CODIGO D

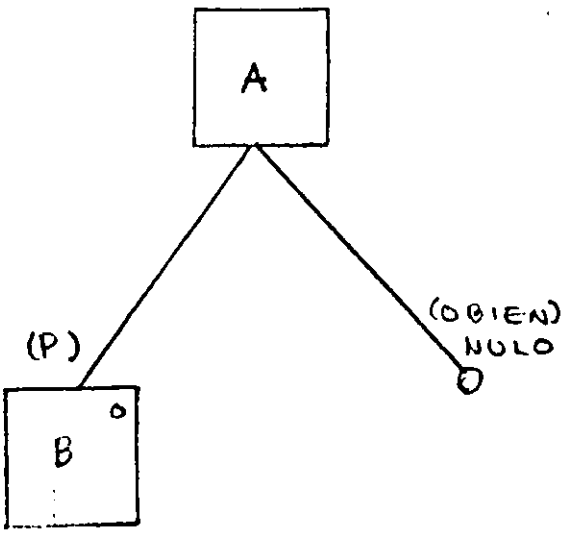
SELECCION



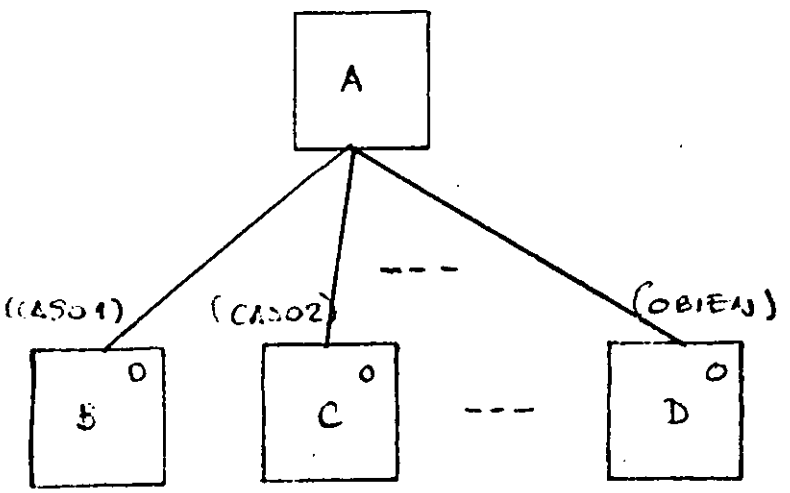
A SI (P1) LUEGO
CODIGO B
OSI (P2) LUEGO
CODIGO C
⋮
OBIEN
CODIGO D
FIN SI.




```
A    SI (P) LUEGO
        CODIGO B
     OBIEN
        CODIGO C
     FINSI
```

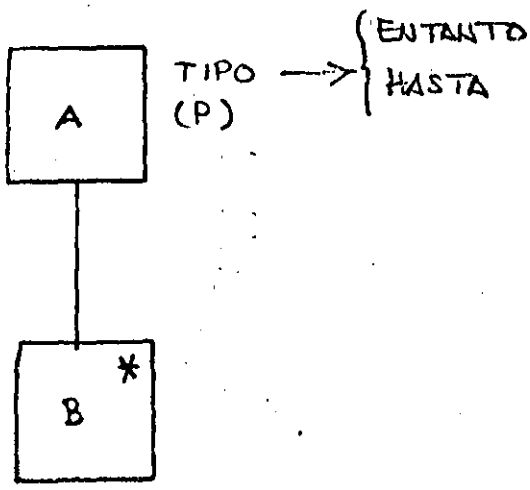



```
A    SI(P) LUEGO
        CODIGO A
     FINSI
```




```
A    SELECCIONAR (CASO) DE
     (CASO 1)
          CODIGO B
     (CASO 2)
          CODIGO C
          :
     (OBIEN)
          CODIGO D
     FIN SELECCIONAR
```

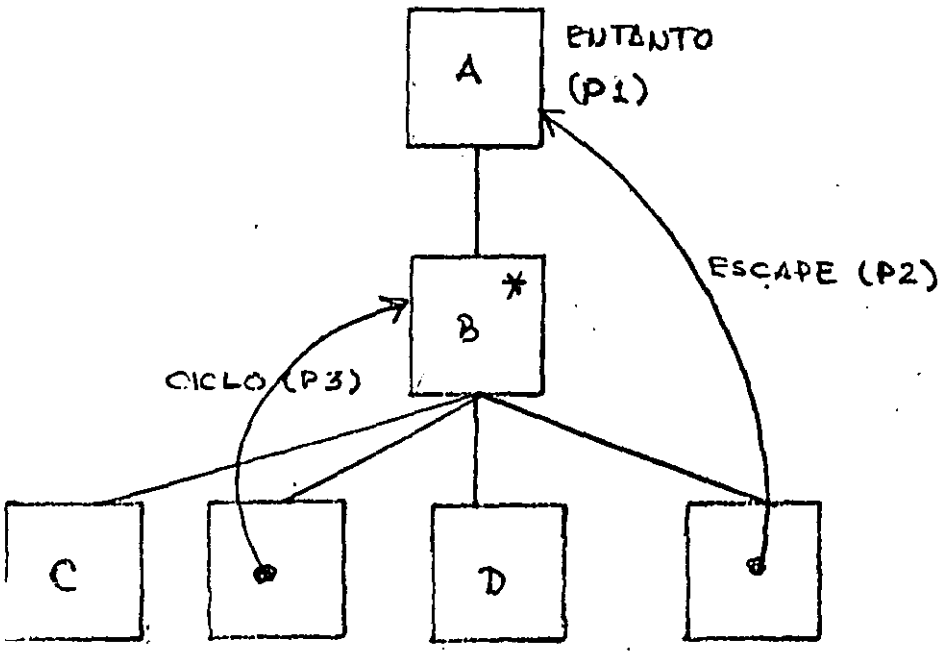
REPETICION



A ENTANTO (P) REPETIR
CODIGO B
FIN ENTANTO

A REPETIR
CODIGO B
HASTA (P)

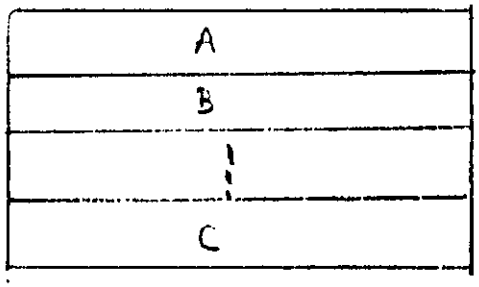
SALIDA



A ENTANTO (P1) REPETIR
B CODIGO C
SI (P3) LUEGO
CICLO A
FINSI
CODIGO D
SI (P2) LUEGO
ESCAPE A
FINSI
FIN ENTANTO

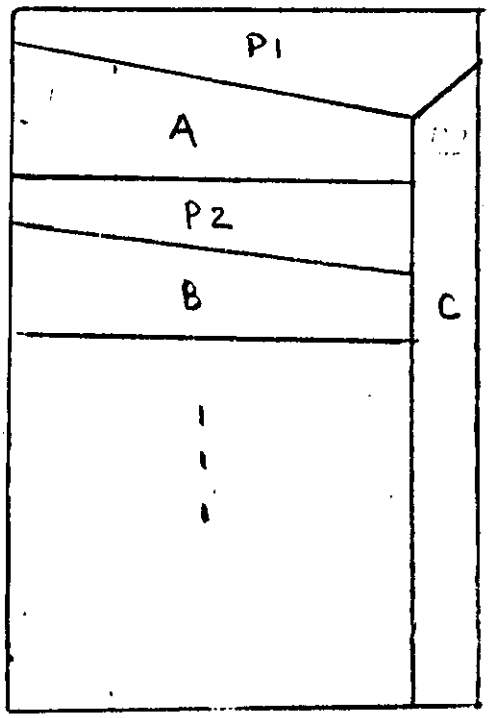
CARTAS DE NASSI-SHNEIDERMAN (CARTAS DE CHAPIN)

SECUENCIA

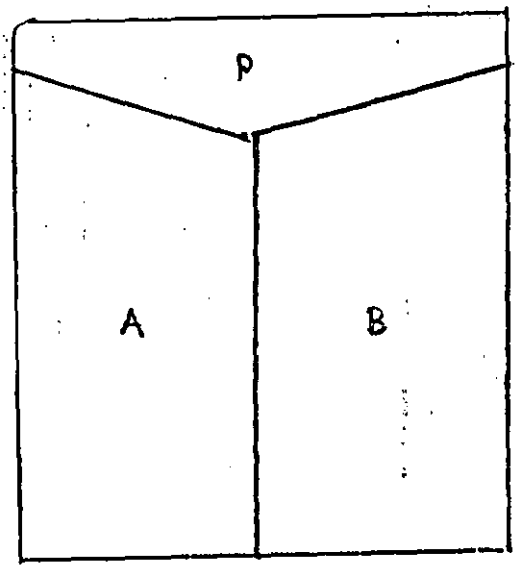



```
CODIGO A
CODIGO B
   :
CODIGO C
```

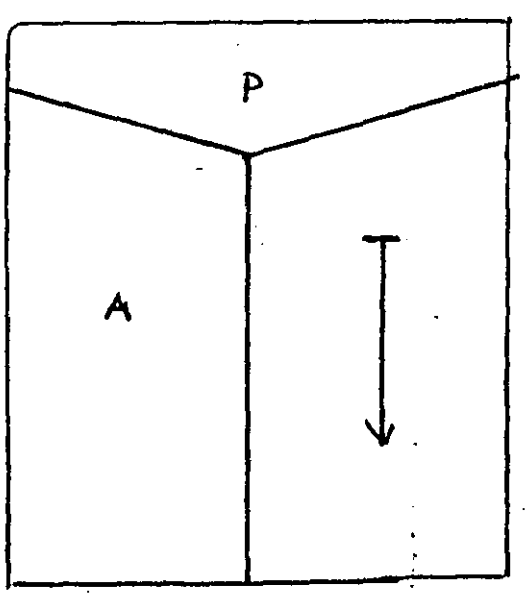
SELECCION

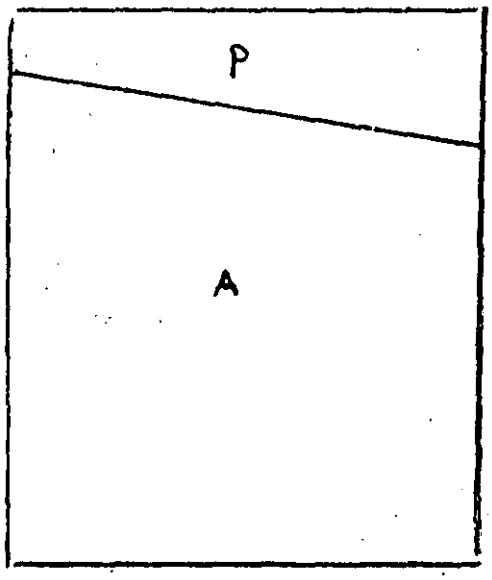



```
SI( P1) LUEGO
     CODIGO A
OSI (P2) LUEGO
     CODIGO B
        :
OBIEN
     CODIGO C
FINSI
```

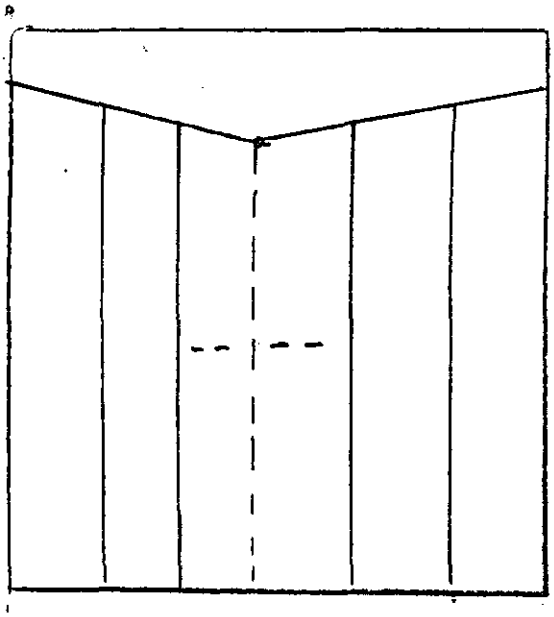



```
SI (P) LUEGO
    CODIGO A
OBIEN
    CODIGO B
FINSI
```

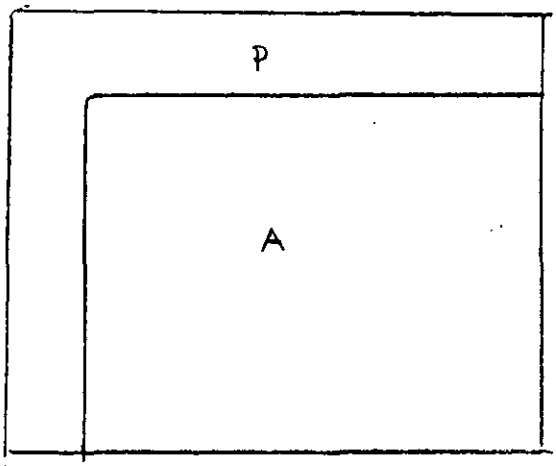




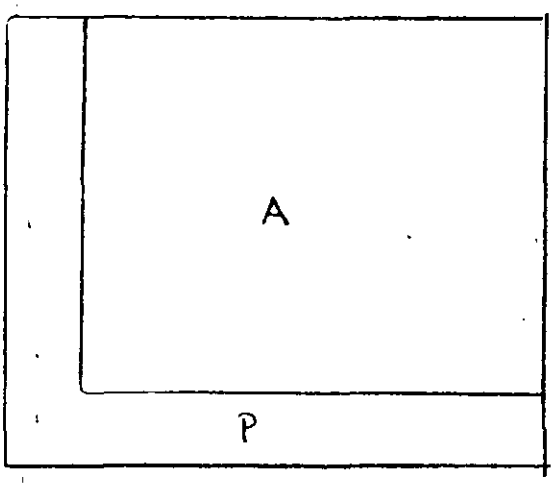

```
SI (P) LUEGO
    CODIGO A
FINSI
```



REPETICION

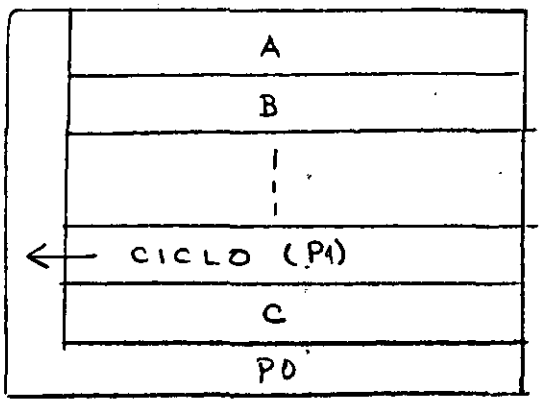


ENTANTO (P) REPETIR
CODIGO A
FINENTANTO

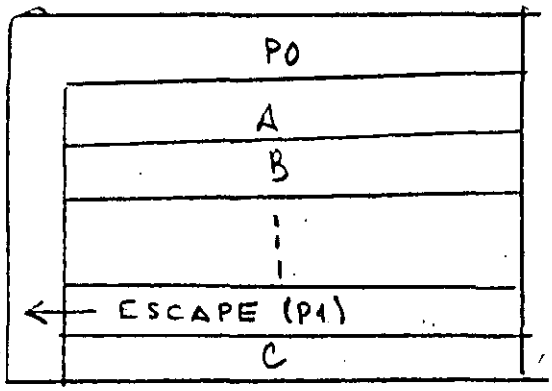


REPETIR
CODIGO A
HASTA (P)

SALIDA



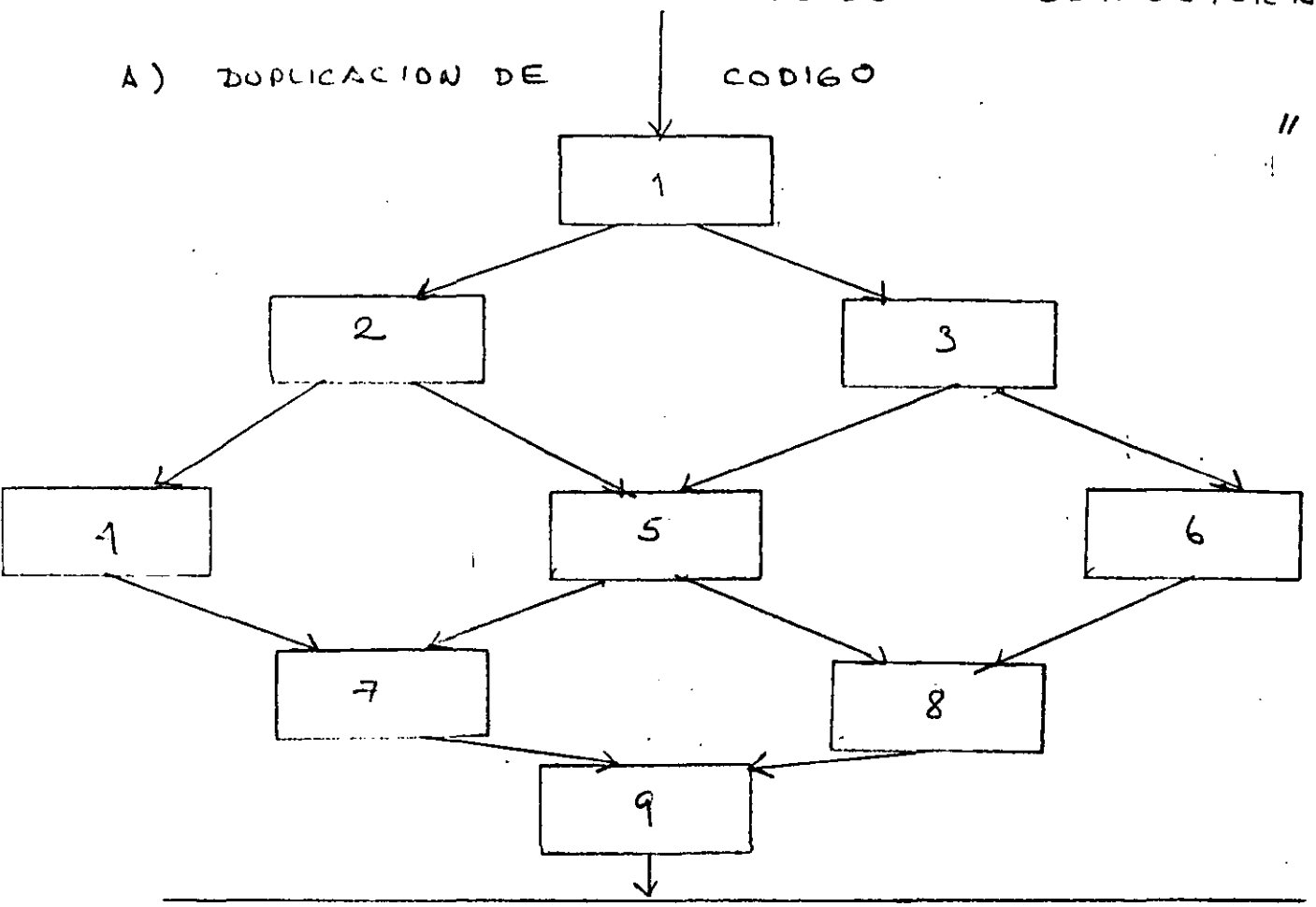

```
REPETIR
    CODIGO A
    CODIGO B
      .
      .
    SI (P1) LUEGO
        CICLO
    FINSI
    CODIGO C
HASTA (PO)
```

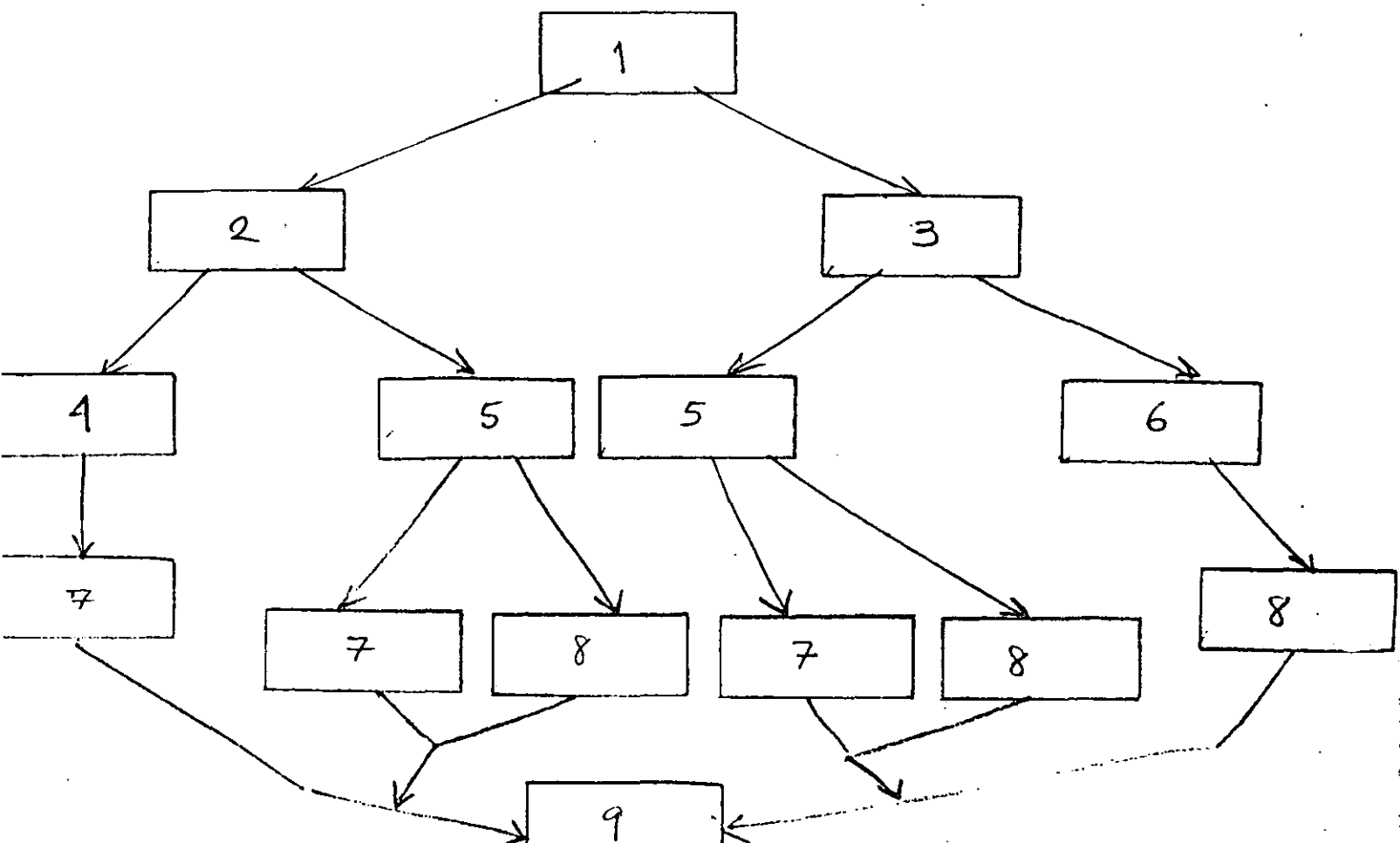



```
MIENTRAS (PO) REPETIR
    CODIGO A
    CODIGO B
      .
      .
    SI (P1) LUEGO
        ESCAPE
    FINSI
    CODIGO C
FINMIENTRAS.
```

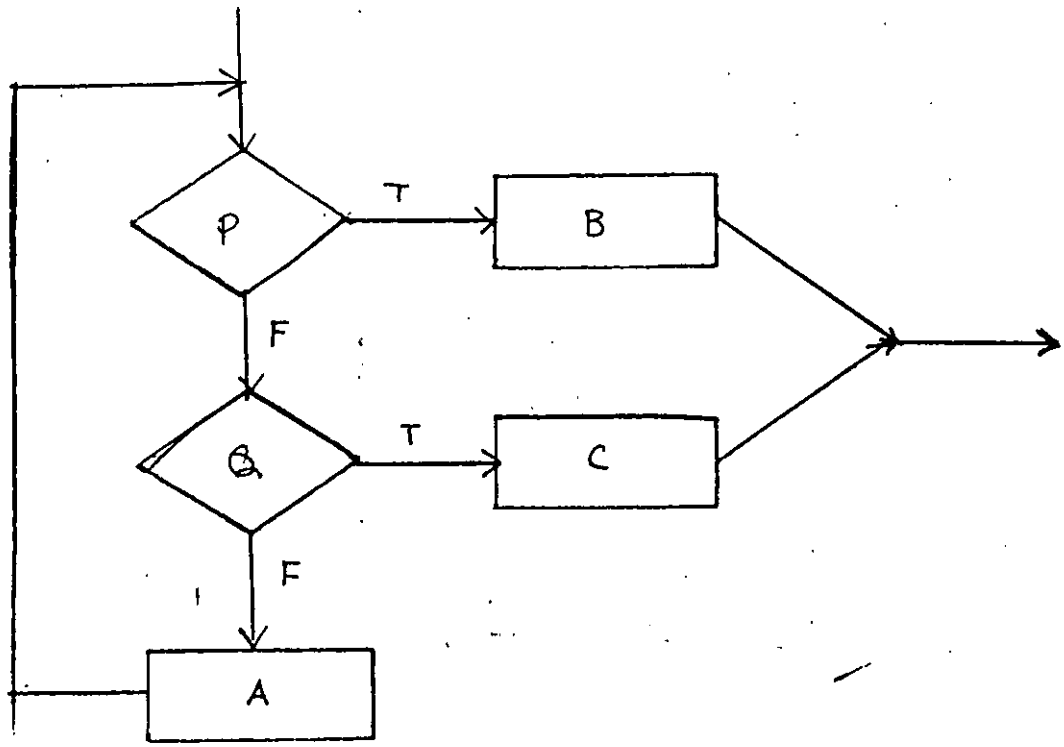
ESTRUCTURACION DE CODIGO NO ESTRUCTURADO

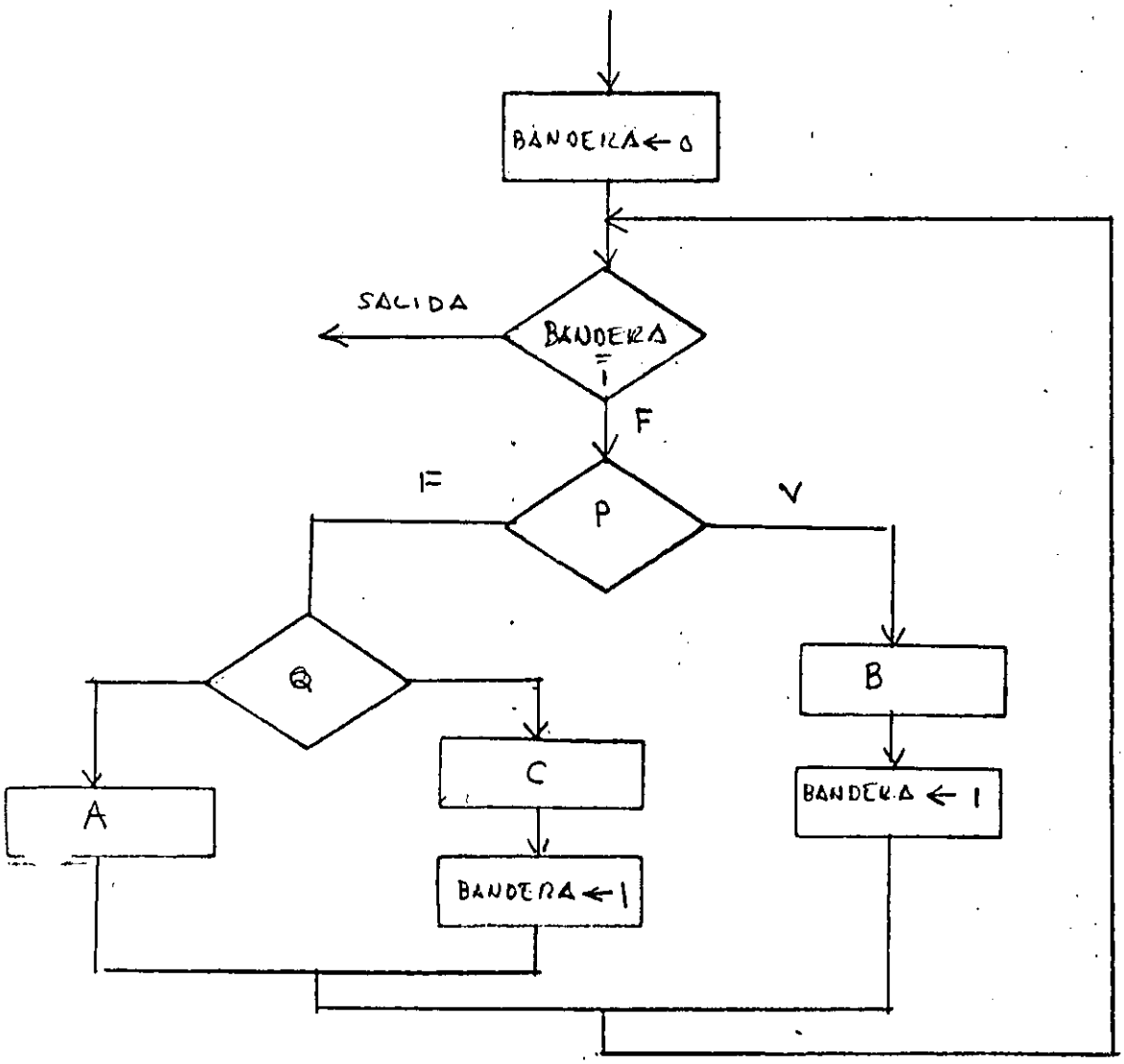
A) DUPLICACION DE CODIGO



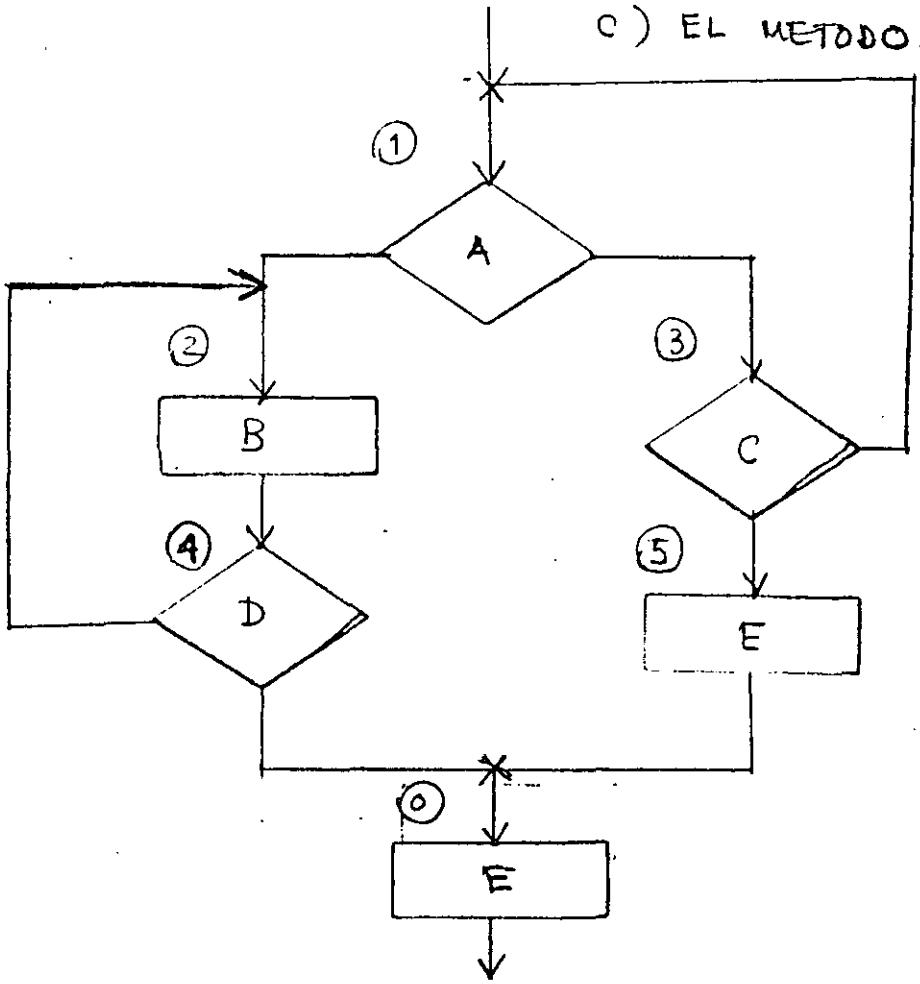


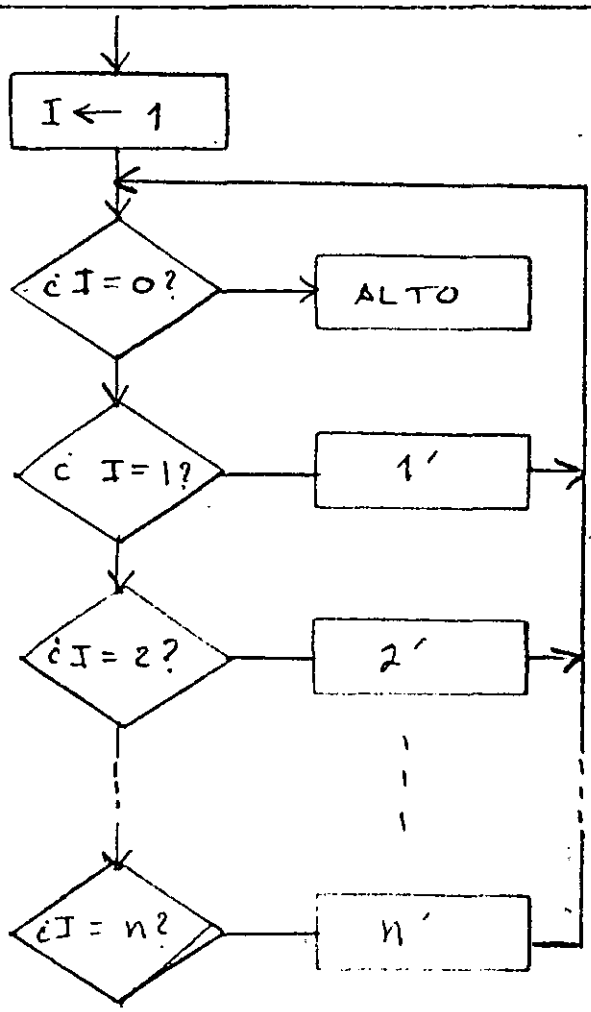
B) EL METODO DE LA BANDERA BOOLEANA



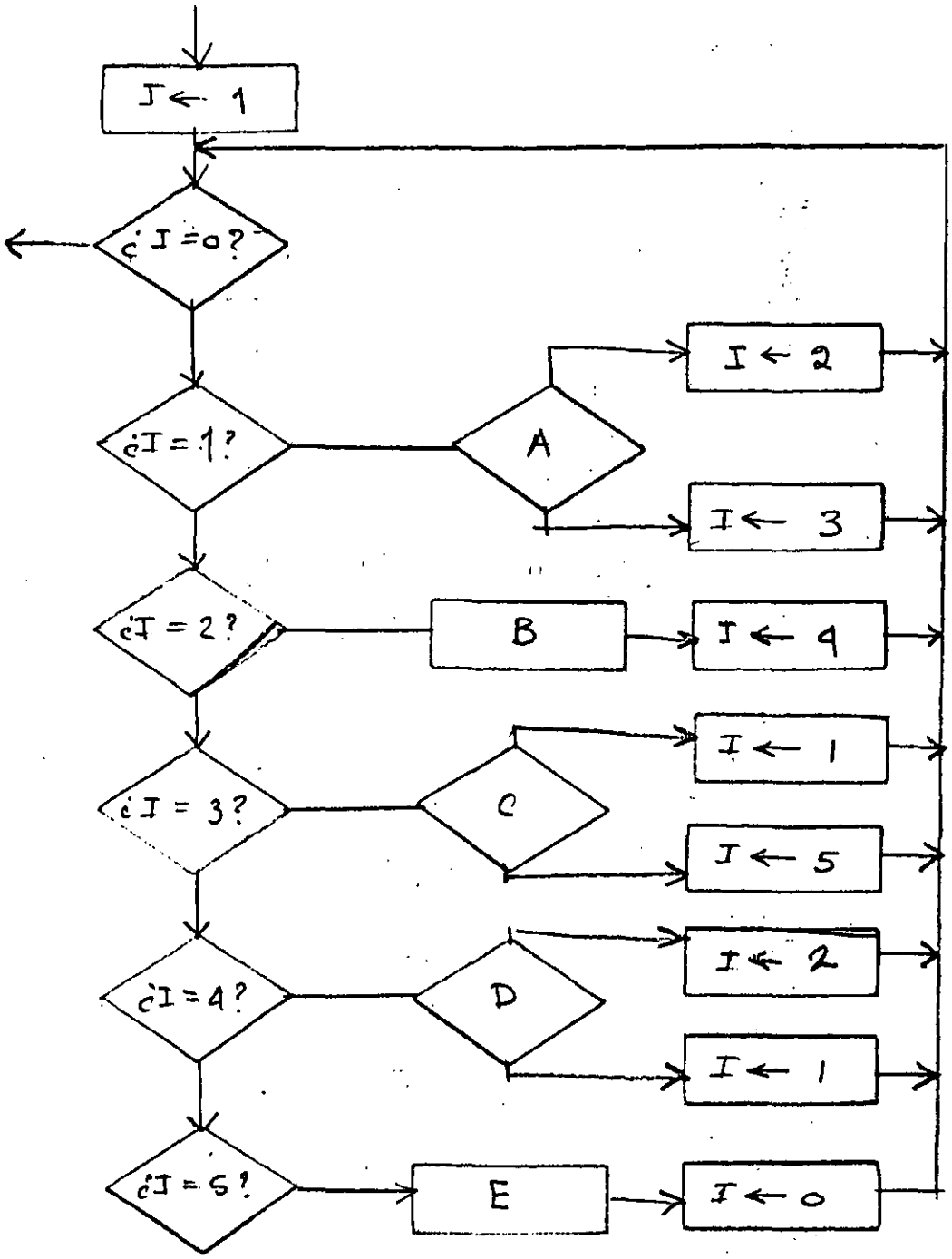


c) EL METODO DE LA VARIABLE DE ESTADO





A LA VUELTA ⟶


I ← 1
¿I = 0?
¿I = 1?
A
I ← 2
I ← 3
¿I = 2?
B
I ← 4
¿I = 3?
C
I ← 1
I ← 5
¿I = 4?
D
I ← 2
I ← 1
¿I = 5?
E
I ← 0

Programming Standards 11

Introduction

The use of standards should achieve a system that has a uniform structure and interfaces, and makes consistent use of software techniques (cf. Part I, Section 4.3).

The following is an extract from existing guidelines which were drawn up for a large number of personnel and apply to many data-processing departments.

The chief programmer is responsible for ensuring that the guidelines are observed. Random checks may also be made by a higher authority.

11.1 Programming Languages

COBOL

ANS-COBOL should, in principle, be *the* programming language used for commercial applications.

Assembler

Assembler should be used only for problems that are impossible to formulate using COBOL, which would require a great effort to formulate using COBOL, that involve critical timings or that must satisfy particular requirements because of the way they are implemented (e.g. standard software).

Approval must be given for the use of Assembler. When using Assembler, the same nomenclature rules apply, so far as possible, as for COBOL. Registers and constants should be given symbolically (using EQU).

11.2 Structuring

Structuring conventions for COBOL programs:

(1) The program organization is strictly defined.
(2) Upper levels should contain mainly control functions.

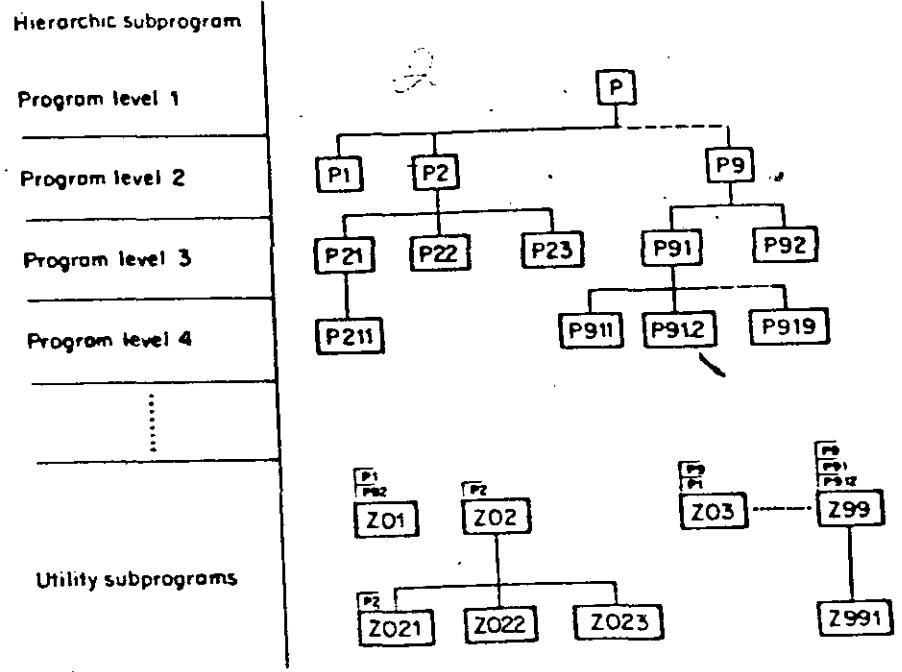


Figure 11.1 Program organization

(3) There should be only one STOP RUN or RETURN (in the highest structure block).
(4) There should be only one INPUT or OUTPUT statement for each data file and processing mode.
(5) Subprograms should be called only using PERFORM and left only by EXIT.
(6) All subprograms have one entry and one exit (dynamic and static in the same place).
(7) Subprograms may only be called at directly subordinate level (exception: utility subprogram).
(8) GO TO instructions may not lead out of the subprogram.
(9) The names of the subprograms should indicate the position within the hierarchy (cf. Figure 11.1).
(10) The sequence of subprograms should be listed in the order corresponding to the program organisation.
(11) Loops may not overlap.
(12) When represented graphically, a structure block should not exceed one A4 sheet.

11.3 Formation of Structure Blocks (Figure 11.2)

Structure blocks are built up *exclusively* by adding one to another (sequence) or by inserting one basic element into another. When adding one structure block to another, the lower edge of the preceding structure block must meet the upper

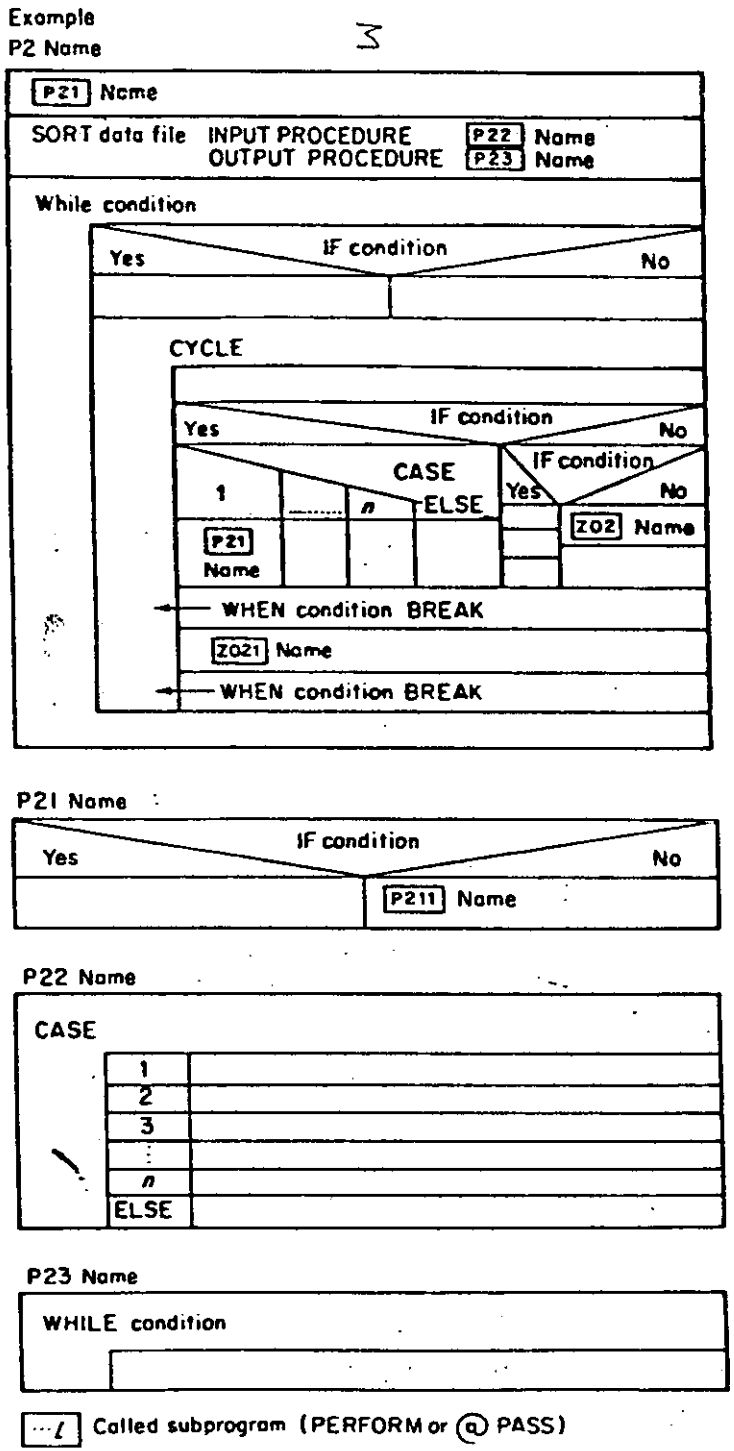


Figure 11.2 Formation of structure blocks

...e of the following structure block *exactly*. When inserting elemer... the inner element (or the sequence of the inner elements) must fill the outer rectangle *completely*.

11.4 COBOL Conventions

Formal conventions

Level numbers should be allocated in groups of five so that it is easier to make later additions. Indentations should follow the guidelines in the COBOL form. All PICTURE entries start in the same column (preferably 41).

Example

```
01  Z1-EXPENSE-RECORD.
    05  Z1-ACCOUNT.
        01.  Z1-ACCOUNT-1-6      PIC X(6).
        01   Z1-ACCOUNT-7-9      PIC X(3).
```

Paragraph names should be written on a separate line. There should be only one statement per line. In general, statements should start in column 12 (except IF commands). For IF commands, the commands that are executed for the YES and NO cases should be indented (using the guidelines in the COBOL form): 'ELSE' should be written on a separate line and start in the same column as the preceding 'IF'.

Example

```
    MOVE E1-CUSTGP TO Z1-CUSTGP.
U230-05.
    IF E1-TYPE = "320" OR "330"
        ADD E1-VALUE TO T1-VALUE
    ELSE
        ADD E1-PRICE TO E1-VALUE.
```

The headings for data definitions and sub-programs should be clearly indicated using '*' cards.

Security

This should record all modifications to the program.

From:	Date of the application for amendment (clear cross reference)
On:	Date of the program modification
Vers:	Number of the version or modification
Name:	Initials of the programmer
Modification:	A brief summary of the modification or the reference to a document must be given.

Example

```
SECURITY   MODIFICATIONS CARRIED OUT
FROM    ON      VERS. NAME MODIFICATION
09.01.74 23.01.74   002    B1   NAME OF COUNTRY. 037 BELGIUM
                                PERMITTED
18.06.74 10.08.74   003    KE1  ADDITIONAL FILE ISSUED FOR
                                CONFIRMING DEADLINES
```

Remarks

This gives a brief description of the components. Abbreviations that are used later should also be included here.

Example
The example is that given on page 121.

Remarks.

This program lists all order transaction data according to the criteria given in the parameter cards. Within these criteria, the summaries and listings are in the sequence specified by the user on the line parameter cards. The listing of order transactions comprises the data for orders and turnover and the sales statistics listing prints the data for turnover and expenditure.

Environment division

Select

The name of the data file is allocated in the specification phase.

Example

```
CONDITION   FILE
FD  TF75201 RECORDING MODE IS F
            BLOCK CONTAINS 1 RECORDS
            RECORD CONTAINS 513 CHARACTERS
            LABEL RECORD IS STANDARD
            DATA RECORD IS E3-CONDITION-RECORD
01  E3-CONDITION-RECORD
```

In BS 1000, columns 73 to 80 can be used for:

(a) date of the modification;
(b) number of the application for modification;

(c) program number;
(d) other identification.

The use of these columns must be consistent within a system.

Note

In BS 2000 this will result in 30 to 40% more space used in the source program library.

11.5 Structure of a COBOL Program

Identification division

Author

This should include the name, office, location, and telephone number of the member of staff.

Example

```
AUTHOR.    MEIER, N DD VE 15, MCH-H/BR, TEL. 42888
```

11.6 Recommendations

(1) Use condition names (88-level):
  (a) to interrogate coded field (IF MALE instead of IF E1-SEX = 1);
  (b) to interrogate status variables (switches with more than two states).
(2) Switch names should describe the processing state:

  (H10-DATA-FILE-OPENED, H11-FIRST-PASS instead of SWITCH-1, SWITCH-2)

(3) Switch values should describe symbolically the state of the switch:

  (YES, NO instead of 1, 0)

(4) Clarify the COBOL code adequately using comments (* cards).
(5) Subprograms should be a reasonable size (up to 200 instructions).
(6) Program listings clarified by separating divisions, data areas, and sections.
(7) Data fields numbered in ascending order and written in the correct sequence (auxiliary fields, constants, etc.).
(8) Definitive data names as defined in the specifications.
(9) Constants in the WORKING STORAGE SECTION defined by value clauses.
(10) Identifying constant with program names, version number, and associated data at the start of the WORKING STORAGE SECTION; this constant should appear in the program log.
(11) Definitive section headings.
(12) Paragraph names on a separate line and numbered in ascending order within a section.

Table 11.1 Use of COBOL instructions

May not be used	Should be used
ENVIRONMENT DIVISION	
APPLY ... TO FORM-OVERFLOW ON (Use line count) SAME AREA FOR	
DATA DIVISION	
	BLOCK CONTAINS ... RECORDS FILLER always if the field is not addressed (except when using with COPY) RECORD CONTAINS ... CHARACTERS
RENAMES use	
REDEFINES	REDEFINES Redefining data names must contain the redefined data names VALUE if a fixed initial value is required at the outset in the WORKING STORAGE SECTION
PROCEDURE DIVISION	
ADD CORRESPONDING	COMPUTE should be used for the sake of clarity for comprehensive arithmetic operations, where a large number of ADD, SUBTRACT, MULTIPLY and DIVIDE operations would be unclear
MOVE CORRESPONDING not possible when adhering to the data name conventions	
SUBTRACT CORRESPONDING	
ALTER	
GO TO (also GO TO DEPENDING ON)	Exception: only within a SECTION and in a forward direction and for forming loops within a SECTION
PERFORM THRU	
IF double negative tests	

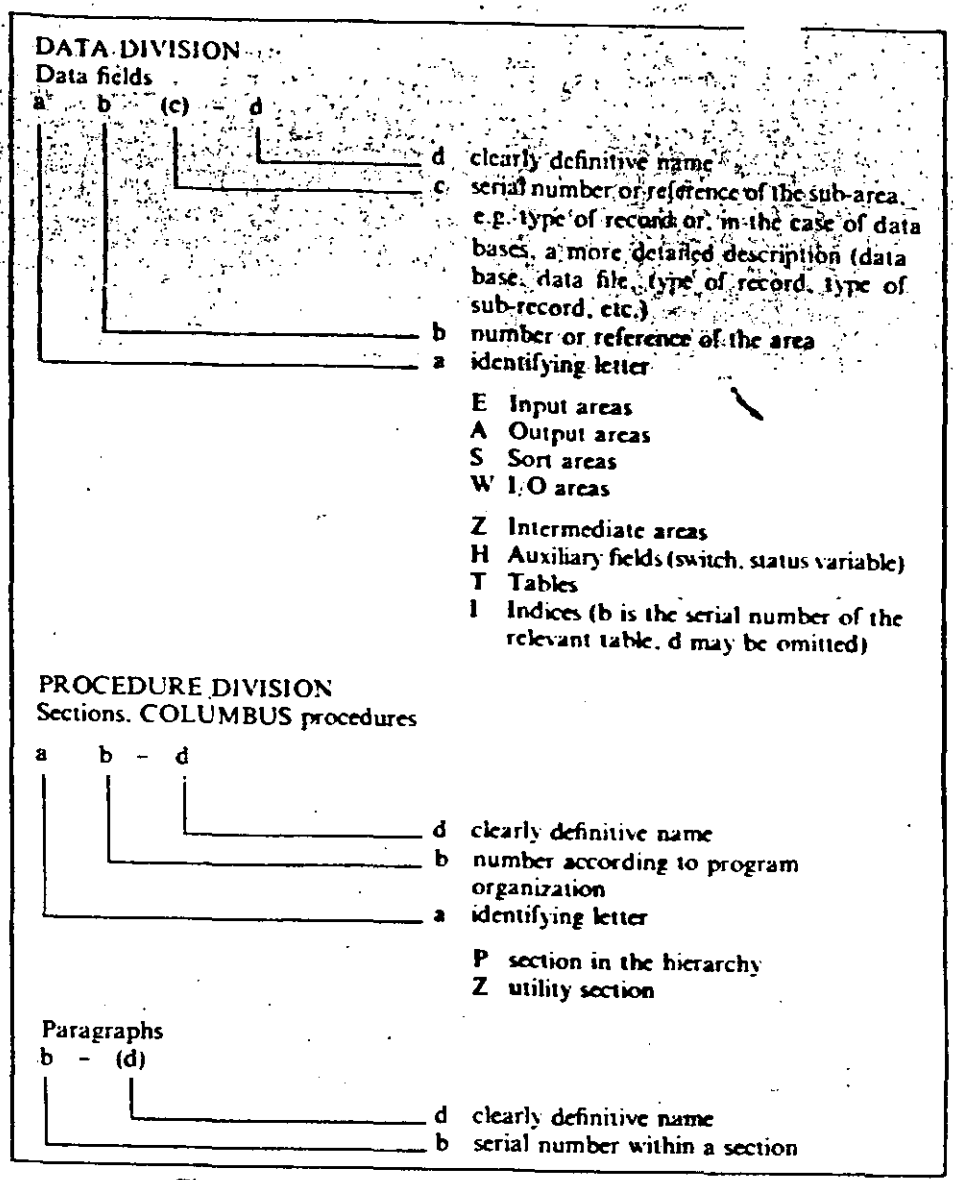


Figure 11.3 Naming conventions in COBOL programs

(13) Actions for special exits (AT END, INVALID KEY, ...) on a separate line and indented.

11.7 Data Dictionary

As already explained in Section 3.4, it is necessary to set up a data dictionary in order to have a clear document of the data required for processing. This becomes more important with increasing use of data-processing and increasing integration of systems within the company.

The purpose of the data dictionary is:

(1) to record available data, describe data (e.g. names, brief descriptions, etc.);
(2) to distinguish data (content of data and description of data for the purpose of clarity);
(3) to provide information about names and brief descriptions;
(4) to provide information about the range of data;
(5) to provide information about dependencies;
(6) to provide information about authorized access;
(7) to provide information about security measures;
(8) to formalize descriptions of data;
(9) to standardize descriptions of data;
(10) to act as a tool for designing or reorganizing data structures and data storage;
(11) to be the basis for implementing systems.

Descriptions of the (data field, data record, data file, data permanence) should include the individual points suggested in the following section.

Content of a data dictionary

Identification

Brief designation	(observe the conventions of the programming language)
Name	(in full—normal business terminology)
Meaning	(including differentiation from similar data)
Synonyms	(brief designation of similar information)
Valid area	(where the description is valid)
Revision number	
Validity	(from to)

Form

Format; structure	(e.g. length of fields, sequence, record format)
Range of values; length of records	
Data files used	
Compression	(details of the method of compression)
Form of storage	
Method of access	(e.g. HASH, ISAM, VSAM)
Reorganization	(periodic or specific conditions)

Relationships

Formal relationships within the task to upper and lower levels	(system, program, elementary process)
Logical relationships	

Authorized Access

To read	
To read/modify	
To read/modify/delete	(protection and security of data, cf. Part 1, page 80)
Authority to release	

Security Measures

Risk classification	
Archiving	

(4) Other requirements:

(a) usage;
(b) emulation;
(c) examination of costs and schedule;
(d) probable run times;
(e) documentation.

Function of the components

The function of the components should be described without giving details of specific applications. This serves as an introduction to the components.

Example

Remarks.

This program lists all order transaction data according to the criteria given in the parameter cards. Within these criteria, the summaries and listings are in the sequence specified by the user on the line parameter cards. The listing of order transactions comprises the data for orders and turnover and the sales statistics listing prints the data for turnover and expenditure.

Definition of the data-processing design for each component

Structure

The structure of the components is determined by system requirements and is formed of structure blocks. It must be very detailed so that other programming departments can execute the programming without significant queries. The following points should be considered when determining the structure:

(1) The structure of the components should be refined in stages from the top downwards by subdividing and expanding. This produces the structure blocks.
(2) The structuring should be done in 'levels'. A level is formed by collecting structure blocks which contain a similar amount of detail.
(3) The determination of a level may only begin when the preceding levels have been fully completed and defined. A level of structure blocks is completely defined when:

(a) the number of structure blocks which it contains is determined;
(b) the interfaces of each block to the next level have been described in detail;
(c) the content of the majority of the structure blocks has been completely expanded (i.e. programming can be commenced without further comments being necessary). Structure blocks which contain functional or system material that has not yet been defined should be sketched in outline and a note made that they should be completed later.

(4) The following requirements should also be considered when forming the structure blocks. They should be of a reasonable size, i.e.

(a) they should not contain more than one decision table;
(b) they should not exceed approximately 200 instructions or 20 independent conditions.

(5) Input and output should be executed in separate structure blocks.

Program organization

The structure of a component is set out in the program organization (Figure 4.3). This is the structure of the program represented as a structure diagram. 'Hierarchical structure blocks', which are firmly established in the hierarchy, should be distinguished in the diagram from 'utility structure blocks' which may be referenced from several structure blocks.

When defining the program organization:

(1) The connecting lines must indicate the paths between the structure blocks.
(2) A path between the structure blocks may only go vertically without bypassing, i.e. blocks on the same level may not be connected to each other.
(3) Utility structure blocks should be shown separately from the hierarchical blocks and all entry points should be indicated.
(4) The name used should be the same as those in the subsequent source program.
(5) The numbering of the structure blocks should reflect their position in the hierarchy.

External data interfaces

The inputs and outputs of the components should be described in detail or be taken from the functional specification. The following single tasks must be done:

(1) The record format and structure should be adapted to and supplemented by the external data format and description in accordance with the functional specification (Form 4.2). They should be defined for the transfer data and data files.
(2) The data dictionary should be checked to ensure that it is complete and that there are no overlaps or conflicts and, if necessary, amended.
(3) 'Ready-made' routines should be checked to see whether they can be used for the data definition and input and output modules which are required in several components. This is particularly appropriate for the systems which are to be implemented in one program (cf. page 93).

Considerable advantages are gained if the required amendments and extensions can be done centrally and be incorporated in all components during program linking.

Name of data file: Order data file – header record

Record format

Length: 512 –

Format of field: a alphabetic; an alphanumeric; b binary; n numeric unpacked; p numeric packed; f1 simple floating point; f2 double floating point

Content for processing: C condition; T decision table; R record address; En error no.

Definition is valid for several components

Field no.	Position from	Position to	Length (bytes)	Format	Note for processing	Name of field	Comments
1	1	10	10	an	R	Order number	
2	11	11	1	an	R	Type of record	Classification
3	12	18	7	n	R	Item	
4	19	20	2	an	R	Link	
5	21	23	3	n	R	Customer number	
6	24	25	2	n		Month	Supply deadlines
7	26	27	2	n		Year	

Name of data file: Goods ordered

Record format: Variable

Format of field: a alphabetic; an alphanumeric; b binary; n numeric unpacked; p numeric packed; f1 simple floating point; f2 double floating point

Content for processing: C condition; T decision table; R record address; En error no.

Definition is valid for several components

Field no.	Position from	Position to	Length (bytes)	Format	Note for processing	Name of field	Comments
1	1	10	10		R x(10)	WA-ORD-NO	
2	11	11	1		R x	WA-RECTYPE	
3	12	13	2		R 99	WA-ITEM-B	Corresponds to Field Number 3 in functional specification
4	14	16	3		R 999	WA-ITEM-C	
5	17	18	2		R 99	WA-ITEM-D	
6	19	20	2		R xx	WA-LINK	
7	21	23	3		999	WA-CUST-NO	
8	24	25	2		99	WA-DEL-MON	
9	26	27	2		99	WA-DEL-YR	

Form 4.2 Description of the data file

Description of the structure blocks

The input, operations, and output of each structure block should be described.

For input, the data records and input parameters used by each structure block should be described (a cross reference to the 'external data interfaces' may be sufficient in certain circumstances).

For the operations, the methods used in each structure block should be described. The operations should be represented using a flow chart, a

VENTAJAS DE LOS ARBOLES DE DECISION

1.- ESCLARECEN LA LOGICA DE COMBINACIONES DE CONDICIONALES

2.- SON FACILES DE LEER

3.- SON BUENOS VERIFICADORES DE ESPECIFICACIONES

4.- AYUDAN A SIMPLIFICAR LA CODIFICACION DE PROGRAMAS

5.- SON UTILES COMO UNA FORMA DE DOCUMENTACION

DESVENTAJAS DE LOS ARBOLES DE DECISION

1.- NO ES FACIL VERIFICAR DE QUE TODAS LAS COMBINACIONES POSIBLES DE CONDICION PERMITIDAS ESTEN PRESENTES

2.- NO ES UNA HERRAMIENTA FORMAL

3.- CUANDO OCURREN MUCHAS COMBINACIONES DE CONDICIONES Y EL NUMERO DE ACCIONES INDEPENDIENTES ES GRANDE EL ARBOL DE DECISION SE EXPANDE GRANDEMENTE

PASOS EN LA DERIVACION DE UN ARBOL DE DECISION

1.- IDENTIFIQUENSE TODAS LAS CONDICIONES POSIBLES DEL PROBLEMA

2.- IDENTIFIQUE TODAS LAS COMBINACIONES DE CONDICIONES EXISTENTES

3.- IDENTIFIQUE TODAS LAS ACCIONES INDEPENDIENTES DEL PROBLEMA

4.- TRACE EL FORMATO DE UN ARBOL DE DECISION AL PROBAR CADA CONDICION EN SECUENCIA

5.- EVALUE LAS ACCIONES A LO LARGO DE CADA RAMA

6.- IDENTIFIQUE CUALQUIER OMISION, AMBIGUEDAD O CONTRADICCION EN EL PROBLEMA

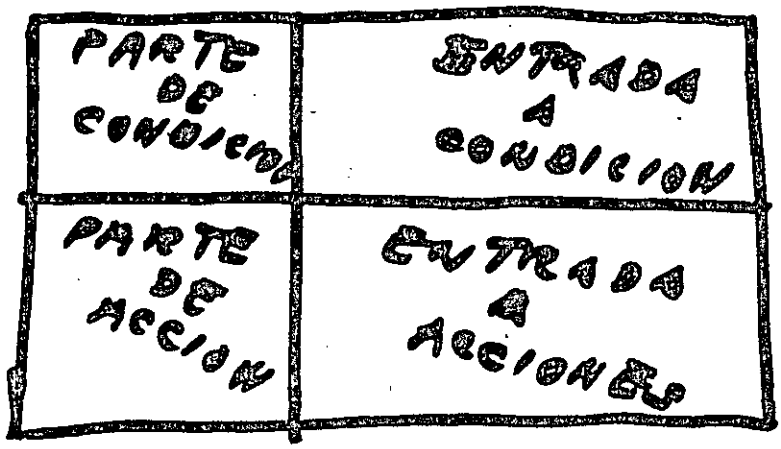
TABLAS DE DECISION

UNA TABLA DE DECISION, AL IGUAL QUE UN ARBOL, ES UNA HERRAMIENTA QUE PERMITE ESCLARECER LA LOGICA DE CONDICIONALES COMPLEJOS.

A DIFERENCIA DE UN ARBOL, UNA TABLA SE DERIVA DE UNA MANERA MUCHO MAS SISTEMATICA ADEMAS DE TENER UN FORMATO MUCHO MAS COMPACTO.

EXISTEN 3 TIPOS DE TABLAS DE DECISION

A. ENTRADA LIMITADA
B. ENTRADA EXTENDIDA
C. ENTRADA MIXTA



PARTE DE CONDICION (1)	ENTRADA DE CONDICION (3)
PARTE DE ACCION (2)	ENTRADA DE ACCION (4)

1.- EN EL CUADRANTE SUPERIOR IZQUIERDO ESTA LA PARTE DE CONDICION. ESTA AREA DEBE CONTENER (EN FORMA DE PREGUNTA) TODAS AQUELLAS CONDICIONES EXAMINADAS PARA UN PROBLEMA DADO.

2.- EN EL CUADRANTE INFERIOR IZQUIERDO ESTA LA PARTE DE ACCION. ESTA AREA DEBE CONTENER EN FORMA DE NARRATIVA SIMPLE TODAS LAS ACCIONES POSIBLES RESULTANTES DE LAS CONDICIONES LISTADAS ARRIBA

3.- EN EL CUADRANTE SUPERIOR DERECHO ESTA LA ENTRADA DE CONDICION. EN ESTA AREA TODAS LAS PREGONTAS HECHAS EN LA PARTE DE CONDICION DEBEN RESPONDERSE Y TODAS LAS COMBINACIONES POSIBLES DE ESTAS RESPUESTAS DEBEN DESARROLLARSE. SI UNA RESPUESTA NO SE INDICA PUEDE ASUMIRSE QUE LA CONDICION NO FUE SOMETIDA A PRUEBA EN ESA COMBINACION PARTICULAR.

4.- EN EL CUADRANTE INFERIOR DERECHO ESTA LA ENTRADA DE ACCION. LAS ACCIONES APROPIADAS RESULTANTES DE LAS DIVERSAS CONDICIONES DE LAS RESPUESTAS A LAS CONDICIONES DE ARRIBA SE INDICAN AQUI. UNA O MAS ACCIONES PUEDEN INDICARSE PARA CADA COMBINACION DE RESPUESTAS DE CONDICION.

LAS DIVERSAS COMBINACIONES DE RESPUESTAS A CONDICIONES MOSTRADAS EN LA ENTRADA DE CONDICION DE LA TABLA Y SUS ACCIONES RESULTANTES SE LLAMAN REGLAS. A CADA UNA SE LE DA UN NUMERO PARA PROPOSITOS DE IDENTIFICACION.

OTRO ELEMENTO QUE REQUIRE UNA TABLA COMO UN MEDIO DE DISTINGUIRLA DE OTRA TABLA ES UN NOMBRE.

TABLAS DE ENTRADA LIMITADA

ESTE TIPO DE TABLAS SON LAS MAS AMPLIAMENTE USADAS

1.- LA PARTE DE CONDICION DEBE MOSTRAR LA CONDICION Y SU ESTADO O VALOR

2.- LA ENTRADA DE CONDICION DEBE MOSTRAR SOLAMENTE SI ó NO O UN BLANCO

3.- LA PARTE DE ACCION DEBE MOSTRAR LAS ACCIONES A TOMAR

4.- LA ENTRADA DE ACCION DEBE MOSTRAR UNA X INDICANDO QUE ESA ACCION SE TOMA O UN BLANCO INDICANDO QUE ESA ACCION NO SE TOMA.

UN EJEMPLO SIMPLE

SI	¿LLUEVE?	S	N	N	S
	¿HACE FRIO?	N	S	N	S
	LLEVAR PARAGUAS	X			X
LUEGO	LLEVAR SWEATER		X		X
	NORMAL			X	

considerese el anunciado con arboles de decisión

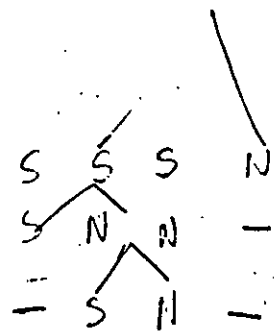
(1)

SI	¿LLUEVE?,	S	S	N	N
	¿HACE FRIO?	S	N	S	N
	LLEVAR PARAGUAS	X	X		
LUEGO/OBIEN	LLEVAR SWEATER	X		X	
	NORMAL			X	X

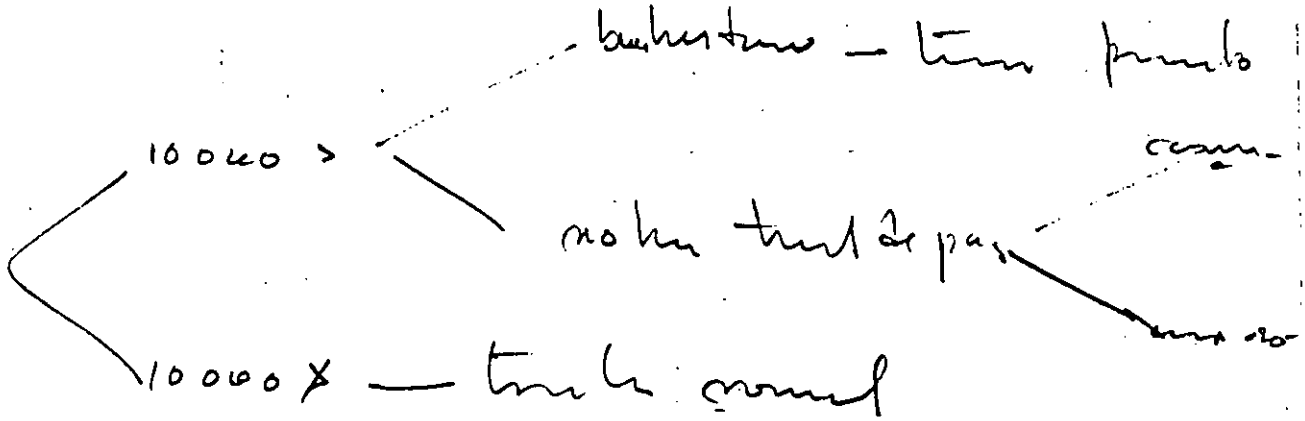
Poner papel en blanco aquí

CLIENTES QUE NOS COMPRAN MAS DE 10 000 PESOS AL AÑO Y ADEMAS, O TIENEN UN BUEN HISTORIAL DE PAGO O HAN ESTADO CON NOSOTROS POR MAS DE 20 AÑOS RECIBEN TRATAMIENTO PRIORITARIO

MAS DE 10 000 AL AÑO	S	S	S	S	N	N	N	N
BUEN HISTORIAL DE PAGO	S	S	N	N	$\bar{S}$	$\bar{S}$	$\bar{N}$	$\bar{N}$
CON NOSOTROS MAS DE 20 AÑOS	$\bar{S}$	$\bar{N}$	S	N	$\bar{S}$	$\bar{N}$	$\bar{S}$	$\bar{N}$
TRATAMIENTO PRIORITARIO	X	X	X					
TRATAMIENTO NORMAL				X	X	X	X	X



1 - 2 4 - 5
2 - 3 4 - 6
3 - 4 4 - 7



PROBLEMA

UN EDITOR DE REVISTAS HA PROMOVIDO UNA CAMPAÑA DE SUSCRIPCIONES EN LA QUE LOS LECTORES SON MOTIVADOS A SUSCRIBIRSE DE UNO A DOS AÑOS. CUANDO SE RECIBE UNA ORDEN DE SUSCRIPCION, SI ES LA FORMA DE ORDEN ESPECIAL INDICANDO CON ESTO QUE FUE RESULTADO DE LA CAMPAÑA, SE CLASIFICA COMO PROMOCION, SI NO ES LA FORMA ESPECIAL SE CLASIFICA 'REGULAR'. SI LA SUSCRIPCION ES POR UN AÑO SE CLASIFICA 'UN AÑO, SI POR DOS AÑOS SE CLASIFICA 'DOS AÑOS'. LAS ORDENES ACOMPAÑADAS POR EL PAGO SE CLASIFICAN 'PAGADA', Y LAS QUE NO SE CLASIFICAN COMO 'FACTURADOS'. AQUELLAS SUSCRIPCIONES PARA LA PROPIA CIUDAD SE CLASIFICAN COMO 'ENVIO LOCAL' Y LOS ENVIOS FUERA DE LA CIUDAD COMO 'ENVIO FORANEO'.

TABLAS DE ENTRADA EXTENDIDA

EN UNA TABLA DE ENTRADA EXTENDIDA LA PARTE DE CONDICION SOLO SIRVE PARA IDENTIFICAR A LAS VARIABLES QUE SE SOMETEN A PRUEBA, ENTANTO QUE LA ENTRADA DE CONDICION DEFINE EL VALOR O ESTADO DE LA VARIABLE.

UNA TABLA DE ENTRADA EXTENDIDA TIENDE A SER MAS CORTA VERTICALMENTE Y OFRECE LA POSIBILIDAD DE CONSIDERAR MAS DE 2 VALOREE A UNA CONDICION DADA.

VENTAJAS DE LAS TABLAS DE DECISION

1.- ES UNA HERRAMIENTA SISTEMATICA EN EL ESCLARECIMIENTO DE LOGICA DIFICIL (COMBINACION DE CONDICIONES)

2.- TIENEN UN FORMATO COMPACTO

3.- ES UNA HERRAMIENTA UTIL PARA:

ENCONTRAR INCONSISTENCIAS
ENCONTRAR FALLAS EN LA LOGICA
VERIFICAR ESPECIFICACIONES

4.- SON ONA EXCELENTE HERRAMIENTA PARA DOCUMENTACION

5.- PUEDEN USARSE PARA AYUDAR A ESCRIBIR DECISIONES COMPLEJAS DE UNA MANERA DIRECTA Y RAZONABLE

CUANDO USAR TABLAS DE DECISION O ARBOLES DE DECISION

1.- USE UN ARBOL DE DECISIONES CUANDO EL NUMERO DE ACCIONES SEA PEQUEÑO Y NO OCURRAN MUCHAS COMBINACIONES DE CONDICIONES, USE UNA TABLA DE DECISION CUANDO EL NUMERO DE ACCIONES ES GRANDE Y OCURREN MUCHAS COMBINACIONES DE CONDICIONES

2.- USE UNA TABLA DE DECISION SI DUDA QUE EL ARBOL DE DECISION NO MUESTRA TODA LA COMPLEJIDAD DEL PROBLEMA

3.- AUN SI USTED NECESITA UNA TABLA DE DECISION LA LOGICA DEL PROBLEMA TERMINE PRESENTANDOLO COMO UN ARBOL DE DECISION SIN VIOLAR 1.

PASOS EN LA DERIVACION DE UNA TABLA DE DECISION

PASO 1

IDENTIFIQUE TODAS LAS CONDICIONES Y TODOS LOS VALORES POSIBLES DE ESAS CONDICIONES EN EL PROBLEMA

CONDICIONES	VALORES
ORDEN > LIMITE DE CREDITO	S O N
APROBACION ESPECIAL DE CREDITO	S O N
ORDEN < CANTIDAD DE ENVIO	S O N
APROBACION ESPECIAL DE ENVIO	S O N

PASO 2

CALCULE CUANTAS COMBINACIONES DE CONDICIONES SON POSIBLES AL MULTIPLICAR EL NUMERO DE VALORES ENTRE SI DE LAS CONDICIONES

NRO DE COMBINACIONES =

NRO DE VALORES CONDI 1 X NRO DE VALORES CONDI X NRO DE VALORES CONDI J X ...

PASO 3

IDENTIFIQUE TODAS LAS ACCIONES INDEPENDIENTES DEL PROBLEMA

ACCIONES

PROCESAR ORDEN
RECHAZAR CREDITO
RECHAZAR ENVIO

PASO 4

TRACE EL FORMATO DE UNA TABLA DE DECISION AL NUMERAR CADA UNA DE LAS COMBINACIONES DE LAS CONDICIONES EN LA PARTE SUPERIOR DE LA TABLA Y LISTE LAS CONDICIONES Y CADA UNA DE LAS ACCIONES EN EL EXTREMO IZQUIERDO DEL DIAGRAMA

PASO 5

APLIQUE LA SIGUIENTE FORMULA PARA COLOCAR LOS VALORES DE LA TABLA PARA LA CONDICION DE MAS ABAJO

$$\frac{\text{NRO TOTAL DE COMBINACIONES DEL PROBLEMA}}{\text{NRO DE VALORES DENTRO DE LA CONDICION}} = \text{FACTOR DE REPETICION PARA CADA VALOR DENTRO DE LA CONDICION}$$

PASO 6

PARA CADA CONDICION SIGUIENTE LISTADA EN LA TABLA DE DECISION, APLIQUE LA SIGUIENTE FORMULA

$$\frac{\text{FACTOR DE REPETICION DE LA CONDICION LISTADA PREVIAMENTE}}{\text{NUMERO DE VALORES DENTRO DE LA CONDICION}} = \text{FACTOR DE REPETICION PARA CADA VALOR DENTRO DE LA CONDICION ACTUAL}$$

PASO 7

ENCUENTRE LAS ACCIONES APROPIADAS PARA CADA CONJUNTO DE CONDICIONES APROPIADAS

PASO 8

IDENTIFIQUE CUALQUIER OMISION, AMBIGUEDAD O CONTRADICCION EN EL PROBLEMA

PASO 9

ALINIE CONDICIONES VERTICALES DE ACUERDO A LA ACCION RESULTANTE

PASO 10

TRATE DE CONDENSAR LA TABLA AL BUSCAR INDIFERENCIAS ENTRE 2 REGLAS RESULTANTE EN LAS MISMAS ACCIONES

1.- OBJETIVO

2.- DESCRIPCION DETALLADA DE LA SOLUCION

2.1 DIAGRAMAS DE LA ORGANIZACION DEL PROGRAMA

2.2 INTERFACES DE DATOS EXTERNOS
- CONTENIDO DE LOS REGISTROS
- ESTRUCTURA DEL REGISTRO.

2.3 DESCRIPCION DE LOS MODULOS

DESCRIPCION DEL MODULO 1
ENTRADA
MECANICA DEL MODULO
SALIDA
DESCRIPCION DEL MODULO 2
ENTRADA
MECANICA DEL MODULO
SALIDA
:
:
DESCRIPCION DEL MODULO N
ENTRADA
MECANICA DEL MODULO
SALIDA

3.- OTROS DETALLES

CONFIGURACION DEL HARDWARE
TECNICAS DE SOFTWARE
USO DE SOFTWARE EXISTENTE
NORMAS Y GUIAS
SEGURIDAD DE DATOS

PASO 11 29

ESCRIBA EL CODIGO CORRESPONDIENTE

LISTA DE VERIFICACION 30

1.- ¿LOS OBJETIVOS DE CADA COMPONENTE HAN SIDO DESCRITOS?

2.- ¿EXISTE UN DIAGRAMA DE LA ORGANIZACION DEL PROGRAMA?

3.- ¿ESTAN LOS MODULOS DEL PROGRAMA SEPARADOS DE LOS MODULOS DE UTILERIA?

4.- ¿ESTAN LOS MODULOS DE UTILERIA MARCADOS PARA MOSTRAR QUE MODULOS DEL PROGRAMA HACEN USO DE ELLOS?

5.- ¿LA NUMERACION DE LOS MODULOS CORRESPONDE A SU POSICION EN LA JERARQUIA?

6.- ¿ESTAN LAS ENTRADAS, PROCESOS Y SALIDAS DE CADA MODULO DESCRITAS?

7.- ¿EXISTE UNA DESCRIPCION DE CADA ARCHIVO DE DATOS EN UNA FORMA TABULAR O GRAFICA?

8.- ¿EXISTE UNA LISTA DE POSIBLES ERRORES Y LOS MENSAJES Y ACCIONES REQUERIDAS?

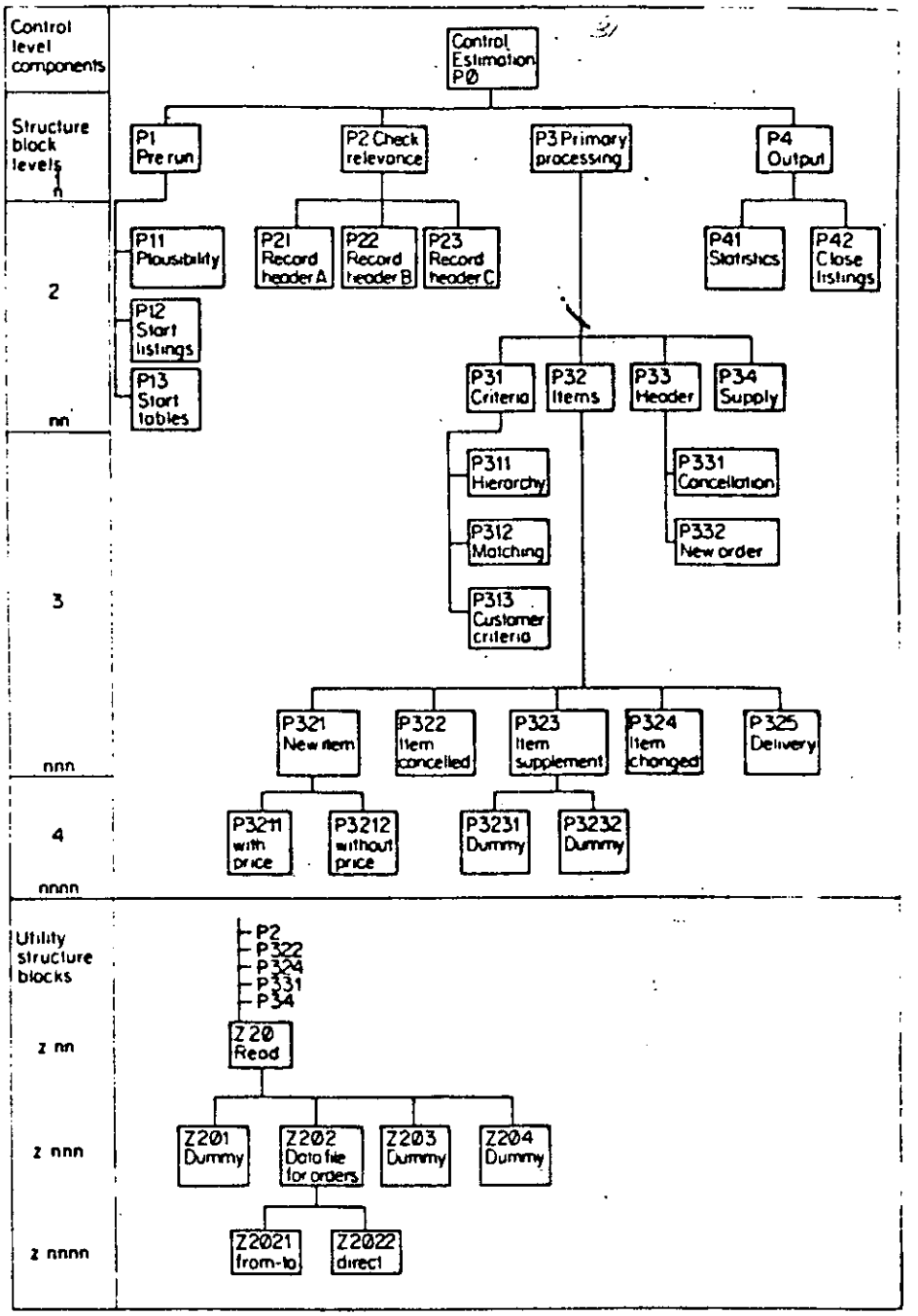


Figure 4.3 Example of a program organization

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

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

CALIDAD DEL SOFTWARE

ING. DANIEL RIOS ZERTUCHE

MAYO, 1985

(1)

Calidad del Software

Cuando iniciamos nuestra preocupación acerca de la calidad del Software, la primera intención es voltear la vista hacia el --- Hardware, en donde ya hay una gran experiencia acumulada, sin embargo en muchos casos se olvida que no es lo mismo Hardware que Software; la primera diferencia que es de llamar la - atención es que el Software no se degrada con el uso, nadie ha visto un if-them-else en el que la condición se debilite con el uso, ni una operación lógica en la que un falso contacto introduzca ruido.

En el caso del Hardware uno de los papeles predominantes del Control de Calidad ha sido la inspección de los productos, básicamente para asegurar que el diseño original es copiado fielmente en las unidades de producción. En el Software esto carece - de importancia ya que el duplicar el Software no tiene problemas. Por otro lado tenemos que generalmente la falla de Hardware avisa ya sea por problemas de calentamiento, ruido etc.- que nos lleva a preveer una falla próxima; en el caso del Software esto no sucede. Estos puntos nos indican que la calidad y confiabilidad tanto en el Hardware como el Software son muy distintos.

Por calidad del Software entenderemos sus propiedades de confiabilidad, engrandecimiento, mantenibilidad, operacionalidad y economía.

O bien podemos decir que el Software de Calidad debe ser confiable, conciso, consistente, eficiente, mantenible, portable y entendible. A lo que se le puede agregar que sea producido dentro del itinerario y presupuesto presentado.

Las anteriores definiciones nos aclaran un tanto que características debe cubrir el Software para considerarlo de calidad, pero se puede notar también que todas estas características son abstractas por lo que la apreciación de la calidad a la fecha es un tanto subjetiva.

Concretando lo anterior podemos decir que el Software de Calidad debe ser mantenible, efectivo y eficiente.

Eficiente en cuanto a que aproveche los recursos de la computadora adecuadamente.

Efectivo en cuanto a que satisfaga las especificaciones, mucho se ha hablado de un programa correcto para el problema incorrecto.

Mantenible en cuanto a la facilidad con que el Software pueda ser expandido o contraido para satisfacer nuevos requerimientos o pueda ser corregido cuando se detecten errores o deficiencias.

Del punto anterior acerca del mantenimiento surge inmediatamente una duda en relación a lo que significa en el lenguaje coloquial en término mantenimiento, básicamente por mantenimiento en términos de Hardware entendemos el restaurar su condición inicial. Y vemos que de dar mantenimiento al Software resulta una nueva definición del producto, con la consecuente actualización de la documentación de los programas para no afectar la posibilidad de éxito de las subsecuentes modificaciones.

El Mantenimiento de Software tiene básicamente los siguientes propósitos:

Corrección

Adaptación

Engrandecimiento

Reestructuración

Corrección. Es la modificación para corregir. Esto es lograr que el Software realice la función que se pretende, sin tener que ver con que sí la función satisface

o no la necesidad real de los clientes.

Adaptación. Es la modificación debida a un cambio en el Hardware o Software en el que reside el Sistema.

Engrandecimiento. Las modificaciones que permitirán al sistema realizar nuevas funciones en función a los requerimientos del usuario, en la que comunmente incurre el mantenimiento en la mayoría de las instalaciones.

Reestructuración. Generalmente se efectua con el objeto de mejorar la estructura interna conservando su comportamiento externo.

Esta amplitud de funciones bajo el paraguas de mantenimiento ha creado controversias respecto a sí el término mantenimiento es el correcto o sí deberá cambiarse por otro más adecuado; a la fecha estas discusiones no han llegado a nada claro pero si han sacado a la luz situaciones como que si a una corrección se le llama mantenimiento y el proveedor la puede cobrar y este tipo de situación es muy común, por lo que creo lo más conveniente continuar llamando mantenimiento a el conjunto de actividades antes descritas.

(5)

Para los Ingenieros de calidad la suma de diferencias entre -- Hardware y Software, implican el mensaje que mientras la disciplina del aseguramiento de calidad tradicional se aplique, las practicas deben diferir entre Hardware y Software y más especificamente para el Software deben enfatizar el concepto de calidad interconstruida, basado en la idea de "hágalo bien la primera-vez".

La tarea esencial de la calidad es enfocar su atención en el establecimiento de estandares que conduzcan a producir Software de calidad, y auditar la fidelidad con la que se respeten estos estandares.

Calidad es la conformidad con los requerimientos y prevención de efectos. Es la responsabilidad de la calidad actuar como - instrumento independiente en la auditoria de todos los aspectos del desarrollo del Software y su mantenimiento, a través de la revisión de los planes, especificaciones, diseño de pruebas, documentación, control de la configuración y estandares de programación.

Gran parte de los problemas a los que se enfrenta el encargado de dar mantenimiento a un sistema se deben a que el grupo de desarrollo no se preocupo por preveer la fase del mantenimiento, raro es el que desarrolla un sistema y lo diseñe para poder ser modificado fácilmente.

Con el objeto de lograr un producto mantenible se deben cumplir los siguientes estandares al menos.

1. Requerimientos

   Los requerimientos deben ser escritos, priorizados y definidos en términos que se puedan probar fácilmente.

   Requerimientos opcionales y los requerimientos futuros deben ser diferenciados.

   Los requerimientos deben incluir los recursos de cómputo necesarios para operación y para pruebas de mantenimiento.

2. Especificaciones

   Las especificaciones deben estar escritas en términos que se puedan probar.

   Las funciones deben estar diferenciadas en requeridas, opcio

nales y futuras.

3. Diseño

Las facilidades del diseño para extenderlo, contraerlo y - adaptarlo deben explicarse con ejemplos y ejercicios de los cambios esperados.

4. Código Fuente

Se deben emplear lenguajes de alto nivel siempre que sea posible.

Unicamente versiones estandar y características estandar del lenguaje de programación deberán permitirse.

Todo el código deberá estar bien estructurado.

Todo el código deberá documentarse para explicar el propósito de cada módulo, sus entradas y salidas y las variables para facilitar la prueba del módulo.

5. Información del Sistema.

Todos los documentos del sistema se deberán entregar al encargado del mantenimiento del sistema inmediatamentes después de la definición de estas incluyendo el UDF.

Proceso de Mantenimiento

Definiremos el proceso de mantenimiento incluyendo los siguientes pasos:

1. Entendimiento del Software
2. Identificación del objetivo de la modificación y el enfoque de la modificación.
3. Implantación de la modificación
4. Revalidación del Software.

Las áreas de problemas frecuentemente citados al efectuar el proceso de mantenimiento incluyen:

a) La Calidad del Software Original
b) La Calidad de la Documentación
c) Las Limitaciones en recursos para pruebas
d) La dificultad para entender el Software debido a su complejidad.

Como es inutil agregar todas las áreas de problemas anteriores quedan resueltas si se satisfacen los estandares antes descritos.

Ejecución del Mantenimiento

De igual forma que para el desarrollo de un sistema, se requie

ren los tres ingredientes básicos para el mantenimiento de programas.

1) Herramientas Técnicas
2) Experiencia Técnica
3) Técnicas de Administración

1) Herramientas Técnicas

Tiempo de máquina libre para las tareas de mantenimiento aún en las horas pico.

Un medio ambiente de pruebas capaz de simular un medio ambiente operacional.

Generadores de datos de prueba y verificadores

Librerias

Programas de diagnóstico en línea que ofrezcan trace, snap, dump y capacidad para cambios en línea.

Auditores de código para checar la estructura y la complejidad.

2) Experiencia Técnica

La labor de mantenimiento es muy compleja y de gran

responsabilidad, por lo que se debe buscar que la lleve a cabo personal con la experiencia debida. La practica tradicional de dedicar a el mantenimiento a los programadores novatos puede ser muy costosa ya que una modificación mal diseñada puede destuir la integridad del producto. No solo a las tareas de mantenimiento se debe dedicar una persona con experiencia en el desarrollo de Software sino que debe tener experiencia en el área de mantenimiento.

3) Técnicos de Administración

Con el fin de mantener la calidad y confiabilidad del producto se deben implantar un conjunto de técnicas que garanticen la adecuada adhesión a los estandares. Siendo fundamental para tal efecto el incorporar los siguientes cuatro técnicas.

Establecer las prioridades y metas buscadas.

Continuar exigiendo los mismos estandares que fueron usados para controlar la calidad durante el desarrollo.

Documentar el proceso de mantenimiento así como las modificaciones del Software.

Establecer auditorias periodicas de control de calidad así como revisiones de aceptación.

Lineamientos para el Mantenimiento del Software:

1. Técnicas

1.1 Use herramientas en el estado del arte tales como sistemas operativos modernos, auditores de código, generadores de referencias cruzadas, generadores de documentación, generadores de datos de prueba, programas de diagnóstico en línea, etc.

1.2 Emplee programación estructurada.

1.3 Use en el diseño una combinación de Top-down y Botton-up.

1.4 Desarrolle un plan de revalidación como parte del plan de modificación.

1.5 Use Lenguaje de Alto Nivel.

1.6 Esfuercese por lograr independencia de la máquina y código compatible con lenguaje estandar y estandares de desarrollo de Software.

1.7 Opte por un producto mantenible sobre uno eficiente.

1.8 Mantenga un conjunto de funciones comunes a ser empleadas en las modificaciones al código.

1.9 Busque técnicas de Ingeniería de Software utiles y adaptelas para su empleo en el medio ambiente del mantenimiento.

2. Control del Producto.

2.1 Defina las tareas de mantenimiento en términos de requerimiento en términos de requerimientos requeridos opcionales y futuros.

2.2 Modifique el Software con la idea de que continue siendo mantenible.

2.3 Opte por la claridad y sencillez sobre la integridad.

2.4 Efectue revisiones periodicas de control de calidad del producto.

2.5 Involucre al usuario en las revisiones.

2.6 Actualice la documentación de usuario.

(13)

2.7 Actualice la Documentación del sistema.

3. Control del Proyecto

3.1 Desarrolle un plan de soporte a mantenimiento y uselo para administrar el proceso de mantenimiento.

3.2 Defina explicitamente las metas de mantenimiento y las prioridades para el grupo de mantenimiento.

3.3 Produzca los requerimientos del usuario claros y concisos, a ser revisados periodicamente con el usuario.

3.4 Organice los grupos de mantenimiento con poca gente de buen nivel.

3.5 Mantenga una contabilidad clara de lo efectuado por cada miembro del grupo de mantenimiento.

3.6 Cree un plan de carrera, escala de salarios, así como beneficios y recompensas para los elementos del grupo de mantenimiento con alto desempeño.

3.7 No trate de substituir la buena administración con técnicas automatizadas.

(14)

3.8 Desarrolle librerías de programas para la administración del mantenimiento.

3.9 Evalue el éxito del mantenimiento en término de sus metas.

El Plan de Soporte a mantenimiento debe contener los siguientes puntos:

1. Procedimientos para reporte y corrección de fallas de Software.

2. Procedimiento de solicitud de cambios e implantación de estos.

3. Plan de protección para la Mantenibilidad y Control de Calidad en general del producto.

4. Procedimientos de Revalidación, incluyendo casos de prueba, datos de prueba y resultados de prueba.

5. Procedimientos de actualización de la Documentación.

6. Requerimientos de Soporte
   Configuración del equipo
   Herramientas Técnicas

Requerimientos de personal (15)

Soporte e Interfase con el usuario

Facilidades para prueba

Manuales de usuario y operación

Documentos generales del sistema

Listados del código fuente

7. Procedimientos para nueva liberación.

Lineamientos para el Análisis de requerimientos para el proceso de mantenimiento.

1. Determine el objetivo del mantenimiento para hacer la modificación.

2. Determine los requerimientos para la modificación con el usuario.

3. Defina los requerimientos en términos que se puedan - probar.

4. Considere cambios en tiempos de procesamiento, requerimientos de almacenamiento, probabilidad de error, - personal de operación, y otras versiones instaladas del Software.

(16)

5. Considere el efecto de las modificaciones en los aspectos de Ingeniería Humana del Software.

6. Identifique requerimientos en términos de contracciones y ó expansiones del Software existente.

7. Determine la compatibilidad de las modificaciones con otros cambios en el Software que pueden ocurrir en el futuro.

8. Justifique las modificaciones en términos costo, tiempo para implantar y riesgo de degradar la calidad del Software.

9. Busque la aprobación del usuario y el administrador antes de proceder a las especificaciones definitivas para la - modificación.

Lineamientos para la fase de especificación del proceso de mantenimiento.

1. Desarrolle especificaciones para las modificaciones siguiendo los mismos estandares usados para desarrollar las especificaciones durante el desarrollo del Software.

2. Describa las especificaciones para la modificación en términos que se puedan probar y que incluyan métodos de prueba.

3. Identifique programas existentes, módulos y ó paquetes a ser empleados durante la modificación.

4. Examine el impacto de la modificación en el Software instalado y los recursos necesarios para soportar la modificación.

5. Busque la aprobación del usuario y el administrador de las especificaciones para la modificación antes de proceder al diseño.

Lineamientos para efectuar la fase de diseño del proceso de mantenimiento.

1. Examine diseños alternativos buscando un diseño para la modificación que sea compatible con la filosofía original del diseño.

2. Busque la simplicidad en el diseño; seleccione la alternativa de diseño que modifique el menor número de mó

dulos menos complejos y el menor número de variables globales.

3. Documente el diseño y el proceso de diseño para las modificaciones siguiendo el mismo estandar usado cuando inicialmente se desarrollo el diseño.

4. Considere la factibilidad del diseño para la modificación por su efecto en el resto del sistema.

5. Evalue la generalidad del diseño en términos de su habilidad para ser usado en varias versiones del Software con diferentes sistemas operativos y en diferentes configuraciones de Hardware.

6. Evalue la flexibilidad del diseño en términos de su habilidad para aislar funciones especializadas en módulos separados y proveer interfases entre los módulos que sean insensitivos a cambios posteriores.

7. Busque la aprobación del usuario y el administrador del diseño antes de proceder a la implantación.

Los lineamientos para la implantación buscan cubrir dos objetivos

el primero traducir correctamente el diseño de la modificación en código bien estructurado, el segundo minimizar el impacto - de la modificación en el resto del Software.

Lineamientos que buscan conservar la correcta estructura del código:

1. Use programación estructurada y estandares de codificación.

2. Use herramientas en el estado del arte como apoyos de programación en línea tablas de decisión, etc.

3. Documente todos los cambios en el código y conserve - versiones del código y conserve versiones del código - antes de modificar.

4. Duplique el código en vez de crear rutinas comunes.

5. Codifique cambios en una manera que no degrade la - mantenibilidad del Software, por llevar la eficiencia de masiado lejos.

6. Opte por Ingeniería Humana en vez de eficiencia.

7. Registre el proceso de mantenimiento en el UDF.

8. Actualice los manuales de usuario y operación así como los documentos del sistema para que reflejen la modificación.

9. Efectue una revisión de código para garantizar la conservación de la calidad y el cumplimiento de los estandares.

Lineamientos que ayudan a evitar la presencia de efectos de segundo orden:

1. Cambie lo menos posible
2. Cambie el menor número de variables posibles, en particular, las menos variables globales posibles.

3. Cuando un módulo común sea cambiado, examine todos los módulos que lo invoque ; para determinar si estos son afectados por el cambio.

4. Cuando una variable local se altere, examine el código en el módulo que haga referencia a ella para determinar si otra función en el módulo o fuera de el se afecta.

5. Cuando una variable global es cambiada, examine todos los módulos que hacen referencia a ella para determinar

su efecto.

6. Cuando se tengan que hacer múltiples cambios a un sistema ordenelos en la siguiente forma:

   i) Agrupe los cambios por módulo

   ii) Planee la secuencia de módulos a ser cambiados siguiendo la estrategia del más sencillo primero.

   iii) Modifique un módulo a la vez

   iv) Por cada módulo cambiado determine los efectos de segundo orden de ese cambio antes de continuar la secuencia de cambios.

7. Use el efecto de segundo orden y la medida de complejidad para determinar la dificultad de hacer un cambio.

Lineamientos para la Revalidación:

1. Revalide el Software empleando pruebas unitarias, pruebas de integración, prueba del sistema y pruebas de aceptación, todos estos adaptados de la fase de pruebas durante el desarrollo.

2. Efectue pruebas unitarias a cada módulo modificado. Cuando sea posible use las pruebas y datos empleados en el desarrollo compare los resultados para encontrar las discrepancias.

3. Efectue pruebas de regresión conforme cada módulo modificado es reintegrado al sistema de Software, para determinar si alguna otra parte del Sistema ha sido afectada por la modificación.

4. En base al perfil de complejidad, ejecute pruebas de integración, de sistema y de aceptación que se concentren en las partes más complejas del Software.

5. Realice pruebas de sistema adaptando los datos de prueba de las pruebas de desarrollo y compare resultados para encontrar discrepancias.

6. Realice pruebas de aceptación empleando las pruebas y datos de las pruebas de desarrollo así como pruebas suministradas por el usuario.

7. Use herramientas de prueba en el estado del arte.

8. Integre la Historia de pruebas de Modificación.

Applying the Technique of Configuration Management to Software

(23)

- 23

As the field of data processing continues to expand, the need to discipline the growth of computer programs (software) becomes more obvious. One management technique which promises to be effective is Configuration Management (C.M.). While this method was originally designed to control hardware production, its principles can be tailored and refined to relate to the development and production of computer software.

The Growth of Software

In the last 10 years the field of computer software has expanded to make use of the greater speed and power of increasingly sophisticated computer hardware. Today high-level programming languages and complex operating systems are considered the norm. The natural path of growth has been to more diverse applications, including:

- Large defense systems.
- Air traffic control systems.
- Medical software.

The complexity of the applications has also grown, resulting in software containing several hundred thousand lines of code.

The growth of the software industry has precipitated a rise in costs for software development, such that the expense of hardware is no longer the prime concern. For example, in 1971 the Air Force estimated their software expenses to be $1-1.5 billion, which is about three times the expense of computer hardware.[1] The World Wide Military Command and Control System is estimated to cost $42 to $206 million for hardware and $722 million for software.[2] Due to the cost of programming for larger and more complex software systems, software costs will continue to increase over hardware costs, as shown in Figure 1[3] (see page 24).

In general, poor planning can be blamed for most of the software industry's inability to cope with these new demands and rapid growth. Projects were initiated without a clear goal; thus, as programmers coded, they made their own assumptions about the purpose of the programs and this often adversely affected the final product. Even when the goals were clearly specified, the development process suffered from inappropriate planning. Some managers and programmers measured progress in terms of the number of lines of code produced, and rushed to get something running as soon as possible. Auspicious beginnings proved misleading as unforeseen difficulties emerged. To solve the problem, much of the early code was rewritten, and eventually caused delays in delivery dates or the delivery of

by Rita McCarthy
Burroughs Corporation
Goleta, Calif.

[1] *B. W. Boehm, "Software and Its Impact: A Quantitative Assessment,"* Datamation, *May 1973, pp. 48-49.*

[2] *Phil Hirsch, "GAO Hits Wimmix Hard; FY'72 Funding Prospects Fading Fast,"* Datamation, *March 1, 1971, p. 41.*

[3] *B. W. Boehm, "Software and Its Impact: A Quantitative Assessment,"* Datamation, *May 1973, pp. 48-49.*

incomplete products. Either alternative meant higher costs.

The final product was often characterized by a lack of reliability, which refers to the ability of a program to produce correct results when given a specific input. The consequences of such errors ranged from minor to disastrous. Minor problems do not have destructive side effects, but are often extremely annoying; for example, an incorrect page control on a printed report that causes a blank form on every other page. The failure of the Mariner I interplanetary probe, however, resulted from a disastrous error: the absence of one bar over a letter in a computational equation resulted in an unrecoverable problem, leaving no alternative but to destruct the $18.5 million rocket shortly after launch.[1]

The user of new software typically experienced very high error rates. Even after the first obvious errors were corrected and the product became operational, users continued to have a nearly constant pattern of failure. This was attributed to new errors introduced while correcting other problems and to the discovery of dormant errors as previously unused functions were tried. This phenomenon is discussed in an article by Jerry Ogdin, and is represented in Figure 2.[2] He also pointed out that modifications to already operational programs resulted in a rash of new failures, thus explaining the peak in the figure. While there is no single solution to all of these problems, the disciplines embodied in Configuration Management do provide a global framework in which specific solutions can be combined and monitored to attack specific parts of a problem.

Defining Configuration Management

It is much easier to define Configuration Management by inspecting each word individually. "Configuration" applies to an interrelated group of programs that operate as a system. The term applies equally as well to the interrelating modules of one program. "Management" is the process of establishing and organizing objectives, followed by planning and employing resources to accomplish these objectives. The term "Configuration Management" is all-embracing, covering the management of every detail of a software project from inception through development to completion and maintenance of the product.

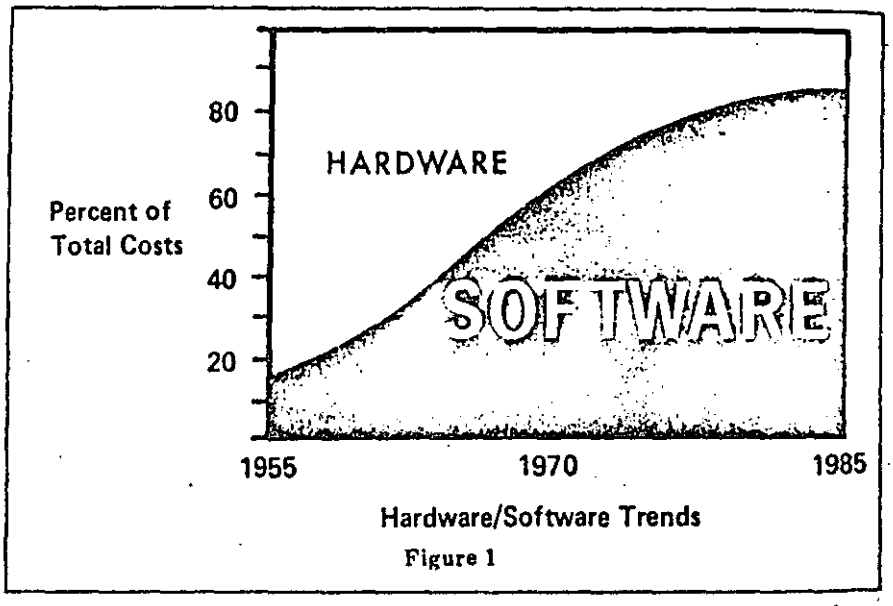
The objective of Configuration Management is to control the costs and the reliability of a software system. To achieve this, C.M. focuses on three areas:

- Identification.
- Control.
- Accounting.

The importance of these focal points is that they are directed

[1] *J. L. Santer, "Reliability in Computer Programs,"* Mechanical Engineering, *February 1969, p. 24.*

[2] *Jerry L. Ogdin, "Designing Reliable Software,"* Datamation, *July 1972, pp. 71-78.*



Hardware/Software Trends

Figure 1

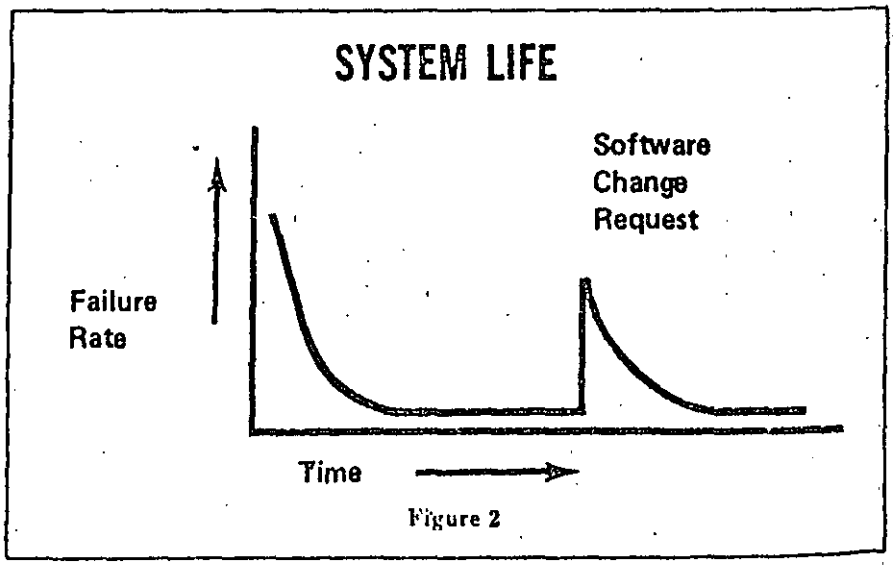


Figure 2

(95)

toward all people involved with the product. C.M. attempts to combine the user, the administrators, the code developers, and the validation (test) team within the same framework, as shown in Figure 3.

Identification

The philosophy of identification is to determine the exact nature of the problem, a suitable method of solution and the goals to guide the project before any actual coding begins. The assumption is that clear and complete information produces a cohesive, reliable product.

The identification process is concerned with the documentation of a design in progressively finer levels of detail. This is accomplished by a series of reports which are individually described in the following.

First Report. An initial report on System Performance and Design Requirements must come from the customer and cover the following points:

- Definition of the desired product.
- Specific functions to be performed.
- Product environment, *e.g.*, hardware and operating systems, and limits on resources, *e.g.*, memory and storage media.
- Expected level of performance (speed).
- Reliability requirements (error tolerance).
- Maintenance and support needs.

The first report is considered an important reliability tool. Features in the software that do not function according to expectations are commonly classified as errors. Therefore, it is imperative the customer be specific at this point, if the ultimate product is to be responsive to his needs. There should be no incomplete, conflicting, or uncertain terminology, which may lead to later problems in the software development process.

Second Report. As a reply to the initial report, the Part I Specification (Performance and Design Requirements for Computer Programs) is prepared. The de-

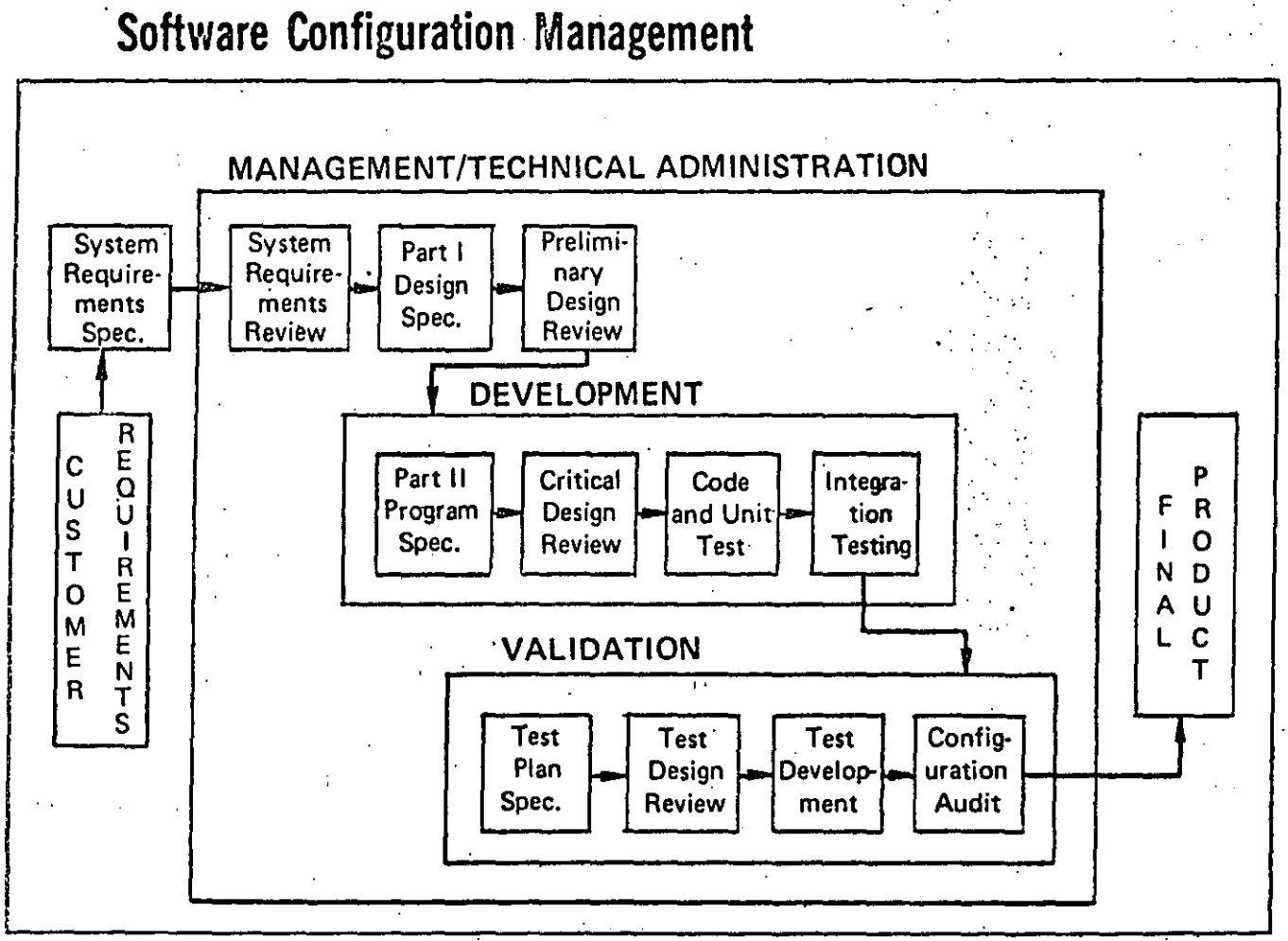


Figure 3

velopment people herein outline their method of solving the problem and their plan for ensuring reliability. Topics covered are discussed below.

• *General information flow.* This should include block diagrams showing input, processing and outputs indicating the sequence of events. There should be enough detail to provide the initial material for further program design.

• *Interface requirements.* An interface is a common boundary between parts of the system. For example, if one program accepts a file as input created by a previous program, the file becomes the path of their interface; errors can obviously occur at this point. Therefore, these interdependencies must be explicitly defined to assure that both programs are making the same assumptions about their interface. In some systems it may also be necessary to clarify the interface between the hardware and the software.

• *Expendability plan.* To plan a system which is easily modified and maintained, it is necessary to separate into independent areas those functions whose definitions are likely to change or expand. Thus, future modifications will be well isolated from other portions of the system and side-effect errors will not be introduced into already working code.

• *Test plan.* A test plan should be outlined for the development programmers and an independent validation (test) group. The development people must consider the testability of their design and ensure that code primitives can be exhaustively tested before the next higher level of code is added. Early location and correction of errors results in a much more reliable program.

Specifications should be included for diagnostic tools that aid in the location of errors, *e.g.*, execution time monitors or traces, and formated memory or program dumps. Depending on the project, this list may be expanded to include hardware help such as readout displays. The development of these tools takes time, but it is more than recovered in assisting programmers to quickly and accurately locate their errors.

A solid test plan should provide for an independent validation team to be established at the beginning of the project. The responsibilities of this group are to follow the complete design of the product and independently specify, design and implement a comprehensive functional test library to be used in qualifying the final product before its release to the customer. The establishment of this independent group, which is less likely to make assumptions about the validity of the code, is a key step in assuring software reliability.

• *Reliability plan.* Programming standards or style to be imposed on all code should be outlined. A great deal has been learned recently about coding practices that increase program reliability (see Dijkstra [6] and Mills [7]). One proposed practice is called "Structured Programming," which involves dividing a complex program into progressively smaller modules, each of which has a well-defined task. The most refined modules are small and logically straightforward, have limited control structures and one entry and exit point, and are named by their function. The conciseness of the modules allows the programmer to use formal mathematics to prove the correctness of the code (see Floyd [8] and London [9] for insights into the technique of proof of correctness).

While the primary intention of this second report is to interpret the problem and propose a solution to the customer, it also establishes an environment in which the solution can be achieved. Programmers and managers can consider testability, reliability and expandability on an equal priority with the process of coding.

Third Report. The Part II Specification (Product Specification for Computer Programs) is a complete and detailed technical description of the computer program(s) which describes how the solution to the problem is accomplished through the hierarchy of the code. Each refined module of the code is discussed. The details for each module include a description of its function, its expectations about global data and its effect on that data, and a description of its input and output. Such a design report essentially solves the entire programming problem in a narrative fashion before any coding begins. While this may be a trying experience for man-

[6] *E. W. Dijkstra, "Notes on Structured Programming," in* Structured Programming, *Academic Press, New York, 1972.*

[7] *H. D. Mills, "Structured Programming in Large Systems," in* Debugging Techniques in Large Systems, *R. Rustin (ed.), Prentice-Hall, Inc., Englewood Cliffs, New Jersey.*

[8] *R. W. Floyd, "Assigning Meaning to Programs,"* Proc. Symp. in Applied Mathematics, *19, J. T. Schwartz (ed.), American Math. Soc., Providence, Rhode Island, 1967, pp. 19–32.*

[9] *R. L. London, "Proving Programs Correct: Some Techniques and Examples," BIT Volume 10, 1970, p. 168.*

agers accustomed to seeing lines of code and not technical documents, the results are encouraging.

The advantages of a good product design are twofold. First, it provides time for separating the complex problem into smaller, well-defined modules which are easier to understand, code, test and eventually modify. Secondly, it makes programmers confident of their approach, freeing them to concentrate on the accuracy of the code they write.

Final Report. The preparation of a Test Plan Specification should coincide with the preparation of the Part II Specification. This final report outlines the approach to be taken by the validation team in determining the functional accuracy and acceptable performance level of the software. The goal should be to provide for a test library of programs that are easy to operate, provide repeatable sequences of events, and are self-checking in nature. The detailed description of each test series includes: purpose, range of input data, expected output, priority and time required to do the test. A test plan that is complete in the coverage of the specified product can enhance confidence in the final product during the validation process.

Control

The control process is concerned with changes to be introduced into the software product. Because software is much more dynamic than hardware, there will always be new features to be incorporated, corrections to be made and code efficiency to be considered. Configuration control provides for these situations and establishes procedures for introducing proposals for change, evaluating these proposals, monitoring the status of the resulting action, and documenting the effect of any change. Figure 4 is an example of a control procedure that incorporates several very worthwhile features:

- Requests can have variable origins.
- All requests follow the same steps.
- Management screens all requests to determine their impact on work loads.
- Requests and their status are maintained in a data base accessible to all.
- The data base can be used for reporting and accumulating statistics. It is possible to determine such things as the number and origin of errors reported, the amount of code changed during a given period of time, and the impact of these changes on other parts of the system.

The natural result of the control scheme is good communication. All levels of managers, de-

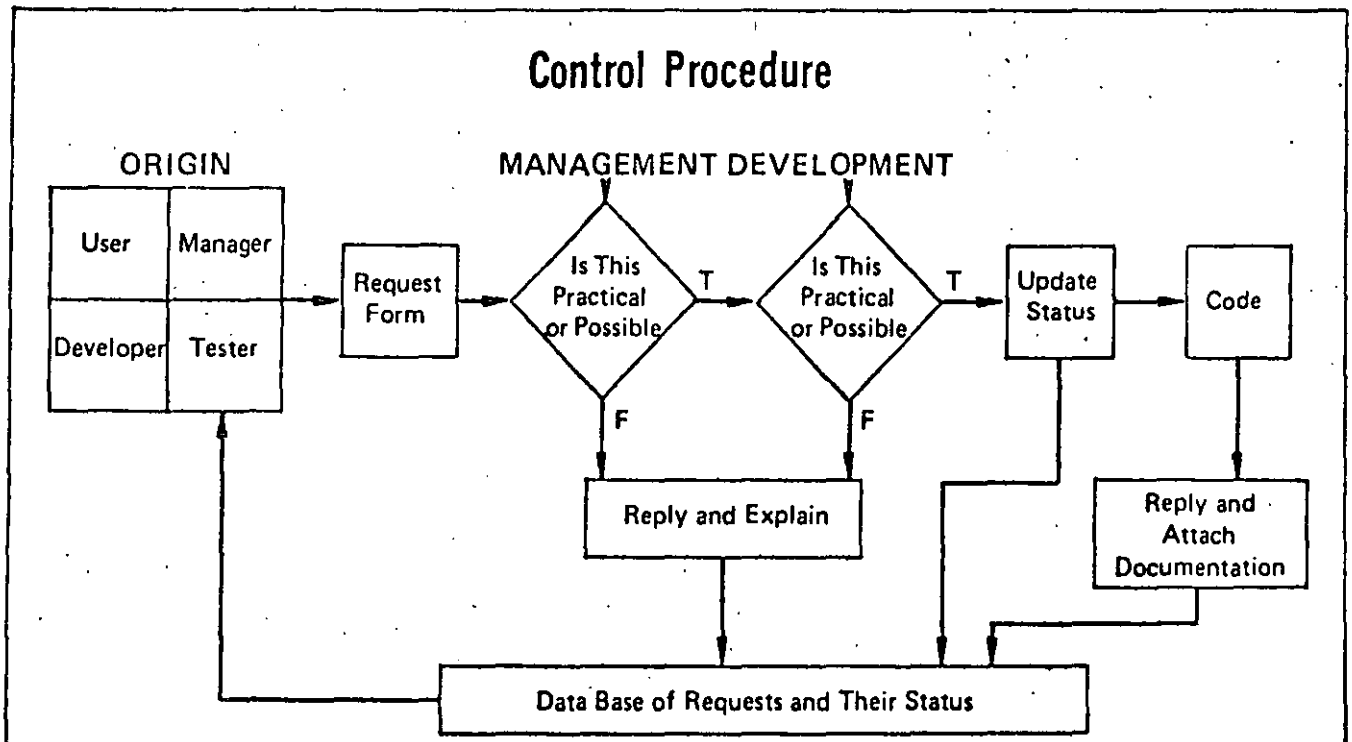


Figure 4

velopers, validators, and other personnel have a better chance of doing their work correctly and efficiently.

Accounting

Configuration accounting provides continual visibility into software development through program reviews conducted at major milestones throughout the development, as well as through software documentation and product validation.

Formal program reviews are held for all involved with the software. The System Design Review covers the functional requirements of the system and the environment in which the product is to work. The Preliminary Design Review covers the overall design, plus the plans for expandability, testability and reliability. The Critical Design Review covers the technical details of each program of the system.

The usefulness of the reviews for each attendee varies depending on his position within the project. The development people are generally able to conceptualize their approach much better after a review. In summarizing the task, they are forced to reexamine all previous thought processes, and the strengths and weaknesses of the design become more apparent. The reviews are also useful for discussing the interfaces in software configurations of more than one program, assuring that the same interface procedure is being used with all of the programs.

During a review, managers gain insight into the progress of the project. An incomplete or disorganized presentation may reflect the actual state of affairs. It may even be necessary to reschedule a review, requiring the development people to tie together loose ends. Participation in the reviews by the validation people helps them to better understand the product, enhancing the probability of thorough testing on their part and valid criticism of the generated documentation.

Documentation

Management of the software documentation is another important aspect of configuration accounting. Without documentation, there is no history of the details, and it becomes difficult for programmers to compare approaches and verify interfaces. Furthermore, management has no visible sign of the progress of the programmers.

A part of the documentation related to the software is provided in the specification written during the identification procedure. This material describes the goals of the development effort and the technical details of the programs. Further, it is important to prepare a user's manual to describe the operational interface to the software system, enabling users to treat the software like a black box and ignore its internal workings.

The final point of accounting is also the last step of the development process: the Product Configuration Audit. Manuals, listings and programs (source and object) are delivered to the validation team, which has developed a complete test library paralleling the development of the actual software. The team is responsible for the quality assurance of the product and must determine if the software is performing according to expectations. Change requests must be submitted for problems that are uncovered, and testing continues. If major problems occur, it may be necessary to suspend testing until the change requests are processed and new software is submitted.

The Product Configuration Audit measures the success of previous work. Therefore, complete records must be kept relating to the number of problems found and the progress made toward completion of the audit process. Determining the number and status of problems is relatively easy since this information is recorded in the data base of change requests.

Progress toward completing the audit is facilitated by a checklist of items to be tested. As each feature is verified, the date of checkout and the results are recorded. If there are problems, the feature is requalified after it is fixed by development, and the final checkout is recorded. When all tests are completed and the important problems are fixed, the software is delivered to the user with the user's manual and notification of any problems.

Conclusion

The three essential requirements of identification, control and accounting provide a comprehensive base for a Configuration Management program, where details are flexibly tailored to meet the needs and goals of specific projects. The advantages to be accrued by Configuration Management are numerous, and any experience with its methods will help in facing the growing challenges for software in the future. □

(29)

THE UNIT DEVELOPMENT FOLDER (UDF): AN EFFECTIVE MANAGEMENT TOOL FOR SOFTWARE DEVELOPMENT

Prepared by

Frank S. Ingrassia

October 1976

TRW

DEFENSE AND SPACE SYSTEMS GROUP

SYSTEMS ENGINEERING AND INTEGRATION DIVISION

ONE SPACE PARK, REDONDO BEACH, CALIFORNIA 90278

1 from TRW Technical Report TRW-SS-76-11.

(30)

ABSTRACT

This paper describes the content and application of the Unit Development Folder, a structured mechanism for organizing and collecting software development products (requirements, design, code, test plans/data) as they become available. Properly applied, the Unit Development Folder is an important part of an orderly development environment in which unit-level schedules and responsibilities are clearly delineated and their step-by-step accomplishment made visible to management. Unit Development Folders have been used on a number of projects at TRW and have been shown to reduce many of the problems associated with the development of software.

THE UNIT DEVELOPMENT FOLDER (UDF): AN EFFECTIVE MANAGEMENT TOOL FOR SOFTWARE DEVELOPMENT

(3)

One of the main side effects resulting from the invention of computers has been the creation of a new class of frustrated and harried managers responsible for software development. The frustration is a result of missed schedules, cost overruns, inadequate implementation and design, high operational error rates and poor maintainability, which have historically characterized software development. In the early days of computer programming, these problems were often excused by the novelty of this unique endeavor and obscured by the language and experience gap that frequently existed between developers and managers. Today's maturity and the succession of computer-wise people to management positions does not appear to have reduced the frustration level in the industry. We are still making the same mistakes and getting into the same predicaments. The science of managing software development is still in its infancy and the lack of a good clear set of principles is apparent.

The problems associated with developing software are too numerous and too complex for anyone to pretend to have solved them, and this paper makes no such pretensions. The discussion that follows describes a simple but effective management tool which, when properly used, can reduce the chaos and alleviate many of the problems common to software development. The tool described in this paper is called the Unit Development Folder (UDF) and is being used at TRW in software development and management.

What is a UDF? Simply stated, it is a specific form of development notebook which has proven useful and effective in collecting and organizing software products as they are produced. In essence, however, it is much more; it is a means of imposing a management philosophy and a development methodology on an activity that is often chaotic. In physical appearance, a UDF is merely a three-ring binder containing a cover sheet and is organized into several predefined sections which are common to each UDF. The ultimate objectives that the content and format of the UDF must satisfy are to:

(1) Provide an orderly and consistent approach in the development of each of the units of a program or project

(2) Provide a uniform and visible collection point for all unit documentation and code

(3) Aid individual discipline in the establishment and attainment of scheduled unit-level milestones

(4) Provide low-level management visibility and control over the development process

Figure 1 illustrates the role of the UDF in the total software development process.

If one follows a fairly standard design approach, the completion of the preliminary design activity marks the point at which UDFs are created and initiated for all units comprising the total product to be designed and coded. Therefore, the first question to be answered is, "What is a unit?" It was found that, for the purpose of implementing a practical and effective software development methodology to meet the management objectives stated earlier, a unique element of software architecture needed to be defined. This basic functional element is designated a "unit" of software and is defined independently of the language or type of application. Experience has indicated that it is unwise to attempt a simple-minded definition which will be useful and effective in all situations. What can be done is to bound the problem by means of some general considerations and delegate the specific implementation to management judgment for each particular application.

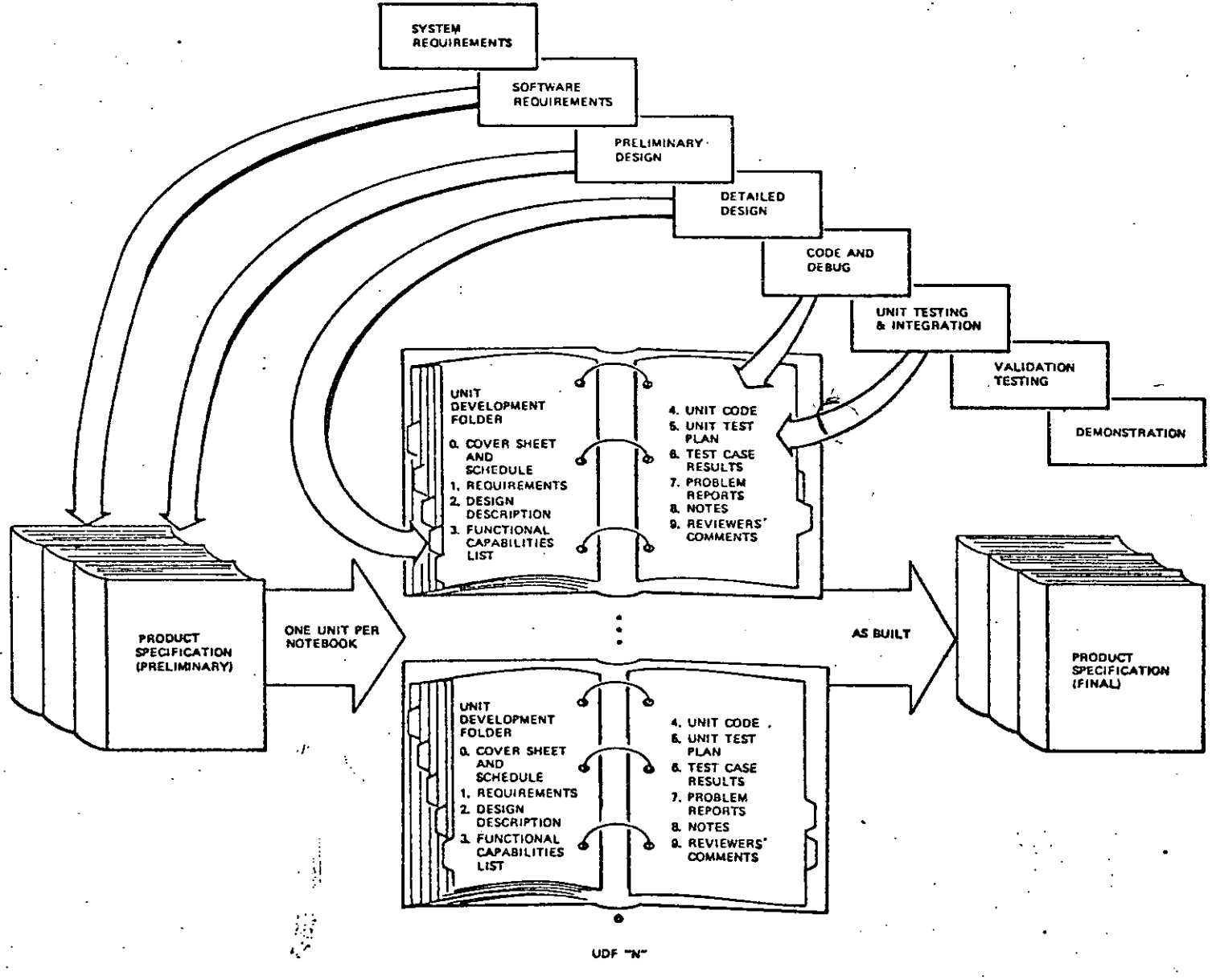


Figure 1. The UDF in the Development Process

(33)

At the lower end of the scale a "unit" can be defined to be a single routine or subroutine. At the upper end of the scale a "unit" may contain several routines comprising a subprogram or module. However it is defined, a unit of software should possess the following characteristics:

(1) It performs a specific defined function

(2) It is amenable to development by one person within the assigned schedule

(3) It is a level of software to which the satisfaction of requirements can be traced

(4) It is amenable to thorough testing in a disciplined environment.

The key word in the concept is manageability – in design, development, testing and comprehension.

A natural question that may arise at this point is, "Why should a unit contain more than one routine?" The assumption for this proviso is that the design and development standards impose both size and functional modularity. Since functional modularity can be defined at various levels, the concept can become meaningless if it is not accompanied by a reasonable restriction of size. Consequently, the maximum size constraint on routines may sometimes result in multiple-routine units.

The organization and content of a UDF can be adapted to reflect local conditions or individual project requirements. The important considerations in the structuring of a UDF are:

(1) The number of subdivisions is not so large as to be confusing or unmanageable

(2) Each of the sections contributes to the management and visibility of the development process

(3) The content and format of each section are adequately and unambiguously defined

(4) The subdivisions are sufficiently flexible to be applicable to a variety of software types

(5) The individual sections are chronologically ordered as nearly as possible.

The last item is very important since it is this aspect of the UDF that relates it to the development schedule and creates an auditable management instrument. An example of a typical cover sheet for a UDF is shown in Figure 2; the contents of each section will be briefly described in subsequent paragraphs.

The UDF is initiated when requirements are allocated to the unit level and at the onset of preliminary design. At this point it exists in the skeletal form of a binder with a cover sheet (indicating the unit name and responsible custodian) and a set of section separators. The first step in the process is for the responsible work area manager to integrate the development schedules and responsibilities for each of his UDFs into the overall schedule and milestones of the project. A due date is generated for the completion of each section and the responsibility for each section is assigned. The originators should participate in establishing their interim schedules within the constraints of the dictated end dates.

The organization and subdivisions of the UDF are such that the UDF can accommodate a variety of development plans and approaches; it can be used in a situation where one person has total responsibility, or in the extreme where specialists are assigned to the particular sections. However, in the one-man approach it is still desirable that certain sections, indicated in the following discussion, be assigned to other individuals to gain the benefits of unbiased reviews and assessments.

The development of the UDF is geared to proceed logically and sequentially, and each section should be as complete as possible before proceeding to the next section. This is not always possible, and software development is usually an iterative rather than a linear process. These situations only serve to reinforce the need for an ordered process that can be understood and tracked even under adverse conditions.

Once a specific outline and UDF cover sheet have been established, it is imperative that the format and content of each section be clearly and completely defined as part of the project/company standards to avoid ambiguity and maintain consistency in the products. The following discussion expands and describes the contents of the UDF typified by the cover sheet shown in Figure 2.

Section 0. COVER SHEET AND SCHEDULE

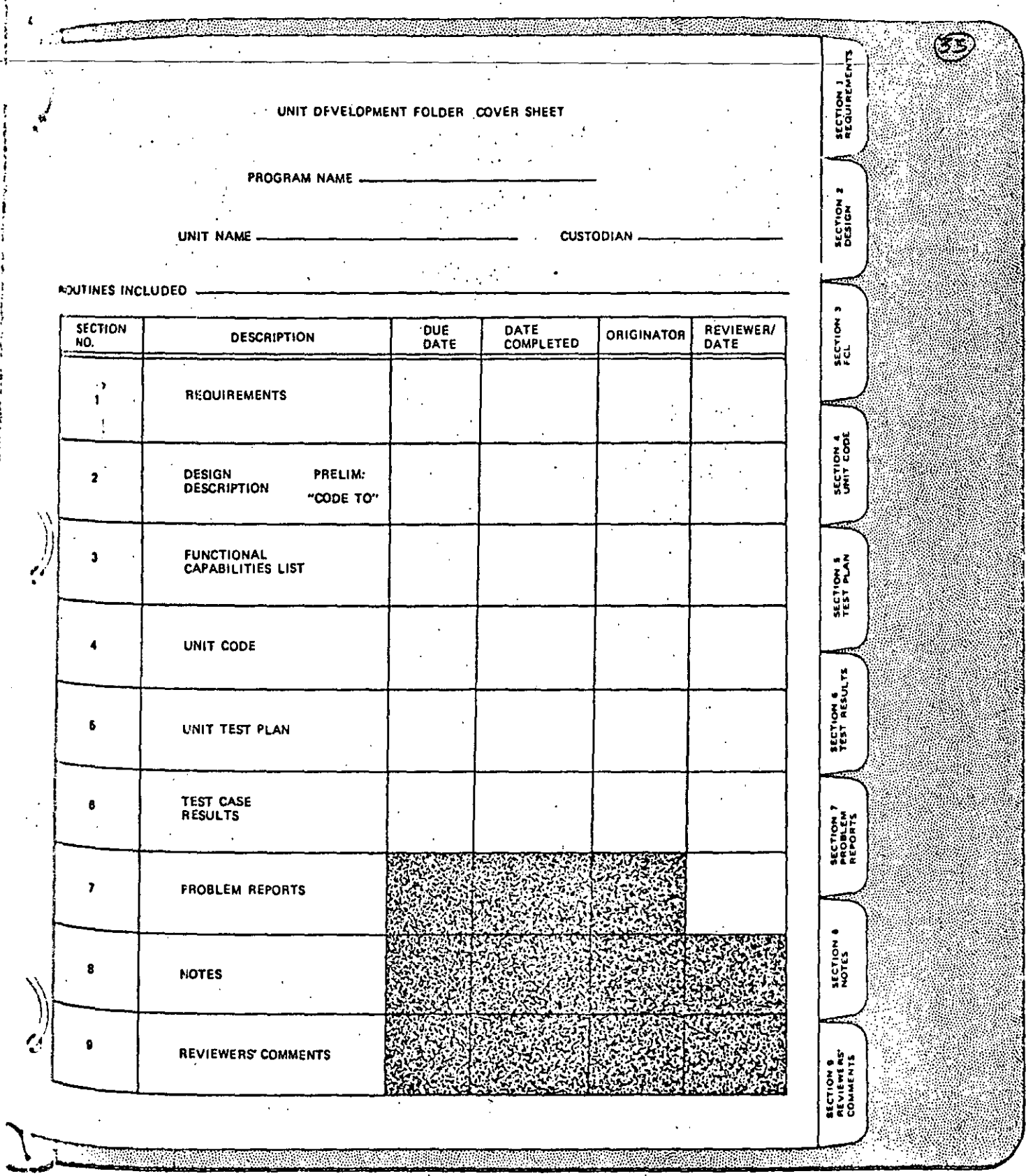
This section contains the cover sheet for the unit, which identifies the routines included in the UDF and which delineates, for each of the sections, the scheduled due dates, actual completion dates, assigned originators and provides space for reviewer sign-offs and dates. In the case of multiple-routine units, it may be advisable to include a one-page composite schedule illustrating the section schedules of each item for easy check-off and monitoring. Following each cover sheet, a UDF Change Log should be included to document all UDF changes subsequent to the time when the initial development is completed and the unit is put into a controlled test or maintenance environment. Figure 3 illustrates a typical UDF Change Log.

Section 1. REQUIREMENTS

This section identifies the baseline requirements specification and enumerates the requirements which are allocated for implementation in the specific unit of software. A mapping to the system requirements specification (by paragraph number) should be made and, where practical, the statement of each requirement should be given. Any assumptions, ambiguities, deferrals or conflicts concerning the requirements and their impact on the design and development of the unit should be stated, and any design problem reports or deviations or waivers against the requirements should be indicated. In addition, if a requirement is only partially satisfied by this unit it will be so noted along with the unit(s) which share the responsibility for satisfaction of the requirement.

Section 2. DESIGN DESCRIPTION

This section contains the current design description for each of the routines included in the UDF. For multiple routine units, tabbed subsection separators are used for handy



UNIT DEVELOPMENT FOLDER COVER SHEET

PROGRAM NAME _____

UNIT NAME _____ CUSTODIAN _____

ROUTINES INCLUDED _____

SECTION NO.	DESCRIPTION	DUE DATE	DATE COMPLETED	ORIGINATOR	REVIEWER/ DATE
1	REQUIREMENTS				
2	DESIGN DESCRIPTION PRELIM: "CODE TO"				
3	FUNCTIONAL CAPABILITIES LIST				
4	UNIT CODE				
5	UNIT TEST PLAN				
6	TEST CASE RESULTS				
7	PROBLEM REPORTS				
8	NOTES				
9	REVIEWERS' COMMENTS				

Figure 2. UDF Cover Sheet and Layout

(26)

UDF CHANGE LOG

UNIT NAME _____ VERSION _____ CUSTODIAN _____

DATE	DPR/DR Number	Section(s) Affected and Page Numbers	Retest Method	Mod No.

NOTE: This revision change log is to be used for all changes made in the UDF after internal baseline (i.e., subsequent to mod number assignment). It is inserted immediately after the coversheet.

Figure 3. Example of a UDF Change Log

indexing. A preliminary design description may be included if available; however, the end item for this section is detailed design documentation for the unit, suitable to become (part of) a "code-to" specification. The format and content of this section should conform to established documentation standards and should be suitable for direct inclusion into the appropriate detailed design specification (Figure 1). Throughout the development process this section represents the current, working version of the design and, therefore, must be maintained and annotated as changes occur to the initial design. A flowchart is generally included as an inherent part of the design documentation. Flowcharts should be generated in accordance with clear established standards for content, format and symbol usage.

When the initial detailed design is completed and ready to be coded, a design walk-through may be held with one or more interested and knowledgeable co-workers. If such a walk-through is required, the completion of this section may be predicated on the successful completion of the design walk-through.

Section 3. FUNCTIONAL CAPABILITIES LIST

This section contains a Functional Capabilities List (FCL) for the unit of software addressed by the UDF. An FCL is a list of the testable functions performed by the unit; i.e., it describes what a particular unit of software does, preferably in sequential order. The FCL is generated from the requirements and detailed design prior to development of the unit test plan. Its level of detail should correspond to the unit in question but, as a minimum, reflect the major segments of the code and the decisions which are being made. It is preferred that, whenever possible, functional capabilities be expressed in terms of the unit requirements (i.e., the functional capability is a requirement from Section 1 of the UDF). Requirements allocated to be tested at the unit level shall be included in the FCL. The FCL provides a vector from which the TEST CASE/REQUIREMENTS/FCL matrix (Figure 4) is generated. The FCL should be reviewed and addressed as part of the test plan review process.

The rationale for Functional Capabilities Lists is as follows:

(1) They provide the basis for planned and controlled unit-level testing (i.e., a means for determining and organizing a set of test cases which will test all requirements/functional capabilities and all branches and transfers).

(2) They provide a consistent approach to testing which can be reviewed, audited, and understood by an outsider. When mapped to the test cases, they provide the rationale for each test case.

(3) They encourage another look at the design at a level where the "what if" questions can become apparent.

Section 4. UNIT CODE

This section contains the current source code listings for each of the routines included in the unit. Indexed subsection separators are used for multiple routine units. The completion date for this section is the scheduled date for the first error-free compilation or assembly when the code is ready for unit-level testing. Where code listings or other

(38)

TEST CASE/REQUIREMENTS/FCL MATRIX

REQUIREMENTS DOCUMENT _____

DATE _____

Req'ts Paragraph Number	FCL NO.	TEST CASE NUMBER 1	2	3	4	5	6	7	8	9	0	OTHER ROUTINES

INSTRUCTIONS: Mark an X in the appropriate box when a particular test case fully tests a particular requirement. Mark a "P" when a test partially tests a requirement. If a requirement is partially tested in another routine, mark a "P" in the "other routines" column. If more space is required, attach additional copies of this figure.

Figure 4. Example Test Case/Requirements/FCL Matrix

relevant computer output are too large or bulky to be contained in a normal three-ring binder, this material may be placed in a separate companion binder of appropriate size which is clearly identified with the associated UDF. In this event, the relevant sections of the UDF will contain a reference and identification of the binder with a history log of post-baselined updates. Figure 5 illustrates a typical reference form.

An independent review of the code may be optional; however, for time-critical or other technically important units, a code walk-through or review is recommended.

Section 5. UNIT TEST PLAN

This section contains a description of the overall testing approach for the unit along with a description of each test case to be employed in testing the unit. The description must identify any test tools or drivers used, a listing of all required test inputs to the unit and their values, and the expected output and acceptance criteria, including numerical outputs and other demonstrable results. Test cases shall address the functional capabilities of the unit, and a matrix shall be placed into this section which correlates requirements and functional capabilities to test cases. This matrix will be used to demonstrate that all requirements, partial requirements, and FCLs of the unit have been tested. An example of the test case matrix is shown in Figure 4. Check marks are placed in the appropriate squares to correlate test cases with the capabilities tested. Sufficient detail should be provided in the test definition so that the test approach and objectives will be clear to an independent reviewer.

The primary criteria for the independent review will be to ascertain that the unit development test cases adequately test branch conditions, logic paths, input and output, error handling, a reasonable range of values and will perform as stipulated by the requirements. This review should occur prior to the start of unit testing.

Section 6. TEST CASE RESULTS

This section contains a compilation of all current successful test case results and analyses necessary to demonstrate that the unit has been tested as described in the test plan. Test output should be identified by test case number and listings clearly annotated to facilitate necessary reviews of these results by other qualified individuals. Revision status of test drivers, test tools, data bases and unit code should be shown to facilitate retesting. This material may also be placed in the separate companion binder to the UDF.

Section 7. PROBLEM REPORTS

This section contains status logs and copies of all Design Problem Reports, Design Analysis Reports and Discrepancy Reports (as required) which document all design and code problems and changes experienced by the unit subsequent to baselining. This ensures a clear and documented traceability for all problems and changes incurred. There should be separate subsections for each type of report with individual status logs that summarize the actions and dispositions made.

LISTINGS/TEST RESULTS

SEE SEPARATE NYLON PRONG BINDER IDENTIFIED AS

_____FOR CODE LISTINGS

OR TEST RESULTS.

HISTORY LOG

CODE MOD NUMBER	DATE	REVIEWED BY

Figure 5. Example Reference Log for Separately-Bound Material

Section 8. NOTES

This section contains any memos, notes, reports, etc., which expand on the contents of the unit or are related to problems and issues involved.

Section 9. REVIEWERS' COMMENTS

This section contains a record of reviewers' comments (if any) on this UDF, which have resulted from the section-by-section review and sign-off, and from scheduled independent audits. These reviewers' comments are also usually provided to the project and line management supervisors responsible for development of the unit.

SUMMARY

(42)

The UDF concept has evolved into a practical, effective and valuable tool not only for the management of software development but also for imposing a structured approach on the total software development process. The structure and content of the UDF are designed to create a series of self-contained systems at the unit level, each of which can be easily observed and reviewed. The UDF approach has been employed on several software projects at TRW and continues to win converts from the ranks of the initiated. The concept has proved particularly effective when used in conjunction with good programming standards, documentation standards, a test discipline and an independent quality assurance activity.

The principal merits of the UDF concept are:

(1) It imposes a development sequence on each unit and clearly establishes the responsibility for each step. Thus the reduction of the software development process into discrete activities is logically extended downward to the unit level.

(2) It establishes a clearly-discernible timeline for the development of each unit and provides low-level management visibility into schedule problems. The status of the development effort becomes more visible and measurable.

(3) It creates an open and auditable software development environment and removes some of the mystery often associated with this activity. The UDFs are normally kept "on the shelf" and open to inspection at any time.

(4) It assures that the documentation is accomplished and maintained concurrent with development activities. The problem of emerging from the development tunnel with little or inadequate documentation is considerably reduced.

(5) It reduces the problems associated with programmer turnover. The discipline and organization inherent in the approach simplifies the substitution of personnel at any point in the process without a significant loss of effort.

(6) It supports the principles of modularity. The guidelines given for establishing the unit boundaries assure that at least a minimum level of modularity will result.

(7) It can accommodate a variety of development plans and approaches. All UDF sections may be assigned to one performer, or different sections can be assigned to different specialists. The various sections contained in the UDF may also be expanded, contracted or even resequenced to better suit specific situations.

As a final comment, it must be emphasized that no device or approach can be effective without a strong management commitment to see it through. Every level of management needs to be supportive and aware of its responsibilities. Once the method is established it also needs to be audited for proper implementation and problem resolution. An independent software quality assurance activity can be a valuable asset in helping to define, audit and enforce management requirements.

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

ARTICULOS CORRESPONDIENTES AL TEMA CONTROL DE CALIDAD

M. EN C. MARCIAL PORTILLA ROBERTSON

MAYO, 1985

Perspectives on Software Engineering

MARVIN V. ZELKOWITZ

Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C. 20234, and Department of Computer Science, University of Maryland, College Park, Maryland 20742

Software engineering refers to the process of creating software systems. It applies loosely to techniques which reduce high software cost and complexity while increasing reliability and modifiability. This paper outlines the procedures used in the development of computer software, emphasizing large-scale software development, and pinpointing areas where problems exist and solutions have been proposed. Solutions from both the management and the programmer points of view are then given for many of these problem areas.

Keywords and Phrases: certification, chief programmer team, program correctness, program design language (PDL), software reliability, software development life cycle, software engineering, structured programming, top-down design, top-down development, validation, verification.

CR Categories: 1.3, 4.0, 4.6

INTRODUCTION

Software development usually proceeds in one of two ways: either the programmer works alone in designing, implementing, and testing a software system, or he is a member of a group of from three up to several hundred, working together on a large software system. Although software engineering embraces both approaches, here we are interested mainly in large-scale program development.

When the Verrazano Narrows Bridge in New York City was started in 1959, officials estimated that it would cost $325 million and be completed by 1965. It is the largest suspension bridge ever built, yet it was completed in November 1964, on target and within budget [ENR61, ENR64]. No similar pattern has been observed when we build software systems larger than those which had been built previously.

Software is often delivered late. It is frequently unreliable and usually expensive to maintain. The IBM OS project, which involved over 5,000 man-years of effort, was years late [BR0075]. Why is bridge engineering so exact while software engineering flounders so?

Part of the answer lies in the greater ease with which a civil engineer can see the added complexity of a larger bridge than a software engineer the complexity of a larger program. Part of today's "software problem" stems from our attempt to extrapolate from personal experiences with smaller programs to large systems programming projects.

We begin here by outlining the general approach used in developing program products, emphasizing aspects which are still poorly understood. Later, we enumerate the techniques which have been used to solve these problems. We do not attempt to cover all of the relevant topics in depth, but we give many references for further reading.

Software engineers are currently study-

CONTENTS

ing the causes of these problems and the mechanisms of software development. They seek both constraints on programming which will render software less expensive and more reliable and also the theoretical foundations upon which programs are built. Software engineering is not the same as programming, although programming is an important component. It is not the study of compilers and operating systems, although compiler writers and operating system implementors use similar techniques. It is not electrical engineering, although electronics does provide the basis for implementing the computer [JEFF77].

Software engineering is interdisciplinary. It uses mathematics to analyze and certify algorithms, engineering to estimate costs and define tradeoffs, and management science to define requirements, assess risks, oversee personnel, and monitor progress.

1. STAGES OF SOFTWARE DEVELOPMENT

The complexity of a large software system surpasses the comprehension of any one individual. To better control the development of a project, software managers have identified six separate stages through which software projects pass; these stages are collectively called the *software development life cycle*:

- Requirements analysis;
- Specification;
- Design;
- Coding;
- Testing;
- Operation and maintenance.

Figure 1, a pie chart, shows the approximate amount of time each stage takes. The stages are discussed in the following subsections.

Requirements Analysis

This first stage, curiously absent from many projects, defines the requirements for an acceptable solution to the problem. The statement "Write a COBOL program of not more than 50,000 words to produce payroll checks" is not a requirement; it is the partial specification of a computer solution to the problem. The computer is merely a tool for solving the problem. The requirements analysis focuses on the interface between

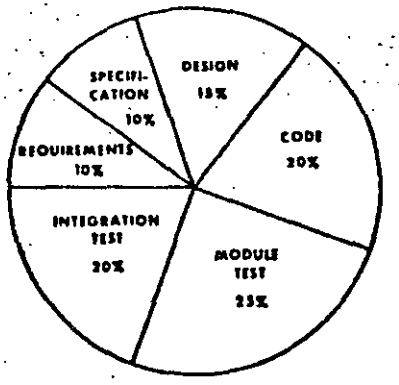


FIGURE 1. Effort required on various development activities (excluding maintenance).

the tool and the people who need to use it. For example, a company may consider several methods of paying its employees: 1) pay employees in cash; 2) use a computer to print payroll checks; 3) produce payroll checks manually; or 4) deposit payroll directly into employees' bank accounts.

Other aspects, such as processing time, costs, error probability, and chance of fraud or theft, must be considered among the basic requirements before an appropriate solution may be chosen. A requirements analysis can aid in understanding both the problem and the tradeoffs among conflicting constraints, thereby contributing to the best solution.

Hard requirements and the optional features must be distinguished. Are there time or space limitations? What facilities of the system are likely to change in the future? What facilities will be needed to maintain different versions of the system at different locations?

The resources needed to implement the system must be determined. How much money is available for the project? How much is actually needed? How many computers or computer services are affordable? What personnel are available? Can existing software be used? After the first questions are answered, project schedules must be planned. How will progress be controlled and monitored? What has been learned from previous efforts? What checkpoints will be inserted to measure this progress? Once all these questions have been answered, specification of a computer solution to the problem may begin.

Specification

While requirement analysis seeks to determine whether to use a computer, *specification* (also called *definition* [FIFE77]) seeks to define precisely what the computer is to do. What are the inputs and outputs? In the payroll example: Are employee records in a disk file? On tape? What is the format for each record in the file? What is the format for the output? Are checks to be printed? Is another tape to be written containing information for printing the checks offline? Will printed reports accompany the checks? What algorithms will be needed for computing deductions such as tax, unemployment and health insurance, or pension payments?

Since commercial systems process considerable amounts of data, the database is a central concern. What files are needed? How will they be formatted, accessed, updated, and deleted?

When the new system supersedes an older process (for example, when an automatic payroll system replaces a manual system), the conversion of the existing database to the new format must be part of the design. Conversion may require a special program which is discarded after its first and only use. Since the company may be using the older system in its day-to-day operation, bringing the new system online presents a problem. Can the old and the new systems run side by side for awhile?

The answers to these questions are set forth in the *functional specification*, a document describing the proposed computer solution. This document is important throughout the project. By defining the project, the specification gives both the purchaser and the developer a concrete description. The more precise the specifications are, the less likely will be errors, confusion, or recriminations later. The specifications enable test data to be developed early; this means that the performance of the system can be tested objectively, since the test data will not be influenced by implementation. Because it describes the scope of the solution, this document can be used for initial estimates of time, personnel, and other resources needed for the project.

These specifications define only what the system is to do, but not how to do it. Detailed algorithms for implementation are premature and may unduly constrain the designers.

Design

In the design stage, the algorithms called for in the specifications are developed, and the overall structure of the computer system takes shape. The system must be divided into small parts, each of which is the responsibility of an individual or a small

team. Each such module thus defined must have its constraints: its function, size, and speed.

As submodules are specified, they are represented in a tree diagram showing the nesting of the system's components. Figure 2 illustrates this for a typical compiler. This illustration, sometimes called a *baseline diagram*, is not by itself an adequate specification of the system.

Because the solution may not be known when the design stage starts, decomposition into small modules may be quite difficult. For older applications (such as compiler writing) this process may become standardized, but for new ones (such as defense systems or spacecraft control) it may be quite difficult.

A common problem is that the buyer of a system often does not know exactly what he wants, especially in state-of-the-art areas such as defense systems. As he sees the project evolve, the buyer often changes the specifications. If this occurs too often, the project may flounder. We discuss this problem later.

Coding

Coding is usually the easiest stage. High-level languages and structured programming simplify the task. In one study, Boehm [BOEH75] found that 64% of all errors occurred in design, but only 36% in coding. Hamilton and Zeldin [HAMI76] report that in the NASA Apollo project about 73% of all errors were design errors. We have mastered coding better than any other stage of software development.

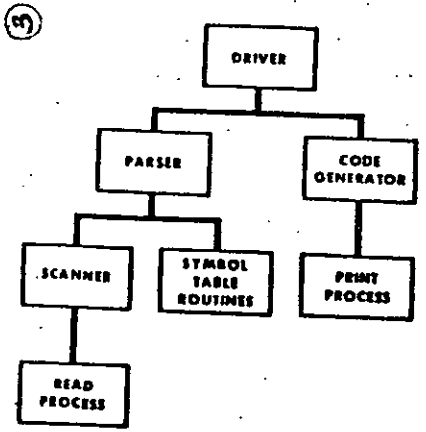


FIGURE 2. Sample baseline diagram for a compiler.

Testing

The testing stage may require up to half of the total effort. Inadequately planned testing often results in woefully late deliveries.

During testing the system is presented with data representative of that for the finished system; thus test data cannot be chosen at random. The test plan should, in fact, be designed early and most of the test data should be specified during the design stage of the project.

Testing is divided into three distinct operations:

1) *Module testing* subjects each module to the test data supplied by the programmer. A test driver simulates the software environment of the module by containing dummy routines to take the place of the actual subroutines that the tested module calls. Module testing is sometimes called *unit testing*. A module that passes these tests is released for integration testing.
2) *Integration testing* tests groups of components together. Eventually, this procedure produces a completely tested system. Integration testing frequently reveals errors missed in module tests. Correcting them may account for about a quarter of the total effort.
3) *Systems testing* involves the test of the completed system by an outside group. The independence of this group is important.

The buyer may also insist on his own systems test, or *acceptance test*, before formally accepting the product. Comparison of the performance of several systems (such as those of a given software product already available from several sources) is called *benchmark testing*.

During testing, many criteria are used to determine correct program execution. Among other important criteria, the program is considered correct if:

1) every statement has been executed at least once by the test data;
2) every path through the program has been executed at least once by the test data; and
3) for each specification of the program, test data demonstrate that the program performs the particular specification correctly.

These three different criteria show that there is no single acceptable criterion defining a "well-tested" program. Goodenough and Gerhart [GOOD76] proposed a set of consistent definitions for "testing" and showed that some of these definitions of testing are, in theory, insufficient. We return to this subject later. For a survey of good testing techniques, see [HUAN75].

Closely related to testing are verification and validation (V/V). A system is *validated* when testing shows that the system performs according to its specifications. A system is *verified* when it has been proved to meet its specifications. Current technology is inadequate for achieving both these objectives. A validated system may misbehave for cases not included in the test data. A verified system is correct relative only to the initial specifications and assumptions about the operating environment; formal proofs tend to be lengthy, making them subject to error or incredulity. *Certification* sometimes refers to the overall process of creating a correct program by validation and verification.

In certifying a program, three terms must be distinguished. A *failure* in a system is an event which marks a violation of the system's specifications. An *error* is an item of information which, when processed by the normal algorithms of the system, produces a failure. Since error recovery may be built into the program (for example, ON units in PL/I), not every error will produce a failure. A *fault* is a mechanical or algorithmic defect which generates an error (for example, a programming "bug") [DENN76a].

Reliability is a concept which must not be confused with correctness. A *correct* program is one that has been proved to meet its specifications. In contrast, a *reliable* program need not be correct, but gives acceptable answers even if the data or environment do not meet the assumptions made about them. We would like a system to be highly robust, that is, to accept a large class of input data and to process it correctly under adverse conditions. Parnas [PARN75] describes a correct system as one that is free from faults and has no errors in its internal data. A program is reliable if failures do not seriously impair its satisfactory operation.

Operating systems with "fail-soft" procedures illustrate the difference between reliability and correctness. A detected error causes the system to shut down without losing information, possibly restarting after error recovery. Such a system may not be correct because it is subject to errors, but it is reliable because of its consistent operation. A real-time program may be correct as long as a sensor reports correctly, but it may be unreliable if bad sensor readings have not been considered.

Operation and Maintenance

Figure 1 shows the disposition of software costs in developing a new project. But this can be the wrong chart! The activities noted in Figure 1 are only 25% to 33% of the effort required during the life of the system. Figure 3 illustrates that maintenance costs ultimately dwarf development costs.

No computer system is immutable. Since a buyer seldom knows what he wants, he seldom is satisfied. Probably, he will request changes in the delivered system. Errors missed in testing will later be discovered. Different installations will need special modifications for local conditions. The management of multiple copies of a system is another difficult problem that must be handled early in development. Once the first line of code is written, the structure of the resulting maintenance operation may already be fixed, so it is best to plan for it then.

The division of effort indicated in Figure 3 greatly affects system development. Because of hidden maintenance costs, techniques that rush development and provide

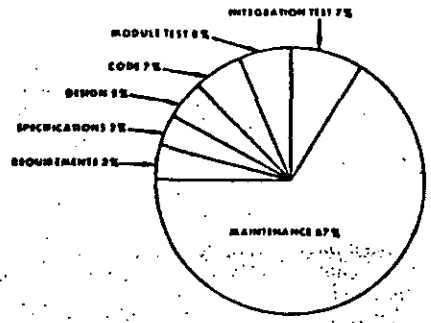


FIGURE 3. True effort on many large-scale software systems.

for very early initial implementation may be trading early execution for a much more extensive maintenance operation.

The maintenance problem is sometimes referred to as the "parts number explosion." For example, a certain system contains components A, B, and C. Installation I finds and reports an error. The developer fixes the error and sends a corrected module A' to all installations using the system.

Installations II and III ignore the replacement and continue with the original system. Installations I and II discover another error in module A. The developer must now determine whether both of these errors are the same, since different versions of module A are involved. The correction of this error involves correction of both A' (for I) and A (for II) yielding A'' and A'''. There are now three versions of the system.

To avoid this growth, systems often receive updates, called releases, at fixed intervals. A useful tool for dealing with myriad maintenance problems is a "systems database" started during the specifications stage. This database records the characteristics of the different installations. It includes the procedures for reporting, testing, and repairing errors before distributing the corrections.

Themes of Software Engineering

It should be clear that each software development stage may influence earlier stages. The writing of specifications gives feedback for evaluating resource requirements; the design often reveals flaws in these specifications; coding, testing, and operation reveal problems in design. The goals of software engineering are thus to:

- Use techniques that manage system complexity.
- Increase system reliability and correctness.
- Develop techniques to predict software costs more accurately.

In the following sections, we discuss approaches to some of these problems. The list of techniques is divided into management and programmer issues. Management issues concern the effective organization of personnel on a project. Programmer issues concern the techniques used by individual programmers to improve their performance.

2. MANAGEMENT ISSUES

A manager controls two major resources: personnel and computer equipment. This section surveys techniques for optimizing the use of these resources.

Size and Cost Control

A project may fail when management is not aware of developing problems; a year's delay comes "one day at a time" [BROO75]. Faced with catastrophic failure (for example, needed hardware is delayed six months), a resourceful manager can usually find alternatives. However, it is easy to ignore day-to-day problems (such as sick employees or many errors during testing).

Most problems occur at the interfaces of modules written by different programmers. Since the number of such interfaces is on the order of the square of the number of individuals involved, the problem becomes unwieldy when the number of persons in a development group grows to four or more.

As an example of the communications problem, assume that a single programmer is capable of writing a 5,000-line program in a year, and that a programming system requires about 50,000 lines of code and is to be completed in two years. Five programmers would seem to be sufficient (see Figure 4a).

However, the five programmers must communicate with one another. Such communication takes time and also causes some loss in productivity since finding misunderstood aspects will require additional testing. For this simple analysis, assume that each communication path "costs" a programmer 250 lines of code per year. Each of the five programmers, therefore, can produce only 4,000 lines per year and only 40,000 lines are completed within two years (see Figure 4b).

This means that eight programmers producing 3,250 lines per year are actually needed in order to produce the required 50,000. A manager is required for direction of this large effort. Therefore, in summary, eight programmers and a manager, each producing an average of 3,000 lines per year, are actually needed (see Figure 4c).

As we shall see, simply counting lines of code is not a good way to estimate productivity. The figures in this example are only given to illustrate a point, but they are representative of the problem. There are also techniques designed to limit this communications "explosion" and to increase programmer productivity.

Project Personnel

Software can usually be divided into three categories: 1) control programs (such as operating systems), 2) systems programs (such as compilers), and 3) applications programs (such as file management systems). A single programmer working on a control program can produce about 600 lines of code per year, whereas he can produce about 2,000 lines if working on a systems program and about 6,000 if working on an applications program [WOLV74]. The type of task certainly affects the productivity that can be expected from a given programmer. However, as the previous example demonstrates, the organization of personnel also affects performance. For example, with the approach of deadlines, docu-

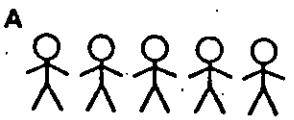


FIGURE 4(a). Single projects: 5,000 lines per year = 50,000 lines in two years (no communication between programmers).

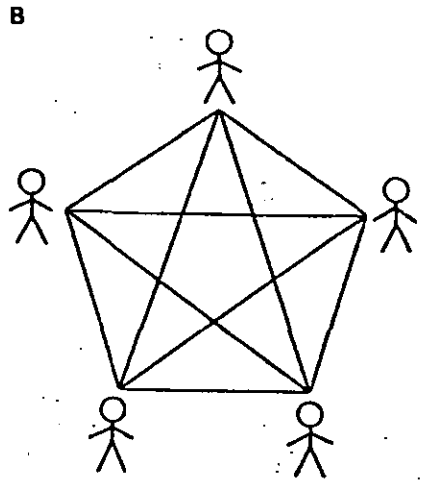


FIGURE 4(b). Five-member group: 4,000 lines per year = 40,000 lines in two years (ten communication pairs).

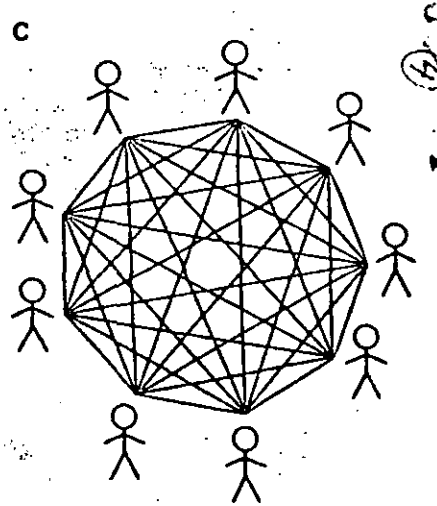


FIGURE 4(c). Nine-member team: 3,000 lines per year = 50,000 lines in two years (36 communication pairs).

1977 the cost had risen past $9 billion [ENR77]. In this case, the design was altered continuously as the federal government imposed new environmental standards (that is, changing specifications), and new technologies were needed to move large quantities of oil in a cold weather environment. Previous experience was only marginally helpful.

Results from computer hardware reliability theory are now starting to play a role in software estimation [PUTN77]. The cumulative expenditures over time for large-scale projects have been found to agree closely with the following equations:

$$E = K(1-e^{-at^2})$$

where E is the total amount spent on the project up to time t, K is the total cost of the project, and a is a measure of the maximum expenditures for any one time period. This relationship is usually expressed in its differential form, called a Rayleigh curve:

$$E' = 2Kate^{-at^2}$$

where E' is the rate of expenditures, or the amount spent on the project during year number t. Since 70% of the cost of a project occurs during the maintenance stage, it is not surprising that the maximum expenditures will occur just before the product is released, a time when it is usually assumed that the effort is winding down before termination (see Figure 6).

The Rayleigh curve has two parameters, K and a; however, a system can be described by three general characteristics: 1) total cost, 2) rate of expenditure, and 3) completion date. Two of these characteristics are enough to determine the constants K and a. When a project is initiated, the proposed budget is an estimate of K, and the available personnel permits a to be calculated. Assuming that requirement analysis determines that these figures represent an accurate assessment of the complexity of the problem, the estimated completion date (the date when the expenditures reach a maximum) can be computed, and thus cannot be set arbitrarily during the requirements or specification stage. This method provides the basis for a cost estimation strategy that has been applied to smaller projects in the 100 man-month range [BASI78]. We may be close to a mathematical theory of cost estimation which will greatly reduce our need to "guess" at project costs.

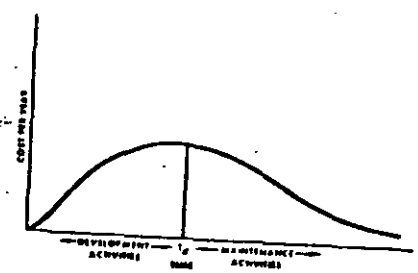


FIGURE 6. Yearly rate of expenditures approximates the Rayleigh curve. Total cost (area under curve) $= K$; $a = 1/T_d^2$; rate $= 2Kate^{-at^2}$.

Milestones

A *milestone* is the specification of a demonstrable event in the development of a project. Milestones are scheduled by management to measure progress. "Coding is 90% complete" is not a milestone because the manager cannot know when 90% of the code is complete until the project itself is complete.

There are many candidates for milestones: publication of the functional specifications, writing of individual module designs, module compiling without errors, units that have been tested successfully, and so on. Milestones are scheduled fairly often to detect early slippage. PERT charts may be used to estimate the effects of slippage in one stage on later stages.

Reporting forms can give information useful for estimating when a future milestone will be reached. A general project summary, describing such overall characteristics as system size, cost, completion dates, or complexity, can be resubmitted with each milestone. Change reports can be submitted each time a module is altered. The use of a librarian probably means that such a form already exists. Weekly personnel and computer reports monitor expenditures. Although they add a minor overhead to the project, the information helps management keep abreast of progress [BASI78, WALS77].

Development Tools

Compilers and certain debugging facilities have been available for some time. In contrast, other programming aids are new and experience with them is less estensive. Cross referencing, attribute listings, and symbolic storage maps are examples of such aids. Auditors or database systems can help to control the organization of the developing system. The Problem Statement Language/Problem Statement Analyzer (PSL/PSA) of the ISDOS project of the University of Michigan is one of the first database systems for providing a module library for storing source code, and includes a language for specifying interfaces in system design which can be checked automatically [TEIC77]. RSL/SSL is a similar system designed to specify requirements and to design interfaces via a data management system [DAVI77].

An alternative approach is the Programmer's Workbench developed by Bell Telephone Laboratories [DOLO76]. A PDP 11 based system provides a set of support routines for module development, library maintenance, documentation, and testing. Proper use of these facilities allows accessing information in an easier, controlled environment.

Reliability

Conceptual Integrity

Conceptual integrity, uniformity of style and simplicity of structure, are usually achieved by minimizing the number of individuals in the project. A chief programmer team greatly enhances conceptual integrity.

A small group minimizes contradictory aspects of a design. In the PL/I language, for example, the PICTURE attribute declaration may be abbreviated as either PIC or P, but in format specifications it may only be P [ANSI76]. In FORTRAN, the right side of an assignment statement can be an arbitrary arithmetic expression, but DO loop indices must be integer constants or variables, and subscripts to arrays are limited to seven basic forms [ANSI66]. These are difficult idiosyncracies to remember. They illustrate a lack of conceptual integrity that can arise when many people with different objectives become involved in a project. A consistent design is less prone to errors because the user can follow a simple set of rules.

Continual System Validation

A *walkthrough* is a management review to discover errors in a system. In one study, TRW discovered that the cost of fixing an error at the coding stage is about twice that of fixing it at the design stage, and catching it in testing costs about ten times as much as it does in design [BOEH76].

A walkthrough is scheduled periodically for all personnel. In attendance are the project manager (chief programmer), the person reviewed, and several others knowledgeable about the project. One section of the system is selected for review and each individual is given information about that section (for example, design document for a design walkthrough, code for a coding walkthrough) before the review. The person being reviewed then describes the module under study.

The walkthrough is intended to detect errors, not to correct them. Also, the walkthrough is brief—not more than two hours. By explaining the design to others, the person reviewed is likely to discover vague specifications or missing conditions.

An important point for management is that the walkthrough is *not* for personnel evaluation. If the person reviewed perceives that he is being evaluated, he may attempt to cover up problems or present a rosy picture.

An informal yet very effective version of the walkthrough is *code reading*. A second programmer reviews the code for each module. This technique frequently turns up errors when the second reader, failing to understand some aspects of the code, asks the author for an explanation.

3. PROGRAMMER ISSUES

Each stage of the software development life cycle has its own set of problems and solutions. The most advanced techniques apply to the last stages; the first stages are the least developed. For example, testing and

mentation is often given lower priority. However, since 70% of the total system cost may occur during the maintenance state (where the documentation is heavily used), this may be a false economy of effort.

Use of a librarian is one way to avoid this problem. A librarian provides the interface between the programmer and the computer. Programs are coded and given to the librarian for insertion into the online project library. The actual debugging of the module is carried out by the programmer, but changes to the official module in the library are made by the librarian. The use of a library is further enhanced when an online data management system is used.

The use of a librarian has another beneficial effect. All changes in modules in the project library are handled by one individual and are easy to monitor; they are often reviewed by the project manager before insertion. This prevents "midnight patches" from being quickly incorporated into a system and forces the programmer to think carefully about each change. It also gives the manager disciplined product control and helps with audit trails.

On larger projects, a technical writer may perform much of the documentation, thus freeing programmers for the tasks for which they are most skilled.

The culmination of this trend is the *chief programmer team* concept developed by IBM [BAKE72]. The concept recognizes that programmers have different levels of competence; therefore, the most competent should do the major work, while others function in supporting roles. As the earlier example shows, interfacing problems greatly reduce programmer productivity. The chief programmer team is one way of limiting this complexity.

The chief programmer, an excellent programmer and a creative and well-disciplined individual, is the head of the team. He may be five or more times more productive than the lowest member of the team [BOEH77]. He functions as the technical manager of the project, designs the system, and writes the top-level interfaces for all major modules.

If a project is large, a team may also have an administrative manager to handle such responsibilities as budgeting time, vacations, office space, and other resources, and reporting to upper-level management. The administrative manager often administers several programming teams.

The backup programmer works with the chief programmer and fills in details assigned by the chief programmer. Should the chief programmer leave the project, the backup programmer would take over. This means that he also must be an excellent programmer. The backup programmer also fulfills an important role by providing the chief programmer with a peer with whom he can discuss the design.

There are also two or three junior programmers assigned to the team to write the low-level modules defined by the chief programmer. The term "junior" in this context means "less experienced," not "less capable." As Boehm states, the best results occur with fewer and better people.

Using the example illustrated by Figure 4, a chief programmer team of five individuals has only seven communications paths, and the chief programmer, being that rare individual, can produce more than his quota of 5,000 lines (see Figure 5). Thus productivity per programmer could be greater than 5,000 lines per year, instead of the previous figure of only 4,000.

The team has a librarian to manage the project library—both the online module library and the offline project documentation (also called the project notebook). The project notebook contains, among other things, records of compilations and test runs of all modules. It is important to the team structure, since all development is now accountable and open for inspection, and code is no longer the "private property" of any individual programmer.

Programmers have traditionally been reluctant to exhibit their products until completion, since discovered errors have traditionally been viewed as a personal failure. The absurdity of this approach is clear enough. If the ego element is removed from programming, programmers may openly ask others for advice when they need it, instead of trying to solve all problems themselves [WEIN71].

The team may include other supporting

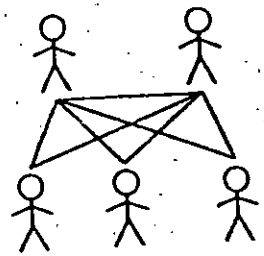


FIGURE 5. Fewer communications paths in a chief programmer team.

personnel such as secretaries and technical writers. Experience shows that ten is the upper bound to team size.

This structure, however, will not solve all problems in development. With a smaller number of individuals involved, competence is crucial. It is not possible to "work around" a nonproductive individual as one might do in a large project. There are also extremely large projects where a group of ten is simply too small to tackle development. Larger teams are not efficient.

A man-month, or the amount of work performed by one individual in one month, is a deceptive measure for estimating project productivity. A project requiring four programmers for a year cannot be completed by 48 programmers in one month. The example of the 50,000 line system needed in two years shows some of the problems inherent in trying to exchange programmers for time. "Adding manpower to a late software project makes it later" [BROO75]. New personnel divert existing personnel needed to train them; they require more supervision; they complicate communication and interfere with the design since they are unfamiliar with the project structure.

However, man-months do serve a purpose as a useful measure of project costs. By adding more data, such as the rate of using man-months, accurate cost estimation techniques can be utilized. These are explained in the following subsection.

Estimation Techniques

One of the most important aspects of engineering is estimating the resources needed to complete a project. As previously mentioned, the Verrazano Narrows Bridge in New York City was completed at the projected time and within the estimated budget. How was such accuracy achieved?

Most engineering disciplines have highly developed methods of estimating resource needs. One such technique is the following [GALL65]:

1) Develop an outline of the requirements from the Request for Quotation (RFQ);
2) Gather similar information, for example, data from similar projects;
3) Select the basic relevant data;
4) Develop estimates;
5) Make the final evaluation.

Although this approach has been advocated for software development, software projects have difficulty passing Step 1 [WOLV74]. Engineers have been building bridges for 6,000 years but software systems for only 30 years. Prior experience to develop the true requirements may not be available. Moreover, with very little background to build on, the developer has little knowledge of similar systems to use in evaluation (Step 2).

In developing the estimates (Step 4), the following tasks must be undertaken:

4a) Compare the project to similar previous projects.
4b) Divide the project into units and compare each unit with similar units;
4c) Schedule work and estimate resources by the month.
4d) Develop standards that can be applied to work.

Note that for Step 4a), the lack of previous experience presents a continuing problem. Also, for Step 4d), an adequate set of standards does not yet exist.

Experience is the key to accurate estimation. Even civil engineering projects may fail badly when established techniques are not followed. Although the Verrazano Narrows Bridge was the world's largest suspension bridge, its engineers had much experience with other similar structures. On the other hand, the Alaskan oil pipeline was estimated to cost $900 million, yet by mid-

debugging problems are apparent to every programmer; these tools are the oldest and most advanced. Techniques for improving coding were developed next. The most recent developments have related to requirements and specifications. Although many technical problems have not been solved, an effective methodology is emerging. Some of these techniques are presented in the following subsections.

Verification and Validation

Verification and validation (module and integration testing) of a system occupy about half of the development time of a project. Many debugging aids have been developed to facilitate this effort; most are implemented as programs to test some feature of a system.

Automated Tools

The earliest and most primitive debugging tools were the dump and the trace. A *dump* is a listing of the contents of the machine's memory. This listing can often reveal unintelligible data or errors. Unfortunately, a dump may not be taken until long "after the fact" and the cause of the error may not then be apparent. A *trace* is a printout showing the values of selected variables after each statement is executed. It may help a programmer to discover errors.

These techniques are not usually very effective because they supply much data with little or no interpretation. More advanced methods are needed to reduce this data to an intelligible form.

Flowgraph analyzers are capable of detecting references to variables which are never initialized or never reused after receiving a value; these usually indicate errors. Test data generators are also available. Assertion checkers validate that given conditions are true at indicated points of a program. Automatic verification systems have been implemented for small languages [KING69] and symbolic execution has been proposed as a practical means for validating programs in a more complex language. The PSL/PSA system is an example of a tool for assisting in design and specification. Symbolic dumps and traces are generated with compilers like PL/C [CONW73] or PLUM [ZELK75]. Ramamoorthy and Ho [RAMA75] survey many of these tools.

Certification

Programs can be verified at several levels. Conway [CONW78] lists eight different verification conditions:

- A program contains no syntactic errors.
- A program contains no compilation errors or faults during program execution.
- There exist test data for which the program gives correct answers.
- For typical sets of test data, the program gives correct answers.
- For difficult sets of test data, the program gives correct answers.
- For all possible sets of data which are valid with respect to the problem specification, the program gives correct answers.
- For all possible sets of valid test data and all likely conditions of erroneous input, the program gives correct answers.
- For all possible input, the program gives correct answers.

Some people are optimistic that one day complete automatic program verification will be possible. Today's tools operate a posteriori, demonstrating that a given program works. Tomorrow's tools will also operate a priori, helping to develop programs which are correct before they are ever run. Such tools can reduce the amount of testing required for a completed project [DIJK76].

Verification techniques have the following general structures. A program is represented by a flowchart. Associated with each arc in the flowchart is a predicate, called an *assertion*. If A_i is the assertion associated with an arc entering statement S_i and A_j is the assertion on the arc following the statement, then the statement "If A_i is true, and if statement S is executed, then assertion A_j will be true" must be proved (see Figure 7).

This process can be repeated for each statement in a program. If A_1 is the assertion immediately preceding the input node to the flowchart (that is, the initial assertion), and if A_n is the assertion at the exit node (for example, the final assertion), then the statement "If A_1 is true, and the program is executed, then A_n is true" will be the theorem that states that the program meets its specifications (A_1 and A_n) (see Figure 8). This approach was formalized by Hoare [HOAR69] who defined a set of axioms for determining the effects upon the assertions (preconditions and postconditions) by each statement type in a language. Thus verifying program correctness reduces to proving a theorem of the predicate calculus.

FIGURE 7. Assertions A_i and A_j surround each statement of a program.

Certification technique development is still in a preliminary stage and does not meet the challenge of a modern large system. In addition, axiomatic certification is weak in the sense that the output assertion is proved true only if the program terminates. Axiomatic methods are incapable of proving termination. However, termination can often be proved informally by the programmer.

A typical approach to proving that program loops terminate is the following:

1) Find some number P that is always nonnegative within the loop.
2) Show that for each execution of the loop, P is decremented by at least a fixed amount.

If both conditions are always true, the loop must terminate before P becomes negative. A programmer who uses such rules, even informally, will seldom write nonterminating loops.

Consider this program fragment:

```
while x<y do
  ...
  x := x+1
  ...
end
```

Let quantity P be the expression $y-x$, and let $P(i)$ refer to the value of P during the ith execution of the loop. Because $x<y$ must be true for each next iteration, $y-x$ is always nonnegative and condition 1) is satisfied for each execution of the loop. Since the loop contains the statement $x := x+1$, $P(i+1) = P(i)-1$, satisfying condition 2). Therefore the loop must terminate.

Certification will not solve all our software problems, although it is an important tool. Gerhart and Yelowitz [GERH76] have shown that there are many published "certified" programs that contain errors. Even experts err.

Formal Testing

Goodenough and Gerhart [GOOD75] have clarified the concepts of testing. A *domain* is the set of permissible inputs to a program, and a *test* is a subset of the domain. A *testing criterion* specifies what is to be tested (for example, specifications, all statements, all paths).

A test is *complete* if the test meets all the requirements of the testing criterion, and a complete test is *successful* if the program gives correct results for each input in the test.

With these definitions, we can define program reliability and validity. A program is *reliable* if every found error is revealed by every complete test. A program is *valid* if every error is revealed by some complete test.

With these definitions, several important results can be proved. Among these are:

- If a program is both reliable and valid, then it is correct if and only if any complete test is also successful.
- The criterion "execute every path" is not valid; there exist programs all of whose test sets succeed, but which produce the wrong results for some input.

While this framework is somewhat technical and is not applicable to all programming, it is an important step in formalizing this area. We now have a basis for talking

FIGURE 8. Predicates A_1 and A_n specify input-output behavior of a program.

grammer's task is made easier when the computer does more work.

Structured Programming

A major development in facilitating the programming task is known as *structured programming*, which has been erroneously called "gotoless" programming. Fortunately, the debate about "to goto or not to goto" has mostly disappeared, and some clear ideas have emerged. The premise of structured programming is to use a small set of simple control and data structures with simple proof rules. A program then is built by nesting these statements inside each other. This method restricts the number of connections between program parts and thereby improves the comprehensibility and reliability of the program.

The if-then-else, while-do, and sequence statements are a commonly suggested set of control structures for this type of programming; however, there is nothing sacred about them. Knuth [KNUT74] has argued that the goto statement is irrelevant to the true goals of structured programming.

These simple control structures help programmers certify programs, even at an informal level. For example, a program can be represented as a function from its input data to its output data. Suppose $f(x)$ represents a segment of a program given by the following if-then-else statement:

$$\text{if } p(x) \text{ then } g(x) \text{ else } h(x).$$

Because functions g and h are simpler than function f, their specifications should be simpler. If their specifications are known, the overall function f is defined by

$$f(x) = (p(x) \rightarrow g(x)) \wedge (\neg p(x) \rightarrow h(x)).$$

The programmer can express the formal definition of f in terms of the simpler definitions of g and h.

Languages such as ALGOL, PASCAL, and certain subsets of PL/1 contribute to good programming practices by providing these facilities. In order to repair FORTRAN's lack of structure, over 50 preprocessors for translating well-structured pseudo-FORTRAN programs into true FORTRAN have been developed [REIF76]. An if-then-else has been added to the new FORTRAN-77 standard, although a general while is still missing from the language.

System Design

A technique related to structured programming is *top-down design*, in which a programmer first formulates a subroutine as a single statement, which is then expanded into one or two of the basic control structures mentioned earlier. At each level the function is expanded in increasingly greater detail until the resulting description becomes the actual source language program in some programming language.

Using this approach, also called *stepwise refinement* [WIRT71, WIRT74], the program is hierarchically structured and is described by successive refinements. Each refinement is interpreted by referring to other refinements of which it is a component. Concerning this method, Wirth states:

> I should like to stress that we should not be led to infer that actual program conception proceeds in such a well organized, straightforward, "topdown" manner. Later refinement steps may often show that earlier decisions are inappropriate and must be reconsidered. But this neat, *nested factorization* of a program serves admirably well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal, and even formal proofs of correctness. The emerging modularity is particularly welcome if programs have to be adjusted to changed or extended specifications. [WIRT74, p. 251]

Operating systems are often modeled as hierarchies of *abstract* or *virtual* machines [BRIN77]. At the lowest level of the system is the physical hardware. Each new level provides additional *capabilities*, or allowable functions on data, and hides some of the details of a lower level. For example, if one level accesses the paging hardware of the computer and provides a large virtual memory for all other processes, other abstract machines at higher levels can be implemented as if they had unlimited memory since this detail is controlled by a lower level.

The concept of a *program design language* (PDL) to aid in this development

about such concepts as reliability and correctness.

Mean Time Between Failure

While useful for focusing our attention, analogies with other engineering fields must be used with care. Reliability is one area of incomplete analogies. The concept of *mean time between failure (MTBF)* does not apply directly to software although it sometimes is used as if it does.

Systems built from physical components wear out; transistors fail; motors burn out; soldered joints break. This is also true for the hardware of the computer. However, the logical components of software are durable. A given program will always produce the same answer for the same input, as long as the hardware does not fail. When a software module "fails," it has been presented with an input that finally revealed an error present from the start.

The MTBF measures the time between revelations of errors. This, in turn, depends on the kinds of inputs presented. A compiler used only for short jobs from students may have a long MTBF; but if it is suddenly used for other applications, its MTBF may decrease sharply as unsuspected errors are exercised. A large MTBF can thus be interpreted only as an indication of possible reliability, not as a proof of it.

Error Days

Since formal certification of large classes of programs is still unattainable, techniques for estimating the validity of programs are still being considered. Most of these techniques measure the number of errors discovered, which are assumed to be representative of the total number of errors present in the system, and hence a measure of the reliability of the system.

Mills [MILL76] defines an *error day* as a measure stating that one error remains undetected in a system for one day. The total number of error days in a system is computed by summing, for each error, the length of time that error was in the system. A high error day count may reveal many errors (poor design) or long-lived errors (poor development).

The assumption is made that if a program is delivered with a low error day count, then there is a good chance that it will remain low during future use. However, two major problems remain before this measure can be widely used. First, it is difficult to discover when a particular error first entered a system. Second, it may be difficult to obtain such information from the developer of a delivered product.

Programming Techniques

Several authors have mentioned that the number of lines of code produced by a programmer in a given time tends to be independent of the language used. This implies that higher level languages enhance productivity [BROO75, HALS77]. This is true even though assembly language programs are potentially more efficient; their potential is seldom realized in practice.

The goals in developing early higher level languages were to be able to express clearly an algorithm and translate it into efficient machine language programs. The efficiency of the resulting code was all important. This led to some anomalies in FORTRAN arising from the structure of the IBM 704 for which it was developed (for example, the three-way branch of the arithmetic IF). ALGOL, which was developed as a machine-independent way of expressing algorithms, contained concepts whose implementation on conventional hardware was inefficient (e.g., recursion, call-by-name); this may explain why ALGOL is not widely used.

By the late 1960s it was accepted that the language should facilitate writing the program and that the machine should be designed to create an efficient run-time environment. Today there is a definite shift toward using the language to make programming and documentation easier and to produce reliable and correct software.

This does not mean, however, that efficiency is ignored today. Whereas PL/1 permits the writing of simple programs whose execution time is quite long, PASCAL was designed to exclude constructs whose machine code is inefficient. Since hardware is less expensive than programmers, reliability has become a major factor. The pro-

has been defined [CAIN75]. This type of language contains two structures: "outer" syntax of basic statement types, such as if-then-else, while, and sequence for connecting components, and an "inner" syntax that corresponds to the application being designed. The inner syntax is English statement oriented, and is expanded, step by step, until it expresses the algorithm in some programming language. Figure 9 represents an example of a PDL design.

It should be noted here that PSL/PSA and PDL complement each other. PSL/PSA is a specifications tool that validates correct data usage between two modules (interfaces). A system like PDL is useful for describing a given module at any level of detail. Both PSL/PSA and PDL can contribute to success in a large project.

Even though designed from the top down, many systems are implemented from the bottom up. Low-level routines are first coded with drivers to test them; then new modules, using these low-level routines, are added, and the system is built up.

Top-down development is another technique for implementing hierarchically structured programs. Here the top-level routines are written first and lower level routines, called *stubs*, are written to interface with these. The stubs return control after printing a simple message and may return some fixed sample test values. The stub is eventually replaced by the full module which now includes calls to other stubs. In this manner an entire system can be gradually developed.

If used carefully, this technique can be valuable; however, the system's correctness is assumed, not proved, until the last stub has been replaced [DENN76a]. The documentation specifies the assumptions on each stub. For example, if

$$f(x) = \text{if } p(x) \text{ then } g(x) \text{ else } h(x)$$

is a program fragment calling stubs g and h, then f will be correct only if the modules eventually replacing the stubs g and h are correct.

Via top-down development, a user sees the top-level interfaces in the system very early. He can then make changes relatively easily and soon. Another approach with the same goal is *iterative enhancement* [BASI 75]. Using this technique, a subset of the problem is first designed and implemented. This gives the user a running system early in the life cycle when changes are easier to make. This process is repeated to develop successively larger subsets until the final product is delivered.

Brooks [BROO75] believes that the first version of a system is always "thrown away," because the concrete specifications for a system are often not defined until the system is completed, a time when the initial product meets those specifications rather poorly. It is often cheaper and faster to rebuild a system from scratch than to try to modify an existing product to meet these specifications. However, a developer will often deliver such a modified system as a "pre-release" if a deadline is near and the purchaser is demanding results. The buyer then suffers with this version, replete with errors, until he throws it away or has the product rebuilt. Iterative enhancement can make rebuilding less chaotic since there is a running system (not meeting all the requirements) early in the development cycle.

```
max: PROCEDURE (list);
     /* Find maximum element in a list */
     DECLARE (maximum, next) integer;
     DECLARE list list of integers;
     maximum = first element of list;
     DO WHILE (more elements in list);
       next = next element of list;
       maximum = largest of next and maximum;
       END;
     RETURN (maximum);
     END max;
```

FIGURE 9. PDL of a program to find the largest element in a list (outer syntax is in upper case; inner syntax in lower case).

Performance Issues

The chosen algorithms and data structures have a much greater influence on program performance than code optimization or the programming language. Before choosing an algorithm, the programmer faces these questions:

- Can previously written software be used?
- If a new module must be written, what algorithms and data structures will give an efficient solution?

Programming languages usually include standard mathematical functions such as sine, logarithm, and square root. They give the programmer ready access to libraries of standard software packages. This allows the programmer to use results of previous work. In preparing programs for standard libraries, analysts have included many options in a single package. The effect can be a large cumbersome package which is inefficient because only a small part of it is applicable at any one time. This can be avoided by installing multiple versions of the module for each special case.

Many opportunities remain for more packaging and use of existing software. Difficulties in achieving this include:

- Identifying which standard algorithm to package. This is easier in mathematical areas such as statistical testing, integration, differentiation, and matrix computations than in many non-numerical areas such as business applications.
- Transporting and interfacing with packaged software. Some progress has been made with programs stored in read-only memories which plug into microprocessors, or with interface processors on computer networks. A major problem area lies in interfacing software directly to other software, since there are no conventions. Some help is afforded by such concepts as the "pipeline" in UNIX, which provides a general communications channel between programs [RITC74].

Algorithm Analysis

Sometimes the program specification is not changeable, and the analyst must find the best possible algorithm. Sometimes, however, the specifications can be altered to permit a more efficient solution. In some instances we can show that there are no algorithms guaranteed to be efficient in all cases; here approximate algorithms that are efficient in most cases but need not give exact solutions must be used.

The fast Fourier transform illustrates the most efficient form for computing the Fourier transform, a technique useful in waveform analysis [COOL65]. This transform is based on a finite set of points rather than on a complex integral which is harder to compute. Language analysis (parsing) in a compiler illustrates how changing the specification can permit a more efficient solution. Any string of N symbols in an arbitrary context-free language can be parsed in time of order $O(N^{**}3)$ [YOUN67]; however, a programming language need not include all features of an arbitrary context-free language. PASCAL is an example of a language which can be parsed by a deterministic top-down parser in average time of order $O(N)$ [AHO72]. If we are free to set language specifications, we can choose the language and be rewarded with efficient compilers.

Many practical problems, such as job scheduling or network commodity flow, involve enumeration of a combinatorially large number of alternatives and selection of a best solution. In these cases it may be better to restrict the search for a suboptimal but good answer. We recommend the paper by Weide [WEID77] for a discussion of the issues and a state-of-the-art survey of algorithm analysis.

Efficiency

In many cases the results of algorithmic analysis are not extensive enough to help the programmer; thus we need to offer techniques which can help locate and remove sources of inefficiency. One such tool is an optimizing compiler which, for some languages, can yield significant improvements [LOWR69]. The value of such tools, however, is limited [KNUT71] and may be realized only for programs which are used often enough to justify the investment in optimization.

One of the most powerful aids is the *frequency histogram*, which reveals how often each statement of a program is executed. It is not unusual to find that 10% of the statements account for 80% of the execution time [KNUT71]. A programmer who concentrates on these "bottlenecks" in his algorithms can realize significant performance improvements at a minimum investment. This technique has been used in some interactive operating systems, such as

UNIX and MULTICS, which started out as high-level language operating systems. Bottlenecks have been replaced by assembly language routines in less than 20% of the system.

Theory of Specifications

One area of software engineering that is now under study is system specifications. The objective is to state the specifications early using a metalanguage. This places restrictions on the design and may help establish whether the specifications are met.

An early example of such a specification was the so-called "gotoless programming" [DIJK68, KNUT74]. It is properly called "structured programming." It restricts the form of statements a programmer may use, but this restriction contributes to comprehensibility and enhances a correctness proof.

A second set of such rules employs the concepts of levels of abstraction, information hiding, and module interfacing to restrict access to the internal structure of data. Parnas [PARN72] formalized these ideas which were standard practices of expert programmers. He defines data as a collection of logical objects, each with a set of allowable states. Procedures can then be written to hide the representation of these objects inside separate modules. The user manipulates the objects by calling the special procedures.

Several languages that facilitate the use of these concepts have been developed. Among these are EUCLID [POPE77], CLU [LISK77], and ALPHARD [WULF76]. These languages permit programmers to define abstract data types having the property to encapsulate the representation of the logical objects [LISK75]. When concurrency is an issue, the use of abstract objects must be controlled by synchronization (for example, locks, signals); in this case the abstract type managers are called *monitors*.

Another kind of specification consists of "higher order software axioms" (HOS) [HAMI76], which are a set of six axioms that specify allowable interactions among processes in a real-time system. One axiom prohibits a process from controlling its own execution, thereby ruling out recursion in a design. Another axiom states that no module controls its own input data space and is therefore unable to alter its input variables. While these axioms are not complete, they are a first step at formalizing specifications for system design.

SUMMARY

Boehm has stated seven principles that have helped organize the techniques discussed in this paper [BOEH76].

1) *Manage using a sequential life cycle plan.* This means to follow the software development life cycle outlined earlier. It allows for feedback which updates previous stages as the consequences of previous decisions become unknown. It encourages milestones to measure progress.

2) *Perform continuous validation.* Certify each new refinement of a module. Use walkthroughs and code reading. Display the hierarchical structure of the system clearly in all documentation.

3) *Maintain disciplined product control.* All output of a project—design documents, source code, user documentation, and so forth—should be formally approved. Changes to documents and program libraries must be strictly monitored and audited. Code reading, project reporting forms, librarians, a development library, and a project notebook all contribute to this goal.

4) *Use enhanced top-down structured programming.* PL/I and PASCAL have good control and data structures. Preprocessors exist which augment FORTRAN for these structures. Description techniques such as stepwise refinement, nested data abstractions, and data flow networks should be used.

5) *Maintain clear accountability.* Use milestones to measure progress, and a project notebook to monitor each individual's efforts.

6) *Use better and fewer people.* The chief programmer team, in which each individual is skilled and accountable for his actions, and good results are rewarded, aids in this effort.

7) *Maintain commitment to improve process.* Settle only for the best; strive for improvement. Be open to new develop-

ments in software engineering, but do not sacrifice reliability for modifiability while pursuing them.

Progress has been made in understanding how large-scale software systems are built, yet more needs to be done. Management aids must be improved and project control techniques developed. The role of software management is coming more to resemble that of engineering management in other disciplines. We can no longer afford costly mistakes when systems are so large and we depend so much on them. Most importantly, we must be patient; we need to gain experience on which future theories can rely.

ACKNOWLEDGMENTS

The author is indebted to Peter Denning for his detailed review and to the referees for their valuable comments on this paper. This work was partially supported by grant number DCR 74-11520-AO1 from the National Science Foundation to the National Bureau of Standards.

REFERENCES

[AHO72] AHO, A.; AND ULLMAN, J. *Theory of parsing, translation, and compiling.* Prentice Hall, Inc., Englewood Cliffs, N. J., 1972.

[ANSI66] *American Standard FORTRAN.* American Natl. Standards Inst., x3.9-1966, March, 1966.

[ANSI76] *American Standard PL/1.* American Natl. Standards Inst., x.53-1976, Aug., 1976.

[BAKE72] BAKER, F. T. "Chief programmer team management of production programming," *IBM Syst. J.* 11, 1 (1972), 56–73.

[BASI78] BASILI, V.; AND ZELKOWITZ, M. "Analyzing medium scale software development," *Third Int. Conf. Software Engineering*, 1978.

[BASI75] BASILI, V.; AND TURNER, A. J. "Iterative enhancement: a practical technique for software development," *IEEE Trans. Softw. Eng.* 1, 4 (Dec. 1975), 390–396.

[BOEH75] BOEHM, B.; MCCLEAN, R.; AND URFRIG, D. "Some experience with automated aids to the design of large scale reliable software," *Int. Conf. on Reliable Software*, 1975, ACM, New York, pp. 105–113.

[BOEH77] BOEHM, B. "Seven basic principles of software engineering," in *Infotech state of the art report on software engineering techniques*, 1977, Infotech International Ltd., Maidenhead, UK, 1976.

[BRIN77] BRINCH HANSEN, P. *Architectures of concurrent programs.* Prentice Hall, Inc., Englewood Cliffs, N. J., 1977.

[BROO75] BROOKS, F. P. *The mythical man month,* Addison-Wesley Publ. Co., Reading, Mass., 1975.

[CAIN75] CAINE, S. H.; AND GORDON, E. K. "PDL—a tool for software design," in *Proc. 1975 AFIPS Natl. Computer Conf.*, Vol. 44, AFIPS Press, Montvale, N. J., pp. 271–276.

[CONW78] CONWAY, R. *A primer on disciplined programming,* Winthrop Publishers, Cambridge, Mass., 1978.

[CONW73] CONWAY, R.; AND WILCOX, T. "Design and implementation of a diagnostic compiler for PL/1," *Commun. ACM* 16, 3 (March 1973), 169–179.

[COOL65] COOLEY, J. W.; AND TUKEY, J. W. "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19, 90 (1965), 299–301.

[DAVI77] DAVIS, C. G.; AND VICK, C. R. "The software development system," *IEEE Trans. Softw. Eng.* 3, 1 (Jan., 1977), 69–84.

[DENN76a] DENNING, P. J. "A hard look at structured programming," in *Infotech state of the art report on structured programming*, 1976, Infotech International Ltd., Maidenhead, UK, pp. 183–202.

[DENN76b] DENNING, P. J. "Fault tolerant operating systems," *Comput. Surv.* 8, 4 (Dec. 1976), 359–389.

[DIJK68] DIJKSTRA, E. "GOTO statement considered harmful," *Commun. ACM* 11, 3 (March 1968), 147–148.

[DIJK76] DIJKSTRA, E. *A discipline of programming,* Prentice Hall, Inc., Englewood Cliffs, N. J., 1976.

[DOLO76] DOLOTTA, T. A.; AND MASHEY, J. R. "An introduction to the programmer's workbench," in *Second Int. Conf. Software Engineering*, 1976, pp. 164–168.

[ENR61] "Everything about the Narrows Bridge is big, bigger, or biggest," *Eng. News Record* 166, June 29, 1961, 24–28.

[ENR64] "Narrows Bridge opens to traffic," *Eng. News Record* 173, Nov. 19, 1964, 33.

[ENR77] "Alaskan pipe cost probe hits snag," *Eng. News Record* 198, April 7, 1977, 14.

[FIFE77] FIFE, D. *Computer software management: a primer for project management and quality control,* Natl. Bureau of Standards, Inst. Computer Sciences and Technology, Special Publications, April 1977.

[GALL65] GALLAGHER, P. F. *Project estimating by engineering methods,* Hayden Book Co., New York, 1965.

[GERH76] GERHART, S.; AND YELOWITZ, L. "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. Softw. Eng.* 2, 3 (Sept. 1976), 195–207.

[GOOD75] GOODENOUGH, J. B.; AND GERHART, S. "Toward a theory of test data selection," *IEEE Trans. Softw. Eng.* 1, 2 (June 1975), 156–173.

[HALS77] HALSTEAD, M. *Elements of software science,* Elsevier North Holland, Inc., New York, 1977.

[HAMI76] HAMILTON, M.; AND ZELDIN, S. "Higher order software—a methodology for defining software," *IEEE Trans. Softw. Eng.* 2, 1 (March 1976), 9–32.

[HOAR69] HOARE, C. A. R. "An axiomatic basis for computer programming," *Commun*

ACM 12, 10 (Oct. 1969), 576–580, 583.

[HUAN75] HUANG, J. C. "An approach to program testing," *Comput. Surv.* 7, 3 (Sept. 1975), 113–128.

[JEFF77] JEFFERY, S.; AND LINDEN, T. "Software engineering is engineering," in *IEEE Computer Science and Engineering Curricula Workshop*, 1977, IEEE, New York, pp. 112–115.

[KING69] KING, J. C. "A program verifier," PhD Dissertation, Computer Science Dept., Carnegie-Mellon Univ. Pittsburgh, Pa., 1969.

[KNUT71] KNUTH, D. "An empirical study of FORTRAN programs," *Softw. Pract. Exper.* 1, 2 (1971), 105–133.

[KNUT74] KNUTH, D. "Structured programming with statements," *Comput. Surv.* 6, 4 (Dec. 1974), 261–301.

[LISK75] LISKOV, B.; AND ZILLES, S. "Specification techniques for data abstractions," *IEEE Trans. Softw. Eng.* 1, 1 (1975), 9–19.

[LISK77] LISKOV, B.; SNYDER, A.; ATKINSON, R.; AND SCHAFFERT, C. "Abstraction mechanisms in CLU," *Commun. ACM* 20, 8 (Aug. 1977), 564–576.

[LOWR69] LOWRY, E. S.; AND MEDLOCK, C. W. "Object code optimization," *Commun. ACM* 12, 1 (Jan. 1969), 13–22.

[MILL76] MILLS, H. D. "Software development," *IEEE Trans. Softw. Eng.* 2, 4 (1976), 265–273.

[PARN72] PARNAS, D. L. "On the criteria for decomposing systems into modules," *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058.

[PARN75] PARNAS, D. L. "The influence of software structure on reliability," in *Int. Conf. Reliable Software*, 1975, pp. 358–362; (*ACM SIGPLAN Notices* 10, 6 June 1975).

[POPE77] POPEK, G. J.; HORNING, J. J.; LAMPSON, B. W.; MITCHELL, J. G.; AND LONDON, R. L. "Notes on the design of EUCLID," in *Proc. ACM Conf. Language Design for Reliable Software*, ACM, New York, 1977, pp. 11–18.

[PUTN77] PUTNAM, L.; AND WOLVERTON, R. *Quantitative management: software cost estimating*, (tutorial), IEEE Computer Society, Nov. 1977, IEEE, New York.

[RAMA75] RAMAMOORTHY, C. V.; AND HO, S. F. "Testing large software with automated software evaluation systems," *IEEE Trans. Softw. Eng.* 1, 1 (1975), 46–58.

[REIF76] REIFER, D. J. "The structured FORTRAN dilemma," *SIGPLAN Notices* 11, 2 (1976), 30–32.

[RITC74] RITCHIE, D. M.; AND THOMPSON, K. "The UNIX time-sharing system," *Commun. ACM* 17, 7 (July 1974), 365–375.

[TEIC77] TEICHROEW, D.; AND HERSHEY, E. A. "PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Softw. Eng.* 3, 1 (1977), 41–48.

[WALS77] WALSTON, C. E.; AND FELIX, C. P. "A method of programming measurements and estimation," *IBM Syst. J.* 16, 1 (1977), 54–73.

[WEID77] WEIDE, B. "A survey of analysis techniques for discrete algorithms," *Comput. Surv.* 9, 4 (Dec. 1977), 291–313.

[WEIN71] WEINBERG, G. M. *The psychology of computer programming*, Van Nostrand Reinhold, New York, 1971.

[WIRT71] WIRTH, N. "Program development by stepwise refinement," *Commun. ACM* 14, 4 (April 1971), 221–227.

[WIRT74] WIRTH, N. "On the composition of well-structured programs," *Comput. Surv.* 6, 4 (Dec. 1974), 247–259.

[WOLV74] WOLVERTON, R. W. "The cost of developing large scale software," *IEEE Trans. Comput.* 23, 6 (1974), 615–636.

[WULF76] WULF, W.; LONDON, R.; SHAW, M. "An introduction to the construction and verification of ALPHARD programs," *IEEE Trans. Softw. Eng.* 2, 4 (1976), 253–264.

[YOUN67] YOUNGER, D. "Recognition and parsing of context-free languages in time n^3," *Inf. Control* 10, 2 (1967), 189–208.

[ZELK75] ZELKOWITZ, M. V. "Third generation compiler design," in *ACM Natl. Comput. Conf.*, 1975, ACM, New York, pp. 253–258.

RECEIVED MARCH 14, 1977; FINAL REVISION ACCEPTED MARCH 7, 1978

Making The Move To Structured Programming

by Edward Yourdon

The best way to ensure that people will resist the change is to try to implement all the new techniques at one time.

The most common objection to structured programming takes the form, "Gosh, it sounds great and we'd like to do it, but. . . ." More specific attacks have been leveled against the PERFORM statement and other forms of subroutine calls, against nested IF statements, and against the elimination of GOTO statements. A more subtle and powerful form of attack is this: "Oh, you're just talking about modular programming; we've been doing that for ten years!"

For structured programming to have the proper opportunity to show its strengths and not be rejected outright by an organization, consideration should be given to these questions:

1. What will the specific objections to it be? Are there any potential disadvantages that may be experienced as a result of using the techniques?

2. Which of the techniques should be attempted first, assuming that you cannot use them all? For example, should you attempt to use structured programming first, and then try top-down design on a later project?

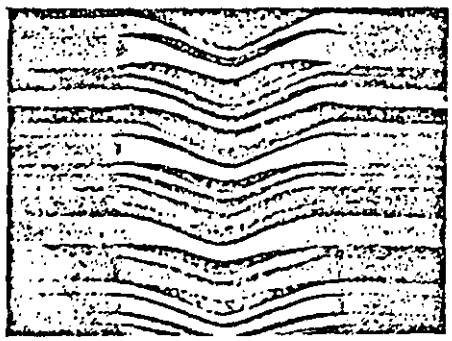
3. What kind of programming project should you begin with as an experiment to demonstrate the benefits of the structured programming techniques?

4. How should you evaluate the success of the experiment?

Common objections

It would be unrealistic to assume that structured programming, top-down design, chief programmer teams, structured walkthroughs (one programmer explains his code to others), program librarians, and structured design would be accepted without argument in any organization. Here are some of the more common objections:

1. Many managers and programmers point out that the structured programming techniques are primarily intended for new development projects; for maintenance of existing unstructured programs, they seem to be of limited use (though the librarian concept and the structured walkthrough concept would still be quite useful). This point is basically valid, though it is usually possible to add new



sections of code in a top-down structured manner, e.g., when completely new features are being added to a system at the request of a user.

This problem may be solved eventually with the aid of "structuring engines" that will automatically convert unstructured logic into structured form; while such an "engine" cannot magically transform "bad" code into "good" code, it will at least foster some standardization.

2. Some managers point out that their programming projects are typically too small to require a team of programmers; therefore, they argue, they don't need any of the new "programming productivity" techniques. Since most managers apparently are not prepared to fire most of their mediocre programmers and replace them with one highly competent chief programmer, we must accept this objection as a fairly valid one—but only for the chief programmer team concept.

There is no reason why the existing programmers in the organization, even working by themselves, could not use structured programming and top-down design.

3. Still other managers object to the cost of training their programmers in the techniques of structured programming. This training exercise is admittedly nontrivial, though it depends on the programmer's experience. (Junior programmers learn the techniques more easily than senior programmers; the author's group trained 120 programmers in an Australian government agency prior to beginning our payroll project, and of the 20 who scored at the top of the class, eight were novice programmers with less than six months experience.)

Ease of training also depends on the programming language; the techniques are generally easiest in PL/1, reasonably easy in COBOL, and more difficult in FORTRAN and assembly language. In general, we found that programmers require three to five days of classroom training to learn the techniques, and approximately one month of programming (when they are at least as productive as they were previously) to become comfortable with the new techniques. The investment in training is thus relatively small compared to the benefits that were discussed in the preceding section.

4. The manager often has to overcome technical objections raised by the programmers; these objections most frequently come from senior programmers, many of whom are now project leaders, who still fondly recall the "good old days" of the 1401 and the 650. The common programmer-oriented objections are: it is awkward and inconvenient at first; the programming language is inadequate for the strict discipline imposed by structured programming; it is not obvious that the new approach will actually reap the benefits discussed above; and finally, there is a concern that the top-down structured approach will lead to tremendously inefficient programs.

The awkwardness and inconvenience is largely a matter of training. The question of language adequacy can be a relevant one, and one answer might be to convince the programmers (and their managers!) to begin using PL/1 and the other ALGOL-like languages in preference to the primitive COBOL-like and FORTRAN-like languages.

The objection about efficiency can only be answered by appealing to the programmer's common sense: the battle for efficiency is generally won or lost at the system or program design level (e.g., by making sure the system is running on an efficient hardware configuration, and that it isn't doing things it wasn't intended to do), and not at the bit-fiddling level (i.e., where the programmer "wastes" a microsecond or two indulging in a subroutine call).

Obviously, there are some exceptions to this, in some real-time systems, for example; but as Professor Bill Wulf of Carnegie-Mellon Univ. points out, "More computing sins are committed in name of efficiency (without nec- ...y achieving it) than for any other single reason—including blind stupidity."

5. In addition to programmer retraining costs, many managers object to the cost of developing new standards to conform to the new programming techniques. Some organizations may have only recently finished developing standards for testing—and they tend to be "bottom-up" standards, in direct contrast to the top-down methodology currently being advocated. While the cost of developing new standards may well be substantial, it is difficult to think of any way of avoiding it. Indeed, even if structured programming had not appeared on the scene, surely some new techniques would eventually appear, forcing the voluminous standards manuals to be rewritten . . . it seems to be an inevitable fact of life.

This may be an academic point, but I know of one large bank and one insurance company that have apparently rejected structured programming for this reason alone—all of which seems rather sad.

Some managers have expressed th. ar that the structured programming concepts might not work (or that their programmers might not be sufficiently familiar with the new techniques) on a significant project whose failure would have disastrous consequences. There is a very simple solution to this problem: if you have not tried structured programming before, you probably should not use a critical project as a "guinea pig."

I have seen three projects in early 1975 fail in their attempt to use various aspects of structured programming. One midwestern company used top-down testing at the program level, but then integrated the programs in a bottom-up fashion to build a system—and they couldn't understand why the "top-down" approach had failed to give them miraculous results.

Another company in New England made an unsuccessful attempt at using the program librarian concept, but it appears that the librarian was a secretary who was required to spend six hours a day typing envelopes and the other two hours supporting the progr... team.

... another programming project failed in its attempt to use the chief programmer concept. In apparent ignorance of the whole philosophy of the concept, their chief programmer did not write any code during the entire project!

7. Some managers are concerned that they may not be able to see the benefits of the new techniques. This is often because they have no statistics to compare their current techniques with the new ones. To be more blunt, many organizations have no idea how many lines of debugged code their programmers generate each day, nor how many test shots the average programmer requires before he delivers his program to the user. Nor have they any idea how many residual bugs are found in programs that have been delivered to the user. Thus, one of the first results of structured programming may be an unpleasant awareness of just how bad things are; while this may well be unpleasant, it can also be a healthy shock.

The lack of statistical evidence has been used as a reason for not using structured programming; it usually takes the form of: "Oh, well, I'm pretty sure our programmers are above average anyway, so we don't need to use these new techniques."

Sometimes the problem is more blatantly political: when management does find out how bad the programming productivity currently is, they try very hard to cover it up to avoid the obvious accusation that they have been doing their job poorly for the past several years. A battle of precisely this sort is going on in one of the larger dp organizations in Detroit, where one of the people on the research staff has made some rather interesting studies based on a sample of 100,000 PL/1 statements: the average module size was 900 statements (so much for small, independent modules!); only four statements were DO-WHILE statements (so much for the assumption that all PL/1 programmers instinctively know about the three basic forms of structured coding); in a substantial number of the programs, none of the IF statements had an ELSE clause; and various other statistics suggest that the programmers have been writing "rat's-nest" code for the past several years.

8. There is an interesting variation on the preceding objection: some managers worry that the structured programming techniques may improve the productivity of their programmers by only 10% instead of the five-fold improvement generally advertised. Of course, this may be because the programmers were already following an informal semi-structured, semi-top-down approach.

Nevertheless, some managers worry that they (and presumably their programmers as well) will be judged incompetent if they experience less than a five-fold improvement! One hardly knows what to say about this head-in-the-sand objection, except the obvious point that a 10% improvement in productivity (with commensurate improvements in software reliability and maintenance) is better than no improvement at all!

9. Finally, some managers suggest that the fanfare and publicity associated with structured programming may be a disguised form of the Hawthorne effect—i.e., the programmers are more productive because they know they are being observed. It is hard to believe that any manager who has the slightest familiarity with programming would believe this, though perhaps that is the problem—many dp managers are about as familiar with programming as they are with the theory of relativity.

Even if the Hawthorne effect were relevant, so what? Why fight it? On the Australian payroll project, I was criticized for giving the programmers tee-shirts that had the word "superprogrammer" emblazoned on the front. Some people felt this was an "unfair" method of attempting to increase their productivity! But if it contributed to a five-fold improvement in programming productivity, it was worth it!

Picking the right combination

As mentioned earlier, some organizations are concerned about the difficulty of implementing all of the new programming techniques simultaneously. In most such discussions, four major techniques are considered: structured programming, top-down implementation, chief programmer teams (with structured walkthroughs), and the program librarian. In even the most progressive organization, it can be difficult to implement all of these techniques at the same time; other organizations feel that some of the techniques are simply not applicable for their programming projects.

It is important to emphasize that while the four techniques are usually used together, they do not have to be. It is possible to use structured programming without top-down implementation, or top-down implementation without structured programming. Similarly, it is possible to use the chief programmer approach without using the librarian concept, or vice versa. And it is possible to use the chief programmer concept and the librarian concept without using structured programming or top-down design.

While each of the concepts can be used separately, some of them are almost inevitably joined with others. If an organization decides to use the chief programmer team approach, it almost always uses the librarian and some form of structured walkthroughs; on the other hand, I have seen several projects where the librarian concept was used without the chief program-

mer team concept. Similarly, if structured programming is used, top-down design is also used; conversely, if an organization decides to use top-down design and top-down testing, it is possible that they may elect not to use structured programming (this seems especially true in COBOL shops that have been brainwashed for years to avoid nested IF statements).

It is difficult to make any general suggestions about the order in which the techniques should be implemented in a typical organization. Perhaps the most important point to recognize is that a sharp distinction can be made between the technical concepts of structured programming, top-down implementation, and structured design, and between the organizational concepts of chief programmer teams, structured walkthroughs, and program librarians. One should choose the combination of techniques that will give the greatest improvement for the least effort.

The librarian concept can usually be implemented fairly easily, since it does not affect the user and does not require any major retraining on the part of the programmers (though it does get them to change their attitudes: "Whaddya mean I can't type my own program at the time-sharing terminal? So what if I can only type two words a minute?"). On the other hand, it does require some standards to be developed for proper use of the librarian, as well as some training for the librarian; and it does cause some problems for organizations that find they have no budgets, no "job titles," and no place in the organization chart for the librarian. Except in government agencies and some other large bureaucratic organizations I have visited recently, these are usually minor problems.

Top-down design, structured programming, and the chief programmer concept tend to have a larger impact on an organization, and should be implemented with more care. Structured programming generally implies "GOTO-less" programming (or at least less GOTO programming), which is a jolt to many experienced programmers. The chief programmer team approach usually includes the concept of structured walkthroughs which suggests that each programmer should make a formal presentation of his code to all of the programmers on the team for their review and criticism. This too is a jolt to the average programmer (as a programmer at Shell Oil in London said, "Reading someone else's program is like reading their personal mail—it's an invasion of privacy in which civilized people simply do not indulge").

Top-down design and top-down testing are a bit easier for the programmer to swallow, but they can cause severe management disruptions in some cases. They imply, for example, that a computer is available for testing at an early stage in the project (in contrast to the common management policy of not supplying machine time until the final stages of the project).

None of these problems is insurmountable; indeed, many organizations have implemented all of the techniques simultaneously without experiencing any major catastrophes. Nevertheless, the cautious manager may wish to begin with only one new idea at a time; specific conditions within the organization will usually dictate which technique is to be implemented first.

A good pilot project

In some organizations, the techniques of structured programming and top-down design, after some experience with them, are considered "intuitively obvious" to managers and programmers alike. Once exposed to the concepts, everyone begins using the techniques without any significant prodding. Many organizations start using structured programming with a great deal of trepidation; in most cases, this is done by using the techniques on an "experimental" project.

This leads to an obvious question: what kind of project should be used as an experiment? Experience with several organizations during the past two years suggests the following:

1. *The project should be visible.* If the experiment is an academic project that nobody will use, then nobody will care about the results. The project should be a "real" one—one that users will use, and one whose results people will care about. Indeed, this is one of the reasons the New York Times project had such a profound impact upon IBM and the rest of the industry—it was real.

2. *The project should be nontrivial.* One of the keystones of the "structured" approach is its attempt to break complex tasks into simpler ones. By working with less complex program components, the programmer is less likely to make mistakes, and he is more likely to produce something that can be maintained. However, if the program is already quite trivial (e.g., less than 100 lines of code), the improvement in productivity and maintenance will not be as obvious. The experimental project should be at least 4 man-months in duration.

3. *The project should be noncritical.* There are enough problems in becoming familiar with structured programming and top-down implementation in a relatively small project; there is no point in putting additional pressure on the programmers by forcing them to use the techniques on a critical project—one whose failure will have disastrous consequences in the organization.

On the other hand, if management and programmers agree that the techniques are "intuitively obvious" and feel relatively comfortable with the techniques (a more and more common occurrence, considering that most programmers now graduating with a B.S. in Computer Science have had a healthy exposure to structured programming), then there should be no danger.

There is the case of an organization that was faced with an "impossible" project and in desperation used it as a pilot structured programming project. The Australian Taxation Dept. (roughly equivalent to the IRS) was urged in 1973 to adopt structured programming, top-down design, chief programmer teams, and program librarians to assist in a project involving conversion of machines, conversion of languages, and redesign of major existing systems within a timeframe that would otherwise have been considered impossible. The results were highly successful.

Evaluating the experiment

Clearly, the object of an experiment is to gain information for future reference: a structured programming experiment should give the organization valuable information about the usefulness of the techniques for future projects. From the discussion in the previous sections, we see that there are several statistics the project manager should try to capture; most of these can be gathered by answering the following questions:

a. Was the project finished on schedule? How accurate were the estimates for milestones during the project? At various stages during the project, was it possible to estimate accurately the amount of coding remaining to be done?

b. How productive were the programmers? In particular, how many debugged statements per day were they able to produce? Also, how did the programmers distribute their working time during the project (some tentative results from an experiment at Aetna Life & Casualty suggest that slightly under 6% of the programmer's time was spent on code reviews and walkthroughs, which seems to squelch the common complaint that these things take too much time).

c. How efficient were the resulting programs? This may be difficult to judge unless the program is a redesigned version of an earlier (presum-

ably unstructured) program. However, the manager and/or the programmers should be able to make some qualitative judgments about the presence or absence of substantial overhead in the final program.

d. How much test time was required for the project? Specifically, was the test time distributed fairly evenly throughout the project?

e. What was the ratio of development costs to maintenance costs? Is it significantly better or worse than other "unstructured" projects within the organization?

f. How effective were the structured walkthroughs? Roughly how many bugs were found per man-hour of walkthrough, and roughly how long would it have taken the original programmer to find those bugs? More important, how many of the bugs would have remained unnoticed until the program began running in production?

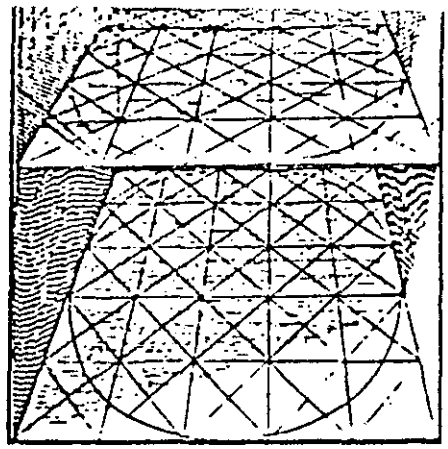
I find that a reviewing audience can easily spot five to six bugs in a 200 statement program within 15 minutes. It is important to recognize that the programmer who wrote the code was usually convinced that his code was correct, and his test data would usually be a self-serving attempt to confirm that feeling.

g. How many bugs were discovered during user acceptance testing? How many bugs were discovered after several months of production? How does this compare with other "normal" programs in the organization? □

Mr. Yourdon, president of Yourdon Inc., has consulted and lectured on program design and on-line computer systems in the U.S., Canada, Europe, and Australia. He began his career at Digital Equipment Corp., where he developed assemblers, FORTRAN IV Executives, and math libraries for various machines. At General Electric, he developed an operating system for a hospital information system on the GE-635. He has authored several books and numerous articles, and is currently completing a two-volume series, "Advanced Programming Techniques."

a DATAMATION. reprint

An Example of Structured Design



An Example of Structured Design

by Bill Inmon

Producing 11 lines of code per hour through structured design and programming is one of the advantages.

Quick and accurate development of computer programs is a longstanding goal of the data processing community. Recently programmers and program designers have begun to formalize some of their practices by using concepts of structured programming and structured design. How do these concepts interact with the goals of quick and accurate program development?

An example of how one system was developed using these tools in a limited amount of time is given here. The specifics of the system may not be important—it happened to be a cost accounting system, necessitated by a large government contract, which broke down the cost of the final product to costs of component parts at the lowest level. However the problems that arose in the development of this system, and the way they were handled, are important because these same basic problems usually occur in development of most types of systems.

System overview

Here is a profile of the system under discussion:

1. Length of system development —4 months.
2. Manpower—14 man-months (7 programmers—2 full time, 5 part time).
3. Scope of project—tasks included internal design (i.e., the building of detailed programming specifications from a broad description of the problem), programming, unit testing, integrated testing, documentation, and the building and maintenance of a large data base.
4. Total number of lines of coding —approximately 31,000, not including system-support coding, such as Job Control coding, test-data generators, etc.
5. Language—COBOL interfacing the data base with DL1.
6. Machine—IBM 370/145, with TSO.
7. Data base environment.
8. Programmer coding productivity @ 200 hours per month—that is, approximately 11 lines *per hour*.
9. Number of programs and modules—50
10. Median module size — 5.4 pages.
11. Type of application—financial.
12. Technical complexity—mild.
13. Programming logic complexity —moderate.
14. Design considerations—structured programming, structured design.

A group of programmers was organized into a "chief programmer team." The chief programmer was responsible for the design of all programs. He did the actual coding of some critical programs, and organized and coordinated the programming of the other programs in the system. Two programmers were on the team full time and five other programmers contributed to the team effort. Programs were designed adhering to the principles of structured design. Principally, programs were kept small and designed "functionally." Large programs were divided into modules. The coding of programs was structured. No GOTO statements were allowed, and other conventions of structured programming were followed.

The overriding constraint on the system development was time. Because of contractual obligations program design, programming, debugging, and implementation had to be completed in four months. The deadline was not extendable.

The structuring of the data base reflected a strong user orientation and was not designed for easy program manipulation. At the outset of the project, only two programmers had extensive data base experience. In preparation for future on-line considerations, the system was to be updated by transactions. Reports were generated directly from the data base. In terms of programming logic complexity, the system was straightforward except for several internal relationships. Technically, the complexity of the system was not beyond the ability of the chief programmer team.

Design goals

The design goals of the system were greatly influenced by these program development priorities:

1. Speed of development
2. Program and algorithmic accuracy
3. Maintainability of programs
4. System flexibility

The traditional consideration of production run efficiency was only of incidental importance. It was not ignored; it simply gave way to the higher priorities. The major attention was directed towards the building of a workable system. Once the requirements of the system were met, consideration was given to run time efficiency (i.e., when there was enough time to make such a consideration, which usually was not the case).

Several schemes were contemplated for organizing the project. The only way deemed feasible because of the time constraint was to write program specifications while concurrently programming and testing other parts of the system. Obvious pitfalls to this approach were recognized before proceeding; however, because of the lack of time, no other method was possible. This type of project organization probably should be used only when the situation demands it. In such a case, very close control and coordination is an absolute must because poor communications can lead to a large scale waste of effort.

To attain the design goals, all large programs were highly modularized. As much as possible, a module was designed for general usage, so that it would be reusable in other parts of the system. In this way repetitious coding was eliminated or at least drastically reduced. Each module was physically separate from any other module, i.e., the source text which comprised a module was developed, compiled, and stored apart from any other module. The intent of the design was to make modules small, about four pages of source text per module. However, the size of a module was ultimately determined by its function. Each module had a limited and well-defined function, and the environment of the module was likewise limited and well-defined. In this manner, the overall functions of a program were broken into smaller, isolated functions.

Early in the system design, it was recognized that there were different levels of functionality along which a program could be divided. An example of levels of functionality is shown by the difference between two large updating programs that were part of the system. Both programs were to read transactions and update the data base according to the contents of the transaction.

Another way to view an updating program is from the traditional actions of adding, deleting, or replacing data in the data base. In this case, as a transaction enters the program, it is categorized into one of the three major functions, and is handled along those functional lines.

Program flexibility and modularization

The flexibility of the programs in the system was difficult to assess. However two characteristics of the system point to the fact that structured design produces some flexibility. One is that changes in it were made with a minimum of effort. The largest change required four days, and the next largest, less than a day. The second characteristic is the fact that errors were quickly located and corrected. It appeared that the major factor contributing to system flexibility was the isolation by function that was achieved by structured design. When a problem occurs, it is easy to eliminate many modules from consideration since their functions may have nothing to do with the problem.

Data coupling was extensively used in the interfacing of modules with each other. (Data coupling occurs when all input and output to and from the called module are passed as parameters or arguments—i.e., as data elements). The independence gained here enhanced both the reusability of a module and the isolation of a module by function from other modules.

In other ways, the modular approach accelerated the development of the system. The physical separation of modules meant that more than one person could simultaneously contribute to a program. In fact at one time, five programmers were actively working on modules of the same program.

A byproduct of using a modular approach was the lessening of the total learning time involved in the translation of programming specifications into code. This occurred because, as a programmer completed a module, he was assigned another similar one. He still had the previous coding fresh in mind and could therefore quickly grasp the new programming specifications.

Another feature of adopting a modular approach occurred when a programmer completed a module and was then free to work on other systems. In a conventional system, a programmer is tied to a program until he finishes it; and if a problem in his area of expertise arises while writing the program, something must suffer. However, if a program is broken into small modules, many breaking points result naturally, and the problems involved in conventional methods of programming are minimized.

Despite advantages in using a modular approach, there were also some disadvantages. One problem arose in the interfacing of modules. In spite of careful attention given to the flow of data to and from other modules, there still were some errors. The number and order of parameters, their characteristics, and the interpretation of their values were in some cases misunderstood. Another problem arose in the organization necessary for the direction of several people working on the same problem simultaneously. Modules were being developed so rapidly that coordination of testing, linking, and integrating them became a large task.

Structured walkthrough

One of the techniques that led to the quick completion of the project was that of the structured walkthrough. A walkthrough, or mental execution of the program by the programming team, was done for each program.

After a programmer had compiled a program and scrutinized it for obvious errors, he sent the chief programmer and several other programmers involved copies of his source text. After time was allowed for the group to examine the text (usually a few hours), the team met and collectively performed a mental execution of the program. A list of errors was made as each logical path was followed. The interface with modules either calling or called by the module being examined was carefully checked as well. At the end of the walkthough, the list of errors was given to the author.

In this manner most errors were caught before any testing had occurred. Also, the team members who reviewed the text became familiar with a part of the system other than their own. This helped to establish a common base of understanding of all components of the system.

Coincidentally, while the team reviewed the interface with other modules, errors (usually from miscommunication) were also spotted in other modules. It was also not uncommon to have a program or module execute correctly the first time it was tested. One of the major factors in the success of the system was the shortness of the time spent in testing, and the major contributor to it was structured walkthroughs.

Program testing

One reason why testing and debugging went smoothly was the fact that the design of the system included programming specifications for debugging. Not only did programming specifications define the function of a program or module, but they also required variables to be inserted and manipulated solely for the purpose of debugging.

The most effective technique was using an activity variable in each

module. Initially the activity variable is set up as inactive. As a module was invoked, the value of the activity variable was changed to indicate that it was active. As control was relinquished, the variable was returned to the inactive state. When a dump occurred, it was easy to trace the path of active modules, arriving at the final one in which activity had occurred (see Fig. 1).

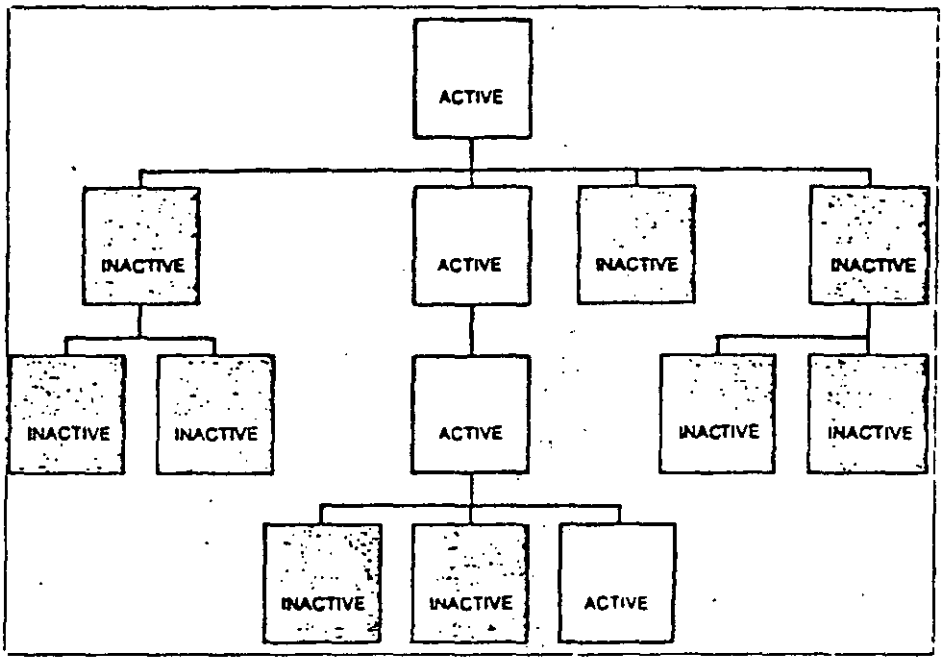


Fig. 1. By using an "activity" variable in each module, the flow of control through the program can be reconstructed and non-active modules ignored in debugging.

Another useful technique was to display the parameters of a module at the time of invocation. Because of data coupling, the complete set of relevant data was available for determining the state of the module.

An additional technique that proved to be useful was the gathering of statistics internally by each module. Typically, a module would keep track of how many times it had been invoked, the parameters it had been called with, what type of internal results it had generated, etc. Data of this type made reconstruction and analysis of program activity an easy task. They also helped to identify areas of code that had never been tested. Also, it helped to locate areas of code that were critical in the efficient running of the program.

Summary

The most pleasing and surprising aspect of the project was the smoothness with which debugging and testing were accomplished. This smoothness is reflected by the high programmer productivity rate of approximately 11 lines per hour. We felt that several factors contributed to this success—structured walkthroughs, the breakdown of large programs into modules, and structured programming.

The major criterion, that of speed of development, was certainly enhanced by structured design. A modular programming approach alone would not have made it possible to write the large programs needed in the amount of time allotted.

The other design goals—flexibility and maintainability—cannot yet be effectively evaluated since the system has remained relatively stable in its short lifetime. The few changes made do point to a high degree of maintainability. Structured design, used in conjunction with the complementary techniques of structured programming and walkthroughs, did make possible a level of speed and accuracy in system development not previously attainable. Based upon our experience with developing this system, structured design is as powerful in practice as has been predicted in theory. ✱

Mr. Inmon is a chief programmer for GTE Sylvania Electronics Systems Group in Mountain View, Calif. He received an M.S. in computer science from New Mexico State Univ.

30

THE NEED FOR SOFTWARE ENGINEERING

Ware Myers
Contributing Editor

Introduction

Early in this decade a set of programming practices began to appear that seemed to offer a way out of the software difficulties accompanying the development of large systems. These practices, developed by Brooks,[1] Baker,[2] Dijkstra,[3] Mills,[4,5] and others, included structured programming, top-down development, chief programmer teams, HIPO (hierarchy/input-process-output) documentation, development support library, and structured walk-throughs. But despite the increasing amount of software development and its rising cost relative to the defense budget, corporation expenditures, and even the gross national product, the new programming techniques have not been adopted by acclamation. McClure,[6] surveying the scene at COMPCON '76 Spring, saw "the great masses of programmers conducting their business exactly as they did five years ago." Nor was there the slightest sign in McClure's 5-year projection of "strong winds of change." His intuition was later supported by a survey of major Los Angeles area corporations,[7] which concluded that, for all the fanfare, "the techniques are simply not widely used."

Why are modern programming practices propagating so slowly? In the judgment of some seasoned observers, the reason lies in the complexity of the techniques and the difficulties management and programmers face in implementing them. As with modern management practices, modern programming practices are intangibles that have to be disseminated by "soft" means such as education and training.

Fortunately, there are positive forces at work—the Department of Defense, NASA, IBM, defense and space contractors, software houses, and universities. The professional societies cover the area in their conferences, tutorials, and journals. The General Accounting Office is studying the use and value of software development techniques. These influences, together with the growing realization that projected software development costs are becoming a greater factor than hardware costs in deciding to develop a system, are literally forcing progress to be made.

The software predicament

The general character of the software predicament can be seen clearly, although consistent numbers with which to characterize it more precisely are hard to come by. Because less expensive hardware is bringing more applications within economic reach, the amount of software to be developed is increasing. Also, because more software is already in existence, there is more of it to be maintained. But the productivity of programmers is improving rather slowly, especially by the standards of hardware price/performance, with the result that the overall cost of software development is tending to increase. Or, if this increase is being limited by budgetary considerations, opportunities to apply computer technology more fully to both old and new applications are being passed over.

Software growth. Estimates of the overall cost of software development and maintenance in the United States range from $15 to $25 billion.[8] At DOD it is running in the $3 billion-per-year range. It represents 4-5 percent of the Air Force budget, 6 percent of the NASA budget.

According to Walter R. Beam,[9] deputy for advanced technology to the assistant secretary of the Air Force for research, development, and logistics, the number of functions—most of them new—being done by software is growing at a prodigious rate. Beam, who gave the software technical keynote at

COMPCON 77 Fall in Washington, D.C., last September, estimated this rate to be a factor of 2 every three years. The rate is probably greater in defense than in industry, because DOD is moving rapidly from analog to digital weapon systems. For example, recently purchased aircraft are heavily software controlled because these new systems are more effective than the old hardware systems.

In another area Gnostic Concepts[10] projects data processing spending, exclusive of office automation, to increase at about 15 percent per year, while information systems business grows at better than 20 percent.

Operations-type systems for steel mills, retail stores, banks, etc., are growing at about twice the rate of other applications, according to Joe M. Henson, vice president for market planning at IBM's Data Processing Division, in another technical keynote address given at COMPCON 77 Fall.[11] Because these systems are inherently complex, they require more programming manhours than simpler systems. Henson projected the number of on-line systems to grow from 9500 in 1975 to 23,000 by 1980. More systems are coming in the area of management and technical planning, such as forecasting, modeling, and design, and they will demand more data. Meeting this need through large integrated data bases will again add to the volume of software development.

Last November the computer industry was reminded that not only do business system manufacturers not agree on standardized programs; users aren't settling for them anyway. "We're going the other way as customers demand even more diversity," Jay R. Hosler, computer systems consultant, told Interface West.[12] Small business systems are only a part of the total software picture, but they exemplify one of the ways the programming workload grows.

Maintenance ratio. Because a great deal of software is already in existence, some of it of low quality, more and more effort, of necessity, has to be devoted to maintenance. Henson, for example, estimated that up to 80 percent of his company's application development resources are devoted to maintenance. Similar figures were reported by Elshoff,[13] who noted that 75 percent of General Motors' commercial software effort was spent on maintenance. This ratio, according to Elshoff, is fairly typical of large-industry software activities. As more software manpower goes into maintenance, less is available for new development.

Productivity. Estimates of the long-term productivity improvement rate of programmers range from about 3 percent to 7 percent. For example, IFIP President Richard I. Tanaka estimated the current rate to be about 3 percent per year.[14] Henson gave a 4 percent to 7 percent rate, depending on the assumptions made, for the 30-year period from 1955 to 1985.

Within IBM, on large programming projects, "productivity, measured in terms of the number of lines of code produced per programmer, has improved at about 20 percent over the years, due in part to the increasing use of higher level languages," according to Ted Climis of IBM, addressing the keynote session of COMPSAC 77.[15] Climis, a vice-president of his company's General Products Division, expected this rate to continue, but noted that productivity data on programs of under 20,000 lines of code is more variable and seems to depend largely upon the individuals involved.

Indeed, as Climis' last statement suggests, measuring programmer productivity is a complicated undertaking. For example, one investigator, finding that reported productivities ranged from one line of code per hour to 30 or more, concluded that more precise definitions of a program, a manday, and even a line of code itself were needed. Using his own definitions Johnson[16] found a productivity range on 16 commercial systems-programming products ranging from about 9 lines of code per hour (average of five smaller projects) to about 3 lines of code per hour (average of 11 larger projects). In general, his results were close to those published earlier by Fred Brooks.[1]

Software/hardware ratio. Because the cost of hardware is dropping rapidly—an order-of-magnitude improvement in the hardware price/performance ratio every 10 years—while software productivity improves only slowly, the cost of software relative to hardware is increasing. This change has been noted by many observers. Tanaka, for example, saw information processing becoming the most labor-intensive of all industries by 1985, if better methods were not used. Beam characterized it as a "cottage industry"—hardly an image compatible with the notion of computers as typifying modern technology.

In NASA the software/hardware ratio was about 2:1 five years ago. Writing for *Datamation* in 1973,[8] Boehm projected the ratio for the Air Force as going to 10:1 by 1985. On one existing program, the World Wide Military Command and Control System, the ratio was already in that vicinity—$722 million for software to $50-$100 million for hardware. This system, with 35 locations, ties the President and the Pentagon with commands in the United States, Europe, and the Pacific, using airborne command posts in part.[17]

Reliability. The enormous size of WWMCCS offers a dramatic insight into another dimension of the software predicament—the need for correct operation. This need was highlighted by a recent news report by Greg Rushford, who writes on national security subjects: "The record of military communications is replete with failures, stemming in no small part from the system's complexities."[18] He attributed to communications difficulties at least part of the blame for such serious incidents as the

Gulf of Tonkin crisis in the Vietnam war, the bombing of a U.S. warship off the Sinai peninsula by the Israelis, the Pueblo affair off North Korea, and others. These breakdowns in communications occurred before the new computer-controlled system was installed, but they underscore the critical importance of its correct operation.

Before modern programming came into use, programs were large, complex, neither modular nor portable, almost unreadable, and expensive to maintain.

Program quality. What were typical programs like before modern programming practices? Elshoff[13] analyzed 120 PL/I programs collected from General Motors commercial computing installations in late 1973. He concluded that the individual programs were not modularized and were quite large—853 PL/I statements on the average. They were very complex, not portable, almost unreadable by others, and expensive to maintain.* From interviews with GM programmers who had worked elsewhere, he learned that this state of affairs seemed to be typical of other installations as well—especially Cobol installations. Perhaps this was roughly the stage of the programming art when modern programming practices appeared on the scene.

Need for discipline

The notion that a more disciplined approach to programming would alleviate the software predicament has been current for at least 10 years. It was, in fact, the belief that systematic engineering methods could be applied to the software process that led to the coining of the term "software engineering" in 1968.[19]

The engineering way. The very term, engineering, implies "that the entire development of a product from initial conception through testing and maintenance is organized in an orderly, manageable way. The quality, performance, and cost of the product must be predictable, and an appropriate compromise between cost and reliability must be achieved."[20]

Engineering development generally proceeds through a series of stages: product planning, specification, design, documentation, fabrication, and test, with engineering change control running through the sequence. The development of any particular product may pass through this sequence several times, as model, prototype, and finally production. Engineers are trained in these stages from school onward.

A series of "dragons" enforce the discipline—the machinist or technician wants an exact print he can build to, the drafting manager insists that documentation follow the rules, component engineers try to standardize the building blocks, and manufacturing engineers pour over the drawings to establish that they are complete.

The software way. The software process differs from the hardware process in many ways. For one thing the fabrication step, in the sense of reproducing the program tapes, is insignificant. Also, the habit of going through the stages several times has not caught on in spite of Brooks' advice: "Plan to throw one away; you will, anyhow."

In addition, past practice has tended to start with instruction writing and then work back to the earlier stages of design and requirements. This method worked well enough on the relatively small programs that characterized the early history of programming, but it ran out of steam on large developments.

Another factor in the programming way is the backgrounds of its practitioners. They tend to come from more varied sources than do the hardware engineers, who are usually the products of a systematic 4- or 5-year curriculum. Programmers may have degrees in mathematics, English, history, journalism, or whatever. Whatever they have learned—and it may be a great deal—it probably does not include engineering methods. This lack of common background may be part of the cause for Brooks' complaint that "techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering."

A comparison. It seems intuitively that systematic development procedures would lead to better results. Daly[21] attempted to measure the difference in the results obtained by the disciplined engineering approach and the less disciplined programming approach (before modern programming practices). For this purpose he studied the recorded statistics of a large real-time system consisting of 160,000 instructions and 170,000 logic gates (not counting gates in duplicated circuits). He felt the size and complexity in each area were about the same.

Here is what he found:

- twice as much effort had been required to develop an instruction as a logic gate;
- four times as many design maintenance corrections had been made per instruction as per gate;
- four times as much cost had been incurred for design maintenance of software as hardware.

Daly thought he recognized five underlying reasons:

- management techniques and development procedures were more advanced in hardware than in software;
- hardware designers were more experienced and employed a more "structured design."

*Elshoff warned that the conditions were as of 1973. Since then GM has made many improvements.

- the basic building blocks used in software design (i.e., different types of source statements) were more numerous and complex than the AND, OR, and NOT concepts used in hardware;
- hardware got a double dose of testing and evaluation, one by itself and the second as a natural byproduct of software testing.

Software engineering process

In recent years the stages of the software process have been analyzed by many researchers and practitioners.[11] This work is summarized in Table 1, which presents an ordering of the eight main stages of software development and a second-order listing of some of the management and design practices characteristic of each stage.

The concept of a design stage in software and its separation from the programming stage is relatively new. In the design stage the intent is to work out completely and unambiguously the software system necessary to meet the specifications, using English, pidgin English, or a program design language. No instructions or code are written in this stage.

Managers complain that programmers resist the new ideas, while programmers retort that managers don't understand the problem.

The purpose is to create a logical structure which can be checked back against the specifications and internally within its own structure before proceeding to the additional labor of writing instructions in a high- or low-level programming language.

There may be exceptions, however, to the write-no-instructions rule. One is when some novel problem must be solved at the programming level to assure that the design-level structure is workable. Another is reiteration, in which problems occurring in a later stage have to be fed back and necessarily cause changes in an earlier stage.

Table 1. The software development process.

1. REQUIREMENTS ANALYSIS AND DEFINITION[23]
2. SPECIFICATIONS:[24] AS A
   - CHECKPOINT FOR AGREEMENT BY USER AND DEVELOPER ON THE FUNCTIONS REQUIRED;
   - REFERENCE POINT FOR DESIGN DEVELOPMENT;
   - COMPARISON POINT FOR PROGRAM VERIFICATION;
   - PLANNING POINT FOR PROGRAM TESTING;
3. DESIGN (IN THIS STAGE MANY OF THE NEW PROGRAMMING PRACTICES ARE USEFUL):
   - TEAM CONCEPT: CHIEF PROGRAMMER, SUPPORT LIBRARIAN, ETC
   - TOP-DOWN DESIGN, OR STRUCTURED DESIGN, COMPOSITE DESIGN, STEPWISE REFINEMENT, HIERARCHICAL OR MODULAR DECOMPOSITION;
   - PROGRAM DESIGN LANGUAGES, OR PSEUDO-CODE, OR PIDGIN ENGLISH (THESE ARE NOT PROGRAMMING LANGUAGES);
   - PROJECT WORKBOOK, OR UNIT DEVELOPMENT FOLDER;
   - FORMAL REVIEW, PEER REVIEW, ETC.;
   - DOCUMENTATION—E.G., HIPO

   (MANY OF THESE PRACTICES CARRY THROUGH TO OR HAVE THEIR COUNTERPART IN THE NEXT STAGE.)
4. PROGRAMMING:
   - VARIOUS LEVELS OF PROGRAMMING LANGUAGES;
   - STRUCTURED PROGRAMMING (OR CODING);
   - INTERNAL OR SELF DOCUMENTATION;
   - STRUCTURED WALK-THROUGH, OR CODE REVIEW;
   - DEBUGGING.
5. VERIFICATION AND TESTING:
   - CORRECTNESS (TO SPECIFICATIONS);
   - RELIABILITY (IN USER ENVIRONMENT);
   - TOP-DOWN TESTING;
   - AUTOMATED AIDS.
6. PERFORMANCE:[25]
   - EFFICIENCY, TIME, MEMORY SIZE
   - QUALITY
   - ADAPTABILITY, FLEXIBILITY, PORTABILITY.
7. OPERATION AND MAINTENANCE:
   - DESIGN OR PROGRAMMING ERROR CORRECTION;
   - MINOR UPDATING OR ENHANCEMENT
8. CONFIGURATION MANAGEMENT.
   - BASELINING AT VARIOUS STAGES AGAINST FURTHER CHANGES.
   - PROBLEM REPORTING.
   - CHANGE CONTROL BOARD AND PROCEDURES

Modern programming practices. Although these practices have been widely discussed in the literature, Holton[7] had to drop one-third of the companies he started out to survey because they had not even considered using them. At panel discussions, managers complain from the podium that programmers resist the new ideas, while programmers retort from the floor that managers are too far from the nitty gritty to understand the problem. One can only conclude that some people need further information about modern programming practices.

Table 2 defines the core practices briefly (the main purpose of this article, after all, is to establish the need for and the value of these techniques, not to elucidate them in detail). For that purpose the practices are referenced. In addition, Freeman and Wasserman have annotated 10 full-length books on various aspects of software design in their tutorial.[26] They also reprinted 24 of the more useful papers.

Several books are too new to be listed by Freeman and Wasserman. Tausworthe's[27] volume, produced at the Jet Propulsion Laboratory under a NASA contract, covers the development stages of Table 1 and the programming practices of Table 2 (and others) in a systematic manner. Hughes and Michtom[28] cover a somewhat narrower field (top-down development, structured programming, stepwise refinement, and structured walk-throughs) but

also treat structured programming in three common languages: Cobol, Fortran, and PL/I.

The DOD view

The Department of Defense has taken steps to encourage more effective software engir ering. For example, a 1976 directive provided guidelines intended to create a discipline of software engineering.[41] Earlier, in March 1974, the Army Computer Systems Command and the Air Force Rome Air Development Center jointly sponsored a contract with IBM Federal Systems Division to document everything that was known about structured programming technology. This effort resulted in a 15-volume series.[42]

However, it is difficult for DOD to dictate just how contractors will go through the design and development process. It does not attempt to do so for hardware. Its proper sphere is to establish endproduct requirements.

Fortunately, a number of major software contractors have responded to the challenge of modern programming practices and have created disciplined development processes, taking advantage of the new practices in various ways. Not many people in the defense community disagree on the general value of those processes today.

Problems remain. There is still the problem of getting contractors to progress from where they are to the software engineering system that each one thinks is best or, perhaps, to the system he

Table 2. Some definitions of modern programming practices.

CHIEF PROGRAMMER TEAMS[1,2,4]

To reduce the coordination problems on large projects, Mills suggested the use of teams, each headed by an especially capable chief programmer, who would be assisted by specialists in each function needed to support him. Brooks characterized it as the "surgical team." Coordination would be the province of the chief programmer alone, thus reducing the number of minds involved in project communication by a factor of five or six.

DEVELOPMENT SUPPORT LIBRARIAN

One of the team members is the librarian. He is responsible for the programming-product library, containing both machine- and human-readable material. This function helps to transform programming "from a private art to public practice."

TOP-DOWN DEVELOPMENT[5]

Once requirements are firmed up, the development process decomposes the proposed system into a series of levels in a hierarchy, beginning at the top and working down. The highest level is then designed, coded, and subsequently tested first, using *stubs* with dummy code to stand in for lower-level units that are involved, and so on.

MODULAR DECOMPOSITION[29,30,31]

To isolate the entire system into independent partitions, each module is constructed to work with others on control signals and data transfers, but to be uninvolved in the detailed internal structure of other modules. With inter-module interfaces carefully specified, the relatively independent modules become easier to code, test, and later change than more dependent modules.

STRUCTURED DESIGN[32]

Structured design is a set of techniques for reducing the complexity of large new programs by dividing them into independent modules. Working with separate pieces permits the programmer to code, debug, test, and modify a functional module with minimal effect on other modules of the entire system. Concentrating effort in this way enhances efficiency and quality and reduces bugs. Moreover, to the extent that the independent modules are portable, further systems can be developed with less need for new code.

PROGRAM DESIGN LANGUAGE[33,34,35]

Intended to be comparable to the blueprint in hardware programming, design languages strive to communicate the concept of the software design in all necessary detail, using a formal or structured version of English, sometimes called pidgin English or pseudo code.

PROJECT WORKBOOK[1]

Design efforts inevitably produce much written material—memoranda, explanations, reports. The trick is to capture and organize it so as to be sure it reaches all who need to know and is available for use later.

HIPO—HIERARCHY/INPUT-PROCESS-OUTPUT[36,37]

This part documentation, part analytical technique consists of hierarchy charts and the corresponding input-process-output charts. The hierarchy chart is a set of blocks, similar to an organization chart, showing each function and its division into subfunctions. For each function or subfunction, an input-process-output chart, roughly similar to the block diagram in logic design, shows the inputs and outputs and the processes joining them. If the HIPO charts themselves are arranged in a hierarchy, the techniques can be used to graphically document top-down design or structured design.

STRUCTURED PROGRAMMING[38,39]

To enhance readability and maintainability, a program is structured so that the logic flow proceeds from beginning to end without arbitrary branching. This approach is based on the theorem that any program with one entry and one exit can be constructed from only three control structures: SEQUENCE, IF-THEN-ELSE, and DO-WHILE. It is analogous to the formation of complex logic functions from only AND and NOT building blocks.

STRUCTURED WALK-THROUGH[40]

The structured walk-through, sometimes called peer review, is the old design review with significant modifications:

- The reviewee takes the initiative to plan and run the session; management does not attend, thus encouraging a non-defensive atmosphere.
- The new design and programming practices provide an understandable structure the reviewers can readily "walk through."
- The emphasis in the session is on *error detection*; the reviewee is solely responsible for *correction*, which he does later.

thinks the government would like him to use. And assuming modern programming practices as a group *are* effective, there is the further problem of determining just which practices are most suitable in particular applications. And inasmuch as modern programming practices cost budget money, there is the problem of finding out which ones are most cost-effective. There is the problem—is software engineering sufficiently mature to be ready for standardization? Many observers think there is still much to learn and organizations should continue for some time yet to experiment with new techniques.

Software analysis. One clear need is for more data on the software development process. To meet this need, the Air Force is setting up through the Rome Air Development Center a software-analysis center.[4] It is to gather program development statistics in order to determine the factors that influence the productivity of programmers and the cost of software development and maintenance. A second task is to understand the nature, causes, and effects of software failures. To accomplish these purposes, it appears that some definitions will have to be hammered out to make data from various organizations comparable.

Correlations. One way to establish the value of modern programming practices is to do correlation studies. Such studies attempt to correlate one or more programming practices against various measures of the effectiveness of the software development process, such as cost, errors, and programmer productivity. It would be interesting to know if some of the practices are more effective than others. In an attempt to answer this kind of question, the Rome Air Development Center commissioned a number of studies, three of which were reported to COMPCON Fall 77.

Cost relation. The first study, by Rachel Black of Boeing, measured the effects of modern programming practices on software development costs for five projects of Boeing Computer Services.[4] However, two of the projects were very small, so in view of the belief that productivity on small projects is highly dependent on the individuals involved, the results from these two projects have been eliminated from the data summarized here.

The key comparison was made between man-months for the projects as forecasted by Boeing's traditional estimating procedure and manmonths actually incurred. The traditional procedures, said to predict costs within ± 15 percent, had not assumed the use of modern programming practices. However, the projects, as executed, did employ a variety of new practices.

The general effect of using modern programming practices was positive, as shown in Figure 1. The average improvement of the three projects, weighted for size, was 73 percent. This figure represents the difference between the forecast manmonths and the actual manmonths, divided by the forecast manmonths.

Boeing found modern programming practices reduced actual costs over forecast costs by 73 percent.

The practices employed on the three projects are listed in Table 3 in the order of Black's estimate of their impact on cost. It does appear that some practices are more effective than others on the cost dimension.

These three projects, as well as the two smaller projects previously discarded, also utilized top-down design techniques, structured programming, and programming support tools and libraries. Because the two smaller projects had shown little improvement using these techniques, Black had eliminated them as contributors to the benefits the three larger projects achieved—probably unwisely. However, her interviews with project personnel

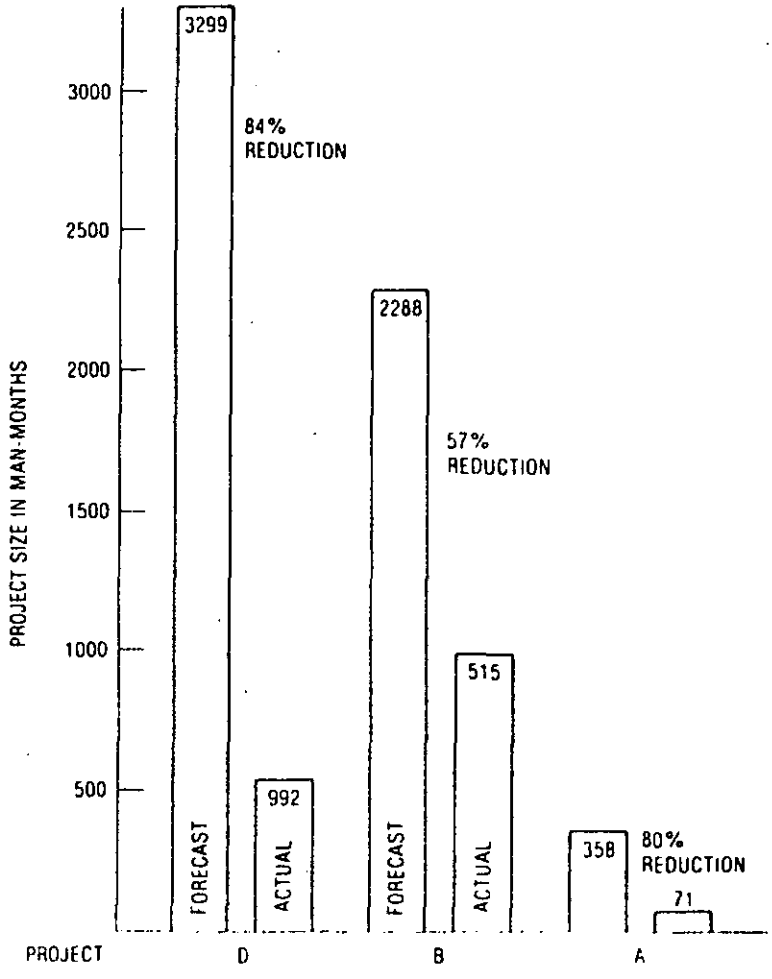


Figure 1. Improvement of actual man-months over forecast man-months on three Boeing software projects as a result of adopting modern programming practices.

Table 3. Modern programming practices used on three Boeing projects (listed in order of impact on cost).

PRACTICES ASSOCIATED WITH PROGRAM MANAGERS' MANAGEMENT METHODS (USED BY ALL THREE PROJECTS)
WRITTEN TASK ASSIGNMENTS
FORMAL CUSTOMER REVIEWS
EARLY DOCUMENTATION
UNIT DEVELOPMENT FOLDERS (PROGRAMMERS' WORKBOOK)
DESIGN REVIEW PRIOR TO CODING (STRUCTURED WALKTHROUGH)
FORMAL TESTING (USED BY TWO LARGER PROJECTS ONLY)
CONSTRUCTION PLANNING (INCLUDING PLANNING FOR INTEGRATION AND FUNCTIONAL TESTING)
CODE VERIFICATION (PEER CODE REVIEWS)
ACCEPTANCE TESTING (FORMAL DEMONSTRATION)
FUNCTIONAL TESTING
CONTROL OF TEST MATERIALS
CONFIGURATION MANAGEMENT (USED BY TWO LARGER PROJECTS ONLY)
BASELINING
PROBLEM REPORTING
CHANGE CONTROL BOARD

revealed universal positive feelings about the effectiveness of top-down design techniques:

> Probably the improved customer/project and intra-project communication cited by our interviewees as a consequence of top-down design has its primary benefit in establishing a sound basis for formal testing. In particular, two of the program managers interviewed felt that their testing activities proceeded more smoothly, and that fewer errors were discovered during testing as a direct result of the top-down design techniques they employed. For lack of supporting data in this study, we can only conclude that the cost benefits of top-down design may not be evident until a software system is in operation and maintenance.

On the other hand, the absence of a positive correlation between modern programming practices and cost on the two smaller projects was more likely the result of the idiosyncracies of small projects than it was of the employment of top-down design practices. The positive correlations found on the three larger projects, which also used top-down design and the other techniques, is more persuasive.

Black also attempted to measure the impact of modern programming practices on the percentage distribution of costs over the four main project phases used by Boeing:

- definition (essentially requirements and specifications)
- design
- construction (includes coding, integration, and functional testing)
- demonstration (acceptance testing and installation).

The results, summarized for the three projects in Figure 2, showed that costs were shifted into earlier stages by the use of modern programming practices. Unfortunately for the study design, Boeing included both coding and testing in the third stage, construction. Consequently, the shift to the left was less marked than it might have been if test costs were broken out separately.

Programming standards. In the first of two related studies, Brown[4] interviewed a cross section of management and performer personnel on TRW's very large ballistic-missile-defense programming project, as well as key staff personnel outside the project. This survey covered the impact of 18 detailed programming standards (e.g., structured coding) on 30 characteristics of software or the development process (e.g., cost, schedule, or programmer productivity). That made a matrix of 540 relationships, each of which was categorized over some nine levels of combined influence and assertion strength from very strong positive to very strong negative. However, 43 percent of the 540 intersections were labeled indifferent or inconclusive, implying either that there really was very little relationship between these variables or that the interviewees were ignorant of them.

As a brief summary of some of Brown's findings, four of the more significant rows (representing various coding standards) and seven of the more meaningful columns (representing various software characteristics) have been brought together in Table 4. Brown's ratings (strong positive, for example, means strong assertion, positive influence) were converted to weighted integers (strong positive = +3), both for simplicity and to permit the values to be summed across the rows. Thus, routine size or modularity is related to the seven software characteristics to the degree of 57 percent of the maximum positive rating. It is related to all 30 software characteristics (not shown in Table 4) only to the degree of 28 percent. This drop is not suprising, since the seven columms, with one exception, were selected to show the stronger relationships. Structured coding, which was not strongly correlated, was included as a matter of interest, since it is one of the core practices.

Structured coding evoked strong feelings on the part of the interviewees. Overall it had 18 positive ratings (average +1.7) and seven negative ratings (average −2.86). As Brown pointed out, this was seven out of a total of only 17 negative ratings in the whole survey. He felt that it "indicates a relatively dim view of structured coding on the part of (project) personnel."

"However," he went on, "there are some good reasons for this to be the case. First, the standard was not explicitly defined and enforced until almost two years after [the project] began, and the requirement to restructure existing code was felt to be counterproductive. Moreover, writing structured code in standard Fortran is awkward and introduced some inefficiencies in core and execution time making it difficult to satisfy demanding requirements levied on the real-time software. Finally, [the project] has not yet reached the phase during

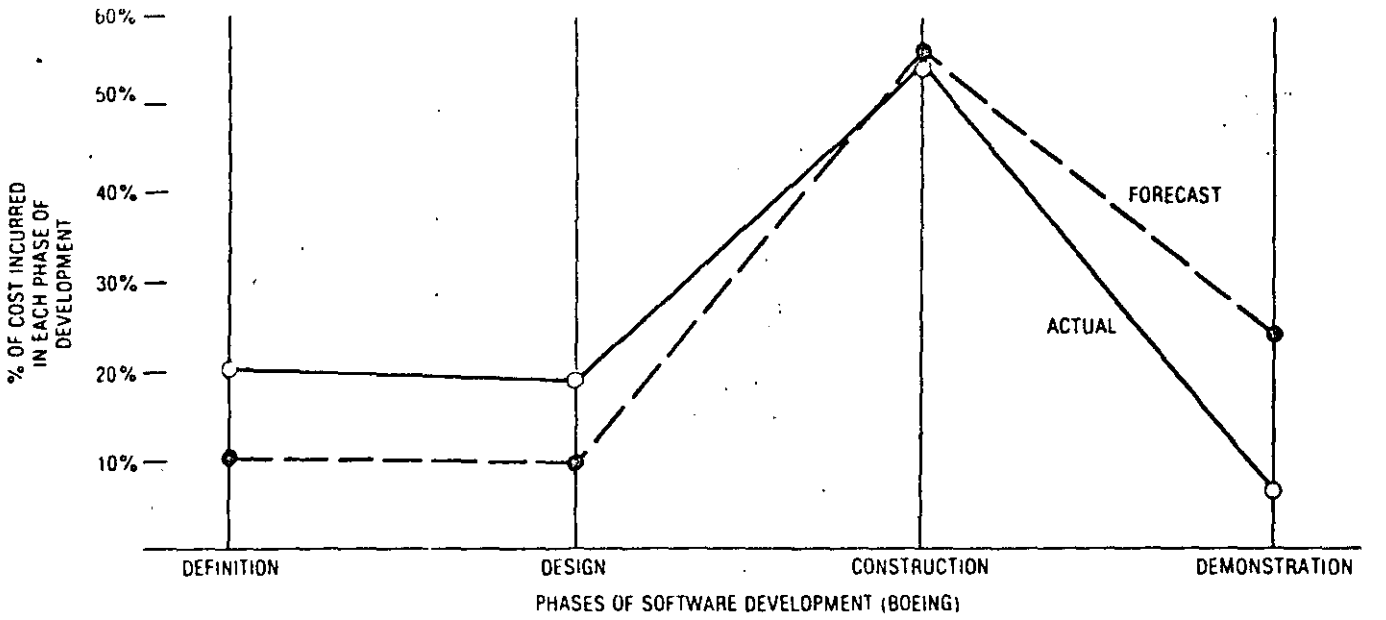


Figure 2. Forecast cost distribution shifts to earlier phases when modern programming practices are employed (average of three projects).

Table 4. Relationship of selected programming standards and software characteristics (first TRW study).

PROGRAMMING STANDARDS	SOFTWARE CHARACTERISTICS								
	CODE AUDITABILITY	CODE UNDERSTAND-ABILITY/ READABILITY	CODE MAINTAIN-ABILITY/ USABILITY	TESTABILITY	OPERATIONAL RELIABILITY	CODING ERROR FREQUENCY	PROGRAMMER PRODUCTIVITY	TOTALS	
								FOR 7	FOR 30
ROUTINE SIZE (MODULARITY)	+3	+3	+3	+3	+2	+2	0	+16 +57%	+34 +28%
IN-LINE COMMENTARY	+3	+4	+4	+1	+1	+2	0	+15 +54%	+37 +31%
STRUCTURED CODING	+3	+3	0	+2	+2	0	−3	+7 +25%	+11 +9%
NAMING CONVENTION	+2	+3	+2	+1	+1	+2	+2	+13 +46%	+38 +32%

which the major benefits of structured programming (i.e., improved readability and maintainability) were expected to be reaped."

On the relationships between coding standards and the characteristics of cost, schedule, and programmer productivity, the study was two-thirds inconclusive and one-third negative. That is, about two-thirds of the intersections for these categories were labeled inconclusive or indifferent. The remaining one-third of the intersections were mostly negative, showing the following net percentages:

Cost	Schedule	Programmer Productivity
−33 percent	−25 percent	+7 percent

Perhaps the reasons for the poor showing on these characteristics are similar to those quoted above on structured coding.

In summation, the overall weighted ratings for 18 programming standards against 30 software characteristics were 59 percent positive, 36 percent indifferent or inconclusive, and 5 percent negative.

In addition, two hypotheses were tested. The first is that programming (i.e., documentation and coding) standards, if rigorously defined, systematically enforced, and supported by tools, help to make possible the production of software of higher than usual quality. To this proposition those interviewed agreed 85 percent, disagreed 4 percent, and

didn't know 11 percent. To the second hypothesis—i.e., that these standards help to make possible the production of software of lower than usual cost—those interviewed agreed only 28 percent, disagreed 50 percent, and didn't know 22 percent.

MPP impact. The second TRW evaluation survey generated a matrix of 11 modern programming practices against 12 software problems, making 132 intersections in all. In this study personnel were asked to respond on both an *actual* and a *theoretical* basis. The theoretical responses showed a higher degree of relationship than the actual by a weighted margin of better than 3 to 2. In addition, the "indifferent" or "inconclusive" intersections dropped from 40 to eight. Of the 40 indifferent or inconclusive intersections in the actual comparison, 26 were involved in cost or schedule overruns or inefficient use of resources, again suggesting the interviewees were currently dubious in these areas. On a theoretical basis, however, the indifferent and inconclusive intersections to these three software problems dropped to five, implying that with more experience, modern programming practices would have a more favorable impact on these problems. In fact, the ratings for these three problems increased from "inconclusive" to "medium" or "strong positive."

The average impact of each modern programming practice on the twelve software problems is graphed in Figure 3. The theoretical ratings are considerably better, this time by about 2 to 1 in weighted terms.

In Brown's own words, "There is strong agreement among (project) personnel as to the four MPP of greatest importance and impact and strong agreement on their relative ranking:

- Requirements analysis and validation
- Baselining of requirements specification
- Complete preliminary design
- Process design

There is strong agreement on the importance and positive impact of the next three most highly ranked MPP, but the relative ranking among them is less clear:

- Incremental development
- Unit development folders
- Software development tools

There is strong agreement on the importance and positive impact of the four lower ranked MPP, but the relative ranking among them is not at all clear:

- Independent testing
- Enforced programming standards
- Software configuration management
- Formal inspection of documentation and code."

As to the general MPP hypothesis, the query and the results were as follows: Rules governing software development, evaluation, and documenta-

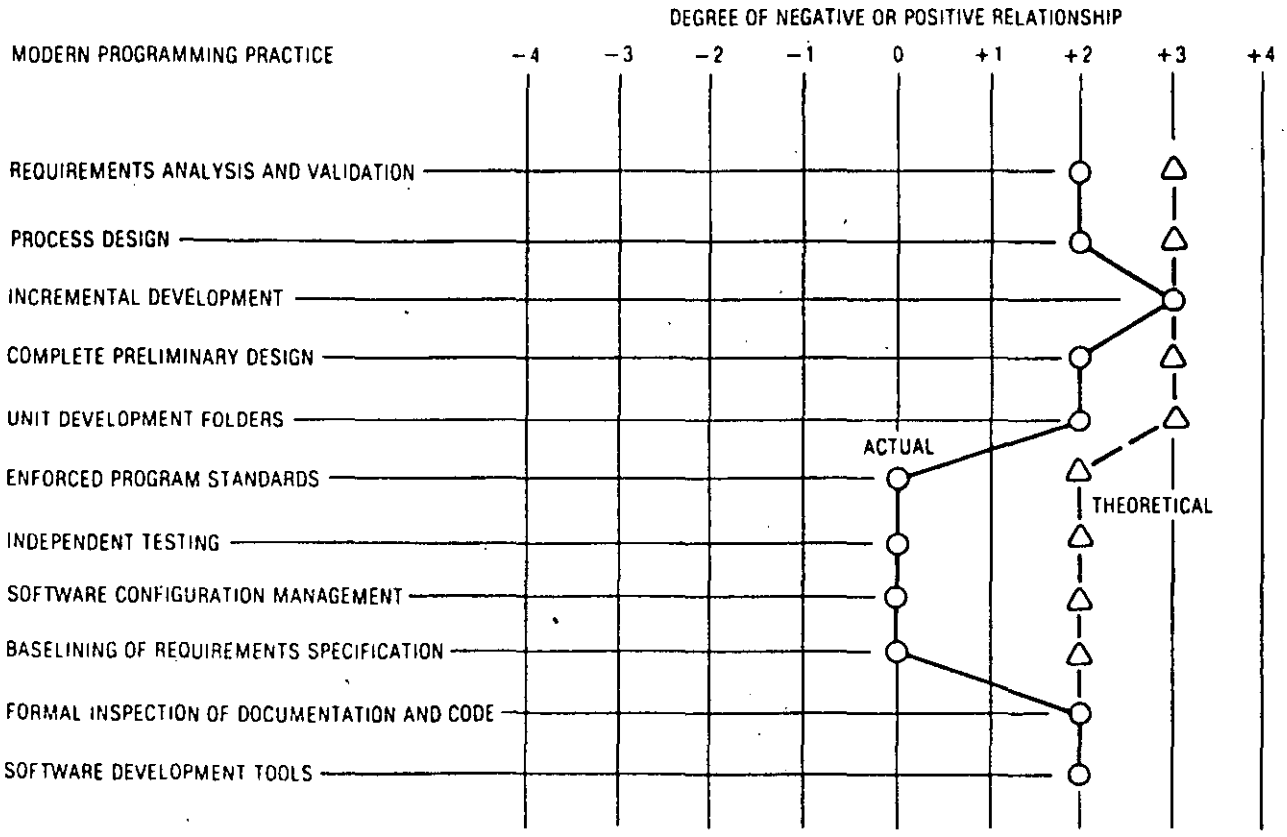


Figure 3. The correlationships between the modern programming practices and the summation of the twelve software problems is stronger in the theoretical case than in the actual case by about two to one.

tion, if rigorously defined and applied and supported by modern programming practices (techniques and tools), make possible the production—

	True	False	?
—of higher than usual quality software	85 percent	4 percent	11 percent
—of lower than usual life cycle cost	49 percent	15 percent	36 percent

Error relation. Operational reliability, documentation error frequency, and coding error frequency were three of the 30 software characteristics employed by Brown in his first study. The weighted positive rankings were as follows:

Operational reliability	31 percent
Documentation error frequency:	14 percent
Coding error frequency:	31 percent

In his second study, he found a more positive relationship between eleven modern programming practices and reliability, as follows:

Actual:	57 percent
Theoretical:	73 percent

Belford, Donahoo, and Heard[46] used software error as the only criterion in their study of 21 technical and five management software engineering techniques. In the absence of firm historical records, data was obtained by interviews with experienced personnel. The basic data was in two categories:

- the percentage of total errors estimated to occur in each of seven phases of the software life cycle; however, only the distribution of errors in the code and checkout phase was employed in this study;
- the probability of detection of each of the twelve error types by each of the 26 software engineering techniques.

This interview data was processed to obtain index numbers ranging from 0 to 75. These numbers indicate the effectiveness of each software engineering technique in reducing errors, as summarized in Table 5.

The authors regard their present method as the prototype of a still unproved process that "can reduce the identification and selection of optimum software engineering techniques to a straightforward, well-defined procedure."

To carry the procedure further, better historical data is needed, as well as information on other phases of the life cycle.

The broader community

So, members of the defense industry are making headway in the effort to embrace software development in engineering discipline. But what of the broader data processing community—the banks, insurance companies, and commercial establishments? Not long ago, Harlan Mills felt still able to say: "The idea of a rigorous rather than a heuristic program design method is new, and is still largely unknown in programming as practiced today."[48]

The IBM approach. Many of the improved programming technologies were developed by IBM scientists—Mills for one. The big computer maker has been a leader in their application, both within the company and to its customers. One task the industry faces, according to one IBM executive, is the need for a concerted effort by all elements—the manufacturers, the service bureaus, software houses, schools, and universities—to bring within their own training programs or curricula instruction in the improved programming technologies.

The increase—the delta—of programmers being added to the industry can then be addressed through the classical education channels. A bigger problem, with knowledge in the field moving so rapidly, is recycling established professionals—bringing their skills up to the latest levels of knowledge. In this area IBM has included the new programming practices in its educational offerings to customers. It is also working to instill a knowledge of these practices into its own very extensive programming population.

Table 5. Effectiveness of software engineering techniques in detecting errors (based on code and checkout phase only). Techniques are listed in order of effectiveness, with management techniques in parenthesis.

VERY EFFECTIVE (INDEX NUMBER 45 TO 75)
PROGRAM REVIEWS
CHIEF PROGRAMMER
CHIEF PROGRAMMER TEAM
BUILD LEADER (A COMPUTER SCIENCE CORP. TECHNIQUE)[47]
STRUCTURED WALK-THROUGHS
EFFECTIVE (28-33)
INDEPENDENT TEST AND EVALUATION
EXECUTION ANALYSIS
PROGRESSIVE TESTING
LESS EFFECTIVE (10-16)
PROGRAMMING TECHNIQUES
STRUCTURED PROGRAMMING
TOP-DOWN PROGRAMMING
VERIFICATION PROCEDURES
SUPPORT PROGRAMS
(TOP-DOWN DEVELOPMENT)
LEAST EFFECTIVE (.04-4)
(INSPECTION TEAMS)
TOP-DOWN DESIGN
STRUCTURED DESIGN
BUILDS (A CSC TECHNIQUE)
(THREADS MANAGEMENT SYSTEM) (A CSC TECHNIQUE)
PROGRAMMING DESIGN LANGUAGE
AUTOMATED NETWORK ANALYSIS
(PROJECT REVIEWS)
(MANAGEMENT DATA COLLECTION)
PROGRAMMING SUPPORT LIBRARY
SOFTWARE CONFIGURATION MANAGEMENT
PROGRAMMING LIBRARIANS

JPL Improves Software Development Process

At Caltech's Jet Propulsion Laboratory, about one-fifth of the budget and one-sixth of the manpower are devoted to some aspect of computing. All the software development organizations, including the Mission Control Center, Deep Space Network, spacecraft on-board computing, spacecraft testing, and the centralized computing facilities, have shown a keen interest in systematic design methods and, beginning early in the 1970's, in structured programming, top-down development, and other modern programming practices.

Management of JPL deliberately took a low-key approach to the introduction of these practices. At first there was a minimum of formal directives and a maximum of suggestion and example. The programming librarian idea was transformed by the Deep Space Net into a means of assisting programmers called the "programming secretariat." In the last several years the Deep Space Net has issued a set of rigorous software development standards, as listed in the accompanying box. In addition, a JPL staff member, Robert C. Tausworthe, prepared a 379-page monograph which presents these practices in a logical, tutorial form.[27]

JPL took a low-key approach to introducing the new techniques.

Flight projects and the Mission Control Center, which is shared by the projects, have not specified software development methods to the same level of detail as the Deep Space Net. The particular way in which these methods are implemented is adapted by each project organization to the tools available on the computer used for each task. However, the use of a program design language, structured programming, top-down development, and related methods are de facto standards in the project areas. Since much of the programming is scientific or engineering in content, an early task was the development of SFTRAN, Structured-Programming-To-Fortran Translator. Recently a preprocessor to facilitate structured programming in assembly language has been developed.

So, although the laboratory does not insist upon completely standardized organization-wide programming practices, the underlying principles are common to all sections. The differing detailed procedures endorsed by various program offices are the result of differences in the kind of applications. For example, the Deep Space Network develops software to be operated at overseas sites by personnel who are not JPL employes but are permanent residents of the countries in which the stations are located. Conversely, flight projects utilize the same personnel who develop the software to operate, maintain, and modify it during the mission. Moreover, flight software is much more subject to frequent modification in response to changes in the mission than is the software in the Deep Space Net.

Personnel. By 1975 an informal census of a cross section of programmers found about 50 percent definitely favorable to the new practices, about 40 percent rather neutral, and about 10 percent hostile to them. However, it was not only the modern programming practices that were important in forming these attitudes, but also the environment in which the work was done—the accessibility of text editing, interactive computing, and tools and facilities that enabled the programmer to get his job done with a minimum of running around and standing in line. They like the programming secretariat and support concepts. This environment made their lives easier and more productive.

Still there is great variability in the individual productivity of programmers, perhaps by a factor of 10 to 1, or even more sometimes. The techniques of structured design do not, of course, change dunces into brilliant programmers. They do make it possible to break up a large project in such a way that the more competent personnel can be assigned to the difficult early stages and

Software Standard Practices

DEEP SPACE NETWORK STANDARD PRACTICES

- 810-13 (AUGUST 15, 1975) SOFTWARE IMPLEMENTATION GUIDELINES AND PRACTICES
- 810-16 (DECEMBER 15, 1975) PREPARATION OF SOFTWARE REQUIREMENTS DOCUMENTS
- 810-17 (JULY 15, 1976) PREPARATION OF SOFTWARE DEFINITION DOCUMENTS
- 810-19 (MARCH 1, 1977) PREPARATION OF SOFTWARE SPECIFICATION DOCUMENTS
- 810-20 (FEBRUARY 1, 1977) PREPARATION OF SOFTWARE OPERATOR'S MANUALS
- 810-21 (NOVEMBER 15, 1976) PREPARATION OF SOFTWARE TEST AND TRANSFER DOCUMENTS

PROJECT DOCUMENTS

- PD618-58 (November 13, 1974) MARINER JUPITER/SATURN 1977 SOFTWARE MANAGEMENT PLAN (REVISED SEPTEMBER 7, 1976)
- PD622-35 (SEPTEMBER 7, 1977) SEASAT-A ADF PROGRAMMING STANDARDS

GENERAL SOFTWARE DOCUMENTS

- NASA SOFTWARE MANAGEMENT GUIDELINES
- JPL PUBLICATION 77-24 (JULY 1, 1977) SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE BY HENRY KLEINE
- INTEROFFICE COMPUTING MEMORANDUM NO 337 (JULY 31, 1973) SFTRAN USER GUIDE (STRUCTURED-PROGRAMMING-TO-FORTRAN TRANSLATOR) BY JOHN A FLYNN
- COMPUTING MEMORANDUM 432 (NOVEMBER 11, 1977) PROGRAMMING FOR PORTABILITY, BY FRED T. KROUGH, CHARLES L. LAWSON, AND MICHAEL R WARNER

the less creative people can be assigned to implementation tasks that have been fairly well structured. The structured methods also enable managers to identify the real incompetents sooner than older methods did.

Advantages. The payoff from the adoption of modern programming practices has come in terms of significantly improved schedule performance and manpower productivity. There is no doubt that these techniques have resulted in dramatic improvements in cost, quality, and schedule. For example, one of the programs for the Voyager project—several hundred thousand lines of high-level language—was recently completed with labor expenditures within budget and computer time at something like half of budget. The reduction in computer time implies that the programs were of better quality, because the programmers had to do much less debugging and testing. In the entire Voyager software development program, just finished, only one "tiger team" had to be set up. That was in the data records area which had to be done in assembly language because a higher-level language was not available.

A recent Deep Space Net project, one that had been budgeted for two years, came in two weeks over budget, but this overage had been detected a year ahead and appropriate readjustments had been made. In this case, too, debugging and testing took half the time originally budgeted. The reason: very few errors. In fact, there were 350 errors, including those in documentation, in 300,000 lines of code—a rate of only 20 percent of previous experience. Perhaps of equal importance to those only too accustomed to nerve-wracking "tiger team" efforts to meet flight dates, this time there was no tiger team. There were no hassles and no one aggravated his ulcers. It seemed that software management was coming of age.

Moreover, the full extent of the savings from the use of the new practices will only become apparent several years into the maintenance phase. This phase is clearly going to be much more efficient. The programs will be more reliable; they will be easier to change when necessary. These improvements follow from the experience in the implementation phase. When this phase is completed more quickly and with fewer errors, there is more confidence that the program itself is closer to full correctness.

JPL is finding a greater percentage of the overall software development effort being concentrated at the front end under the new practices. Coding is expected to remain a small fraction, even as the proportion of resources devoted to testing and maintenance declines. The 40-20-40 percent rule for the division of resources between definition and design, coding, and testing generally represents the laboratory's experience.

Still, there is an element of management in the problem. There is a need for disciplined thinking. Rather than focus on the fact that more money is being spent on software development, the big users have tended to emphasize fine tuning. Lately there does seem to be an increasing awareness by management of the dichotomy between the price/performance improvement of the hardware and the lack of that kind of improvement in software development. In the final analysis the users themselves have got to be willing to devote effort and money to improving their software development process.

Productivity relation. Although Brown found little relationship (+7 percent) between his 18 coding and documentation standards and programmer productivity, another approach showed a significantly high correlation. Walston and Felix[9] set up a software measurement project in the IBM Federal Systems Division in 1972. One of its purposes was to assess the effects of structured programming, then just beginning to be used, on the software development process. Data from 60 completed projects is now in their data base, representing projects ranging from 4000 to 467,000 source lines and 12 to 11,578 manmonths. Source lines were defined as the input to a language processor.

Using this data base, 68 variables were analyzed and 29 showed a high correlation with productivity.

The overall judgment is: modern programming practices are ready for use.

The data available enabled a comparison to be made between delivered source lines per manmonth and the percentage of code developed using each technique. These relationships, visualized in Figure 4, showed a rate of productivity improvement averaging 70 percent between "little use" of the new techniques and "much use."

Real-time design. The Advanced Technology Center of the Army's Ballistic Missile Defense has been engaged since 1972 in identifying and refining techniques to improve the software designer's ability to program large real-time processes in which the timing and order of events are critical. The result of its efforts, called process design methodology, embraces structured programming, top-down development, and other methods.

Gaulding and Lawson[10] analyzed a data base collected during experimental development work containing over 10,000 labor and computer-run entries. Two distinct advantages of the new design methodology over conventional approaches emerged:

- a majority of software errors was detected prior to the 50 percent project completion point, the opposite of conventional experience; and
- productivity was 3.3 equivalent machine instructions per manhour, said to be "considerably

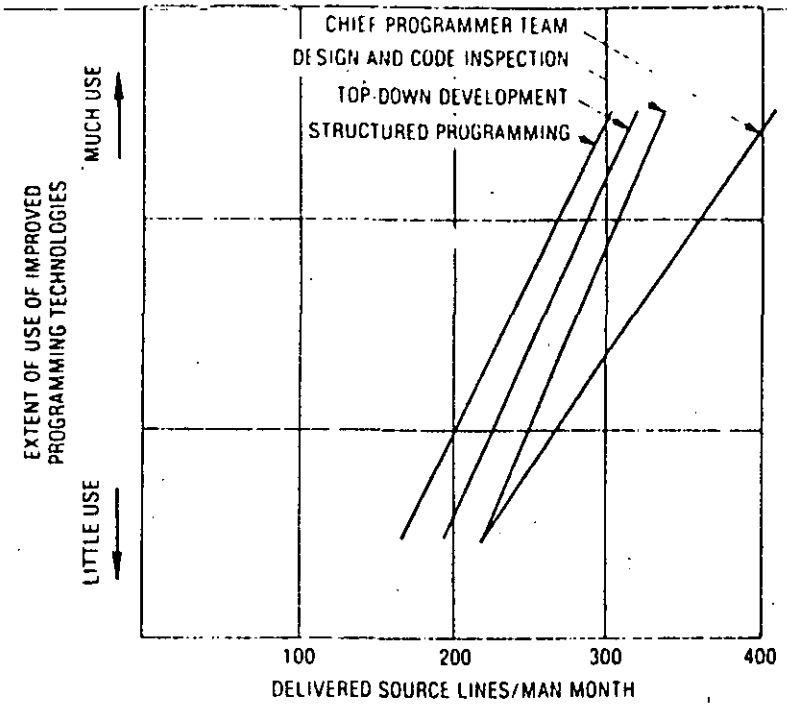


Figure 4. Production appears to increase dramatically with the use of improved programming technologies.

better" than the results on comparable real-time projects programmed by conventional methods.

Data management. Hsiao[51] reported the development of a highly secure data management system composed of 80 modules with 40K words of code. The total design and implementation were completed in 3 manyears with an elapsed time of less than 11 months. Using a chief programmer team, structured programming, and the other concepts of modern programming practices, he felt that the effort could not have been completed in such a short period of time without the use of these techniques. In addition, said Hsiao, extension of the system "has been greatly facilitated by the complete development-support library, clear system interface, high program modularity, and well-structured code."

Japanese findings. The Nippon Electric Company has modified its data base management system, programmed in Cobol, to take advantage of the concepts of structured programming.[52] This experience, plus some experiments in using the new techniques, has convinced them that the methods can be successfully applied to the forthcoming development of a very large on-line banking system. Their experiments indicate the following:

- number of steps programmed per working hour doubled;
- number of steps programmed per machine hour used tripled;
- number of database handling errors decreased 30 percent;
- other types of programming errors decreased 65 percent;
- average length of a bug find-and-fix cycle has been reduced to about one-third the conventional time.

These improvements were accompanied by small degradations in program performance.

Early assessment. Back in 1974, after only a few years of experience with the group of techniques that were then called structured programming, the IEEE Computer Society's Lake Arrowhead Workshop found that savings of "over 50 percent had been achieved, relative to previous performance on similar projects."[53] Companies providing data included IBM Federal Systems Division (40 percent average improvement over 20 projects), McDonnell-Douglas Automation Company (36 percent improvement on three projects; 5 percent on a fourth), and Hughes Aircraft Company (50 percent improvement on two real-time projects).

New practices judged effective

Software is in a predicament: it is too costly, too error-prone, too complex, too hard to maintain. This predicament may delay new applications of computers unless the effectiveness of software development can be substantially improved.

Modern programming practices are not the only way, of course. Higher-level programming languages, improved life cycle planning and management,* automated development tools, better personnel selection and training, and, of course, more sophisticated management will undoubtedly help.

The studies brought together here indicate that software effectiveness can be achieved. Where the results have been reduced to numbers, they ranged from about 25 percent to about 75 percent on such factors as cost, error reduction, and productivity.

Too much weight should not be placed on any one number. Studies of this kind are difficult to make and are usually not comparable from one organization to another, because definitions of terms vary. Moreover, averaging various amounts of knowledge—or degrees of ignorance—from interviews does not necessarily provide precise numbers. And summarizing detailed results, as has been done in this article, almost inevitably obscures the nuances of the original papers.

Rather, it is the overall judgment that is impressive: Whether from formal studies or the impressions of experienced executives, it seems clear that modern programming practices are effective in improving the processes of software development. They are ready for use. ■

*An article on life cycle planning is scheduled for the second quarter of 1978.

Acknowledgments

Interviews with the following experienced observers of software development helped guide me through the literature and practices of this field:

Walter R. Beam, Office of the Assistant Secretary of the Air Force for Research, Development, and Logistics

Joe M. Henson, IBM Data Processing Division

Robert Jirka, Michael R. Plesset, Edward C. Posner, and Michael R. Warner, all of Jet Propulsion Laboratory, California Institute of Technology.

References

1. Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays On Software Engineering*, Addison-Wesley Publishing Co., Reading, Massachusetts, 195 pp., 1974.

2. F. T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Sys. J.*, 11,1 (1972), pp. 56-73.

3. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972, 220 pp.

4. Harlan D. Mills, "Chief Programmer Teams, Principles, and Procedures," IBM Federal Systems Division Report FSC 71-5108, Gaithersburg, Maryland, 1971.

5. Harlan D. Mills, "Top Down Programming In Large Systems," in *Debugging Techniques in Large Systems*, R. Rustin (ed.) Prentice-Hall, Englewood Cliffs, New Jersey, 1971, pp. 41-55.

6. Robert M. McClure, "Software—The Next Five Years," *Digest of Papers, COMPCON Spring 76*, pp. 6-7.

7. John B. Holton, "Are The New Programming Techniques Being Used?" *Datamation*, July 1977, pp. 97-103.

8. Barry W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 48-59.*

9. Walter R. Beam, "Can Software Be More Like Hardware? Should It Be?" Keynote speech at COMPCON Fall 77, September 7, 1977 (not in *Digest of Papers*).

10. Edward W. Pullen and Robert G. Simko, "Our Changing Industry," *Datamation*, January 1977, pp. 49-55.

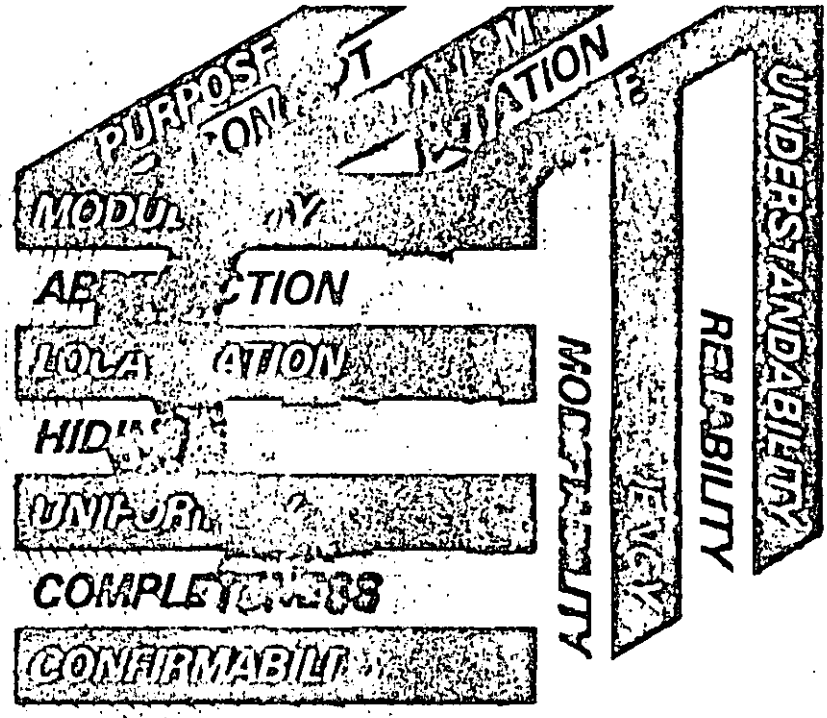
11. Joe M. Henson, "Computer Applications: Trends and Directions," Keynote speech at COMPCON Fall 77, September 7, 1977 (not in *Digest of Papers*).

12. "Business Software Makes For Problems, Hosler Charges," *Electronics*, November 24 1977, p. 14

13. J. L. Elshoff, "An Analysis of Some Commercial PL/1 Programs," *IEEE Trans. Software Engineering*, June 1976, pp. 113-120.

14. N. French, "Programmer Productivity Rising Too Slowly: Tanaka," *Computerworld*, 1977, 11-02? p. 1

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



SOFTWARE ENGINEERING: PROCESS, PRINCIPLES, AND GOALS

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

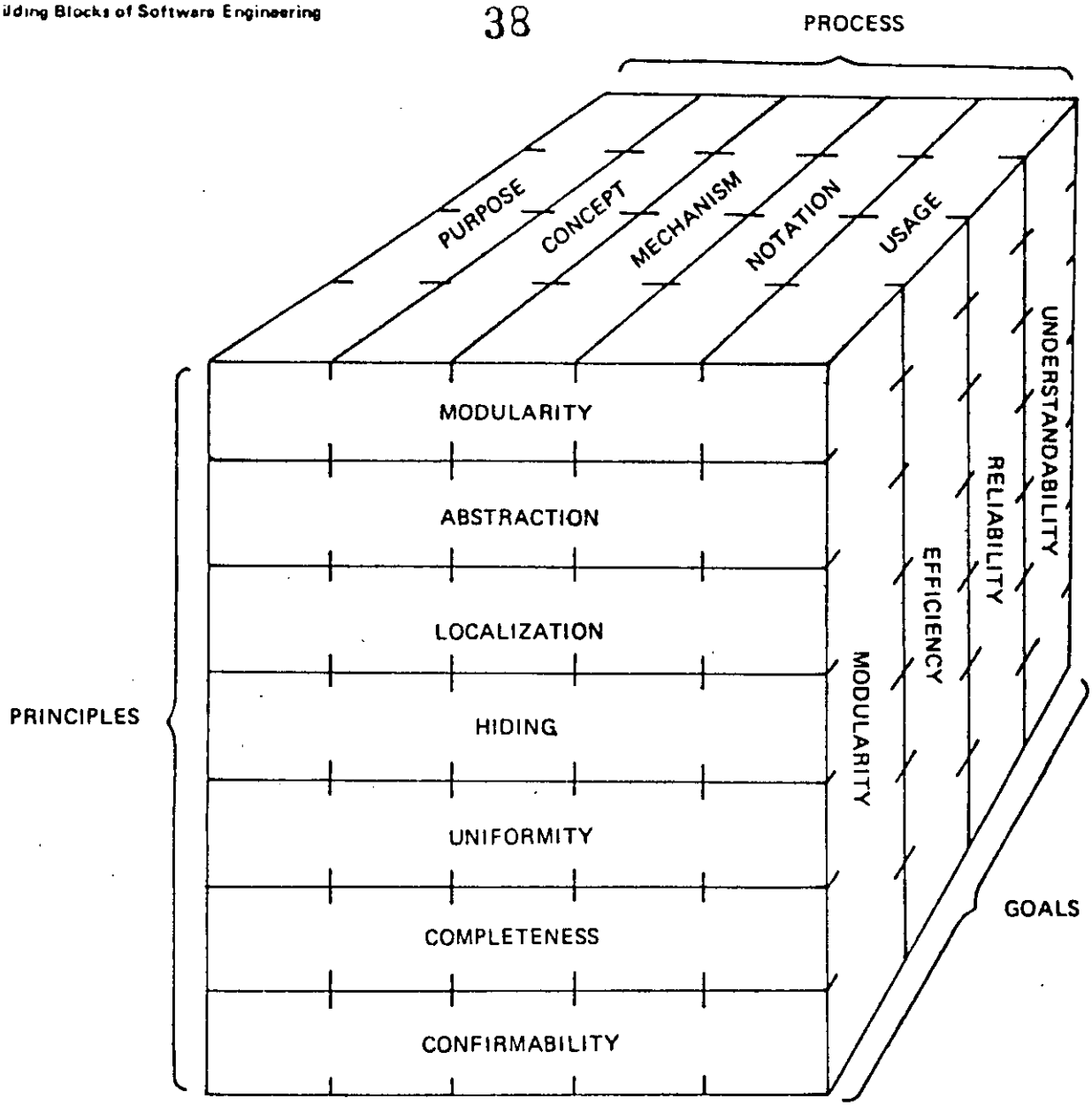
Introduction

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

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

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

Figure 1. The Building Blocks of Software Engineering



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

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

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

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

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

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

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

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

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

purpose – define the requirements for a system;

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

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

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

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

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

mechanism – the computer on which the software runs;

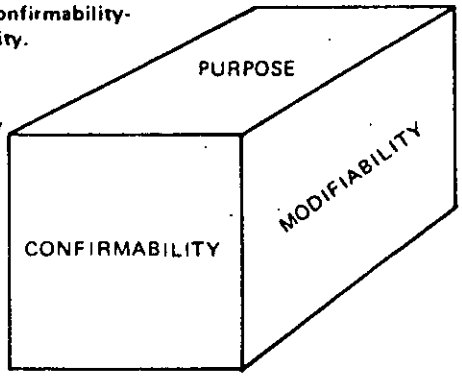
notation – the programming language in which the software will be written;

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

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

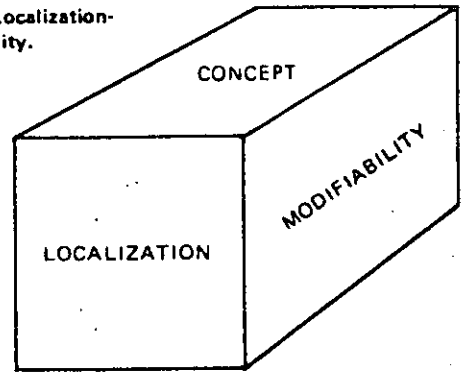
Given that part/whole hierarchical decomposition plays a pervasive role, the fundamental process interacts further with the various principles and goals of software engineering as depicted in Figure 1. Figure 1 is our framework for discussing the nature and issues of software engineering. Clearly an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

Figure 2. Purpose-Confirmability-Modifiability.



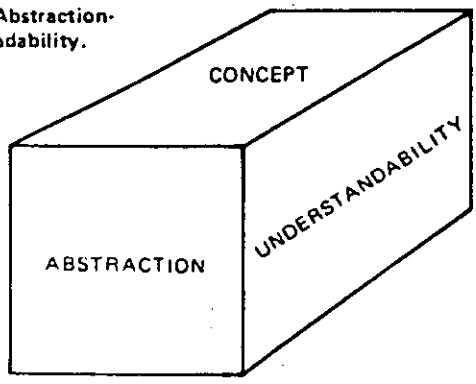
- The purpose of a design choice may be to structure a system so the effects of a change can be predicted with assurance.
- Confirmability applied to the process of defining purposes for modifiability demands that purposes be stated in a form that makes it possible to easily check if they have been achieved,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

Figure 3. Concept-Localization-Modifiability.



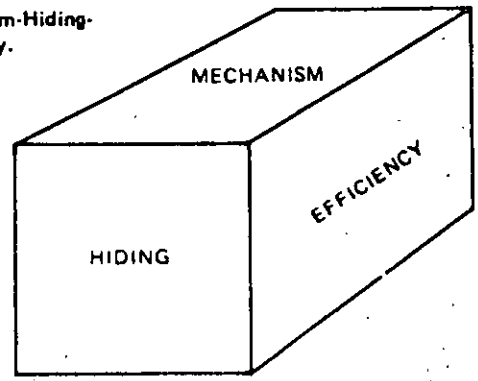
- Identifying one module for implementing a scheduling policy in an operating system is an example of the effect of the localization principle on design concepts to enhance modifiability, since localizing the policy rather than scattering policy decisions throughout a system makes it easier to change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

Figure 4. Concept-Abstraction-Understandability.



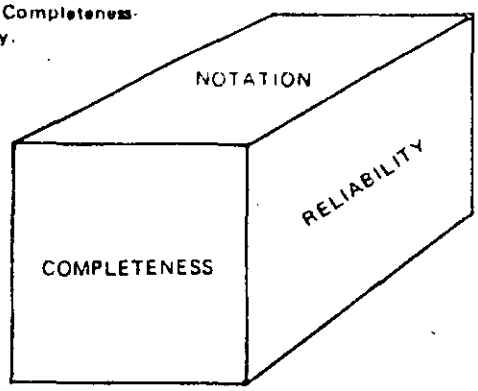
- The use of levels of abstraction[3,11,20] to define an understandable program or system of programs shows how abstraction can make designs more understandable.
- High-order languages represent a concept for improving the understandability of programs by abstracting from the details of computer instruction sets.

Figure 5. Mechanism-Hiding-Efficiency.



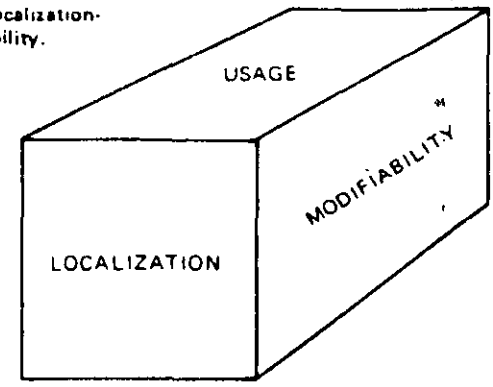
- Knowing that only certain subroutines have access to shared data (e.g., as in a function cluster[12]) may permit fewer checks to be made on the validity of stored values, and thereby increase efficiency.
- Forbidding users from directly accessing I/O devices permits an operating system to optimize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

Figure 6. Notation-Completeness-Reliability.



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

Figure 7. Usage-Localization-Modifiability.



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

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

Hierarchical Decomposition

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

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

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

The Goals of Software Engineering

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

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

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

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

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

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

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

The Principles of Software Engineering

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

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

> "[One objective of modular programming is to be able to convince oneself] of the correctness of a program module, independently of the context of its use in building larger units of software." (Reference 2, p. 129)
>
> "Modularity denotes the ability to combine arbitrary program modules into larger modules without knowledge of the construction of the modules." (Reference 9, p. 54)
>
> "Modularity is not . . . simply the arbitrary division of a large program into smaller parts or modules. The primary goal . . . should be to decompose the program in such a way that the modules are highly independent from one another." (Reference 13, p. 100)

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

> "A program is modular if it is written in many relatively independent parts or modules which have well-defined interfaces such that each module makes no assumptions about the operation of other modules except what is contained in the interface specifications." (Reference 4, p. 1)
>
> "Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules. . . . A definition of "good" modularity must emphasize the requirement that modules be as disjoint as possible." (Reference 11, p. 192)
>
> "A modular program is a program [having a hierarchical structure]. (Reference 1, p. 34)
>
> "Modular programming is the organizing of a complete program into a number of small units . . . where there is a set of rules which controls the characteristics of those units. (Cited in Reference 10, p. 29)

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

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

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

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

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

The principle of modularity can be further illustrated by considering modularity issues arising in deciding what part whole relationships should be considered in developing hierarchical decompositions.

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

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

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

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

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

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

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

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

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

Hiding Another principle familiar to many readers is *hiding*. Parnas,[15] for example, uses it as the major criterion for a decomposition into modules. It is related to the idea of "postponing binding decisions" in top-down problem-solving, although it is not the same. The purpose of hiding is similar to that of abstraction in that it requires making visible only those properties of a module needed to interface with other modules. But hiding differs from abstraction in that the purpose of hiding is *to make inaccessible* certain details that should not affect other parts of a system. Abstraction helps to identify details that should be hidden. Hiding is concerned with *defining and enforcing access constraints* that, without the hiding principle, would otherwise only be implicit in some purpose, concept, mechanism, notation, or usage description.

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

Uniformity Uniformity (the lack of inconsistencies and unnecessary differences) is also an important principle. When applied to notational matters, uniformity yields a notation free of confusing and perhaps costly inconsistencies. When also combined with the abstraction principle, uniformity implies a notation that permits arbitrary mechanization of the internal detailing of a mechanism—the notation does not constrain one's choice of implementation. And when the hiding principle is added, the result is a notation that does not merely permit several implementation choices, but also ensures that no unnecessary details of a specific implementation are revealed by the notation. For example, if a subroutine parameter is to be a stack, the representation of the stack should usually be hidden from the user of the subroutine. Thus, the user should be able to allocate storage for stacks and pass them to subroutines without having access to the individual components of a stack. A notation for representing such abstract data types is given in References 7 and 12. Another example is given in a recent paper on exception handling methods,[8] in which a uniform notation is proposed whose semantics can be realized using various traditional methods of handling error returns from subroutines.

In its essence, notation satisfying the uniformity and abstraction concepts has been called elsewhere[16] the

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

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

Confirmability Confirmability is a principle that directs attention to methods for finding out whether stated goals have been achieved. Applied to design issues, confirmability refers to the structuring of a system so it is readily tested. It must be possible to stimulate the constructed system in a controlled manner so its response can be evaluated for correctness. Applied to notational matters, confirmability means that a notation should require explicit specification of constraints that affect the correctness of a design or implementation (e.g., data declarations that specify range of values and units of value as well as mode of representation). Applied to the practice of software engineering, confirmability refers to the use of such methods as structured walk-throughs of designs, egoless programming,[19] and other methods that help to ensure that nothing has been overlooked.

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

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

An Example of the Framework's Utility

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

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

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

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

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

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

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

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

Notation:

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

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

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

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

Figure 8. Top Level Explication of the Subroutine Call Concept

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

It is useful to note the difference in the "Purpose" components of Figures 9 and 10. Although the goals inherited by decomposition from Figure 8 are the same–improve efficiency–note in Figure 10 that inline subroutines can improve efficiency in two ways: 1) by eliminating subroutine call overhead and 2) by performing certain computations at compile-time rather than at run-time, using actual parameter values. For example, if all parameters are constants, it may be possible to compute the value of the subroutine at compile time with consequent enormous savings in run-time efficiency. Wegbreit[18] explores this possibility and related ones in more detail.

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

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

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

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

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

Usage:

Figure 9. Description of the Closed Subroutine Concept

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

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

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

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

Usage:

Figure 10. Description of the Inline Subroutine Concept

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

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

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

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

Confirmability:

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

Uniformity:

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

Completeness:

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

Localization:

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

Hiding:

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

Abstraction:

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

Modularity:

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

Figure 11. Applying the Framework to Subroutine Call Notation Issues

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

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

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

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

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

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

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

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

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

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

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

Conclusion

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

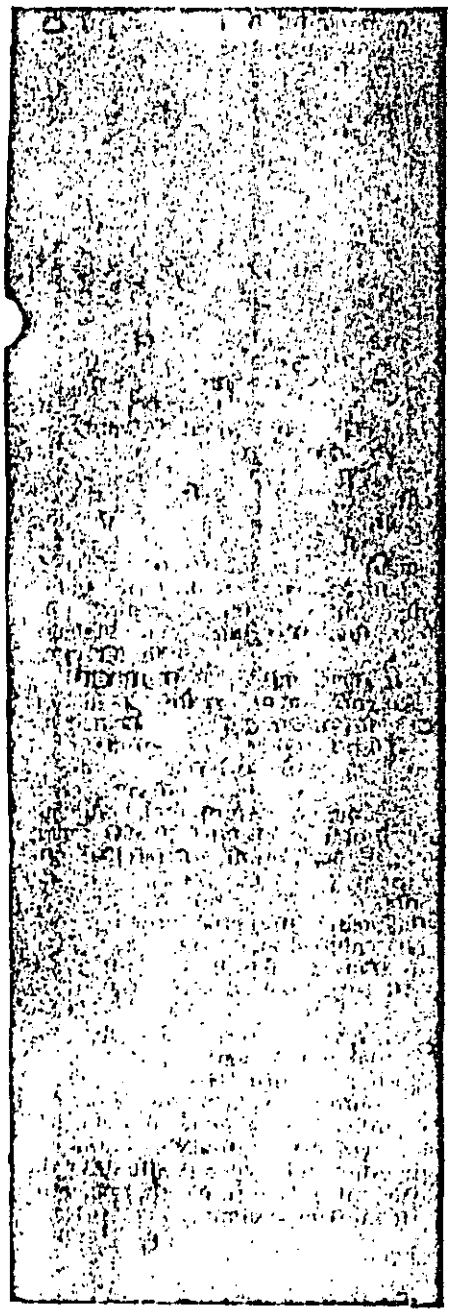
54

THE MYTHICAL MAN-MONTH

47

HOW DOES A PROJECT GET TO BE A YEAR LATE? . . . . . ONE DAY AT A TIME.

By Frederick P. Brooks, Jr.



NO SCENE FROM PREHISTORY is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and saber-toothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.

Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it.

More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this case of disaster so common?

First, our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption which is quite untrue, i.e., that all will go well.

Second, our estimating techniques fallaciously confuse effort with progress, hiding the assumption that men and months are interchangeable.

Third, because we are uncertain of our estimates, software managers often lack the courteous stubbornness required to make people wait for a good product.

Fourth, schedule progress is poorly monitored. Techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering.

Fifth, when schedule slippage is recognized, the natural (and traditional) response is to add manpower. Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline and thus begins a regenerative cycle which ends in disaster.

Schedule monitoring will be covered later. Let us now consider other aspects of the problem in more detail.

Optimism

All programmers are optimists. Perhaps this modern sorcery especially attracts those who believe in happy endings and fairy godmothers. Perhaps the hundreds of nitty frustrations drive away all but those who habitually focus on the end goal. Perhaps it is merely that computers are young, programmers are younger, and the young are always optimists. But however the selection process works, the result is indisputable: "This time it will surely run," or "I just found the last bug."

So the first false assumption that underlies the scheduling of systems programming is that *all will go well*, i.e., that *each task will take only as long as it "ought" to take.*

The pervasiveness of optimism among programmers deserves more than a flip analysis. Dorothy Sayers, in her excellent book, *The Mind of the*

Maker divides creative activity into three stages: the idea, the implementation, and the interaction. A book, then, or a computer, or a program comes into existence first as an ideal construct, built outside time and space but complete in the mind of the author. It is realized in time and space by pen, ink, and paper, or by wire, silicon, and ferrite. The creation is complete when someone reads the book, uses the computer or runs the program, thereby interacting with the mind of the maker.

This description, which Miss Sayers uses to illuminate not only human creative activity but also the Christian doctrine of the Trinity, will help us in our present task. For the human makers of things, the incompletenesses and inconsistencies of our ideas become clear only during implementation. Thus it is that writing, experimentation, "working out" are essential disciplines for the theoretician.

In many creative activities the medium of execution is intractable. Lumber splits; paints smear; electrical circuits ring. These physical limitations of the medium constrain the ideas that may be expressed, and they also create unexpected difficulties in the implementation.

Implementation, then, takes time and sweat both because of the physical media and because of the inadequacies of the underlying ideas. We tend to blame the physical media for most of our implementation difficulties; for the media are not "ours" in the way the ideas are, and our pride colors our judgment.

Computer programming, however, creates with an exceedingly tractable medium. The programmer builds from pure thought-stuff: concepts and very flexible representations thereof. Because the medium is tractable, we expect few difficulties in implementation; hence our pervasive optimism. Because our ideas are faulty, we have bugs; hence our optimism is unjustified.

In a single task, the assumption that all will go well has a probabilistic effect on the schedule. It might indeed go as planned, for there is a probability distribution for the delay that will be encountered, and "no delay" has a finite probability. A large programming effort, however, consists of many tasks, some chained end-to-end. The probability that each will go well becomes vanishingly small.

The mythical man-month

The second fallacious thought mode is expressed in the very unit of effort used in estimating and scheduling: the man-month. Cost does indeed vary as the product of the number of men and the number of months. Progress does not. *Hence the man-month as a unit for measuring the size of a job is a dangerous and deceptive myth.* It implies that men and months are interchangeable.

Men and months are interchangeable commodities only when a task can be partitioned among many workers *with no communication among them* (Fig. 1). This is true of reaping wheat or picking cotton; it is not even approximately true of systems programming.

When a task cannot be partitioned

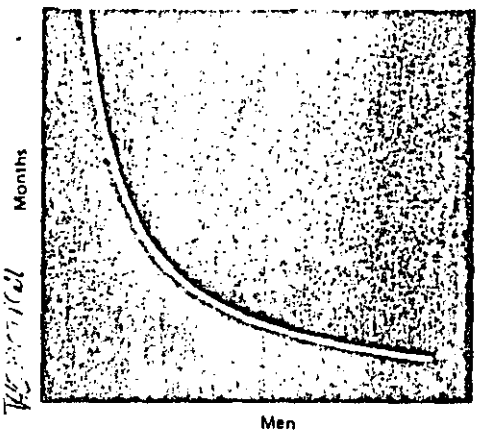


Fig. 1. The term "man-month" implies that if one man takes 10 months to do a job, 10 men can do it in one month. This may be true of picking cotton.

because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned. Many software tasks have this characteristic because of the sequential nature of debugging.

In tasks that can be partitioned but which require communication among the subtasks, the effort of communication must be added to the amount of work to be done. Therefore the best that can be done is somewhat poorer than an even trade of men for months (Fig. 2).

The added burden of communication is made up of two parts, training and intercommunication. Each worker must be trained in the technology, the goals of the effort, the overall strategy, and the plan of work. This training cannot be partitioned, so this part of the added effort varies linearly with the number of workers.

V. S. Vyssotsky of Bell Telephone Laboratories estimates that a large project can sustain a manpower buildup of 30% per year. More than that strains and even inhibits the evolution of the essential informal structure and its communication pathways. F. J. Corbató of MIT points out that a long project must anticipate a turnover of 20% per year, and new people must be both technically trained and integrated into the formal structure.

Intercommunication is worse. If each part of the task must be separately coordinated with each other part, the effort increases as $n(n-1)/2$. Three workers require three times as much pairwise intercommunication as two; four require six times as much as two. If, moreover, there need to be conferences among three, four, etc., workers to resolve things jointly, matters get worse yet. The added effort of communicating may fully counteract the division of the original task and bring us back to the situation of Fig. 3.

Since software construction is inherently a systems effort—an exercise in complex interrelationships—communication effort is great, and it quickly

Fig. 2. Even on tasks that can be nicely partitioned among people, the additional communication required adds to the total work, increasing the schedule.

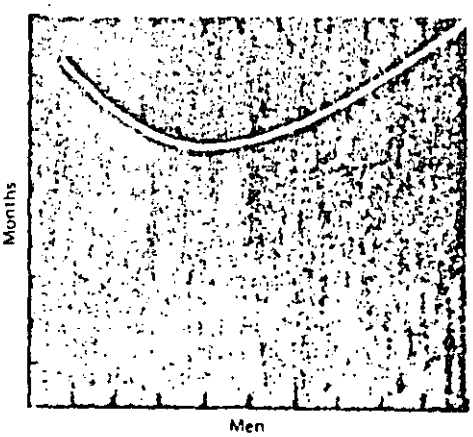


Fig. 3. Since software construction is complex, the communications overhead is great. Adding more men can lengthen, rather than shorten, the schedule.

dominates the decrease in individual task time brought about by partitioning. Adding more men then lengthens, not shortens, the schedule.

Systems test

No parts of the schedule are so thoroughly affected by sequential constraints as component debugging and system test. Furthermore, the time required depends on the number and subtlety of the errors encountered. Theoretically this number should be zero. Because of optimism, we usually expect the number of bugs to be smaller than it turns out to be. Therefore testing is usually the most mis-scheduled part of programming.

For some years I have been successfully using the following rule of thumb for scheduling a software task:

1/3 planning
1/6 coding
1/4 component test and early system test
1/4 system test, all components in hand.

This differs from conventional scheduling in several important ways:

1. The fraction devoted to planning is larger than normal. Even so, it is barely enough to produce a detailed and solid specification, and not enough to include research or exploration of totally new techniques.
2. The *half* of the schedule devoted to debugging of completed code is much larger than normal.
3. The part that is easy to estimate, i.e., coding, is given only one-sixth of the schedule.

In examining conventionally scheduled projects, I have found that few allowed one-half of the projected schedule for testing, but that most did indeed spend half of the actual schedule for that purpose. Many of these were on schedule until and except in system testing.

Failure to allow enough time for system test, in particular, is peculiarly disastrous. Since the delay comes at the end of the schedule, no one is aware of schedule trouble until almost the delivery date. Bad news, late and without warning, is unsettling to customers and to managers.

Furthermore, delay at this point has unusually severe financial, as well as psychological, repercussions. The project is fully staffed, and cost-per-day is maximum. More seriously, the software is to support other business effort (shipping of computers, operation of new facilities, etc.) and the secondary costs of delaying these are very high, for it is almost time for software shipment. Indeed, these secondary costs may far outweigh all others. It is therefore very important to allow enough system test time in the original schedule.

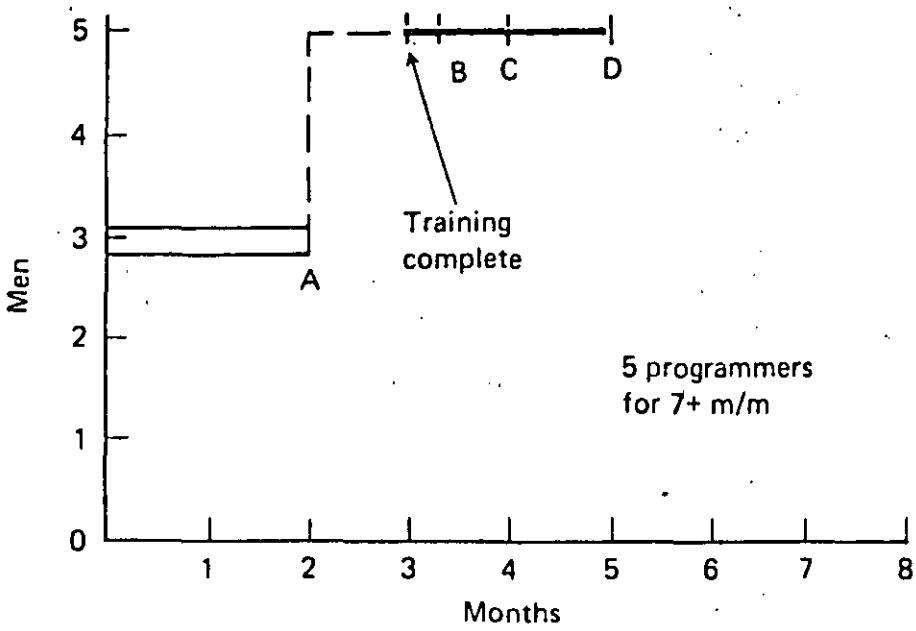


Fig. 4. Adding manpower to a project which is late may not help. In this case, suppose three men on a 12 man-month project were a month late. If it takes one of the three an extra month to train two new men, the project will be just as late as if no one was added.

Gutless estimating

Observe that for the programmer, as for the chef, the urgency of the patron may govern the scheduled completion of the task, but it cannot govern the actual completion. An omelette, promised in ten minutes, may appear to be progressing nicely. But when it has not set in ten minutes, the customer has two choices—wait or eat it raw. Software customers have had the same choices.

The cook has another choice: he can turn up the heat. The result is often an omelette nothing can save—burned in one part, raw in another.

Now I do not think software managers have less inherent courage and firmness than chefs, nor than other engineering managers. But false scheduling to match the patron's desired date is much more common in our discipline than elsewhere in engineering. It is very difficult to make a vigorous, plausible, and job-risking defense of an estimate that is derived by no quantitative method, supported by little data, and certified chiefly by the hunches of the managers.

Clearly two solutions are needed. We need to develop and publicize productivity figures, bug-incidence figures, estimating rules, and so on. The whole profession can only profit from sharing such data.

Until estimating is on a sounder basis, individual managers will need to stiffen their backbones, and defend their estimates with the assurance that their poor hunches are better than wish-derived estimates.

Regenerative disaster

What does one do when an essential software project is behind schedule? Add manpower, naturally. As Figs. 1 through 3 suggest, this may or may not help.

Let us consider an example. Suppose a task is estimated at 12 man-months and assigned to three men for four months, and that there are measurable mileposts A, B, C, D, which are scheduled to fall at the end of each month.

Now suppose the first milepost is not reached until two months have elapsed. What are the alternatives facing the manager?

1. Assume that the task must be done on time. Assume that only the first part of the task was misestimated. Then 9 man-months of effort remain, and two months, so 4½ men will be needed. Add 2 men to the 3 assigned.
2. Assume that the task must be done on time. Assume that the whole estimate was uniformly low. Then 18 man-months of effort remain, and two months, so 9 men will be needed. Add 6 men to the 3 assigned.
3. Reschedule. In this case, I like the advice given by an experienced hardware engineer, "Take no small slips." That is, allow enough time in the new schedule to ensure that the work can be carefully and

thoroughly done, and that rescheduling will not have to be done again

4. Trim the task. In practice this tends to happen anyway, once the team observes schedule slippage. Where the secondary costs of delay are very high, this is the only feasible action. The manager's only alternatives are to trim it formally and carefully, to reschedule, or to watch the task get silently trimmed by hasty design and incomplete testing.

In the first two cases, insisting that the unaltered task be completed in four months is disastrous. Consider the regenerative effects, for example, for the first alternative (Fig. 4 preceding page). The two new men, however competent and however quickly recruited, will require training in the task by one of the experienced men. If this takes a month, *3 man-months will have been devoted to work not in the original estimate.* Furthermore, the task, originally partitioned three ways, must be repartitioned into five parts, hence some work already done will be lost and system testing must be lengthened. So at the end of the third month, substantially more than 7 man-months of effort remain, and 5 trained people and one month are available. As Fig. 4 suggests, the product is just as late as if no one had been added.

To hope to get done in four months, considering only training time and not repartitioning and extra systems test, would require adding 4 men, not 2, at the end of the second month. To cover repartitioning and system test effects, one would have to add still other men. Now, however, one has at least a 7-man team, not a 3-man one; thus such aspects as team organization and task division are different in kind, not merely in degree.

Notice that by the end of the third month things look very black. The March 1 milestone has not been reached in spite of all the managerial effort. The temptation is very strong to repeat the cycle, adding yet more manpower. Therein lies madness.

The foregoing assumed that only the first milestone was misestimated. If on March 1 one makes the conservative assumption that the whole schedule was optimistic one wants to add 6 men just to the original task. Calculation of the training, repartitioning, system testing effects is left as an exercise for the reader. Without a doubt, the regenerative disaster will yield a poorer product later, than would rescheduling with the original three men, unaugmented.

Oversimplifying outrageously, we state Brooks' Law:

Adding manpower to a late software project makes it later.

This then is the demythologizing of the man-month. The number of months of a project depends upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using fewer men and more months. (The only risk is product obsolescence.) One cannot, however, get workable schedules using more men and fewer months. More software projects have gone awry for lack of calendar time than for all other causes combined.

Calling the shot

How long will a system programming job take? How much effort will be required? How does one estimate?

I have earlier suggested ratios that seem to apply to planning time, coding, component test, and system test. First, one must say that one does *not* estimate the entire task by estimating the coding portion only and then applying the ratios. The coding is only one-sixth or so of the problem, and errors in its estimate or in the ratios could lead to ridiculous results.

Second, one must say that data for building isolated small programs are not applicable to programming systems products. For a program averaging about 3,200 words, for example, Sackman, Erikson, and Grant report an average code-plus-debug time of about 178 hours for a single programmer, a figure which would extrapolate to give an annual productivity of 35,800 statements per year. A program half that size took less than one-fourth as long, and extrapolated productivity is almost 80,000 statements per year.[1]. Planning, documentation, testing, system integration, and training times must be added. The linear extrapolation of such spring figures is meaningless. Extrapolation of times for the hundred-yard dash shows that a man can run a mile in under three minutes.

Before dismissing them, however, let us note that these numbers, although not for strictly comparable problems, suggest that effort goes as a power of size *even* when no communication is involved except that of a man with his memories.

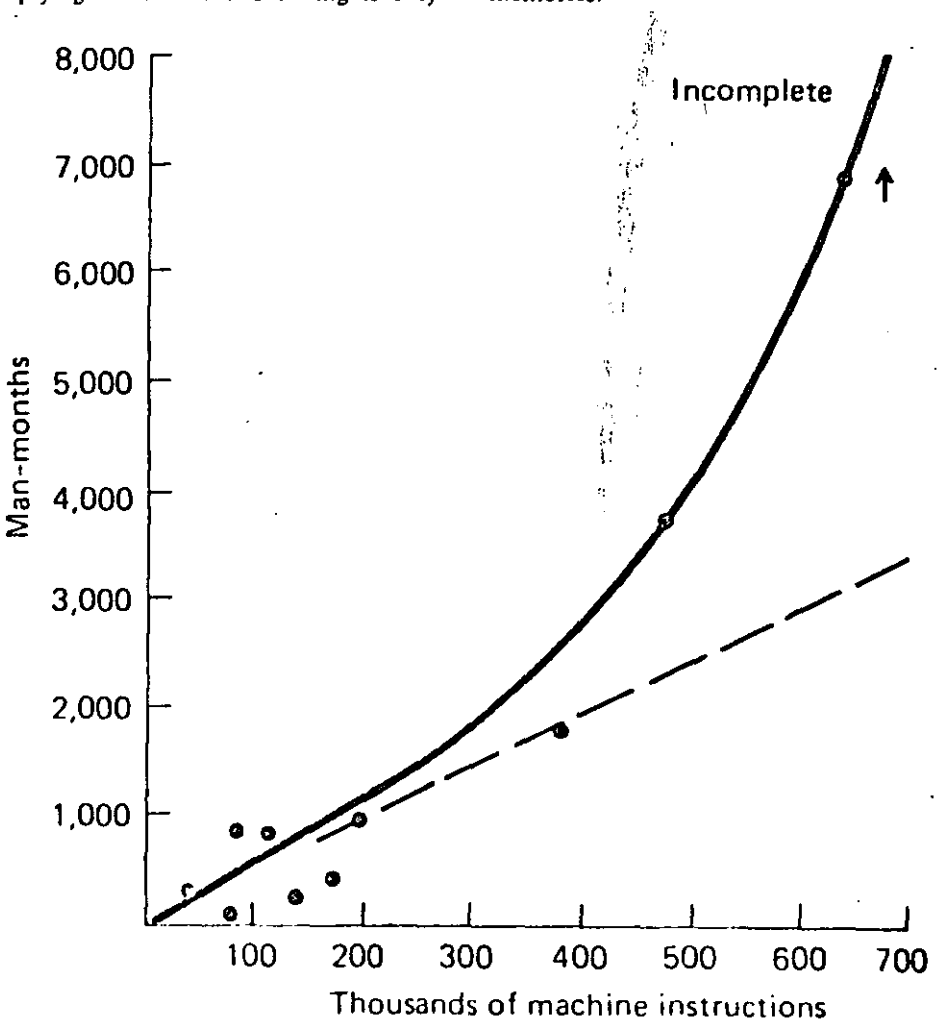


Fig. 5. As a project's complexity increases, the number of man-months required to complete it goes up exponentially.

Fig. 5 tells the sad story. It illustrates results reported from a study done by Nanus and Farr[2] at System Development Corp. This shows an exponent of 1.5; that is,

effort $\doteq$ (constant) $\times$ (number of instructions)$^{1.5}$

Another soc study reported by Weinwurm[3] also shows an exponent near 1.5.

A few studies on programmer productivity have been made, and several estimating techniques have been proposed. Morin has prepared a survey of the published data.[4] Here I shall give only a few items that seem especially illuminating.

Portman's data

Charles Portman, manager of ICL's Software Div., Computer Equipment Organization (Northwest) at Manchester, offers another useful personal insight.

He found his programming teams missing schedules by about one-half—each job was taking approximately twice as long as estimated. The estimates were very careful, done by experienced teams estimating man-hours for several hundred subtasks on a PERT chart. When the slippage pattern appeared, he asked them to keep careful daily logs of time usage. These showed that the estimating error could be entirely accounted for by the fact that his teams were only realizing 50% of the working week as actual programming and debugging time. Machine downtime, higher-priority short unrelated jobs, meetings, paperwork, company business, sickness, personal time, etc. accounted for the rest. In short, the estimates made an unrealistic assumption about the number of technical work hours per man-year. My own experience quite confirms his conclusion.

An unpublished 1964 study by E. F. Bardain shows programmers realizing only 27% productive time.[5]

	Prog. units	Number of programmers	Years	Man-years	Program words	Words/man-yr.
Operational	50	83	4	101	52,000	515
Maintenance	36	60	4	81	51,000	630
Compiler	13	9	2¼	17	38,000	2230
Translator (Data assembler)	15	13	2½	11	25,000	2270

Table 1. Data from Bell Labs indicates productivity differences between complex problems (the first two are basically control programs with many modules) and less complex ones. No one is certain how much of the difference is due to complexity, how much to the number of people involved.

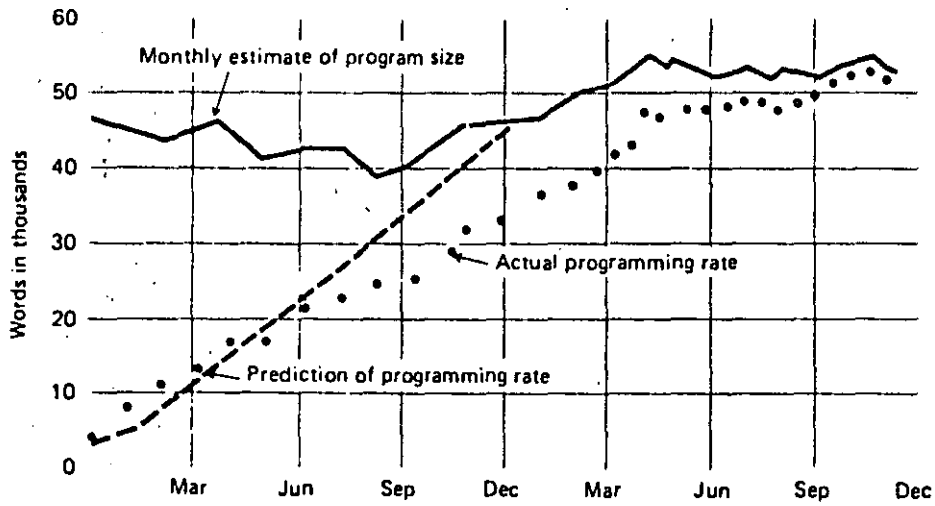


Fig. 6. Bell Labs' experience in predicting programming effort on one project.

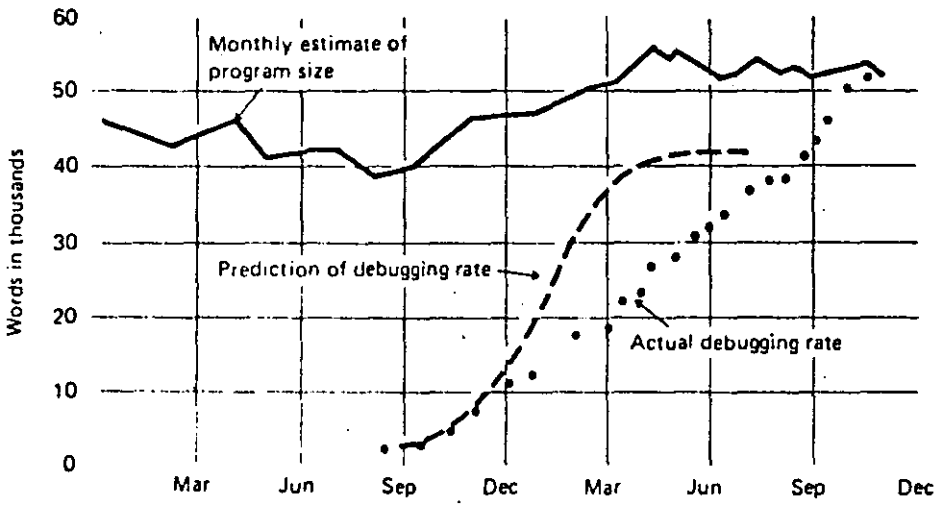


Fig. 7. Bell's predictions for debugging rates on a single project, contrasted with actual figures.

Aron's data

Joel Aron, manager of Systems Technology at IBM in Gaithersburg, Maryland, has studied programmer productivity when working on nine large systems (briefly, *large* means more than 25 programmers and 30,000 deliverable instructions). He divides such systems according to interactions among programmers (and system parts) and finds productivities as follows:

Very few interactions	10,000 instructions per man-year
Some interactions	5,000
Many interactions	1,500

The man-years do not include support and system test activities, only design and programming. When these figures are diluted by a factor of two to cover system test, they closely match Harr's data.

Harr's data

John Harr, manager of programming for the Bell Telephone Laboratories' Electronic Switching System, reported his and others' experience in a paper at the 1969 Spring Joint Computer Conference.[6] These data are shown in Table 1 and Figs. 6 and 7.

Of these, Fig. 6 is the most detailed and the most useful. The first two jobs are basically control programs; the second two are basically language translators. Productivity is stated in terms of debugged words per man-year. This includes programming, component test, and system test. It is not clear how much of the planning effort, or effort in machine support, writing, and the

like, is included

The productivities likewise fall into two classifications: those for control programs are about 600 words per man-year; those for translators are about 2,200 words per man-year. Note that all four programs are of similar size—the variation is in size of the work groups, length of time, and number of modules. Which is cause and which is effect? Did the control programs require more people because they were more complicated? Or did they require more modules and more man-months because they were assigned more people? Did they take longer because of the greater complexity, or because more people were assigned? One can't be sure. The control programs were surely more complex. These uncertainties aside, the numbers describe the real productivities achieved on a large system, using present-day programming techniques. As such they are a real contribution.

Figs. 6 and 7 show some interesting data on programming and debugging rates as compared to predicted rates.

OS/360 data

IBM OS/360 experience, while not available in the detail of Harr's data, confirms it. Productivities in range of 600-800 debugged instructions per man-year were experienced by control program groups. Productivities in the 2,000-3,000 debugged instructions per man-year were achieved by language translator groups. These include planning done by the group, coding component test, system test, and some support activities. They are comparable to Harr's data, so far as I can tell.

Aron's data, Harr's data, and the OS/360 data all confirm striking differences in productivity related to the complexity and difficulty of the task itself. My guideline in the morass of estimating complexity is that compilers are three times as bad as normal batch application programs, and operating systems are three times as bad as compilers.

Corbató's data

Both Harr's data and OS/360 data are for assembly language programming. Little data seem to have been published on system programming productivity using higher-level languages. Corbató of MIT's Project MAC reports, however, a mean productivity of 1,200 lines of debugged PL/I statements per man-year on the MULTICS system (between 1 and 2 million words).[7]

This number is very exciting. Like the other projects, MULTICS includes control programs and language translators. Like the others, it is producing a system programming product, tested and documented. The data seem to be comparable in terms of kind of effort included. And the productivity number is a good average between the control program and translator productivities of other projects.

But Corbató's number is *lines* per man-year, not *words!* Each statement in his system corresponds to about three-to-five words of handwritten code! This suggests two important conclusions:

- Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include.
- Programming productivity may be increased as much as five times when a suitable high-level language is used. To back up these conclusions, W. M. Taliaffero also reports a constant productivity of 2,400 statements/year in Assembler, FORTRAN, and COBOL.[8] E. A. Nelson has shown a 3-to-1 productivity improvement for high-level language, although his standard deviations are wide.[9]

Hatching a catastrophe

When one hears of disastrous schedule slippage in a project, he imagines that a series of major calamities must have befallen it. Usually, however, the disaster is due to termites, not tornadoes; and the schedule has slipped imperceptibly but inexorably. Indeed, major calamities are easier to handle; one responds with major force, radical reorganization, the invention of new approaches. The whole team rises to the occasion.

But the day-by-day slippage is harder to recognize, harder to prevent, harder to make up. Yesterday a key man was sick, and a meeting couldn't be held. Today the machines are all down, because lightning struck the building's power transformer. Tomorrow the disc routines won't start testing, because the first disc is a week late from the factory. Snow, jury duty, family problems, emergency meetings with customers, executive audits—the list goes on and on. Each one only postpones some activity by a half-day or a day. And the schedule slips, one day at a time.

How does one control a big project on a tight schedule? The first step is to *have* a schedule. Each of a list of events, called milestones, has a date. Picking the dates is an estimating problem, discussed already and crucially dependent on experience.

For picking the milestones there is only one relevant rule. Milestones must be concrete, specific, measurable events, defined with knife-edge sharpness. Coding, for a counterexample, is "90% finished" for half of the total coding time. Debugging is "99% complete" most of the time. "Planning complete" is an event one can proclaim almost at will.[10]

Concrete milestones, on the other hand, are 100% events. "Specifications signed by architects and implementers," "source coding 100% complete, keypunched, entered into disc library," "debugged version passes all test cases." These concrete milestones demark the vague phases of planning, coding, debugging.

It is more important that milestones be sharp-edged and unambiguous than that they be easily verifiable by the boss. Rarely will a man lie about mile-

> None love
> the bearer of bad news.
> *Sophocles*

stone progress, *if* the milestone is so sharp that he can't deceive himself. But if the milestone is fuzzy, the boss often understands a different report from that which the man gives. To supplement Sophocles, no one enjoys bearing bad news, either, so it gets softened without any real intent to deceive.

Two interesting studies of estimating behavior by government contractors on large-scale development projects show that:

1. Estimates of the length of an activity made and revised carefully every two weeks before the activity starts do not significantly change as the start time draws near, no matter how wrong they ultimately turn out to be.
2. *During* the activity, *over*estimates of duration come steadily down as the activity proceeds.
3. *Underestimates* do not change significantly during the activity until about three weeks before the scheduled completion.[11]

Sharp milestones are in fact a service to the team, and one they can properly expect from a manager. The fuzzy milestone is the harder burden to live with. It is in fact a millstone that grinds down morale, for it deceives one about lost time until it is irremediable. And chronic schedule slippage is a morale-killer.

"The other piece is late"

A schedule slips a day; so what? Who gets excited about a one-day slip? We can make it up later. And the other piece ours fits into is late anyway.

A baseball manager recognizes a nonphysical talent, *hustle*, as an essential gift of great players and great teams. It is the characteristic of running faster than necessary, moving sooner than necessary, trying harder than necessary. It is essential for great programming teams, too. Hustle provides the cushion, the reserve capacity, that enables a team to cope with routine mishaps, to anticipate and forfend minor calamities. The calculated response, the measured effort, are the wet blankets that dampen hustle. As we have seen, one *must* get excited about a one-day slip. Such are the elements of catastrophe.

But not all one-day slips are equally disastrous. So some calculation of response is necessary, though hustle be dampened. How does one tell which slips matter? There is no substitute for a PERT chart or a critical-path schedule. Such a network shows who waits for what. It shows who is on the critical path, where any slip moves the end date. It also shows how much an activity can slip before it moves into the critical path.

The PERT technique, strictly speaking, is an elaboration of critical-path scheduling in which one estimates three times for every event, times corresponding to different probabilities of meeting the estimated dates. I do not find this refinement to be worth the extra effort, but for brevity I will call any critical path network a PERT chart.

The preparation of a PERT chart is the most valuable part of its use. Laying out the network, identifying the dependencies, and estimating the legs all force a great deal of very specific planning very early in a project. The first chart is always terrible, and one invents and invents in making the second one.

As the project proceeds, the PERT chart provides the answer to the demoralizing excuse, "The other piece is late anyhow." It shows how hustle is needed to keep one's own part off the critical path, and it suggests ways to make up the lost time in the other part.

Under the rug

When a first-line manager sees his small team slipping behind, he is rarely inclined to run to the boss with this woe. The team might be able to make it up, or he should be able to invent or reorganize to solve the problem. Then why worry the boss with it? So far, so good. Solving such problems is exactly what the first-line manager is there for. And the boss does have enough real worries demanding his action that he doesn't seek others. So all the dirt gets swept under the rug.

But every boss needs two kinds of information, exceptions for action and a status picture for education.[12] For that purpose he needs to know the status of all his teams. Getting a true picture of that status is hard.

The first-line manager's interests and those of the boss have an inherent conflict here. The first-line manager fears that if he reports his problem, the boss will act on it. Then his action will preempt the manager's function, diminish his authority, foul up his other plans. So as long as the manager thinks he can solve it alone, he doesn't tell the boss.

Two rug-lifting techniques are open to the boss. Both must be used. The first is to reduce the role conflict and inspire sharing of status. The other is to yank the rug back.

Reducing the role conflict

The boss must first distinguish between action information and status information. He must discipline himself *not* to act on problems his managers can solve, and *never* to act on problems when he is explicitly reviewing status. I once knew a boss who invariably picked up the phone to give orders before the end of the first para-

SYSTEM/360 SUMMARY STATUS REPORT
OS/360 LANGUAGE PROCESSORS + SERVICE PROGRAMS
AS OF FEBRUARY 01,1965

A=APPROVAL
C=COMPLETED

*=REVISED PLANNED DATE
NE=NOT ESTABLISHED

PROJECT	LOCATION	COMMITMNT ANNOUNCE RELEASE	OBJECTIVE AVAILABLE APPROVED	SPECS AVAILABLE APPROVED	SRL AVAILABLE APPROVED	ALPHA TEST ENTRY EXIT	COMP TEST START COMPLETE	SYS TEST START COMPLETE	BULLETIN AVAILABLE APPROVED	BETA TEST ENTRY EXIT
OPERATING SYSTEM										
32K (DESIGN LEVEL (E)										
ASSEMBLY	SAN JOSE	04/--/64 C 12/31/65	10/28/64 C	10/13/64 C 01/11/65	11/13/64 C 11/18/64 A	01/15/65 C 02/22/65				09/01/65 11/30/65
FORTRAN	POK	04/--/64 C 12/31/65	10/28/64 C	10/21/64 C 01/22/65	12/17/64 C 12/10/64 A	01/15/65 C 02/22/65				09/01/65 11/30/65
COBOL	ENDICOTT	04/--/64 C 12/31/65	10/28/64 C	10/15/64 C 01/20/65 A	11/17/64 C 12/08/64 A	01/15/65 C 02/22/65				09/01/65 11/30/65
RPG	SAN JOSE	04/--/64 C 12/31/65	10/28/64 C	09/30/64 C 01/05/65 A	12/02/64 C 01/18/65 A	01/15/65 C 02/22/65				09/01/65 11/30/65
UTILITIES	TIME/LIFE	04/--/64 C 12/31/65	06/24/64 C		11/20/64 C 11/30/64 A					09/01/65 11/30/65
SORT 1	POK	04/--/64 C 12/31/65	10/28/64 C	10/19/64 C 01/11/65	11/12/64 C 11/30/64 A	01/15/65 C 03/22/65				09/01/65 11/30/65
SORT 2	POK	04/--/64 C 06/33/66	10/28/64 C	10/19/64 C 01/11/65	11/12/64 C 11/30/64 A	01/15/65 C 03/22/65				03/01/66 05/30/66
44K DESIGN LEVEL (F)										
ASSEMBLY	SAN JOSE	04/--/64 C 12/31/65	10/28/64 C	10/13/64 C 01/11/65	11/13/64 C 11/18/64 A	02/15/65 03/22/65				09/01/65 11/30/65
COBOL	TIME/LIFE	04/--/64 C 06/33/66	10/28/64 C	10/15/64 C 01/20/65 A	11/17/64 C 12/08/64 A	02/15/65 03/22/65				03/01/66 05/30/66
NPL	HURSLEY	04/--/64 C 03/31/66	10/28/64 C							
2250	KINGSTON	03/30/64 C 03/31/66	11/05/64 C	12/08/64 C 01/04/65	01/12/65 C 01/29/65	01/04/65 C 01/29/65				01/03/66 NE
2280	KINGSTON	06/33/66 C 09/33/66	11/05/64 C			06/01/65 06/30/65				01/25/66 NE
200K DESIGN LEVEL (H)										
ASSEMBLY	TIME/LIFE		10/28/64 C							
FORTRAN	POK	04/--/64 C 06/33/66	10/28/64 C	10/18/64 C 01/11/65	11/11/64 C 12/10/64 A	02/15/65 03/22/65				03/01/66 05/30/66
NPL	HURSLEY	04/--/64 C 03/31/67	10/28/64 C			07/--/65				01/--/67
NPL H	POK	04/--/64 C	03/30/64 C			02/01/65 04/01/65				10/15/65 12/15/65

Fig. 8. A report showing milestones and status is a key document in project control. This one shows some problems in OS development: specifications approval is late on some items (those without "A"); documentation (SRL) approval is overdue on another; and one (2250 support) is late coming out of alpha test.

graph in a status report. That response is guaranteed to squelch full disclosure.

Conversely, when the manager knows his boss will accept status reports without panic or preemption, he comes to give honest appraisals.

This whole process is helped if the boss labels meetings, reviews, conferences, as *status-review* meetings versus *problem-action* meetings, and controls himself accordingly. Obviously one may call a problem-action meeting as a consequence of a status meeting, if he believes a problem is out of hand. But at least everybody knows what the score is, and the boss thinks twice before grabbing the ball.

Yanking the rug off

Nevertheless, it is necessary to have review techniques by which the true status is made known, whether cooperatively or not. The PERT chart with its frequent sharp milestones is the basis for such review. On a large project one may want to review some part of it each week, making the rounds once a month or so.

A report showing milestones and actual completions is the key document. Fig. 8 (preceding page), shows an excerpt from such a report. This report shows some troubles. Specifications approval is overdue on several components. Manual (SRL) approval is overdue on another, and one is late getting out of the first state (ALPHA) of the independently conducted product test. So such a report serves as an agenda for the meeting of 1 February. Everyone knows the questions, and the component manager should be prepared to explain why it's late, when it will be finished, what steps he's taking, and what help, if any, he needs from the boss or collateral groups.

V. Vyssotsky of Bell Telephone Laboratories adds the following observation:

I have found it handy to carry both "scheduled" and "estimated" dates in the milestone report. The scheduled dates are the property of the project manager and represent a consistent work plan for the project as a whole, and one which is a priori *a reasonable plan. The estimated dates are the property of the lowest level manager who has cognizance over the piece of work in question, and represents his best judgment as to when it will actually happen, given the resources he has available and when he received (or has commitments for delivery of) his prerequisite inputs. The project manager has to keep his fingers off the estimated dates, and put the emphasis on getting accurate, unbiased estimates rather than palatable optimistic estimates or self-protective conservative ones. Once this is clearly established in everyone's mind, the project manager can see quite a ways into the future where he is going to be in trouble if he doesn't do something.*

The preparation of the PERT chart is a function of the boss and the managers reporting to him. Its updating, revision, and reporting requires the attention of a small (one-to-three-man) staff group which serves as an extension of the boss. Such a "Plans and Controls" team is invaluable for a large project. It has no authority except to ask all the line managers when they will have set or changed milestones, and whether milestones have been met. Since the Plans and Controls group handles all the paperwork, the burden on the line managers is reduced to the essentials—making the decisions.

We had a skilled, enthusiastic, and diplomatic Plans and Controls group on the OS/360 project, run by A. M. Pietrasanta, who devoted considerable inventive talent to devising effective but unobtrusive control methods. As a result, I found his group to be widely respected and more than tolerated. For a group whose role is inherently that of an irritant, this is quite an accomplishment.

The investment of a modest amount of skilled effort in a Plans and Controls function is very rewarding. It makes far more difference in project accomplishment than if these people worked directly on building the product programs. For the Plans and Controls group is the watchdog who renders the imperceptible delays visible and who points up the critical elements. It is the early warning system against losing a year, one day at a time.

Epilogue

The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man's handiworks. The management of this complex craft will demand our best use of new languages and systems, our best adaptation of proven engineering management methods, liberal doses of common sense; and a God-given humility to recognize our fallibility and limitations.

References

1. Sackman, H., W. J. Erikson, and E. E. Grant, "Exploratory Experimentation Studies Comparing Online and Offline Programming Performance," *Communications of the ACM*, 11 (1968), 3-11.
2. Nanus, B., and L. Farr, "Some Cost Contributors to Large Scale Programs," *AFIPS Proceedings, SJCC* 25 (1964), 239-248.
3. Weinwurm, G. F., *Research in the Management of Computer Programming*, Report SP 2059, 1965, System Development Corp., Santa Monica.
4. Morin, L. H., *Estimation of Resources for Computer Programming Projects*, M.S. thesis, Univ. of North Carolina, Chapel Hill, 1974.
5. Quoted by D. B. Mayer and A. W. Stalnaker, "Selection and Evaluation of Computer Personnel," *Proceedings 23rd ACM Conference*, 1968, 661.
6. Paper given at a panel session and not included in the *AFIPS Proceedings*.
7. Corbató, F. J., *Sensitive Issues in the Design of Multi-Use Systems*, Lecture at the opening of the Honeywell EDP Technology Center, 1968.
8. Taliaferro, W. M., "Modularity the Key to System Growth Potential," *Software*, 1 (1971), 245-257.
9. Nelson, E. A., *Management Handbook for the Estimation of Computer Programming Costs*, Report TM-3225, System Development Corp., Santa Monica, pp. 66-67.
10. Reynolds, C. H., "What's Wrong with Computer Programming Management?" in *On the Management of Computer Programming*, Ed. G. F. Weinwurm, Philadelphia: Auerbach, 1971, pp. 35-42.
11. King, W. R., and T. A. Wilson, "Subjective Time Estimates in Critical Path Planning—a Preliminary Analysis," *Management Sciences*, 13 (1967), 307-320, and sequel, W. R. King, D. M. Witterrongel, and K. D. Hezel, "On the Analysis of Critical Path Time Estimating Behavior," *Management Sciences*, 14 (1967), 79-84.
12. Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*, New York: Wiley, 1969, pp. 428-430. □

Dr. Brooks is presently a professor at the Univ. of North Carolina at Chapel Hill, and chairman of the computer science department there. He is best known as "the father of the IBM System/360," having served as project manager for the hardware development and as manager of the Operating System/360 project during its design phase. Earlier he was an architect of the IBM Stretch and Harvest computers.

At Chapel Hill he has participated in establishing and guiding the Triangle Universities Computation Center and the North Carolina Educational Computing Service. He is the author of two editions of "Automatic Data Processing" and "The Mythical Man-Month: Essays on Software Engineering" (Addison-Wesley), from which this excerpt is taken.

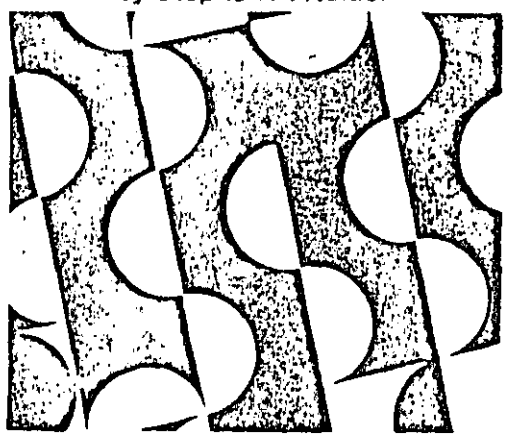
62

In contrast to Dr. Brooks' presentation, this portrait of failure is for those who learn best from looking at bad examples.

55

WHY PROJECTS FAIL

by Stephen P. Keider



ONE OF THE PRIMARY causes for the failure of data processing projects is that such projects are often not initially defined, and therefore may lack a beginning and an end. Once a project has begun, no one seems to know:

- how the project was started;
- what the staffing is, or was, at any one point in time;
- what activities have been performed;
- when the project will end;
- what the project will accomplish.

Essentially, because projects are rarely formally defined, they are rarely completed. Completion occurs usually upon the death—or resignation—of the user the project services, or when the system is due for conversion. Completion is also a prerequisite for success, but a project is considered successful only if completed within the original time or budget estimates, and by how well it satisfies the user's needs.

An unsuccessful project, however, can be identified during several phases of its life cycle; and I shall here try to point to those very indicators.

Logically, any project can be time-divided into five distinct phases:

a) Pre-initiation period (usually measured in weeks or months)
b) Initiation period (measured in weeks)
c) Project duration (in months or years)
d) Project termination period (in weeks or months)
e) Post-termination period (occurring several months after project termination)

In each of the above phases, errors of commission or omission can have major impact upon the success of the total project.

Pre-initiation period

1) No standards exist for estimating how long the project will take. That is, each project is treated as a new and novel system with some individual responsible for estimation. His estimate will be based upon his own understanding of the project and its tasks, and on how quickly *he* can accomplish the subtasks. Little use is made of a history file of similar projects and actual versus originally estimated times.

2) Estimation is not done by the probable project leader, but rather, by whoever happens to be available at estimating time.

3) The project is not adequately defined. The request for an estimate usually takes the form of "John, we're planning to redo the payroll system. What do you think it will require?" "Payroll" may mean a number of different things to different people. Does it involve labor distribution? personnel information? leave accounting? salary, hourly and executive payroll? Any of the above can measurably impact the estimate of the project.

4) Short lead times are allowed for estimates, with corresponding inaccuracy as a result.

5) Personnel availability for the project is unknown. Estimates are usually prepared irrespective of who will perform the work. That is, an estimate of 34 days may be made, but only very junior personnel may be available; this will inflate the actual time. Although the resulting price/performance ratio may be excellent, the success of the project is rated in terms of actual versus estimated time, and on that basis the project may be a failure.

6) Staff desires are unknown. A project may be very appealing to one staff member, but repugnant to another. In both cases the actual time will be affected. Consequently, the Systems Manager must understand staff desires and assign projects accordingly where possible.

Initiation of project

1) Little documentation is available for existing, similar, or interfacing systems to provide the project leader with a data base to build upon.

2) Project leader responsibility is undefined. The leader has no idea what is expected of him, in regard to the project or the personnel assigned to work on it. Should he recommend alternative solutions? Can he recommend terminating the project? Can he remove personnel from it? Can he recommend dismissal?

3) Paper flow is handled poorly (or is nonexistent). Documentation regarding responsibilities, acceptance criteria, system objectives, etc., is not developed. Rather, documentation is limited to the technical aspects of the project.

4) Knowledge of "tools" to perform the project more efficiently is lacking. Are there modules, or subroutines already available which can be used? Is there a test data generator available? What about system design or documentation aids?

5) Definition of the project is vague, misleading, or totally wrong.

6) The project, between the time of the original estimate and its initiation, has changed without a corresponding change in the estimate.

7) Little or no time is spent in planning the project. Rather, analysis design and/or coding is begun immediately upon the project approval. The project leader is not permitted the "luxury" of planning: how he will attack the project, what tasks will be done first, second or third; what approach he will use, or what similar projects he will investigate or review.

8) Problem avoidance is not understood or considered. Oddly

enough, all projects begin with the premise that everything will go smoothly. Items such as lack of test time due to year-end closing are not considered until after the problem has occurred. By then, the project has already lost several days, or it is too late to provide an alternate source.

9) Resource requirements are not scheduled for the project. Critical items, such as keypunch, test time, user manual typing, secretarial, and printing requirements become a problem, and are addressed only *after* they have affected the project.

10) The project team's activities are not clearly presented to the end user. Only too often, the result is a series of "I thought . . ." "I assumed . . ." "Isn't he . . . ?" comments.

11) Project completion elements are not defined. That is, the project leader is not aware of what constitutes completion of the project. What is the end product? What test/acceptance criteria will be used? Who must sign off on project turnover? What constitutes turnover?

Duration of the project

1) Posting or reporting of project information is not performed, resulting in the project leader being unaware of what the completion percentage is, and the user being unaware of the impact of changes upon the original system.

2) Project reviews are typically exercises in trivia. They constitute a "How's it going, Jack? Any problems? No? Good! See you next week." The weak systems manager does not ask probing, detailed questions. He does not require that his personnel anticipate problems, but is primarily concerned with identifying problems which his project leader already has recognized.

3) Change of personnel is one of the major reasons why projects fail. Personnel, including project leaders, are removed from the project, with no adjustments to the schedule for time lost due to the changes. Whenever a team member is added to a project, there is a learning curve which impairs his efficiency on the project. It may be a day, or a month, but unfortunately, people movement is considered to be transparent to the project completion.

4) Adherence to standards and specifications is either not defined or, if defined, not followed. More often than not, standards do exist, especially in larger installations. They address documentation techniques, labeling, file names, etc. However, once an initial indoctrination is provided for a programmer/analyst, follow-up is ignored. The most expedient solutions are followed, resulting in several steps (modules) in the same program sequence addressing the identical file with different mnemonics. It results for example, in sketchy operations documentation without consideration for restart procedures. Maintenance then becomes a major part of project development.

5) Resource requirements are not anticipated. The major offenders in this area are:

- Data entry. Inadequate time is permitted for turnaround of source code preparation and/or test file operation. Worse, verification may not be performed, which almost invariably adds at least one day to the program development cycle.
- Computer Test Time. The lack of adequate test time becomes extremely critical toward the end of a project, when only one or two programs are being finalized. If turnaround is overnight, each minor change to a program adds at least one full day to the duration.
- Design Level Reviews. Whereas most of the time these are considered in project planning, it is rare that anything longer than a minute is assumed for duration between submission of design specifications and approval.

6) "Brute Force" Approach. In this type of shop, everything is designed and implemented from scratch with no thought given to the use of past projects, tools, or work simplification methods available to shorten the development cycle.

7) Lack of a project manager. It sounds strange, but many projects flounder through to completion without a rudder. The "DP Manager" is normally the project leader and he provides as much attention as he can considering his other duties. In general, very few installations have one man accountable for an entire project, but rather fragment the responsibilities to the point where no one person is accountable.

8) Lack of a Project Log. A project log can be an invaluable tool in performing post-mortems. Further, in companies which charge-back to the user the cost of resources used; it can be the mainstay in justifying such charge-backs.

9) Lack of a project audit trail. Data audit trails are considered the key to the development of any financially

sound accounting system. Yet very few project managers concern themselves with maintenance of a project workbook to provide a similar audit trail for project development.

10) Lack of a skills inventory. Many projects are pursued with the project manager completely unaware of the skills available to him within his own shop. A skills inventory of past accomplishments of each staff member simplifies the staffing of a project and ensures that experience is "recyclable."

11) Lack of project milestones. Because project milestones are not determined at the onset of a project, percentage of completion is usually equated to percentage of hours expended. For example, a project for which 100 hours has been estimated is 60% complete when 60 hours have been expended; when 90% of the hours have been expended, it is 90% complete. This can likewise be extrapolated to 140% complete when 140 hours have been expended.

12) Staff members are considered "universally expert." During the estimation stage, and again during implementation, staff members are considered to be equally competent analysts, designers, programmers, librarians, documentation specialists, etc. They are assigned any of these functions with little consideration given to their ability. Invariably, this results in project delay.

13) Utilization Philosophy. A most fundamental problem which affects many large companies is one which demands maximizing the utilization of personnel, as opposed to a project-oriented approach. When a lull occurs in a particular project, staff members are reassigned, because it is anathema to have people not performing "useful" (that is, design or programming) work. Consequently, when the project restarts, the same people may not be available, or worse yet, are available part time. This is a disastrous approach, because while it assures that people are always assigned to a project and utilization is high, it places an emphasis upon effort, not results.

Termination of the project

In the first place, it is my opinion that projects never terminate. Rather, they become like Moses, condemned to wander till the end of their days without seeing the promised land. However, for those projects that do "terminate," the following are key deficiencies:

1) History statistics are not determined or not updated. For example, at project termination, the project leader should make some attempt to determine performance in light of certain objectives, or measurable criteria: how many programs were written? how many lines of code generated? average lines of code per day? average source statements per programmer? cpu test time required per programmer? per program? All of the above can be invaluable tools in the estimating and evaluating of future projects. It becomes the first step in the development of a "cost resource accounting system" for future projects.

2) Quality Control. Typically, when a project is completed, it is never evaluated for quality. The QC criteria is "does the program run?" There are no grades (i.e., A, B, C, D, F) of programs. They are either "As" or "Fs."

The manager may evaluate personnel based upon quantity of code, programs or documentation produced, but in fact he never even considers evaluation based upon the quality of coding techniques used.

3) Knowledge gained is rarely transferable. Once a project is completed, it goes through a procedure similar to "de-Stalinization," wherein all vestiges of association with a project are forgotten lest one be stuck with program maintenance. Inadequate time is allowed at the conclusion of the project for staff members to "dump" the knowledge gained or even provide meaningful insight into techniques used.

4) Personnel are not evaluated. There is an ideal time, and only one, to evaluate performance of an individual on a project, and that is immediately at the conclusion of a project. Yet, only too often, personnel evaluation is tied into employment anniversary dates. Between the time an individual has completed a project and his next appraisal, a year may have lapsed. During that year he has had the opportunity to perpetuate mistakes initially made 12 months ago.

5) Lack of formal turnover. Typically, a project termination is first known by the appearance of a new report. More realistically, a formal presentation should take place addressing:

a) initial objectives of the project
b) performance against these objectives
c) review of the end product
d) designation of principal contact for maintenance, etc.

6) Recommendations for enhancement are not documented. At the conclusion of a project (if not earlier) the project team is in an ideal position to recommend enhancements to the system. If these are not quantified immediately, they will be lost forever.

Post termination

64

The key ingredient here is the conducting of user satisfaction surveys six to nine months after the completion of a project. The survey should address:

a) results versus objective
b) integrity of data
c) freedom from bugs
d) quantification of changes required
e) usefulness of information (i.e., should the system be continued?)

Summary

As a result of reviewing the development of a number of major systems, the above faults exist more often than not. However, the key problems appear in failing to understand the characteristics of a project:

- It has a beginning.
- It has an end.
- It uses multiple, finite resources.
- It has an objective.
- Its success can be measured in terms of time or dollars.
- It requires a leader.
- It requires a staff.
- It must be planned.
- Performance against plan must be reviewed.
- It coexists with other projects but is distinct from them.
- It is measurable (quantifiable).
- It may be a bad project (from the standpoint of usefulness). If it is, it must be altered, or terminated.
- Internal and external forces will affect a project; they must be identified.
- A project is a group of sub-projects.
- No project is unique.

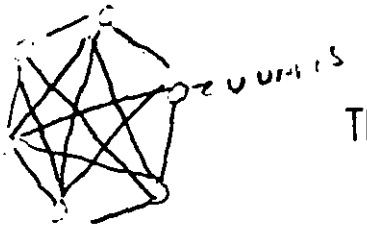
Unless full attention is paid to each of these aspects of a project, the history of project failure will be played out once again.

A vice president and senior consultant with Neoterics, Inc., a Cleveland consulting firm, Mr. Kelder has specialized in operations and systems auditing with emphasis on project management. He held positions in systems engineering, education management, and systems management while with IBM, where he spent 12 years prior to joining Neoterics.

INTEGRATED COMPUTER SYSTEMS™

from Brooks

COMMUNICATION	UNITS	UNITS	INTERACTION
o—o	1	1	2
	3	3	6
	5	10	20



THIS PAGE INTENTIONALLY LEFT BLANK.

v	$\frac{v(v-1)}{2} \sim v^2$	v^2

Look for no more than 7 interactions

PERSPECTIVES ON SOFTWARE ENGINEERING	MARVIN V. ZELKOWITZ
MAKING THE MOVE TO STRUCTURED PROGRAMMING	EDWARD YOURDON
AN EXAMPLE OF STRUCTURED DESIGN	BILL INMON
THE NEED FOR SOFTWARE ENGINEERING	WARE MYERS
SOFTWARE ENGINEERING: PROCESS, PRINCIPLES, AND GOALS	DOUGLAS T. ROSS, JOHN GOODENOUGH, C.A. IRVINE
THE MYTHICAL MAN-MONTH	FREDERICK P. BROOKS,JR
WHY PROJECTS FAIL	STEPHEN P. KEIDER

Perspectives on Software Engineering

MARVIN V. ZELKOWITZ

Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C. 20234, and Department of Computer Science, University of Maryland, College Park, Maryland 20742

Software engineering refers to the process of creating software systems. It applies loosely to techniques which reduce high software cost and complexity while increasing reliability and modifiability. This paper outlines the procedures used in the development of computer software, emphasizing large-scale software development, and pinpointing areas where problems exist and solutions have been proposed. Solutions from both the management and the programmer points of view are then given for many of these problem areas.

Keywords and Phrases: certification, chief programmer team, program correctness, program design language (PDL), software reliability, software development life cycle, software engineering, structured programming, top-down design, top-down development, validation, verification.

CR Categories: 1.3, 4.0, 4.6

INTRODUCTION

Software development usually proceeds in one of two ways: either the programmer works alone in designing, implementing, and testing a software system, or he is a member of a group of from three up to several hundred, working together on a large software system. Although software engineering embraces both approaches, here we are interested mainly in large-scale program development.

When the Verrazano Narrows Bridge in New York City was started in 1959, officials estimated that it would cost $325 million and be completed by 1965. It is the largest suspension bridge ever built, yet it was completed in November 1964, on target and within budget [ENR61, ENR64]. No similar pattern has been observed when we build software systems larger than those which had been built previously.

Software is often delivered late. It is frequently unreliable and usually expensive to maintain. The IBM OS project, which involved over 5,000 man-years of effort, was years late [BROO75]. Why is bridge engineering so exact while software engineering flounders so?

Part of the answer lies in the greater ease with which a civil engineer can see the added complexity of a larger bridge than a software engineer the complexity of a larger program. Part of today's "software problem" stems from our attempt to extrapolate from personal experiences with smaller programs to large systems programming projects.

We begin here by outlining the general approach used in developing program products, emphasizing aspects which are still poorly understood. Later, we enumerate the techniques which have been used to solve these problems. We do not attempt to cover all of the relevant topics in depth, but we give many references for further reading.

Software engineers are currently study-

CONTENTS

ing the causes of these problems and the mechanisms of software development. They seek both constraints on programming which will render software less expensive and more reliable and also the theoretical foundations upon which programs are built. Software engineering is not the same as programming, although programming is an important component. It is not the study of compilers and operating systems, although compiler writers and operating system implementors use similar techniques. It is not electrical engineering, although electronics does provide the basis for implementing the computer [JEFF77].

Software engineering is interdisciplinary. It uses mathematics to analyze and certify algorithms, engineering to estimate costs and define tradeoffs, and management science to define requirements, assess risks, oversee personnel, and monitor progress.

1. STAGES OF SOFTWARE DEVELOPMENT

The complexity of a large software system surpasses the comprehension of any one individual. To better control the development of a project, software managers have identified six separate stages through which software projects pass; these stages are collectively called the *software development life cycle*:

- Requirements analysis;
- Specification;
- Design;
- Coding;
- Testing;
- Operation and maintenance.

Figure 1, a pie chart, shows the approximate amount of time each stage takes. The stages are discussed in the following subsections.

Requirements Analysis

This first stage, curiously absent from many projects, defines the requirements for an acceptable solution to the problem. The statement "Write a COBOL program of not more than 50,000 words to produce payroll checks" is not a requirement; it is the partial specification of a computer solution to the problem. The computer is merely a tool for solving the problem. The requirements analysis focuses on the interface between

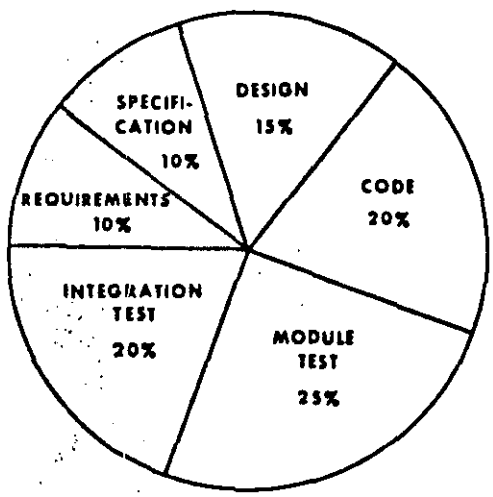


FIGURE 1. Effort required on various development activities (excluding maintenance).

the tool and the people who need to use it. For example, a company may consider several methods of paying its employees: 1) pay employees in cash; 2) use a computer to print payroll checks; 3) produce payroll checks manually; or 4) deposit payroll directly into employees' bank accounts.

Other aspects, such as processing time, costs, error probability, and chance of fraud or theft, must be considered among the basic requirements before an appropriate solution may be chosen. A requirements analysis can aid in understanding both the problem and the tradeoffs among conflicting constraints, thereby contributing to the best solution.

Hard requirements and the optional features must be distinguished. Are there time or space limitations? What facilities of the system are likely to change in the future? What facilities will be needed to maintain different versions of the system at different locations?

The resources needed to implement the system must be determined. How much money is available for the project? How much is actually needed? How many computers or computer services are affordable? What personnel are available? Can existing software be used? After the first questions are answered, project schedules must be planned. How will progress be controlled and monitored? What has been learned from previous efforts? What checkpoints will be inserted to measure this progress? Once all these questions have been answered, specification of a computer solution to the problem may begin.

Specification

While requirement analysis seeks to determine whether to use a computer, *specification* (also called *definition* [FIFE77]) seeks to define precisely what the computer is to do. What are the inputs and outputs? In the payroll example: Are employee records in a disk file? On tape? What is the format for each record in the file? What is the format for the output? Are checks to be printed? Is another tape to be written containing information for printing the checks offline? Will printed reports accompany the checks? What algorithms will be needed for computing deductions such as tax, unemployment and health insurance, or pension payments?

Since commercial systems process considerable amounts of data, the database is a central concern. What files are needed? How will they be formatted, accessed, updated, and deleted?

When the new system supersedes an older process (for example, when an automatic payroll system replaces a manual system), the conversion of the existing database to the new format must be part of the design. Conversion may require a special program which is discarded after its first and only use. Since the company may be using the older system in its day-to-day operation, bringing the new system online presents a problem. Can the old and the new systems run side by side for awhile?

The answers to these questions are set forth in the *functional specification*, a document describing the proposed computer solution. This document is important throughout the project. By defining the project, the specification gives both the purchaser and the developer a concrete description. The more precise the specifications are, the less likely will be errors, confusion, or recriminations later. The specifications enable test data to be developed early; this means that the performance of the system can be tested objectively, since the test data will not be influenced by implementation. Because it describes the scope of the solution, this document can be used for initial estimates of time, personnel, and other resources needed for the project.

These specifications define only what the system is to do, but not how to do it. Detailed algorithms for implementation are premature and may unduly constrain the designers.

Design

In the design stage, the algorithms called for in the specifications are developed, and the overall structure of the computer system takes shape. The system must be divided into small parts, each of which is the responsibility of an individual or a small

team. Each such module thus defined must have its constraints: its function, size, and speed.

As submodules are specified, they are represented in a tree diagram showing the nesting of the system's components. Figure 2 illustrates this for a typical compiler. This illustration, sometimes called a *baseline diagram*, is not by itself an adequate specification of the system.

Because the solution may not be known when the design stage starts, decomposition into small modules may be quite difficult. For older applications (such as compiler writing) this process may become standardized, but for new ones (such as defense systems or spacecraft control) it may be quite difficult.

A common problem is that the buyer of a system often does not know exactly what he wants, especially in state-of-the-art areas such as defense systems. As he sees the project evolve, the buyer often changes the specifications. If this occurs too often, the project may flounder. We discuss this problem later.

Coding

Coding is usually the easiest stage. High-level languages and structured programming simplify the task. In one study, Boehm [BOEH75] found that 64% of all errors occurred in design, but only 36% in coding. Hamilton and Zeldin [HAMI76] report that in the NASA Apollo project about 73% of all errors were design errors. We have mastered coding better than any other stage of software development.

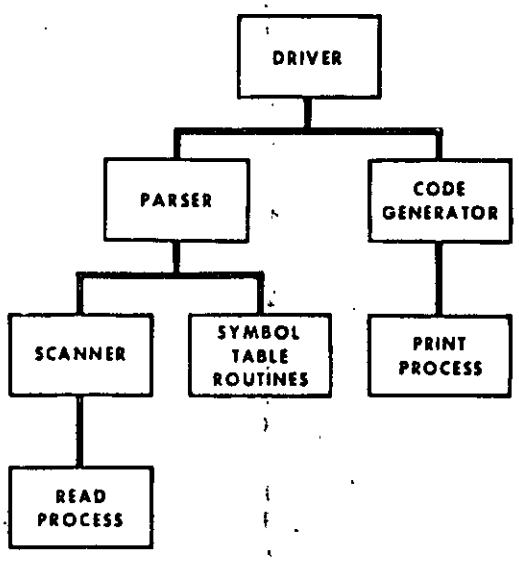


FIGURE 2. Sample baseline diagram for a compiler.

Testing

The testing stage may require up to half of the total effort. Inadequately planned testing often results in woefully late deliveries.

During testing the system is presented with data representative of that for the finished system; thus test data cannot be chosen at random. The test plan should, in fact, be designed early and most of the test data should be specified during the design stage of the project.

Testing is divided into three distinct operations:

1) *Module testing* subjects each module to the test data supplied by the programmer. A test driver simulates the software environment of the module by containing dummy routines to take the place of the actual subroutines that the tested module calls. Module testing is sometimes called *unit testing*. A module that passes these tests is released for integration testing.
2) *Integration testing* tests groups of components together. Eventually, this procedure produces a completely tested system. Integration testing frequently reveals errors missed in module tests. Correcting them may account for about a quarter of the total effort.
3) *Systems testing* involves the test of the completed system by an outside group. The independence of this group is important.

The buyer may also insist on his own systems test, or *acceptance test*, before formally accepting the product. Comparison of the performance of several systems (such as those of a given software product already available from several sources) is called *benchmark testing*.

During testing, many criteria are used to determine correct program execution. Among other important criteria, the pro-

gram is considered correct if:

1) every statement has been executed at least once by the test data;
2) every path through the program has been executed at least once by the test data; and
3) for each specification of the program, test data demonstrate that the program performs the particular specification correctly.

These three different criteria show that there is no single acceptable criterion defining a "well-tested" program. Goodenough and Gerhart [GOOD76] proposed a set of consistent definitions for "testing" and showed that some of these definitions of testing are, in theory, insufficient. We return to this subject later. For a survey of good testing techniques, see [HUAN75].

Closely related to testing are verification and validation (V/V). A system is *validated* when testing shows that the system performs according to its specifications. A system is *verified* when it has been proved to meet its specifications. Current technology is inadequate for achieving both these objectives. A validated system may misbehave for cases not included in the test data. A verified system is correct relative only to the initial specifications and assumptions about the operating environment; formal proofs tend to be lengthy, making them subject to error or incredulity. *Certification* sometimes refers to the overall process of creating a correct program by validation and verification.

In certifying a program, three terms must be distinguished. A *failure* in a system is an event which marks a violation of the system's specifications. An *error* is an item of information which, when processed by the normal algorithms of the system, produces a failure. Since error recovery may be built into the program (for example, ON units in PL/I), not every error will produce a failure. A *fault* is a mechanical or algorithmic defect which generates an error (for example, a programming "bug") [DENN76a].

Reliability is a concept which must not be confused with correctness. A *correct* program is one that has been proved to meet its specifications. In contrast, a *reliable* program need not be correct, but gives acceptable answers even if the data or environment do not meet the assumptions made about them. We would like a system to be highly robust, that is, to accept a large class of input data and to process it correctly under adverse conditions. Parnas [PARN75] describes a correct system as one that is free from faults and has no errors in its internal data. A program is reliable if failures do not seriously impair its satisfactory operation.

Operating systems with "fail-soft" procedures illustrate the difference between reliability and correctness. A detected error causes the system to shut down without losing information, possibly restarting after error recovery. Such a system may not be correct because it is subject to errors, but it is reliable because of its consistent operation. A real-time program may be correct as long as a sensor reports correctly, but it may be unreliable if bad sensor readings have not been considered.

Operation and Maintenance

Figure 1 shows the disposition of software costs in developing a new project. But this can be the wrong chart! The activities noted in Figure 1 are only 25% to 33% of the effort required during the life of the system. Figure 3 illustrates that maintenance costs ultimately dwarf development costs.

No computer system is immutable. Since a buyer seldom knows what he wants, he seldom is satisfied. Probably, he will request changes in the delivered system. Errors missed in testing will later be discovered. Different installations will need special modifications for local conditions. The management of multiple copies of a system is another difficult problem that must be handled early in development. Once the first line of code is written, the structure of the resulting maintenance operation may already be fixed, so it is best to plan for it then.

The division of effort indicated in Figure 3 greatly affects system development. Because of hidden maintenance costs, techniques that rush development and provide

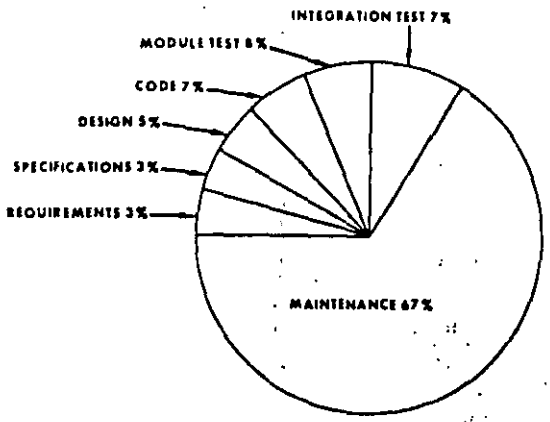


FIGURE 3. True effort on many large-scale software systems.

for very early initial implementation may be trading early execution for a much more extensive maintenance operation.

The maintenance problem is sometimes referred to as the "parts number explosion." For example, a certain system contains components A, B, and C. Installation I finds and reports an error. The developer fixes the error and sends a corrected module A' to all installations using the system.

Installations II and III ignore the replacement and continue with the original system. Installations I and II discover another error in module A. The developer must now determine whether both of these errors are the same, since different versions of module A are involved. The correction of this error involves correction of both A' (for I) and A (for II) yielding A'' and A'''. There are now three versions of the system.

To avoid this growth, systems often receive updates, called releases, at fixed intervals. A useful tool for dealing with myriad maintenance problems is a "systems database" started during the specifications stage. This database records the characteristics of the different installations. It includes the procedures for reporting, testing, and repairing errors before distributing the corrections.

Themes of Software Engineering

It should be clear that each software development stage may influence earlier stages. The writing of specifications gives feedback for evaluating resource requirements; the design often reveals flaws in these specifications; coding, testing, and operation reveal problems in design. The goals of software engineering are thus to:

- Use techniques that manage system complexity.
- Increase system reliability and correctness.
- Develop techniques to predict software costs more accurately.

In the following sections, we discuss approaches to some of these problems. The list of techniques is divided into management and programmer issues. Management issues concern the effective organization of personnel on a project. Programmer issues concern the techniques used by individual programmers to improve their performance.

2. MANAGEMENT ISSUES

A manager controls two major resources: personnel and computer equipment. This section surveys techniques for optimizing the use of these resources.

Size and Cost Control

A project may fail when management is not aware of developing problems; a year's delay comes "one day at a time" [BROO75]. Faced with catastrophic failure (for example, needed hardware is delayed six months), a resourceful manager can usually find alternatives. However, it is easy to ignore day-to-day problems (such as sick employees or many errors during testing).

Most problems occur at the interfaces of modules written by different programmers. Since the number of such interfaces is on the order of the square of the number of individuals involved, the problem becomes unwieldy when the number of persons in a development group grows to four or more.

As an example of the communications problem, assume that a single programmer is capable of writing a 5,000-line program in a year, and that a programming system requires about 50,000 lines of code and is to be completed in two years. Five programmers would seem to be sufficient (see Figure 4a).

However, the five programmers must communicate with one another. Such communication takes time and also causes some loss in productivity since finding misunderstood aspects will require additional testing. For this simple analysis, assume that each communication path "costs" a programmer 250 lines of code per year. Each of the five programmers, therefore, can produce only 4,000 lines per year and only 40,000 lines are completed within two years (see Figure 4b).

This means that eight programmers producing 3,250 lines per year are actually needed in order to produce the required 50,000. A manager is required for direction of this large effort. Therefore, in summary, eight programmers and a manager, each producing an average of 3,000 lines per year, are actually needed (see Figure 4c).

As we shall see, simply counting lines of code is not a good way to estimate productivity. The figures in this example are only given to illustrate a point, but they are representative of the problem. There are also techniques designed to limit this communications "explosion" and to increase programmer productivity.

Project Personnel

Software can usually be divided into three categories: 1) control programs (such as operating systems), 2) systems programs (such as compilers), and 3) applications programs (such as file management systems). A single programmer working on a control program can produce about 600 lines of code per year, whereas he can produce about 2,000 lines if working on a systems program and about 6,000 if working on an applications program [WOLV74]. The type of task certainly affects the productivity that can be expected from a given programmer. However, as the previous example demonstrates, the organization of personnel also affects performance. For example, with the approach of deadlines, docu-

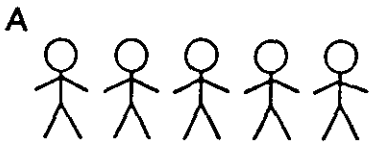


FIGURE 4(a). Single projects: 5,000 lines per year = 50,000 lines in two years (no communication between programmers).

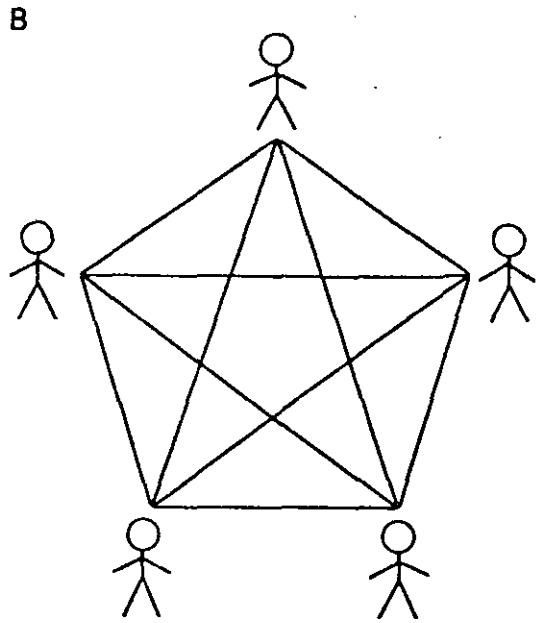


FIGURE 4(b). Five-member group: 4,000 lines per year = 40,000 lines in two years (ten communication pairs).

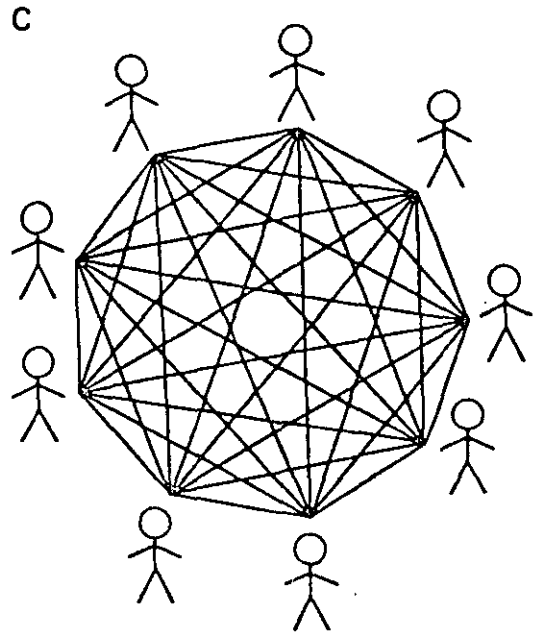


FIGURE 4(c). Nine-member team: 3,000 lines per year = 50,000 lines in two years (36 communication pairs).

mentation is often given lower priority. However, since 70% of the total system cost may occur during the maintenance state (where the documentation is heavily used), this may be a false economy of effort.

Use of a librarian is one way to avoid this problem. A librarian provides the interface between the programmer and the computer. Programs are coded and given to the librarian for insertion into the online project library. The actual debugging of the module is carried out by the programmer, but changes to the official module in the library are made by the librarian. The use of a library is further enhanced when an online data management system is used.

The use of a librarian has another beneficial effect. All changes in modules in the project library are handled by one individual and are easy to monitor; they are often reviewed by the project manager before insertion. This prevents "midnight patches" from being quickly incorporated into a system and forces the programmer to think carefully about each change. It also gives the manager disciplined product control and helps with audit trails.

On larger projects, a technical writer may perform much of the documentation, thus freeing programmers for the tasks for which they are most skilled.

The culmination of this trend is the *chief programmer team* concept developed by IBM [BAKE72]. The concept recognizes that programmers have different levels of competence; therefore, the most competent should do the major work, while others function in supporting roles. As the earlier example shows, interfacing problems greatly reduce programmer productivity. The chief programmer team is one way of limiting this complexity.

The chief programmer, an excellent programmer and a creative and well-disciplined individual, is the head of the team. He may be five or more times more productive than the lowest member of the team [BOEH77]. He functions as the technical manager of the project, designs the system, and writes the top-level interfaces for all major modules.

If a project is large, a team may also have an administrative manager to handle such responsibilities as budgeting time, vacations, office space, and other resources, and reporting to upper-level management. The administrative manager often administers several programming teams.

The backup programmer works with the chief programmer and fills in details assigned by the chief programmer. Should the chief programmer leave the project, the backup programmer would take over. This means that he also must be an excellent programmer. The backup programmer also fulfills an important role by providing the chief programmer with a peer with whom he can discuss the design.

There are also two or three junior programmers assigned to the team to write the low-level modules defined by the chief programmer. The term "junior" in this context means "less experienced," not "less capable." As Boehm states, the best results occur with fewer and better people.

Using the example illustrated by Figure 4, a chief programmer team of five individuals has only seven communications paths, and the chief programmer, being that rare individual, can produce more than his quota of 5,000 lines (see Figure 5). Thus productivity per programmer could be greater than 5,000 lines per year, instead of the previous figure of only 4,000.

The team has a librarian to manage the project library—both the online module library and the offline project documentation (also called the project notebook). The project notebook contains, among other things, records of compilations and test runs of all modules. It is important to the team structure, since all development is now accountable and open for inspection, and code is no longer the "private property" of any individual programmer.

Programmers have traditionally been reluctant to exhibit their products until completion, since discovered errors have traditionally been viewed as a personal failure. The absurdity of this approach is clear enough. If the ego element is removed from programming, programmers may openly ask others for advice when they need it, instead of trying to solve all problems themselves [WEIN71].

The team may include other supporting

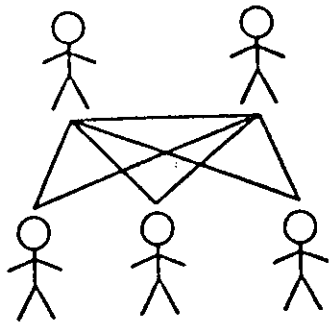


FIGURE 5. Fewer communications paths in a chief programmer team.

personnel such as secretaries and technical writers. Experience shows that ten is the upper bound to team size.

This structure, however, will not solve all problems in development. With a smaller number of individuals involved, competence is crucial. It is not possible to "work around" a nonproductive individual as one might do in a large project. There are also extremely large projects where a group of ten is simply too small to tackle development. Larger teams are not efficient.

A man-month, or the amount of work performed by one individual in one month, is a deceptive measure for estimating project productivity. A project requiring four programmers for a year cannot be completed by 48 programmers in one month. The example of the 50,000 line system needed in two years shows some of the problems inherent in trying to exchange programmers for time. "Adding manpower to a late software project makes it later" [BROO75]. New personnel divert existing personnel needed to train them; they require more supervision; they complicate communication and interfere with the design since they are unfamiliar with the project structure.

However, man-months do serve a purpose as a useful measure of project costs. By adding more data, such as the rate of using man-months, accurate cost estimation techniques can be utilized. These are explained in the following subsection.

Estimation Techniques

One of the most important aspects of engineering is estimating the resources needed to complete a project. As previously mentioned, the Verrazano Narrows Bridge in New York City was completed at the projected time and within the estimated budget. How was such accuracy achieved?

Most engineering disciplines have highly developed methods of estimating resource needs. One such technique is the following [GALL65]:

1) Develop an outline of the requirements from the Request for Quotation (RFQ);
2) Gather similar information, for example, data from similar projects;
3) Select the basic relevant data;
4) Develop estimates;
5) Make the final evaluation.

Although this approach has been advocated for software development, software projects have difficulty passing Step 1 [WOLV74]. Engineers have been building bridges for 6,000 years but software systems for only 30 years. Prior experience to develop the true requirements may not be available. Moreover, with very little background to build on, the developer has little knowledge of similar systems to use in evaluation (Step 2).

In developing the estimates (Step 4), the following tasks must be undertaken:

4a) Compare the project to similar previous projects.
4b) Divide the project into units and compare each unit with similar units;
4c) Schedule work and estimate resources by the month.
4d) Develop standards that can be applied to work.

Note that for Step 4a), the lack of previous experience presents a continuing problem. Also, for Step 4d), an adequate set of standards does not yet exist.

Experience is the key to accurate estimation. Even civil engineering projects may fail badly when established techniques are not followed. Although the Verrazano Narrows Bridge was the world's largest suspension bridge, its engineers had much experience with other similar structures. On the other hand, the Alaskan oil pipeline was estimated to cost $900 million, yet by mid-

1977 the cost had risen past $9 billion [ENR77]. In this case, the design was altered continuously as the federal government imposed new environmental standards (that is, changing specifications), and new technologies were needed to move large quantities of oil in a cold weather environment. Previous experience was only marginally helpful.

Results from computer hardware reliability theory are now starting to play a role in software estimation [PUTN77]. The cumulative expenditures over time for large-scale projects have been found to agree closely with the following equations:

$$E = K(1-e^{-at^2})$$

where E is the total amount spent on the project up to time t, K is the total cost of the project, and a is a measure of the maximum expenditures for any one time period. This relationship is usually expressed in its differential form, called a Rayleigh curve:

$$E' = 2Kate^{-at^2}$$

where E' is the rate of expenditures, or the amount spent on the project during year number t. Since 70% of the cost of a project occurs during the maintenance stage, it is not surprising that the maximum expenditures will occur just before the product is released, a time when it is usually assumed that the effort is winding down before termination (see Figure 6).

The Rayleigh curve has two parameters, K and a; however, a system can be described by three general characteristics: 1) total cost, 2) rate of expenditure, and 3) completion date. Two of these characteristics are enough to determine the constants K and a. When a project is initiated, the proposed budget is an estimate of K, and the available personnel permits a to be calculated. Assuming that requirement analysis determines that these figures represent an accurate assessment of the complexity of the problem, the estimated completion date (the date when the expenditures reach a maximum) can be computed, and thus cannot be set arbitrarily during the requirements or specification stage. This method provides the basis for a cost estimation strategy that has been applied to smaller projects in the 100 man-month range [BASI78]. We may be close to a mathematical theory of cost estimation which will greatly reduce our need to "guess" at project costs.

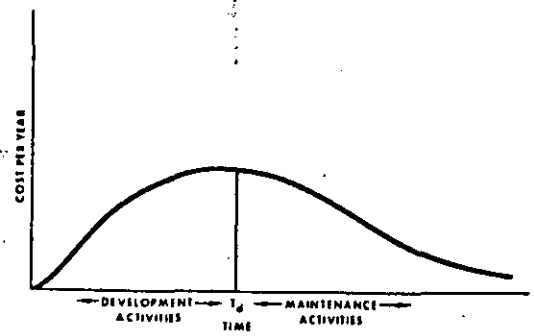


FIGURE 6. Yearly rate of expenditures approximates the Rayleigh curve. Total cost (area under curve) = K; $a = 1/T_d^2$; rate = $2Kate^{-at^2}$.

Milestones

A *milestone* is the specification of a demonstrable event in the development of a project. Milestones are scheduled by management to measure progress. "Coding is 90% complete" is not a milestone because the manager cannot know when 90% of the code is complete until the project itself is complete.

There are many candidates for milestones: publication of the functional specifications, writing of individual module designs, module compiling without errors, units that have been tested successfully, and so on. Milestones are scheduled fairly often to detect early slippage. PERT charts may be used to estimate the effects of slippage in one stage on later stages.

Reporting forms can give information useful for estimating when a future milestone will be reached. A general project summary, describing such overall characteristics as system size, cost, completion dates, or complexity, can be resubmitted with each milestone. Change reports can be submitted each time a module is altered. The use of a librarian probably means that such a form already exists. Weekly personnel and computer reports monitor expenditures, Although they add a minor overhead to the project, the information helps management keep abreast of progress [BASI78, WALS77].

Development Tools

Compilers and certain debugging facilities have been available for some time. In contrast, other programming aids are new and experience with them is less estensive. Cross referencing, attribute listings, and symbolic storage maps are examples of such aids. Auditors or database systems can help to control the organization of the developing system. The Problem Statement Language/Problem Statement Analyzer (PSL/PSA) of the ISDOS project of the University of Michigan is one of the first database systems for providing a module library for storing source code, and includes a language for specifying interfaces in system design which can be checked automatically [TEIC77]. RSL/SSL is a similar system designed to specify requirements and to design interfaces via a data management system [DAVI77].

An alternative approach is the Programmer's Workbench developed by Bell Telephone Laboratories [DOLO76]. A PDP 11 based system provides a set of support routines for module development, library maintenance, documentation, and testing. Proper use of these facilities allows accessing information in an easier, controlled environment.

Reliability

Conceptual Integrity

Conceptual integrity, uniformity of style and simplicity of structure, are usually achieved by minimizing the number of individuals in the project. A chief programmer team greatly enhances conceptual integrity.

A small group minimizes contradictory aspects of a design. In the PL/I language, for example, the PICTURE attribute declaration may be abbreviated as either PIC or P, but in format specifications it may only be P [ANSI76]. In FORTRAN, the right side of an assignment statement can be an arbitrary arithmetic expression, but DO loop indices must be integer constants or variables, and subscripts to arrays are limited to seven basic forms [ANSI66]. These are difficult idiosyncracies to remember. They illustrate a lack of conceptual integrity that can arise when many people with different objectives become involved in a project. A consistent design is less prone to errors because the user can follow a simple set of rules.

Continual System Validation

A *walkthrough* is a management review to discover errors in a system. In one study, TRW discovered that the cost of fixing an error at the coding stage is about twice that of fixing it at the design stage, and catching it in testing costs about ten times as much as it does in design [BOEH76].

A walkthrough is scheduled periodically for all personnel. In attendance are the project manager (chief programmer), the person reviewed, and several others knowledgeable about the project. One section of the system is selected for review and each individual is given information about that section (for example, design document for a design walkthrough, code for a coding walkthrough) before the review. The person being reviewed then describes the module under study.

The walkthrough is intended to detect errors, not to correct them. Also, the walkthrough is brief—not more than two hours. By explaining the design to others, the person reviewed is likely to discover vague specifications or missing conditions.

An important point for management is that the walkthrough is *not* for personnel evaluation. If the person reviewed perceives that he is being evaluated, he may attempt to cover up problems or present a rosy picture.

An informal yet very effective version of the walkthrough is *code reading*. A second programmer reviews the code for each module. This technique frequently turns up errors when the second reader, failing to understand some aspects of the code, asks the author for an explanation.

3. PROGRAMMER ISSUES

Each stage of the software development life cycle has its own set of problems and solutions. The most advanced techniques apply to the last stages; the first stages are the least developed. For example, testing an

debugging problems are apparent to every programmer; these tools are the oldest and most advanced. Techniques for improving coding were developed next. The most recent developments have related to requirements and specifications. Although many technical problems have not been solved, an effective methodology is emerging. Some of these techniques are presented in the following subsections.

Verification and Validation

Verification and validation (module and integration testing) of a system occupy about half of the development time of a project. Many debugging aids have been developed to facilitate this effort; most are implemented as programs to test some feature of a system.

Automated Tools

The earliest and most primitive debugging tools were the dump and the trace. A *dump* is a listing of the contents of the machine's memory. This listing can often reveal unintelligible data or errors. Unfortunately, a dump may not be taken until long "after the fact" and the cause of the error may not then be apparent. A *trace* is a printout showing the values of selected variables after each statement is executed. It may help a programmer to discover errors.

These techniques are not usually very effective because they supply much data with little or no interpretation. More advanced methods are needed to reduce this data to an intelligible form.

Flowgraph analyzers are capable of detecting references to variables which are never initialized or never reused after receiving a value; these usually indicate errors. Test data generators are also available. Assertion checkers validate that given conditions are true at indicated points of a program. Automatic verification systems have been implemented for small languages [KING69] and symbolic execution has been proposed as a practical means for validating programs in a more complex language. The PSL/PSA system is an example of a tool for assisting in design and specification. Symbolic dumps and traces are generated with compilers like PL/C [CONW73] or PLUM [ZELK75]. Ramamoorthy and Ho [RAMA75] survey many of these tools.

Certification

Programs can be verified at several levels. Conway [CONW78] lists eight different verification conditions:

- A program contains no syntactic errors.
- A program contains no compilation errors or faults during program execution.
- There exist test data for which the program gives correct answers.
- For typical sets of test data, the program gives correct answers.
- For difficult sets of test data, the program gives correct answers.
- For all possible sets of data which are valid with respect to the problem specification, the program gives correct answers.
- For all possible sets of valid test data and all likely conditions of erroneous input, the program gives correct answers.
- For all possible input, the program gives correct answers.

Some people are optimistic that one day complete automatic program verification will be possible. Today's tools operate a posteriori, demonstrating that a given program works. Tomorrow's tools will also operate a priori, helping to develop programs which are correct before they are ever run. Such tools can reduce the amount of testing required for a completed project [DIJK76].

Verification techniques have the following general structures. A program is represented by a flowchart. Associated with each arc in the flowchart is a predicate, called an *assertion*. If A_i is the assertion associated with an arc entering statement S, and A_j is the assertion on the arc following the statement, then the statement "If A_i is true, and if statement S is executed, then assertion A_j will be true" must be proved (see Figure 7).

This process can be repeated for each statement in a program. If A_1 is the assertion immediately preceding the input node to the flowchart (that is, the initial asser-

FIGURE 7. Assertions A_1 and A_2 surround each statement of a program.

tion), and if A_n is the assertion at the exit node (for example, the final assertion), then the statement "If A_1 is true, and the program is executed, then A_n is true" will be the theorem that states that the program meets its specifications (A_1 and A_n) (see Figure 8). This approach was formalized by Hoare [HOAR69] who defined a set of axioms for determining the effects upon the assertions (preconditions and postconditions) by each statement type in a language. Thus verifying program correctness reduces to proving a theorem of the predicate calculus.

Certification technique development is still in a preliminary stage and does not meet the challenge of a modern large system. In addition, axiomatic certification is weak in the sense that the output assertion is proved true only if the program terminates. Axiomatic methods are incapable of proving termination. However, termination can often be proved informally by the programmer.

A typical approach to proving that program loops terminate is the following:

1) Find some number P that is always nonnegative within the loop.
2) Show that for each execution of the loop, P is decremented by at least a fixed amount.

If both conditions are always true, the loop must terminate before P becomes negative. A programmer who uses such rules, even informally, will seldom write nonterminating loops.

Consider this program fragment:

```
while x<y do
  ...
  x := x+1
  ...
end
```

Let quantity P be the expression $y-x$, and let $P(i)$ refer to the value of P during the ith execution of the loop. Because $x<y$ must be true for each next iteration, $y-x$ is always nonnegative and condition 1) is satisfied for each execution of the loop. Since the loop contains the statement $x := x+1$, $P(i+1) = P(i)-1$, satisfying condition 2). Therefore the loop must terminate.

Certification will not solve all our software problems, although it is an important tool. Gerhart and Yelowitz [GERH76] have shown that there are many published "certified" programs that contain errors. Even experts err.

Formal Testing

Goodenough and Gerhart [GOOD75] have clarified the concepts of testing. A *domain* is the set of permissible inputs to a program, and a *test* is a subset of the domain. A *testing criterion* specifies what is to be tested (for example, specifications, all statements, all paths).

A test is *complete* if the test meets all the requirements of the testing criterion, and a complete test is *successful* if the program gives correct results for each input in the test.

With these definitions, we can define program reliability and validity. A program is *reliable* if every found error is revealed by every complete test. A program is *valid* if every error is revealed by some complete test.

With these definitions, several important results can be proved. Among these are:

- If a program is both reliable and valid, then it is correct if and only if any complete test is also successful.
- The criterion "execute every path" is not valid; there exist programs all of whose test sets succeed, but which produce the wrong results for some input.

While this framework is somewhat technical and is not applicable to all programming, it is an important step in formalizing this area. We now have a basis for talking

FIGURE 8. Predicates A_1 and A_n specify input-output behavior of a program.

grammer's task is made easier when the computer does more work.

Structured Programming

A major development in facilitating the programming task is known as *structured programming*, which has been erroneously called "gotoless" programming. Fortunately, the debate about "to goto or not to goto" has mostly disappeared, and some clear ideas have emerged. The premise of structured programming is to use a small set of simple control and data structures with simple proof rules. A program then is built by nesting these statements inside each other. This method restricts the number of connections between program parts and thereby improves the comprehensibility and reliability of the program.

The **if-then-else**, **while-do**, and **sequence** statements are a commonly suggested set of control structures for this type of programming; however, there is nothing sacred about them. Knuth [KNUT74] has argued that the **goto** statement is irrelevant to the true goals of structured programming.

These simple control structures help programmers certify programs, even at an informal level. For example, a program can be represented as a function from its input data to its output data. Suppose $f(x)$ represents a segment of a program given by the following **if-then-else** statement:

$$\textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } h(x).$$

Because functions g and h are simpler than function f, their specifications should be simpler. If their specifications are known, the overall function f is defined by

$$f(x) = (p(x) \rightarrow g(x)) \wedge (\neg p(x) \rightarrow h(x)).$$

The programmer can express the formal definition of f in terms of the simpler definitions of g and h.

Languages such as ALGOL, PASCAL, and certain subsets of PL/I contribute to good programming practices by providing these facilities. In order to repair FORTRAN's lack of structure, over 50 preprocessors for translating well-structured pseudo-FORTRAN programs into true FORTRAN have been developed [REIF76]. An **if-then-else** has been added to the new FORTRAN-77 standard, although a general **while** is still missing from the language.

System Design

A technique related to structured programming is *top-down design*, in which a programmer first formulates a subroutine as a single statement, which is then expanded into one or two of the basic control structures mentioned earlier. At each level the function is expanded in increasingly greater detail until the resulting description becomes the actual source language program in some programming language.

Using this approach, also called *stepwise refinement* [WIRT71, WIRT74], the program is hierarchically structured and is described by successive refinements. Each refinement is interpreted by referring to other refinements of which it is a component. Concerning this method, Wirth states:

> I should like to stress that we should not be led to infer that actual program conception proceeds in such a well organized, straightforward, "topdown" manner. Later refinement steps may often show that earlier decisions are inappropriate and must be reconsidered. But this neat, *nested factorization* of a program serves admirably well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal, and even formal proofs of correctness. The emerging modularity is particularly welcome if programs have to be adjusted to changed or extended specifications. [WIRT74, p. 251]

Operating systems are often modeled as hierarchies of *abstract* or *virtual* machines [BRIN77]. At the lowest level of the system is the physical hardware. Each new level provides additional *capabilities*, or allowable functions on data, and hides some of the details of a lower level. For example, if one level accesses the paging hardware of the computer and provides a large virtual memory for all other processes, other abstract machines at higher levels can be implemented as if they had unlimited memory since this detail is controlled by a lower level.

The concept of a *program design language* (PDL) to aid in this development

about such concepts as reliability and correctness.

Mean Time Between Failure

While useful for focusing our attention, analogies with other engineering fields must be used with care. Reliability is one area of incomplete analogies. The concept of *mean time between failure (MTBF)* does not apply directly to software although it sometimes is used as if it does.

Systems built from physical components wear out; transistors fail; motors burn out; soldered joints break. This is also true for the hardware of the computer. However, the logical components of software are durable. A given program will always produce the same answer for the same input, as long as the hardware does not fail. When a software module "fails," it has been presented with an input that finally revealed an error present from the start.

The MTBF measures the time between revelations of errors. This, in turn, depends on the kinds of inputs presented. A compiler used only for short jobs from students may have a long MTBF; but if it is suddenly used for other applications, its MTBF may decrease sharply as unsuspected errors are exercised. A large MTBF can thus be interpreted only as an indication of possible reliability, not as a proof of it.

Error Days

Since formal certification of large classes of programs is still unattainable, techniques for estimating the validity of programs are still being considered. Most of these techniques measure the number of errors discovered, which are assumed to be representative of the total number of errors present in the system, and hence a measure of the reliability of the system.

Mills [MILL76] defines an *error day* as a measure stating that one error remains undetected in a system for one day. The total number of error days in a system is computed by summing, for each error, the length of time that error was in the system. A high error day count may reveal many errors (poor design) or long-lived errors (poor development).

The assumption is made that if a program is delivered with a low error day count, then there is a good chance that it will remain low during future use. However, two major problems remain before this measure can be widely used. First, it is difficult to discover when a particular error first entered a system. Second, it may be difficult to obtain such information from the developer of a delivered product.

Programming Techniques

Several authors have mentioned that the number of lines of code produced by a programmer in a given time tends to be independent of the language used. This implies that higher level languages enhance productivity [BROO75, HALS77]. This is true even though assembly language programs are potentially more efficient; their potential is seldom realized in practice.

The goals in developing early higher level languages were to be able to express clearly an algorithm and translate it into efficient machine language programs. The efficiency of the resulting code was all important. This led to some anomalies in FORTRAN arising from the structure of the IBM 704 for which it was developed (for example, the three-way branch of the arithmetic IF). ALGOL, which was developed as a machine-independent way of expressing algorithms, contained concepts whose implementation on conventional hardware was inefficient (e.g., recursion, call-by-name); this may explain why ALGOL is not widely used.

By the late 1960s it was accepted that the language should facilitate writing the program and that the machine should be designed to create an efficient run-time environment. Today there is a definite shift toward using the language to make programming and documentation easier and to produce reliable and correct software.

This does not mean, however, that efficiency is ignored today. Whereas PL/I permits the writing of simple programs whose execution time is quite long, PASCAL was designed to exclude constructs whose machine code is inefficient. Since hardware is less expensive than programmers, reliability has become a major factor: The pro-

has been defined [CAIN75]. This type of language contains two structures: "outer" syntax of basic statement types, such as **if-then-else**, **while**, and **sequence** for connecting components, and an "inner" syntax that corresponds to the application being designed. The inner syntax is English statement oriented, and is expanded, step by step, until it expresses the algorithm in some programming language. Figure 9 represents an example of a PDL design.

It should be noted here that PSL/PSA and PDL complement each other. PSL/PSA is a specifications tool that validates correct data usage between two modules (interfaces). A system like PDL is useful for describing a given module at any level of detail. Both PSL/PSA and PDL can contribute to success in a large project.

Even though designed from the top down, many systems are implemented from the bottom up. Low-level routines are first coded with drivers to test them; then new modules, using these low-level routines, are added, and the system is built up.

Top-down development is another technique for implementing hierarchically structured programs. Here the top-level routines are written first and lower level routines, called *stubs*, are written to interface with these. The stubs return control after printing a simple message and may return some fixed sample test values. The stub is eventually replaced by the full module which now includes calls to other stubs. In this manner an entire system can be gradually developed.

If used carefully, this technique can be valuable; however, the system's correctness is assumed, not proved, until the last stub has been replaced [DENN76a]. The documentation specifies the assumptions on each stub. For example, if

$$f(x) = \textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } h(x)$$

is a program fragment calling stubs g and h, then f will be correct only if the modules eventually replacing the stubs g and h are correct.

Via top-down development, a user sees the top-level interfaces in the system very early. He can then make changes relatively easily and soon. Another approach with the same goal is *iterative enhancement* [BASI 75]. Using this technique, a subset of the problem is first designed and implemented. This gives the user a running system early in the life cycle when changes are easier to make. This process is repeated to develop successively larger subsets until the final product is delivered.

Brooks [BROO75] believes that the first version of a system is always "thrown away," because the concrete specifications for a system are often not defined until the system is completed, a time when the initial product meets those specifications rather poorly. It is often cheaper and faster to rebuild a system from scratch than to try to modify an existing product to meet these specifications. However, a developer will often deliver such a modified system as a "pre-release" if a deadline is near and the purchaser is demanding results. The buyer then suffers with this version, replete with errors, until he throws it away or has the product rebuilt. Iterative enhancement can make rebuilding less chaotic since there is a running system (not meeting all the requirements) early in the development cycle.

```
max: PROCEDURE (list);
     /* Find maximum element in a list */
     DECLARE (maximum, next) integer;
     DECLARE list list of integers;
     maximum = first element of list;
     DO WHILE (more elements in list);
       next = next element of list;
       maximum = largest of next and maximum;
       END;
     RETURN (maximum);
     END max;
```

FIGURE 9. PDL of a program to find the largest element in a list (outer syntax is in upper case; inner syntax in lower case).

Performance Issues

The chosen algorithms and data structures have a much greater influence on program performance than code optimization or the programming language. Before choosing an algorithm, the programmer faces these questions:

- Can previously written software be used?
- If a new module must be written, what algorithms and data structures will give an efficient solution?

Programming languages usually include standard mathematical functions such as sine, logarithm, and square root. They give the programmer ready access to libraries of standard software packages. This allows the programmer to use results of previous work. In preparing programs for standard libraries, analysts have included many options in a single package. The effect can be a large cumbersome package which is inefficient because only a small part of it is applicable at any one time. This can be avoided by installing multiple versions of the module for each special case.

Many opportunities remain for more packaging and use of existing software. Difficulties in achieving this include:

- Identifying which standard algorithm to package. This is easier in mathematical areas such as statistical testing, integration, differentiation, and matrix computations than in many non-numerical areas such as business applications.
- Transporting and interfacing with packaged software. Some progress has been made with programs stored in read-only memories which plug into microprocessors, or with interface processors on computer networks. A major problem area lies in interfacing software directly to other software, since there are no conventions. Some help is afforded by such concepts as the "pipeline" in UNIX, which provides a general communications channel between programs [RITC74].

Algorithm Analysis

Sometimes the program specification is not changeable, and the analyst must find the best possible algorithm. Sometimes, however, the specifications can be altered to permit a more efficient solution. In some instances we can show that there are no algorithms guaranteed to be efficient in all cases; here approximate algorithms that are efficient in most cases but need not give exact solutions must be used.

The fast Fourier transform illustrates the most efficient form for computing the Fourier transform, a technique useful in waveform analysis [COOL65]. This transform is based on a finite set of points rather than on a complex integral which is harder to compute. Language analysis (parsing) in a compiler illustrates how changing the specification can permit a more efficient solution. Any string of N symbols in an arbitrary context-free language can be parsed in time of order $O(N**3)$ [YOUN67]; however, a programming language need not include all features of an arbitrary context-free language. PASCAL is an example of a language which can be parsed by a deterministic top-down parser in average time of order $O(N)$ [AHO72]. If we are free to set language specifications, we can choose the language and be rewarded with efficient compilers.

Many practical problems, such as job scheduling or network commodity flow, involve enumeration of a combinatorially large number of alternatives and selection of a best solution. In these cases it may be better to restrict the search for a suboptimal but good answer. We recommend the paper by Weide [WEID77] for a discussion of the issues and a state-of-the-art survey of algorithm analysis.

Efficiency

In many cases the results of algorithmic analysis are not extensive enough to help the programmer; thus we need to offer techniques which can help locate and remove sources of inefficiency. One such tool is an optimizing compiler which, for some languages, can yield significant improvements [LOWR69]. The value of such tools, however, is limited [KNUT71] and may be realized only for programs which are used often enough to justify the investment in optimization.

One of the most powerful aids is the *frequency histogram*, which reveals how often each statement of a program is executed. It is not unusual to find that 10% of the statements account for 80% of the execution time [KNUT71]. A programmer who concentrates on these "bottlenecks" in his algorithms can realize significant performance improvements at a minimum investment. This technique has been used in some interactive operating systems, such as

UNIX and MULTICS, which started out as high-level language operating systems. Bottlenecks have been replaced by assembly language routines in less than 20% of the system.

Theory of Specifications

One area of software engineering that is now under study is system specifications. The objective is to state the specifications early using a metalanguage. This places restrictions on the design and may help establish whether the specifications are met.

An early example of such a specification was the so-called "gotoless programming" [DIJK68, KNUT74]. It is properly called "structured programming." It restricts the form of statements a programmer may use, but this restriction contributes to comprehensibility and enhances a correctness proof.

A second set of such rules employs the concepts of levels of abstraction, information hiding, and module interfacing to restrict access to the internal structure of data. Parnas [PARN72] formalized these ideas which were standard practices of expert programmers. He defines data as a collection of logical objects, each with a set of allowable states. Procedures can then be written to hide the representation of these objects inside separate modules. The user manipulates the objects by calling the special procedures.

Several languages that facilitate the use of these concepts have been developed. Among these are EUCLID [POPE77], CLU [LISK77], and ALPHARD [WULF76]. These languages permit programmers to define abstract data types having the property to encapsulate the representation of the logical objects [LISK75]. When concurrency is an issue, the use of abstract objects must be controlled by synchronization (for example, locks, signals); in this case the abstract type managers are called *monitors*.

Another kind of specification consists of "higher order software axioms" (HOS) [HAMI76], which are a set of six axioms that specify allowable interactions among processes in a real-time system. One axiom prohibits a process from controlling its own execution, thereby ruling out recursion in a design. Another axiom states that no module controls its own input data space and is therefore unable to alter its input variables. While these axioms are not complete, they are a first step at formalizing specifications for system design.

SUMMARY

Boehm has stated seven principles that have helped organize the techniques discussed in this paper [BOEH76].

1) *Manage using a sequential life cycle plan.* This means to follow the software development life cycle outlined earlier. It allows for feedback which updates previous stages as the consequences of previous decisions become unknown. It encourages milestones to measure progress.

2) *Perform continuous validation.* Certify each new refinement of a module. Use walkthroughs and code reading. Display the hierarchical structure of the system clearly in all documentation.

3) *Maintain disciplined product control.* All output of a project—design documents, source code, user documentation, and so forth—should be formally approved. Changes to documents and program libraries must be strictly monitored and audited. Code reading, project reporting forms, librarians, a development library, and a project notebook all contribute to this goal.

4) *Use enhanced top-down structured programming.* PL/I and PASCAL have good control and data structures. Preprocessors exist which augment FORTRAN for these structures. Description techniques such as stepwise refinement, nested data abstractions, and data flow networks should be used.

5) *Maintain clear accountability.* Use milestones to measure progress, and a project notebook to monitor each individual's efforts.

6) *Use better and fewer people.* The chief programmer team, in which each individual is skilled and accountable for his actions, and good results are rewarded, aids in this effort.

7) *Maintain commitment to improve process.* Settle only for the best; strive for improvement. Be open to new develop-

ments in software engineering, but do not sacrifice reliability for modifiability while pursuing them.

Progress has been made in understanding how large-scale software systems are built, yet more needs to be done. Management aids must be improved and project control techniques developed. The role of software management is coming more to resemble that of engineering management in other disciplines. We can no longer afford costly mistakes when systems are so large and we depend so much on them. Most importantly, we must be patient; we need to gain experience on which future theories can rely.

ACKNOWLEDGMENTS

The author is indebted to Peter Denning for his detailed review and to the referees for their valuable comments on this paper. This work was partially supported by grant number DCR 74-11520-A01 from the National Science Foundation to the National Bureau of Standards.

REFERENCES

[AHO72] AHO, A.; AND ULLMAN, J. *Theory of parsing, translation, and compiling*. Prentice Hall, Inc., Englewood Cliffs, N. J., 1972.

[ANSI66] *American Standard FORTRAN*, American Natl. Standards Inst., x3.9-1966, March, 1966.

[ANSI76] *American Standard PL/1*, American Natl. Standards Inst., x.53-1976, Aug., 1976.

[BAKE72] BAKER, F. T. "Chief programmer team management of production programming," *IBM Syst. J.* **11**, 1 (1972), 56-73.

[BASI78] BASILI, V.; AND ZELKOWITZ, M. "Analyzing medium scale software development," *Third Int. Conf. Software Engineering*, 1978.

[BASI75] BASILI, V.; AND TURNER, A. J. "Iterative enhancement: a practical technique for software development," *IEEE Trans. Softw. Eng.* **1**, 4 (Dec. 1975), 390-396.

[BOEH75] BOEHM, B.; MCCLEAN, R.; AND URFRIG, D. "Some experience with automated aids to the design of large scale reliable software," *Int. Conf. on Reliable Software*, 1975, ACM, New York, pp. 105-113.

[BOEH77] BOEHM, B. "Seven basic principles of software engineering," in *Infotech state of the art report on software engineering techniques*, 1977, Infotech International Ltd., Maidenhead, UK, 1976.

[BRIN77] BRINCH, HANSEN P. *Architectures of concurrent programs*, Prentice Hall, Inc., Englewood Cliffs, N. J., 1977.

[BROO75] BROOKS, F. P. *The mythical man month*, Addison-Wesley Publ. Co., Reading, Mass., 1975.

[CAIN75] CAINE, S. H.; AND GORDON, E. K. "PDL—a tool for software design," in *Proc. 1975 AFIPS Natl. Computer Conf.*, Vol. 44, AFIPS Press, Montvale, N. J., pp. 271-276.

[CONW78] CONWAY, R. *A primer on disciplined programming*, Winthrop Publishers, Cambridge, Mass., 1978.

[CONW73] CONWAY, R.; AND WILCOX, T. "Design and implementation of a diagnostic compiler for PL/1," *Commun. ACM* **16**, 3 (March 1973), 169-179.

[COOL65] COOLEY, J. W.; AND TUKEY, J. W. "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* **19**, 90 (1965), 299-301.

[DAVI77] DAVIS, C. G.; AND VICK, C. R. "The software development system," *IEEE Trans. Softw. Eng.* **3**, 1 (Jan., 1977), 69-84.

[DENN76a] DENNING, P. J. "A hard look at structured programming," in *Infotech state of the art report on structured programming*, 1976, Infotech International Ltd., Maidenhead, UK, pp. 183-202.

[DENN76b] DENNING, P. J. "Fault tolerant operating systems," *Comput. Surv.* **8**, 4 (Dec. 1976), 359-389.

[DIJK68] DIJKSTRA, E. "GOTO statement considered harmful," *Commun. ACM* **11**, 3 (March 1968), 147-148.

[DIJK76] DIJKSTRA, E. *A discipline of programming*, Prentice Hall, Inc., Englewood Cliffs, N. J., 1976.

[DOLO76] DOLOTTA, T. A.; AND MASHEY, J. R. "An introduction to the programmer's workbench," in *Second Int. Conf. Software Engineering*, 1976, pp. 164-168.

[ENR61] "Everything about the Narrows Bridge is big, bigger, or biggest," *Eng. News Record* 166, June 29, 1961, 24-28.

[ENR64] "Narrows Bridge opens to traffic," *Eng. News Record* 173, Nov. 19, 1964, 33.

[ENR77] "Alaskan pipe cost probe hits snag," *Eng. News Record* 198, April 7, 1977, 14.

[FIFE77] FIFE, D. *Computer software management: a primer for project management and quality control*, Natl. Bureau of Standards, Inst. Computer Sciences and Technology, Special Publications, April 1977.

[GALL65] GALLAGHER P. F. *Project estimating by engineering methods*, Hayden Book Co., New York, 1965.

[GERH76] GERHART, S.; AND YELOWITZ, L. "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. Softw. Eng.* **2**, 3 (Sept. 1976), 195-207.

[GOOD75] GOODENOUGH, J. B.; AND GERHART, S. "Toward a theory of test data selection," *IEEE Trans. Softw. Eng.* **1**, 2 (June 1975), 156-173.

[HALS77] HALSTEAD, M. *Elements of software science*, Elsevier North Holland, Inc., New York, 1977.

[HAMI76] HAMILTON, M.; AND ZELDIN, S. "Higher order software—a methodology for defining software," *IEEE Trans. Softw. Eng.* **2**, 1 (March 1976), 9-32.

[HOAR69] HOARE, C. A. R. "An axiomatic basis for computer programming," *Commun.*

ACM 12, 10 (Oct. 1969), 576–580, 583.

[HUAN75] HUANG, J. C. "An approach to program testing," *Comput. Surv.* 7, 3 (Sept. 1975), 113–128.

[JEFF77] JEFFERY, S.; AND LINDEN, T. "Software engineering is engineering," in *IEEE Computer Science and Engineering Curricula Workshop*, 1977, IEEE, New York, pp. 112–115.

[KING69] KING, J. C. "A program verifier," PhD Dissertation, Computer Science Dept., Carnegie-Mellon Univ. Pittsburgh, Pa., 1969.

[KNUT71] KNUTH, D. "An empirical study of FORTRAN programs," *Softw. Pract. Exper.* 1, 2 (1971), 105–133.

[KNUT74] KNUTH, D. "Structured programming with statements," *Comput. Surv.* 6, 4 (Dec. 1974), 261–301.

[LISK75] LISKOV, B.; AND ZILLES, S. "Specification techniques for data abstractions," *IEEE Trans. Softw. Eng.* 1, 1 (1975), 9–19.

[LISK77] LISKOV, B.; SNYDER, A.; ATKINSON, R.; AND SCHAFFERT, C. "Abstraction mechanisms in CLU," *Commun. ACM* 20, 8 (Aug. 1977), 564–576.

[LOWR69] LOWRY, E. S.; AND MEDLOCK, C. W. "Object code optimization," *Commun. ACM* 12, 1 (Jan. 1969), 13–22.

[MILL76] MILLS, H. D. "Software development," *IEEE Trans. Softw. Eng.* 2, 4 (1976), 265–273.

[PARN72] PARNAS, D. L. "On the criteria for decomposing systems into modules," *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058.

[PARN75] PARNAS, D. L. "The influence of software structure on reliability," in *Int. Conf. Reliable Software*, 1975, pp. 358–362; (*ACM SIGPLAN Notices* 10, 6 June 1975).

[POPE77] POPEK, G. J.; HORNING, J. J.; LAMPSON, B. W.; MITCHELL, J. G.; AND LONDON, R. L. "Notes on the design of EUCLID," in *Proc. ACM Conf. Language Design for Reliable Software*, ACM, New York, 1977, pp. 11–18.

[PUTN77] PUTNAM, L.; AND WOLVERTON, R. *Quantitative management: software cost estimating*, (tutorial), IEEE Computer Society, Nov. 1977, IEEE, New York.

[RAMA75] RAMAMOORTHY, C. V.; AND HO, S. F. "Testing large software with automated software evaluation systems," *IEEE Trans. Softw. Eng.* 1, 1 (1975), 46–58.

[REIF76] REIFER, D. J. "The structured FORTRAN dilemma," *SIGPLAN Notices* 11, 2 (1976), 30–32.

[RITC74] RITCHIE, D. M.; AND THOMPSON, K. "The UNIX time-sharing system," *Commun. ACM* 17, 7 (July 1974), 365–375.

[TEIC77] TEICHROEW, D.; AND HERSHEY, E. A. "PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Softw. Eng.* 3, 1 (1977), 41–48.

[WALS77] WALSTON, C. E.; AND FELIX, C. P. "A method of programming measurements and estimation," *IBM Syst. J.* 16, 1 (1977), 54–73.

[WEID77] WEIDE, B. "A survey of analysis techniques for discrete algorithms," *Comput. Surv.* 9, 4 (Dec. 1977), 291–313.

[WEIN71] WEINBERG, G. M. *The psychology of computer programming*, Van Nostrand Reinhold, New York, 1971.

[WIRT71] WIRTH, N. "Program development by stepwise refinement," *Commun. ACM* 14, 4 (April 1971), 221–227.

[WIRT74] WIRTH, N. "On the composition of well-structured programs," *Comput. Surv.* 6, 4 (Dec. 1974), 247–259.

[WOLV74] WOLVERTON, R. W. "The cost of developing large scale software," *IEEE Trans. Comput.* 23, 6 (1974), 615–636.

[WULF76] WULF, W.; LONDON, R.; AND SHAW, M. "An introduction to the construction and verification of ALPHARD programs," *IEEE Trans. Softw. Eng.* 2, 4 (1976), 253–264.

[YOUN67] YOUNGER, D. "Recognition and parsing of context-free languages in time n^3," *Inf. Control* 10, 2 (1967), 189–208.

[ZELK75] ZELKOWITZ, M. V. "Third generation compiler design," in *ACM Natl. Comput. Conf.*, 1975, ACM, New York, pp. 253–259.

RECEIVED MARCH 14, 1977; FINAL REVISION ACCEPTED MARCH 7, 1978.

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

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

A P L I C A C I O N E S

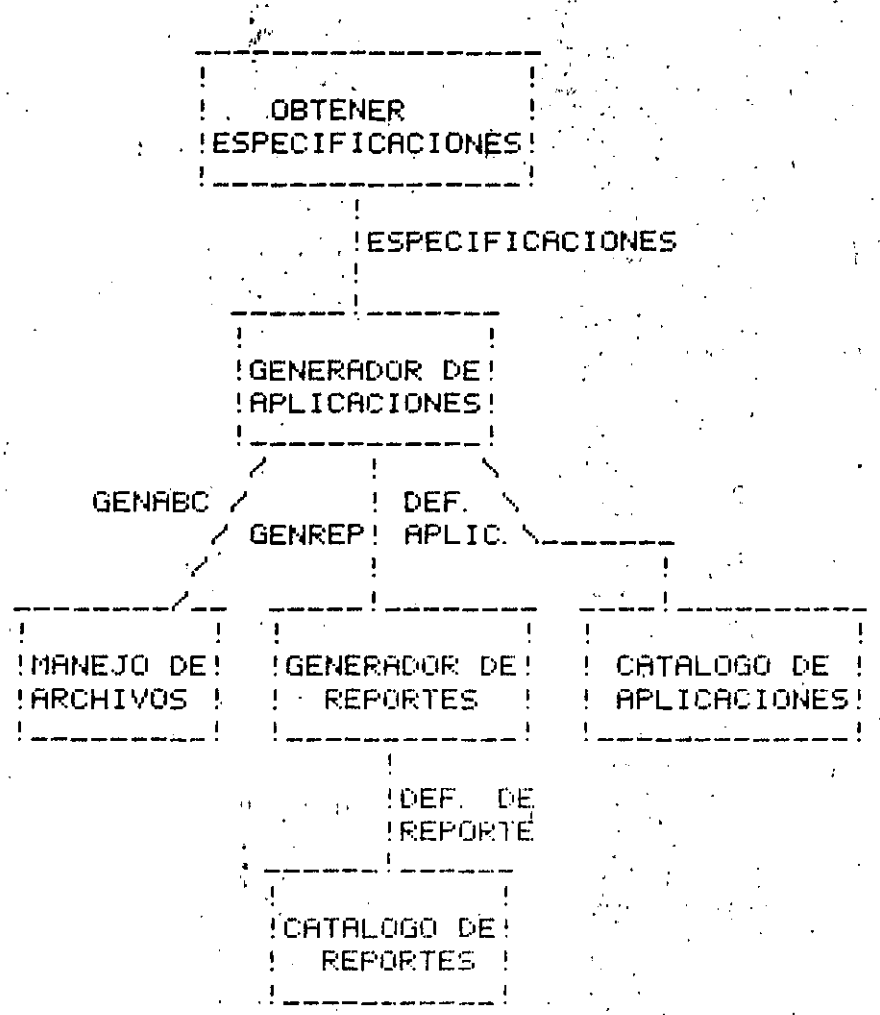
ING. SALVADOR PEREZ VIRAMONTES

JUNIO, 1984

INTRODUCCION.

COMO UN EJEMPLO DE APLICACION PRACTICA DE LAS TECNICAS ESTUDIADAS Y DE LO QUE PUEDE LOGRARSE CON LA APLICACION SISTEMATICA DE LAS MISMAS, SE ANALIZARAN LAS CARACTERISTICAS DE LOS PROGRAMAS GENERADOS POR UN GENERADOR DE APLICACIONES.

EL DIAGRAMA FUNCIONAL DEL SISTEMA ES EL SIGUIENTE:



ESPECIFICACIONES.

EL USUARIO FINAL INTERACTUA DIRECTAMENTE CON EL SISTEMA PROPORCIONAR LAS ESPECIFICACIONES DE LA APLICACION DESEADA EN LA FIGS. 1 Y 2 SE MUESTRAN LAS ESPECIFICACIONES QUE PUEDEN SERVIR PARA DEFINIR UNA APLICACION DE EMISION DE ETIQUETAS DE CORREO Y DE CONTROL DE REFERENCIAS BIBLIOGRAFICAS

PAG. 1

SISTEMA: SUSCRIPTORES DE LA REVISTA
ARCHIVO: SUSCR
VOLUMEN: SUSCR:

CAMPOS

CAMPO #	ID:	TIPO	LONGITUD	TITULO:	POSX	POSY
1	NUMSU	N	6	NUM. DE SUCRIPCION	0	4
2	TIPSU	A	1	TIPO	30	4
3	NOMB	S	25	NOMBRE	3	6
4	DIREC	A	25	DIRECCION	0	7
5	COL	S	20	COLONIA	2	8
6	CP	N	5	CP	7	10
7	LOC	A	25	LOCALIDAD	0	11
8	PAIS	A	15	PAIS	5	12
9	FEVEN	F	8	FECHA DE VENCIMIENTO	0	15
10	TELEF	A	10	TEL.	20	10

DESPLEGADO(S)

DESP #	POSX	POSY

LLAVE(S)

LLAVE #	CAMPO
1	NUMSU
2	FEVEN
3	CP

FIG. 1 ESPECIFICACIONES DE UNA APLICACION (PARTE 1)

NUM. DE SUCRIPCION: _____ TIPO: _

NOMBRE: _____

DIRECCION: _____

COLONIA: _____

CP: _____ TEL.: _____

LOCALIDAD: _____

PAIS: _____

FECHA DE VENCIMIENTO: _____

FIG. 1 ESPECIFICACIONES DE UNA APLICACION (PARTE 2)

PAG. 1
SISTEMA: REFERENCIAS BIBLIOGRAFICAS
ARCHIVO: BIB
VOLUMEN: BIBL1:

CAMPOS

CAMPO #	ID:	TIPO	LONGITUD	TITULO:	POSX	POSY
1	TITU	A	40		0	7
2	AUTOR	S	30	AUTOR(ES)	0	9
3	EDIT	S	30	PUBLICADO POR	0	11
4	FECHA	F	8	FEC. DE PUBLIC.	0	13
5	TIPO	A	2	TIPO	27	13
6	DISP	C	1	DISPONIBLE(S/N)	0	16
7	PAGS	N	4	NUM. DE PAGS.	20	16
8	TEMA	S	20	TEMA	0	18

DESPLEGADO(S)

DESP #		POSX	POSY
1	TITULO	0	5
2	AAMMDD	17	14

LLAVE(S)

LLAVE #	CAMPO
1	TITU
2	TEMA
3	AUTOR

FIG 2 ESPECIFICACIONES DE UNA APLICACION (PARTE 1)

5

TITULO

: _____

AUTOR(ES): _____

PUBLICADO POR: _____

FEC. DE PUBLIC.: _____ TIPO: __

AAMMDD

DISPONIBLE(S/N): _ NUM. DE PAGS.: ____

TEMA: _____

FIG 2. ESPECIFICACIONES DE UNA APLICACION

MENUS:

PARA OPERAR EL SISTEMA NO SE REQUIERE APRENDER COMANDOS SINO QUE EL SISTEMA VA GUIANDO AL USUARIO FINAL POR MEDIO DE MENUS Y LO UNICO QUE SE REQUIERE ES SELECCIONAR LA OPCION DESEADA EN UN MOMENTO DADO. ESTO SE ILUSTRA EN LA FIGURA 3.

PROGRAMA(S) GENERADOS:

A CONTINUACION SE PRESENTA UN LISTADO DEL PROGRAMA GENERADO PARA LA APLICACION DE CONTROL DE REFERENCIAS BIBLIOGRAFICAS.

```
PASCALGEN/U: REFERENCIAS BIBLIOGRAFICAS
---------------------------------------
BANCO DE DATOS:

     A. INCORPORAR REGISTROS
     B. ELIMINAR REGISTROS
     C. MODIFICAR REGISTROS
     D. CONSULTAR REGISTROS
     E. IMPRIMIR FORMAS
     F. FIN

SELECCIONE OPCION  [A-F]:
```

```
PASCALGEN/U: REFERENCIAS BIBLIOGRAFICAS
---------------------------------------
CONSULTAS POR:

     A. TIT
     B. TEMA
     C. AUTOR(ES)
     D. FIN DE LAS CONSULTAS

SELECCIONE OPCION  [A-D]:
```

FIG. 3 SISTEMA MANEJADO POR MENUS

```
  1:
  2: (*-----------------------------------*
  3:
  4:            PROGRAMA ESCRITO POR:
  5:
  6:              PASCALGEN/U  [2.3]
  7:
  8:      DERECHOS RESERVADOS SICISA,  1983
  9:
 10: (*-----------------------------------*)
 11:
 12: (*$S++*)
 13: (*$V-*)
 14: PROGRAM ABCBIB;
 15:
 16: USES
 17:     VIDEOCTRL
 18:    ,ISAM
 19:    ,REPORTLI
 20:    ;
 21:
 22: CONST
 23:
 24:     BVNAME  = 'ACCESS.DATA';
 25:     SYSNAME = 'REFERENCIAS BIBLIOGRAFICAS';
 26:     VOLNAME = 'BIBL1:';
 27:     CATNAME = 'BIB.DATA';
 28:     NOFKEYS = 3;
 29:     NFIELDS = 8;
 30:     N3FIELDS= 24;
 31:
 32: TYPE
 33:    CATREC  = PACKED RECORD
 34:                 TITU : STRING[40];
 35:               AUTOR : STRING[30];
 36:                EDIT : STRING[30];
 37:               FECHA : DATE;
 38:                TIPO : STRING[2];
 39:                DISP : BOOLEAN;
 40:                PAGS : INTEGER;
 41:                TEMA : STRING[20];
 42:              END;
 43:
 44:    BITVECTOR = PACKED ARRAY [0..7] OF BOOLEAN;
 45:
 46:
 47:    MENU=PACKED ARRAY[1..16] OF STRING[30];
 48:
 49: VAR
 50:    BVFILE  : FILE OF BITVECTOR;
 51:    CATFILE : FILE OF CATREC;
 52:    RCAT    : CATREC;
 53:    NRECS,
 54:    MAXCAT  : INTEGER;
 55:    KEYITEM,
 56:    SECKEY  : STRING;
 57:
 58:
 59: PROCEDURE OPENFILES ; FORWARD;
 60: PROCEDURE CLOSEFILES; FORWARD;
 61: PROCEDURE INSSEC    (I:INTEGER;RCAT:CATREC); FORWARD;
 62: PROCEDURE DELSEC    (I:INTEGER;RCAT:CATREC); FORWARD;
 63: PROCEDURE STATUS    (RECNO:INTEGER)        ; FORWARD;
 64: PROCEDURE DISPLAY   (JUSTDISP:BOOLEAN)     ; FORWARD;
 65: FUNCTION  GETKEYITEM : BOOLEAN; FORWARD;
 66: PROCEDURE PCHAIN    ; FORWARD;
 67:
 68:
 69: SEGMENT PROCEDURE INITIALIZE;
 70:   VAR
 71:      R:INTEGER;
 72:      BV:BITVECTOR;
 73:
```

```
  PROCEDURE CREATE;
    VAR
       I:INTEGER;

    PROCEDURE CREATEISAM;
      BEGIN
      IF NOT ISCREATE(1,CONCAT(VOLNAME,'ISO1'),MAXCAT,NOFKEYS-1) THEN
         ABORT('ISCREATE');
      END;

    BEGIN
    PROMPTAT(0,1,'CREACION DE ARCHIVOS');
    CLEARFROM(3);
    MAXCAT:=0;
    GETVAINT(5,15,4,'NUMERO DE REGISTROS: ',MAXCAT,0,MAXINT,FALSE);
    NULLREC;
    REWRITE(CATFILE,CONCAT(VOLNAME,CATNAME));
    PROMPTAT(5,15,'VERIFICANDO REGISTRO:');
    FOR I:=0 TO MAXCAT DO
       BEGIN
       CATFILE^:= RCAT;
       PUT(CATFILE);
       CAPTAINT(15,16,4,'',I,TRUE);
       END;
    CLOSE(CATFILE,LOCK);
    MCSAVE(MAXCAT,CONCAT(VOLNAME,CATNAME));
    FOR I:=0 TO 7 DO BV[I]:=FALSE;
    REWRITE(BVFILE,CONCAT(VOLNAME,BVNAME));
    SEEK(BVFILE,0);
    BVFILE^:=BV;
    FOR I:=0 TO MAXCAT DIV 8 DO PUT(BVFILE);
    BV[0]:=TRUE;
    BVFILE^:=BV;
    SEEK(BVFILE,0);
    PUT(BVFILE);
    CLOSE(BVFILE,LOCK);
    CREATEISAM;
    END; (* CREATE *)

  BEGIN
  SCREENHDR;
  IF NOT MOUNTDK(CONCAT('2,',VOLNAME,'/')) THEN
     PCHAIN ;
  R:=FILEOK(CONCAT(VOLNAME,BVNAME));
  IF R=10 THEN
     CREATE;
  OPENFILES;
  MCLOAD(MAXCAT,CONCAT(VOLNAME,CATNAME));
  END; (* INITIALIZE *)

SEGMENT PROCEDURE ADDREC;
  VAR
      I:INTEGER;
      B:BOOLEAN;

  FUNCTION SEARCHBV:INTEGER;
    VAR
       I,J : INTEGER;
    BEGIN
    FOR I:=0 TO MAXCAT DIV 8 DO
       BEGIN
       SEEK(BVFILE,I);
       GET(BVFILE);
       J:=0;
       WHILE ((BVFILE^[J]) AND (J < 7)) DO J:=J+1;
       IF NOT BVFILE^[J] THEN
          BEGIN
          SEARCHBV:=I * 8 + J;
          EXIT(SEARCHBV);
          END;
       END;
    END;

  FUNCTION SPACE: BOOLEAN;
    BEGIN
    IF NRECS < MAXCAT THEN
       SPACE := TRUE
     ELSE
       BEGIN
       ERRMSG('NO HAY ESPACIO PARA MAS REGISTROS',TRUE);
       SPACE := FALSE;
       END;
    END;
```

```
157:
158:   BEGIN
159:   (*$R ISAM*)
160:   IF SPACE THEN
161:      BEGIN
162:      B := FALSE;
163:      CLEARFROM(3);
164:      REPEAT
165:        STATUS(NRECS+1);
166:        NULLREC;
167:        DISPLAY(TRUE);
168:        IF NOT GETKEYITEM THEN
169:           BEGIN
170:           IF B THEN
171:              BEGIN
172:              CLOSEFILES;
173:              OPENFILES
174:              END;
175:           EXIT(ADDREC);
176:           END;
177:        I:= ISFIND(1,KEYITEM,0);
178:        IF I>0 THEN
179:           ERRMSG('YA EXISTE ESE REGISTRO',TRUE)
180:         ELSE
181:           BEGIN
182:           REPEAT
183:             CLEARFROM(23);
184:             DISPLAY(FALSE);
185:           UNTIL YES(0,23,'ESTA CORRECTO (S/N)? ');
186:           I:= SEARCHBV;
187:           IF NOT ISINSERT(1,KEYITEM,I,0) THEN ABORT('ISINSERT');
188:           WRITEREC(I);
189:           ONOFFBV(I,TRUE);
190:           NRECS:= ISPOP(1);
191:           INSSEC(I,RCAT);
192:           B := TRUE;
193:           CLEARFROM(23);
194:           END;
195:      UNTIL NOT SPACE;
196:      END;
197:   END;
198:
199: SEGMENT PROCEDURE DELETEREC;
200:   VAR
201:      I:INTEGER;
202:   BEGIN
203:   (*$R ISAM*)
204:   CLEARFROM(3);
205:   IF NOT GETKEYITEM THEN EXIT(DELETEREC);
206:   I:= ISFIND(1,KEYITEM,0);
207:   IF I=0 THEN
208:      BEGIN
209:      ERRMSG('NO EXISTE ESE REGISTRO',TRUE);
210:      EXIT(DELETEREC)
211:      END;
212:   STATUS(I);
213:   READREC(I);
214:   DISPLAY(TRUE);
215:   IF YES(0,23,'ESTA CORRECTA LA ELIMINACION (S/N)? ') THEN
216:      BEGIN
217:      IF NOT ISDELETE(1,KEYITEM,I,0) THEN ABORT('ISDELETE');
218:      ONOFFBV(I,FALSE);
219:      DELSEC(I,RCAT);
220:      NRECS:= ISPOP(1);
221:      CLOSEFILES;
222:      OPENFILES
223:      END;
224:   END; (* DELETEREC *)
225:
226:
227: SEGMENT PROCEDURE CHANGEREC;
228:   VAR
229:      I:INTEGER;RC1:CATREC;B:BOOLEAN;
230:   BEGIN
231:   (*$R ISAM*)
232:   B:=FALSE;
233:   REPEAT
234:     CLEARFROM(3);
235:     IF NOT GETKEYITEM THEN
236:        BEGIN
237:        IF B THEN
238:           BEGIN
239:           CLOSEFILES;
```

```
240:          { OPENFILES
241:            END;
242:        EXIT(CHANGEREC);
243:        END;
244:      I:= ISFIND(1,KEYITEM,0);
245:      IF I=0 THEN
246:        ERRMSG('NO EXISTE ESE REGISTRO',TRUE)
247:       ELSE
248:        BEGIN
249:        STATUS(I);
250:        READREC(I);
251:        DISPLAY(TRUE);
252:        RC1:= RCAT;
253:        PROMPTAT(0,23,'OPRIMA <RETORN> PARA NO CAMBIAR');
254:        DISPLAY(FALSE);
255:        IF YES(0,23,'ESTA CORRECTO (S/N)? ') THEN
256:          BEGIN
257:          DELSEC(I,RC1);
258:          INSSEC(I,RCAT);
259:          WRITEREC(I);
260:          B:=TRUE;
261:          END;
262:        END;
263:    UNTIL FALSE;
264:    END; (* CHANGEREC *)
265:
266:  SEGMENT PROCEDURE INQUIREREC;
267:    VAR
268:       I,K:INTEGER;
269:       C:CHAR;
270:       MCONS:MENU;
271:
272:    PROCEDURE INITMCONS;
273:      BEGIN
274:      MCONS[1]:='';
275:      MCONS[2]:='TEMA';
276:      MCONS[3]:='AUTOR(ES)';
277:      MCONS[NOFKEYS+1]:='FIN DE LAS CONSULTAS';
278:      END;
279:
280:    FUNCTION GETSOMEKEY(K:INTEGER):BOOLEAN;
281:      BEGIN
282:      GETSOMEKEY:= TRUE;
283:      CASE K OF
284:         1:GETSOMEKEY:= GETKEYITEM;
285:         2:BEGIN
286:           RCAT.TEMA:= '';
287:           CAPTASTR(0,18,20,'TEMA:',RCAT.TEMA,FALSE);
288:           IF RCAT.TEMA = '' THEN GETSOMEKEY:=FALSE;
289:           END;
290:         3:BEGIN
291:           RCAT.AUTOR:= '';
292:           CAPTASTR(0,9,30,'AUTOR(ES):',RCAT.AUTOR,FALSE);
293:           IF RCAT.AUTOR = '' THEN GETSOMEKEY:=FALSE;
294:           END;
295:       END;
296:      END;
297:
298:
299:    FUNCTION LOOKFORKEY(K:INTEGER):INTEGER;
300:      VAR
301:         I:INTEGER;L:LINTEGER;
302:      BEGIN
303:      IF K=1 THEN
304:         LOOKFORKEY:=ISFIND(1,KEYITEM,0)
305:       ELSE
306:         BEGIN
307:         CASE K OF
308:           2:    COMPACT(RCAT.TEMA,SECKEY);
309:           3:    COMPACT(RCAT.AUTOR,SECKEY);
310:          END;
311:         LOOKFORKEY := ISFIND(1,SECKEY,K-1);
312:         END;
313:      END;
314:
315:    BEGIN
316:    INITMCONS;
317:    REPEAT
318:      PROMPTAT(0,3,'CONSULTAS POR:');
319:      K:=MENUSELECT(5,22,NOFKEYS+1,MCONS);
320:      IF K=NOFKEYS+1 THEN EXIT(INQUIREREC);
321:      CLEARFROM(3);
322:      NULLREC;
```

```
323:     DISPLAY(TRUE);
324:     IF GETSOMEKEY(K) THEN
325:        BEGIN
326:        I:= LOOKFORKEY(K);
327:        WHILE I>0 DO
328:          BEGIN
329:          STATUS(I);
330:          READREC(I);
331:          DISPLAY(TRUE);
332:          PROMPTAT(0,22,'OPRIMA: <RETURN>     <ESC>      <I>');
333:          PROMPTAT(0,23,'  PARA: CONTINUAR    TERMINAR   IMPRIMIR');
334:          C:=GETCHAR([CHR(13),CHR(27),'I','i']);
335:          IF C=ESC THEN
336:             I:=0
337:           ELSE
338:             BEGIN
339:             IF (C='I') OR (C='i') THEN
340:                PRINTFORM(FALSE);
341:             I:= ISNEXT(1,K-1);
342:             IF I > 0 THEN
343:                BEGIN
344:                SCREENHDR;
345:                PROMPTAT(0,1,MCONS[K]);
346:                END;
347:             END;
348:          END;
349:        END;
350:   UNTIL FALSE;
351:   END; (* INQUIREREC *)
352:
353:
354: PROCEDURE OPENFILES;
355:   BEGIN
356:   RESET(BVFILE,CONCAT(VOLNAME,BVNAME));
357:   RESET(CATFILE,CONCAT(VOLNAME,CATNAME));
358:   IF NOT ISOPEN(1,CONCAT(VOLNAME,'ISO1'),FALSE) THEN
359:     ABORT('ISOPEN');
360:   NRECS := ISPOP(1);
361:   END;
362:
363:
364: PROCEDURE CLOSEFILES;
365:   BEGIN
366:   CLOSE(CATFILE,LOCK);
367:   CLOSE(BVFILE,LOCK);
368:   ISCLOSE(1);
369:   END;
370:
371:
372: PROCEDURE INSSEC(* I:INTEGER;RCAT:CATREC *);
373:   VAR
374:      L:LINTEGER;
375:   BEGIN
376:   COMPACT(RCAT.TEMA,SECKEY);
377:   IF NOT ISINSERT(1,SECKEY,I,1) THEN ABORT('ISINSERT');
378:   COMPACT(RCAT.AUTOR,SECKEY);
379:   IF NOT ISINSERT(1,SECKEY,I,2) THEN ABORT('ISINSERT');
380:   END;
381:
382:
383: PROCEDURE DELSEC(*I:INTEGER;RCAT:CATREC*);
384:   VAR
385:      L:LINTEGER;
386:   BEGIN
387:   COMPACT(RCAT.TEMA,SECKEY);
388:   IF NOT ISDELETE(1,SECKEY,I,1) THEN ABORT('ISDELETE');
389:   COMPACT(RCAT.AUTOR,SECKEY);
390:   IF NOT ISDELETE(1,SECKEY,I,2) THEN ABORT('ISDELETE');
391:   END;
392:
393: FUNCTION GETKEYITEM(*:BOOLEAN*);
394:   VAR
395:      L:LINTEGER;
396:   BEGIN
397:   RCAT.TITU:= '';
398:   CAPTASTR(0,7,40,':',RCAT.TITU,FALSE);
399:   IF RCAT.TITU = '' THEN GETKEYITEM:=FALSE;
400:   KEYITEM:= RCAT.TITU;
401:   END;
402:
403: PROCEDURE STATUS(*RECNO:INTEGER*);
404:   VAR
405:      S:STRING[4];
```

```
406:    BEGIN
407:    GOTOXY(WIDTH-11,1);
408:    FORM(RECNO,3,0,TRUE,S);
409:    WRITE(S,'/');
410:    FORM(ISPOP(1),3,0,TRUE,S);
411:    WRITE(S,'/');
412:    FORM(MAXCAT,3,0,TRUE,S);
413:    WRITE(S);
414:    END;
415:
416:  PROCEDURE DISPLAY(*JUSTDISP:BOOLEAN*);
417:    VAR
418:       B:BOOLEAN;
419:    BEGIN
420:    WITH RCAT DO
421:      BEGIN
422:      IF JUSTDISP THEN
423:      CAPTASTR(0,7,40,':',TITU,JUSTDISP);
424:      CAPTASTR(0,9,30,'AUTOR(ES):',AUTOR,JUSTDISP);
425:      CAPTASTR(0,11,30,'PUBLICADO POR:',EDIT,JUSTDISP);
426:      CAPTAFECHA(0,13,'FEC. DE PUBLIC.:',FECHA,JUSTDISP);
427:      CAPTASTR(27,13,2,'TIPO:',TIPO,JUSTDISP);
428:      B:= DISP;
429:      CAPTABOOL(0,16,'DISPONIBLE(S/N):',B,JUSTDISP,TRUE);
430:      DISP:= B;
431:      CAPTAINT(20,16,4,'NUM. DE PAGS.:',PAGS,JUSTDISP);
432:      CAPTASTR(0,18,20,'TEMA:',TEMA,JUSTDISP);
433:      IF NOT JUSTDISP THEN EXIT(DISPLAY);
434:      PROMPTAT(0,5,'TITULO');
435:      PROMPTAT(17,14,'AAMMDD');
436:      END;
437:    END;
438:
439:  PROCEDURE PCHAIN;
440:    BEGIN
441:    CHAIN('X*MAESTRO^',TRUE);
442:    EXIT(PROGRAM);
443:    END;
444:
445:  PROCEDURE SELECTMODULE;
446:    VAR
447:       K:INTEGER;
448:       MMAIN:MENU;
449:    BEGIN
450:    MMAIN[1]:='INCORPORAR REGISTROS';
451:    MMAIN[2]:='ELIMINAR REGISTROS';
452:    MMAIN[3]:='MODIFICAR REGISTROS';
453:    MMAIN[4]:='CONSULTAR REGISTROS';
454:    MMAIN[5]:='IMPRIMIR FORMAS';
455:    MMAIN[6]:='FIN';
456:    K:= MENUSELECT(5,22,6,MMAIN);
457:    CASE K OF
458:       1:ADDREC;
459:       2:DELETEREC;
460:       3:CHANGEREC;
461:       4:INQUIREREC;
462:       5:PRINTFORM(TRUE);
463:       6:BEGIN CLOSEFILES;PCHAIN;END;
464:       END;
465:    END;
466:
467:
468:  BEGIN (* PROGRAMA *)
469:  (*$N+*) (*$R VIDEOCTRL*)
470:  (* INITIALIZE *);
471:  REPEAT
472:    SCREENHDR;
473:    SELECTMODULE;
474:  UNTIL FALSE;
475:  END.
```

LOS PROGRAMAS GENERADOS TIENEN LAS SIGUIENTES CARACTERISTICAS.

+ UNA ESTRUCTURA JERARQUICA PERFECTAMENTE DEFINIDA.

+ PROGRAMADOS ESTRUCTURADAMENTE

+ UN BAJO ACOPLAMIENTO ENTRE LOS COMPONENTES DEL SISTEMA. PUEDEN ELIMINARSE COMPONENTES (PARA PRUEBAS POR EJEMPLO O PARA REEMPLAZARLOS POR OTROS NUEVOS) Y EL RESTO DEL SISTEMA PODRA OPERAR PARCIALMENTE.

+ LOS MODULOS PRESENTAN EN GENERAL UNA ALTA COHESION.

+ LA UTILIZACION DE LA ESTRUCTURA DE BLOQUES QUE POSEE PASCAL PARA UNA MEJOR UTILIZACION DE LA MEMORIA.

+ UN ESTILO DE PROGRAMACION

+ USO DE IDENTIFICADORES ADECUADOS QUE AYUDAN A QUE EL PROGRAMA SEA AUTODOCUMENTADO

+ USO DE CONSTANTES PARA FACILITAR LOS CAMBIOS EN CASO NECESARIO.

+ EL USO DE MUCHAS FUNCIONES DE LIBRERIA

+ FACIL DE MODIFICAR

+ UN DESARROLLO TOP-DOWN

LA APLICACION DE ESTE CONCEPTO PUEDE VERSE CLARAMENTE EN LAS RUTINAS DE LIBRERIA UTILIZADAS POR LOS PROGRAMAS GENERADOS. ALGUNAS DE LAS FUNCIONES QUE LLEVAN A CABO DICHAS RUTINAS SON LAS SIGUIENTES:

+ RUTINAS DE MANEJO DE LA PANTALLA

+ CAPTURA Y VALIDACION DE LA INFORMACION

+ FORMATEO DE NUMEROS Y FECHAS

+ APERTURA Y CIERRE DE ARCHIVOS

+ CREACION DE ARCHIVOS

+ ACTUALIZACION DE ARCHIVOS (INSERCION, ELIMINACION Y MODIFICACION DE REGISTROS)

+ CONSULTA DE REGISTROS (EN FORMA SECUENCIAL Y RANDOM) UTILIZANDO LAS LLAVES PREVIAMENTE DEFINIDAS.

+ RUTINAS DE BUSQUEDA FONETICA Y LLAVES PARCIALES.

+ MANEJO DE CONDICIONES PARA SELECCION DE REGISTROS.

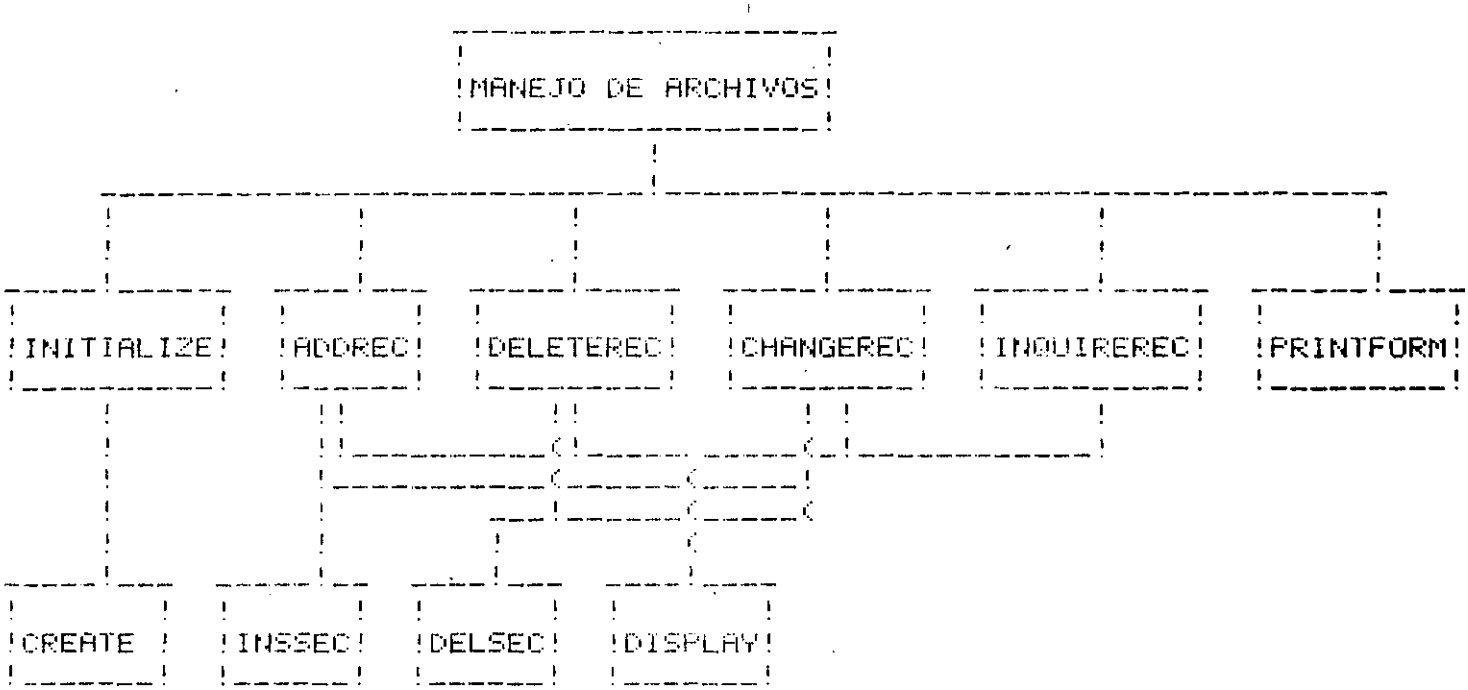
+ DIRECCIONAMIENTO DE LA SALIDA DE LOS PROGRAMAS

+ MANEJO DE LOS REPORTES (CONTADOR DE LINEA, CAMBIO DE PAGINA, ETC)

+ MANEJO DE ERRORES

ESTRUCTURA JERARQUICA:

LA ESTRUCTURA JERARQUICA DE UNO DE LOS PROGRAMAS GENERADOS SE PUEDE APRECIAR EN LA SIGUIENTE CARTA DE ESTRUCTURA:



LA ESTRUCTURA JERARQUICA COMPLETA SE MUESTRA EN LA FIGURA 4.

ESTRUCTURA JERARQUICA
ABCBIB

```
ADDREC
        CLOSEFILES
        DISPLAY
        GETKEYITEM
        INSSEC
        NULLREC
        ONOFFBV
        OPENFILES
        SEARCHBV
        SPACE
        STATUS
        WRITEREC
CHANGEREC
        CLOSEFILES
        DELSEC
        DISPLAY
        GETKEYITEM
        INSSEC
        OPENFILES
        READREC
        STATUS
        WRITEREC
DELETEREC
        CLOSEFILES
        DELSEC
        DISPLAY
        GETKEYITEM
        ONOFFBV
        OPENFILES
        READREC
        STATUS
FINIS
        CLOSEFILES
INITIALIZE
        CREATE
                CREATEISAM
                NULLREC
        INITMCONS
        OPENFILES
        SCREENHDR
INQUIREREC
        DISPLAY
        GETNEXTKEY
        GETSOMEKEY
                GETKEYITEM
        LOOKFORKEY
        MENUSELECT
                SCREENHDR
        NULLREC
        READREC
        SCREENHDR
        STATUS
MENUSELECT
        SCREENHDR
PRINTFORM
        GETSPECS
        PRINT
                BLANKS
                NEXTFORM
```

FIG. 4

PARA CONTROLAR LA CALIDAD DE LOS PROGRAMAS GENERADOS Y LOGRAR UNA ALTA CONFIABILIDAD DICHOS PROGRAMAS. ALGUNOS DE LOS ELEMENTOS UTILIZADOS SON:

+ EL DESARROLLO DE HERRAMIENTAS ADECUADAS (EJ. CROSS REFERENCE, ANALIZADOR ESTRUCTURAL)

+ DICCIONARIO DE DATOS

+ REVISION FRECUENTE DE LOS AVANCES

+ PRUEBAS EXHAUSTIVAS DE MODULOS, QUE POSEAN UN BAJO ACOPLAMIENTO Y UNA ALTA COHESION.

+ ESTANDARIZACION

LA ESTRUCTURA ESTATICA DEL PROGRAMA SE MUESTRA EN LA FIGURA 3:

LA FORMA EN LA CUAL LAS RUTINAS ESTAN RELACIONADAS EN CUANTO A QUE RUTINAS UTILIZAN A CUALES SE PUEDE VER EN LA FIGURA 6.

PRODUCTIVIDAD:

LA PRODUCTIVIDAD EN LA ELABORACION DEL SOFTWARE SE LOGRA CON:

+ COMUNICACION E INTERCAMBIO DE EXPERIENCIAS.

+ APRENDIZAJE DEL TRABAJO DE OTROS

+ RESPALDOS ADECUADOS.

+ CONTROL SOBRE LOS CAMBIOS

+ NO REINVENTANDO CONTINUAMENTE LAS COSAS.

ESTRUCTURA ESTATICA


```
ABCBIB
        OPENFILES
        CLOSEFILES
        NULLREC
        ONOFFBV
        INSSEC
        DELSEC
        READREC
        WRITEREC
        STATUS
        DISPLAY
        SCREENHDR
        SEARCHBV
        SPACE
        GETKEYITEM
        GETNEXTKEY
        GETSOMEKEY
        LOOKFORKEY
        MENUSELECT
        INITIALIZE
                CREATE
                        CREATEISAM
                INITMCONS
        PRINTFORM
                NEXTPAGE
                NEXTFORM
                GETSPECS
                BLANKS
        ADDREC
        DELETEREC
        CHANGEREC
        INQUIREREC
        FINIS
        SCREENHDR
        OPENFILES
        CLOSEFILES
        NULLREC
        ONOFFBV
        SEARCHBV
        INSSEC
        DELSEC
```

FIG. 5

```
PROCEDURE    LLAMADO
                 FOR

ADDREC
             ABCBIB

BLANKS
             PRINT

CHANGEREC
             ABCBIB

CLOSEFILES
             ADDREC
             CHANGEREC
             DELETEREC
             FINIS

CREATE
             INITIALIZE

CREATEISAM
             CREATE

DELETEREC
             ABCBIB

DELSEC
             CHANGEREC
             DELETEREC

DISPLAY
             ADDREC
             CHANGEREC
             DELETEREC
             INQUIREREC

FINIS
             ABCBIB

GETKEYITEM
             ADDREC
             CHANGEREC
             DELETEREC
             GETSOMEKEY

GETNEXTKEY
             INQUIREREC

GETSOMEKEY
             INQUIREREC

GETSPECS
             PRINTFORM

INITIALIZE
             ABCBIB

INITMCONS
             INITIALIZE

INQUIREREC
             ABCBIB
```


FIG. 6

```
INSSEC
            ADDREC
            CHANGEREC

LOOKFORKEY
            INQUIREREC
MENUSELECT
            ABCBIB
            INQUIREREC
            MENUSELECT

NEXTFORM
            PRINT

NULLREC
            ADDREC
            CREATE
            INQUIREREC

ONOFFBV
            ADDREC
            DELETEREC

OPENFILES
            ADDREC
            CHANGEREC
            DELETEREC
            INITIALIZE

PRINT
            PRINTFORM

PRINTFORM
            ABCBIB
            INQUIREREC

READREC
            CHANGEREC
            DELETEREC
            INQUIREREC

SCREENHDR
            INITIALIZE
            INQUIREREC
            MENUSELECT

SEARCHBV
            ADDREC

SPACE
            ADDREC

STATUS
            ADDREC
            CHANGEREC
            DELETEREC
            INQUIREREC

WRITEREC
            ADDREC
            CHANGEREC
```


FIG 6 CONT

DEMOSTRACION DE LOS PROGRAMAS GENERADOS.

A CONTINUACION SE MUESTRAN EN LAS FIGURAS 7 A 11, ALGUNOS DE LOS DESPLEGADOS QUE APARECEN DURANTE LA EJECUCION DE LOS PROGRAMAS GENERADOS.

EN SEGUIDA SE MOSTRARA EL DISENO DE UNA APLICACION DESDE SU INICIO UTILIZANDO EL GENERADOR DE APLICACIONES LLAMADO PASCALGEN/U.

FINALMENTE SE MOSTRARA LA OPERACION DE LOS PROGRAMAS GENERADOS Y LAS FACILIDADES QUE OFRECEN.

INSTALACIONES ELECTRICAS INDUSTRIALES I

B I B L I O G R A F I A

MAYO, 1985.

BIBLIOGRAFIA RECOMENDADA PARA EL CURSO DE INSTALACIONES ELECTRICAS INDUSTRIALES

1. "IEEE RECOMMENDED PRACTICE FOR ELECTRIC POWER DISTRIBUTION FOR INDUSTRIAL PLANTS" (IEEE STD. 141-1976. RED BOOK)

2. "IEEE RECOMMENDED PRACTICE FOR ELECTRIC POWER SYSTEMS IN COMMERCIAL BUILDINGS" (IEEE STD. 241-1974, GRAY BOOK)

3. "INDUSTRIAL POWER SYSTEMS HANDBOOK",DONALD BEEMAN, EDICION 1955

4. "I.E.S. LIGHTING HANDBOOK 1981" VOL. 1: REFERENCE VOLUME; VOL.2: APPLICATION VOLUME

5. "NORMAS TECNICAS DE EL REGLAMENTO DE INSTALACIONES ELECTRICAS". SEPAFIN, 1982.

6. NATIONAL ELECTRIC CODE, 1984.

7. "IEEE RECOMMENDED PRACTICE FOR PROTECTION AND COORDINATION OF INDUSTRIAL AND COMMERCIAL POWER SYSTEMS" (IEEE STD. 242-1975, BUFF BOOK).

8. "IEEE RECOMMENDED PRACTICE FOR POWER SYSTEM ANALYSIS" (IEEE STD. 399-1980, BROWN BOOK).

9. "IEEE RECOMMENDED PRACTICE FOR GROUNDING INDUSTRIAL AND COMERCIAL POWER SYSTEMS" (IEEE STD 142-1982, GREEN BOOK).

10. "IEEE RECOMMENDED PRACTICE FOR EMERGENCY AND STANDBY POWER SYSTEMS FOR INDUSTRIAL AND COMMERCIAL APPLICATIONS" (IEEE STD. 446-1980 ORANGE BOOK).

11. "IEEE 80 GUIDE FOR SAFETY GROUNDING IN AC SUBSTARIONS", 1976. (PARA SISTEMAS DE TIERRAS, NORMA IEEE)

12. "INDUSTRIAL POWER SYSTEMS DATA BOOK, GENERAL ELECTRIC,SCHENECTADY, N.Y. USA.

13. ELECTRICAL TRANSMISSION AND DISTRIBUTION REFERENCE BOOK, WESTERN GHOUSE.

14. STANDARD HANDBOOK FOR ELECTRICAL ENGINEERS. ARCHER E. KNOWLTON 9TH. EDITION

15. IEEE TRANSACTIONS ON "INDUSTRY APPLICATIONS" (SE NECESITA INSCRIBIRSE AL IEEE)

16. IEEE TRANSACTIONS ON "POWER APPARATUS AND SYSTEMS" (SE NECESITA INSCRIBIRSE AL IEEE)

17. APPLIED PROTECTIVE RELAYING, 1979, WESTINGHOUSE

DIRECTORIO DE ALUMNOS DEL CURSO "METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION" IMPARTIDO EN ESTA DIVISION DEL 31 DE MAYO AL 28 DE JUNIO .

1.- AGUIRRE TELLEZ GUILLERMO
S. C. T.
ANALISTA
LAZARO CARDENAS No. 567
COL. NARVARTE
03700 MEXICO, D.F.
519-40-99

CHIAPAS No. 49
COL. COSMOPOLITA
DELEGACION AZCAPOTZALCO
02670 MEXICO, D.F.
355-28-14

2.- ALCEDA HERNANDEZ LUIS J.
ING. SIST. DE TRANSP. METROPOLITANO
JEFE DEPTO. PROYECTOS VIALES
LEGARIA No. 252
COL. PENSIL
399-69-22

JAIME TORRES BODET No. 58 EDIF. B-103
COL. CUAUHTEMOC
DELEGACION CUAUHTEMOC
06400 MEXICO, D.F.

3.- ALVARADO VAZQUEZ MOISES
UNIV. AUT. DEL EDO. DE MORELOS
PROGRAMADOR
AV. UNIVERSIDAD No. 1001
COL. CHAMILPA
13-26-44

PRIV. E. ZAPATA No. 99
CUERNAVACA, MOR.

4.- ANGELES DELGADO ELIGIO
S. A. R. H.
TECNICO MEDIO
GOMEZ FARIAS No. 2-3er. PISO
COL. TABACALERA
DELEGACION CUAUHTEMOC
566-38-37

TULTEPEC No. 2 MZA. 99
COL. ALTAVILLA
ECATEPEC DE MORELOS
EDO. DE MEXICO
569-02-23

5.- ANGULO MARCIAL NOEL
DIREC. GRAL. EST. SUP. O.S.E.P.
JEFE DEPTO. INFORMACION
INSURGENTES SUR No. 1378
COL. SAN ANGEL
550-90-00 ext. 194

CALLE SALVADOR SANCHEZ COLIN No. 19-B
COL. PROVIDENCIA
DELEGACION AZCAPOTZALCO
02440 MEXICO, D.F.
561-97-50

6.- ARANDA CRUZ MIGUEL
TESORERIA DEL DISTRITO FEDERAL
ANALISTA TECNICO
NINOS HEROES Y DR. LAVISTA
COL. DOCTORES

SINALOA No. 11
COL. PEÑON DE LOS BAÑOS
DELEGACION VENUSTIANO CARRANZA
15520 MEXICO, D.F.
762-79-26

7.- BAIZ RISENDIZ NABOR FABIAN
UNAM
PROF. ASIGNATURA "A"
CIUDAD UNIVERSITARIA

LAZARO CARDENAS No. 201-9
COL. DOCTORES
DELEGACION CUAUHTEMOC
06720 MEXICO, D.F.
761-63-25

8.- BALDERRAMA CAMPOS HECTOR
BACARDI Y CIA., S.A. DE C.V.
ANALISTA PROGRAMADOR
AUTOPISTA MEX. QRO. KM. 34.5
TULTITLAN EDO. DE MEXICO
565-01-77

CORDILLERA No. 48
SECC. ATLANTA
CUAUTITLAN IZCALLI EDO. DE MEXICO
565-01-77

9.- BAUTISTA GONZALEZ GUILLERMO
S. C. T.
PROGRAMADOR
LAZARO CARDENAS No. 576
COL. NARVARTE
530-30-60

NORTE 88 A No. 5221
COL. GERTRUDIS SANCHEZ
DELEGACION GUSTAVO A. MADERO
551-18-94

10.- BELMONTE SAMUEL
S. C. T.

11.- BENAVIDES HUERTO LUIS ANTONIO
SEGUROS DE MEXICO
SUBGERENTE PROCEDIMIENTOS AUTOMAT.
AV. UNIVERSIDAD No. 1311
COL. DEL VALLE
DELEGACION COYOACAN
03300 MEXICO, D.F.
534-00-34

RANUTO DEL ARCO No. 40-403
COL. GIRASOLES
DELEGACION COYOACAN
04920 MEXICO, D.F.

12.- BRITO ALDAY CELSO
DIREC. GRAL. CONSTRUC. OPERAC. HIDRAUL.
ANALISTA DE SISTEMAS
SAN ANTONIO ABAD No. 231
COL. OBRERA
DELEGACION MIGUEL HIDALGO
588-31-21

VICENTE GUERRERO No. 25-2
COL. APATLACO
DELEGACION IZTAPALAPA
09140 MEXICO, D.F.
657-88-80

13.- CARRASCO MARMOLEJO ALBERTO
S. C. T.
JEFE DEPTO.
AV. SAN FRANCISCO No. 1625-5o. PISO
COL. DEL VALLE
DELEGACION BENITO JUAREZ
03120 MEXICO, DF.
651-80-35 ext. 290

TONALA No. 396-65
DELEGACION BENITO JUAREZ
03100 MEXICO, D.F.
687-11-74

14.- CASTILLO GARCIA FERNANDO JAVIER
SEDUE
JEFE DE OFICINA
REFORMA No. 20-5o. PISO
COL. JUAREZ
DELEGACION CUAUHTEMOC
06040 MEXICO, D.F.
535-07-91

AV. UNIVERSIDAD No. 1960 EDIF. 30-101
DELEGACION COYOACAN
04310 MEXICO, D.F.
658-39-65

15.- DAVILA NARVAEZ CONSTANTINO
S. C. T.
JEFE AREA ATENCION A USUARIOS
JOSE DE TERESA No. 176
COL. TLACOPAC
DELEGACION ALVARO OBREGON
550-16-99

CALLE AZUCENA MZ. 22 65-11
DELEGACION ALVARO OBREGON
01840 MEXICO, D.F.
548-90-29

16.- CASTILLO VARGAS GUILLERMO
S. C. T.
TECNICO ANALISTA
LAZARO CARDENAS No. 567
COL. ALAMOS
549-26-26

CRUZ AZUL No. 227
COL. INDUSTRIAL
DELEGACION GUSTAVO A. MADERO
07800 MEXICO, D.F.
537-07-64

17.- CORDERO HERRERA JOSE MANUEL
SERV. ASES. EMPRESARIALES
ANALISTA PROGRAMADOR
NICOLAS SAN JUAN No. 225
COL. DEL VALLE
DELEGACION BENITO JUAREZ
03100 MEXICO, D.F.
536-50-83

18.- DIAZ ENRIQUEZ JORGE
DISTRIBUIDORA ANAHUAC, S.A.
ANALISTA PROGRAMADOR
CALLE HEROES DE CHURUBUSCO No. 40
COL. TACUBAYA
DELEGACION MIGUEL HIDALGO
11820 MEXICO, D.F.
271-75-99

OYAMALES No. 632
DELEGACION AZCAPOTZALCO
394-35-03

19.- ESPINOLA ARZATE GERARDO
UNIVERSIDAD AUTONOMA DE MORELOS
OPERADOR
AV. CHAMILPA No. 1001
CUERNAVACA, MOR.
13-26-44

PRIV. DE ORION No. 12
TEJALPA, MORELOS
13-26-47

20.- FERNANDEZ SARDA JUAN CARLOS
UNIVERSIDAD LA SALLE
COORDINADOR DEL CENTRO COMPUTO
BENJAMIN FRANKLIN No. 57
COL. CONDESA
DELEGACION CUAUHTEMOC
516-99-60

MOLINOS DEL CAMPO No. 15-7
COL. SAN MIGUEL CHAPULTEPEC
DELEGACION MIGUEL HIDALGO
11850 MEXICO, D.F.
516-26-13

21.- FLORES LOPEZ ANGEL
COMISION FEDERAL DE ELECTRICIDAD
INGNEIERIA ELECTRICA
RODANO No. 14
COL. CUAUHTEMOC
DELEGACION CUAUHTEMOC
06598 MEXICO, D.F.
553-71-33 ext. 2161

PASEO DE LA REFORMA No. 730-1501
EDIF. ZACATECAS
TLATELOLCO
DELEGACION CUAUHTEMOC
06900 MEXICO, D.F.
782-09-40

22.- FRAGOSO VILLARUEL MA. CONCEPCION
S. A. R. H.
ANALISTA
GOMEZ FARIAS No. 2
COL. TABACALERA
535-67-27

CALLE 2 No. 88
COL. INDEPENDENCIA
DELEGACION BENITO JUAREZ
672-73-12

23.- GOMEZ ALPUCHE L. FERNANDO
S. A. R. H.
JEFE DEPTO.
GOMEZ FARIAS No. 2
COL. TABACALERA
DELEGACION CUAUHTEMOC
535-67-27

CALZ. LA VIGA No. 1147
DELEGACION IXTACALCO
08830 MEXICO, D.F.
579-63-09

24.- GONZALEZ GOMEZ MONICA
COLEGIO BACHILLERES EDO. MICHOACAN
JEFE CENTRO COMPUTO
COR. AMADO CAMCHO ESQ. MANUEL M. PONCE
COL. ACUEDUCTO
MORELIA, MICH.
482-49

AV. LAZARO CARDENAS No. 1015
MORELIA, MICH.
58040

25.- GONZALEZ SANTILLAN JORGE
S.E.D.U.E.
ANALISTA FINANCIERO
REFORMA No. 20-5o. PISO
COL. CENTRO

GELATI No. 15-14
DELEGACION MIGUEL HIDALGO
15-90-36

26.- GONZALEZ MARQUEZ LUIS ARTURO
S. C. T.
PROYECTISTA TERRACERIAS
XOLA Y LAZARO CARDENAS
COL. NARVARTE
DELEGACION BENITO JUAREZ
30-30-00 ext. 410

UNION POSTAL No. 80
DELEGACION BENITO JUAREZ
03410 MEXICO, D.F.
696-78-30

27.- GUTIERREZ JEREZ JORGE
DIREC. GRAL. SIST. DE A.P. Y ALCANT.
JEFE DE OFICINA
REFORMA No. 20-5o. PISO
COL. CENTRO
DELEGACION CUAUHTEMOC
06040 MEXICO, D.F.
535-07-91

E. ZAPATA No. 15
COL. B. SIERRA
DELEGACION MAGDALENA CONTRERAS
10380 MEXICO, D.F.
595-34-50

28.- HERNANDEZ MECINAS PABLO
IPESA
JEFE DE PROYECTO
SAN LORENZO No. 153-6o. PISO
COL. DEL VALLE
DELEGACION BENITO JUAREZ
03100 MEXICO, D.F.
559-17-45

QUIOSCO No. 14
EDO. DE MEXICO
575-40-77

29.- HERNANDEZ TORRES CARLOS
SERV. ASESORIAS EMPRESARIALES
PROGRAMADOR
NICOLAS SAN JUAN No. 225
COL. DEL VALLE
DELEGACION BENITO JUAREZ
03100 MEXICO, D.F.
536-50-83

30.- JAIMES AVILES J. JESUS
S. E. D. U. E.
ANALISTA ESPECIALIZADO
REFORMA No. 20
COL. JUAREZ
DELEGACION CUAUHTEMOC
535 01 91

PORTALES No. 291
AMPL. V. VILLADA
CD. NETZAHUALCOYOTL
EDO. DE MEXICO

31.- LOPEZ CANO HECTOR D.
DIRECCION GRAL. DE OBRAS MARITIMAS
PROVIDENCIA No. 807-4o. PISO
COL. DEL VALLE
687 76 80

E-4 No. 10
UNIDAD STO. DOMINGO
DELEGACION ALVARO OBREGON
277 79 73

32.- LOPEZ REYES MARCOS
BACARDI Y CIAL, S.A. DE C.V.
ANALISTA
KM. 34 1/2
CARR. MEXICO-QUERETARO
565 01 77

PISCIS No. 14
VALLE DE LA HACIENDA
CUAUTITLAN, EDO. DE MEXICO

33.- MANRIQUEZ AYALA FRANCISCO
S. E. D. U. E.
ANALISTA TECNICO
REFORMA No. 20
COL. JUAREZ
DELEGACION CUAUHTEMOC
591 13 16

RETORNO No. 44
TLANEPANTLA
EDO. DE MEXICO
368 14 96

34.- MARIN MARTINEZ JOSE ALBERTO
SRIA. DE COMUNICACIONES Y TRANSPORTES
ANALISTA DE SISTEMAS
EJE CENTRAL LAZARO CARDENAS No. 567
COL. NARVARTE
530 30 60 ext. 572

CORREGIDORA No. 78-6
COL. SANTA ANITA
DELEGACION IXTACALCO
03800 MEXICO, D.F.
650 38 62

35.- MARTINEZ MEDINA E. GABRIEL
DIRECCION GRAL. DE OBRAS MARITIMAS
JEFE DE SECCION DE VOCACION
PORTUARIA COMERCIAL
PROVIDENCIA No. 807
COL. CONDESA

INSURGENTES NTE. No. 240-21
DELEGACION CUAUHTEMOC
06400 MEXICO, D.F.

36.- MAYA ARADILLAS LUCIO
COMISION FEDERAL DE ELECTRICIDAD
INGENIERO ELECTRICISTA
RODANO No. 14
COL. CUAUHTEMOC
DELEGACION CUAUHTEMOC
6598 MEXICO, D.F.
553 71-33 ext. 2161 y 22 31

AV. 565 No. 147
DELEGACION GUSTAVO A. MADERO
760 10 79

37.- MEJIA ACEVEDO MIGUEL ANGEL
S. A. R. H.
PROGRAMADOR
GOMEZ FARIAS No. 2-1er PISO
COL. TABACALERA
DELEGACION CUAUHTEMOC
06030 MEXICO, D.F.
535 67 27

LA TOLTECA No. 37
COL. INDUSTRIAL
DELEGACION GUSTAVO A. MADERO
07800 MEXICO, D.F.
537 94 11

38.- MEJIA GOMEZ ABRAHAM
TESORIRIA DEL D.F.F.
SUPERIVISOR DE ANALISTA
DR. LAVISTA No. 144 - ESQ. NIÑOS HEROES
COL. DOCTORES
DELEGACION CUAUHTEMOC
588 10 37

ORIENTE 251 No. 157-5 A
COL. AGIRCOLA ORIENTAL
DELEGACION IZTACALCO
08500 MEXICO, D.F.
763 86 92

39.- MIRANDA DEL VALLE JOSE FRANCISCO
DIRECCION GRAL. DE AEROPUERTOS
JEFE DE SECCION
CHIAPAS No. 121
COL. ROMA

MANZANA 28 LOTE 331
COL. SECC No. 16
DELEGACION TLALPAN
01480 MEXICO, D.F.

40.- MONROY G. MARIA CRISTINA
S. C. T.
DIRECCION GENERAL DE TELECOMUNICACIONES
ANALISTA PROGRAMADOR
AV. EJE CENTRAL
530 60 09

RETORNO 65 No. 5
COL. AVANTE
DELEGACION COYOACAN
677 40 35

41.- MONTOYA SANCHEZ EDUARDO
S. C. T.
JEFE DEL PROYECTO DE PLANEACION
AV. SN. FRANCISCO 1626-7o. PISO ALA SUR
COL. DEL VALLE
DELEGACION BENITO JUAREZ
534 79 22

IXTLEMEMELIXTLE No. 43
COACALCO
EDO. DE MEXICO 55700
874 82 09

42.- MORALES PAVAN MARIANO ARMANDO
PETROLEOS MEXICANOS
ANALISTA P.
MARINA NACIONAL No. 329

CAROLINA No. 56
COL. INDUSTRIAL
DELEGACION GUSTAVO A. MADERO
07800 MEXICO, D.F.
781 01 80

43.- MORENO ORTIZ RAFAEL
S E D U E
ANALISTA TECNICO
REFORMA No. 77-11° PISO
COL. SAN RAFAEL
DELEGACION CUAUHTEMOC
535 72 90 y 546 91 82

MARIANO AZUELA No. 120-304
DELEGACION CUAUHTEMOC
06400 MEXICO, D.F.

44.- NATIVIDAD RICO JUAN CARLOS
CENTRO DE INFORMACION CIENTIFICA
Y HUMANISTICA
ANALISTA PROGRAMADOR
CIUDAD UNIVERSITARIA
DELEGACION COYOACAN
550 59 05

MITLA No. 214-1
DELEGACION BENITO JUAREZ
03020 MEXICO, D.F.
590 72 81

45.- NINO PARROQUIN OSCAR R.
D. D. F.

46.- ORTEGA LUJAN JORGE LUIS
S. A. R. H.
SUBJEFE DE DEPARTAMENTO
REFORMA No. 133
COL. SAN RAFAEL
DELEGACION CUAUHTEMOC
566 89 24

JOSE T. CUELLAR No. 122-E
DELEGACION CUAUHTEMOC

47.- ORTIZ CADENA FROYLAN
DIRECCION GRAL. DE OBRAS MARITIMAS
INGENIERO ESPECIALIZADO (ANALISTA)
PROVIDENCIA No. 807-4o. PISO
COL. DEL VALLE
DELEGACION BENITO JUAREZ
523 45 38

SABINO No. 60-13
COL. STA. MA. LA RIBERA
DELEGACION CUAUHTEMOC
06400 MEXICO, D.F.
541 12 33

48.- OSORIO GARCIA ALVARO
DIRECCION GRAL. DE OBRAS MARITIMAS
JEFE DE OFICINA
PROVIDENCIA No. 807
COL. DEL VALLE
687 76 80

EDIFICIO JESUS TERAN B-606
UNIDAD TLATELOLCO
DELEGACION CUAUHTEMOC

49.- PICAZO SALINAS JORGE
DIRECCION GRAL. DE CARRETERAS FEDERALES
ANALISTA PROGRAMADOR
CENTRO SCOP BASAMENTO
AV. UNIVERSIDAD Y XOLA
COL. NARVARTE
DELEGACION BENITO JUAREZ
03028 MEXICO, D.F.
519 92 21

BARRANCA DE PILARES No. 1
COL. LAS AGUILAS
DELEGACION ALVARO OBREGON
01710 MEXICO, D.F.

50.- PONCE MEDINA AGUSTIN
TESORERIA DEL D.D.F.
JEFE DE OFICINA
DR. LAVISTA No. 144
COL. DOCTORES
588 10 37

MANZANA 3 LOTE 18
REMANENTE IV
DELEGACION IZTAPALAPA
797 87 81

51.- PLANCARTE VAZQUEZ JOSE ALBERTO
INSTITUTO MEXICANO DEL PETROLEO
INGENIERO
EJE CENTRAL LAZARO CARDENAS No. 152
SAN BARTOLO ATEPEHUACAN
DELEGACION GUSTAVO A. MADERO
07730 MEXICO, D.F.
567 66 00 ext. 20504

SUR 103 No. 1406-4
COL. AERONAUTICA MILITAR
DELEGACION VENUSTIANO CARRANZA
15970 MEXICO, D.F.

52.- RENTERIA MORA MARIANO
TESORERIA DEL D.F.
JEFE DE OFICINA

PROL. RAMON Y CAJAL No. 160
DELEGACION IXTACALCO
08220 MEXICO, D.F.
590 27 88

53.- RESENDIZ MORALES FERNANDO
SRIA. DE DESARROLLO URBANO Y ECOLOGIA
ANALISTA DE INFORMACION
REFORMA No. 77
COL. REVOLUCION
DELEGACION CUAUHTEMOC
06030 MEXICO, D.F.
535 54 18

PATZCUARO No. 9
COL. SAN JAVIER
TLANEPANTLA
54030 EDO. DE MEXICO
390 46 60

54.- REYES HERNANDEZ PATRICIA GEORGINA
INDUSTRIALIZADORA DE ACEITES, S.A.
INGENIERO DE PROCESOS
COL. INDUSTRIAL BENITO JUAREZ
QUERETARO, QRO.
4 55 12

TANCOYOL No. 17
FRACC. MISIONES
QUERETARO, QRO.

55.- REYES ORTEGA JORGE
TESORERIA DEL D.F.
ANALISTA DE ORGANIZACION Y METODOS
DR. LAVISTA No. 144
COL. DOCTORES
DELEGACION CUAUHTEMOC
588 10 37

MADRID No. 25
COL. DEL CARMEN
DELEGACION COYOACAN
04100 MEXICO, D.F.
524 05 62

56.- RICO MEJIA GABRIEL
S. E. D. U. E.
JEFE DE OFICINA
REFORMA No. 20-5o. PISO
COL. JUAREZ
DELEGACION CUAUHTEMOC
06040 MEXICO, D.F.
535 07 91

HUICHAPAN No. 53
COL. MICHOACANA
DELEGACION VENUSTIANO CARRANZA

57.- RODRIGUEZ LUGO SERGIO DANIEL
DIRECCION GRAL. DE CARRETERAS FEDERALES
ANALISTA PROGRAMADOR
XOLA Y AV. UNIVERSIDAD
COL. NARVARTE
DELEGACION BENITO JUAREZ
03028 MEXICO, D.F.
519 92 21

BODOQUEPA No. 30-3
COL. LA ASUNCION
DLELGACION XOCHIMILCO
16040 MEXICO, D.F.
676 70 26

58.- RODRIGUEZ VALENZUELA EDMUNDO
PROGRAMA UNIVERSITARIO DE COMPUTO
JEFE DE SECCION
CIUDAD UNIVERSITARIA
CIRCUITO EXTERIOR
DELEGACION COYOACAN
04000 MEXICO, D.F.
550 52 15

MIGUEL ANGEL DE QUEVEDO No. 972-6
COYOACAN
04000 MEXICO, D.F.
544 04 01