ADMINISTRACION DE PROYECTOS

SOFTWARE

ANEXOS

M.en C.MARCIAL PORTILLA ROBERTSON

FEBRERO, 1982

ADMINISTRACION DE PROYECTOS DE SOFTWARE

Marcial Portilla Robertson.

. " El problema del Software"

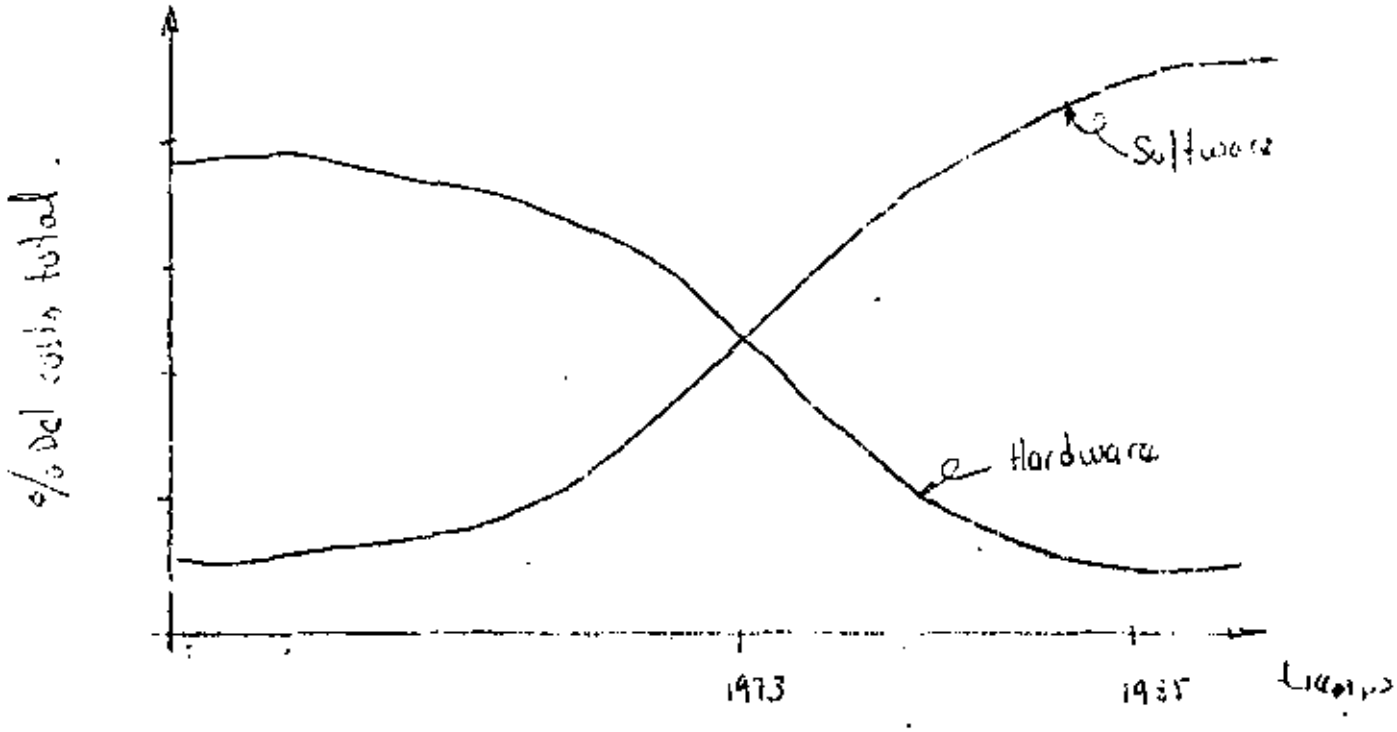
. Estándares
. Nuevas herramientes : Automatización

Todo proyecto de Software tiene: (como todo proyecto).

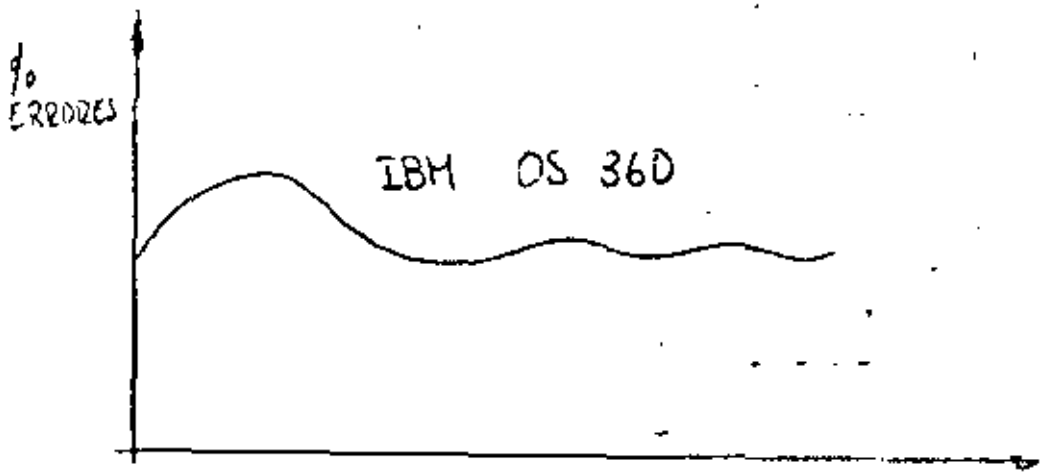
. Costos de desarrollo...
. Confiabilidad del producto
. Soporte

COSTO DE SOFTWARE.

En los E.U. en 1976 se gastan 80 billones de dólares en escribir software, en 1980 en probable que la cifra llegue a 100 billones de dólares (II-EE procedings sept. 1980).



En 1985 70% Soft
30% Hard



$1\ BUG = 0.96 \rightarrow BUG$ NUEVO

El peor ——

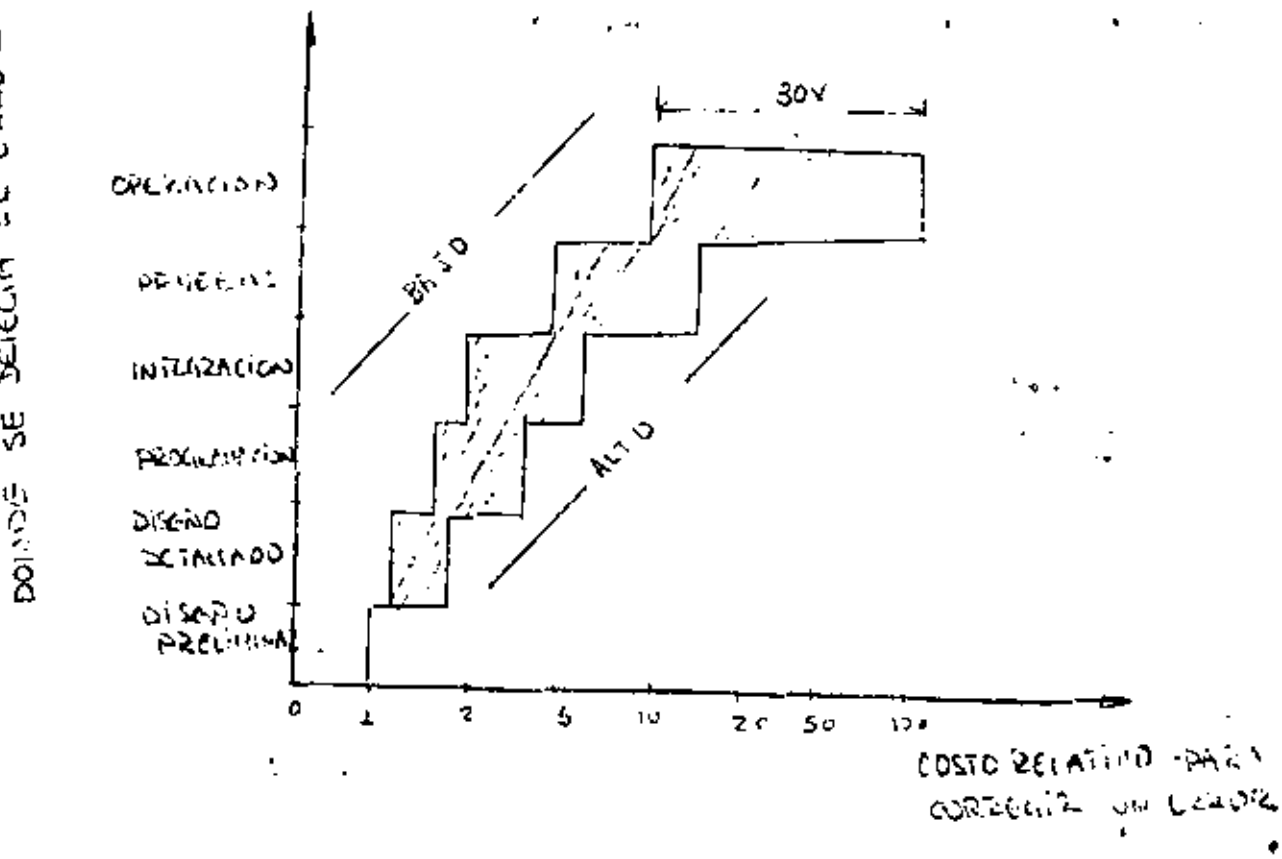
cuesta $4,000 el corregir una línea de código en el 'aeroespacio' y cuenta $75 escribirla.

POR QUE AUMENTAN LOS COSTOS DE SW. (Además de la inflación).

- Resulta difícil diseñar, construir, y probar un programa de computadora, que satisface totalmente al clienta o al usuario.

- Muy frecuentamente el usuario y el analista Subestiman la dificultad del problema.

- La mayoría de los errores ocurren antes de empezar a escribir código, y esto ocurre por que se tienen requerimientos muy pobres, y se logran detectar muy pocos de estos errores cuando ocurren.



Como se ve en la figura anterior aproximadamente 72% de los -- errores se detectan en la construcción o el mantenimiento, y - este error puede costar hasta 30 veces mas corregirlo, que lo- que hubiese costado si se corrige 'donde se origna'.

Los errores y las omisiones tienden a multiplicarse con el avance del proyecto. Requisitos incompletos, el no entender el -- problema, decisiones vagas, pruebas pobres son las mayores causantes de los problemas en dos palabras: PLANEACION POBRE.

¿ QUE ES LA INGENIERIA DE SOFTWARE?

Que soluciones ó sugerencias tiene la industria para disminuir los costos de SW y mejorar la calidad.

() MOVILIDAD DE SOFTWARE.

- Compiladores compatibles con los nuevos sistemas.
- Traductores para los Leng. ensambladores.
- Compatibilidad en el desarrollo de herramientas <HW y SW.>
- ESTANDARIZACION.
- Nuevos lenguajes ADA, ACTOR, ACTOR, C, PASCAL.

() REDUCCION DE COSTOS EN MANTENIMIENTO Y DESARROLLO.

- Mejores lenguajes de algo nivel
  - PASCAL
  - C
  - ACTOR
  - ADA
  - FORTRAN 77
- Maximizar la reutilización de software.
- Procurar cambiar el software de hoy por hadware mañana (en PROM, PLA'ER).
- "Desarrollo" de herramientas para el desarrollo y mantenimiento de SW.

() MAYOR CONTROL DE DESARROLLO.

- Programas verificadores de estándares (i.e. CODE CHE -- CKER).
- Programas para seguir actividades (CPM PERT)
- Uso de técnicas de "ingeniería de software"
- Diseño y programación estructurado.
- Mejorar las técnicas de "MANAGEMENT".

' SOFTWARE DE CALIDAD ' ?

- Antes de continuar hablando de la producción de SW, hay que definir que hace
  B UENO
  SW B ONITO
  B ARATO.

- Debido a que los humanos no se pueden comunicar a la "perfección" se requieren técnicas y metodologías en al 'management' y el control de calidad en el SW a producir.

CARACTERISTICAS DE SOFTWARE DE CALIDAD.

CONBIABLE. - Los programas se comportan tal y como fueron diseñados o c ncebidos?- como se comportan los programas cuando ocurren vi laciones en la entrada?

MANTENIMIENTO. - Se puede leer los programas? existen manuales y documentos en cristiano? se han aislado las funciones de un programa en módulos diferenciables, de manera que un programador pueda fácilmente llevar a cabo el cambio o corregir el error.

PORTABLE. - Se los programas transportar fácilmente a otros sistemas? son independientes de un HW o con O.S. específicos.

PROBARSE. - Son los programas de estructura simple, de manera que los algoritmos, l E/S puedan fácilmente PROBARSE.

EFICIENCIA. - (Economía) - los programas utilizan en forma racional los recursos del sistema.

EL PROCESO DE DESARROLLO DE SW.

1.- ANALISIS DE LOS REQUERIMIENTOS Y DEFINICION.

( ) Entender el problema desde el punto de vista del usuario y mercado.
( ) Documentarlo 'modular' y precisamente.
( ) Utilizar un lenguaje común con el usuario.

2.- ESPECIFICACION FUNCIONAL

( ) Transformar las especificaciones del problema a DESCRIPCION DE FUNCIONES IMPLEMENTABLES.
( ) De inir 'que' se va a hacer.
( ) Utilizar una lista (CHECKDOINT) con el usuario.
( ) Utilizar la lista para el diseño y las pruebas.

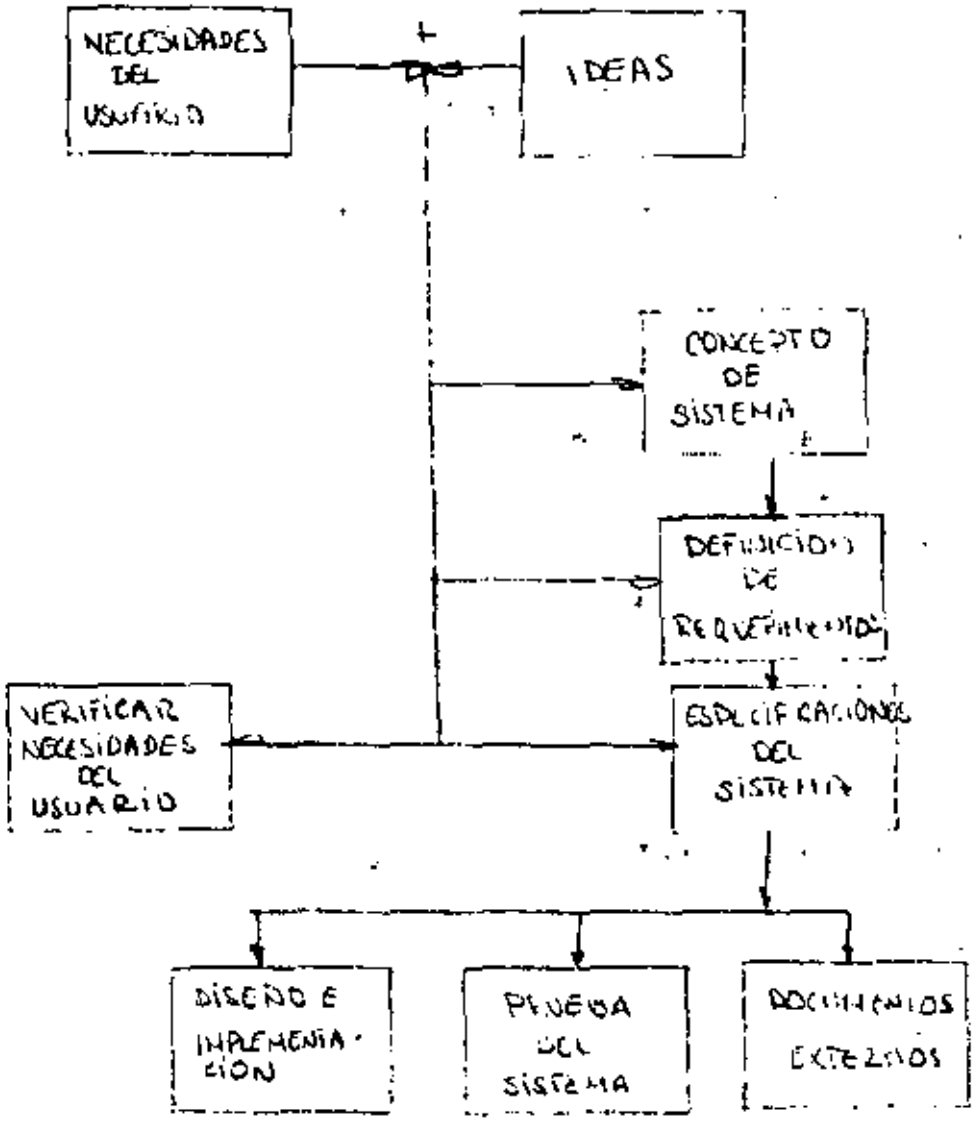
3.- DISEÑO.

( ) Cargar una 'arquitectura del sistema'.
( ) Descomponer las funciones en módulos implerecutables.
( ) Documentar el resultado en términos de como vamos a implementar.

UTILIZAR.

Concepto de grupos.
Jefe de programadores.
B S P
Descomposición jerárquica.
Diseño 'TOP-DOWN'.
Diseño estructurado.

C A P I T U L O 2



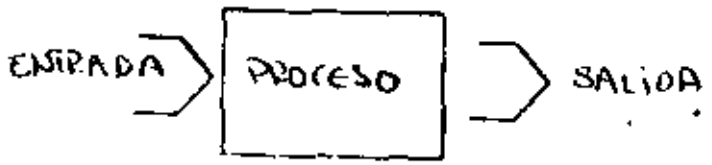
( ) ANALISIS DE LOS REQUERIMIENTOS.

. Identificar la necesidad.
. Necesidad =//= concepto de sistema (informático)
. Definir requerimientos.

( ) IDENTIFICAR LA NECESIDAD.

. Quien lo va hacer
. Se ha satisfecho la necesidad.
. Necesitamos "algún concepto técnico".
. Tenemos los elementos para llevar a cabo la tarea.

CONCEPTO DE CAJA NEGRA:



. USO DE PROTOTIPOS O SIMULADORES.

  . No se esta familiarizando con el problema.
  . El usuario no "entiende" el problema.

. HAY QUE REVISAR LOS REQUERIMIENTOS.

  . Maximizar eficiencia y confiabilidad a elinimizando costo, memoria, equipo ER. (GUAL).
  . No se entiende el objetivo.
  . No se cuantifican las metas.

. OBJETIVO DEL PRODUCTO.

  ( ) CONFIABILIDAD.
      . Tiempo 1/2 de falla.
      . # de fallas.
      . Tolerancias.
  ( ) PROTECCION DE DATOS E 6S
  ( ) DETECCION/ ERRORES.
  ( ) TIEMPO DE RESPUESTA.
  ( ) USO DE MEMORIA Y PERIFERICOS.

C A P I T U L O 3

COSTO*, TIEMPO*, CONTROLES*

METODOLOGIA UTILIZADA EN LA ESTIMACION.

( ) PRINCIPIOS.

- Descomposición del problema en bloques pequeños.
- Estimar el costo y "dificultad de cada bloque".
- Desarrollar un plan del proyecto en detalle.
- Incluir M.O, Equipo, Costos, Admon, Varios, Asesores ER.

( ) VENTAJAS.

- Hay que hacer un análisis detallado.
- Hay que estimar en detalle y con 'exactitud'.
- La estimación se torna en una parte del proyecto.
- Las faltas y errores tendrán repercuciones proventas.

( ) DESVENTAJAS.

- Hay que gasta tiempo y dinero en estimar.
- La actualización requiere de técnicas automáticas.
- ¿ Proyectos pequeños ? Vale la pena.

Si en su plan.

SI FALLA EN PLANEAR, UD. PLANEA FALLAR

PLAN PARA EL CONTROL.

- Costos.
- Calendarios.
- Eficiencia.
- Estándares y C.C.
- Personal.
- Cursos, Asesorias, etc.

ELEMENTOS DEL PLAN DEL PROYECTO.

1. Descripción del proyecto.

+ Breve descripción del mismo.
  - Objetivo, metas.
  - Razón del proyecto.
  - Organización involucrada.
  - Beneficios a obtener.

+ Breve resumen de los requerimientos.
  - M.O.
  - Duración.
  - Costo.

+ Aspectos particulares de interés.
  - Nuevas tecnologías.
  - Situación de competencia insumos.
  - Nuevos métodos de implementar.

+ Entregas.
  - Documentos.
  - SW.
  - HW.

+ Breve análisis de los riesgos.
  - Tiempos.
  - Costos.
  - Insumos.

2. ESTRUCTURA DE TRABAJO - Descripción de cada bloque.

Proyecto	Depto.	Fase	Nomenclatura	Descripción
ALFA	PROG.	DISEÑO	DISEÑO O.S.	Diseñar el OS. del Sist RT. - O.S. Interno - Manual Usuario

6. ESTIMACION DE PERSONAL. (Ver gráfica)

- + Por individuo.
- + Por actividad.
- + Por paquete

Hacer un cálculo de Hrs hombre y perfil

- + Salarios + asesores
- + Cursos (hasta 30%)
- + Viáticos.

7. ESTIMACION FINANCIERA.

( ) PRESUPUESTO.

- + Año anterior
- + Año presente
- + Año futuro.

( ) DESARROLLAR FORMATOS ESTANDARES PARA FACILITAR ESTA ACTIVIDAD.

+ Desgloce de partidas.

8. EQUIPOS.

- + Rentas.
- + Capital.
- + Desarrollo.

+ Cuando y donde se necesita.
+ Duración de uso.

14. CONTROL DE CAMBIOS.

➡ ELEMENTO CLAVE.

- Por que el cambio.
- Costo de cambio.
- Cambio de calendario.
- Quien aprueba.
- Información necesaria para llevar a cabo el cambio. (Ver procedimientos de la B.S.P.)

15. PLAN DE INTEGRACION Y PRUEBAS.

( ) PREPARAR EL PLAN DE ARRIBA-HACIA-ABAJO.
(Implementar posiblemente mejor que Bottom-UP_
( ) DESCRIPCION DEL PROCESO.
( ) SALIDAS (QUE DEBEMOS VER). Alarmas
( ) INTERFAZ HOMBRE-MAQUINA Acciones
etc.
( ) RESPONSABILIDADES

16. PROGRAMAS DE ENTRENAMIENTO

Seminarios, cursos (lugar, costo, tiempo)
Conferencias.

( ) DEFINIR UN PROGRAMA DE ENTRENAMIENTO.

17. PLAN DE SOPORTE.

- Una vez terminado un producto hay que soportarlo.

  - Mantenimiento
  - Cursos.
  - Garantias, etc.

DISEÑO.

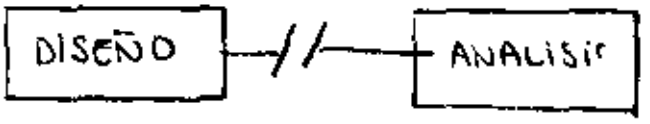
1. Aspectos directivos.
2. Elementos de diseño.
3. Consideraciones especiales.

1. Aspectos Directivos.

. Diseño.
. Organización.
. Personal.
. Documentos.
. Revisiones sobre el diseño.
. Control en los cambios.

1.1 Diseño.

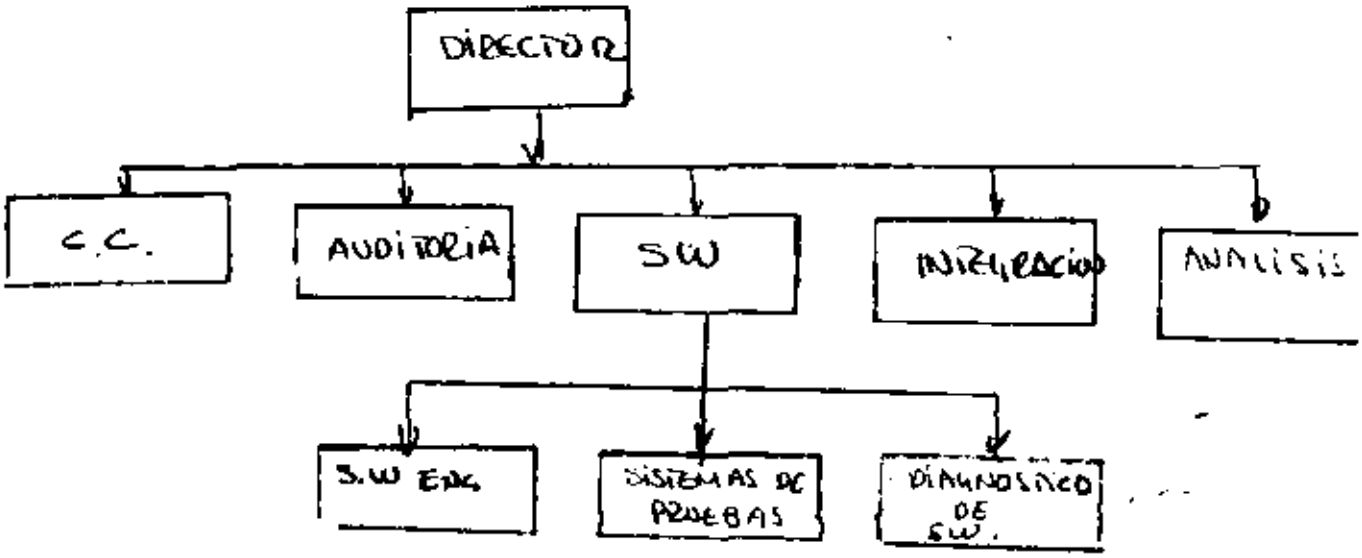
El diseño se inicia una vez que fue aprobado 'lo que no se va a producri' y el plan de implementación. Se específica como se 'rompe' cada función en partes, estas partes serán los módulos a programar.



. Deben 'direccionar' todos los aspectos del sistema (sub-- sistema)

  Algunas preguntas que nos tenemos que responder son:

  . Que tan grandes son los módulos.
  . Como dividir el diseño y la implementación.
  . Como y que vamos a probar.
  . Límites en: Memoria
    Disco
    Tiempo CPU
  . Hay que modelar ó simular.



PERSONAL.

+ Diseño funcional
  - . Orientado al producto.
  - . Orientado al sistema.
    - Programadores
    - Consultores.

+ Diseño de sistemas
  - . Con experiencia arcos.
  - . Con experiencia en programación

  Problemas Modelo
  Síntesis
  Análisis

. DOCUMENTOS.

. REVISIONES.

- . Calendario de revisiones
- . En la revisión se deben tener los documentos (sistemas) a revisar antes de la sesión.
- . No transforme las sesiones de revisión en sesiones de diseño.
- . Anote los problemas.
- . Asigne fechas y responsables.

UTILIZAR EN LA DESCRIPCION DEL PROBLEMA.

1. DIAGRAMAS DE ESTRUCTURA.
2. HIPO'S.
3. DIAGRAMAS DE FLUJO DE INFO.
   (Burbujas - Yourdon)
4. LENGUAJE PARA DESCRIBIR PROGRAMAS
   PDL o PSEUDO (o Dino)

NO UTILIZAR. DIAGRAMAS DE FLUJOS

MEJOR ó EN LUGAR DE:

WARNIER — Los veremos un poco mas adelante*

ó

NASSI — SCHNEIDERMAN

* Tomado del curso 'Desarrollo de Sistemas' F.A. Rosenblueth.

EJEMPLO DE PDL (CASI PDL)

SUBRUTINA SCIIA EBC

LEE Longitud del Buffer de entrada en Bufent

$ PTR+0

LLAMA el MGR del espacio libre para que de un Buffer igual a Longitud (BUF. Salida)

IF

SI no hay Buffer ENTONCES regresa con + no BUFF.

ELSE HAS HASTA todos los caracteres del buffer de entrada se han convertido

READ

LEE los caracteres ASCIL del buffer de entrada un INBUFF $PRT + 1

Convierte a EBCIDIC (luego decidimos el algoritmo)

WRITE

ESCRIBE los caracteres EBCIDIC and buffer de salida.

FINDO

CALL

LLAMA al MGR del espacio libre para que 'deje' el buffer de entrada

Regresa el estado = termine y OUTBUFF$PTR

CONSIDERACIONES ESPECIALES

° El SW debe escribirse para el FUTURO

° La transportabilidad requiere consideraciones especiales

° Hay que diseñar con la idea de MANTENER

Veamos esto

Costos de Desarrollo

En la implementación de 'grandes proyectos. se recomienda

° Organizacion Funcional por Subsistema y tipo

( ) Lo que mantiene modulos lógicos agrupados

( ) Minimiza el 'SPAN' de las comunicaciones

( ) Mejora el 'inventario'

i.e.

SISTEMA	DIAGNOSTICO	HERRAMIENTAS	COMUNICACIONES (SW)
- OS	+ COMPONENTES	+ TRADUCTORES	+ SDLC
- DRIVERS	+ PERIFERICOS	+ INTERPRETES	+ BISYNC
- UTILERIAS DEL SISTEMA		+ EMULADORES	+ ASYNC
- CONTROL ARCHIVOS		+ REVISORES DE CODIGO	+ AUTO ANSWER PISTL
DATA BASE		+ SIMULADORES	

GENTE (El mayor problema)

Fijese usted: No hay un tipo de programadores

+ Administradores	+ matematicos	+ Musicos
+ Informaticos	+ Actuarios	+ Artistas
+ Ingenieros	+ Psicologos	

META:

Construir un sistema que funcione

° Por pasos/Etapas

° En cada etapa "hay que demostrar"

FILOSOFIA PARA IMPLEMENTAR

TAMAÑO DEL PROYECTO

+ Los proyectos pequeños se pueden implementar con mayor facilidad hay que romper en modulos

+ Mejor control del personal en proyectos pequeños

SI PODEMOS ESCOGER

+ Tener el HW Corriendo

+ Tener el S.O. Funcionando

+ Tener las utilerias y herramienta de desarrollo

HAY QUE

Minimizar el tiempo/costo de desarrollo

Minimizar utilizacion de HW

Maximizar el mantenimiento

CODIFICACION ESTRUCTURADA

CODIFICACION ESTRUCTURADA:

- Genera modulos que

1.- Son relativamente pequeños

° alrededor de 60 a 100 líneas de codigo incluyendo comentarios

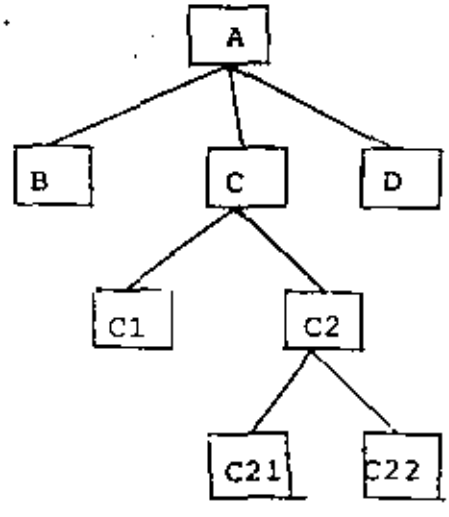
° Se puede 'pensar' en todo el algoritmo

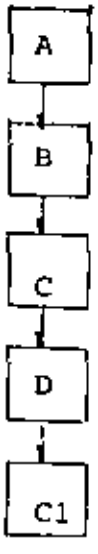
2.- Relajadamente acoplados

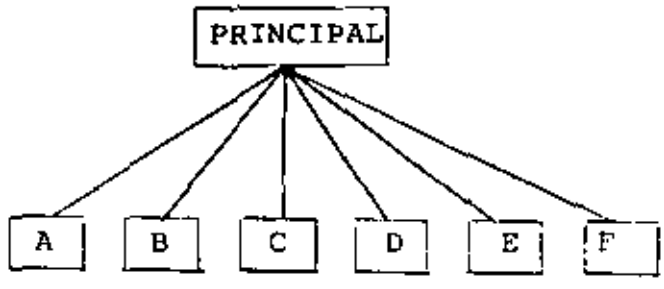
° Una sola entrada, llamadas simples a rutinas

° una sola salida, pasar parametros en forma simple

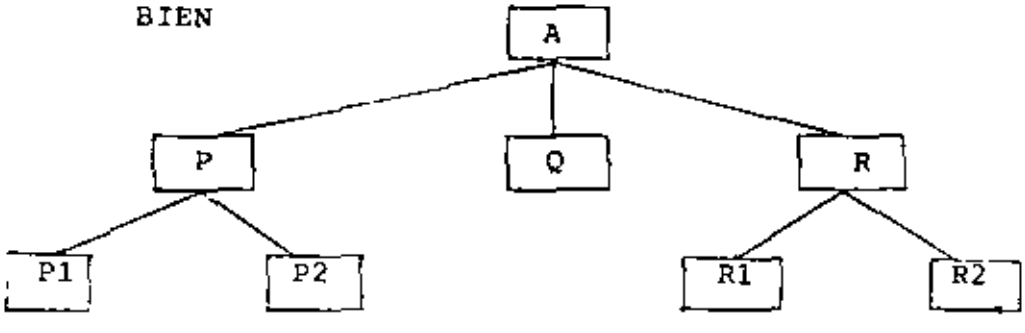
i.e. MAL







BIEN



4.- Funcionalidad

°Los modulos deben ejecutar funciones simples y relacionadas

°Se deben poder definir en terminos de una caja negra (características externas)

5.- Codigo documentado

ENCABEZADO

Nombre del Programa

Fecha Revisión

Programo Reviso

Fecha

° Objetivo

° Entradas

° Salidas

° Logicas

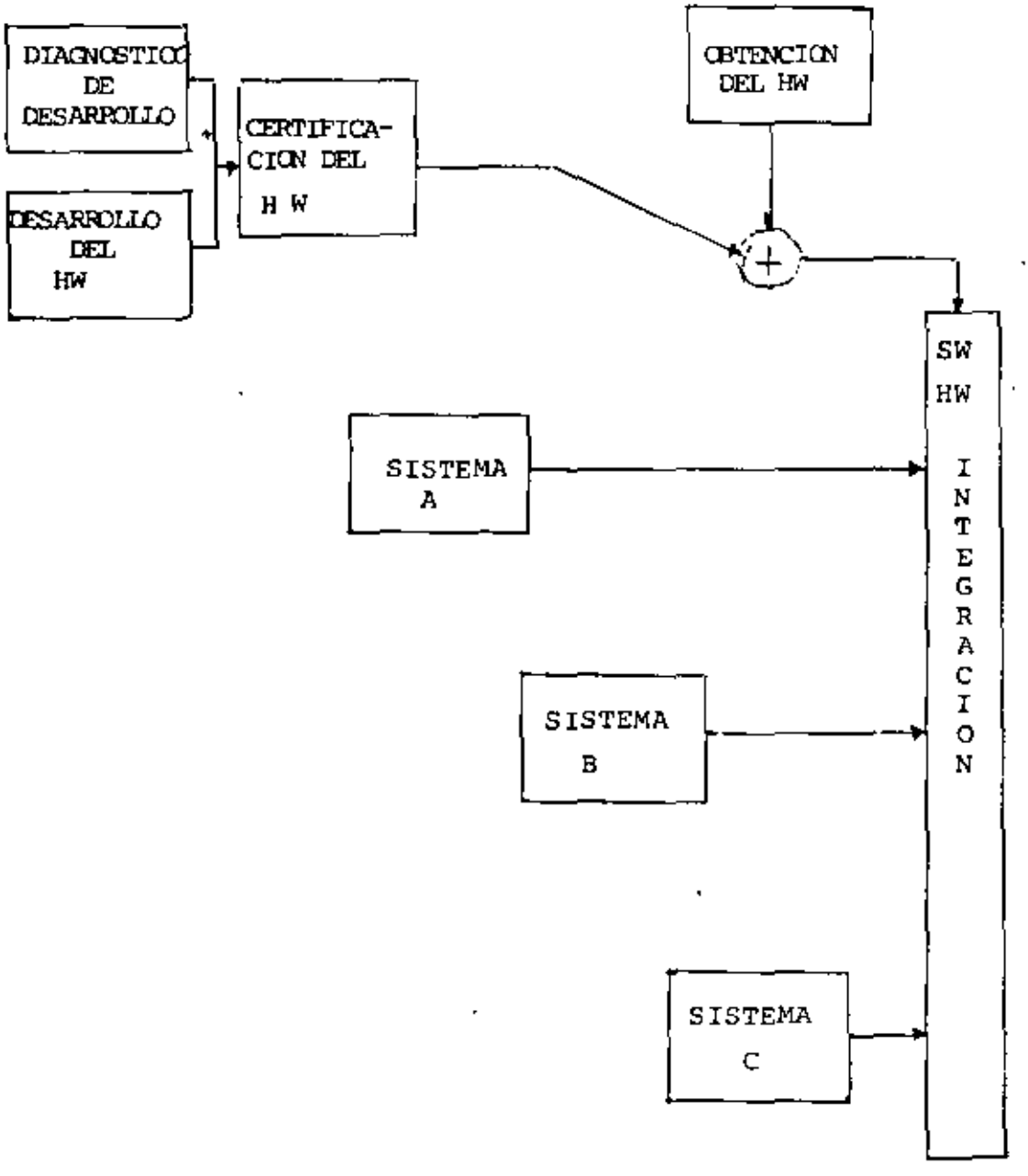
° Externas

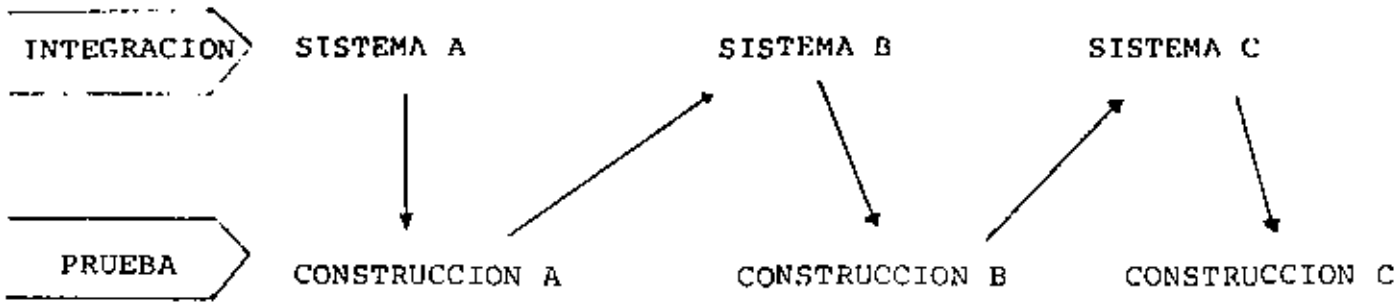
°

PDL

FASE DE INTEGRACION

"Desde el punto de vista de SW la manera ideal de integrar SW es hacerlo en HW totalmente operando (Versión a utilizar)





La integración involucra 3 aspectos principales:

( ) El proceso de integración (de acuerdo al plan)

- referenciamiento de la unidad probada a la libreria
- hacer link' de los modulos para tener una nueva construcción
- preparar la transmisión documentación de liberación

( ) El proceso de prueba

+ 'correr' la construcción de acuerdo a la 'prueba'
+ hacer diagnostico y "atrapar" errores
+ Generar un reporte de la construcción

( ) Proceso de evaluación

+ Evaluar los resultados de la prueba
+ Decidir si se acepta, para llevar a cabo la siguiente construcción

Que pasa si hay errores de diseño o

la integración

+ Recuerde que por razones de 'magia' los errores salen en la demostración al cliente y estos son peores si se trabaja en linea

Por lo que

( ) En las pruebas de diseño, para cada elemento se debe utilizar una matriz de prueba correspondiente de prueba indicando la función.

+ La operación de cada función es calificada de "correcta" si cumple la prueba cuando se corre.
+ Los problemas se deben reportar en la forma adecuada.

( ) La matriz de correspondencia debe incluir

+ Operaciones así como rangos
+ operacion con entradas anormales o cursos extraños
+ operación por actividad

Se deben tener 3 modos (cuando menos)

MODO 1	FALLA	El sistema va a tener problemas
MODO 2	PREDICCION	El sistema puede tener problemas
MODO 3	OK	El sistema deberá correr OK

Cuando falla sucede lo que se conoce como

LOCURA DE CAMBIO

Hay que justificar todos los cambios

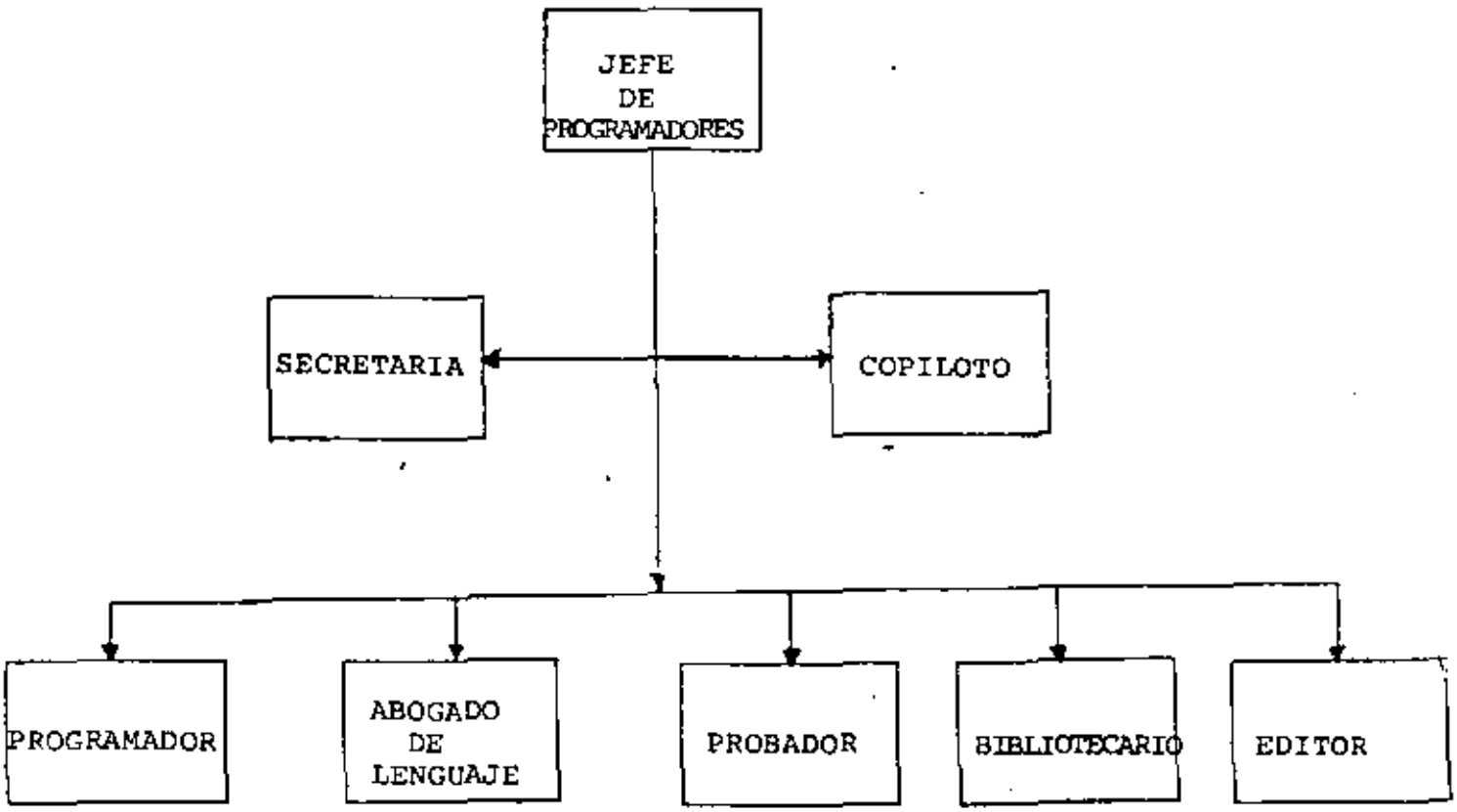
- no existe el cambio pequeño
- cada cambio da lugar a una nueva construcción
- los errores deben eliminarse hay que hacer cambio

LOS CAMBIOS SE HACEN A TRAVES DE LA BIBLIOTECA DE SOPORTE 'DE PROGRAMAS

LEY DE BLOOK'S

Si su proyecto esta atrasado y contrata mas personal para salir adelante, mas tiempo se atrasara y esto se cumple como que 1+1 = 0 y llevamos 1

Mejor organizar por equipos



HAY QUE BUSCAR EDQUIPOS DEMOGRAFICOS O ADAPTIVOS o una FAMILIA DE PROGRAMADORES

Hay que buscar el cambio

'ARTE PRIVADO EN PRACTICA PUBLICA'

'Buscar un ambiente en el cual existan discuciones CONSTRUCTIVAS y no criticonas'

WALKTHROUGH'S o PLATICAS ESTRUCTURADAS

El líder natural

Diferencia entre el equipo clásico y "el otro"

- NADIE ESTA AL MANDO (lo cual pone nerviosas a muchs personas
- no existen los superprogramadores
- trabajo de grupo

WALKTHROUGH 'S Libro de Yourdon

ESTRUCTURED WALKTHROUGHS PRENTICE HALL

YOURDON INC

VEAMOS

COMO SE LLEVA A CABO UNA PLATICA

° Se deben sujetar a un calendario, y repartir a los asistentes los documentos 2 ó 3 días antes de la plática.

° LOS JEFES NO ESTAN INVITADOS

°Se nombra un coordinador-arbitro que evite que se torne la sesión en una sesión de box.

° Se nombra a un 'escribano' quien toma notas, hace una minuta y la distribuye entre los asistentes

° El autor puede o no presentar su trabajo

° Cada revisador comenta en los bugs omisiones, debilidades, etc.

° Los problemas son encontrados pero NO resueltos durante los waklthroughs.

° Debe durar entre 20 min. y 1 hora máximo

° Se termina la sesión con un voto de:

  ° Se acepta el producto

  ° Se acepta el producto con pequeñas revisiones

  ° Hay que reunirse de nuevo.

DESARROLLO DE UN PLAN DE PRUEBAS

Punto básico.- Sin un plan organizado no se pueden tener pruebas 'decentes' por lo que

1.- Responsabilizar a una persona/grupo de las pruebas
2.- Establecer los procedimientos y los STD'S para llevar a cabo las pruebas
3.- Describir los casos de prueba.
    - para aceptación
    - y versión de arriba hacia abajo
4.- Criterios de terminación de pruebas
5.- Calendario de actividades y recursos requeridas, para llevar a cabo las pruebas

MANUAL DE ESTANDARES VOL 31 COMO CONSTRUIR CASOS DE PRUEBA	MANUAL DE ESTANDARES VOL 32 COMO DOCUMENTAR Y USO DE XXX	MANUAL DE ESTANDARES VOL. 32 CONVENCION DE NOMBRES	MANUAL DE ESTANDARES VOL 3 BASE DE DATOS

GENERACION DE DATOS DE PRUEBA

- Para generar datos de prueba 'inteligentemente' debemos considerar:
  - que aspectos del sistema (comportamiento) son los que vamos a probar
    - funciones externas
    - eficiencia
    - logica interna
    - etc.

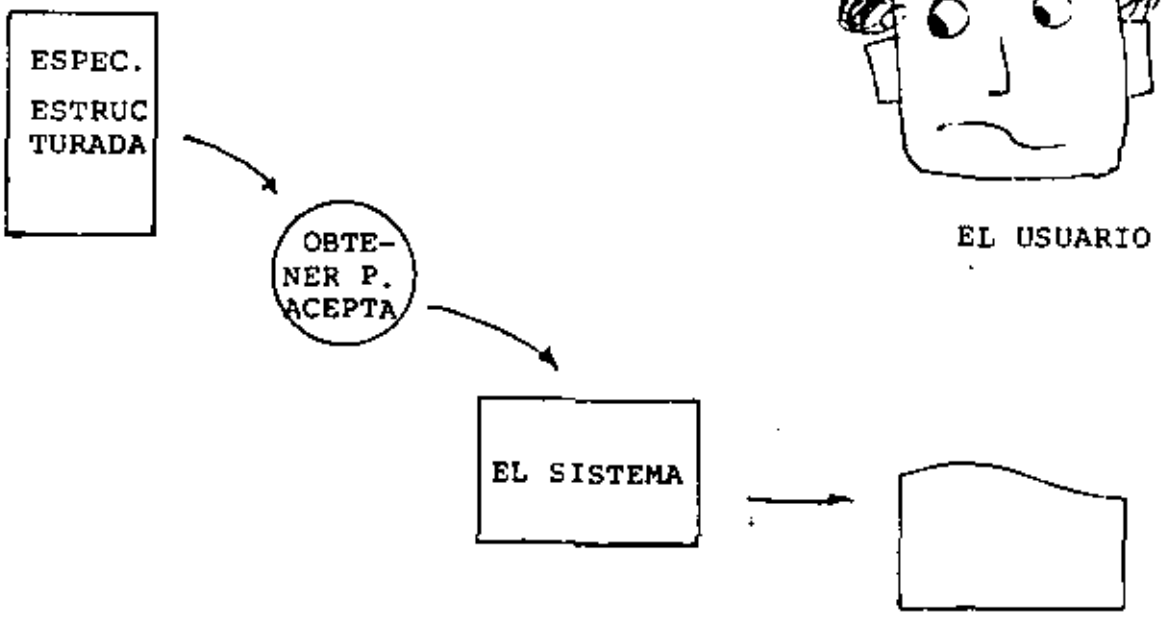
Esto nos determinará los valores de los datos de prueba

- Que "combinación" de los componentes del sistema estamos probando.
  - modulos
  - colecciones de modulos
  - modulos con E/S
  - modulos con acceso a B.D.

Esto nos determinará la complejidad de las pruebas

+ ¿Como podemos utilizar herramientas automatizadas para mejorar la eficiencia y productividad del proceso de prueba?

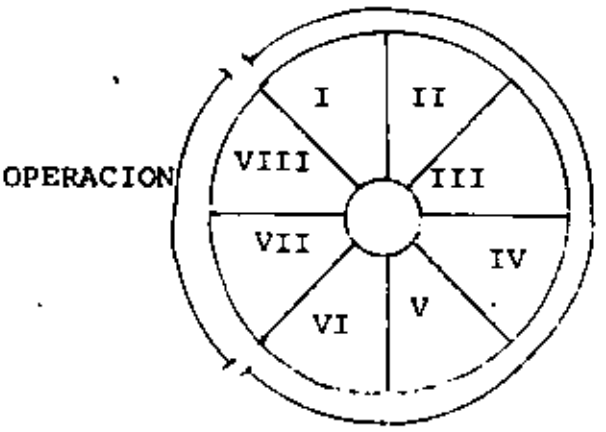
PRUEBAS DE ACEPTACION



- Hay que involucrar al usuario
- las pruebas deben salir solamente de las especificaciones (RPC)
- Si al sistema no pasa las pruebas de aceptación, seguramente hay una falla en el Análisis y no en la Codificación

UN PROYECTO DE SW TIENE 8 FASES QUE SON:

I	INICIO
II	DEFINICION
III	DISEÑO
IV	PROGRAMACION
V	PRUEBA SISTEMA
VI	ACEPTACION
VII	INSTALACION
VIII	OPERACION



FASE DE INICIO.- En esta fase hay que contestar:

- Cuales son los requerimientos
- Que se debe producir
- Quien es el usuario, cual es el mercado
- Cuales son las ventajas/desventajas
- Que características y efectos va a tener
- Es factible
- Se pueden hacer en tiempo/presupuesto

DOCUMENTO QUE RESPONDA A LAS PREGUNTAS ANTERIORES

IV FASE DE PROGRAMACION

En esta fase se toman las desiciones técnicas, así como se hace un diseño detallado de programas de computadora
La fase de programación produce modulos codificados que se pueden probar.

El control de calidad del producto empieza en esta fase

Programas fuentes, objeto, notas bibliografía, papers, reportes de problemas, documentos etc. son entregados a la Biblioteca de soporte de programas.

Esta fase termina con la programación (escribir codigo) de todo el sistema integrado y se tiene una 'primera' versión de manuales.

V FASE DE PRUEBA

Esta fase asegura que el producto cumple con los requerimientos funcionales, además verifica que no se tengan errores 'significativos'.
Las pruebas las lleva a cabo un grupo diferente del grupo que programa, y utiliza el documento de 'diseño' y el manual de usuario para llevar a cabo sus pruebas

Todos los errores/solicitudes de cambio se hacen a traves de la Biblioteca.

ACTIVIDADES

1.- LA ACTIVIDAD DE ANALISIS (inicio y definición)

El proposito de esta actividad es el desarrollo (y bases) para el diseño.

- se debe entender el problema
- se deben conocer los requisitos
- tener en cuanta factores humanos, ingenieria, documentos herramientas equipos, etc.

Lista de Verificación

( ) Libro de bitacora
( ) Definir las necesidades del usuario (Hay que estar seguro)
( ) Comunicación con el usuario
( ) Conocimeinto de HW existente
( ) Analizar la necesidad de simular/modelar
( ) Investigar proyectos semejantes
( ) Definir y evaluar alternativas
( ) Revisar los documentos de definición con el cliente

2.-. PLANEACION.

Se debe desarrollar un documento base que sirva de punto de partida entre el cliente y el diseñador.

Calendario

Recursos

- Fisicos
- Humanos
- Economicos

CONTROL DE LA CONFIGURACION	Definir responsabilidades y controles en cada fase y actividad, así como control en los cambios.
DOCUMENTACION	Definir que documentos se van a elaborar (WP, Estilo, etc)
PLAN DE ENTRENAMIENTO	Definir los requisitos internos y externos de un plan de entrenamiento y los responsables de llevarlo a cabo
REVISIONES, AUDITORIAS Y REPORTES	Definir revisiones en los documentos y productos indicando un avance
PLAN DE INSTALACION Y OPERACION	Describir la forma y requisitos de instalación, así como un plan para llevarlos a cabo

3.- ACTIVIDAD DE DISEÑO

DISEÑAR BIEN = BUEN PRODUCTO

- Diseño General
- Diseño Detallado

Al final de esta fase de deben tener:

- la producción (lo que se va a producir)
- la descripción detallada de cada unidad
- pruebas de integración
- pruebas del sistema
- parametros de control de calidad
- calendario de laticas estructuradas

UN MANUAL DE LO QUE SE VA A PRODUCIR

ACTIVIDAD DE CODIFICACION

Una vez que se ha diseñado en la forma correcta y apropiada se procede a codificar de arriba hacia abajo. Cada programador debe tener una tarea bien definida del diseño detallado y al terminarlo lo deberá probar en forma unitaria.

- Se deberán tener encabezados detallados de cada modulo
- platicas estructuradas
- c.c.
- un plan detallado de pruebas
- reportes de progreso
- reportes de cambio C.C.
- reportes de control
- reportes de problemas
- manuales cuando menos en cada encabezado

Al final se tienen MODULOS CORRIENDO Y MANUALES

LA NECESIDAD DE LOS DOCUMENTOS

Los documentos resuleven problemas de comunicación

3 tipos de documentos:

TECNICOS.- registros de información de como trabaja el sistema y lo que hace

USUARIO.- registros de que hace el sistema y como usarlo

CONTROL.- información acerca del desarrollo del proyecto.

FASE	DOCUMENTO	TIPO
INICIO	especificaciones de marcado analisis de los requerimientos analisis de factibilidad	TECNICO
DEFINICION	Requerimientos funcionales - descripción de los requerimientos del SW - marco de referencia para el diseño - eficiencia y restricciones - calidad	TECNICO

- reportes de cambios

- listados de programas

PRUEBAS + reporte de pruebas

+ reporte de aceptacion TECNICO

PRUEBA DEL SISTEMA

+ STD'S De pruebas

- descripción de las pruebas a llevar a cabo en el sistema TECNICO

OPERACION + Reporte de fallas y problemas

+ Reporte de sugerencias y mejoras

+ Bitacora historica

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

EVALUACION DE REQUERIMIENTOS DE SOFTWARE.

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

LENGUAJE DE PROGRAMACION	ESTIMACION OPTIMISTA	BAJO INCERTIDUMBRE		EXTREMA	
LENGUAJE DE PROGRAMACION	NUMERO DE INSTRUCCIONES EJECUTABLES EN EL LEN-- GUAJE A SER USADO	PARA ESTE LENGUAJE HAGA DOS ESTIMACIONES DEL NUMERO DE INSTRUCCIONES A UTILIZAR UNA POR DEBAJO DE.LA ESTIMACION OPTIMISTA Y OTRA POR ARRIBA		PARA ESTE LENGUAJE HAGA DOS ESTIMACIO- NES ESTREMAS UNA EL NUMERO DE INSTRUCCIO NES EJECUTABLES MINIMO Y OTRA MAXIMO.	
		+	-	+	-

PREPARADOS PARA EL TRABAJO

FACTORES	SI	NO	PARCIALMENTE	NO APLICABLE
SE DEFINIERON LOS REQUISITOS DE HARDWARE				
SE SELECCIONO EL PROVEEDOR				
SE SABE DE RESTRICCIONES DE RECURSOS - - HARDWARE/SOFTWARE				
SE CONOCEN LAS RESTRICCIONES DE LOS -- PRC RESPECTO A HARDWARE/SOFTWARE				
SE HA DISEÑADO LA FUNCION DE INICIALIZA-CION				
SE HAN DESEÑADO PROCEDIMIENTOS DE ERROR/ RECUPERACION				
TIENE EL CLIENTE CONFIANZA EN EL DISEÑO-DEL SISTEMA				
SE TIENE CONTRATADOS A LOS SUBCONTRATIS-TAS DE MANERA QUE SE CUMPLAN PLAZOS				
SE HAN DEFINIDO METODOLOGIAS Y TECNICAS (IC. PLAN DE DESARROLLO TECNICO)				
EXISTE UN JEFE QUIEN ENTIENDE: EL PROBLEMA TECNICO EL PROBLEMA ADMINISTRATIVO/ORGANIZATIVO EL AMBIENTE DE DESARROLLO DEL CLIENTE				
LOS JEFES DE GRUPO O JEFES TECNICOS ES-TAN CONCIENTES DE LAS TAREAS A DESARRO-LLAR Y SUS RESPONSABILIDADES				

TECNOLOGIA EXISTENTE HARDWARE/SOTWARE

AREA TECNICA / ESTADO	SEGURIDAD	BASE DE DATOS	ARQUITECTURA DE COMPUTADORA	INTELIFENCIA ARTIFICIAL / REC. FORMAS	COMUNICACIONES	OTRA
FAR NUNCA HA DESARROLLADO UN SISTEMA EXACTAMENTE COMO ESTE, SIN EMBARGO SE HAN DESARROLLADO SISTEMAS SEMEJANTES Y EXISTE AYUDA TECNICA EN LA FAR						
FAR NUNCA HA DESARROLLADO UN SISTEMA EXACTAMENTE COMO ESTE, SIN EMBARGO EXISTEN SISTEMAS SEMEJANTES Y SE PUEDE OBTENER ASESORIA TECNICA FUERA DE LA FAR						
FAR NUNCA HA DESARROLLADO UN SISTEMA EXACTAMENTE COMO ESTE Y NO EXISTE ASESORIA TECNICA FUERA O DENTRO DE LA FAR						
SE CREE QUE EL SISTEMA PROPUESTO PUEDE SER DESARROLLADO, SIN EMBARGO UN SISTEMA CON LOS REQUISITOS QUE PRESENTA ESTE NUNCA HA SIDO DESARROLLADO						
LA FACTIBILIDAD DEL DESARROLLO DE ESTE SISTEMA DEPENDE DE LA SOLUCION TECNICA DE PUNTOS CLAVES EN ESTA AREA						
PARA CADA AREA TECNICA ESTIME EL PORCENTAJE TOTAL DE ALGORITMOS PIONEROS						

COMPLICACIONES INHERENTES DEL PROBLEMA

FACTORES		SI	NO	PROBABLE-MENTE SI	PROBABLE-MENTE NO	PARECER DEL ANALISTA
EXISTEN CARGAS ASINCRONAS						
HAY QUE NEGOCIAR HARRDWARE/ SOFTWARE ?						
DURANTE LA OPERACION EXISTEN CAMBIOS DINAMICOS EN EL AMBIENTE O LA FUNCIONALIDAD						
SE REQUIERE QUE EL SISTEMA-ESTE LIBRE DE FALLAS PARA:	ENTRADAS/SALIDAS					
	HARDWARE					
	SOFTWARE					
SE REQUIERE RECUPERACION -- DE ERROR PARA:	OPERACION BATCH					
	OPERACION TIEMPO-REAL					
REQUIERE EL SISTEMA NIVELES DE SEGURIDAD	DEDICADO					
	MODO MIXTO					
SE REQUIERE CLAVES ESPECIALES DE SEGURIDAD						
EXISTEN RESTRICCIONES EN LO QUE RESPECTA A: (EFICIENCIA)	TIEMPO DE RESPUESTA					
	TAMAÑO CANT. INFO					
	PRECISION					
	CONFIABILIDAD					
	# ERRORES					
	OTRO					
LA IMPLEMENTACION DEL SISTEMA LLEVA ALAS SIGUIENTES RESTRICCIONES DE RECURSOS	CPU > 75%					
	UTILIZACION MCM CORE > 75%					
	UTILIZACION DE MEMORIA DISCO O CINTA > 75%					
	APLICACIONES INDIVIDUALES DEL 75% PERO EL SISMA TOTAL MAYOR DEL 75%					

PERSONAL

	FACTOR DE CALIFICIACION					PARA USO DEL ANALISTA
	POCO 1	2	3	4	MUCHO 5	
EXPERIENCIA EN LA APLICACION ESPECIFICA DEL PROBLEMA						
EXPERIENCIA EN PROBLEMAS SIMILARES						
EXPERIENCIA EN EL DESARROLLO DE SOFTWARE						
EXPERIENCIA CON EL SISTEMA OPERATIVO/SOPORTE SOFTWARE						
EXPERIENCIA CON EL (LOS) LENGUAJES UTILIZADO (S)						
RESUMEN GENERAL DE LA CALIDAD DEL EQUIPO (PERSONAL)						

DIRECTORIO DE PROFESORES DEL CURSO

ADMINISTRACION DE PROYECTOS DE SOFTWARE 1982.

1. M. EN C. MARCIAL PORTILLA ROBERTSON (Coordinador)
Programa Universitario de Computacion
U N A M
Cubículo PB
Edificio IMASS
México 20, D.F.
550 52 15 Ext. 4546

ADMINISTRACION DE PROYECTOS DE SOFTWARE 1982

Fecha	Tema	Horario	Profesor
Del 15 al 20 de Febrero	INTRODUCCION	17 a 21 h los Viernes	M. en C. Marcial Fortilla Robertson
	PROBLEMAS EN EL DESARROLLO DE PROGRAMAS	9 a 14 h los Sábados	
	NECESIDADES DE LA ADMINISTRACION DE SISTEMAS DE SOFTWARE (INGENIERIA DE SOFTWARE)		
	ESTIMACION DE COSTOS		
	PROGRAMACION Y DISEÑO ESTRUCTURADO		
	P. D. L.		
	CONTROL DE CALIDAD		
	PLANES Y PROGRAMAS DE PRUEBAS		
	BIBLIOTECA DE SOPORTE DE PROGRAMAS		
	PLATICAS ESTRUCTURADAS		
	MESA REDONDA (DISCUSION)		

'edcs.

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

ADMINISTRACION DE PROYECTOS DE SOFTWARE

INGENIERIA DE SOFTWARE

M. EN C. MARCIAL PORTILLA ROBERTSON

NOTAS REPRODUCIDAS CON
AUTORIZACION DE LA FUNDACION
ARTURO ROSENBLUETH

FEBRERO, 1982

1. ASPECTOS GENERALES

1.1 The need for software Engineer
Mayers W. :Computer Feb. 1978.

1.2 Software Engineering: Process, Principles and Goals.
Ross T, Goodenough J., Irvine C.
Computer Mayo 1975.

1.3 The Mythical Man-Month
Brooks Jr.
Datamation 1974.

1.4 Why Projects Fail
Keider W.
Datamation December 1974.

2. ESTIMACION DE COSTOS

2.1 The cost of developing Large-Scale Software
Wolverton R. IEEE transactions on Computers.
June 1974.

2.2 Estimating Software Costs
Putnam L. Fitzsimmons
Datamation, Septiembre 1979.

2.3 Estimating Software Costs.
Putnam L, Fitzsimmons A.
Datamation Oct. 1979.

2.4 Estimating Software Costs
Putnam L, Fitzsimmons A.
Datamation Nov. 1979.

2.5 Estimación Costos de Desarrollo de Programas.
Portilla Marcial (Documentación Interna F.A.R.)

3. PROGRAMACION, DISEÑO ESTRUCTURADO PDL,
Convención uso de nombres y variables.

3.1 On the Composition of Well-Structured Programs
Wirth N. Computing' Surveys Dec. 1974.

3.2 Structure Design.
Stevens P., Myers G, Constantine L.
IBM System Journal 1974.

3.3 Convención en el uso de cuentas, nombres....
Portilla Marcial, IIE

3.4 PDL
Portilla Marcial Documento F.A.R.

3.5 Software Represention Composition Techniques
Peters L. Proceeding IEEE Sept. 1980.

4. INGENIERIA DE SOFTWARE

4.1 Measurement and Experimentation in Software Engineering
Curtis B. Proceedings of the IEEE Sept. 1980.

4.2 Software Proyect Verification Validation
Deutsch M.
Computer IEEE, 1981.

4.3 Software Management. A Survey of the Practice in 1980. Distaso J.
Proceedings of the IEEE Sept. 1980.

4.4 Software Quality Assurance: Testing and Validation
Goodenough J. Mc. Gowan C.
Procedeening IEEE Sept. 1980.

4.5 Specifyng Software Requirements
Yeh, Raymond, Proceedings IIEE, Sept. 1980

4.6 Specifyng Software Requirements
Yeh, Raymond, Proceedings IIEE, Sept. 1980

5. BIBLIOTECA DE SOPORTE DE PROGRAMAS.

6. FORMAS VARIAS.

1.- ASPECTOS GENERALES:

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 1956 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,[9] Boehm projected the ratio for the Air Force as going to 10:1 by 1985. On one existing program, the World Wide Military Command and Control System, the ratio was already in that vicinity—$722 million for software to $50-$100 million for hardware. This system, with 35 locations, ties the President and the Pentagon with commands in the United States, Europe, and the Pacific, using airborne command posts in part.[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* 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.[4]

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."[5]

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.

*Elshoff warned that the conditions were as of 1973. Since then GM has made many improvements.

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[6] 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."

- 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.[7] 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.

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 (MAJOR AND MINOR ONES)

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 engineering. 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[3]

Once requirements are turned 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[30,31,32]

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[14,31,45]

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[34,35]

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[31]

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. This encourages 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 corrections, 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.[5] However, two of the projects were very small, so in view of the belief that productivity on small projects is highly dependent on the individuals involved, the results from these two projects have been eliminated from the data summarized here.

The key comparison was made between man-months for the projects as forecasted by Boeing's traditional estimating procedure and 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

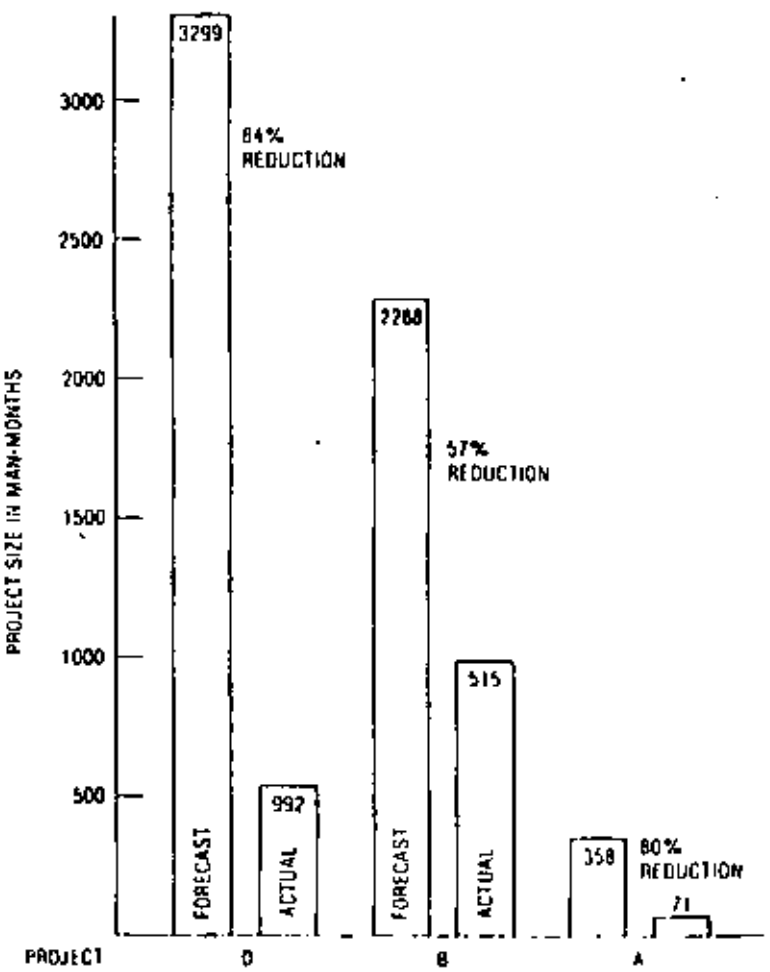


Figure 1. Improvement of actual man months over forecast man months on three Boeing software projects as a result of adopting modern programming practices.

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[9] interviewed a cross section of management and performer personnel on TRW's very large ballistic-missile-defense programming project, as well as key staff personnel outside the project. This survey covered the impact of 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 columns, with one exception, were selected to show the stronger relationships. Structured coding, which was not strongly correlated, was included as a matter of interest, since it is one of the core practices.

Structured coding evoked strong feelings on the part of the interviewees. Overall it had 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

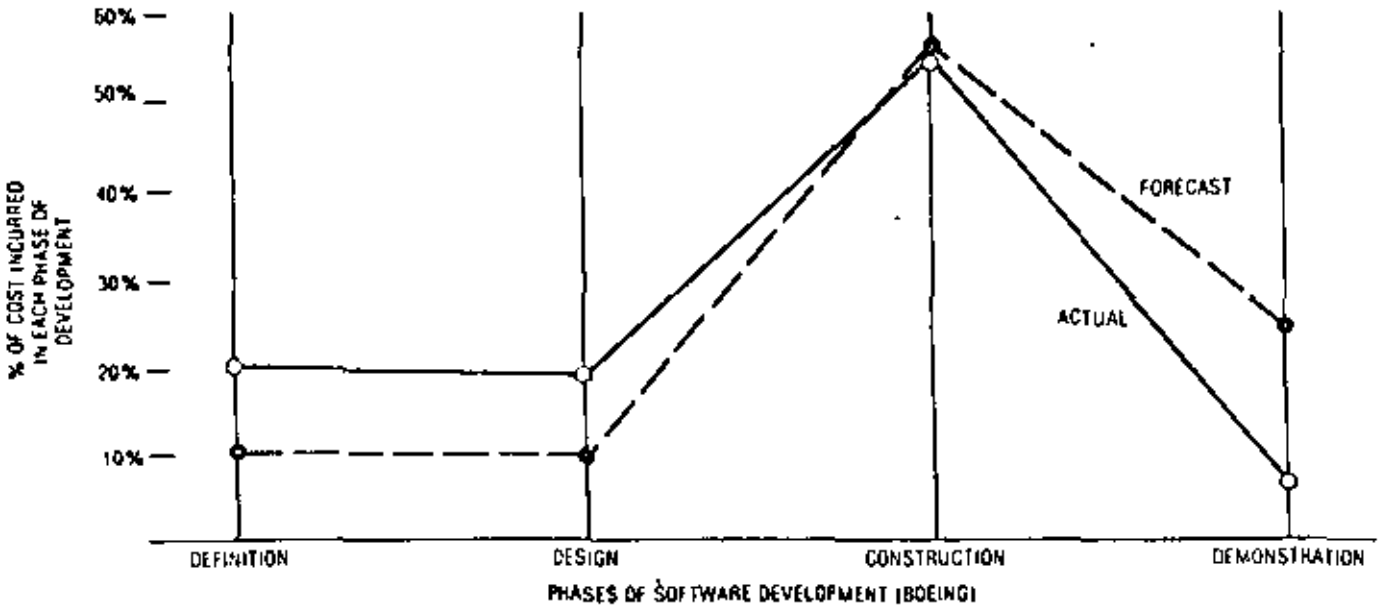


Figure 2. Forecast cost distribution shifts to earlier phases when modern programming practices are employed (average of three projects).

Table 4. Relationship of selected programming standards and software characteristics (first TRW study).

PROGRAMMING STANDARDS	SOFTWARE CHARACTERISTICS								
	CODE AUDITABILITY	CODE UNDERSTANDABILITY/ READABILITY	CODE MAINTAINABILITY/ USABILITY	TESTABILITY	OPERATIONAL RELIABILITY	CODING ERROR FREQUENCY	PROGRAMMER PRODUCTIVITY	TOTALS FOR 7	TOTALS FOR 30
ROUTINE SIZE (MODULARITY)	+3	+3	+3	+3	+2	+2	0	+16 +57%	+34 +26%
IN LINE COMMENTARY	+3	+4	+4	+1	+1	+2	0	+15 +54%	+37 +31%
STRUCTURED CODING	+3	+3	0	+2	+2	0	−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 THW evaluation survey generated a matrix of 11 modern programming practices against 12 software problems, making 132 intersections in all. In this study personnel were asked to respond on both an *actual* and a *theoretical* basis. The theoretical responses showed a higher degree of relationship than the actual by a weighted margin of better than 3 to 2. In addition, the "indifferent" or "inconclusive" intersections dropped from 40 to eight. Of the 40 indifferent or inconclusive intersections in the actual comparison, 26 were involved in cost or schedule overruns or inefficient use of resources, again suggesting the interviewees were currently dubious in these areas. On a theoretical basis, however, the indifferent and inconclusive intersections to these three software problems dropped to five, implying that with more experience, modern programming practices would have a more favorable impact on these problems. In fact, the ratings for these three problems increased from "inconclusive" to "medium" or "strong positive."

The average impact of each modern programming practice on the twelve software problems is graphed in Figure 3. The theoretical ratings are considerably better, this time by about 2 to 1 in weighted terms.

In Brown's own words, "There is strong agreement among [project] personnel as to the four MPP of greatest importance and impact and strong agreement on their relative ranking:

- Requirements analysis and validation
- Baselining of requirements specification
- Complete preliminary design
- Process design

There is strong agreement on the importance and positive impact of the next three most highly ranked MPP, but the relative ranking among them is less clear:

- Incremental development
- Unit development folders
- Software development tools

There is strong agreement on the importance and positive impact of the four lower ranked MPP, but the relative ranking among them is not at all clear:

- Independent testing
- Enforced programming standards
- Software configuration management
- Formal inspection of documentation and code."

As to the general MPP hypothesis, the query and the results were as follows: Rules governing software development, evaluation, and documenta-

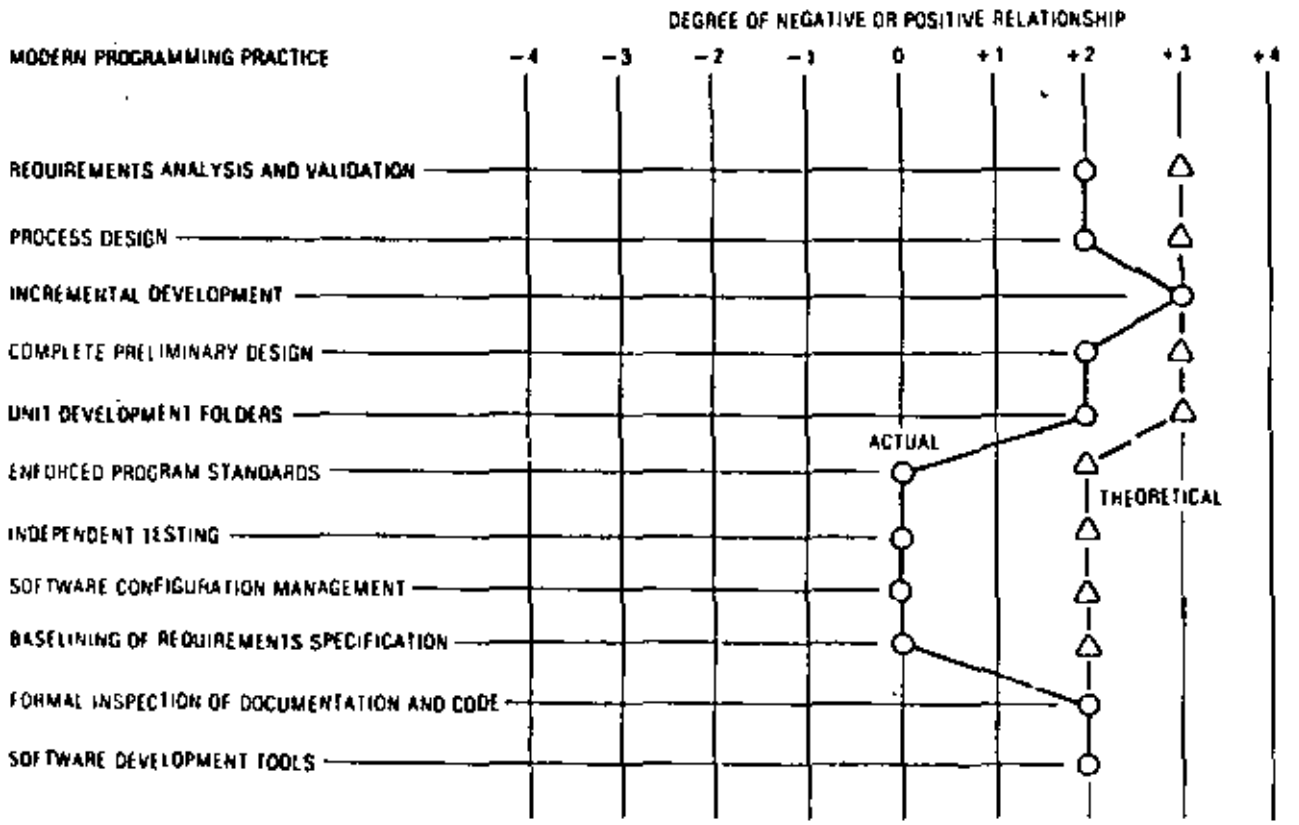


Figure 3 The correlationships between the modern programming practices and the summation of the twelve software problems is stronger in the theoretical case than in the actual case by about two to one.

tion, if rigorously defined and applied and supported by modern programming practices (techniques and tools), make possible the production—

	True	False	?
—of higher than usual quality software	65 percent	4 percent	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."[47]

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 (RANGE NUMBER 45 TO 75)
PROGRAM REVIEWS
CHIEF PROGRAMMER
CHIEF PROGRAMMER TEAM
BUILD LEADER (A COMPUTER SCIENCE CORP. TECHNIQUE)[47]
STRUCTURED WALKTHROUGHS
EFFECTIVE (28 33)
INDEPENDENT TEST AND EVALUATION
EXECUTION ANALYSIS
PROGRESSIVE TESTING
LESS EFFECTIVE (10-15)
PROGRAMMING TECHNIQUES
STRUCTURED PROGRAMMING
TOP DOWN PROGRAMMING
VERIFICATION PROCEDURES
SUPPORT PROGRAMS
(TOP-DOWN DEVELOPMENT)
LEAST EFFECTIVE (0-4)
(INSPECTION TEAMS)
TOP-DOWN DESIGN
STRUCTURED DESIGN
BUILDS (A CSC TECHNIQUE)
(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 secretarial." In the last several years the Deep Space Net has issued a set of rigorous software development standards, as listed in the accompanying box. In addition, a JPL staff member, Robert C. Tausworthe, prepared a 379-page monograph which presents these practices in a logical, tutorial form.[1]

JPL took a low-key approach to introducing the new techniques.

Flight projects and the Mission Control Center, which is shared by the projects, have not specified software development methods to the same level of detail as the Deep Space Net. The particular way in which these methods are implemented is adapted by each project organization to the tools available on the computer used for each task. However, the use of a program design language, structured programming, top-down development, and related methods are de facto standards in the project areas. Since much of the programming is scientific or engineering in content, an early task was the development of SFTRAN, Structured-Programming-To-Fortran Translator. Recently a preprocessor to facilitate structured programming in assembly language has been developed.

So, although the laboratory does not insist upon completely standardized organization-wide programming practices, the underlying principles are common to all sections. The differing detailed procedures endorsed by various program offices are the result of differences in the kind of applications. For example, the Deep Space Network develops software to be operated at overseas sites by personnel who are not JPL employes but are permanent residents of the countries in which the stations are located. Conversely, flight projects utilize the same personnel who develop the software to operate, maintain, and modify it during the mission. Moreover, flight software is much more subject to frequent modification in response to changes in the mission than is the software in the Deep Space Net.

Personnel. By 1975 an informal census of a cross section of programmers found about 50 percent definitely favorable to the new practices, about 40 percent rather neutral, and about 10 percent hostile to them. However, it was not only the modern programming practices that were important in forming these attitudes, but also the environment in which the work was done—the accessibility of text editing, interactive computing, and tools and facilities that enabled the programmer to get his job done with a minimum of running around and standing in line. They like the programming secretariat and support concepts. This environment made their lives easier and more productive.

Still there is great variability in the individual productivity of programmers, perhaps by a factor of 10 to 1, or even more sometimes. The techniques of structured design do not, of course, change dunces into brilliant programmers. They do make it possible to break up a large project in such a way that the more competent personnel can be assigned to the difficult early stages and

Software Standard Practices

DEEP SPACE NETWORK STANDARD PRACTICES
- 810-13 (AUGUST 15, 1975) SOFTWARE IMPLEMENTATION GUIDELINES AND PRACTICES
- 810-16 (DECEMBER 15, 1975) PREPARATION OF SOFTWARE REQUIREMENTS DOCUMENTS
- 810-17 (JULY 15, 1976) PREPARATION OF SOFTWARE DEFINITION DOCUMENTS
- 810-19 (MARCH 1, 1977) PREPARATION OF SOFTWARE SPECIFICATION DOCUMENTS
- 810-20 (FEBRUARY 1, 1977) PREPARATION OF SOFTWARE OPERATOR'S MANUALS
- 810-21 (NOVEMBER 15, 1976) PREPARATION OF SOFTWARE TEST AND TRANSFER DOCUMENTS

PROJECT DOCUMENTS
- 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. KROGH, CHARLES L. LAWSON AND MICHAEL R. WARNER

the less creative people can be assigned to implement tasks that have been fairly well structured. The structured methods also enable managers to identify the real incompetents sooner than older methods did.

Advantages. The payoff from the adoption of modern programming practices has come in terms of significantly improved schedule performance and manpower productivity. There is no doubt that these techniques have resulted in dramatic improvements in cost, quality, and schedule. For example, one of the programs for the Voyager project—several hundred thousand lines of high-level language—was recently completed with labor expenditures within budget and computer time at something like half of budget. The reduction in computer time implies that the programs were of better quality, because the programmers had to do much less debugging and testing. In the entire Voyager software development program, just finished, only one "tiger team" had to be set up. That was in the data records area which had to be done in assembly language because a higher-level language was not available.

A recent Deep Space Net project, one that had been budgeted for two years, came in two weeks over budget, but this overage had been detected a year ahead and appropriate readjustments had been made. In this case, too, debugging and testing took half the time originally budgeted. The reason: very few errors. In fact, there were 350 errors, including those in documentation, in 300,000 lines of code—a rate of only 20 percent of previous experience. Perhaps of equal importance to those only too accustomed to nerve-wracking "tiger team" efforts to meet flight dates, this time there was no tiger team. There were no hassles and no one aggravated his ulcers. It seemed that software management was coming of age.

Moreover, the full extent of the savings from the use of the new practices will only become apparent several years into the maintenance phase. This phase is clearly going to be much more efficient. The programs will be more reliable, they will be easier to change when necessary. These improvements follow from the experience in the implementation phase. When this phase is completed more quickly and with fewer errors, there is more confidence that the program itself is closer to full correctness.

JPL is finding a greater percentage of the overall software development effort being concentrated at the front end where 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[4] 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.

Goulding and Lawson[5] 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 166 equivalent machine instructions per manhour, said to be "considerably

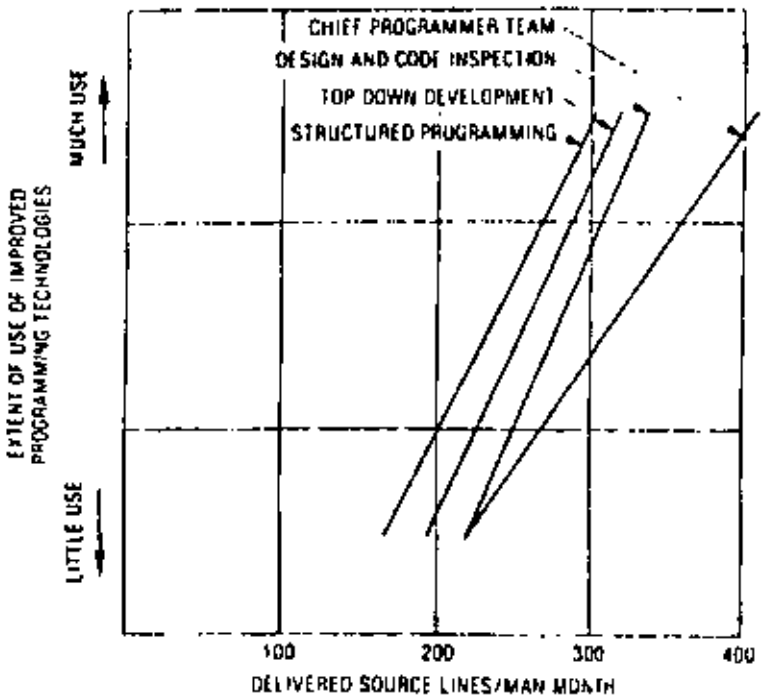


Figure 4 Production appears to increase dramatically with the use of improved programming technologies.

better" than the results on comparable real-time projects programmed by conventional methods.

Data management. Hsiao[11] reported the development of a highly secure data management system composed of 60 modules with 40K words of code. The total design and implementation were completed in 3 manyears with an elapsed time of less than 11 months. Using a chief programmer team, structured programming, and the other concepts of modern programming practices, he felt that the effort could not have been completed in such a short period of time without the use of these techniques. In addition, said Hsiao, extension of the system "has been greatly facilitated by the complete development-support library, clear system interface, high program modularity, and well-structured code."

Japanese findings. The Nippon Electric Company has modified its data base management system, programmed in Cobol, to take advantage of the concepts of structured programming.[12] This experience, plus some experiments in using the new techniques, has convinced them that the methods can be successfully applied to the forthcoming development of a very large on-line banking system. Their experiments indicate the following:

- number of steps programmed per working hour doubled,
- number of steps programmed per machine hour used tripled;
- number of database handling errors decreased 10 percent,
- other types of programming errors decreased 65 percent;
- average length of a bug find-and-fix cycle has been reduced to about one-third the conventional time.

These improvements were accompanied by small degradations in program performance.

Early assessment. Back in 1974, after only a few years of experience with the group of techniques that were then called structured programming, the IEEE Computer Society's Lake Arrowhead Workshop found that savings of "over 50 percent had been achieved, relative to previous performance on similar projects."[13] Companies providing data included IBM Federal Systems Division (40 percent average improvement over 20 projects), McDonnell-Douglas Automation Company (36 percent improvement on three projects; 5 percent on a fourth), and Hughes Aircraft Company (50 percent improvement on two real-time projects).

New practices judged effective

Software is in a predicament: it is too costly, too error-prone, too complex, too hard to maintain. This predicament may delay new applications of computers unless the effectiveness of software development can be substantially improved.

Modern programming practices are not the only way, of course. Higher-level programming languages, improved life cycle planning and management,* automated development tools, better personnel selection and training, and, of course, more sophisticated management will undoubtedly help.

The studies brought together here indicate that software effectiveness can be achieved. Where the results have been reduced to numbers, they ranged from about 25 percent to about 75 percent on such factors as cost, error reduction, and productivity.

Too much weight should not be placed on any one number. Studies of this kind are difficult to make and are usually not comparable from one organization to another, because definitions of terms vary. Moreover, averaging various amounts of knowledge—or degrees of ignorance—from interviews does not necessarily provide precise numbers. And summarizing detailed results, as has been done in this article, almost inevitably obscures the nuances of the original papers.

Rather, it is the overall judgment that is impressive: Whether from formal studies or the impressions of experienced executives, it seems clear that modern programming practices are effective in improving the processes of software development. They are ready for use. ■

*An article on life cycle planning is scheduled for the second quarter of 1978.

Acknowledgments

Interviews with the following experienced observers of software development helped guide me through the literature and practices of this field:

Walter R. Beam, Office of the Assistant Secretary of the Air Force for Research, Development, and Logistics

Joe M. Henson, IBM Data Processing Division

Robert Jirka, Michael R. Plesset, Edward C. Posner, and Michael R. Warner, all of Jet Propulsion Laboratory, California Institute of Technology.

References

1. Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays On Software Engineering*, Addison-Wesley Publishing Co., Reading, Massachusetts, 195 pp., 1974.

2. F. T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Sys J*, 11,1 (1972), pp. 56-73.

3. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972, 220 pp.

4. Harlan D. Mills, "Chief Programmer Teams, Principles, and Procedures," IBM Federal Systems Division Report FSC 71-5108, Gaithersburg, Maryland, 1971.

5. Harlan D. Mills, "Top Down Programming In Large Systems," in *Debugging Techniques in Large Systems*, R. Rustin (ed.) Prentice Hall, Englewood Cliffs, New Jersey, 1971, pp. 41-55

6. Robert M. McClure, "Software—The Next Five Years," *Digest of Papers, COMPCON Spring 76*, pp. 6-7.

7. John B. Holton, "Are The New Programming Techniques Being Used?" *Datamation*, July 1977, pp. 97-103

8. Barry W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 48-59.

9. Walter R. Beam, "Can Software Be More Like Hardware? Should It Be?" Keynote speech at COMPCON Fall 77, September 7, 1977 (not in *Digest of Papers*)

10. Edward W. Pullen and Robert G. Sunku, "Our Changing Industry," *Datamation*, January 1977, pp. 49-55.

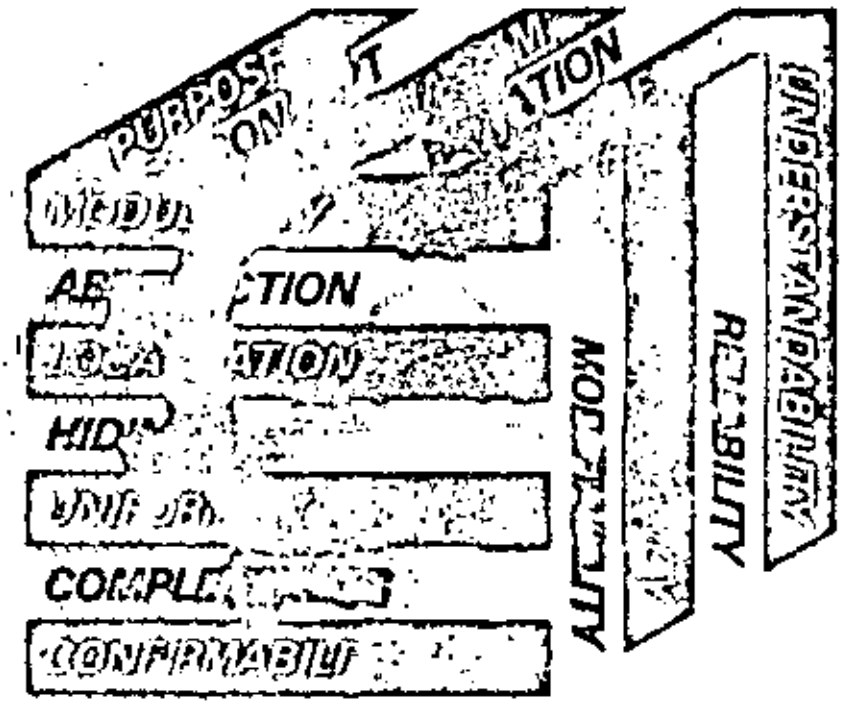
11. Joe M. Henson, "Computer Applications, Trends and Directions," Keynote speech at COMPCON Fall 77, September 7, 1977 (not in *Digest of Papers*)

12. "Business Software Makes Fee Problems, Hosler Charges," *Electronics*, November 24, 1977, p. 14.

13. J. A. 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 (42), 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.

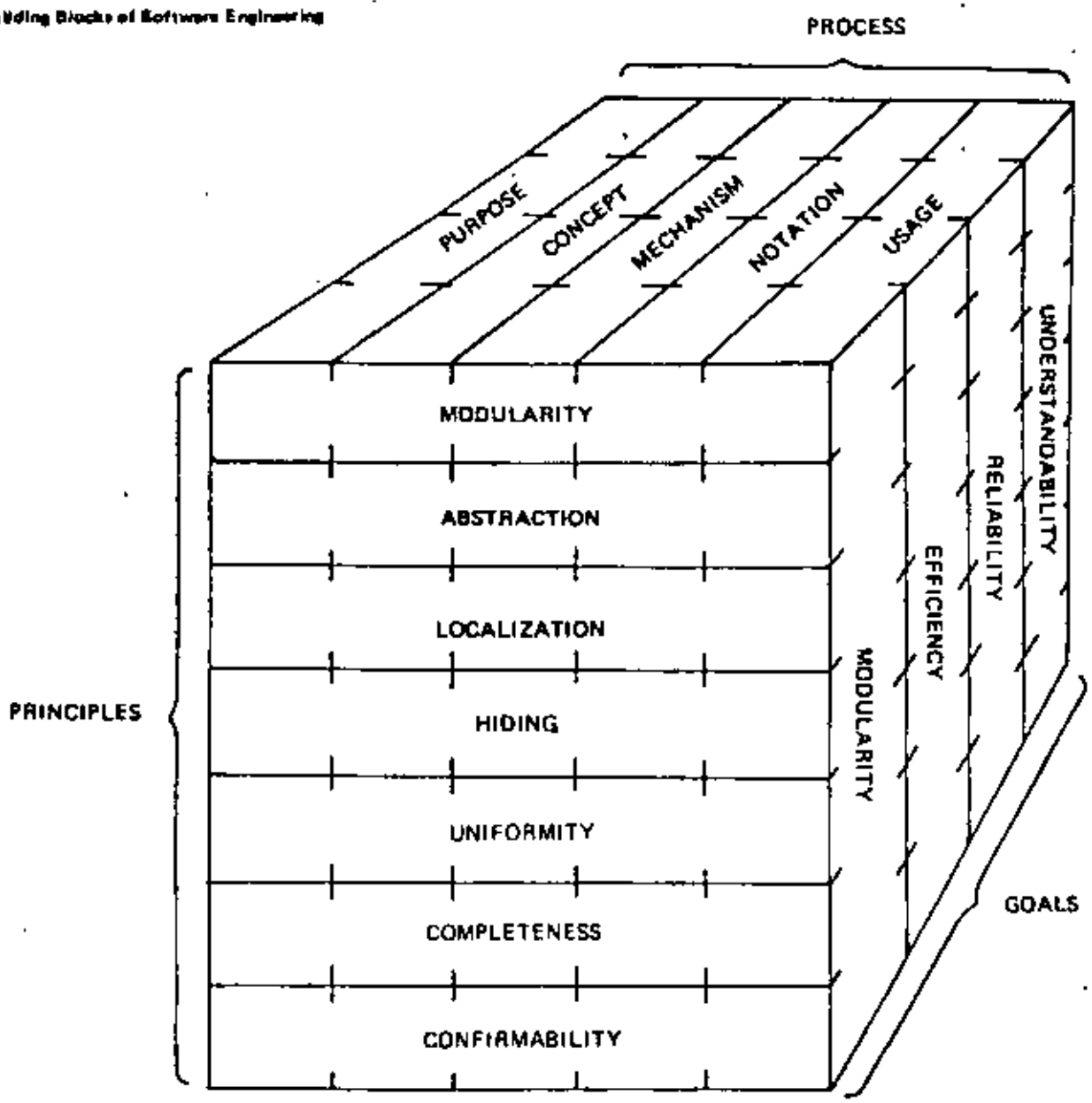
Introduction

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;

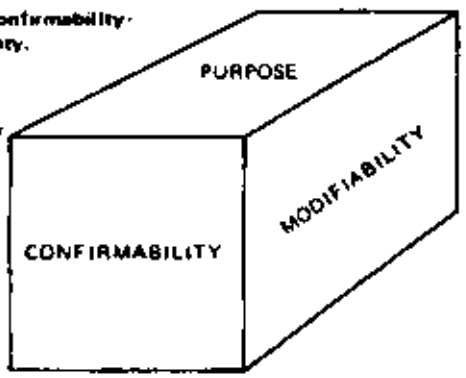
notation – the programming language in which the software will be written;

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.

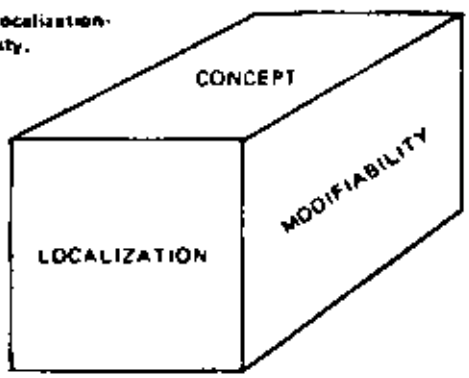
Given that part/whole hierarchical decomposition plays a pervasive role, the fundamental process interacts further with the various principles and goals of software engineering as depicted in Figure 1. Figure 1 is our framework for discussing the nature and issues of software engineering. Clearly an 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.



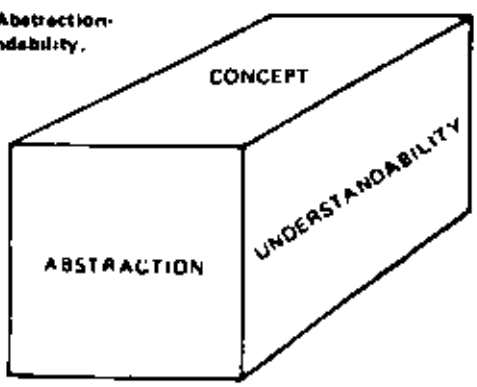
- The purpose of a design choice may be to structure a system so the effects of a change can be predicted with assurance.
- Confirmability applied to the process of defining purposes for modifiability demands that purposes be stated in a form that makes it possible to easily check if they have been achieved, 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 as clearly confirmable, and hence, is not so useful.

Figure 3. Concept-Localization-Modifiability.



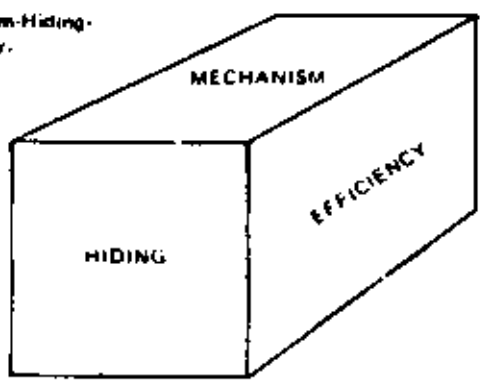
- Identifying one module for implementing a scheduling policy in an operating system is an example of the effect of the localization principle on design concepts to enhance modifiability, since localizing the policy rather than scattering policy decisions throughout a system makes it easier to change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

Figure 4. Concept-Abstraction-Understandability.



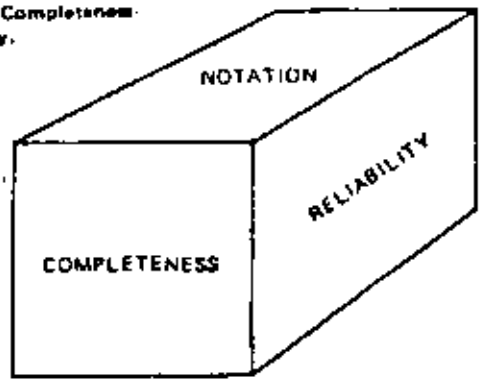
- The use of levels of abstraction[5,11,20] to define an understandable program or system of programs shows how abstraction can make designs more understandable.
- High-order languages represent a concept for improving the understandability of programs by abstracting from the details of computer instruction sets.

Figure 5. Mechanism-Hiding-Efficiency.



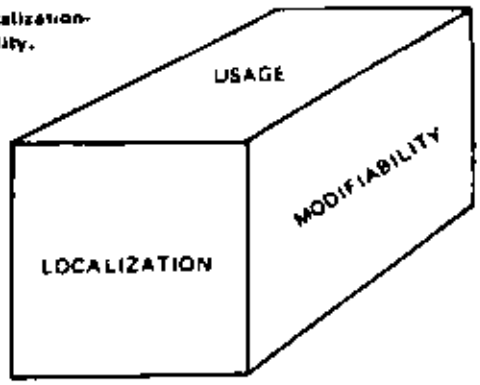
- Knowing that only certain subroutines have access to shared data (e.g., as in a function cluster[13]) may permit fewer checks to be made on the validity of stored values, and thereby increase efficiency.
- Forbidding users from directly accessing I/O devices permits an operating system to optimize 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 the possibility, and hence is not encouraged to develop error recovery algorithms. A complete notation can foster reliability by permitting a programmer to specify error recovery details.

Figure 7. Usage-Localization-Modifiability.



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

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

Hierarchical Decomposition

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

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

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

The Goals of Software Engineering

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

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

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

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

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

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

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

The Principles of Software Engineering

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The principle of modularity can be further illustrated by considering modularity issues arising in deciding what part/whole relationships should be considered in developing hierarchical decompositions:

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

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

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

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

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

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

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

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

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

Hiding Another principle familiar to many readers is *hiding*. Parnas,[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[14] 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

THE MYTHICAL MAN-MONTH

HOW DOES A PROJECT GET TO BE A YEAR LATE? . . . . . ONE DAY AT A TIME.

By Frederick P. Brooks, Jr.

NO SCENE FROM PREHISTORY is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and saber-toothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.

Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it.

More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this case of disaster so common?

First, our techniques of estimating are poorly developed. More seriously, they reflect an invoiced 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

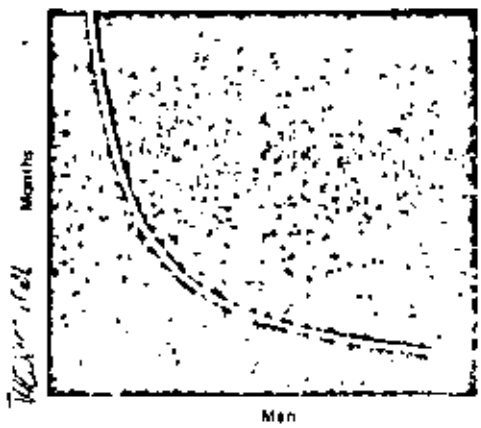


Fig. 1. The term "man-month" implies that if one man takes 10 months to do a job, 10 men can do it in one month. This may be true of picking cotton.

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 build-up of 30% per year. More than that strains and even inhibits the evolution of the essential informal structure and its communication pathways. D. 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.

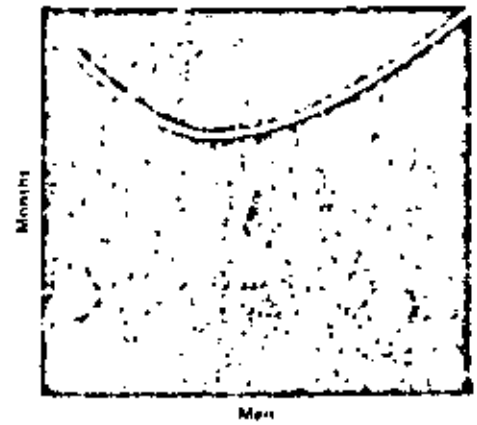


Fig. 3. Since software construction is complex, the communications overhead is great. Adding more men can lengthen, rather than shorten, the schedule.

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.

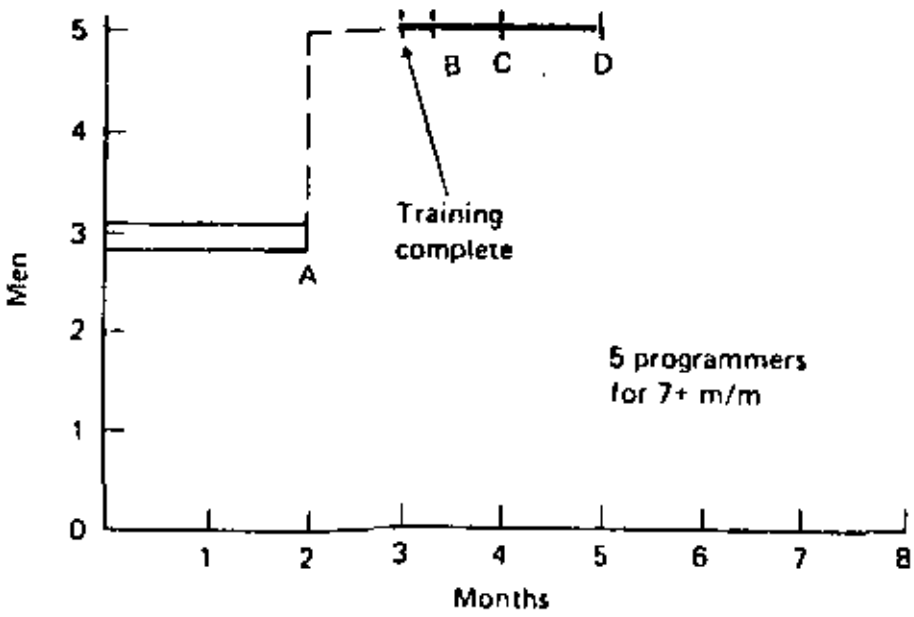


Fig. 4 Adding manpower to a project which is late may not help. In this case, suppose three men on a 12 man-month project were a month late. If it takes one of the three an extra month to train two new men, the project will be just as late as if no one was added.

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

that ... is done, and that rescheduling will not have to be done again.

- 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 sprint 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.

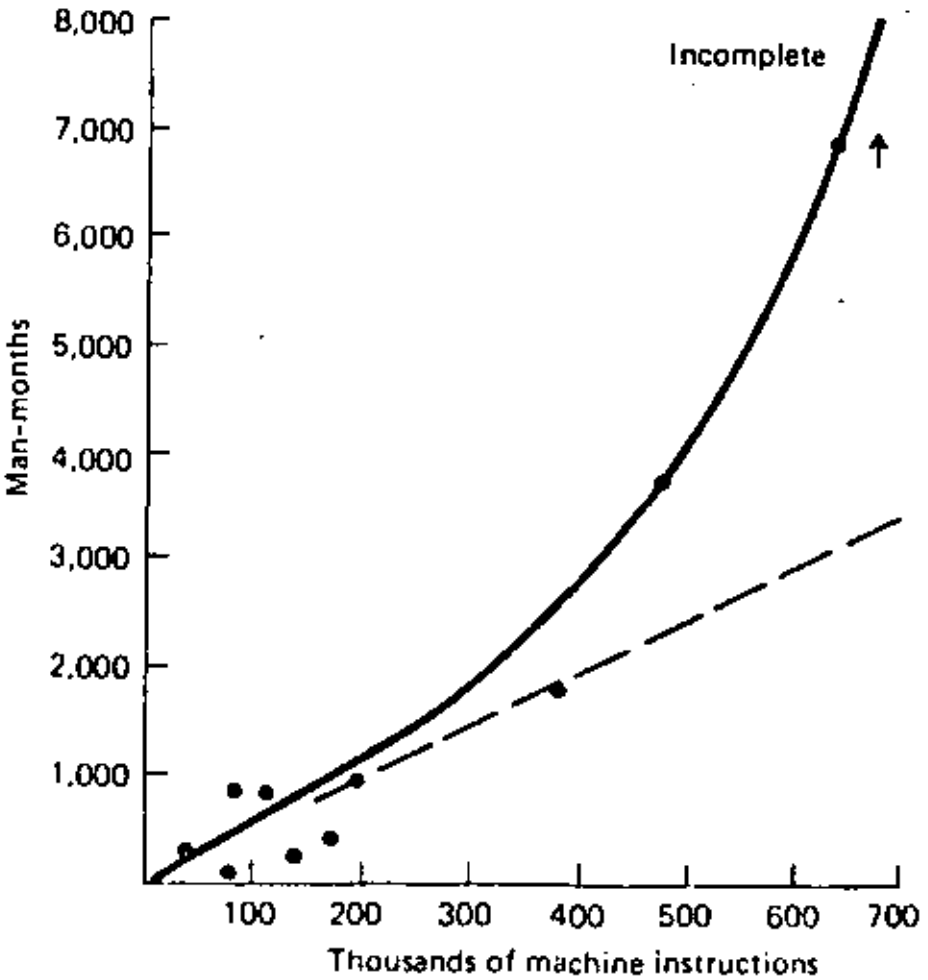


Fig. 5 As a project's complexity increases, the number of man-months required to complete it goes up exponentially.

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 = (constant) × (number of instructions)$^{1.5}$

Another SDC study reported by Weinwurm[3] also shows an exponent near 1.5.

A few studies on programmer productivity have been made, and several estimating techniques have been proposed. Morin has prepared a survey of the published data.[4] Here I shall give only a few items that seem especially illuminating.

Portman's data

Charles Portman, manager of ICL's Software Div., Computer Equipment Organization (Northwest) at Manchester, offers another useful personal insight.

He found his programming teams missing schedules by about one-half—each job was taking approximately twice as long as estimated. The estimates were very careful, done by experienced teams estimating man-hours for several hundred subtasks on a PERT chart. When the slippage pattern appeared, he asked them to keep careful daily logs of time usage. These showed that the estimating error could be entirely accounted for by the fact that his teams were only realizing 50% of the working week as actual programming and debugging time. Machine downtime, higher-priority short unrelated jobs, meetings, paperwork, company business, sickness, personal time, etc. accounted for the rest. In short, the estimates made an unrealistic assumption about the number of technical work hours per man-year. My own experience quite confirms his conclusion.

An unpublished 1964 study by E. F. Bardain shows programmers realizing only 27% productive time.[5]

	Prog units	Number of programmers	Years	Man-years	Program words	Words/ man-yr.
Operational	60	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	2¼	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.

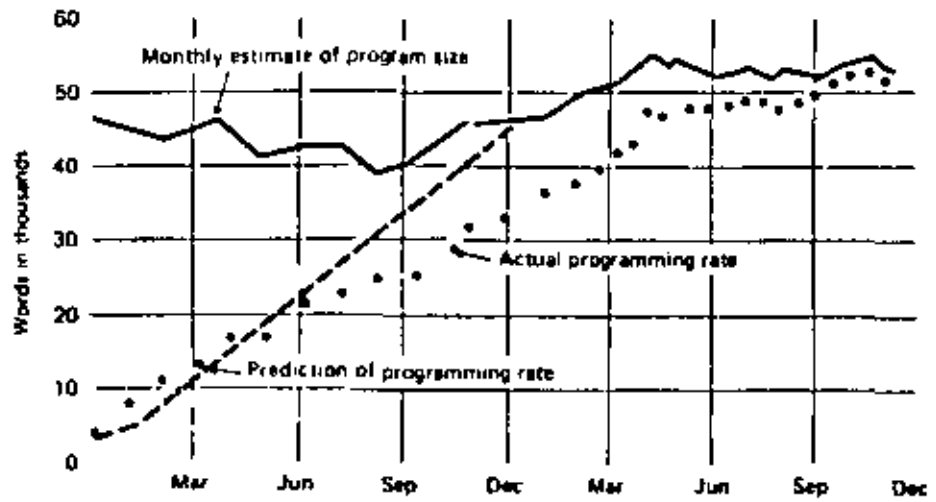


Fig. 6. Bell Labs' experience in predicting programming effort on one project.

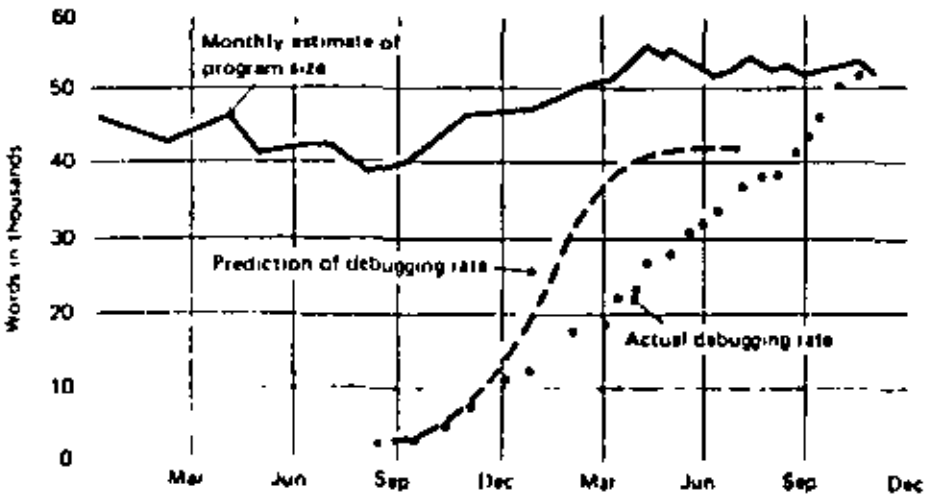


Fig. 7. Bell's predictions for debugging rates on a single project, contrasted with actual figures.

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 interchange	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. Taliaferro 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. *Under*estimates 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.

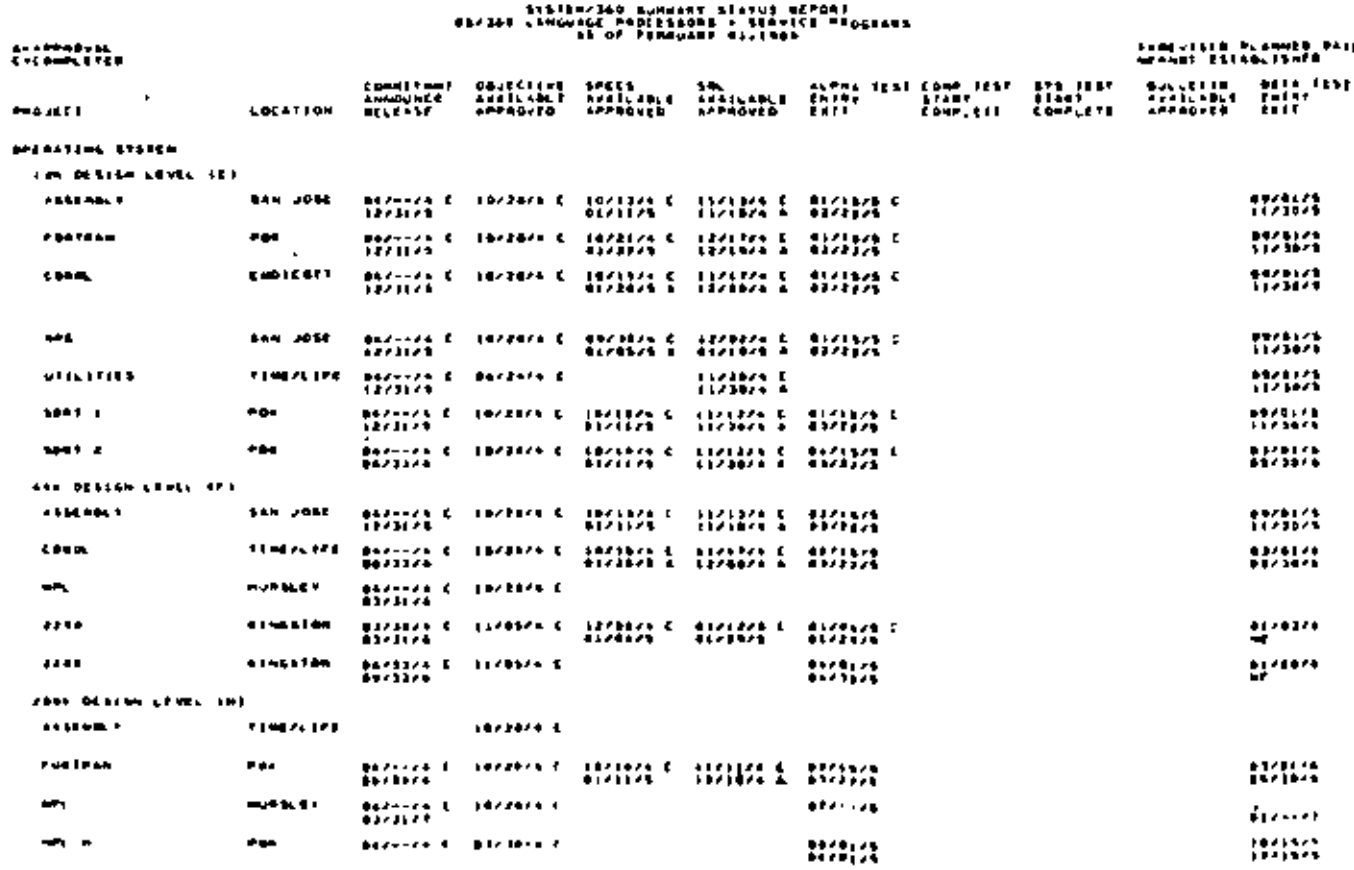
But every boss needs two kinds of information, exceptions for action and a status picture for education.[12] For that purpose he needs to know the status of all his teams. Getting a true picture of that status is hard.

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

PROJECT	LOCATION	COMMITMENT ANNOUNCE RELEASE	OBJECTIVE AVAILABLE APPROVED	SPECS AVAILABLE APPROVED	SPL AVAILABLE APPROVED	ALPHA TEST ENTRY EXIT	CODE TEST START COMP.CEI	SYS TEST START COMPLETE	BULLETIN AVAILABLE APPROVED	BETA TEST ENTRY EXIT
OPERATING SYSTEM										
(65 DESIGN LEVEL 13)										
ASSEMBLY	SAN JOSE	06/--/65 C 12/31/65	10/30/64 C	10/13/64 C 01/11/65	11/13/64 C 11/18/64 A	01/15/65 C 02/28/65				09/01/65 11/30/65
FORTRAN	POK	06/--/65 C 12/31/65	10/30/64 C	10/21/64 C 01/28/65	12/11/64 C 12/18/64 A	01/15/65 C 02/23/65				09/01/65 11/30/65
COBOL	CAMBRIDGE	06/--/65 C 12/31/65	10/30/64 C	10/13/64 C 01/30/65 E	11/11/64 C 12/18/64 A	01/15/65 C 02/23/65				09/01/65 11/30/65
MFS	SAN JOSE	06/--/65 C 12/31/65	10/30/64 C	09/30/64 C 01/06/65 E	12/09/64 C 01/18/65 A	01/15/65 C 02/23/65				09/01/65 11/30/65
UTILITIES	TIMES/LIFE	06/--/65 C 12/31/65	06/30/65 C		11/30/64 C 11/30/64 A					09/01/65 11/30/65
SORT 1	POK	06/--/65 C 12/31/65	10/30/64 C	10/13/64 C 01/11/65	11/13/64 C 11/30/64 A	01/15/65 C 02/23/65				09/01/65 11/30/65
SORT 2	POK	06/--/65 C 06/30/65	10/30/64 C	10/13/64 C 01/11/65	11/13/64 C 11/30/64 A	01/15/65 C 02/23/65				03/01/65 05/30/65
44K DESIGN LEVEL 01)										
ASSEMBLY	SAN JOSE	06/--/65 C 12/31/65	10/30/64 C	10/13/64 C 01/11/65	11/13/64 C 11/18/64 A	01/15/65 02/28/65				09/01/65 11/30/65
COBOL	TIMES/LIFE	06/--/65 C 06/30/65	10/30/64 C	10/13/64 C 01/30/65 A	11/13/64 C 12/18/64 A	01/15/65 02/23/65				03/01/65 05/30/65
MFS	MORSLEY	06/--/65 C 03/31/65	10/30/64 C							
2250	KINGSTON	03/30/65 C 03/31/65	11/05/64 C	12/30/64 C 01/06/65	01/13/65 C 01/30/65	01/15/65 C 01/29/65				01/30/65 HP
2280	KINGSTON	06/30/65 E 09/30/65	11/05/64 C			03/01/65 03/15/65				01/30/65 HP
2250 DESIGN LEVEL 10)										
ASSEMBLY	TIMES/LIFE		10/30/64 C							
FORTRAN	POK	06/--/65 C 09/30/65	10/30/64 C	10/13/64 C 01/11/65	11/13/64 C 12/18/64 A	02/15/65 03/23/65				03/01/65 04/30/65
MFS	MORSLEY	06/--/65 C 03/31/65	10/30/64 C			03/--/65				01/--/-- HP
MFS M	POK	06/--/65 C	01/30/65 C			08/01/65 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 (SALT) 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.

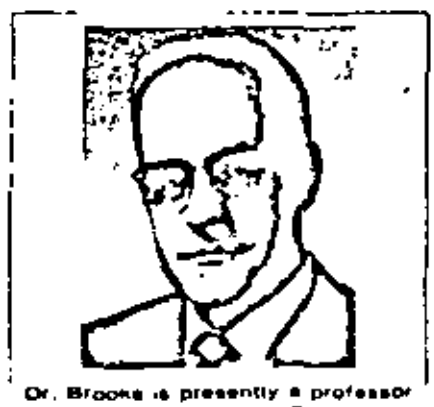
The investment of a modest amount of skilled effort in a Plans and Controls function is very rewarding. It makes far more difference in project accomplishment than if these people worked directly on building the product programs. For the Plans and Controls group is the watchdog who tenders the imperceptible 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 Experimental 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. Wennum, G. F., *Research in the Management of Computer Programming*, Report SP-3059, 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. Corbato, F. J., *Sensitive Issues in the Design of Multi-User Systems*. Lecture at the opening of the Honeywell EDP Technology Center, 1968.
8. Taliaferro, W. M., "Modelocity 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. Kidd, W. R., and T. A. Wilson, "Subjective Time Estimates in Critical Path Planning—a Preliminary Analysis," *Management Sciences*, 11 (1965), 307-320, and reply, W. R. Kidd, D. M. Waterbrook, and R. D. Hexel, "On the Analysis of Critical Path Time Estimating Behavior," *Management Sciences*, 14 (1967), 79-84.
12. Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*, New York: Wiley, 1969, pp. 428-430. □



Dr. Brooks is presently a professor at the Univ. of North Carolina at Chapel Hill, and chairman of the computer science department there. He is best known as "the father of the IBM System/360," having served as project manager for the hardware development and as manager of the Operating System/360 project during its design phase. Earlier he was an architect of the IBM Stretch and Harvest computers.

At Chapel Hill he has participated in establishing and guiding the Triangle Universities Computation Center and the North Carolina Educational Computing Service. He is the author of two editions of "Automatic Data Processing" and "The Mythical Man-Month: Essays on Software Engineering" (Addison-Wesley), from which this excerpt is taken.

In contrast to Dr. Brooks' presentation, this portrait of failure is for those who learn best from looking at bad examples.

WHY PROJECTS FAIL

by Stephen P. Kaider

ONE OF THE PRIMARY causes for the failure of data processing projects is that such projects are often not initially defined, and therefore may lack a beginning and an end. Once a project has begun, no one seems to know:

- how the project was started;
- what the staffing is, or was, at any one point in time;
- what activities have been performed;
- when the project will end;
- what the project will accomplish.

Essentially, because projects are rarely formally defined, they are rarely completed. Completion occurs usually upon the death—or resignation—of the user the project services, or when the system is due for conversion. Completion is also a prerequisite for success, but a project is considered successful only if completed within the original time or budget estimates, and by how well it satisfies the user's needs.

An unsuccessful project, however, can be identified during several phases of its life cycle, and I shall here try to point to those very indicators.

Logically, any project can be time-divided into five distinct phases:

a) Pre-initiation period (usually measured in weeks or months)
b) Initiation period (measured in weeks)
c) Project duration (in months or years)
d) Project termination period (in weeks or months)
e) Post-termination period (occurring several months after project termination)

In each of the above phases, errors of commission or omission can have major impact upon the success of the total project.

Pre-initiation period

1) No standards exist for estimating how long the project will take. That is, each project is treated as a new and novel system with some individual responsible for estimation. His estimate will be based upon his own understanding of the project and its tasks, and on how quickly *he* can accomplish the subtasks. Little use is made of a history file of similar projects and actual versus originally estimated times.

2) Estimation is not done by the probable project leader, but rather, by whoever happens to be available at estimating time.

3) The project is not adequately defined. The request for an estimate usually takes the form of "John, we're planning to redo the payroll system. What do you think it will require?" "Payroll" may mean a number of different things to different people. Does it involve labor distribution? personnel information? leave accounting? salary, hourly and executive payroll? Any of the above can measurably impact the estimate of the project.

4) Short lead times are allowed for estimates, with corresponding inaccuracy as a result.

5) Personnel availability for the project is unknown. Estimates are usually prepared irrespective of who will perform the work. That is, an estimate of 36 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 dp 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

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:

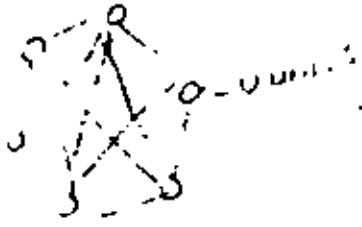
- It has a beginning.
- It has an end.
- It uses multiple, finite resources.
- It has an objective.
- Its success can be measured in terms of time or dollars.
- It requires a leader.
- It requires a staff.
- It must be planned.
- Performance against plan must be reviewed.
- It coexists with other projects 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.

from Brooks

Communications	Users	Lines	Interactions
	1	0	0
	2	1	1
	3	3	6
⋮	5	10	20



THIS PAGE INTENTIONALLY LEFT BLANK.

n	$\frac{n(n-1)}{2} \sim n^2$	n^2

Limit of no more than 7 interactions

1.- ESTIMACION DE COSTOS.

The Cost of Developing Large-Scale Software

RAY W. WOLVERTON, MEMBER, IEEE

Abstract—The work of software cost forecasting falls into two parts. First we make what we call structural forecasts, and then we calculate the absolute dollar-volume forecasts. Structural forecasts describe the technology and function of a software project, but not its size. We allocate resources (costs) over the project's life cycle from the structural forecasts. Judgment, technical knowledge, and econometric research should combine in making the structural forecasts. A methodology based on a 25 × 7 structural forecast matrix that has been used by TRW with good results over the past few years is presented in this paper. With the structural forecast in hand, we go on to calculate the absolute dollar-volume forecasts. The general logic followed in "absolute" cost estimating can be based on either a mental process or an explicit algorithm. A cost estimating algorithm is presented and five tradition methods of software cost forecasting are described: top-down estimating, similarities and differences estimating, ratio estimating, standards estimating, and bottom-up estimating. All forecasting methods suffer from the need for a valid cost data base for many estimating situations. Software information elements that experience has shown to be useful in establishing such a data base are given in the body of the paper. Major pricing pitfalls are identified. Two case studies are presented to illustrate the software cost forecasting methodology and historical results. Topics for further work and study are suggested.

Index Terms—Computer programmer code productivity rates, cost data base used in cost estimation mechanism, cost of developing custom software, incentive fee structure influence on computer software development, principles of pricing computer software proposals, resource allocation in computer program development, software cost estimating methods and general logic, software cost estimation algorithm, software development and test life cycle—cost impact, systems approach to pricing large-scale software projects.

INTRODUCTION

ESTIMATING the cost of a large-scale computer program has traditionally been a risky undertaking. Now that software has been with us for nearly two decades, it is reasonable to hope that we have learned something that would make our predictions less unreliable. At TRW we have been motivated to make a serious effort toward improving our software estimating methods for large-scale software systems, for a set of very compelling reasons.

1) Our customers have shown a growing unwillingness to accept cost and schedule overruns unless the penalties were increasingly borne by the software developer.

2) Partly as a consequence of 1), we have entered into software development contracts where both adherence to predicted costs and on-time delivery were incentivized. That means we made more money if we could predict accurately the cost and the time it would take to do the job.

3) We found that we could improve our estimates only by improving our understanding of exactly what steps and processes were involved in software development, and this understanding enabled us to manage the effort better. The better management, in turn, improved our estimates.

This paper presents the essential results of our efforts to improve our software cost estimating techniques. It should be understood that the specific contracts that are used here were government contracts whose particular provisions shaped those efforts to some degree. Nevertheless, we feel that much of what we learned is of general interest and can be applied or adapted to nearly any software development program of any size.

We can start with a review of some of the ways in which software development differs from hardware development, and which have traditionally contributed to the problem of estimating software costs. One of these ways is the problem of managing the people. The nature of programmers is such that interesting work gets done at the expense of dull work, and documentation is dull work. Doing the job in a clever way tends to be a more important consideration than getting it done adequately, on time, and at reasonable cost. Programmers tend to be optimistic, not realistic, and their time estimates for task completion reflect this tendency. The software engineer, the mathematician, and the programmer have trouble communicating. Visible signs of programming progress are almost totally lacking.

Another set of difficulties arises from the nature of the product. There are virtually no objective standards or measures by which to evaluate the progress of computer program development. Software characteristics are often in conflict, requiring frequent tradeoffs among such factors as core storage requirement versus tight code, fast execution time, external storage to minimize *I/O* time, maintainability by the eventual user, flexibility or adaptability to new needs, accuracy and reliability, and self-check and fail-safe modes of operation. The applications software developer finds himself in a dynamically evolving and constantly changing environment.

Software planning has problems stemming from the natural desire to get going, which means taking shortcuts. What are the steps that should precede the start of coding, and how does the knowledgeable manager allocate his limited resources? Planning also is made difficult by the

Manuscript received April 30, 1973, revised May 4, 1973.
The author is with the TRW Systems Group, Redondo Beach, Calif. 90278.

problem of translating the customer's functional requirements into software requirements, and later defining the procedures by which you determine that those original customer requirements have been met. It is not always easy to determine when you are through and whether you have delivered what you said you were going to deliver.

The overall approach we have developed for dealing with these problems and producing reasonably reliable cost estimates for large-scale software systems consists of the following basic elements, which will be discussed in detail in the remainder of this paper.

1) Understanding exactly what the software development process consists of over its life cycle.

2) Recognizing the pitfalls in the pricing of software.

3) Establishing and maintaining a good data base reflecting our history of actual software development costs.

4) Determining the most cost-effective allocation of resources over the entire period of the expected contract.

THE SOFTWARE DEVELOPMENT CYCLE

Because software is during most of its development cycle and even afterwards a basically intangible product, it is even harder to control than a complex hardware system. We have learned by trial and error that no cost projections can be fulfilled unless the mechanism for management control is solved in advance. We have established two management principles that apply throughout the software development cycle to reduce the problem of control to manageable size.

1) Produce software documentation that meets established programming standards, is accurate, is produced according to the needs of the audience who will use it, and is most of all, an instrument by which management controls the project.

2) Conduct technical reviews against predetermined acceptance criteria to the end that, upon customer approval predetermined baselines are established.

3) Control software physical media (tapes, disks, decks) to assure use of a known configuration in testing, demonstration, certification, and delivery.

4) Apply software configuration management controls and procedures to assure that required changes to baselines are implemented, tested, fully documented, and understood.

5) Provide a data reporting, repository, and control system to assure that all problems, deficiencies, change proposals, and software configuration data are analyzed, reported, recoverable, and available to all project and user personnel when and where needed.

Technical reviews are required during the software development life cycle. Reviews ensure that the resulting software products meet the requirements established for that level of completeness and understanding. A "baseline" is established at designated points in time during the software development cycle when the requirements (or design) have been defined, documented, reviewed, and approved. From that time forward, that baseline serves as the reference point for evaluating and managing any changes in scope, price, or schedule.

In the specific cases to be discussed, certain of these principles were incorporated in government software procurement policies applied to the contract, but no manager of a large software development should ignore them whether they are required by his customer or not. For example, without software configuration management—configuration identification, configuration control, configuration status accounting, and verification—the software development manager will spend much of his time solving problems that need not have arisen. Software is controlled through control of documentation within a structured software development process.

There are several ways of analyzing the software development process, but the one we have chosen leads to the definition of seven phases, or steps, each ending in a discrete event or document (see Fig. 1). The steps are basically sequential, but in practice there are iterations between adjacent steps and even between any of the steps. The seven steps are as follows.

Step 1 Performance and Design Requirements: This document establishes the requirements for performance, design, test, and qualification of the software at the system level.

Step 2 Implementation Concept and Test Plan: This document provides, at an early date, the preliminary design concepts that meet the requirements, and identifies the preferred approaches, design tradeoffs, and alternatives, and leaves no high-risk technology area unresolved.

Step 3 Interface and Data Requirements Specification: This document defines the interfaces between subsystems and major elements within subsystems, including presentation of table and file structure, data origin, destination, and update characteristics.

Step 4 Detailed Design Specification: This document provides, in precise detail, the complete design and acceptance specifications for the software at the routine level, which includes detailed flow charts, equations, and logic. It is the source material from which the program is coded.

Step 5 Coding and Debugging: Upon design review and approval of the Step 4 documentation package, which may be staggered in time and organized into volumes dealing with a single computer program module, the coding of the software can begin. This step is devoted to coding and debugging (and amending documentation) using debugging aids and desk check of the first computations. The phase terminates for a given module when it has passed the development-level verification tests imposed by the designer, and he turns the module over for assembly onto the master tape for the beginning of independent system validation testing.

Step 6 System Validation Testing: As the software

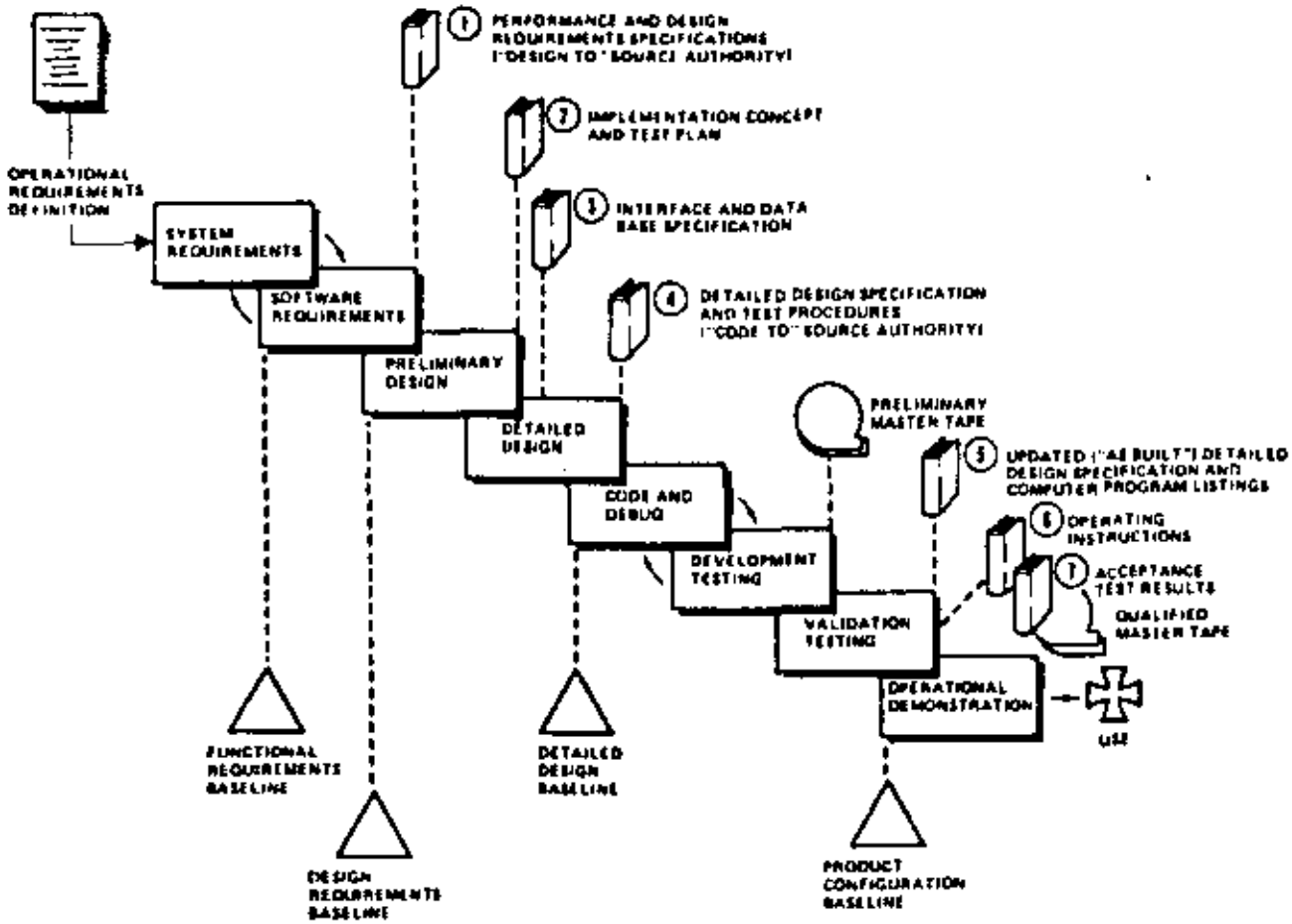


Fig. 1. Typical custom software development and test steps, showing seven unique documents.

modules are released by the design group for assembly onto the master tape, the buildup of the software package into successively more complex interfaces and capabilities can commence. When two or more modules are assembled, the first stages of "system testing" can get underway according to the test plan started at Step 2 and upgraded in level of detail at Step 4 to include test procedures and quantitative acceptance criteria. The phase terminates when all modules have been successfully tested against the predetermined acceptance criteria at the system level.

Step 7—Certification and Acceptance Demonstration: This final step is devoted to formal acceptance of the software system by subjecting it to previously defined acceptance and test specification procedures, which are executed in as near an operational environment and host computer configuration as possible. The phase terminates with successful "operational demonstration," witnessed by the customer and his quality assurance team. The software packages are then ready for installation, integration, and checkout in the operational environment.

The final step in the software development life cycle is the forerunner of the operations and maintenance phase, strictly speaking, Step 8. This phase is concerned with installation, integration, and checkout in the full operational configuration. The software contractor assists in any way required to integrate the software products of the development, or acquisition, phase into an operational software package that meets the original (and updated) performance and design requirements, the product of Step 1 and subsequent updates. The activities of this phase will also include training, rehearsal support, software problem reporting, fault isolation and correction, formal problem closure and documentation update, and finally, first mission operations support. Because this is not part of the software development effort, we cost it differently—usually as a level of effort.

PRICING PITFALLS

Pricing is the complete process by which we arrive at a price that we quote to the customer. Cost estimating is a major part of that process, although it frequently has to be done several times before a final price is arrived at. The general principles involved in pricing large R&D efforts of any kind have been defined by Beveridge [1], and apply to large software developments as well. Beveridge's discussion centers around the preparation of a proposal in response to a government Request for Proposal (RFP), but his common-sense principles are

applicable to any complex pricing exercise where many individuals are involved.

There is a general tendency on the part of designers to gold-plate their individual parts of any system, but in the case of software the tendency is both stronger and more difficult to control than in the case of hardware. A major task of management is to make certain that the design being proposed (and priced) does meet the customer's need, and that each individual component design can be traced to a specific need, while at the same time it does not provide more (more speed, more accuracy, less need for core, etc.) than the customer needs or wants. Anything being offered a customer beyond what he has asked for should be clearly identified (and priced) as an option. That is what is meant by the "fixed-price attitude": what is the absolute minimum I can do to satisfy the needs stated in the RFP?

SOFTWARE COST ESTIMATING METHODS

The software industry is young, growing, and marked by rapid change in technology and application. It is not surprising, then, that the ability to estimate costs is still relatively undeveloped. Even though many organizations are trying to devise more scientific and objective means for estimating costs of large software systems, the present state of the art is largely judgmental. We will, however, discuss certain important exceptions shortly. A review of other firms and agencies confirm that we are facing a problem common to all [2]-[5].

Estimating the cost of producing computer software relies heavily on the judgment of experienced performers. The software analyst, or estimator, normally breaks the total job into elements that are estimated separately and then summarized into an estimate for the total job. The estimating analysis and synthesis may appear as a mental process or may involve an explicit algorithm [6].

In either case, an empirical data base is used as an objective reference, and the estimator uses his judgment to account for differences. Pieces of information used in the comparisons and adjustments for differences include: a) analysis of initial requirements; b) allocation of requirements to software modules; c) estimates of number of object instructions per module; d) complexity and technological risk; e) user environment and characteristics of the customer; f) computer of choice and interchangeability among other user sites; g) higher order language of choice or criteria for use of assembly language; h) type of software to be developed; i) whether software is to be delivered to an operational user, j) technical experience on that type of job, k) capabilities of the member of the technical staff who probably will do the work, l) type of fee and its incentive structure, m) length of development time; n) single-model multiple-release versus multiple-model single-release development concept; o) performance record of other large-scale systems (AWAC, SAGE, etc.) in the number of instructions and development man-months; and p) management factors to do with productivity rates, error rates, work environment, availability of computer time, and many other variables.

Most estimators use a logical sequence in establishing their estimates. The logic uses exchange coefficients between some measurable parameter and its cost and adjustment factors derived from experience. One of the pitfalls is that estimating ratios should be directly traceable to recorded cost facts (not hearsay), and even the factual data can contain varying and unstated allowances for risk (we may have assumed a 40-h work week, our records show we charged that rate, but our records do not show that most personnel worked 45-50 h/week or more). Our long-term objective at TRW has been to create a systematic software development estimation system that can significantly reduce statistical variances between estimated costs and actual costs, and is not only accurate in that sense but also simple, fast, and convenient. We will briefly look at estimation methods in general, and two in particular that have been developed and reduced to practice at TRW.

General Logic Followed in Estimating

Traditional cost estimating procedures start with fixing the size of each activity, its start date, and duration. When necessary, adjustments are made to account for the caliber of performer personnel to be assigned, risk, complexity, uncertainties in requirements, and so on. Finally, the amount and type of manpower (man-months per month) and computing resources (hours per month) are converted to dollar costs by applying bid rates. Other direct charges (documentation costs, travel, etc.) are added, and summaries are made through the pricing system. Traditional methods can be classified as one or more of the techniques described below.

1) *Top-Down Estimating:* The estimator relies on the total cost or the cost of large portions of previous projects that have been completed to estimate the cost of all or large portions of the project to be estimated. History coupled with informed opinion (or intuition) is used to allocate costs between packages. Among its many pitfalls is the substantial risk of overlooking special or difficult technical problems that may be buried in the project tasks, and the lack of details needed for cost justification.

2) *Similarities and Differences Estimating:* The estimator breaks down the jobs to be accomplished to a level of detail where the similarities to and differences from previous projects are most evident. Work units that cannot be compared are estimated separately by some other method.

3) *Ratio Estimating:* The estimator relies on sensitivity coefficients or exchange ratios that are universal (within limits) to the details of design. The software analyst estimates the size of a module by its number of object instructions, classifies it by type, and evaluates its relative complexity. An appropriate cost matrix is constructed

from a cost data base in terms of cost per instruction, for that type of software, at that relative complexity level. Other ratios, empirically derived, can be used in the total estimation process, for instance, computer usage rate based on central processing unit (CPU) time per instruction, peripheral usage to CPU usage, engineers per secretary, and so forth. The method is simple, fast, convenient, and useful in the proposal environment and beyond. It suffers, as do all methods, from the need for a valid cost data base for many estimating situations (business versus scientific, real-time versus nonreal-time, operational versus nonoperational).

4) *Standards Estimating:* The estimator relies on standards of performance that have been systematically developed. These standards then become stable reference points from which new tasks can be calibrated. Many mature industries, such as manufacturing and construction, use this method routinely. The method is accurate only when the same operations have been performed repeatedly and good records are available. The pitfall is that custom software development is not "performed repeatedly."

5) *Bottom-Up Estimating:* This is the technique most commonly used in estimating government research and development contracts. The total job is broken down into relatively small work packages and work units. The work breakdown is continued until it is reasonably clear what steps and talents are involved in doing each task. Each task is then estimated and the costs are pyramided to form the total project cost. An advantage of this technique is that the job of estimating can be distributed to the people who will do the work. A difficulty is the lack of immediate perspective of the most important parameter of all: the total cost of the project. In doing detailed estimates, the estimator is not sensitive to the reasonableness of the total cost of the software package. Therefore, top-down estimation is used as a check on the bottom-up method.

The estimation process is nearly always a combination of two or more of the basic classifications given above. We will briefly discuss two methods that have been used in estimating the cost of developing large-scale software at TRW, a method we used in June 1969 which we will call "smoothing and extrapolation," and the current method we used beginning in October 1970, which we will call "a software cost estimation algorithm."

A Systems Approach to Software Cost Estimation

To be as free as possible to deal with the cost estimating method, we will use hypothetical or normalized numbers where numbers are necessary. Assume the software development and test life cycle consists of the seven steps previously defined, except that the starting point at contract go-ahead is immediately on completion of Step 2. We need to understand this assumption clearly, since it establishes the remaining-milestone events which are included in the cost which was negotiated with the customer. In other words, the proposal was at a level of detail such that it was, by itself, the implementation concept and test plan, Step 2. The RFP package, together with the bidder's briefing, was very detailed, technically complete, and was, by itself, the performance and design requirements, Step 1.

Our reference project for the most recent past successful software project was identified, and its case history was laid out in terms of the actual events, costs, start dates, and durations. The normalized contract cost as a function of the normalized period of performance is shown in Fig. 2. We will call this case history *A*. The cost distribution by development activity, that is, the actions that characterize the intervals between discrete milestone events, is shown in Fig. 3. Using case history *A* we can identify similarities and differences with respect to our cost estimating for our new project. The basic cost method, smoothing and extrapolation, is a combination of earlier methods plus some new ideas for the new project. We will call the new project case history *B*.

A major development cycle difference was that case *A* had a baseline design specification deliverable under contract, which accounted for about 12 percent of the total cost as shown by "analysis," and also had the implementation concept and test plan deliverable under contract, which accounted for another 18 percent as shown by "design" (see Fig. 3). Even though the software development cycle was different, we had exact data on the manpower assigned to the balance of the milestone events, which are similar. We now are in a position to extrapolate from case *A* to case *B*. Furthermore, since case *A* had been produced on time and under cost, we were in a strong position for proving demonstrated cost performance on a previous project.

A major functional difference was that case *B* represented a unified communication, command, and control software system, whereas case *A* did not have the command function. Immediately, greater attention was given to the analysis and design of that new technology area and its interfaces. Even here, important similarities were identified in the history file because in case *A* the "similar" command function was the responsibility of another associate contractor. We concentrated on improving the information transfer by having full control of the interface, and by approaching the solution from an entirely new direction. We decided the technical risk was somewhat higher for case *B* because our cost data base was incomplete, and because the command algorithm design, coding, and testing were innovative. We developed the overall system block diagram, and allocated statement of work tasks to software functions. We were in a position for a first rough cut target cost.

The whole idea in this case was a) to price the total software system from the top down using the experience of the system concept group, b) to price each work

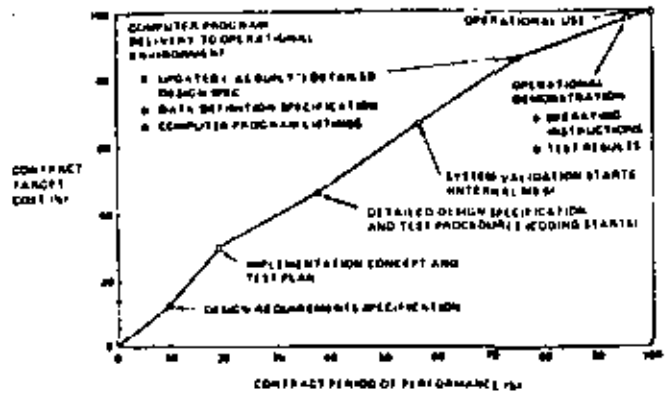


Fig. 2. Typical software development cost experience (case history *A*).

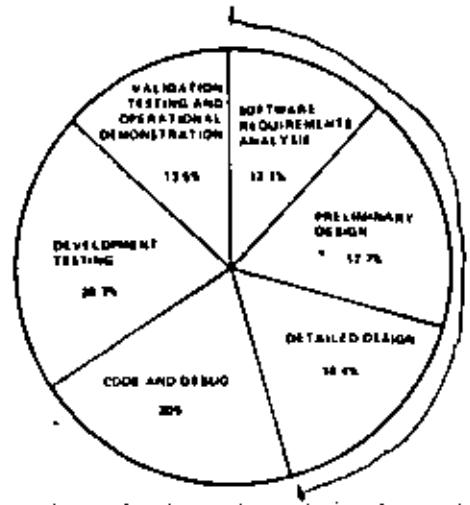


Fig. 3. Cost distribution by activity during full period of performance (case history *A*). All activities include documentation and travel costs.

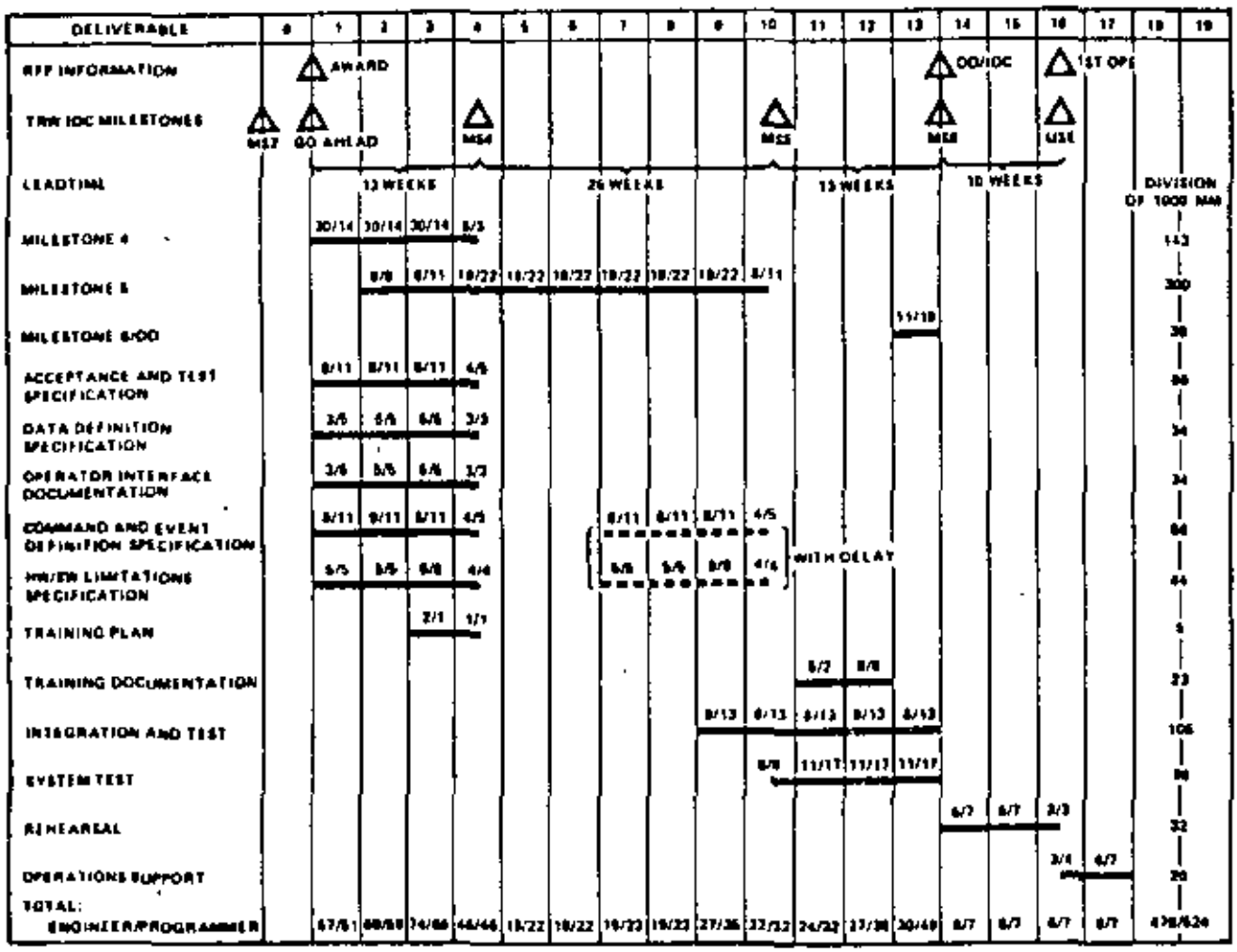
package from the bottom up using the experience of each of the five work package managers, and (c) to conduct a mock negotiation between the system concept group and each work package group taken one at a time. Differences were derived, costs were traced back to source authority requirements in the customer's statement of work, and adjustments made between the cost, probable risk, and scope of work.

In both methods the basic unit of estimation was the "man-month" for a given task or element. The distribution of the work (man-months per month) was driven by two critical variables: initial operational capability (IOC's) and full operational capability (FOC's). Software configurations were negated by the RFP, and a new incentive for schedule time was required by the RFP. Each will be briefly addressed by assessing its impact on the cost method.

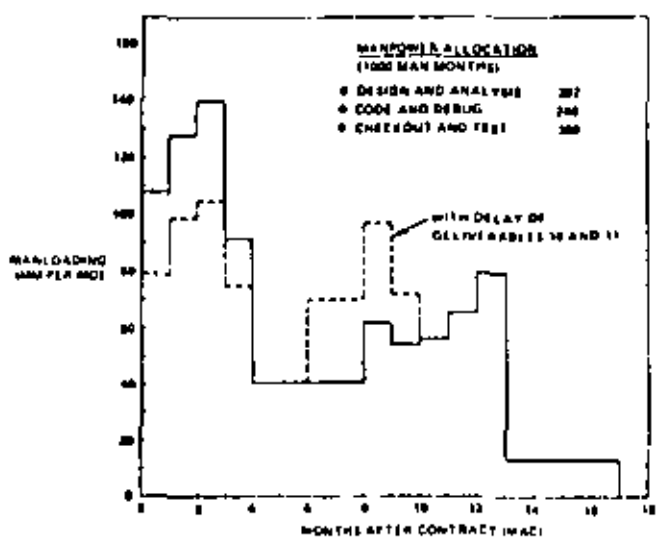
The distribution of manpower spread over the proposed period of performance was initially determined by the system concept group using only a bar chart. That idea was first to assume that the total development manpower resource was 1000 man-months, and then to determine the slope of the manpower distribution over time that maximized the fee incentive with least risk, such that the area under the curve remained constant (namely, 1000 man-months). This curve would serve to generate the master schedule for IOC's and FOC's events. The amplitude of the curve would be the primary focus in the separate negotiations with the five work package managers.

The amplitude represented the number of man-months per month for any given time slice. The sum of the manpower estimates of each of the five work packages at any time point determined that month's manpower requirements, and the sum of those over all time determined the total IOC'/FOC' manpower requirements, hence total cost. Of the five initial estimates, four were judged too high in the internal first cut negotiations, and one was judged too low.

The shape of the first and second estimate by the system concept group is given in Fig. 4. It is a poor distribution with steep manning rates, high peaks, and sharp declines. Considerable smoothing was required to achieve a manpower increasing and decreasing shape, at comfortable manning rates and reasonable levels and still meet the original goal we set with respect to incentive fee. The shape was drastically altered by introducing the concept of "staggered milestone events." That is, instead



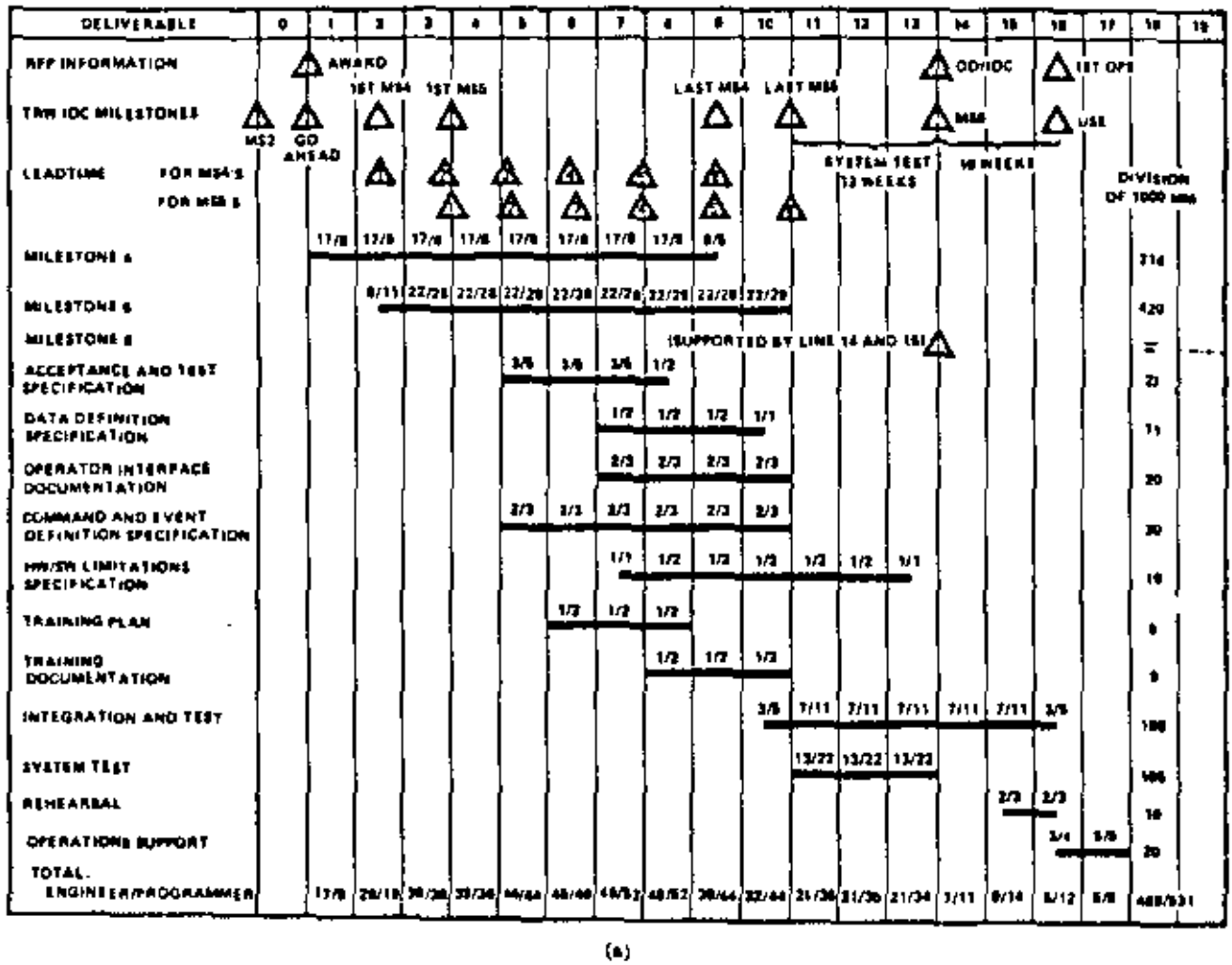
(a)



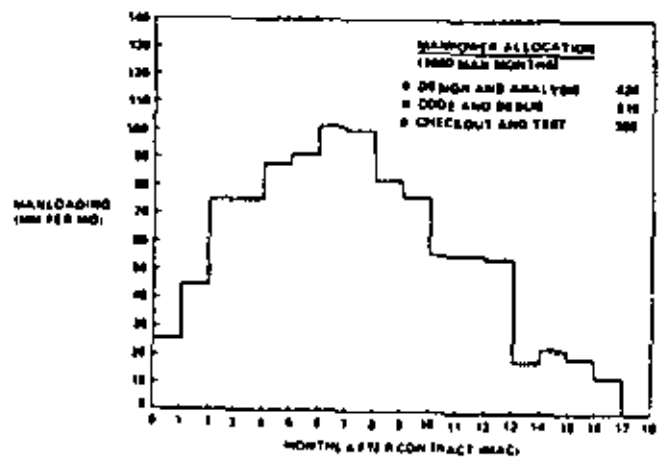
(b)

Fig. 4. (a) Manpower estimate with single-release milestone approach, showing key deliverables and labor allocation of 1000 man-months. (b) Manpower estimate with single-release milestone approach, showing uneven labor spread of 1000 man-months.

of one event 40 and one event 50 steps as previously defined), we planned for a total of six each. We could do this because of modular construction of the software to the end that each module was ready when needed, and system level testing by the independent test and validation group could commence soon enough to meet all incentives, as best by plan. The months shape of the third estimate, and the one that influenced the most



(a)



(b)

Fig. 5. (a) Manpower estimate with staggered-release milestone approach, showing incremental delivery of detailed design specifications. (b) Manpower estimate with staggered-release milestone approach, showing "smoothed" labor spread of 1000 man-months.

placement of the incentive events, is shown in Fig. 5. Both sets of curves show only IOC activities for simplicity; as IOC man loading phased down, FOC phased up, in general.

A simplified description of the fee structure that influenced the shape of Fig. 5 is that cost, schedule, and performance were incentivized (from a maximum of 15 percent of final target cost to a minimum of 0) based on certain predetermined rules that would be applied at three discrete milestones: Step 4), the detailed design specification, Step 5), the updated detailed design specification with computer program listing, and Step 7), opera-

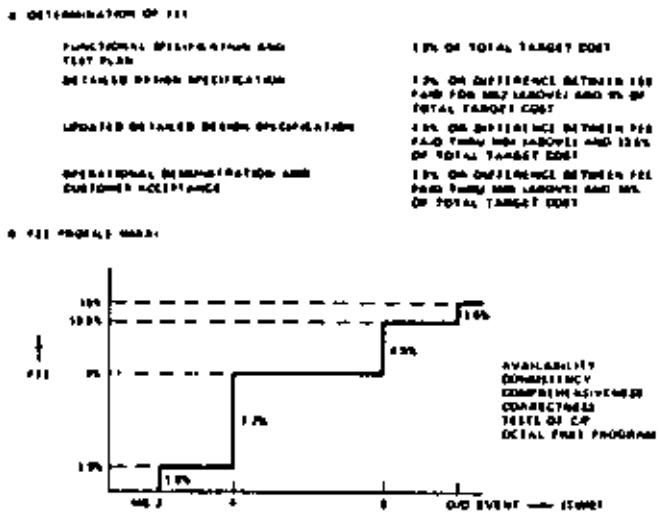


Fig. 6. Incentive fee structure (case history *B*).

tional demonstration (OD). The plan called for three IOC and three FOC dates.

The effect of a two-step procurement, such as an IOC followed by a later FOC, is as follows [7]. The IOC milestone events and OD constitute the only way that any fee can be earned, in discrete steps as shown in Fig. 6. The earned fee percentages apply to the target cost for the total IOC/FOC task. The incentive fee structure provides for fee penalties for failure to meet the contract dates for satisfactory demonstration of milestone steps 4), 5), and the OD. These penalties are assessed as shown in Fig. 7, on a linearly graduated basis in units of whole days late, where "days late" mean calendar days from the date specified in the contract. The penalties apply separately to both IOC and FOC. That is, the maximum penalty for lateness in demonstrating the OD of the IOC is 15 percent; it is also 15 percent for failure to demonstrate the OD of the FOC. This insures priority to the IOC, but also insures responsible attention to the FOC.

To be effective, the basic incentive structure must be simple, even though it is necessary in the contract to address the major contingencies and allowable options. If the basic incentive structure is not simple it will not readily be grasped by the many people at all levels of the contractor's plant whose work affects the chance of success. If they do not understand it, they will not do anything differently because of the incentive structure. If they do not, the incentive contract will have failed to achieve its fundamental purpose. the people who work on all aspects of the entire undertaking must be conscious of the incentive and must do their work with more care and quality because of it.

The entire incentive approach presumes that the contractor will take specific internal implementing action. It should include some tangible internal management actions that place an additional incentive on the work quality. A clear explanation of the essential features of the incentive structure should be given to all who work in

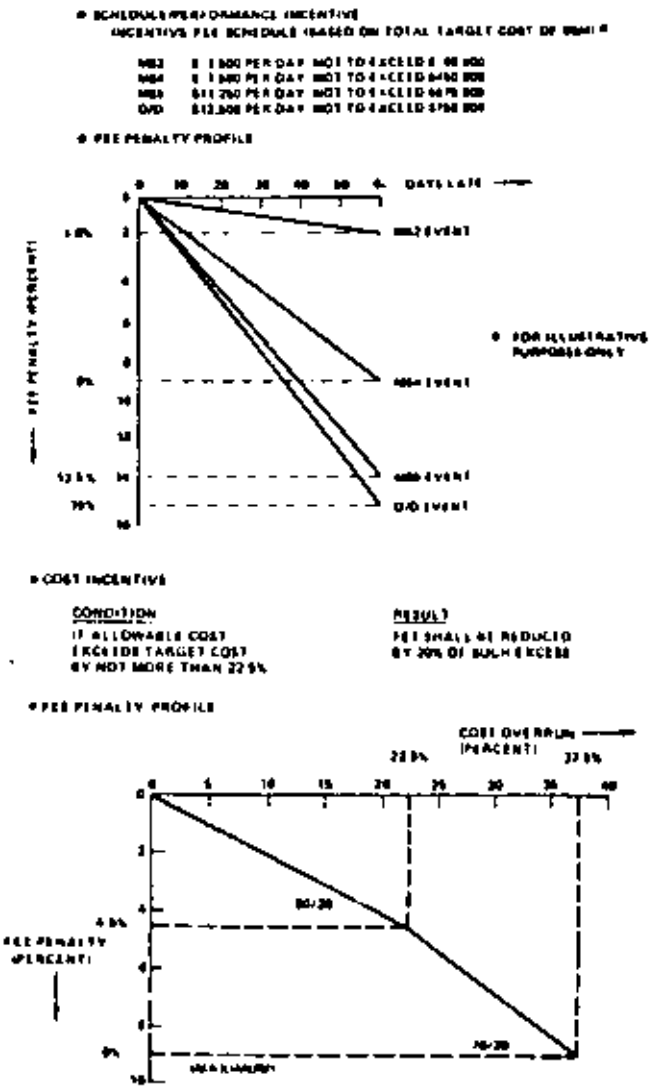


Fig 7. Penalty assessment structure (case history *B*)

any manner on the software, keyed to the contribution each one can make to affect the fee that can be realized by his company.

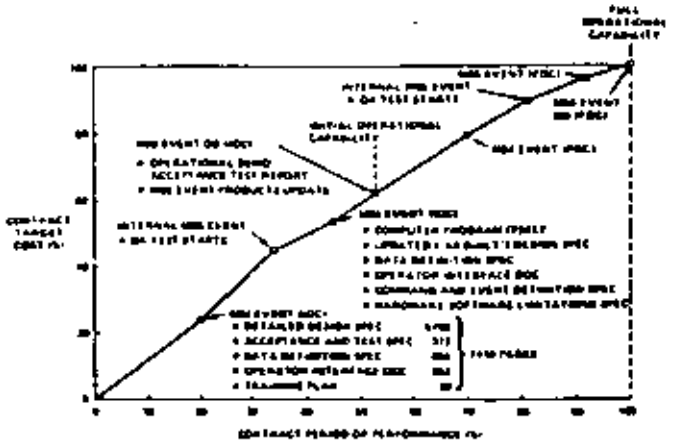


Fig. 8. Typical software development cost experience (case history B—multimodel development).

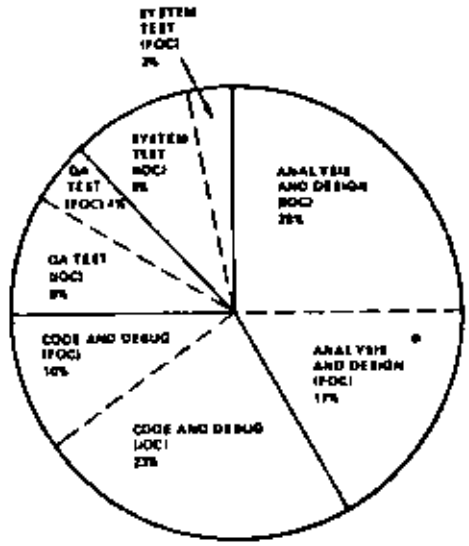


Fig. 9. Cost distribution by activity during full period of performance (case history B—multimodel development). All activities include documentation and travel costs. Distribution excludes O&M costs.

The normalized contract cost as a function of the normalized period of performance is shown in Fig. 8 for case history B. The normalized cost distribution by development activity over the software life cycle, for both IOC and FOC, is shown in Fig. 9. In estimating the cost of large software systems, we treat "operations and maintenance" (O&M) as a level of effort commencing immediately after selloff of the system in its operational environment. In the multimodel software development, O&M of the first model overlaps the purely development activities of the second model. To avoid any misinterpretation of the development costs, the O&M amounted to 15 percent of the total contract value. Comparison of case history A and B in size and cost (man-months), excluding O&M is given below.

	Number of Object Instructions	Man-Months	Rate
	(I)	(M)	(I/M)
Case History A	102 400	570	180
Case History B			
IOC	160 000[a]	945	170
FOC	215 000[a]	630[b]	340

[a] IOC activities included design and production of a prototype software package of an additional 20 000-object instructions. A slightly modified version was made during FOC.

[b] Learning curve phenomenon has been verified by empirical data and controlled tests in other industries [8], but not systematically for the software industry.

A Software Cost Estimation Algorithm

TRW has developed a cost estimation algorithm based on the assumption that costs vary proportionally with the

number of instructions. For each identified routine, the procedure combines an estimate of the number of object instructions, category, relative degree of difficulty, and historic data in dollars per instruction from the cost data base to give a trial estimate of the total cost. The design group estimates the first three software parameters. The system concept group provides the appropriate cost data base used by all designers, as well as the allocation of resources to each phase of the software development cycle, the schedule for each milestone event, and the labor mix. The procedure then spreads the total cost over the period of performance according to these input parameters, gives the man-months per month by labor category, the cost associated with each development activity, and the computer usage by month for checkout and test activities. The output may be considered a "trial estimate" in the sense that the proposal team must be satisfied with the resulting estimate when tested against any conventional method of cost estimating.

The first step in the method is to categorize the software routines that are being considered in the preliminary design. The software categories have been selected based on experience, and are those functionally different kinds of software entities for which a significant cost per instruction (CPI) is expected. Categories that have stood the test of usage in several proposal and preliminary design activities are: a) control routine, which controls execution flow and is routine-critical, *C*; b) input/output routine, which transfers data into or out of the computer, *I*; c) pre- or postalgorithm processor, which manipulates data for subsequent processing or output, *P*; d) algorithm, which performs logical or mathematical operations, *A*; e) data management routine, which manages data transfer within the computer, *D*, and f) time-critical processor, which is a highly optimized machine dependent code, *T*.

The next step is size and complexity estimates by routine, or subprogram, by the designer. To balance cost and risk, the designer may decide to use software elements that are available to him from a software library and need only some degree of modification or adaptation. At the other extreme, a new technique may be required and be estimated as a high technological risk. To account for the degree of difficulty of a given kind of routine, the designer estimates a risk or complexity factor. This is the most crucial step in the estimating process, for it establishes the cost of the routine with all direct and indirect charges amortized against it. The other steps basically determine how the total cost will be spread over the development cycle.

Two consultants have different views of how software parameters should be estimated. In general, Brandon says a single individual should establish a complexity rating scale (*A,B,C,D,E,F*) and make a "standard estimate" for each job based on: a) complexity rating of each job, b) machine used; c) language used; and d) estimated number of instructions. Brandon uses equations fitted to historical data to get standards, and then measures performance against the standard [9].

Lecht believes the estimator should interview the member of the technical staff who will do the job, and negotiate personal agreement on effort. Historical cost data reinforce the estimator's judgment where similar jobs can be found. The estimate is based on: a) similarity with previous modules; b) person doing the job; c) machine used; d) language used; and e) estimated number of instructions. Lecht does not believe meaningful performance standards can be set for software [10].

The simplest technique for narrowing down the many subjective choices early in the estimation process is to ask is the routine new or old, and is it easy, medium, or hard? A complexity rating coefficient can be applied continuously from 1 to 20 as a multiplier, if preferred. The important consideration is allowing sufficient degree of freedom to accommodate a learning experience and individual differences in performance in the effort to obtain a realistic cost. For our present purposes of early preliminary design, we will allow the designer four choices to account for six levels of difficulty for each routine. They are:

	Easy	Medium	Hard
Old	OE	OM	OH
New	NE	NM	NH

The only parameter that changes as a function of degree of difficulty (OE through NH) is cost per instruction. Override control is available to the designer. Sample data sets will be given in the section on Software Cost Data Base.

The next step is to identify the various development and test phases for producing the software from the conceptual stage to delivery of the operational software to the ultimate user. For our present purposes, the seven steps given in the section entitled The Software Development Cycle will be adopted, but they could be any other steps tailored to the needs of the particular application. For each phase an estimate is required for the fraction of the total amount to be allocated to it. In the case of manpower allocation for distribution of 1000 man-months over the development cycle, it was functionally allocated as 424 man-months to design and analysis, 210 man-months to code and debug, and 366 man-months to check out and test (see Fig. 5). This distribution of 12 21 37 percent at this level of detail has been borne out in practice by several other researchers, as will be demonstrated shortly.

The fourth step is to define the activities in each development phase by means of an activity array and associated cost matrix. The activities as a function of development phase is a 25 × 7 matrix, that is, 25 activities are identified for each of 7 phases. In turn, subsets of the

activities may be summed to "super activity" levels, and the makeup may vary from phase to phase. A typical activity array and cost matrix will be presented in the section entitled Resource Allocation.

The final step in setting up the initial conditions for the cost estimation algorithm is to provide schedule data based on the customer's statement of work, or other management considerations. Schedule data are input as months from go-ahead for each of the milestone periods. Burden rates are input for projected overhead rates, general and administrative, and so forth. Labor mix is defined and unburdened bid rates for the labor grades desired for the phase are defined. Selective cost cuts are under management control by the override capability. Other direct charges are input, which for software is typically travel as a percentage of direct labor costs, such as 3 percent and documentation at 10 percent.

Computer usage data for a machine in the CDC 6500 class with time-shared central processor unit based on sampled data from a programming department by month have been [1]

Number of MTS	Total Number of Processor Units Used	Number of Processor Hours per Man-Month	Peripheral Hours per Man-Month
36	55.6	2.20	7.6
35	31.0	1.27	4.4
33	30.0	1.30	4.5
33	46.5	2.02	7.0

Allowing for slightly less power in a 370/155 computer, for instance, a figure of 3 h/month/programmer man-month would be reasonable. At an average productivity of 1 object instruction/h, this is equivalent to 156 instructions/man-month, or 1.2 min/instruction. The data in the third column are based on 1.42 processor units corresponding to 1h of computing time. A computer hours matrix in terms of usage rate per instruction by phase is given in the section entitled Resource Allocation based on the above data.

The summary output from the cost estimation algorithm, SPREAD, includes: a) cost per routine based on either historic burden rates or proposed burden rates; b) average cost per instruction, number of instructions, and category; c) total number of development computer hours per routine; d) simple graphic display of schedule and events; e) cost breakdown by development phase per routine in total dollars and percent; f) cost breakdown by activity, by routine, and summed over all routines in the software, such as management, review, documentation, specifications, design, coding, and testing, and finally, g) manloading and cost summary by segment showing for each month the labor breakdown for senior staff, staff, technical, clerical, also, computer hours by month, other direct charges, cost by month, and cumulative cost.

The outputs from the cost estimation algorithm are considered a "trial set" for the cost estimation group. The trial set is used in combination with all other sources of data to test that cost position against the project objectives. The approved trial set, which contains the best judgmental and quantitative measures of cost per software element and per activity, becomes the input to the official pricing computer run. Necessary translation of the data set to exactly match the customer's work breakdown structure (WBS) or other appropriate cost elements is made for the final pricing run. The official cost figures are produced by cost guidelines, approved rates, and procedures that have been established by the in-house pricing group and approved by the government auditor. The results of the software cost estimation algorithm are retained as cost backup data and for possible use in later cost justification. The official cost figures from the pricing computer run show manloading and costs collected and aggregated against the customer's WBS (see Fig. 10).

This cost estimation approach has the following advantages. a) The amount of data required to obtain a cost estimate is minimal. b) The software designer immediately sees the cost consequences of his preliminary design. c) To the extent that the routine is directly traceable to a requirement, the requirement is directly traceable to a total cost. d) Comparisons of alternate schedules on resource allocation can be made rapidly. e) A cost justification is produced that can stand the test of government audit.

SOFTWARE COST DATA BASE

Sensitivity coefficients or exchange ratios used by the cost estimation algorithm reside in the data base. The objective in cost estimating is to realize greater accuracy and precision while anticipating risk and its impact on profit. The greater the risk, for a given confidence level, the greater the allowance on the part of the estimator for this uncertainty. The more uncertain the estimate, the less competitive is the cost proposal. Further, an estimate of cost backed by relevant cost experience facts gives confidence to negotiator, management, and the customer. Cost justification is demanded by the customer, and is usually provided in the form of lower level back-up data.

Valid data based on proven experience are required for any cost estimation process. Producing good operational software is dependent on the many activities of good people properly directed and motivated. The process is difficult to measure and apply widely beyond the technology center for which the performance measures were derived. There is no universal model. Applying "average productivity rates" without knowledge of the development steps that were used in deriving the rates is wrong. Averages based on large samples are useful in comparison of a local variable against a global variable, if the estimator is reasonably certain he is comparing truly equiv-

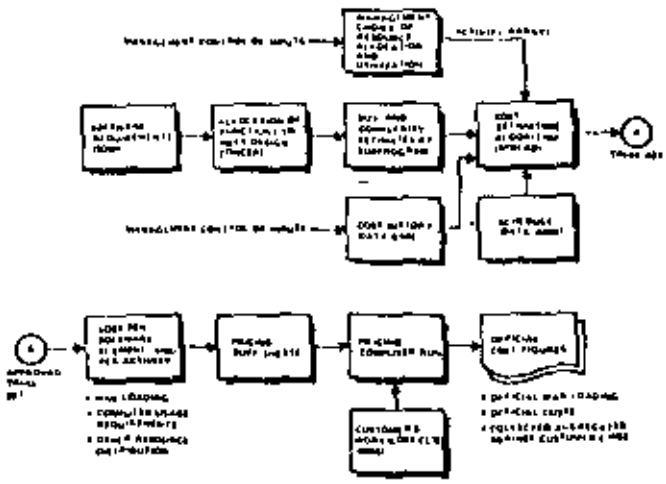


Fig. 10. Simplified costing sequence for TRW software, showing cost estimating algorithm data flow.

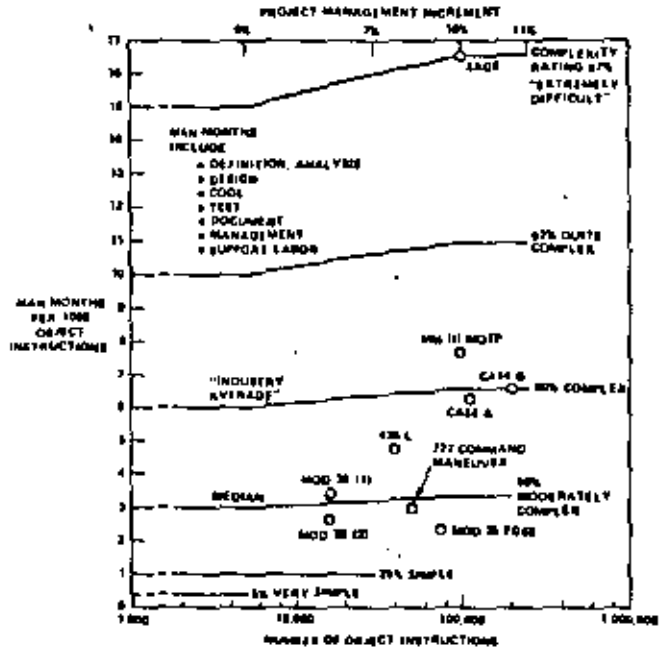


Fig. 11. Cost estimating graph, showing industry-wide averages.

alent measures. Metzelaar has studied the problem of industry-wide productivity rates in terms of program size and complexity, as shown in Fig. 11 [12].

The activities that productively occupy the analyst's time, along with the plant overhead and other direct charges, largely determine the price to the customer. It is in the company's best interest to agree on measurable activities that should be available in the software cost data base for cost justification and estimation of new work. Brief definitions of activities that have been identified in this regard are given below [2].

1) *Learning and Orientation* (L): Those actions necessary to gain understanding of the hardware and software to be applied in the project, and to learn the operating requirements, specifications, and constraints to be satisfied by the software package being produced.

2) *Analysis and Design* (A): All actions necessary to generate technical solutions, which are expected to satisfy requirements and specifications within the imposed constraints. It includes determining and evaluating technical approaches, determining and evaluating the effects of specific requirements and constraints on potential technical solutions, selecting the best solution, and describing that solution so it can be coded. This activity includes the rough documentation that is normally produced in generating a technical solution, but not the efforts of finalizing the documentation package.

3) *Coding* (C): The translation of the detailed steps of the technical solution into machine-readable language; the ordered list, in computer code or pseudocode, of the successive computer operations for solving a specific problem. It does not include actual input to the computer

(except for time-share inputs), compiling, nor testing the adequacy or validity of the instructions.

4) *Test and Checkout (T)*: All actions necessary to assure that the instructions coded by the programmer cause the machine to do what the programmer intended it to do. It covers preparing the test data, compiling the program, running the test data, reviewing the compiler and/or machine output, identifying errors, correcting the errors, and making changes to the program that improve its reliability and efficiency. It does not include verifying that the software product satisfies the requirements and specifications stated by the customer or sponsor.

5) *Verify/Quality (V)*: Those actions necessary to assure that the software product satisfies the requirements and specifications provided by the customer or sponsor. It includes preparation of qualifying test data, verification runs, evaluation of computer run results, and adjustments to the program to correct any deficiencies.

6) *System Integration and Test (I)*: All actions necessary to assure that the subsystem or module as programmed will interface with other subsystems or modules to provide a satisfactory system. It includes preparing data for evaluating the joint functioning of two or more subsystems or modules, making the evaluation runs, evaluating the joint operation, and making necessary modifications to the software in accordance with prevailing configuration control procedures.

7) *Documentation (D)*: Those actions that require technical editing, technical typing, art work, and reproduction of deliverable documents to the customer or sponsor.

An example of the cost per instruction for the significant categories of software previously defined versus degree of difficulty is given in Fig. 12. The cost figures, in principle, include all seven development steps described in the section on software development and test management. Activities that are defined immediately preceding are inherently included in the cost figures as are all other direct charges up to and including general and administrative costs.

In late 1971, TRW compiled in-house data on a wide variety of software development characteristics, which in turn were independently analyzed by Lulejian and Associates under contract to the U.S. Air Force [13], [14]. The variables of particular interest for our present purposes were analyzed by Lulejian and Associates for a package of SS routines developed by TRW. A brief interpretation of four of the analyses that bear on the cost of developing software is presented next.

Consider Fig. 13. This is a plot of cost versus number of instructions for the SS routines. Two conclusions can be drawn, either by direct observation or by regression analysis. a) There is a general trend in the data: larger routines cost more. b) A linear fit to the data is not very good. If one attempts to predict costs from routine size

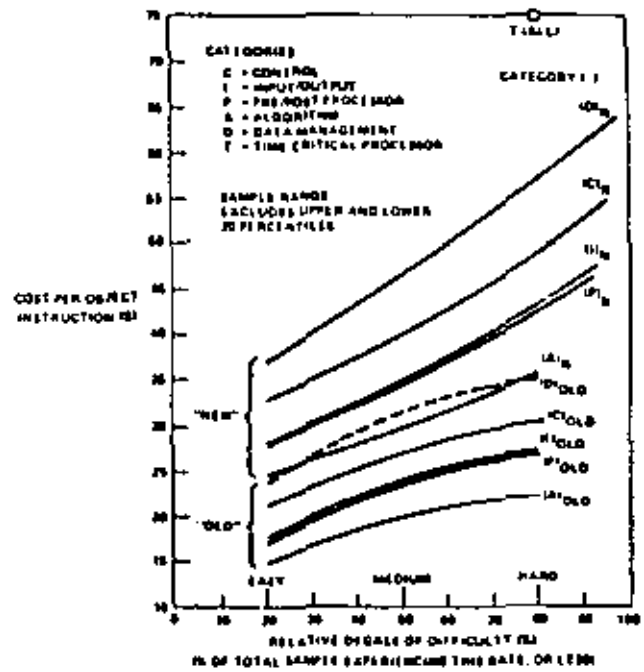


Fig. 12. Cost per object instruction versus relative degree of difficulty.

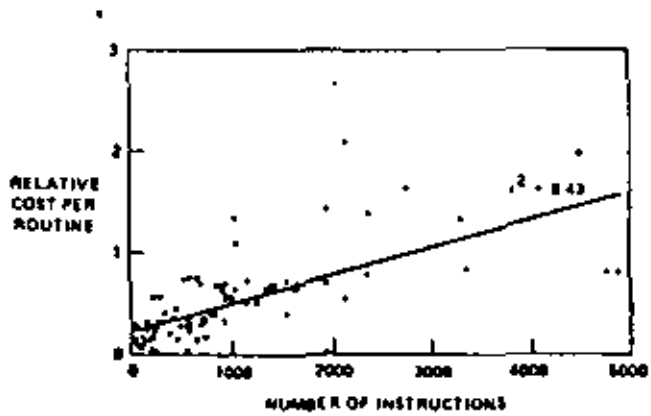


Fig. 13. Relative cost per routine versus number of instructions.

alone, one can expect errors of about 50 percent in the predicted cost. The prediction is better than nothing, but not much better. Fig. 14 shows the same data plotted another way—cost per 1000 instructions versus number of instructions. If the cost of the routine can be estimated by a fixed cost per 1000 instructions, one would expect this data to show constant cost. It appears to be constant, with a large error, especially for small routines. Again, one concludes that cost can be predicted from number of instructions provided that one is willing to accept fairly large errors.

Cost depends on a number of other factors, and one might hope to get better cost estimates by including difficulty, programmer experience, etc. A considerable number of fits were tried. The answers, unfortunately, were negative. Including other variables did not result in an equation that gives any significantly better fit to the data. Fig. 15 shows an example of one of the variables—

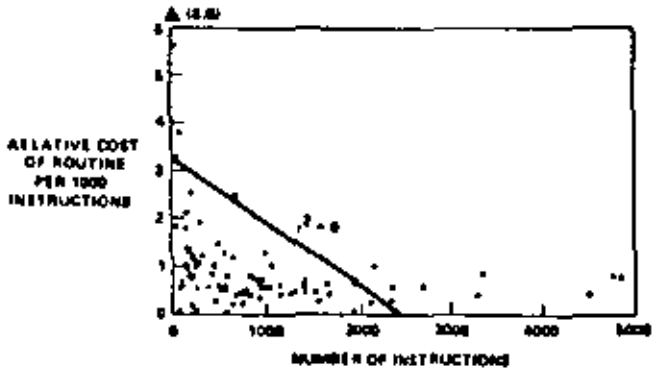


Fig. 14. Routine unit cost versus number of instructions.

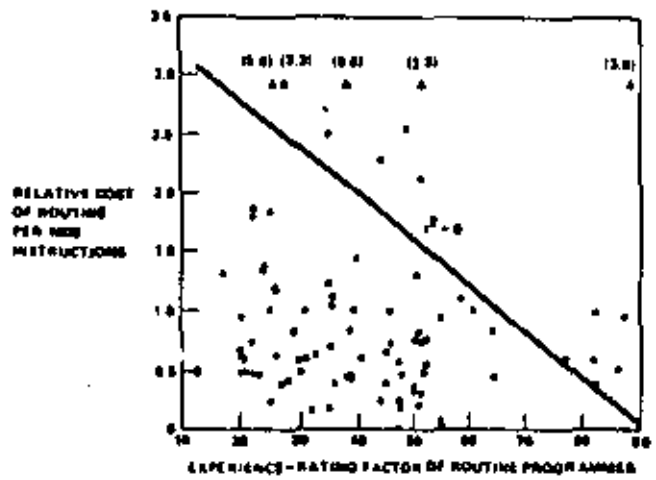


Fig. 15. Routine unit cost versus experience rating factor of programmer.

TABLE I
MULTIVARIATE REGRESSION SUMMARY

EQUATION	GOODNESS OF FIT
COST = 0.28 + 0.00029 (NUMBER OF INSTRUCTIONS)	r^2 = 0.43
COST = 0.021 + 0.00018 (NUMBER OF INSTRUCTIONS) + 0.18 (DIFFICULTY)	r^2 = 0.48
COST = 0.005 + 0.00029 (NUMBER OF INSTRUCTIONS) - 0.00011 (NUMBER OF LOGICAL INSTRUCTIONS) - 0.00027 (NUMBER OF INPUT/OUTPUT INSTRUCTIONS) + 0.0096 (NUMBER OF INTERFACES) + 0.136 (DIFFICULTY) + 0.0034 (PERCENTILE RATING) + 0.0060 (YEARS OF EXPERIENCE) - 0.0258 (YEARS OF SCP EXPERIENCE) - 0.163 (SUPERVISOR'S RATING)	r^2 = 0.60

TABLE II
COMPUTER PROGRAM DEVELOPMENT BREAKDOWN

	ANALYSIS AND DESIGN	CODING AND AUDITING	CHECKOUT AND TEST
SAGE	35%	14%	47%
NTDS	30	20	50
GEMINI	36	17	47
SATURN V	32	24	44

experience of programmer. A glance at the curve suggests that a knowledge of this variable does not help one predict cost. The regression analysis results support this suggestion. Table I shows the results of a multivariate linear regression analysis. Adding the remaining variables gives a better fit, but not much better.

The conclusion is that there are no simple universal rules for costing software accurately. It is necessary to understand the nature of the individual program and the individual routines within the program.

RESOURCE ALLOCATION

Various researchers have separately discovered by empirical methods a rule of thumb, namely, that analysis and design account for 40 percent, coding and debugging account for 20 percent, and checkout and test account for 40 percent of the total resource (cost) spent for software development, the 40-20-40 rule. If the cost of "modeling" of physical systems is to be borne by the software development group, as contrasted to modeling analysis from raw data by technology centers within the company (such as thermal, attitude control and pointing, propulsion, electrical power, and so forth) an extra 30 percent typically would be required in addition to the software development by itself. In other words, the deriving of equations of motion or physical behavior of other physical systems from raw data is not included in the software development resource allocation. Similarly, the cost of computing time is not covered in the resource allocation. Experience has consistently shown that computer costs add an additional 20 to 25 percent of the total cost of the project; that is, a $5 million/year software development contract will cost the government an additional $1 million for GFE computing time to the developer. Boehm discusses his views on resource allocation in [15]; his findings are presented in Table II. The findings of an in-house survey by Metzelaar are presented in Table III. Independently derived resource allocation for the 1000 man-month costing example was given in Figs. 4 and 5.

The basic resource allocation data that are required for application of the cost estimation algorithm are shown in explicit form for purposes of illustration only in this paragraph. Table IV shows a simplified version of how "complexity" might be handled in a preliminary design. For example, if we are designing one routine in a large system that we estimate to have 1000 executable, or object, instructions (not Fortran IV source instructions), and that has as its primary function setting up the input

TABLE III
AVERAGE PERCENTAGE OF ANALYST'S TIME BY ACTIVITY

ACTIVITY	INFORMATICS	RAYTHEON	TRW[a]	AVG
ANALYSIS	20%	20%	20%	20%
DESIGN	16	20	20	18.7
CODE	18	26	24	21.7
TEST	32	26	26	28.3
DOCUMENT	14	8	6	11.3
	100%	100%	100%	100%
MIX SCIENTIFIC	(80)	(0)	(100)	(53.3)
BUSINESS	(40)	(100)	(0)	(46.7)

[a] Substantial variation from project to project.

TABLE IV
COST PER INSTRUCTION AS A FUNCTION OF DEGREE OF DIFFICULTY (EXAMPLE ONLY)

DEGREE OF DIFFICULTY	CATEGORY					
	C	I	P	A	D	T
OE	$21.00	18.00	17.00	15.00	24.00	75.00
OM	27.00	24.00	23.00	20.00	31.00	75.00
OH	30.00	27.00	26.00	22.00	30.00	75.00
NE	33.00	28.00	28.00	24.00	37.00	75.00
NM	40.00	35.00	34.00	30.00	46.00	75.00
NH	48.00	42.00	42.00	36.00	54.00	75.00

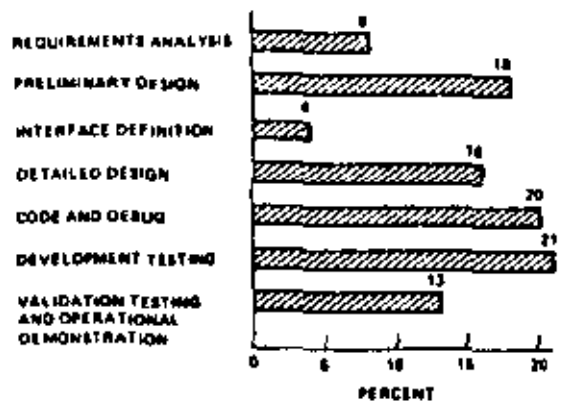


Fig. 16. Typical allocation of resources in custom software development and test.

to the main algorithm, we first categorize it as a "preprocessor," or category P. It is new and seems to be of medium difficulty (compared to others we have seen), therefore it is further categorized as new-medium, or NM. We enter on the worksheet for this routine its name, say PROG, its category P/NM, and a number of executable instructions equal to 1000. The consequence of this action is that PROG, at $34.00/instruction, will cost the customer (i.e., directly timeable to PROG) $34 000. This covers the cost of all activities, including analysis, design, documenting, coding, checkout, and test. These activities are spread into the development phases according to the typical allocation of resources shown in Fig. 16. Summing the first four phases prior to code and debug shows we allocate 46 percent of our total dollar there, coding takes 20 percent, and the two major phases after coding take the remaining 34 percent. This distribution has been tested against at least two other computer program development cycles, provides a good fit to present design techniques, and will be adopted as the nominal.

The resource distribution in the seven phases [steps 1)–7) previously defined correspond to the seven phases A–G] will vary as a function of "category," and may be interpreted as a perturbation about the nominal. In all cases H, the O&M phase, will be treated differently since it is not characterized by the same objectives or activities as the development phases. The percent variation of total resource allocation as a function of category is given in Table V.

After classifying the software package (at the routine

TABLE V
RESOURCE ALLOCATION AS A FUNCTION OF CATEGORY FOR EACH PHASE (EXAMPLE ONLY)

CATEGORY	PHASE						
	A	B	C	D	E	F	G
C	8	20	2	28	14	24	12
I	8	18	4	16	21	21	12
P	8	18	3	14	23	21	12
A	8	18	1	21	22	17	12
D	8	22	4	14	16	24	12
T	8	11	12	10	24	21	14

level) by category and degree of difficulty, and defining the development phases for producing the software, we define the "activities" to be performed during each development phase. The software statement of work is the source authority for defining activities against which costs will be collected at an appropriate level in the work breakdown structure. The 25 activities (or tasks) that comprise the 7 development phases, which in turn lead to preparation of the deliverables for our hypothetical project, are shown in Table VI.

The assignment of resources to each of the 25 activities for each of the 7 development phases is shown in Table VII. Historical data from the cost data base combined with engineering judgment gained from work on previous similar projects are used in making the assignment.

The remaining step is to assign a properly time-phased computer resource to be used for the development and test of the software. The rationale for such a computer hours matrix was given earlier. The numerical results are given in Table VIII, together with an explanatory note on interpreting the data array.

An example in the application of the entire process would be useful. Within the space of this paper it is not practical; however, preliminary design of a large command and control software system was carried out in late 1971 that provides two useful insights for our present purposes. The first is that the command and control portion of the software system consisted of two major subsystems, the first containing 69 subprograms which were considered quasireal-time, and the second containing 29 subprograms which were considered real-time. In fact, some of the former contained real-time subprograms and the latter some nonreal-time subprograms. Thus, if a casual observer were to characterize the software as solely real-time (or nonreal-time), in hopes of deriving useful cost data base parameters, he would be certain to fail in precise interpretation of the performance data. The resource allocation in terms of burdened man-months and computer hours for the effort is given in Table IX at the module level, where routines make up modules, and modules make up the system. If this system had been carried out in a complete development cycle, a new data set would be entered into the cost data base and variances between actual and predicted resources would be available to aid in the next estimating effort.

A second insight that seems useful is the cost estimating relationships that emerge when the software elements are separated into meaningful groupings of, in this case, real-time, real-time plus its supporting environment (pre- and postprocessors programmed in higher order language, for example), and command and control elements that are not time-critical (see Fig. 17). The data are now highly correlated, and the differences in productivity rates between the groupings would be expected to range nearly 3 to 1. The observed differences should not be attributed to lack of correlation between dependent variables that are seen to be members of different sets.

TOPICS FOR FURTHER WORK

The topics that were originally identified for discussion will be summarized, together with key issues that influence cost estimating of large software systems and that need further work and study.

1) What is the typical code production rate per programmer man-month? A working standard may be inferred from the data given of 1 object instruction/man-hour, which is equivalent to 156 instructions/man-month, or 1870 instructions/man-year, or 6.4 man-months/1000 object instructions for nontime-critical software.

If a burdened man-month varies between $4300 and $4900 in 1972 dollars for a typical programming department's labor mix, on the average this working standard translates to $27.50–$31.50 as the cost per instruction. Both case history *A* and *B* correlate strongly with these standards. They are characterized as large software jobs whose development is firmly structured and controlled. But what about the R&D work, not well structured and controlled, and which therefore does not lend itself to estimation by extrapolation from a historical cost data base?

The underlying question is whether we have identified all the right parameters to capture and quantify in our cost data base. Do we understand the causal relationships, and can they be isolated from their effects or symptoms? For example, is the SAGE data point where it is in Fig. 11 because it is "extremely complex" or because the requirements were poorly defined?

2) How does code production rate vary with problem complexity? With more sophisticated physical systems being brought under software command and control, a

TABLE VI
ACTIVITIES AS A FUNCTION OF SOFTWARE DEVELOPMENT PHASE

ACTIVITY \ DEVELOPMENT PHASE		PHASE A: PERFORMANCE AND DESIGN REQUIREMENTS	PHASE B: IMPLEMENTATION CONCEPT AND TEST PLAN	PHASE C: INTERFACE AND DATA REQUIREMENTS SPECIFICATIONS	PHASE D: DETAILED DESIGN SPECIFICATION	PHASE E: CODING AND AUDITING	PHASE F: SYSTEM TESTING	PHASE G: CERTIFICATION AND ACCEPTANCE	PHASE H: OPERATIONS AND MAINTENANCE
MANAGEMENT	1	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT
	2	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL
REVIEWS	3		PRELIMINARY DESIGN REVIEW (PDR)	INTERFACE DESIGN REVIEW (IDR)	CRITICAL DESIGN REVIEW (CDR)	SYSTEM TEST READINESS REVIEW (STR)		ACCEPTANCE TEST REVIEW (ATR)	OPERATIONAL CAPABILITY
DOCUMENTS	4	DOCUMENT AND EDIT	DOCUMENT AND EDIT	DOCUMENT AND EDIT	DOCUMENT AND EDIT	DOCUMENT AND EDIT		DOCUMENT AND EDIT	DOCUMENT AND EDIT
	5	REPRODUCTION	REPRODUCTION	REPRODUCTION	REPRODUCTION	REPRODUCTION		REPRODUCTION	REPRODUCTION
REQUIREMENTS AND SPECIFICATIONS	6	REQUIREMENTS DEFINITION	PDR MATERIAL	EVENT GENERATION INTERFACE	PRODUCT CONFIG DETAILED TECH DESCRIPTION (PART II) (WITHOUT LISTINGS)	TECHNICAL DESCRIPTION UPDATE	PRODUCT CONFIG DETAILED TECH DESCRIPT UPDATE (PART II) (WITH LISTINGS)	REQUIREMENTS CERTIFICATION	SOFTWARE PROBLEM REPORTS (SPR)
	7	REQUIREMENTS ALLOCATION	PERFORMANCE AND DESIGN REQUIREMENTS (PART II)	COMMAND DEFINITION I/F		TRAINING DOCUMENTATION		FINAL DOCUMENTATION UPDATE (PART II)	
	8			TELEMETRY DEFINITION I/F			INTERFACE SPECIFICATIONS UPDATE		
	9			OPERATIONAL ENVIRONMENT I/F					
DESIGN	10	TRADE STUDIES	TRADE STUDIES	TRADE STUDIES	TRADE STUDIES				
	11	INTERFACE REQUIREMENTS	FUNCTIONAL DEFINITION	DATA DEFINITIONS	ALGORITHM DESIGN	ALGORITHM UPDATE	PROGRAM UPDATE		
	12	HUMAN INTERACTION	STORAGE AND TIMING ALLOCATION		PROGRAM DESIGN				
	13	STANDARDS AND CONVENTIONS	DATA BASE DEFINITION	DATA BASE DESIGN		DATA BASE UPDATE	DATA BASE UPDATE		
	14		SOFTWARE OVERVIEW (PRELIMINARY)		SOFTWARE OVERVIEW UPDATE				
CODING	15				PROTOTYPE CODING	OPERATIONAL CODING			
TESTING, CONFIGURATION CONTROL AND QRA	16		PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL
	17		DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL
	18	TEST REQUIREMENTS	TEST PLANS	INTERFACES	TEST PROCEDURES	DEVELOPMENT TESTING PLANNING	SYSTEM TEST PLANNING	ACCEPTANCE AND TEST PLANNING	INTEGRATION TESTING
	19					DEVELOPMENT TEST	SOFTWARE SYSTEM TEST	ACCEPTANCE DEMONSTRATION	SPR CLOSURE
	20						HARDWARE/ SOFTWARE SYSTEM TESTING		
	21	ACCEPTANCE TEST REQUIREMENTS	QUALITY AND RELIABILITY ASSURANCE PLANS	QA AND RA MONITORING	QA AND RA MONITORING	QA AND RA MONITORING	QA AND RA MONITORING	QA AND RA MONITORING	QA AND RA MONITORING
	22					TEST SUPPORT	TEST SUPPORT	TEST SUPPORT	TEST SUPPORT
	23								
OPERATIONS	24		OPERATIONAL CONCEPT	OPERATIONAL TIMELINE	OPERATIONAL CONCEPT UPDATE	USER'S MANUAL (PRELIMINARY)	OPERATIONAL TIMELINE UPDATE	USER'S MANUAL UPDATE	INTEGRATION SUPPORT
	25		TRAINING PLAN		TRAINING PLAN UPDATE	TRAINING	TRAINING	TRAINING AND REHEARSAL	TRAINING AND REHEARSAL

corresponding increase in technological risk and complexity causes an increase in a responsible cost estimate. Reliable estimates place this increase in cost right now from 3 to 5.5 times the average of our historical cost data base. The leading causes of these predicted increases need serious attention and study, and effective management methods to detect and deal with them. For example, we have already observed in two cases that the cost of producing real-time (or time-critical) software is three times more costly than nonreal-time software (see Figs. 12 and 17). Should the development dollar be spent on highly optimized machine-dependent code by the applications programmer or on more efficient compilers and assemblers?

Cost tradeoffs are needed between the crucial param-

TABLE VII
COST MATRIX DATA SHOWING ALLOCATION OF RESOURCES AS A FUNCTION OF ACTIVITY BY PHASE (CATEGORY *F*)

ACTIVITY	PHASE A (8)	B (110)	C (3)	D (14)	E (23)	F (21)	G (12)	H (10)
1	10	6	6	6	7	5	10	3
2	6	3	3	3	3	3	3	5
3		5	4	6		3	8	5
4	13	5	5	6	5	5	4	2
5	5	2	2	3	3	2	2	1
6	22	8	7	12	3	7	6	6
7	10	5	7		3		6	
8			7			5		
9			5					
10	17	10	10	6				
11	2	10	10	6	7	5		
12	2	5		13				
13	4	7	10		3	5		
14		4		3				
15				5	25			
16		4	4	5	4	4	10	10
17		3	6	6	6	5	8	10
18	5	6	3	6	5	2	5	10
19					10	15	14	5
20						10		
21	2	4	5	5	7	6	6	3
22					5	6	5	4
23								
24		4	3	3	3	5	3	25
25		2		1	2	3	5	10

TABLE VIII
COMPUTER HOURS MATRIX SHOWING COMPUTER USAGE ALLOCATION AS A FUNCTION OF PHASE

PHASE	CATEGORY C	I	P	A	D	T
A	0	0	0	0	0	0
B	0	0	0	0	0	0
C	0	0	0.0017	0.0007	0.0007	0.01
D	0.0017	0.0017	0.0017	0.0017	0.0017	0.01
E	0.007	0.005	0.008	0.007	0.007	0.04
F	0.008	0.005	0.01	0.007	0.007	0.04
G	0.005	0.004	0.005	0.004	0.004	0.005
H	0	0	0	0	0	0

TABLE IX
RESOURCE ALLOCATION FOR LARGE COMMAND AND CONTROL SOFTWARE SYSTEM (PRELIMINARY DESIGN)

MODULE	OBJECT INSTRUCTIONS	MAN MONTHS	COMPUTER UNITS
CC (17)	21 700	152	434
CC (15)*	15 630	413	994
CC (14)	20 300	217	609
CC (3)	13 200	182	248
CC (15)*	24 300	254	487
RT (4)**	3 000	44	84
RT (5)**	5 900	68	118
RT (15)**	16 470	256	896
RT (4)***	2 550	41	765
ICS (8)	18 250	126	615
	180,750	1 602	5 534****

* MODULE CONTAINS 1 REAL TIME ROUTINE
** MODULES SUPPORT REAL TIME FUNCTION
*** MODULE ENTIRELY REAL TIME
**** EQUIVALENT COMPUTER HOURS 370/155

LEGEND
CC COMMAND AND CONTROL
RT REAL TIME
ICS INTERPRETIVE COMPUTER SIMULATOR
( ) NUMBER OF ROUTINES

eters that define the problem complexity and the cost benefits. A marginal utility theory is needed for the economics of software development. Parameters that are crucial in defining problem complexity include degree and time phasing of simulation, extent of prototype code, parallel development of different formulations of key algorithms, new computer hardware software, assembly language coding, multisite operations and interchangeability, growth requirements, critical timing and throughput, hardware–software interfaces, fault-tolerant computing, core occupancy constraints, reliability, and safety [16]–[18].

3) How does this rate vary as a function of computer reliability? Accessibility? Configuration? The President's Blue Ribbon Defense Panel reported in June 1970 in its staff report on automatic data processing, that because of the government's method of selection and procurement of computer systems, multiple computers in the same geographical area can result in costs that are as much as five times larger than would be necessary if a few large computers were used in a shared operating mode. The underlying problem here is that of data privacy and protection (security) for enabling the use of remote terminals by government agencies for both classified and unclassified work. An independent evaluation by E. C.

Example: The column sum of category *C*, control routine, is 217×10^{-4} h/instruction, or 1.4 min/instruction. The total resource required is then distributed over phases *D*–*G*, as shown above, e.g., 0.42 min/instruction for phase *E*, coding and auditing.

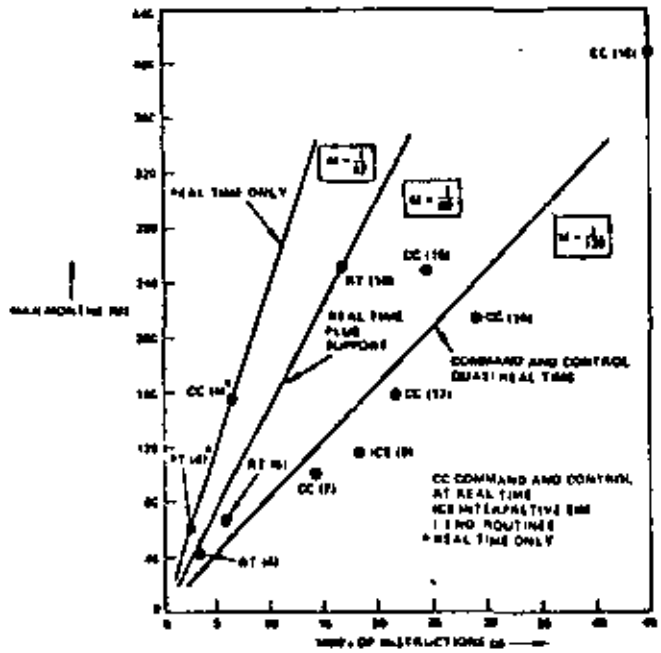


Fig. 17. Cost estimating relationships, showing wide variation within single command and control software system.

Nelson of TitW is that a dedicated computer installation costs three to five times as much to operate as if it were a time-shared installation with proper remote terminals. If private industry can enjoy these benefits of cost effectiveness by use of remote terminals (three to five times less expense), it would seem that some, if not a large part, of government installations could enjoy the same benefits if the problems of data privacy and protection were solved to everybody's satisfaction.

A management goal is to do the difficult parts of the software twice—once to produce prototype code early in the development cycle using perhaps 8-10 percent of the resources and an improved version with the technological risk eliminated for the operational version. One pitfall is that the risk is highest in procurements where a new computer and its operating system are being developed concurrently with advanced applications software, so just when computer time is needed the most, it is available the least. More planning to reduce the computer selection and procurement time is needed.

Furthermore, in some operational systems, the computer configuration available to the user is significantly different from the computer configuration available to the computer program associate contractor. Such a working environment requires an integrating contractor to maintain system-level configuration control. This may add 10 percent to the total cost and is viewed as an insurance policy. To eliminate such occurrences, standards for computer configurations are needed, and configuration control is needed by the associate contractor and, equally importantly, by the user.

f) What are the pieces of information required to make a realistic prediction of software development cost? We have identified the activities that occupy the analyst's time in a 25 × 7 matrix (Table VI) and elsewhere. However it is one thing to identify significant activities for allocation of resources, and it is another thing to hold the right set of individuals accountable for the expenditure of those resources for those things over the life cycle of the software development, which spans 12–24 months or longer. The purpose of a cost estimating system is to reduce the variance between estimates of what a software development should cost and what it actually does cost. The cost history data file is a key element in an improved cost estimating system. Further work is needed in capturing and maintaining historical estimates, costs, and quantitative facts to provide more effective data for estimating future costs and sizes. Further work is needed in maintaining cost control over the actual performance period considering the following.

a) Recording and displaying of both individual and aggregate programmer activities, such as previously defined by L,A,C,T,V,I, and D, by easy-to-use automated project control and prediction tools.

b) Fast computation of project cost- and time-to-complete predictions by current and look-ahead activity statistics, including uncompensated overtime, as is now feasible through automated cost control tools.

c) Provision for storing, retrieving, and tracing of project statement of work deliverables to their cost estimate and cost estimate to actual cost, by activity, over the useful life of the software through cost control tools.

d) How does code production rate vary as a function of

programmer quality? Quality requirements on the final product? The point to be made here is that we have seen convincing evidence of the validity of the 40-20-40 rule. For a $5 million project, $2 million will be spent on check-out and test to the extent the rule applies. A management objective is to check out every logic path in the software to deliver an error-free program. This is a high-quality final product insofar as being error-free is a measure of quality. Achieving this objective may add significantly to the development cost but reduce even more significantly the operations and maintenance cost. With improved test tools and flow measurement tools, considerable improvement in efficiency of more branches checked per computer hour becomes realizable. From the analysis given for one software system, the quality of the programmer in years experience and supervisor's rating seemed to make no difference in the cost in terms of the goodness of fit, the adjusted index of determination (r^2) given in Table I and Fig. 15. As R. J. Hatter says, some of the signs exhibit counter-intuitive behavior.

Technical performance standards and criteria are needed in the software development process. A wrong standard is better than no standard because at least it gives a sense of direction to the effort. A technical performance measurement system is needed for applying to the software development process, and which is understood and applied at all management and performer levels over the life cycle. Since software development is more process oriented than product oriented, such standards become measures of group productivity if not individual productivity. In some organizations it may be interpreted to be against company policy to measure and report on a productivity index of an individual member of the professional staff, such as code production rate, or error rates (as measured, for instance, by software problem reports). Producing quality custom software is a profit-motivated business, and more businesslike procedures are being applied as the product lines mature [19].

6) How does cost vary with completeness of problem formulation? Using the 40-20-40 rule again, but this time dealing with the "analysis and design" 40 percent (which for our example of Fig. 16 was 46 percent), we have evidence that analysis and design takes the lion's share of the software development dollar. A management goal that has proven effective in the past is to produce superior documentation which is reviewed with a knowledgeable technical staff in the customer's complex. Thorough and continuous involvement of the customer in the development process has been a reality of several large software developments. Nothing takes the place of competence and communication when it comes to understanding the customer's or sponsor's requirements. Translating total system requirements, which may not be well understood even by the customer, to the software system requirements is a crucial first step. In the requirements definition and preliminary design phase, there is typically a lack of data but lots of leverage (influence on the ultimate design outcome), in contrast to the operational phase where there are lots of data but little, if any, leverage. This is significant area for cost- and performance-effective improvement.

7) What is the role of "design-to-cost" in the development of large-scale software? The DOD joint logistics commanders have recently entered into an agreement concerning the acquisition and ownership of major weapon systems based on the concept of "design-to-cost" [20]. The term "design-to-cost" is defined as a process utilizing unit cost goals as thresholds for managers and as design parameters for engineers. Cost parameters (goals) are to be established that translate into "design to" requirements. In the past, the order of priorities for major weapon system acquisition and ownership has been: a) performance, b) availability, and c) cost. Today, under the design-to-cost doctrine, the order is: a) cost, b) performance, and c) availability. The applicability of the design-to-cost concept to hardware is now beginning to emerge and is being reduced to practice. The relationship of the design-to-cost concept to software is not now understood.

In the 1975–1980 era, this reordering of priorities for major weapon systems, of which software is generally an integral part, is expected to have a sizeable influence on both hardware and software life cycle acquisition and ownership, particularly in the sense that the unit cost requirement will become a driving parameter in the design of large-scale software. System development will be continuously evaluated against the unit cost requirements with the same rigor as now applied to technical requirements. Practical tradeoffs must be made between system capability, cost, and schedule. Traceability of estimates and costing factors, including those for economic escalation, will have to be maintained.

ACKNOWLEDGMENT

The author wishes to thank Dr. R. H. Pennington, Chief Scientist of System Development Corporation, who invited the author to prepare this basic paper and to serve on a panel at the IEEE International Convention in New York, N. Y., March 20–23, 1972. The other panel members were T. E. Climis of IBM, V. LaBolle of the Los Angeles Department of Water and Power, and Dr. R. E. Merwin of the Safeguard Systems Office, each one having a different approach to the problem of the cost of developing large-scale software [21]. The author also wishes to thank many close TRW colleagues for their help in getting the ideas and material together for this paper, particularly, D. J. Alley, C. A. Bosch, W. V. Buck, R. D. Kennedy, W. L. Hetrick, W. A. Krumreich, W. C. Lynam, P. N. Metzelaar, Dr. D. D. Morrison,

Dr. F. J. Mullin, Dr. E. C. Nelson, F. Putich, E. A. Rollin, and L. L. Wolfson. A special note of recognition due Dr. W. W. Royce, who was the prime originator of TRW's SPREAD cost estimation algorithm. The author must, in the final analysis, take full responsibility for this paper, including errors of fact, omission, or expressions of opinion.

The author wishes to thank R. J. Hatter of Lulejian Associates, Inc., for the release of Figs. 13–15 and Table I from the West Coast Study Group report, and Dr. B. W. Boehm of TRW, formerly of The Rand Corporation, for his resource allocation data, Table II [15], [22].

Guidance from the three referees and Dr. R. A. Short of the IEEE TRANSACTIONS ON COMPUTERS is appreciated. Their decision to depart from the traditional IEEE TRANSACTIONS ON COMPUTERS editorial policy and publish a software technical management paper is gratefully acknowledged by the author. Their aim is to provide information useful to the hardware computer systems designer in learning more about the software development process, in pointing to the tasks where the programmer expends his major effort and, it is hoped, in identifying problem areas to attack so future computer systems may ease the programmer's task.

REFERENCES

[1] J. M. Beveridge, *The Anatomy of a Win*, J. M. Beveridge and Assoc., Inc., Playa Del Rey, Calif., 1970.
[2] W. C. Lynam, P. N. Metzelaar, and D. H. Hibaman, "S&ISD cost estimating system final report," TRW Systems Group, Redondo Beach, Calif., Nov. 13, 1970.
[3] S. Stimler, *Real-Time Data Processing Systems: A Methodology for Design and Cost/Performance Analysis*. New York: McGraw-Hill, 1969.
[4] W. F. Sharpe, *The Economics of Computers*. New York: Columbia Univ. Press, 1969.
[5] G. F. Weinwurm *et al*, *On the Management of Computer Programming*. Princeton, N. J.: Auerbach, 1970.
[6] W. W. Royce and E. A. Rollin, "A software cost estimation Technique," TRW Systems Group, Redondo Beach, Calif., Oct. 27, 1970.
[7] J. L. Martin, Jr., "A specialized incentive contract structure for satellite projects," Space and Missile Systems Office, Los Angeles, Calif., Apr. 18, 1969.
[8] H. E. Boren and H. G. Campbell, "Learning curve tables," Rand Corp., Santa Monica, Calif., Rep. RM-6101-PR, Apr. 1970.
[9] R. H. Brandon, *Management Standards for Data Processing*. New York: Van Nostrand, 1963.
[10] C. P. Lecht, *The Management of Computer Programming Projects*. Amer. Management Assoc., New York, 1967.
[11] F. J. Mullin, "Computing time usage," TRW Systems Group," Redondo Beach, Calif., May 18, 1971.
[12] P. N. Metzelaar, "Cost estimation graph," TRW Systems Group, Redondo Beach, Calif., April 30, 1971.
[13] C. A. Bosch and B. W. Boehm, "Software development characteristics CCIP-85 study group," TRW Systems Group, Redondo Beach, Calif., Oct. 8, 1971.
[14] R. J. Hatter, "CCIP study regarding analysis of TRW software analysis data (Excerpt)," Lulejian and Assoc., Inc., Redondo Beach, Calif., Nov. 29, 1971.
[15] B. W. Boehm, "Some information processing implications of air force space missions: 1970–1980," Rand Corp., Santa Monica, Calif., Jan. 1970.
[16] H. Hecht, "Figure of merit for fault-tolerant space computers," *IEEE Trans. Comput. (Special Issue on Fault-Tolerant Computing)*, vol. C-22, pp. 246–251, Mar. 1973.
[17] G. J. Schick and R. W. Wolverton, "Assessment of software reliability," presented at the 11th Annu. Meeting German Operations Res. Soc., Hamburg, Germany, Sept. 6–8, 1972.
[18] J. R. Brown and H. N. Buchanan, "The quantitative measurement of software safety and reliability," TRW Systems Group, Redondo Beach, Calif., Site Defense Program Rep. SDP 1776, Aug. 24, 1973.
[19] E. R. Mangold, *Software Development and Configuration Management Manual*, Software Product Assurance, TRW Systems Group, Redondo Beach, Calif., Oct. 1973.
[20] H. A. Miley, Jr., J. C. Kidd, Jr., J. J. Catton, and S. C. Phillips, "Joint design-to-cost guide, a conceptual approach for major weapon system acquisition," presented at the Soc. American Value Engineers Conf. Practice of Design-to-Cost, Long Beach, Calif., Oct. 21, 1973.
[21] V. La Bolle, "Cost estimating for computer programing," in *Proc. IEEE 1972 Int. Conv. Exposition* (New York, N. Y., Mar. 20–23, 1972). Contains large bibliography of 99 annotated references, pp. 28–30.
[22] B. W. Boehm, D. W. Kusy, and N. R. Nielsen, "ECSS: A prospective design tool for integrated information systems," presented at the American Institute of Aeronautics and Astronautics, Integrated Information Systems Conf., Palo Alto, Calif., Feb. 17–19, 1971, Paper 71-228.

Ray W. Wolverton (S'49–A'50–M'52) was born in Hopkins, Mo., on October 20, 1924. He received the B.S.E.E. degree from Stanford University, Stanford, Calif., in 1950, and did graduate work in mathematics, controls, and computer system design at the University of Southern California, Los Angeles, and the University of California, Los Angeles, in 1950–1952 and 1952–1955, respectively; and in management theory at the Industrial College of the Armed Forces, Fort Lesley, U. McNair, Washington, D. C. in 1971–1972.

From 1950 to 1955 he was a member of the technical staff at Hughes Aircraft Company, working in the communication, navigation, and landing activity for the MX-1179 manned interceptor. He joined the Ramo-Wooldridge Guided Missile Research Division in 1955, where his initial interests were in closed-loop trajectory simulations and orbital operations. Since 1955 he has been with the TRW Systems Group, Redondo Beach, Calif., working in the fields of flight mechanics, real-time command and control operations, computer software development and test management, and project management. His present interests are in large-scale computer program management, cost modeling and forecasting, and reliability (error-free software). Currently, he is a Senior Staff Engineer with the Systems Design and Integration Operations, Systems Engineering and Integration Division, TRW Systems Group. He was a coauthor and editor of the Flight Performance Handbook for Orbital Operations, produced under contract to NASA (New York: Wiley, 1963).

He is a member of Sigma Xi, the Scientific Research Society of North America, Phi Kappa Phi, the American National Standards Institute, and various professional societies. He is listed in Who's Who in the West and is a recipient of both NASA's Apollo Achievement Award and the astronaut's Silver Snoopy Award.

Managers need an understanding of how application software behaves, what factors can be controlled and what factors are limited by the process itself.

ESTIMATING SOFTWARE COSTS

by Lawrence H. Putnam
and Ann Fitzsimmons

Few managers are able to predict the time and resources needed to develop large-scale software systems. Progress is often measured by the rate of expenditure of resources rather than by some count of accomplishments. Unrealistic estimates often result in last minute efforts to get code written quickly, resulting in cost overruns and poor quality software.

Software development can be brought under control. It requires an understanding of how application software behaves, what factors management can control and what factors are limited by the process itself.

The basis of effective management is the fact that the software development process exhibits a characteristic behavior, which can be exploited, so that the expensive results of unrealistic approaches can be avoided.

Traditionally, managers make two incorrect assumptions about software development: that people and time are interchangeable and that productivity levels are relatively constant for all software projects within the same organization.

The first assumption is that development effort is simply the product of people and time and that the time can be specified arbitrarily by management. Thus, the manning level is the development effort (in man-years) divided by the predetermined development time (see Fig. 1).

For example, suppose that the organization has to work under constraints. If the manpower available were limited to 25 people, the time would be determined as 100 man-years divided by 25 people, or four years. However, if the system had to be finished in two years, the relation would become: Manpower equals 100 man-years/2 years, which equals 50 people.

Managers arrive at the second assumption by taking some overall productivity figures from previous projects that they think are similar. However, they do not examine closely the precise characteristics of that similarity. An estimate of total source statements derived from the specifications is divided by the productivity figures to give a man-year estimate. For example, assume that we have to build a system of 100,000 source statements (SS = 100,000) and our productivity is 1,000 source statements per man-year, by analogy with a previous project. The development effort thus equals 100,000 SS/1,000, which equals 100 man-years.

Unfortunately, our experience shows that these relationships are too simple, except in the case of very small programs, such as those of less than 2,500 source statements, or that employ a few people for a few months.

For larger programs we now know that people and time are not interchangeable. Fred Brooks, manager of the IBM 360 operating system project, has described this phenomenon so graphically that a variant of it has become known as Brooks' law: "Adding people to a late project only makes it later." The reason is clear. As the number of people on a project increases arithmetically, the number of human interactions increases geometrically. More and more time must be spent on human communication and less and less on productive work. The only way to avoid this inevitability is to reduce the number of people who must interact by stretching out the time.

We also now know that productivity is not constant. Rather, it is a complex function of the effort, time and technology tools being applied to the development task. You can't improve productivity without changing these factors. It is not unusual for a group of programmers to achieve a productivity of, say, 5,000 source statements per man-year on a small, relatively simple business application while doing only 1,000 source state-

ments per man-year on a large real-time system.

With this kind of variation in productivity, it is little wonder that estimates based on the constant productivity assumption are not reliable.

SOFTWARE LIFE CYCLE

Over the last five years we have studied the manpower vs. time pattern of several hundred medium- to large-scale software development projects of different classes. These projects all exhibited the same life cycle pattern—a rise in manpower, a peaking and a tailing off (see Fig. 2). Use of the manpower curve and the corresponding equation allows us to determine the number of people needed at any time t. The time of peak effort is denoted by t_d; this time is very close to the development time for the system, which is also the time when the system reaches full operational capability. The falling part of the curve corresponds to the operations and maintenance phase of the system life cycle. During this phase the principal work is modification, minor enhancement and remedial repair (fixing bugs).

The data points shown on the manpower diagram indicate that there is scatter or noise in the data underlying the process. Empirical evidence suggests that the noise component may be up to ±25 percent of the expected manpower value during the rising part of the curve. This part of the curve corresponds to the development effort.

The form of the equation is:

$$y = K/t_d^2 \cdot t \cdot e^{-t^2/2t_d^2}$$

where:

y is the manpower at any time t,
K is the area under the curve and corresponds to the total life cycle effort in man-years,
t_d is the development time (time of peak manpower).

Large software systems and some smaller ones seem to follow this general pattern, called the Rayleigh curve. Other small systems, however, seem to have a more rectangular manpower pattern (see Fig. 1), probably because the manpower allocated is determined by management or by contractual agreements. Many small projects are established as level-of-effort projects, leading to rectangular manloading.

Rectangular manloading patterns, seldom found in large projects, appear because managers have so little insight that they hesitate to specify the resulting pattern. Rather they tend to react to the needs of the system. This reactive approach results in some lags and confusion in underapplication of effort, but the overall effect is a reasonable approximation to Rayleigh manloading.

As we have seen, the manpower equation allows us to determine our staffing—if we know the total effort (K) and the development time (t_d). We can find K and t_d once we know the expected size of the system, but first we must obtain the best estimate of the system size, and do it before development begins.

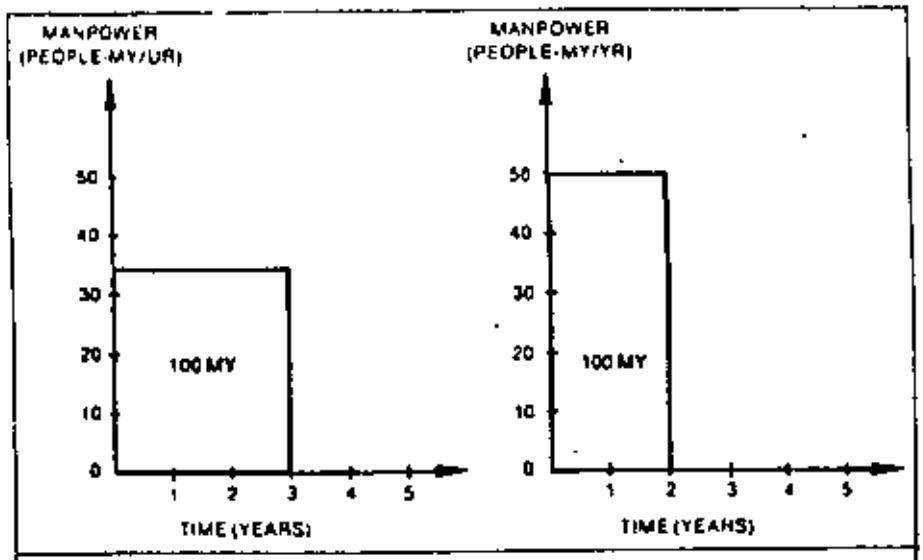


Fig. 1. The assumption that 50 people for two years is equivalent to 33 people for three years turned out not to be valid for large-scale software development.

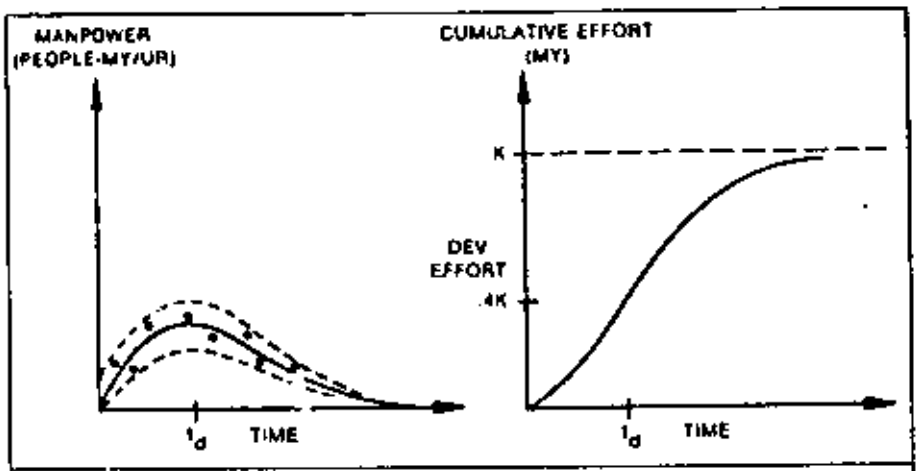


Fig. 2. The curves defined by this equation, originally applied by Lord Rayleigh to describe other scientific phenomena, have been found to fit reasonably well the manpower pattern of software development, at least within the "noise" of the data points (lefthand curve).

Before we begin to size the system, we should decide what we really want from a sizing technique. Obviously we need an estimate of the expected number of source statements. We use source statements rather than machine language instructions because they are what people write and what people can most easily relate to. People have some intuition for the size in source statements, whereas to get machine language estimates requires an uncertain conversion which introduces additional possibilities for error.

Less obvious is the need for an estimate of the uncertainty (or range) of the source statement number. The uncertainty estimate allows us to project the risk associated with our source statement estimate—something every manager should have. It also permits us to generate risk estimates for cost, schedule, and manpower—information we never had before.

At least three different estimates should be made before development of the system begins. They should be made once during the systems definition phase and at least twice during the functional design and specifications phase.

More than one estimate should be made because better and better data are available as we go from early systems definition into the functional design phase. A look at the system life cycle (Fig. 3) helps us decide when these estimates can be made.

FEASIBILITY SIZING

During the early systems definition phase we need broad estimates of the ultimate system size, development time and cost so that we can establish basic economic feasibility. At this point we have no hard data about the system we are considering because it is so early—no design has been done. Therefore, all we really can do is make an intelligent guess as to the range of size of the

system, based on what we've done in the past and what little we do know about it. If we let a equal the lowest possible number of source statements and let b equal the highest possible number of source statements, we can determine the expected size and its standard deviation (or uncertainty) by using the laws of statistics and probability.

Let us assume that we are two weeks into the systems definition of a large-scale inventory control system called SAVE. Based on past experience and what we know about this system at this point, we might broadly estimate it to be between 30,000 and 140,000 source statements. Using our statistical equations, we know that the expected number of source statements is:

$$SS = (a + b)/2 = 190{,}000/2 = 95{,}000$$

The standard deviation is:

$$\sigma SS = b - a/6 = 90{,}000/6 = 15{,}000$$

The expected size is:

$$95{,}000 \pm 15{,}000$$

By "expected" statisticians mean that there are 68 chances out of 100 that the true size lies within one standard deviation of the mean, i.e., between 80,000 and 110,000 source statements. There are 99 chances out of 100 that the true size falls within three standard deviations of the mean, i.e., between 50,000 and 140,000 source statements and less than one chance that it lies outside these limits.

While this method results in what seems to be a disconcertingly large range, it is important to understand that it is as good as we can do at this time, considering that we have almost no solid information about the system we want to build. Managers who insist on getting better estimates (meaning smaller ranges or even absolute numbers) must learn instead to work with averages of the quantities and a measure of the variability of the quantities, i.e., the standard deviation.

This is an important philosophical point because it means that only a certain level of accuracy and precision is possible at this stage and all efforts to do better are futile. As Aristotle wrote, "It is the mark of an instructed mind to rest satisfied with the degree of precision which the nature of the subject admits and not to seek exactness when only an approximation of the truth is possible."

As we continue into the life cycle and learn more about the final system, the statistics improve and uncertainty is reduced. Thus, as we approach the start of detailed design, we can reduce our risk to ranges that are considered to be within the limits of engineering accuracy in other branches of the engineering art. We achieve this result by breaking the system into pieces and estimating the pieces separately. Then we combine the pieces by means of our equations, letting the statistics of aggregation reduce our uncertainty.

Function	Smallest	Most Likely	Largest	Expected	Std Dev
File Handlers	25000.	40000.	70000.	42500.	7500.
Utilities	5000.	15000.	26000.	15167	3500.
System Procs	12000	36000.	50000.	34333.	6333.
Total				92000.	10422.

Table 1. A polling of the project team for SAVE at the early functional design phase provided the smallest, most likely, and largest estimates (average of team) of the number of source statements in each subsystem. From each range the expected value and its standard deviation was calculated.

Toward the beginning of functional design, we should know what the major subsystems will be. At this point, the members of the project team who have worked on the systems definition should estimate the size of each of the major subsystems as follows:

Let a be the smallest possible size (in source statements).

Let m be the most likely size.

Let b be the largest possible size.

The averages of these estimates for SAVE in effect, a Delphi polling of experts resulted in the first three columns of Table 1.

Parenthetically, we might note that we went through this procedure with several groups of systems engineers and they are quite comfortable with it. Most analysts or engineers are reluctant to give a single estimate of size. When they are

forced to do so, they will bias it on the high side. They prefer to give a range of sizes, because they can make this range as large or as small as they need to, depending on what they know about the system at the time. Psychologically, giving a range is not a threatening commitment.

The estimates for the three subsystems in SAVE resulted in a broad range of possible sizes. Note that the distribution is skewed on the high side in each case. This bias is typical of the beta distribution, the characteristics of which are used in PERT estimating. The PERT technique has been used successfully in other fields for more than 15 years and we adopted it here in order to find the overall system size range and distribution.

1. *Expected value.* An estimate of the expected value of a beta distribution is

$$E_t = (a + 4m + b)/6$$

This formula simply biases the result so that the expected value falls on the side about which we are more uncertain. The expected value for each subsystem is listed in the fourth column. Then the overall expected value is just the sum of the individual expected values.

2. *Standard deviation.* An estimate of the standard deviation of any distribution (including the beta) may be found by dividing the range within which 99% of the values are likely to occur by six:

$$\sigma i = |b - a|/6$$

The standard deviation of each subsystem is shown in the fifth column. The overall standard deviation is the square root of the sum of the squares of the individual standard deviations. This value turns out to be much smaller than one would guess by just looking at the individual ranges. The reason is that some actual values will be lower than expected and others will be higher. Such variations cancel each other to some extent.

At this phase the results for SAVE were:

Expected value, SS = 92,000
Standard deviation: σ = 10,422
68% range: 81,578 to 102,422
99% range: 60,735 to 123,264

The chances are fifty-fifty that the actual value will turn out to be either greater than or less than 92,000 source statements. In each case the chance that the ultimate size will be in the range shown is qualified by the proviso that the input estimates do not change. Note that we have reduced the uncertainty significantly--from 15,000 to 10,422—simply by knowing enough about the system to divide it into three major subsystems.

FUNCTIONAL DESIGN PHASE

When the functional design is a little more than half complete, a final investment decision must be made, as well as a manning plan, a life-cycle cost and a milestone schedule. At this time the preliminary specification and system design is nearly completed or at least reasonably well defined. System analysts and engineers should now be able to break the system down into its major functions and have a fairly good idea of the size range of each one.

Now we simply repeat the statistical process we have described, using the larger number of different functions. In effect, breaking the system down into more functions enables us to reduce the uncertainty in our estimate and to obtain a better estimate of the expected size.

The results with the greater number of functions now available are shown in Table 2. At the time of this analysis the project team had been working on the system about 12 weeks, and the key results were:

Expected value: SS = 98,475
Standard deviation: σ = 7,081
68% range: 91,394 to 105,556
99% range: 77,231 to 119,718

Several properties of the successive sets of data may be noted. The expected value of the size has remained within one standard deviation of the previous estimates. The size of the standard deviation has steadily declined—from 15,000 to 10,422, and on the third iteration to 7,081. In addition, what may be called our uncertainty ratio (standard deviation divided by source statements, or σSS/SS) has dropped from 16% to 7%.

Moreover, the results of applying this sizing technique to SAVE—from early systems definition through functional design phase—were similar to our experiences with groups of analysts on other projects. ●

(This is the first in a series of three articles. Part 2 will appear in the October issue.)

Part Sizing

Title: Save (Functional Design Phase) — Date: 15-Feb-79

Function	Smallest	Most Likely	Largest	Expected	Std Dev
Maintained	8675.	13375.	18625.	13467.	1658.
Search	5577.	8988.	13125.	9109.	1258.
Route	3160.	3892.	8800.	4588.	940
Status	850.	1425.	2925.	1579.	346.
Browse	1875.	4052.	8250.	4389.	1063.
Print	1437.	2455.	6125.	2897.	781.
User Aids	6875.	10625.	16250.	10938.	1563.
Incoming Msg	5830	8962.	17750.	9905.	1987.
Sys Man	9375.	14625.	28000	15979.	3104
Sys Mgt	6300	11700	36250	16225.	4992.
Comm Proc	5875	8975.	14625.	9400	1458
Total				98475.	7081

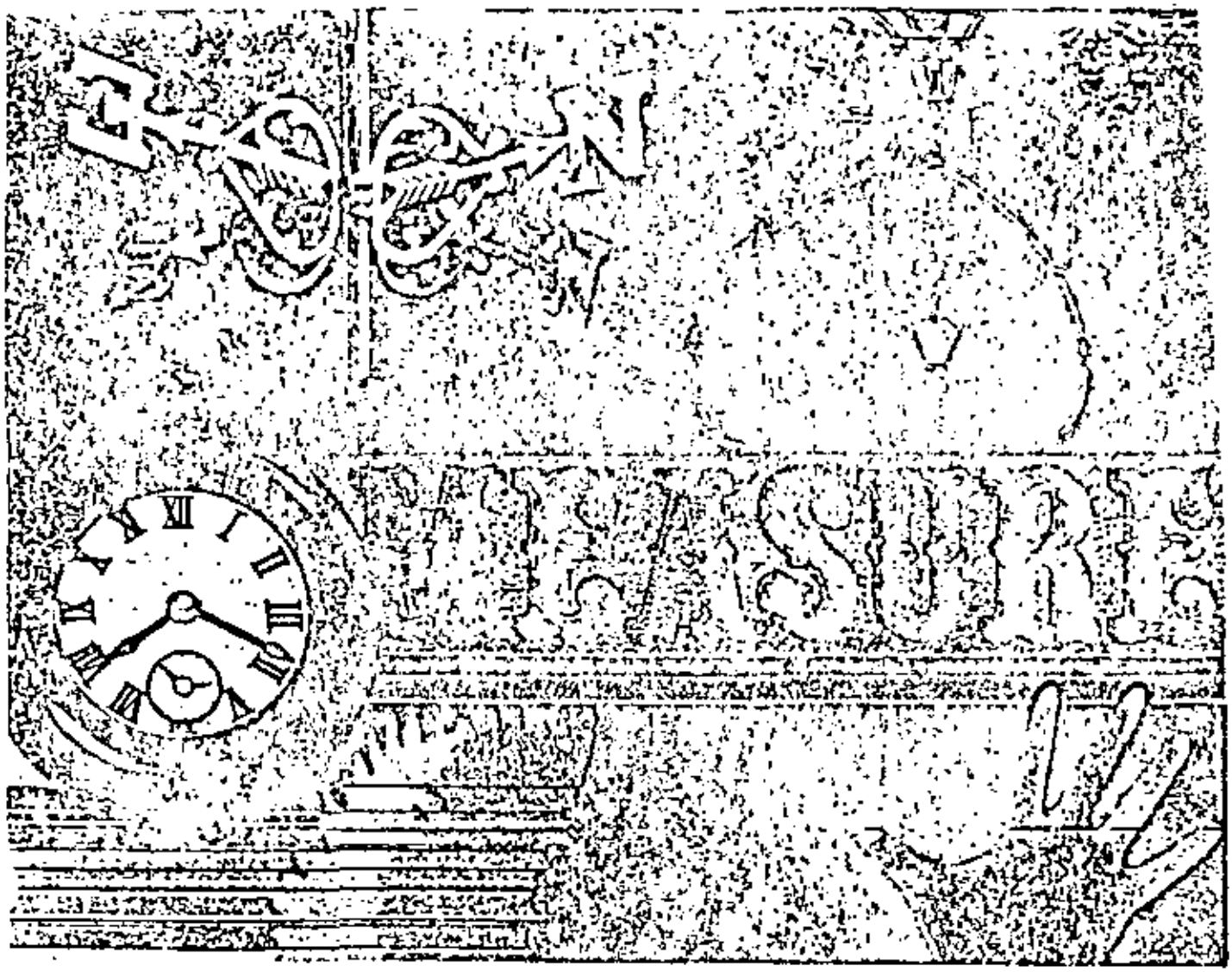
Table 2. Midway through the functional design phase of SAVE, the project team had enough information to break out 11 functions. At this level of detail the uncertainty (standard deviation) was reduced to less than half that of the first set of estimates. Moreover, at about ± 7 percent the uncertainty is no worse than that of many other engineering values.

LAWRENCE H. PUTNAM

Mr. Putnam is president of Quantitative Software Management, Inc., a firm specializing in software cost estimating and life cycle management. He has had extensive experience in industry and government in planning the quantitative aspects of software life cycle management. Mr. Putnam was manager of system technologies for General Electric Co. Prior to that, he developed and implemented systems to plan, budget and control large-scale software systems for the U.S. Army. Mr. Putnam is a member of IEEE, IEEE Computer Society, AIAA and the Society of the Sigma Xi.

ANN FITZSIMMONS

Ms. Fitzsimmons is vice president of Quantitative Software Management, Inc. of McLean, Virginia. She has designed and developed the computer-based implementation of the software estimating system produced by QSM. Before joining QSM, Ms. Fitzsimmons was a research staff member at General Electric/Information Systems Programs, where she was responsible for research projects investigating the use and impact of modern programming practices and management methods on software development projects throughout GE. Ms. Fitzsimmons received her degree in mathematics from the Univ. of North Carolina.



How to convert an estimate of system size into reasonable estimates of time, effort, and cost.

ESTIMATING SOFTWARE COSTS

by Lawrence H. Putnam and Ann Fitzsimmons

How do I know how long a software project will take and how much it will cost? This was the question posed in the September issue in the first of three articles on software estimating. Part 1 described the problems associated with trying to determine the schedule or cost for developing software before the project begins. We showed some traditional approaches and why they failed. This series of articles is designed to show how to bring the problems of estimating schedules and project control within reasonable limits.

Last month we described how to obtain good estimates of the size of the system—before development begins. This month we show how to convert this estimate of size into estimates of time, effort, and cost.

There is a fundamental relationship in software development between the number of source statements in the system and the effort, development time, and the state of technology being applied to the project. This relationship was discovered by Larry Putnam partly from theory and partly from an empirical fit of a substantial body of productivity data. The equation that describes this relationship is

$$S_s = C_k K^{1/3} t_d^{4/3}, \text{ where}$$

S_s is the number of end product source lines of code delivered
K is the life cycle effort in man-years,
t_d is the development time, and
C_k is a state of technology constant.

Fig. 1 shows a parametric graph of this equation. We have substituted development effort (D) for life cycle effort (K) in this graph since this is usually the information that managers need. This relationship is approximately $D = 0.4K$ for large systems.

In Fig. 1 the darker line labeled constraint represents the minimum time

cost estimating and produce reasonable engineering answers.

in which a system can be developed. The area below this line represents the region in which it is not feasible to attempt development of a software system. For example, in this graph we can see that it is not feasible to develop a system of 200,000 lines of code in less than 2½ years. There are other constraint conditions, depending on the type of system. This particular constraint applies to a new standalone system that must be designed and coded from scratch.

While this graph shows the functional relationship between size, time, and effort for all systems, the absolute values will differ for most organizations and even for different projects within a single organization. The C_k value in the equation determines what these absolute values will be and actually represents the state of technology an organization is applying to a system). This value will be determined by the use of modern programming practices, the language used, the development environment (on-line, interactive development versus batch), and the availability of the development machine, among other factors. While C_k is difficult to determine from its individual components because identification of these components and their relative importance is not well understood, nevertheless this value can be calibrated easily for an organization by looking at past projects. The constant should remain quite consistent for similar projects within an organization. The C_k value used in Fig. 1 (10040) represents an average state of the art development environment using on-line, interactive development.

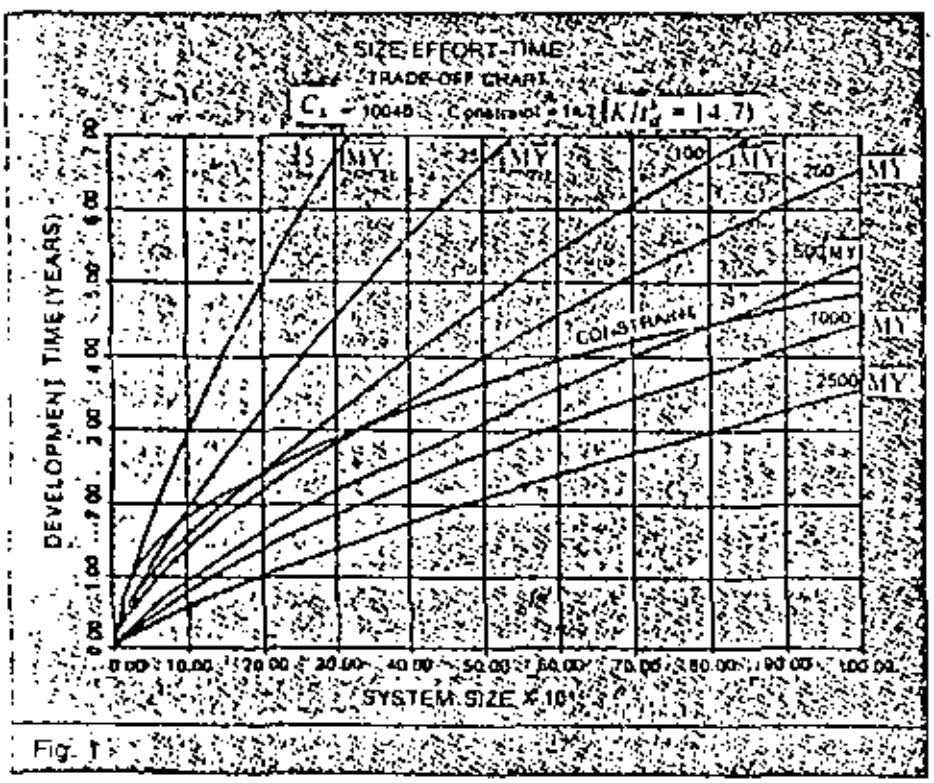


Fig. 1

	SS	DEV. EFF. (MY) (COST × $1,000)	t_d	DEV. EFF. (MY) (COST × $1,000)
		t_d = 2 yrs		FASTEST TIME
-3σ	77000	11.28 ($564)	1.63	25.80 ($1,290)
-1σ	91394	18.86 ($943)	1.75	32.16 ($1,61)
Exp.	98475	23.59 ($1,180)	1.83	35.40 ($1,770)
+1σ	105556	29.05 ($1,450)	1.86	38.73 ($1,840)
+3σ	120000	42.69 ($2,135)	1.97	45.66 ($2,280)

Table 1

A FEASIBLE REGION FOR DEVELOPMENT

Last month we described a real world estimating problem for a system called SAVE. In this example, analysts were 12 weeks into the functional design of the system and had estimated the size to be 98,475, plus or minus 7081 source statements. We will now use this estimate and the graph shown in Fig. 1 to establish a feasible region for our development effort and development time for this system. Table 1 presents three scenarios for five different points of the size distribution curve.

From this table, we can see that the minimum time for development of the system at the expected size (98,475 source statements) is approximately 1.8 years, with a corresponding development effort of about 35 man-years. However, if we take two years to do the job, we can reduce our effort to 25 man-years. We call this the trade-off law in software development. Basically, this law states that if we can relax our schedule, taking more time, we can save a considerable amount of money. Conversely, if we compress the schedule, the cost will go up dramatically.

To many managers this trade-off law may not make sense. However, there is a logical reason for it. As the number of people who must communicate or cooperate increases arithmetically, the number of interactions increases exponentially. This results in more and more time being spent on human communication and less and less being spent on productive work. One way to handle this problem is to limit the number of people working on a project at any one time, and the only way to do this in software development is to stretch out the time schedule.

From Table 1 we can also see what the minimum time schedule (and maximum development effort) would be for a broad range of sizes. Using the probability laws, we know there is less than a 1% probability that the size will be less than 77,000 source statements (as long as our initial input estimates do not change). At this size, the minimum time would be 1.63 years with a development effort of 25.8 man-years. Similarly, at the 99% level for size, the minimum time is two years with a required effort of 45.7 man-years.

We now have estimates of the expected time and effort as well as a 99% range on these parameters. How do we determine the costs?

Most of the costs associated with software development are people costs. Any extra costs, such as computer costs, supplies, etc., can be factored into the average labor rate if we assume this rate to be a fully burdened number, including overhead. Every financial department has

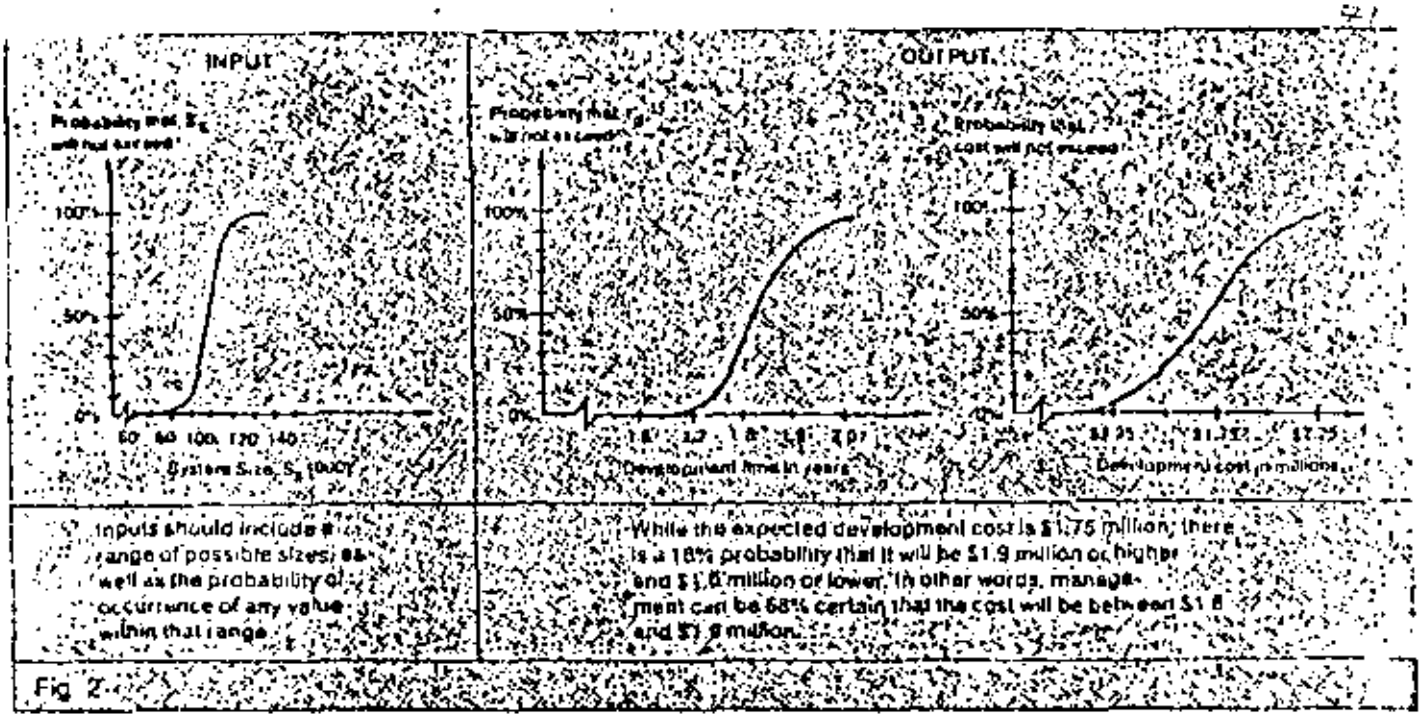


Fig. 2

a very good idea of what this will be for its particular organization; in fact, this is the only really good (stable) number we have in software development. In many industrial environments, this will be around $50,000 to $60,000 per man-year; government labor rates typically run about 10% to 30% less because of different methods of accounting for overhead. For SAVE, we assumed an average labor rate of $50,000 per man-year.

Multiplying this value by the estimates of man-years, we get an expected total cost of $1.77 million, with the range going from $1.3 million to $2.3 million. While this may seem like a disconcertingly large range, it is the best we can do at this time, given the uncertainties in our estimates of size. Moreover, it is very important to know that this uncertainty range exists and how big it is. Better to be approximately right than exactly wrong!

SIMULATION AND RISK ANALYSIS

While Table 1 gives a fairly broad range of solutions that answer many "what if" questions, they are based on the assumption that we know the input information exactly. Of course, we don't. Each input into the software equation (e.g., "the expected size of the final system is 98,475 source statements") involves its own degree—often a high degree—of uncertainty. This is where the element of risk enters the problem, and it is in the evaluation of this risk that the software manager has been able to get little help from currently available tools and techniques.

Let us look, then, at what the software manager really needs to know before he makes an investment decision.

Suppose a company is considering bidding on a large software project. The "best estimate" of the company's development cost is $1.75 million. However, there is also a 1-in-4 chance that the cost will be as high as $2.5 million, and a 1-in-4 chance that the cost will be as low as $1 million. Suppose this were a small software house and if the actual cost came in at $2.5 million it would put the company out of business. Management would be taking a 1-in-4 chance of going bankrupt if only the "best estimate" (expected value) were used. If a risk analysis had been performed, management might have chosen a safer approach.

How, then, does one determine this risk? Since almost every factor entering into our estimate of cost and schedule is subject to some degree of uncertainty, the manager needs to portray the effects of the uncertainty associated with each of these factors. Our objective is to get a realistic picture of the relative risk and the probable odds of coming out greater or less than the expected solution.

This is generally not feasible to do analytically but is nicely handled by Monte Carlo simulation. To carry out the analysis for SAVE, we performed the following steps.

1. Estimate the range of values for each of the key input factors—size of the system, difficulty of the system (handled by the constraint condition described earlier), average software development labor rate, and the probability distribution of the occurrence of each value.

2. Select at random from the distribution of values for each factor a single value. Combine each of these values for all factors and compute the time, effort, and cost for this combination of factors. (For example, one combination might be a system size of 110,000 source statements, a gradient constraint of 18, and an average labor rate of $59,000/man-year. More factors can be entered, depending on the particular problem being analyzed.)

3. Run this problem on the computer several thousand times to define the outcome probability distribution. In other words, we want to find out not only that the expected (average) minimum development time is 1.8 years; we also want to know the probability of being able to complete the system by a contract deadline of two years.

The results of the simulation yield a much better estimate than just a single number answer. Note that the expected development effort is the same as the single point deterministic value. The expected development time is also the same. This is as it should be. The simulation produces the right expected (average) values. The real value of the simulation is an estimate of the variability of the outcome values—a feature heretofore not used in software cost estimating, but one that has been badly needed in view of the poor accuracy we have experienced.

The results of the SAVE simulation for development time, development effort, and development cost can be shown in the form of probability plots to generate the projections of risk as shown in Fig. 2. We can see from these plots that there is more than a 99% probability that the cost will not exceed $2.25 million. There is only a 1% probability (one chance in 100) that the cost will be as low as $1.25 million—given the uncertainties in our input information. As we learn more and more about the final system and development environment, we can reduce the uncertainties in our inputs, thereby reducing the risks in the final estimates.

The result for the development time is extremely important from a conceptual point of view. The small variability is both a curse and a blessing. It says we can determine the development time very accurately, but at the same time it tells us we have little latitude in adjusting the development time to meet contractual requirements.

For example, the standard devia-

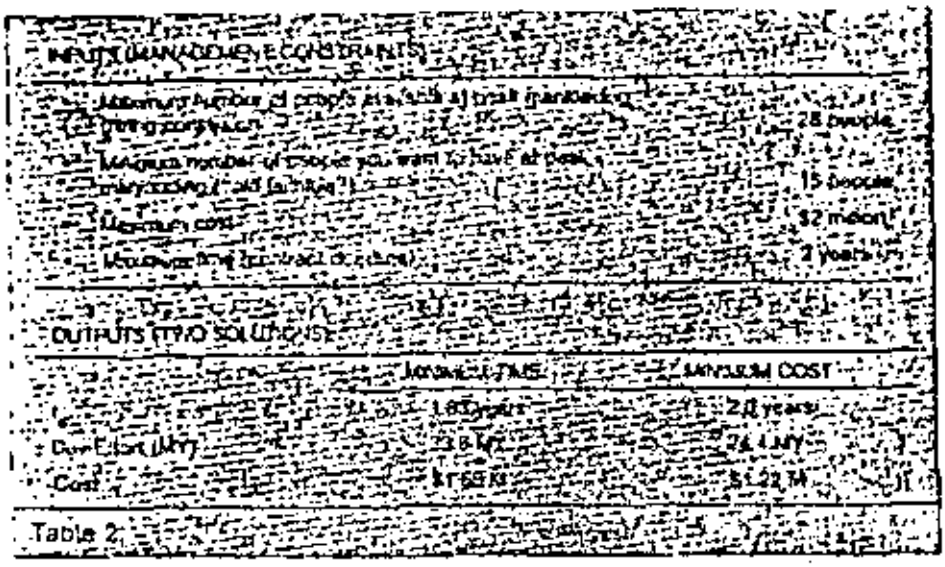


Table 2.

tion is 0.063 years or 3.28 weeks; this means that our 99% range on time is $\pm(3.28) \approx 20$ weeks.

This means that there is less than a 1% chance that development can be done in less than 1.6 years or a little over 19 months (1.4 years − .19 years = 1.6 years). Therefore, if the time schedule has been arbitrarily set by management or the customer at, say, 1.5 years, there is less than 1 chance in 100 of meeting this schedule. Similarly, we can be 99% certain that the minimum time schedule will not exceed two years.

This does not mean that if requirements change or delivery of a computer is late, the software will still come in at plus or minus 10 weeks of the expected time. These are external factors that will change t_d (development time) and must be specifically accounted for.

This is the curse. The system is very sensitive to external perturbation. Requirements changes and external disturbances (say, a 90-day delay in test bed computer delivery) will generally cause development time increments greater than two or three standard deviations.

However, management can use the knowledge of this great time sensitivity effectively in planning and contracting so that risks are always acceptable. The critical point is that time is not a free good. Development time cannot be specified by management, it is determined by the system. Software systems are inherently *narrow* band processes with sharp cutoff characteristics, and decisions need to be made with the minimum development time known beforehand. Until this fact is recognized and used, software projects will continue to "slip" and "overrun."

LINEAR PROGRAMMING TECHNIQUE

The solutions just presented are based primarily on characteristics of the software system itself. However, most managers must also deal with everyday concerns such as cost ceilings, contract deadlines, hiring practices, and capabilities.

Linear programming is a technique that produces the best possible solution to a problem bounded by an array of constraints. Typically, it is used on very large problems, such as optimizing the output of an oil refinery which is subject to such constraints as market demands and varying feedstock mixes. In the case of the software application, the mathematics is trivial, but all the power of the linear programming concept is brought to bear. More important, the manager has, for the first time, an opportunity to apply his own management constraints to the problem of developing large scale software systems.

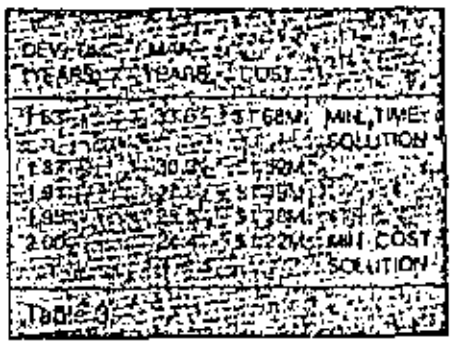


Table 3.

The linear programming algorithm allows us to enter as many of these management constraints as we need into the problem. Then we can solve the problem to either minimize or maximize some objective function (such as cost). In our case, we have two objective functions —cost and time—and usually a manager wants to minimize one or the other. For SAVE, we will solve the linear program twice, first minimizing time and then cost.

Table 2 shows the constraints entered by the manager and the two solutions provided by the linear program.

Our feasible development region is now identified in between these two solutions. In being able to invoke this powerful technique, we produce two constrained optimal solutions, the best that can be done within the constraints, and all other feasible effort-time choices. The simplicity of this statement should not cause us to overlook its importance. We have progressed from an inability to guarantee even one feasible solution to the ability to get all feasible choices and to select the best possible one from among those identified.

Let's examine the range of feasible, time-effort-cost combinations for SAVE identified in Table 3. This will let us identify excellent opportunities to save money on the project.

Note that we can save almost $500,000 by taking the maximum time and minimum cost solution. However, in doing this we increase our risk exposure. If we plan the job at t_d = 1.8 years and 35 man-years ($1.77 million), the probability that it will not take us more than two years to do is very high. But if we elect to take advantage of the trade-off law by planning the job at two years with 24 man-years ($1.2 million), we have reduced the probability that it will not take us more than the contract delivery time to 50%.

These odds are too low for most situations. However, a 90% probability may be very acceptable. This occurs at a planned development time of t_d = 1.92 years with 28 man-years of effort or $1.4 million. So it is possible to keep risk reasonable (<10% chance of exceeding delivery time) and save $300,000 compared with the minimum time solution.

The managerial questions—"Can I do it? How much? How long? How many people? What's the risk? What's the trade-off?"—can be answered with numbers.

With the application techniques described we have been able to quantitatively come to grips with the software cost estimating problem and produce reasonable engineering answers. We need only know the state of technology we are going to apply to the development (C_k), estimates of the number of lines of code, and the software equation which we solve along with a constraint relationship to get the management parameters (K, t_d) of the Rayleigh/Norden equation. Simulation provides suitable statistics for risk estimation. The linear programming alternative provides constrained optimal solutions and the range of all feasible solutions.

The economics of the software development process is startling. The indications are clear that apparently innocent management choices can be made that affect cost by multiples of 5 to 10. With that kind of variation on multimillion dollar projects, managers need to know the choices, sensitivities, and influences they can bring to bear, over the whole life cycle, and in numbers. ●

(This is the second of three articles. Part 3 will appear in the November issue.)

Understanding the software life cycle helps managers avoid inefficient use of manpower.

ESTIMATING SOFTWARE COSTS

by Lawrence H. Putnam
and Ann Fitzsimmons

This is the final section of a three-part article that began in September and continued in October.

Most large-scale software takes one to three years to develop and has an operational life of six to 10 years. This is a life cycle. We have studied the behavior of this life cycle, just as other businesses study the life cycle of a new product. We have developed a model of the software life cycle that can be used by management to forecast and manage the schedules and manloading requirements of software projects. Understanding the life cycle can help prevent managers from misapplication or inefficient management of the available manpower.

Large-scale software development can be thought of as a series of interrelated tasks. At the beginning of the project there may be only one or two major tasks, for example, defining the major system modules and the interactions among them. Each of these major tasks is broken into more and more subtasks. There must eventually come a time when the number of subtasks remaining reaches a peak, levels off, and finally begins to decline.

At the beginning of development, when only a few major tasks have been defined, only a few key people can actually do productive work on the project. These key people must become intimately familiar with the overall function and design of the system. They are responsible for breaking the problem down into manageable pieces—a task that becomes virtually impossible as more and more people are applied to it. However, as the problem becomes better defined, additional people can be applied successfully.

If people are applied as useful work becomes available for them we begin to see a manloading pattern like the one shown in Fig. 1.

We have looked at data from hundreds of past software systems and noticed that almost all large software development efforts show a pattern of manloading very similar to Fig. 1. Moreover, for all large systems, the peak of the curve occurs at approximately the same time the system reaches full operational capability. (We will call this the development time for the system, and denote it as t_d.)

There is a logical reason for this. Just before the system reaches full operational capability, the total development task has been subdivided into many subtasks—testing and final integration, installation, writing documentation, fixing remaining bugs, etc. It is possible to have many types of personnel—programmers, analysts, quality assurance personnel, clerical, administrative and training support—all usefully employed. This was not true earlier in the development phase.

At the time the system is accepted and becomes fully operational we move into the operations and maintenance phase of the software life cycle. This corresponds to the falling part of the curve. The principal work during this phase is modification, minor enhancements, and remedial repair (fixing bugs).

The form of the curve and the accompanying equation allow us to project what the manpower requirements and cashflow for system development will be at any given time. The equation representing this curve is:

$$\dot{Y} = K/t_d^2 \; t \; e^{-t^2/2t_d^2}, \text{ where}$$

K is the life cycle effort in manyears,

t_d is the development time for the system, and

t is the independent variable representing any point in the life cycle—current elapsed time.

Obviously, K and t_d can take on a range of values. A change in K or a change in t_d (or both) will result in a change in the shape and magnitude of the curve. Let's look at what this means to a manager faced with the real problem of applying manpower in the most effective way.

We will assume that K = 100 manyears for a given system. Fig. 2 shows what happens to the distribution of this effort simply by varying the time to reach peak manpower. If the development of the system is set at two years, then the system would require a relatively gradual application of manpower, peaking at 30 people 24 months into the project. However, notice what happens when we reduce the development time to one year (assuming this were feasible)—this would require hiring

and assimilating 61 people in 12 months. While this may be possible in some larger corporations, many organizations would find the practical problems associated with hiring—and productively applying—this many people in this short time nearly impossible.

This is a good example of what project management must contend with when the development time for a system is arbitrarily set by senior management.

There is another factor in software development that makes it unreasonable to try to compress development time. The tasks in a software project must interact with each other; for example, the initiation of one task may depend on the successful completion of a preceding task. Thus the software schedule can be compressed only so much; attempts to reduce it more will result in wasted effort, at best, and may even result in negative effort—work that will have to be redone.

DETERMINING THE BEST MANLOADING

Part II of this article described a software estimating problem for a system called SAVE. We showed that there exists a fundamental relationship between the size of the system, the effort (K), the development time (t_d) and the state of technology an organization applies to the project. Using this fundamental relationship, we determined that the minimum feasible schedule for development of SAVE was 1.81 years with a corresponding development effort of 39.1 man-years and a life cycle effort (K) of 19.2 man-years. Because there was a time constraint on the system, management elected to develop the system in the minimum time (1.81 years).

Using our manpower equation, we can determine the best manloading for this system over the 1.81 years, and even project ahead throughout the life cycle. We substitute our management parameters into the Rayleigh equation,

$$y = 39.1 \cdot t \cdot e^{-t^2/2(1.81)^2}$$

Fig. 3 shows a plot of the expected manloading for SAVE.

The range of values to either side of the curve in Fig. 3 represents the uncertainties in our estimate of how many people can be expected to be working on the project at any time. For example, our best estimate is that there will be 22 people working on the project one year into development; but there is a 10% chance that the number will be 19 or lower, and 25 or higher—simply as a result of the uncertainties in our estimates of K and t_d.

A very experienced software manager knows that he/she must pursue enjoy-

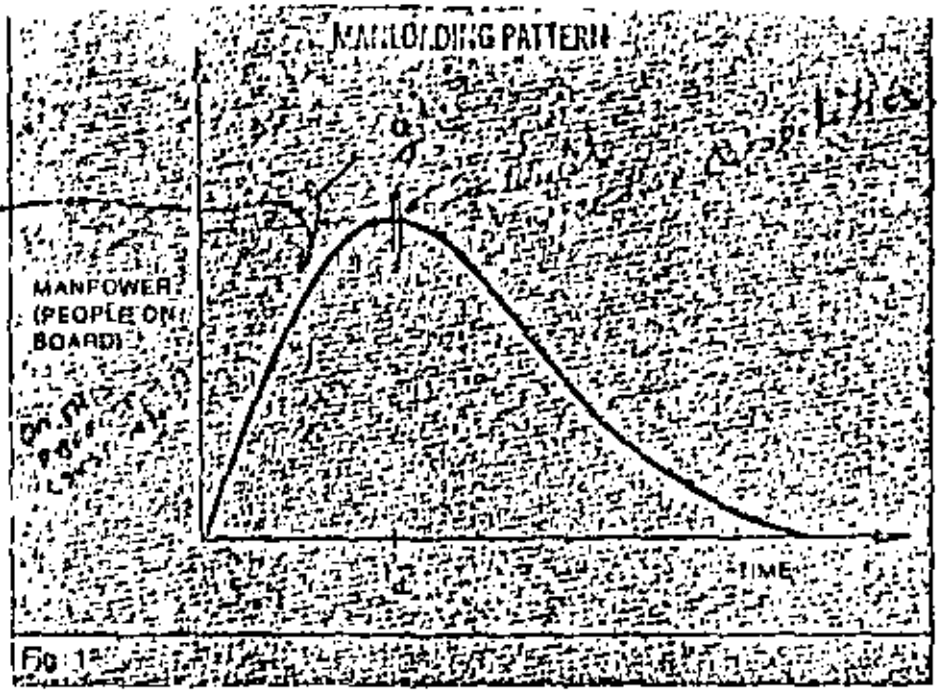


Fig. 1

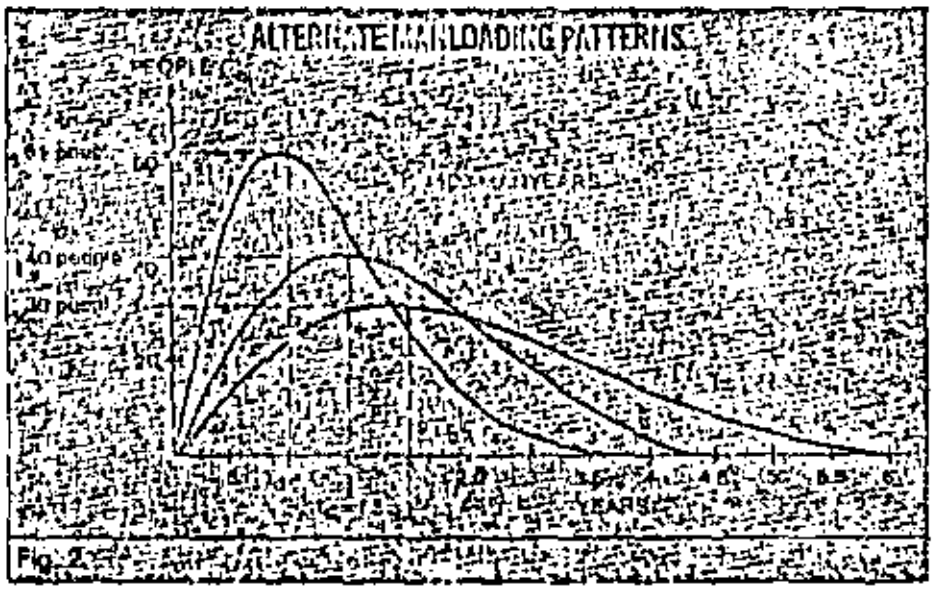


Fig. 2

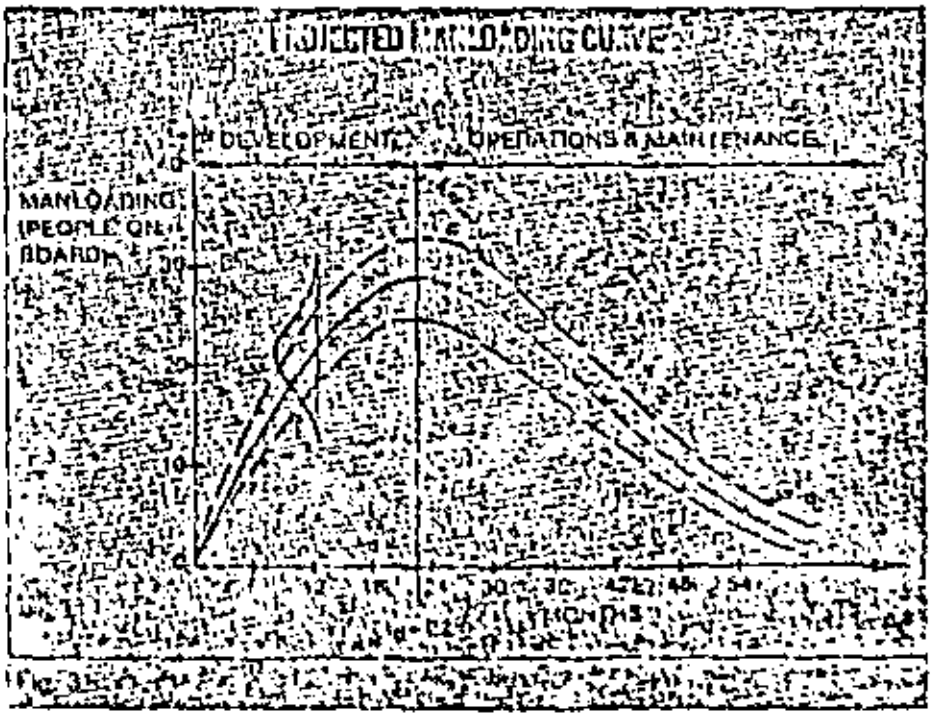


Fig. 3

CASHFLOW PROJECTIONS

MONTH	PEOPLE		COST/MTH (X $1000)		CUM COST (X $1000)	
	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV
JAN 79	1.	0.	5.	1.	5.	1.
FEB 79	3.	0.	14.	2.	18.	3.
MAR 79	5.	1.	23.	4.	41.	7.
APR 79	8.	1.	33.	6.	73.	12.
MAY 79	10.	2.	41.	8.	115.	19.
JUN 79	12.	2.	50.	8.	165.	27.
JUL 79	14.	2.	59.	11.	223.	37.
AUG 79	16.	2.	69.	13.	292.	49.
SEP 79	18.	3.	78.	15.	369.	62.
OCT 79	19.	3.	84.	15.	454.	76.
NOV 79	21.	3.	94.	18.	547.	91.
DEC 79	22.	3.	101.	19.	648.	108.
JAN 80	23.	3.	104.	19.	753.	126.
FEB 80	25.	3.	109.	19.	863.	144.
MAR 80	26.	4.	120.	21.	980.	164.
APR 80	27.	4.	123.	22.	1104.	184.
MAY 80	27.	4.	126.	22.	1231.	206.
JUN 80	28.	4.	128.	23.	1359.	227.
JUL 80	29.	4.	136.	24.	1493.	249.
AUG 80	29.	4.	136.	24.	1630.	272.
SEP 80	29.	4.	136.	23.	1767.	295.
OCT 80	29.	4.	139.	22.	1905.	318.

Table 1

lations in business are data that are not so precise. The software development field is certainly not excluded from this situation; in fact, the data in software development are notoriously imprecise. In this example, we had estimated that the life cycle effort was 89 manyears with a statistical uncertainty of + or − nine manyears. Similarly, the minimum development was 1.81 years + or − 0.63 years.

By solving the manloading equation several thousand times at each time interval, and varying our inputs (K and t_d) according to this statistical uncertainty, we can get a more realistic picture of the odds of having more or less people on board than the expected number at any time.

Now that we have a good estimate of the manpower requirements each month we can determine what our cashflow requirements will be throughout the project. All we need is the average burdened labor rate (including overhead) for software development. For this organization, we knew from past experience that this figure was $50,000/MY. Multiplying this figure by the manpower each month, we obtain the instantaneous and cumulative costs requirements shown in Table 1.

An interesting pattern in the occurrence of major milestones during development can be seen. Data from several hundred systems representing all types of development environments have shown that the relative occurrence of these milestones to the total development time is remarkably stable in most organizations and environments. This pattern has been noted even in organizations where the milestones have been arbitrarily (and often unrealistically) set by management, e.g., "Critical Design Review will occur three months after project start." In almost all these situations, the actual milestone accomplishment has slipped to the demands of the system itself.

TIMES OF MAJOR MILESTONES

EVENT	t/t_d	TIME FROM START (MONTHS) FOR 'SAVE'
CRITICAL DESIGN REVIEW	.43	9
SYSTEMS INTEGRATION TEST	.67	15
PROTOTYPE TEST	.80	17
START INSTALLATION	.93	20
FULL OPERATIONAL CAPABILITY	1.0	22

Table 2

Table 2 shows the time from project start that these major milestones should occur.

This analysis should be used not only as a planning tool, but can be very helpful in measuring actual accomplishment once the project has begun. Past data have shown that if a single milestone slips, there is little hope of catching up later on.

For instance, in the Table 2 plan, if we successfully accomplish the Critical Design Review at 11 months, rather than at nine months, we should rework our proposed schedule rather than trying to speed up development. One of the best methods of preventing severe slippages and minimizing the impact of any slippage is to recognize them early, bite the bullet right then, and immediately revise project plans.

In the past, application software development has been unnecessarily characterized by cost overruns, manpower shortages, and schedule slippages. How much will it cost? How long will it take? What are the chances of the system not being operational on time? How many people will be required? Without reliable answers to these questions, a manager cannot expect to effectively manage the software development process.

A proven model of the software development process has been developed and used in this series of articles to provide numerical answers to the manager's questions. Some of the most powerful problem-solving techniques known today —the PERT algorithm, linear programming and simulation —have been simply applied to provide accurate size, cost, time, people, and milestone estimates for computer software systems development. Limiting constraints and viable alternatives are identified, permitting real design-to-cost and design-to-schedule options. Probability risk profiles can be calculated to provide the decision-maker with the hard numbers required to make sound decisions about software projects. Like ships upon the sea, software people need some numbers to guide them in the right direction at the right time to arrive on schedule at cost. ◻

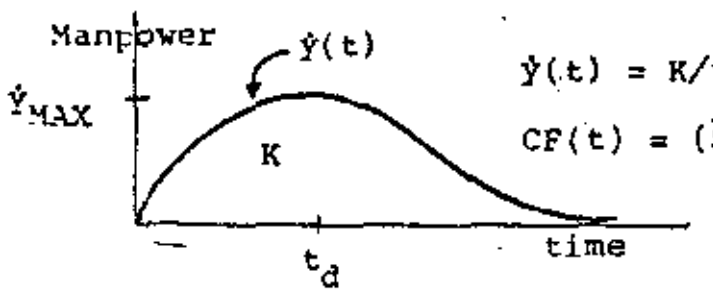
THE FOLLOWING ERRORS WERE PRINTED IN THE OCTOBER 1979 DATAMATICN ARTICLE "ESTIMATING SOFTWARE COSTS."

LOCATICN	AS PRINTED	SHOULD READ
Top of p. 171	$S_s = C_k K^{1/3} t_d$	$S_s = C_k K^{1/3} t_d^{4/3}$
Figure 1, p. 172	C_M = 10040	C_k = 10040
	$ MT	5 MY (manyears)
	25 MT	25 MY
	100 MT	100 MY
	⋮	⋮
	2500 MT	2500 MY (manyears)

The constraint relation is $K/t_d^3 = 14.7$ for the example given.

Solutions are a function of system size. This complicating issue was not presented in the DATAMATION articles. The following observations will help to clarify this situation

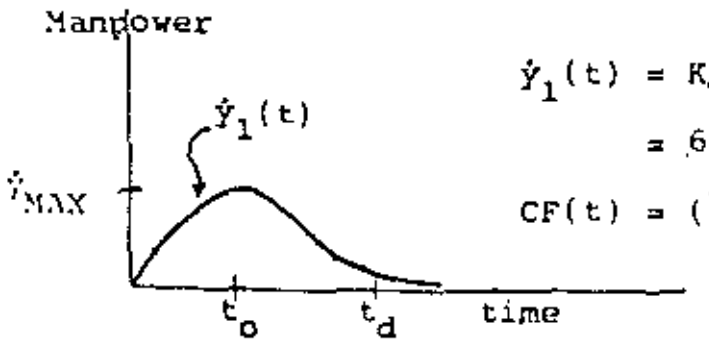
For large systems ($S_s \geq 70{,}000$):



$$\dot{y}(t) = K/t_d^2 . t . \exp(-t^2/2t_d^2)$$

$$CF(t) = (\overline{\$/MY}) . \dot{y}(t)$$

For small systems ($S_s \leq$ [illegible]):



$$\dot{y}_1(t) = K/t_d^2 . t . \exp(-3t^2/t_d^2)$$

$$= 6K_1/t_d^2 . t . \exp(-3t^2/t_d^2)$$

$$CF(t) = (\overline{\$/MY}) . \dot{y}_1(t)$$

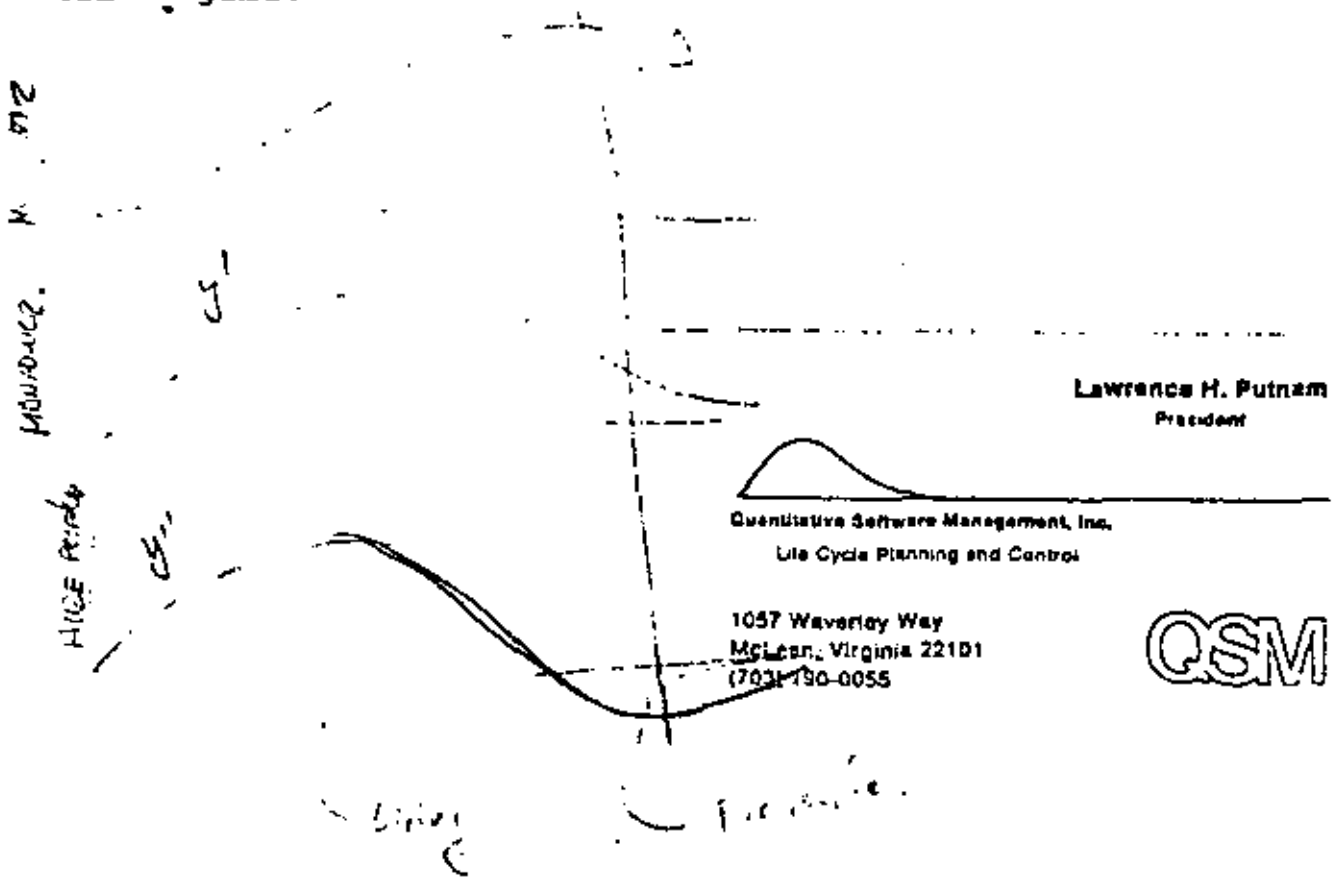
$$K_1 = K/6; \quad t_o = t_d/\sqrt{6}$$

For systems in the size range from 18,000 to [illegible] source statements

In this range the analysis is considerably more complicated because of the rapid growth in overhead or support effort (documentation, testing, integration, hierarchical management, etc.). This requires another parameter to be introduced into the solution process. This in turn makes the solution too complicated to do by hand or graphically without considerable specific training.

The solution has therefore been automated and is entirely consistent with larger and smaller systems. The automated solution is known as SLIM and is available anywhere in [illegible] from a number of overseas locations via a time [illegible] service.

If you work in this size regime frequently it is strongly recommended that you use the automated [illegible] change very quickly [illegible] probability of being [illegible] productivity rules of thumb are almost totally worthless in this size regime.



THIS PAGE INTENTIONALLY LEFT BLANK.

3.- PROGRAMACION Y DISEÑO ESTRUCTURADO

PDP

On the Composition of Well-Structured Programs

Introduction

In the first decade of computers, say up to the early sixties, computers were quite limited in their power. The task of the programmer was to formulate algorithms in the specific order codes of these machines so that they were utilized as effectively as possible. Primarily because of their limitations, this task was achieved by collecting sets of clever techniques and startling tricks, and by finding applications for them as frequently as possible. Examples of such techniques were the programmed self-modification of parts of the program, such as, for instance, the conversion of conditional jumps into dummy instructions and vice versa, or the sharing of store for functionally independent, but never simultaneously used auxiliary variables.

Tricks were necessary at this time, simply because machines were built with limitations imposed by a technology in its early development stage, and because even problems that would be termed "simple" nowadays could not be handled in a straightforward way. It was the programmers' very task to push computers to their limits by whatever means available. We should recall that the absence of index registers (and indirect addressing), for example, made automatic code modification a mere necessity (see also [1, 2]).

The essence of programming was understood to be the *optimization of the efficiency* of particular machines executing particular algorithms. As computers grew more powerful, the problems posed to the programmers grew proportionally, and as a result, the growing power of hardware did not ease, but rather increased the burden. The elimination of deficiencies, errors and blunders — called debugging — became the overwhelming problem.

Understandably, the remedy was sought in the development and use of better tools in the form of programming languages. The amount of resistance and prejudices which the farsighted originators of FORTRAN had to overcome to gain acceptance of their product is a memorable indication of the degree to which programmers were preoccupied with efficiency, and to which trickology had already become an addiction. However, once these adversities and fears had been overcome, FORTRAN had a tremendous impact — an impact that is still felt today. ALGOL 60 followed several years later; it went beyond FORTRAN in several significant respects, but essentially shared the same purpose and intention. In particular, it extended to the level of statements what FORTRAN had introduced on the level of (arithmetic) expressions: *structure*. But ALGOL 60 was not very successful when measured by its frequency of use in technical and commercial applications. There are many reasons for this, one being that it appeared on the scene when the relevance of structure had not yet been widely recognized, and its restrictiveness against the use of clever tricks was considered to be a handicap and a deficiency. The law of the "Wild West of Programming" was still held in too high esteem! The same inertia that kept many assembly code programmers from advancing to use FORTRAN is now the principal obstacle against moving from a "FORTRAN style" to a structured style.

As the power of computers on the one side, and the complexity and size of the programmer's task on the other continued to grow with a speed unmatched by any other technological venture, it was gradually recognized that the true challenge does not consist in pushing computers to their limits by saving bits and microseconds, but in being capable of organizing large and complex programs, and assuring that they specify a process that for all admitted inputs produces the desired results. In short, it became clear that any amount of efficiency is worthless if we cannot provide *reliability* [4]. But how can this reliability be provided? Here structure enters the scene as the one essential tool for mastering complexity, the effective means of converting a seemingly senseless mass of bits or characters into meaningful and intelligible information. We must recognize the strong and undeniable influence that our language exerts on our ways of thinking, and in fact defines and delimits the abstract space in which we can formulate — give form to — our thoughts.

But now the term *structured programming* has been coined, and it seems finally to be achieving what the term "structured language" was unable to suggest. It was first used by E.W. Dijkstra [3], and has spread with various interpretations and connotations since then. It is the expression of a conviction

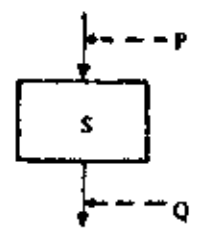
that the programmers' knowledge must not consist of a bag of tricks and trade secrets, but of a general intellectual ability to tackle problems systematically, and that particular techniques should be replaced (or augmented) by a method. At its heart lies an *attitude* rather than a recipe: the admission of the limitations of our minds. The recognition of these limitations can be used to our advantage, if we carefully restrict ourselves to writing programs which we can manage intellectually, where we fully understand the totality of their implications.

1. Intellectual manageability of programs

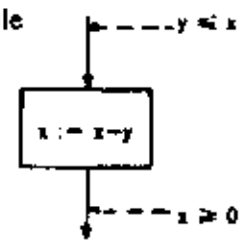
Our most important mental tool for coping with complexity is *abstraction*. Therefore, a complex problem should not be regarded immediately in terms of computer instructions, bits, and "logical words," but rather in terms and entities natural to the problem itself, abstracted in some suitable sense. In this process, an abstract program emerges, performing specific operations on abstract data, and formulated in some suitable notation — quite possibly natural language. The operations are then considered as the constituents of the program which are further subjected to decomposition to the next "lower" level of abstraction. This process of *refinement* continues until a level is reached that can be understood by a computer, be it a high-level programming language, FORTRAN, or some machine code [5, 6].

For the intellectual manageability, it is crucial that the constituent operations at each level of abstraction are connected according to sufficiently simple, well understood *program schemas*, and that each operation is described as a piece of program with *one starting point* and a *single terminating point*. This allows defining states of the computation (P, Q), i.e., relations among the involved variables, and attaching them to the starting and terminating points of each operation (S). It is immaterial, at this point, whether these states are defined by rigorous mathematical formulas (i.e., by predicates of logical calculus) or by sufficiently clear and informative sentences, or by a combination of both. The important point is that the programmer has the means to gain clarity about the interface conditions between the individual building blocks out of which he composes his program [7].

(1)



Example



An example may clarify the issues at this point. The reader should be aware that any example that is sufficiently short to fit onto a single page cannot be much more than a metaphor, probably unconvincing to habitual skeptics. The important thing is to abstract from the example and to imagine the same method being applied to large programming problems.

Example 1: Sequential merging

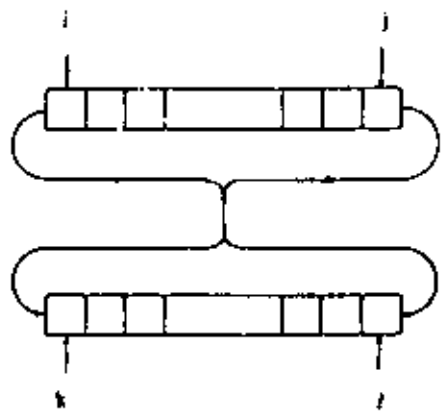
Given a set of $n = 2^N$ integer variables $a_1 \cdots a_n$ find a recipe to permute their values such that $a_1 \leq a_2 \leq \cdots \leq a_n$ using the principle of sequential merging. Thus, we are to sort under the assumption of strictly sequential access. Briefly told, we shall use the following algorithm:

1) Pick individual components $a_i^{(1)}$ and merge them into ordered pairs, depositing them in a variable $a^{(2)}$.

2) Pick the ordered pairs from $a^{(2)}$ and merge them pairwise into ordered quadruples, depositing them in a variable $a^{(3)}$.

3) Continue this game, each time doubling the size of the merged subsequences, until a single sequence of length $n = 2^N$ is generated.

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.



(2)

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 se-

quence. 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 computation of the palindrome property p of d. It is plain that it also consists of the repeated, sequential comparison of corresponding digits. We start by picking the first and the last digits, and then proceed inwards. Let i and j be the indices of the compared digits.

```
i := 1; j := L;                                            (11)
repeat compare the digits;
  i := i+1; j := j-1
until (i ≥ j) or digits are unequal
```

A last refinement leads to a complete solution entirely expressed in terms of a conventional programming language with adequate structuring facilities.

```
n := 0;                                                    (12)
repeat n := n+1; s := n*n; L := 0;
  repeat L := L+1;
    r := s div 10; d[L] := s - 10*r;
    s := r
  until s = 0;
  i := 1; j := L;
  repeat p := d[i] = d[j];
    i := i+1; j := j-1
  until (i ≥ j) or ¬p;
  if p then write (n)
until n = N
```

This ends the presentation of Example 2.

2. Simplicity of composition schemes

In order to achieve intellectual manageability, the elementary composition schemes must be simple. We have encountered most of the truly fundamental ones in this second example. They encompass *sequencing*, *conditioning*, and *repetition* of constituent statements. I should like to elaborate on what is meant by simplicity of composition scheme. To this end, let us select as example the repetitive scheme expressed as

```
while B do S                                               (13)
```

It specifies the repeated execution of the constituent statement S, while — at the outset of each repetition — condition B is satisfied. The simplicity consists in the ease with which we can infer properties about the while-statement from known properties of the constituent statement. In particular, assume that we know that S leaves a property P on its variables unchanged or *invariant* whenever B is true initially; this may be expressed formally as

$$P \wedge B \{S\} P \tag{14a}$$

according to the notation introduced by Hoare [8]. Then we may infer that the while statement also leaves P invariant, regardless of the number of times S was repeated. Since the repetition process terminates only after condition B has become false, we may infer that in addition to P, also $\neg B$ holds after the execution of the while statement. This inference may be expressed formally as

$$P \{\text{while } B \text{ do } S\} P \wedge \neg B \tag{14b}$$

This formula contains the essence of the entire while-construct. It teaches us to look for an invariant property P, and to consider the result of the repetition to be the logical combination of P and the negation of the continuation condition B. A similar pattern of inference governs the repeat-construct used in the preceding examples. Assuming that we can prove

$$Q \vee (P \wedge \neg B) \{S\} P \tag{15a}$$

about S, then we may conclude that

$$Q \{\text{repeat } S \text{ until } B\} P \wedge B \tag{15b}$$

holds for the repeat-construct.

There remains the question, whether all programs can be expressed in terms of hierarchical nestings of the few elementary composition schemes mentioned. Although in principle this is possible, the question is rather, whether they can be expressed conveniently, and whether they *should* be expressed in such a manner. The answer must necessarily be subjective, a matter of taste, but I tend to answer affirmatively. At least an attempt should be made to stick to elementary schemes before using more elaborate ones. Yet, the temptation to rescind this rule is real, and the chance to succumb is particularly great in languages offering a facility like the goto statement, which allows the instantaneous invention of any form of composition, and which is the key to any kind of structural irregularity.

The following short example illustrates a typical situation, and the issues involved.

Example 3: Selecting distinct numbers

Given is a sequence of (not necessarily different) numbers $r_0, r_1, r_2, \cdots$. Select the first n distinct numbers and assign them sequentially to an array variable a with n elements, skipping any number r_i that had already occurred. (The sequence r may, for instance, be obtained from a pseudo-random number generator, and we can rest assured that the sequence r contains at least n different numbers.)

An obvious formulation of a program performing this task is the following:

```
for i := 1 to n do                                              (16)
begin L: get(r);
  for j := 1 to i-1 do
    if a[j] = r then goto L;
  a[i] := r
end
```

It cannot be denied that this "obvious" solution has been suggested by the tradition of expressing a repeated action by a for statement (or a DO loop). The task of computing a value for a is decomposed into n identical steps of computing a single number $a[i]$ for $i = 1 \cdots n$. Another influence leading to this formulation is the tacit assumption that the probability of two elements of the sequence being equal is reasonably small. Hence, the case of a candidate r being equal to some $a[j]$ is considered as the exception: it leads to a break in the orderly course of operations and is expressed by a jump. The elimination of this break is the subject of our further deliberations.

Of course, the goto statement may be easily — almost mechanically — replaced in a transcription process leading to the following goto-less version.

```
for i := 1 to n do                                              (17)
begin
  repeat get(r); ok := true;
      j := 1;
    while (j<i) ∧ ok do
      begin ok := a[j] ≠ r; j := j+1
      end
  until ok;
  a[i] := r
end
```

The transcription consists of the replacement of the for statement with a fixed termination condition depending on the running index j by a more flexible while statement allowing for more complicated, composite termination (or rather continuation) conditions. But this solution appears quite unattractive. It is admittedly less transparent than the program using a jump, in spite of the fact that the most frequently heard objection to the use of jumps is that they

obscure the program. The other objection is that the goto-less version (17) requires more comparisons and tests, and hence is less efficient.

The crux of the matter is that well-structured programs should not be obtained merely through the formalistic process of eliminating goto statements from programs that were conceived with that facility in mind, but that they must emerge from a proper design process. Two alternative solutions are presented here as illustrations.

In the first case, we abandon the notion that the program must necessarily be based on the statement

```
for i := 1 to n do                                    (18)
a[i] := the next suitable number
```

and consider the basic iteration step to consist of the generation of the next element of the sequence 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
```

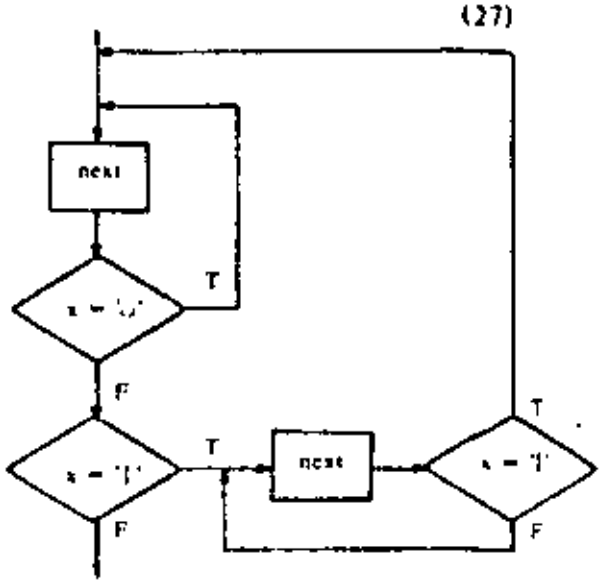
The following, and last two, examples of problems are added to show that often the need for an *exit in the middle* construct is based on a preconceived notion rather than on a real necessity, and that sometimes an even better solution is found when sticking to the fundamental constructs.

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:

(28)

```
next:
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

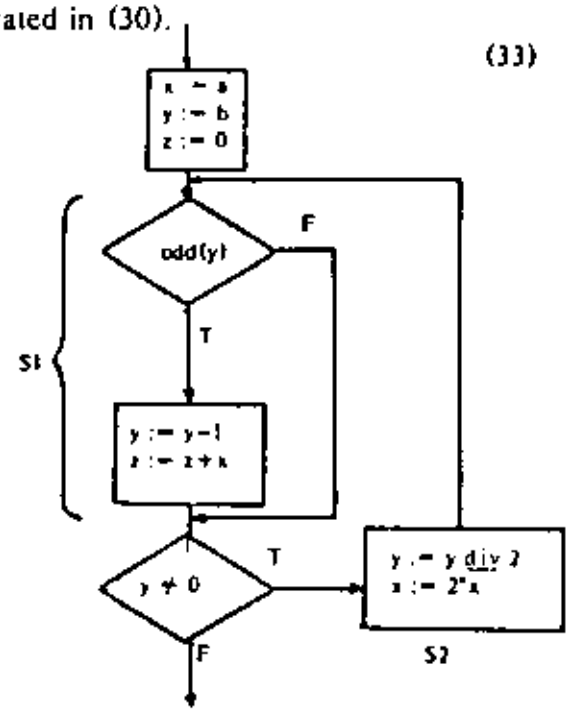
```
while x = '␣' do next                                        (32)
```

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. $x*y+z = a*b$ is invariant under the two repeated assignments. (To verify this claim, substitute $y-1$ for y and $z+x$ for z; this yields $x*(y-1)+(z+x) = x*y+z = a*b$, i.e., the original equation.) At entry the equation is satisfied, since $z = 0$, $x = a$, $y = b$.

Note that the invariant equation combined with the negation of the continuation condition yields $(y = 0)$ and $(x*y+z = a*b)$, i.e., the desired result $z = a*b$.

If we now insert any statement at the place of the invariant which leaves the product $x*y$ unchanged, the result of the program will evidently remain the same. Such a statement is, e.g., the pair of assignments

```
y := y div 2; x := 2*x                                    (36)
```

under the condition that y is even. But if a relation is invariant over a statement, it remains so regardless of how often the statement is executed. This suggests the following, quite evidently correct, efficient, and elegant solution. It contains no exit-in-the-middle loop.

```
x := a; y := b; z := 0;                                   (37)
while y ≠ 0 do
begin {y > 0 and x*y+z = a*b}
  while even(y) do
    begin y := y div 2; x := 2*x
    end;
  y := y-1; z := z+x
end
```

So much for examples, whose purpose was to sketch and elucidate the basic ideas behind the methods of structured programming and stepwise refinement.

Conclusions

Skeptics will, of course, doubt that these methods represent any progress over the techniques of the old days — in fact, that they are *methods* at all. I can merely say that in my own experience, the new approach has improved my attitudes and abilities towards programming very considerably, and the experiences of others confirm this impression [10, 11]. A systematic, orderly, and transparent approach is mandatory in any sizable project nowadays, not only to make it work properly, but also to keep the programming cost within reasonable bounds. It is the very fact that computation has become very cheap in contrast with salaries of programmers, that squeezing the machines to yield their utmost in speed has become much less important than reliability, correctness, and organizational clarity. It is not only more urgent, but also much more costly to correct an efficient, but erroneous program, than to speed up a relatively slow, but correct program. In the past, the debugging phase has taken a ridiculously large percentage of the development cost in most large projects. The aim now is to eliminate the necessity of debugging by creating bug-free products in the first place. Doesn't this bring to mind the medical slogan "prevention is better than healing"!

The criticism has been voiced that the method of structured programming is in essence nothing more than programming by painstakingly avoiding the use of jumps (goto statements). One may, indeed, come to this conclusion by looking at the entire issue in the reverse direction. But in fact, the method of stepwise decomposition and refinement of the programming task automatically leads to goto-free programs; the absence of jumps is not the initial aim, but the final outcome of the exercise. The claim that structured programming was invented by proving that all programs can be formulated without goto statements is therefore based on a fundamental misunderstanding.

The question of whether jumps enter the picture or not is basically a matter of the level of decomposition or refinement to which the programming process is carried. Ultimately — that is in machine code — there can be no doubt about the presence of jump instructions. The moral of the story is that jumps must not be used in the initial conception of a general algorithmic strategy, and in fact should be delayed as long as possible. With today's state of technology, the introduction of jump instructions can be left to compilers of languages that offer adequate, judiciously chosen, disciplined structuring facilities.

One of the essential facilities for this purpose, besides conditional and repetitive statements, is the *recursive procedure*. In many cases it emerges as the natural formulation of a solution, such as, for instance, in most cases of backtracking algorithms. Hardly anywhere else can a natural, concise, and often self-explanatory solution be made more obscure and mystifying than by replacing its recursive formulation by one in terms of repetition and — well — jumps. This process should definitely be left to a compiler, as it concerns what is called *coding* rather than programming (code = system of symbols used in ciphers, secret messages, etc. [Webster]). Modern programming systems, however, offer efficient implementations of recursion, and thereby make "programming around recursion" a largely unnecessary exercise.

Whereas a teacher should not and must not pay attention to "percent issues" as to efficiency while explaining and exemplifying methods of composing well-structured programs, a professional programmer may well be forced to do so. He may sometimes find a dogma of sticking exclusively to a restricted set of program structuring schemas too much of a straight-jacket, and the temptation to break out too powerful. This will be the case as long as compilers are insufficiently sophisticated to take full advantage of disciplined structuring. Naturally, there will always be situations where a compiler is either denied the full information needed for successful code optimization, or where it would be unable to infer the necessary conditions. It is therefore entirely possible that in the future a more interactive mode of operation between compiler and programmer will emerge, at least for the very sophisticated professional. The purpose of this interaction would not, however, be the development of an algorithm or the debugging of a program, but rather its *improvement under invariance of correctness*.

The foregoing discussion also implies an answer to the question of whether structured programming in an unstructured language (such as FORTRAN) is possible. It is not. What is possible, however, is structured programming in a "higher level" language and subsequent hand-translation into the unstructured language. The corollary is that whereas this approach may be practicable with the almost superhuman discipline of a compiler, it is highly unsuited for *teaching* programming. Recognizing that there may be valid economic reasons for learning *coding* in, say, FORTRAN, the use of an unstructured language 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.

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 program designs, and managing large programming projects can all be greatly simplified. And, as the module library grows, increasingly sophisticated programs can be implemented using less and less new code.

Structured design

by W. P. Stevens, G. J. Myers, and L. L. Constantine

Structured design is a set of proposed general program design considerations and techniques for making coding, debugging, and modification easier, faster, and less expensive by reducing complexity.[1] The major ideas are the result of nearly ten years of research by Mr. Constantine.[2] His results are presented here, but the authors do not intend to present the theory and derivation of the results in this paper. These ideas have been called *composite design* by Mr. Myers.[3,4] The authors believe these program *design* techniques are compatible with, and enhance, the *documentation* techniques of HIPO[5] and the *coding* techniques of structured programming.[6]

These cost-saving techniques always need to be balanced with other constraints on the system. But the ability to produce simple, changeable programs will become increasingly important as the cost of the programmer's time continues to rise.

General considerations of structured design

Simplicity is the primary measurement recommended for evaluating alternative designs relative to reduced debugging and modification time. Simplicity can be enhanced by dividing the system into separate pieces in such a way that pieces can be considered, implemented, fixed, and changed with minimal consideration or effect on the other pieces of the system. Observability (the ability to easily perceive how and why actions occur) is another useful consideration that can help in designing programs that can be changed easily. Consideration of the effect of reasonable changes is also valuable for evaluating alternative designs.

Mr. Constantine has observed that programs that were the easiest to implement and change were those composed of simple, independent modules. The reason for this is that problem solving is faster and easier when the problem can be subdivided into pieces which can be considered separately. Problem solving is hardest when all aspects of the problem must be considered simultaneously.

The term *module* is used to refer to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of variable names. Examples of modules are PL/I procedures, Fortran mainlines and subprograms, and, in general, subroutines of all types. Considerations are always with relation to the program statements as *coded*, since it is the programmer's ability to understand and change the *source* program that is under consideration.

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*,[7] 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 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 for 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.

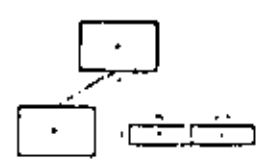
The degree of coupling established by a particular connection is a function of several factors, and thus it is difficult to establish a simple index of coupling. Coupling depends (1) on how complicated the connection is, (2) on whether the connection refers to the module itself or something inside it, 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 I.

Table I. Considering factors

	Complexity of connection	Type of connection	Type of coupling
low	simple, obvious	to module by name	data
COUPLING			control
high	complicated, obscure	to internal elements	hybrid

Figure 1. A structure chart



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 the 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 session

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 XYZ1 and XYZ2 might actually make use of data element A in an "included" common environment of PAY, yet changing the length of A 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,[7] 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 $M(M-1)/2$ or $(M^2-M)/2$ unordered 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 $N(M^2-M)/2$ 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" commons in FORTRAN is one means of accomplishing this goal.

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 fewer coupling than obscure or internal ones. The more symbolic units (such as parameters) in the statement of a connection, the higher the coupling. Thus, extraneous elements irrelevant to the programmer's and the modules' immediate task increase coupling unnecessarily.

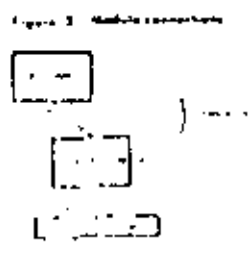
Connections that address or refer to a module as a whole by its name (leaving its contents unknown and irrelevant) yield lower coupling than connections referring to the internal elements of another module. In the latter case, as for example the use of a variable by direct reference from within some other module, the entire content of that module may have to be taken into account to correct an error or make a change so that it does not make an impact in some unexpected way. Modules that can be used easily without knowing anything about their insides make for simpler systems.

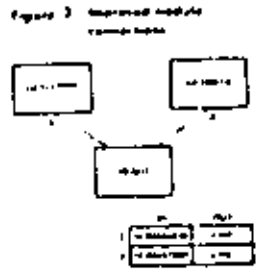
Consider the case depicted in Figure 2. GETCOMM is a module whose function is getting the next command from a terminal. In performing this function, GETCOMM calls the module READT, whose function is to read a line from the terminal. READT requires the address of the terminal. It gets this via an externally 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.

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 (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 1. This simple example shows how references to internal elements of other modules can have an adverse effect on program modification, both in terms of cost and potential bugs.

Modules must at least pass data or they cannot functionally be a part of a single system. Thus connections that pass data are a necessary minimum. (Not so the communication of control. In principle, the presence or absence of requisite input data is sufficient to define the circumstances under which a module should be activated, that is, receive control. Thus the explicit passing of control by one module to another constitutes an additional, theoretically inessential form of coupling. In practice, systems that are *purely* data-coupled require special language and operating system support but have numerous attractions, not the least of which is they can be fundamentally simpler than any equivalent system with control coupling.")

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 passed through which EXECUTECMM instructs GETCMM whether to return a parsed or unparsed command. Separating the two distinct functions of GETCMM results in a structure that is simpler as shown in Figure 5.

The new EXECUTECMM is no more complicated, where once it set a switch and called, now it has two alternate calls. The sum of GETPCMM and GETUCMM is (functionally) less complicated than GETCMM was (by the amount of the switch testing). And the two small modules are likely to be easier to comprehend than the one large one. Admittedly, the immediate gains here may appear marginal, but they rise with time and the number of alternatives in the switch and the number of levels over which it is passed. Control coupling, where a called module "tells" its caller what to do, is a more severe form of coupling.

Modification of one module's code by another module may be thought of as a hybrid of data and control elements since the code is dealt with as data by the modifying module, while it acts as control to the modified module. The target module is very dependent in its behavior on the modifying module, and the latter is intimately involved in the other's internal functioning.

Cohesiveness

Coupling is reduced when the relationships among elements *not* in the same module are minimized. There are two ways of achieving this – minimizing the relationships among modules and maximizing relationships among elements in the same module. In practice, both ways are used.

The second method is the subject of this section. "Element" in this sense means any form of a "piece" of the module, such as a statement, a segment, or a "subfunction". Binding is the measure of the cohesiveness of a module. The objective here is to reduce coupling by striving for high binding. The scale of cohesiveness, from lowest to highest, follows:

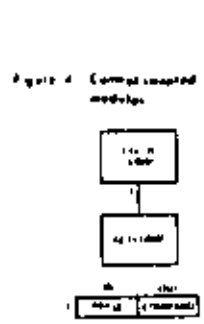
1. Coincidental
2. Logical
3. Temporal
4. Communicational
5. Sequential
6. Functional

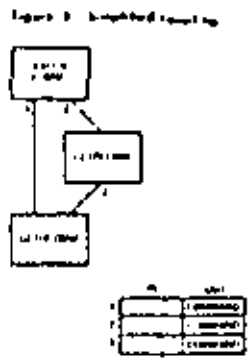
The scale is not linear. Functional binding is much stronger than all the rest, and the first two are much weaker than all the rest. Also, higher-level binding classifications often include all the characteristics of one or more classifications below it *plus* additional relationships. The binding between two elements is the highest classification that applies. We will define each type of binding, give an example, and try to indicate why it is found at its particular position on the scale.

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.

[illegible] example of the difficulty that can result from coincidental [illegible] sequence of instructions appears in several modules and was [illegible]

[illegible] coincidentally bound since [illegible]





other. Suppose in the future we have a need in one of the modules originally containing these instructions to say GETTAPEHEADER instead of GETCARD. 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 *probabilities* of this happening is very high if the module is coincidentally bound.

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 EDITALLDATA 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 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 (e.g., at initialization time).

Modules with temporal binding tend to exhibit the disadvantages of logically bound modules. However, temporally bound modules are higher on the scale since they tend to be simpler for the reason that all of the elements are executable at one time (i.e., no parameters and logic to determine which element to execute).

A module with communicational binding has elements that are related by a reference to the same set of input and/or output data. For example, "print and punch the output file" is communicationally bound. Communicational binding is higher on the scale than temporal binding since the elements of a module with communicational binding have the stronger bond of referring to the same data.

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 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.

Predictable modules

A predictable, or well-behaved, module is one that, when given the identical inputs, operates identically each time it is called. Also, a well-behaved module operates independently of its environment.

To show that dependable (free from errors) modules can still be unpredictable, consider an oscillator module that returns zero and one alternately and dependably when it is called. It might be used to facilitate double buffering. Should it have multiple users, each would be required to call it an even number of times before relinquishing control. Should any of the users have an error that prevented an even number of calls, all other users will fail. The operation of the module given the same inputs is not constant, resulting in the module not being predictable even though error-free. Modules that keep track of their own state are usually not predictable, even when error-free.

This characteristic of predictability that can be designed into modules is what we might loosely call "black-boxness." That is, the user can understand what the module does and use it without knowing what is inside it. Module "black-boxness" can even be enhanced by merely adding comments that make the module's function and use clear. Also, a descriptive name and a well-defined and visible interface enhances a module's usability and thus makes it more of a black box.

Tradeoffs to structured design

The overhead involved in writing many simple modules is in the execution time and memory space used by a particular language to effect the call. The designer should realize the adverse effect on maintenance and debugging that may result from striving just for minimum execution time and/or memory. He should also remember that programmer cost, is, or is rapidly becoming, the major cost of a programming system and that much of the maintenance will be in the future when the trend will be even more prominent. However, depending on the actual overhead of the language being used, it is very possible that a structured design can result in less execution and/or memory overhead rather than more due to the following considerations:

For memory overhead

1. Optional (error) modules may never be called into memory.
2. Structured design reduces duplicate code and the coding necessary for implementing control switches, thus reducing the amount of programmer-generated code.
3. Overlay structuring can be based on actual operating characteristics obtained by running and observing the program.
4. Having many single-function modules allows more flexible, and precise, grouping, possibly resulting in less memory needed at any one time under overlay or virtual storage constraints.

For execution overhead

1. Some modules may only execute a few times.
2. Optional (error) functions may never be called resulting in zero overhead.
3. Code for control switches is reduced or eliminated, reducing the total amount of code to be executed.
4. Heavily used linkage can be recompiled and calls replaced by branches.
5. "Includes" or "performs" can be used in place of calls. (However, the complexity of the system will increase by at least the extra consideration necessary to prevent duplicating data names and by the difficulty of creating the equivalent of call parameters for a well-defined interface.)

6. One way to get fast execution is to determine which parts of the system will be most used so all optimizing time can be spent on those parts. Implementing an initially structured design allows the testing of a working program for those critical modules (and yields a working program prior to any time spent optimizing). Those modules can then be optimized separately and reintegrated without introducing multitudes of errors into the rest of the program.

Structured design techniques

It is possible to divide the design process into general program design and detailed design as follows. General program design is deciding *what* functions are needed for the program (or programming system). Detailed design is *how* to implement the functions. The considerations above and techniques that follow result in an identification of the functions, calling parameters, and the call relationships for a structure of functionally bound, simply connected modules. The information thus generated makes it easier for each module to then be separately designed, implemented, and tested.

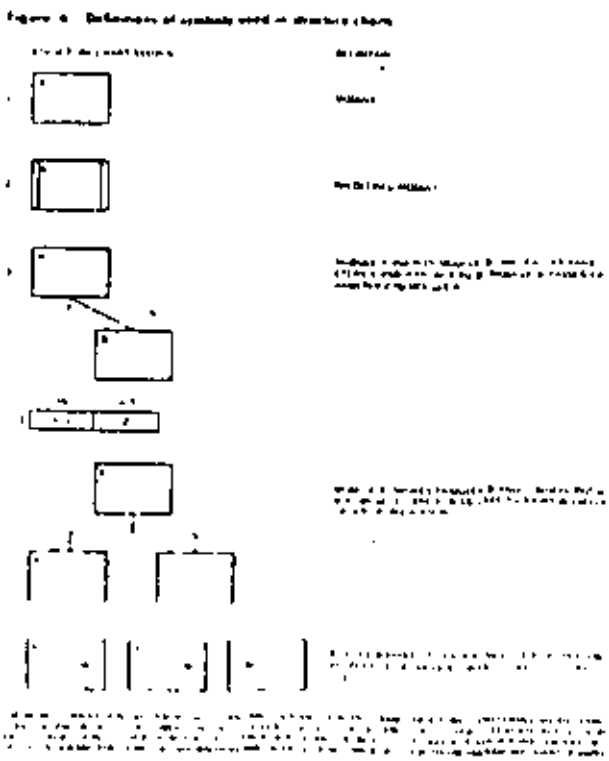
The objective of general program design is to determine what functions, calling parameters, and call relationships are needed. Since flowcharts depict *how* (in what order and under what conditions) blocks are executed, flowcharts unnecessarily complicate the general program design phase. A more useful notation is the structure chart, as described earlier and as shown in Figure 6.

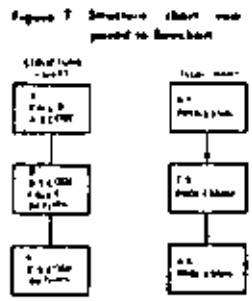
To contrast a structure chart and a flowchart, consider the following for the same three modules in Figure 7—A which calls B which calls C (coding has been added to the structure chart to enable the proper flowchart to be determined). B's code will be executed first, then C's, then A's. To design A's 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.

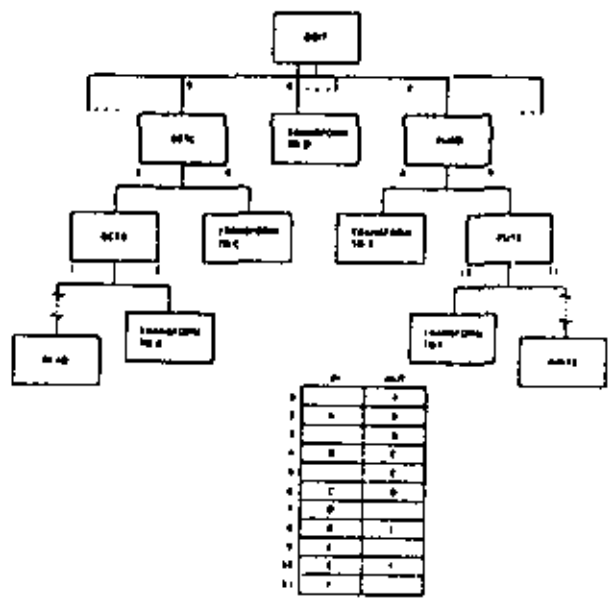
The other major difference that drastically simplifies the notation and analysis during general program design is the absence in structure charts of the decision block. Conditional calls can be so noted, but "decision designing" can be deferred until detailed module design. This is an example of where 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.

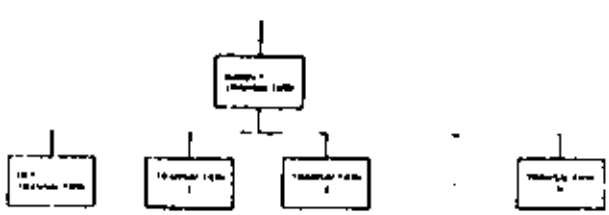
A shortcut for arriving at simple structures is to know the general form of the result. Mr. Constantine observed that programs of the general structure in Figure 8 resulted in the lowest-cost implementations. It implements the input-process-output type of program, which applies to most programs, even 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).







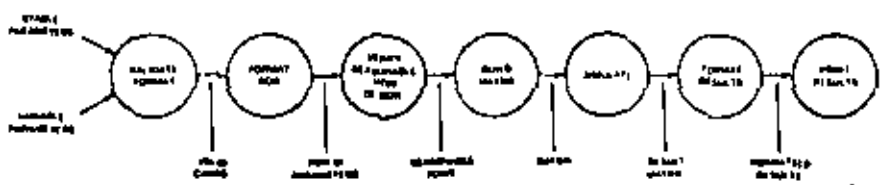


Another structure useful for implementing parts of a design is the transaction structure depicted in Figure 9. A "transaction" here is any event, record, or input, etc. for which various actions should result. For example, a command processor has this structure. The structure may occur alone or as one 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.

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.

Figure 10. Rough structure of simulation system.



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.

The "point of highest abstraction" for an input stream of data is the point in the problem structure where that data is farthest removed from its physical input form yet can still be viewed as coming in. Hence, in the simulation system, the most abstract form of the input transaction stream might be the built matrix. Similarly, identify the point where the data stream can first be viewed as going out — in the example, possibly the result matrix.

Admittedly, this is a subjective step. However, experience has shown that designers trained in the technique seldom differ by more than one or two blocks in their answers to the above.

Step Four. Design the structure in Figure 12 from the previous information with a source module for each conceptual input stream which exists at the point of most abstract input data, do sink modules similarly. Often only single source and sink branches are necessary. The parameters passed are dependent on the problem, but the general pattern is shown in Figure 12.

Describe the function of each module with a short, concise, and specific phrase. Describe what transformations occur when that module is called, not how the module is implemented. Evaluate the phrase relative to functional binding.

When module A is called, the program or system executes. Hence, the function of module A is equivalent to the problem being solved. If the problem is "write a FORTRAN compiler," then the function of module A is "compile FORTRAN program."

Module B's 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 complete, natural 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 functions may not be identified if the dividing is too conservative at this point.

Figure 11. Determining points of highest abstraction.

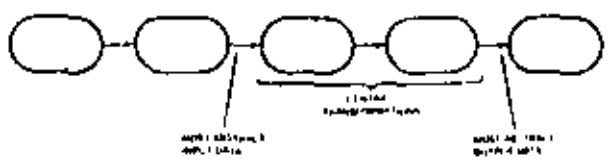


Figure 12. The top level.

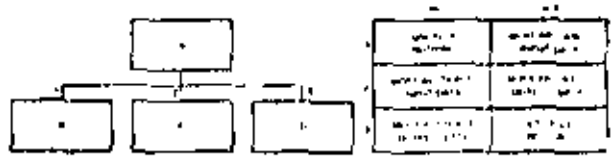
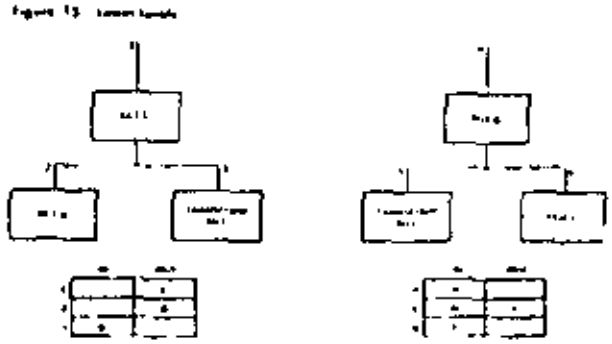


Figure 13. Lower levels.



Design guidelines

The following concepts are useful for achieving simple designs and for improving the "first-pass" structures.

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.

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.

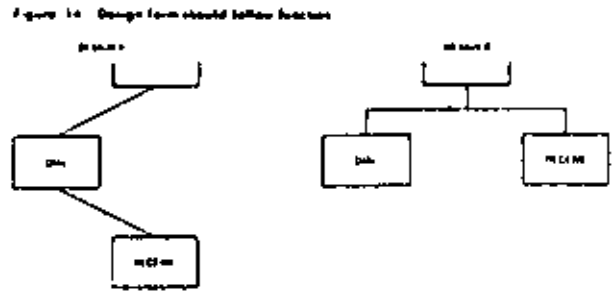
If the execution of some code in A is dependent on the outcome of decision X in module B, then either B will have to return a flag to A or the decision will have to be repeated in A. The former approach results in added coding to implement the flag, and the latter results in some of B's function (decision X) in module A. Duplicates of decision X result in difficulties coordinating changes to both copies whenever decision X must be changed.

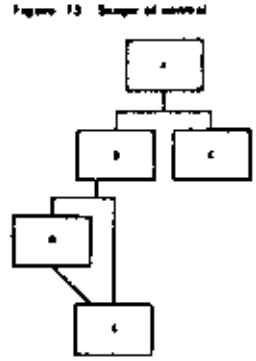
The scope of effect can be brought within the scope of control either by moving the decision element "up" in the structure, or by taking those modules that are in the scope of effect but not in the scope of control and moving them so that they fall within the scope of control.

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."

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 initializer 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 box" rule: 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.

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 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.

Look for ways to reduce the number of parameters passed between modules. Count every item passed as a separate parameter for this objective (independent of how it will be implemented). Do not pass whole records from module to module, but pass only the field or fields necessary for each module to accomplish its function. Otherwise, all modules will have to change if one field expands, rather than only those which directly used that field. Passing only the data being processed by the program system with necessary error and EOF parameters is the ultimate objective. Check binary switches for indications of scope-of-effect/scope-of-control inversions.

Have the designers work together and with the complete structure chart. If branches of the chart are worked on separately, common modules may be missed and incompatibilities result from design decisions made while only considering one branch.

An example

The following example illustrates the use of structured design.

A patient-monitoring program is required for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance.

The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a data base. For each patient, safe ranges for each factor are specified (e.g., patient "X"'s valid temperature range is 98 to 99.5 degrees Fahrenheit). If a factor falls outside of a patient's safe range, or an analog device fails, the nurse's station is notified.

In a real-life case, the problem statement would contain much more detail. However, this one is of sufficient detail to allow us to design the structure of the program.

The first step is to outline the structure of the problem as shown in Figure 16. In the second step, we identify the external conceptual streams of data. In this case, two streams are present, factors from the analog device and warnings to the nurse. These also represent the major input and output streams.

Figure 17 indicates the point of highest abstraction of the input stream, which is the point at which a patient's factors are in the form to store in the data base. The point of highest abstraction of the output stream is a list of unsafe factors (if any). We can now begin to design the program's structure as in Figure 18.

In 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 16 Outline of problem structure

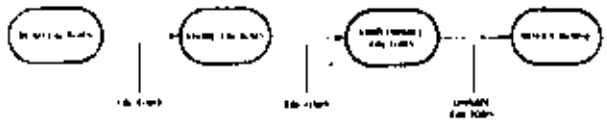


Figure 17 Points of highest abstraction

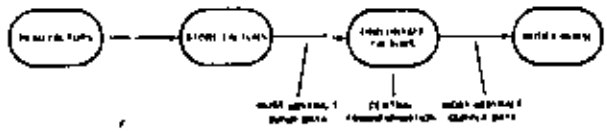


Figure 18 Structure of the top level

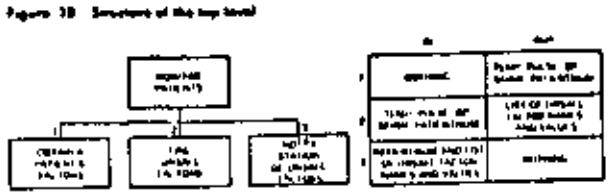


Figure 19 Structure of next level

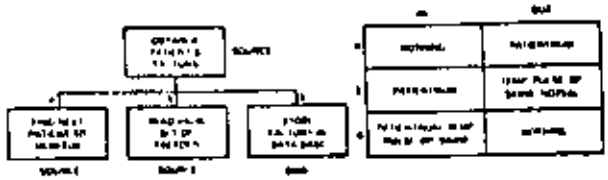
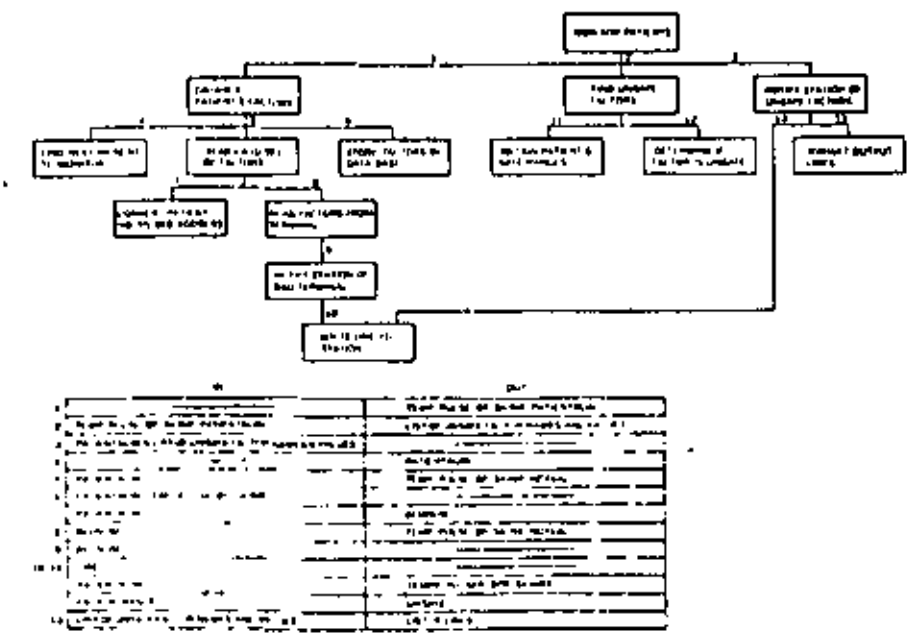


Figure 20 Complete structure chart



control. To correct this problem, we have to take two steps more: we will move the decision up to "READ VALID SET OF FACTORS." We do this by merging "READ FACTORS (and DISPLAY)" into its calling module. We now make "DISPLAY FACTORS (IF SET IS INVALID)" 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 its associated passed parameters. The resulting design can be documented with the HIPO charts. (If the designer decides to have more than one function in any module, the structure chart should show them in the same block. However, the HIPO Hierarchy chart would still show all the functions in separate blocks.) The output of the general program design is the input for the detailed module design. The HIPO input-process-output chart is useful for describing and designing each module.

Structured design considerations could be used to review program designs in a walk-through environment.[11] These concepts are also useful for evaluating alternative ways to comply with the requirement of structured programming for one-page segments.[12]

Structured design reduces the effort needed to fix and modify programs. If all programs were written in a form where there was one module, for example, which retrieved a record from the master file given the key, then changing operating systems, file access techniques, file blocking, or 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 USER 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 succeeding program. Once a module has been written to do a table search, anyone can use it. And, as the module library grows, less and less new code needs to be written to implement increasingly sophisticated systems.

Structured design concepts are not new. The whole assembly-line idea is one of isolating simple functions in a way that still produces a complete, complex result. Circuits are designed by connecting isolatable, functional stages together, not by designing one big, interrelated circuit. Page numbering is being increasingly sectionalized (e.g., 4-10.1) 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. [illegible], *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 TR00.2460, IBM, Poughkeepsie, New York, January 29, 1973.
4. G. J. Myers, "Characteristics of composite design," *Datamation* 19, No. 9, (Sept. 1973), 100-102 (September 1973).
5. G. J. Myers, *Reliable Software through Composite Design*, to be published Fall of 1974 by Mason and Lipscomb Publishers, New York, New York.
6. HIPO—Hierarchical Input-Process-Output documentation technique. An 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 B. Ratliff of the IBM Programming Productivity Techniques Department.
9. J. A. Belady and M. M. Lehman, *Programming System Dynamics or the Meta-dynamics of Systems in Maintenance and Growth*, RC 3546, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1971).
10. L. L. Constantine, "Control of sequence and parallelism in modular programs," *AFIPS Conference Proceedings, Spring Joint Computer Conference* 32, 409 (1968).
11. G. M. Weinberg, *PL/I Programming: A Manual of Style*, McGraw-Hill, New York, New York (1970).
12. *Improved Programming Technologies*, Management Overview, IBM Corporation, Data Processing Division, White Plains, New York (August 1973).

Figure 21 [illegible]

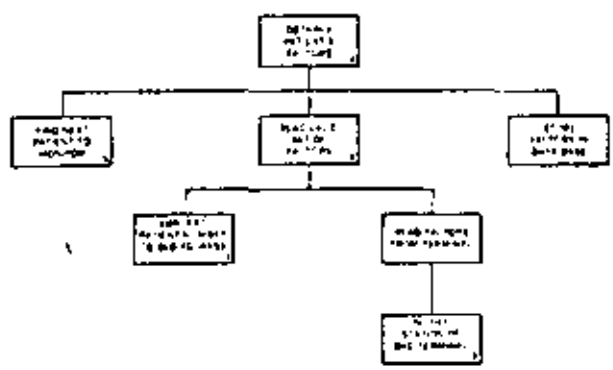
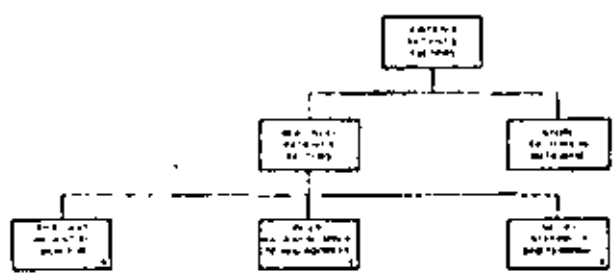


Figure 22 [illegible]



THIS PAGE INTENTIONALLY LEFT BLANK.

INSTITUTO DE INVESTIGACIONES ELECTRICAS
DEPARTAMENTO DE ANALISIS DE REDES
CONTROL DE CALIDAD

CONVENCION EN EL USO DE CUENTAS: NOMBRES DE ARCHIVOS, VARIABLES Y SUBRRUTINAS: Y POLITICA DE RESPALDO PARA LA PROGRAMACION DEL DEPARTAMENTO DE ANALISIS DE REDES EN LA COMPUTADORA HARRIS.

POR: MARCIAL PORTILLA R.
Y VICTOR M. VALADEZ C.

INDICE

1. OBJETIVO

EL PROPOSITO DE ESTE DOCUMENTO ES EL PRESENTAR AL PERSONAL DEL DEPARTAMENTO DE ANALISIS DE REDES (DAR), DEL INSTITUTO DE INVESTIGACIONES ELECTRICAS (IIE), UNA CONVENCION PARA NOMBRAR ARCHIVOS Y VARIABLES EXTERNAS, DE ACUERDO A LOS PROCEDIMIENTOS SUGERIDOS POR LA BIBLIOTECA DE SOPORTE DE PROGRAMAS DEL DEPARTAMENTO, ASI COMO EL DAR A CONOCER LA ASIGNACION DE CLAVES PARA USUARIOS DE LA COMPUTADORA HARRIS DEL CENTRO NACIONAL DE CONTROL (CENACE) Y LA POLITICA PARA EL RESPALDO DE ARCHIVOS DEL DEPARTAMENTO EN ESTA COMPUTADORA.

2. INTRODUCCION

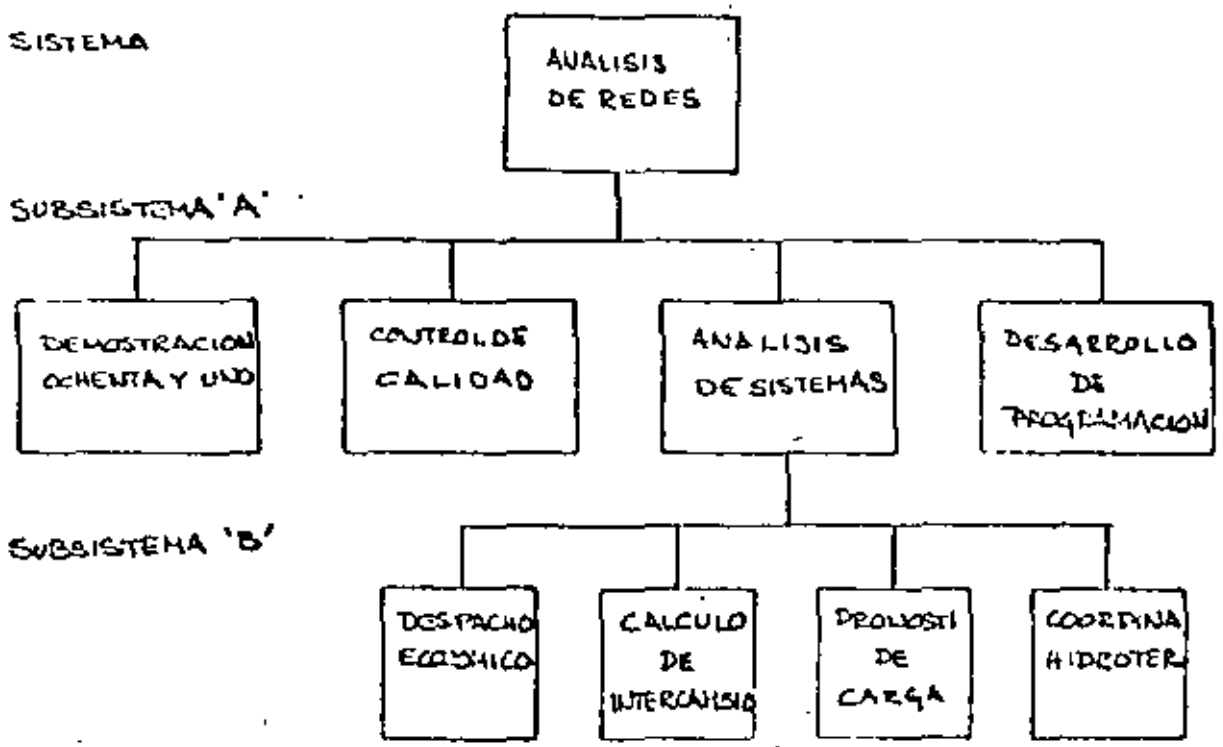
DEBIDO A LA GRAN CANTIDAD DE PROGRAMACION QUE EL IIE DESARROLLA Y DESARROLLARA EN COMPUTADORA Y TOMANDO EN CONSIDERACION LA ORGANIZACION DEL DEPARTAMENTO, SE HA ACORDADO ADOPTAR MEDIDAS QUE PERMITITAN LLEVAR UN CONTROL DE TAL PROGRAMACION Y EN ESPECIAL DE QQUELLA QUE SE EFECTUARA EN LA COMPUTADORA HARRIS.

LAS CONVENCIONES AQUI PRESENTADAS TIENEN TAMBIEN COMO FIN EL ASEGURAR QUE LOS PROGRAMAS Y VARIABLES EXTERNAS SEAN UNICAS ENTRE LOS PROGRAMAS Y FACILES DE IDENTIFICAR, CORREGIR O AGREGAR.

3. ESQUEMA DE ORGANIZACION

A CONTINUACION SE DESCRIBE LA FORMA EN LA CUAL ESTA ORGANIZADO EL DAR Y LAS CONVENCIONES ANTES MENCIONADAS.

EN EL SIGUIENTE ESQUEMA, SE MUESTRA LA ORGANIZACION DEL DAR.



EN ESTA SE DISTINGUEN CUATRO GRUPOS A NIVEL DE SUBSISTEMA A Y CUATRO A NIVEL SUBSISTEMA B.

3.1 CUENTAS PARA USUARIOS DE LA COMPUTADORA HARRIS

APROVECHANDO LAS CARACTERISTICAS DE LAS CUENTAS EN LA COMPUTADORA HARRIS COMPUESTAS DE LA SIGUIENTE FORMA:

CALIFICADOR USUARIO

SE HA ASIGNADO UN CALIFICADOR A CADA GRUPO DEL NIVEL DE SUBSISTEMA A DE LA SIGUIENTE FORMA:

SUBSISTEMA A:	CALIFICADOR
DEMOSTRACION OCHENTA Y UNO	0010DEMO
CONTROL DE CALIDAD	0020COCA
ANALISIS DE SISTEMAS	(VER *)
DESARROLLO DE PROGRAMACION	0040DEPO

(*) AL GRUPO DEL SUBSISTEMA DE ANALISIS DE SISTEMA SE LE HAN - ASIGNADO CUATRO CALIFICADORES DE ACUERDO A CADA GRUPO DEL NIVEL DE SUBSISTEMA B,DE LA SIGUIENTE FORMA:

SUBSISTEMA B	CALIFICADOR
DESPACHO ECONOMICO	0031DEAS
CALCULO DE INTERCAMBIOS	0032CIAS
PRONOSTICO DE CARGA	0033PCAS
COORDINACION HIDROTERMICA	0034CHAS

A CADA USUARIO SE LE HA ASIGNADO UN NUMERO SEGUN EL CALIFICADOR DEL SUBSISTEMA EN EL QUE COLABORA, ESTE LE DARA EXCLUSIVIDAD DE ACCESO A SUS ARCHIVOS SI ASI LO DESEA (SECCION 3.2)..EL NUMERO DE CADA USUARIO SE LE DARA A CONOCER EN FORMA PERSONAL.

NOTA: DADO LAS CARACTERISTICAS DEL SISTEMA OPERATIVO DE LA COMPUTADORA HARRIS, LOS USUARIOS QUE TENGAN EL MISMO CALIFICADOR NO PUEDEN GENERAR ARCHIVOS DIFERENTES CON EL MISMO NOMBRE ESTA RESTRICCION SE SALVARA, INCLUYENDO EN EL NOMBRE DE LOS ARCHIVOS, DOS INICIALES DEL NOMBRE DEL USUARIO A QUIEN PERTENECE (VER SECCION 3.2).

3.2 NOMBRAMIENTO DE ARCHIVOS

CON EL OBJETO DE QUE LOS NOMBRES DE LOS ARVHIVOS GENERADOS EN - LA COMPUTADORA HARRIS SEAN UNICOS, IDENTIFIQUEN A SU PROPIETA-- RIO Y DEN IDEA DE SU CONTENIDO, SE HA CONVENIDO EL USAR EL SI-- GUIENTE FORMATO:

```
TIPO INICIALES CONTENIDO
**** --------- *********
```

DONDE, EL TIPO CONSISTE EN UN CARACTER QUE DESCRIBE EL TIPO DE ARCHIVO DE QUE SE TRATA SEGUN LOS SIGUIENTES POSIBLES:

- F = PROGRAMA FUENTE (LENGUAJE FORTAN, COBOL, ETC.)
- R = MODULO RELOCALIZABLE (RESULTANTE DE UNA COMPILACION)
- M = MODULO EJECUTABLE (RESULTADO DE UN ENCADENAMIENTO)
- C = ARCHIVO DE COMANDOS INDIRECTOS
- E = PROGRAMA EN LENGUAJE ENSAMBLADOR
- M = PROGRAMA MACRO-ENSAMBLADOR
- T = ARCHIVO CON UN TEXTO (DOCUMENTO)
- D = ARCHIVO CON DATOS
- R = ARCHIVO CON RESULTADOS

LAS INICIALES CONSISTEN EN DOS CARACTERES QUE SON LA PRIMERA - LETRA DEL PRIMER NOMBRE Y DEL APELLIDO PATERNO DEL USUARIO QUE GENERA EL ARCHIVO (AREA).

EL CONTENIDO CONSISTE EN HASTA CINCO CARECTERES ALFANUMERICOS QUE DAN IDEA DE LA FUNCION, OBJETO O PROPOSITO DEL CONTENIDO DE EL ARCHIVO

EJEMPLO:

NOMBRE	DESCRIPCION
TMPCOPRE	ARCHIVO QUE CONTIENE UN TEXTO (DOCUMENTO)
*--*****	PERTENECE A MARCIAL PORTILLA ROBERTSON
	Y CONTIENE UN CONTROL DE PRESTAMOS

3.3 NOMBRAMIENTO DE VARIABLES EXTERNAS

CON EL OBJETO DE PODER IDENTIFICAR EN FORMA UNICA Y A LA VEZ, EL QUE SEA REPRESENTATIVO EL NOMBRE DE LAS VARIABLES EXTERNAS QUE SE USEN EN LA PROGRAMACION DEL DAR, SE EMPLEARA LA SIGUIENTE CONVENCION EN SU FORMATO:

FUENTE.MNEMONICO.DESITINO

DONDE, LA FUENTE Y DESTINO CONSISTEN DE DOS CARACTERES LOS CUALES REPRESENTAN EL SUBSISTEMA QUE GENERA LA VARIABLE Y EL QUE LA RECIBE RESPECTIVAMENTE, DENTRO DE LOS SIGUIENTES IDENTIFICADOS:

DE = DESPACHO ECONOMICO
CH = COORDINACION HIDROTERMICA
PC = PRONOSTICO DE CARGA
CI = CALCULO DE INTERCAMBIOS
ON = OPERADOR NACIONAL
OA = OPERADOR DE AREA

EL MNEMONICO ES UN CONJUNTO DE HASTA DIEZ CARECTERES ALFANUMERICOS, QUE IDENTIFICAN LO QUE REPRESENTA LA VARIABLE.
LOS PUNTOS SON SEPARADORES DE LA FUENTE Y EL DESTINO DEL MNEMONICO.

EJEMPLO:

CH.TMVIA.ON = VARIABLE GENERADA POR EL SUBSISTEMA CH Y RECIBIDA POR EL OPERADOR NACIONAL Y QUE REPRESENTA UN 'TIEMPO DE VIAJE!;

3.4 POLITICA DE RESPALDO DE ARCHIVOS EN LA COMPUTADORA HARRIS

CON EL OBJETO DE CONTAR CON RESPALDO DE LOS ARCHIVOS DEL IIE - EXISTENTES EN LA COMPUTADORA HARRIS, SE EFECTUARAN EN FORMA PERIODICA EL COPIADO A CINTA DE ESTOS, DE LA SIGUIENTE FORMA:

SE EFECTUARA RESPALDO DIARIO CON CINCO CINTAS, RESPALDO SEMANAL CON TRES CINTAS Y RESPALDO MENSUAL CON DOS CINTAS.

ESTO PERMITIRA, EN EL MEJOR DE LOS CASOS TENER UNA HISTORIA DE VERSIONES DE UN ARCHIVO CON CINCO VERSIONES DE CINCO DIAS LABORABLES ATRAS, TRES VERSIONES CON ESPACIO DE UNA SEMANA DE TRES SEMANAS ATRAS Y DOS VERSIONES CON ESPACIO DE UN MES DE DOS -- MESES ATRAS.

ESTOS RESPALDOS PERMITIRAN RECUPERAR ARCHIVOS CON VERSIONES ANTERIORES.

3.5 POLITICA DE PURGA Y CONTROL DE ARCHIVOS EN DISCO

DEBIDO A QUE EL DISCO INSTALADO EN LA COMPUTADORA HARRIS TIENE CAPACIDAD LIMITADA, SE SEGUIRA LA SIGUIENTE POLITICA SOBRE EL USO DEL ESPACIO EN DISCO:

TODA NUEVA AREA GENERADA POR LOS USUARIOS DEBERA SER REPORTADA AL ENCARGADO DEL SISTEMA POR PARTE DEL IIE, LLENANDO LA FORMA QUE PARA EL EFECTO SE HA DISPUESTO: LAS AREAS NO REPORTADAS SERAN ELIMINADAS PERIODICAMENTE (PURGA DEL DISCO).

SE COPIARAN EN CINTA CON CARACTER PERMANENTE AQUELLOS ARCHIVOS QUE EL USUARIO DESEE CONSERVAR PERO NO USE EN TIEMPO INMEDIATO Y POR TANTO NO ES NECESARIO QUE ESTEN EN DISCO, DE TAL QUE EL ESPACIO EN DISCO QUE OCUPE NO REPASE DE DIEZ MIL SECTORES -- (1120000 PALABRAS) DENTRO DE LO POSIBLE).

ESTAS MEDIDAS DEBERAN SER SEGUIDAS POR LOS USUARIOS DEL IIE - PARA UN MEJOR USO DEL UNICO DISCO QUE ACTUALMENTE TIENE LA COMPUTADORA HARRIS DEL CENACE.

NORMAS DEL "LENGUAJE PARA DISEÑAR PROGRAMAS"

PDL

MARCIAL PORTILLA ROBERTSON
ROBERTO MANDUJANO WILD

I N D I C E

I. OBJETIVO

EL PROPOSITO DE ESTE DOCUMENTO ES DEFINIR LAS NORMAS PARA EL USO DE UN "LENGUAJE PARA DISEÑAR PROGRAMAS" (LDP).

EL LDP ES FACIL DE SEGUIR Y MAS FACIL DE PROGRAMAR QUE LOS DIAGRAMAS DE FLUJOS, DEBIDO A QUE "ES CASI CODIGO". ESTE "PSEUDO CODIGO" DESCRIBE LA ESTRUCTURA, CONTROL, VIBRACIONES, FLUJO, ETC, DE UN PROGRAMA DE COMPUTADORA, CON LA VENTAJA SOBRE LOS DIAGRAMAS DE FLUJO Y OTROS TIPOS DE DIAGRAMAS, QUE PUEDE ACTUALIZARSE CONTINUAMENTE EN UN PROCESADOR DE TEXTOS.

EL LDP DEBERA INCLUIRSE EN LOS SIGUIENTES DOCUMENTOS:

RPC
DPC
CODIGO

II INDENTACION Y ETIQUETAS

INDENTACION Y ETIQUETAS.-LA SUBORDIDACION SE LLEVA A CABO INDENTANDO LAS LINEAS DEL PSEUDO-CODIGO

EJEMPLO

```
1.  LINEA A
2.  LENEA B
    A. LINEA C
    B. LINEA D
       1. LINEA E
       2. LINEA F
    C. LINEA G
3.  LINEA H
    A. LINEA I
       1. LINEA J
          A. LINEA K
          B. LINEA L
4. CONTINUA CON LDP, REGRESA O TERMINA
```

SOLAMENTE SE UTILIZAN DOS CONJUNTOS DE CARACTERES: NUMEROS Y LETRAS SE UTILIZAN ALTERNATIVAMENTE EN LOS NIVELES DE SUBORDINACION.
Y LA IDENTACION SE HACE CON CUATRO ESPACIOS.

III ESTRUCTURAS DE CONTROL

LAS UNICAS ESTRUCTURAS DE CONTROL SON DE SELECCION, REPETICION Y LLAMADAS A MODULOS.

A.- Seleccion
LAS FORMAS DE SELECCION SON

- A.1 SI (IE)
- A.2 SI-ENTONCES-SINO (IE-THEN-PLSE)
- A.3 CASO (CASE)

R.-REPETICION
LAS FORMAS DE REPETICION:

- R.1 EJECUTA-ITEMIRAS (AHIDE)
- R.2 EJECUTA-HASTA (DO-UNTIL)
- R.3 REPITE (BOB,DO)

C.- LLAMADAS A MODULOS
LA FORMA DE LLAMAR A MODULOS ES LLAMA (CALL).

IV CONSTRUCCIONES

A.-SELECCION

A.1 CONSTRUCCION SI (IE)

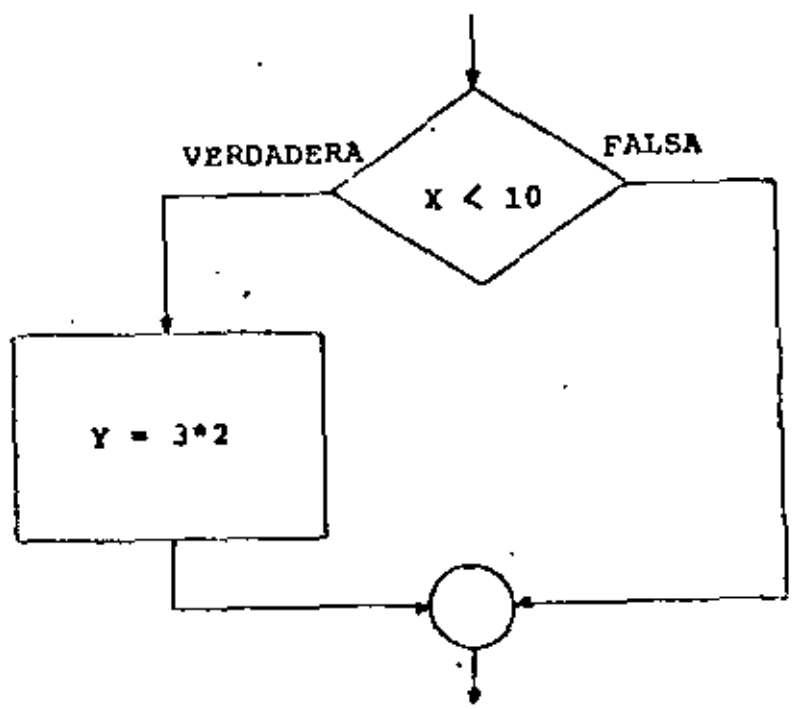
EL FORMATO DEL SI ES:

1. SI CONDICION
   . A. BLOQUE VERDADERO

DONDE CONDICION ES UNA EXPRESION LOGICA Y EL BLOQUE VERDADERO ES UN CONJUNTO DE PROPOSICIONES EN PSEUDO CODIGO.

EJEMPLO

1. SI X<10
   A. Y=3*2



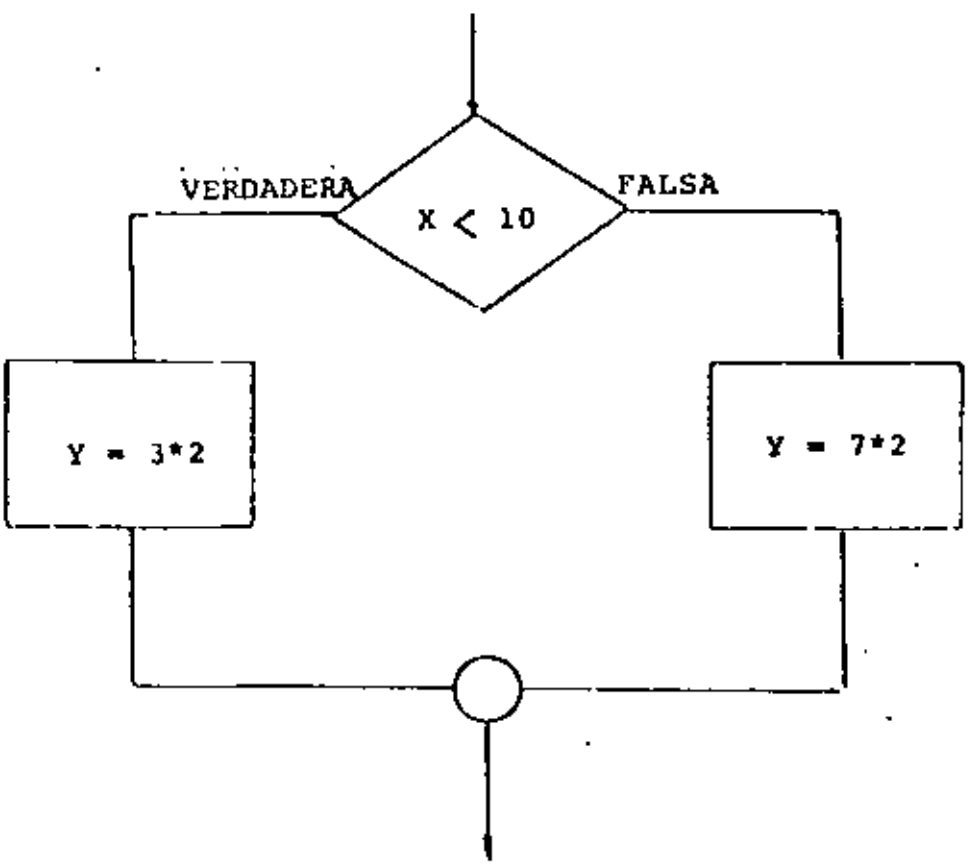
A.2 CONSTRUCCION SI-ENTONCES-SINO (IE-THEN-ELSE)
EL FORMATO DEL SI-ENTONCES-SINO ES:

1. SI CONDICION ENTONCES
   A. BLOQUE VERDADERO
2. SINO
   A. BLOQUE FALSO

DONDE CONDICION ES UNA EXPRESION LOGICA Y LOS BLOQUES VERDADERO Y FALSO SON PROPOSICIONES EN PSEUDO-CODIGO.

EJEMPLO

1. SI X<10 ENTONCES
   A. Y=3*2
2. SINO
   B. Y=7*2



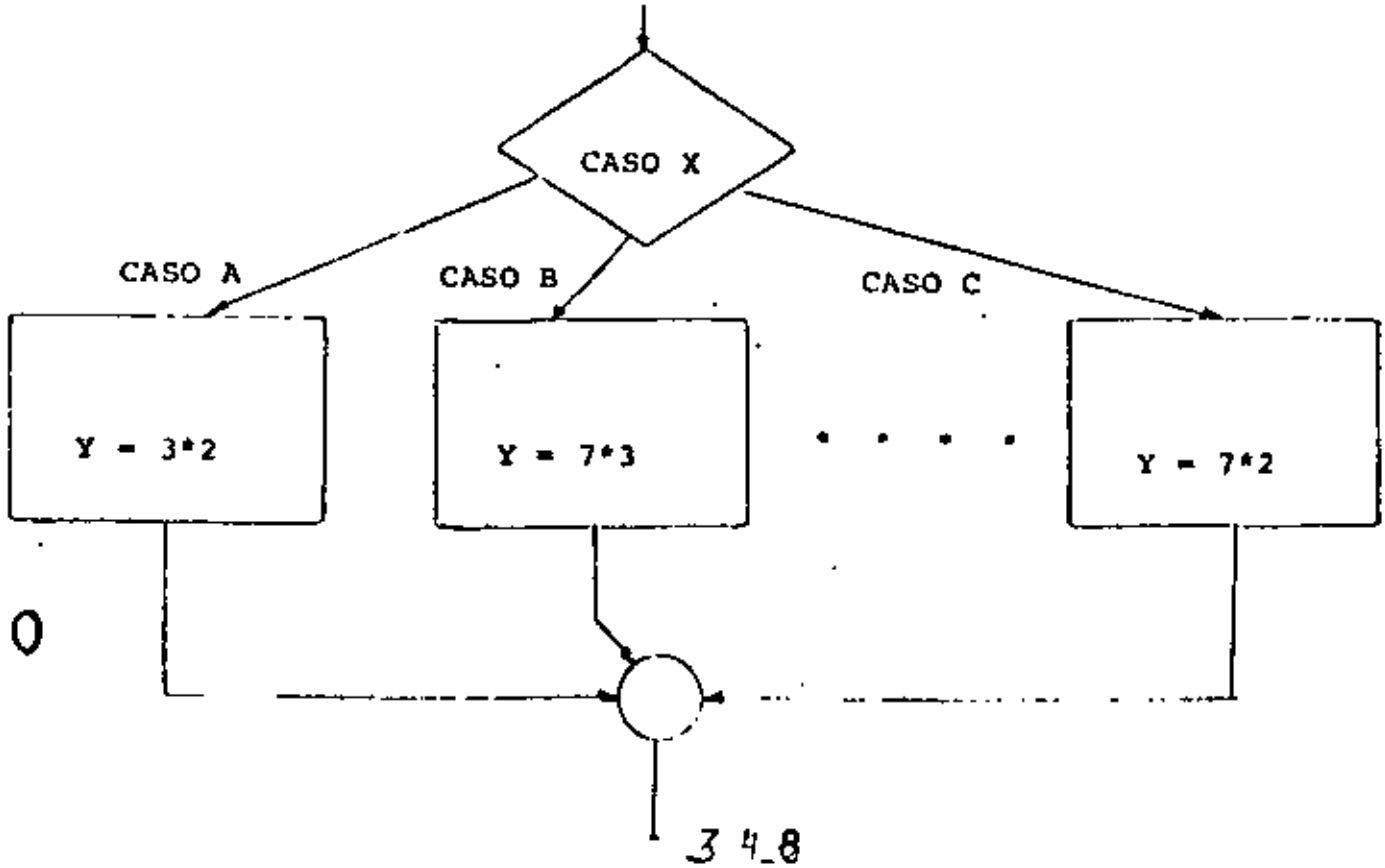
A.3 CONSTRUCCION CASO (CASE)
EL FORMATO DEL CASE ES:

1. CASO (A,B,C,...)CONDICION
   A. BLOQUE A
      1. SEPARADOR DE CASO
   B. BLOQUE B
      1. SEPARADOR DE CASO
   C. BLOQUE C
      1. SEPARADOR DE CASO

DONDE CONDICION PUEDE TOMAR VARIOS VARPRES Y LOS BLOQUES A,B, C,III SON PROPOSICIONES EN PSEUDO-CODIGO.

EJEMPLO

1. VE A (A,B,C)X
   A. Y=3*2
      1. SEPARADOR DE CASO
   B. Y=7*3
      1. SEPARADOR DE CASO
   C. Y=7*2
      1.SEPARADOR DE CASO



B.-REPETICION

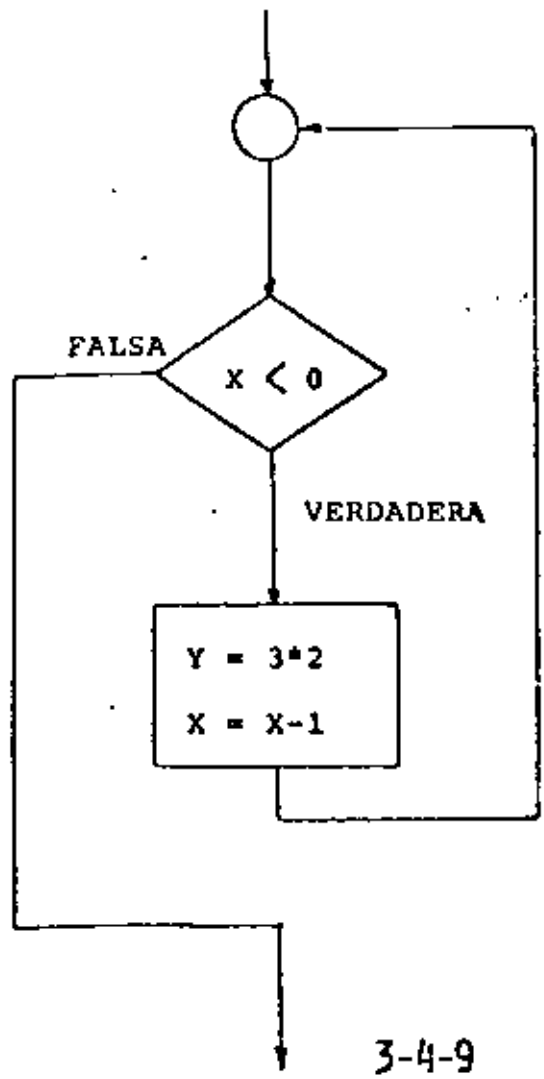
B.1 CONSTRUCCION EJECUTA MIENTRAS (WHILE)
EL FORMATO DEL MIENTRAS ES

1. EJECUTA MIENTRAS CONDICION
   A. BLOQUE

DONDE CONDICION ES UNA EXPRESION LOGICA Y EL BLOQUE A SON PROPOSICIONES EN PSEUDO-CODIGO.
EN ESTE CASO SE CHECA PRIMERO LA CONDICION, Y SI SE CUMPLE, SE EJECUTA EL BLOQUE A.

EJEMPLO

1. MIENTRAS X<10
   A. Y=3*2
   B. X=X-1



3-4-9

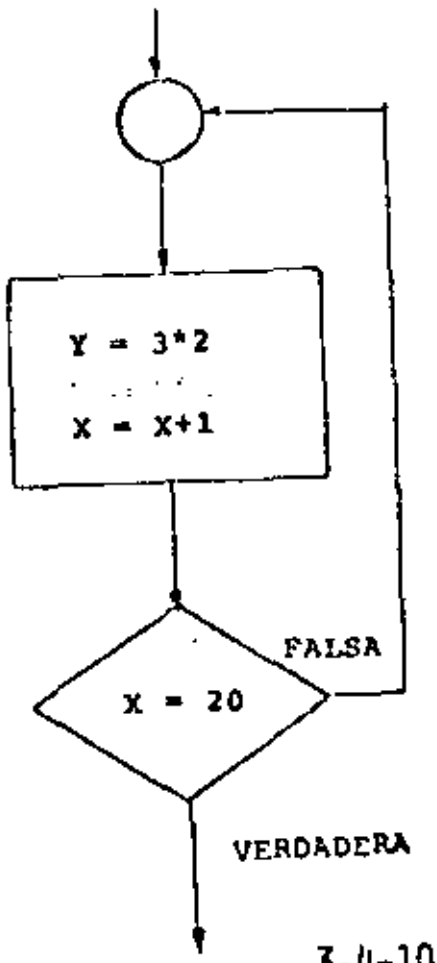
B.2 CONSTRUCCION EJECUTA-HASTA (DO-UNTIL)
EL FORMATO DEL EJECUTA-HASTA ES:

1. EJECUTA
   A. BLOQUE
2. HASTA CONDICION

DONDE CONDICION ES UNA EXPRESION LOGICA Y EL BLOQUE A SON PROPOSICIONES EN PSEUDO-CODIGO.
EN ESTE CASO SE EJECUTA EL BLOQUE A Y LUEGO SE CHECA LA CONDICION, YA SEA PARA CONTINUAR EJECUTANDO O SALIRSE DEL BLOQUE, DE TAL MANERA QUE SIEMPRE SE EJECUTA AL MENOS UNA VEZ EL BLOQUE A.

EJEMPLO

1. EJECUTA
   A. Y=3*2
   B. X=X+1
2. HASTA X=20



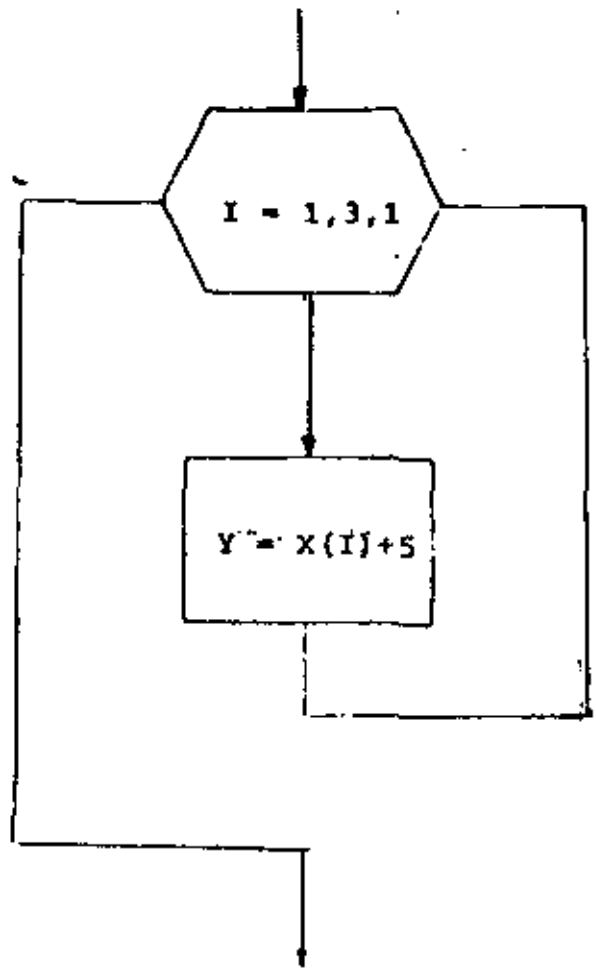
B.3 REPITE (FOR/DO)
EL FORMATO DEL REPITE (FOR) ES:

1. REPITE I=N,M,(S)
   A. BLOQUE

DONDE N Y M SON ENTEROS: N < M Y EL NUMERO DE ITERACIONES SERA IGUAL A M-N+1, S (OPCIONAL) ES EL INCREMENTO. EL BLOQUE ES UN CONJUNTO DE PROPOSICIONES EN PSEUDO-CODIGO.

EJEMPLO

1. REPITE I=1,3,1
   A. Y=X(I)+5



3-4-11

C.-LLAMADAS A MODULOS
EL FORMATO DE LLAMA (CALL) ES:

LLAMA MODULO

DONDE MODULO ES EL NOMBRE DE UNA RUTINA COMPLETA A EJECUTAR.

EJEMPLO

```
        LLAMA RUTINA 1
```

holm, Sweden, 1979.
[19] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming*. Reading, MA: Addison-Wesley, 1979.
[20] B. Liskov and V. Berzins, "An appraisal on program specifications," M.I.T. Tech. Rep., 1977.
[21] H. M. Markowitz, "SIMSCRIPT," RC 6811, IBM Res. Div., 1978.
[22] T. W. Mao and R. T. Yeh, "Communication Port: A language concept for concurrent programming," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 194-204, 1980.
[23] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 128-138, 1979.
[24] F. Parr, "Software maintenance," in *Proc. Beyond Structured Programming*, INFOTECH State of the Art Conf., 1978.
[25] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223-252, 1977.
[26] W. E. Riddle and R. E. Fairley, Eds., in Proc. Software Development Tools Workshop Conference (Pingree Park, CO), May 1979.
[27] L. Robinson and R. C. Holt, "Formal specifications for solutions to synchronization problems," Comput. Sci. Group, Stanford Res. Inst., Stanford, CA, 1975.
[28] D. T. Ross, "Structured analysis (SA): A language for communicating ideas," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 16-34, 1977.
[29] N. Roussopoulos, "A semantic network model of data bases," Ph.D. dissertation, Dep. Comput. Sci., Univ. of Toronto, Toronto, Ont., Canada, TR 104, 1976.
[30] H. Simon, *The Sciences of the Artificial*. Cambridge, MA: M.I.T. Press, 1970.
[31] J. Smith and D. Smith, "Data base abstractions: Aggregation and generalization," *Assoc. Comput. Mach. TODS*, vol. 2, pp. 105-133, 1977.
[32] D. Teichroew and E. H. Hershey, "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 41-48, 1977.
[33] R. T. Yeh, *Current Trends in Programming Methodology, Vol. 1-Software Specification and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1977.
[34] R. T. Yeh, A. Augustine, R. Mittermeir, W. Mao, and F. Evans, "Software requirement engineering: A perspective," in *Proc. Beyond Structured Programming*, INFOTECH State of the Art Conference, London, England, Nov. 1978.
[35] R. T. Yeh, N. Roussopoulos, and P. Chang, "Systematic derivation of software requirements through structured analysis," Tech. Rep. SOBEG-15, Dep. Comput. Sci., Univ. of Texas, Austin, Sept. 1979.
[36] P. Zave, "A comprehensive approach to requirements problems," in *Proc. COMPSAC '78* (Chicago, IL), pp. 117-123, Nov. 1978.
[37] P. Zave, "Formal specification of complete and consistent performance requirements," in *Proc. 8th Texas Conf. Computing Systems* (Dallas, TX), pp. 4B-18-4B-25, Nov. 1979.

Software Representation and Composition Techniques

LAWRENCE J. PETERS

Abstract—The field of software development has undergone some of its most profound changes in the last ten years. Much of this change has been in response to ever increasing demands on software systems in terms of their complexity, reliability, and resiliency. Symptomatic of such rapid evolution is the proliferation of methods and techniques intended to solve "the" software problem. However, real-world software design problems often exhibit characteristics that make them unique. This forces the software engineer to seek alternative ways of composing and documenting a design. This paper describes a sampling of the alternatives and directs the reader to sources of more detailed information regarding their use as well as a larger survey of this rapidly changing field.

I. Introduction

SOFTWARE development challenges the software engineer in several ways. Unlike many other fields, systems of software will not be mass produced. This divorces the software engineer from many problems associated with manufacturing. However, since he is dealing with logic—the abstract—the results of his labor are difficult to identify with. Software operates on a time scale and reference frame which is incomprehensible by human standards. These factors, and the lack (for the most part) of an engineering background on the part of software developers, have led to the naive view that software design is unique and that its problems are exclusively those of software. The software engineer also has the least guidance of any technical field regarding the scope of his problem or the acceptability of his solution. For example, in aircraft design, a set of strictly adhered to parameters called a design envelope are established early in the design effort. The task of the aircraft design team is to create a design which meets the requirements and falls within the envelope. Factors included in the envelope may include size, weight (dry and maximum takeoff), range, and other parameters dictated by marketing and manufacturing considerations. In such hardware-oriented efforts, design envelopes are the result of experience, common practice, economics, and production capabilities. In the case of software, a concept such as the design envelope is

Manuscript received March 18, 1980; revised May 23, 1980.
Portions of this article are excerpted from [11], and appear here with the permission of the publisher.
The author is with Yourdon, Inc., 1133 Avenue of the Americas, New York, NY 10036.

0018-9219/80/0900-1085$00.75

less straightforward—a large experience base related to a particular type of software is not available, economics is a matter of "black magic," and there is no common practice or discipline. This is particularly true in the case of software design documentation. Hence, the software designer has little or no guidance in the execution of his task(s).

Several authors have attempted to rescue the software engineer from this dilemma by publishing a wide variety of tactics and strategies with which to address software development. Each author relates the efficacy of his approach to their own experience usually in some limited area of application. Each utilizes a system of representation techniques designed to transmit the attributes of the design considered important by such authors. We will examine representative examples of these, comment on their source and utility, and finally describe current trends and what they signal for future technological developments.

II. Background

The emphasis in early software development was on obtaining a program which worked. That is, it gave answers which agreed with accepted values or, where accepted values did not exist, saved large amounts of manual labor. For example, Gustav Mie [1] solved Maxwell's equations for the case of electromagnetic waves passing through a colloidal suspension containing dielectric spheres in about 1905. In the 1930's, using comptometers and a lot of manual labor, less than a few dozen of these coefficients useful in estimating information loss in infrared, microwave, and laser transmission had been computed. By the mid-1960's, several thousand values could be computed in a matter of seconds [2]. But by the 1960's, other more challenging problems were being attempted. Instead of developing single programs or small sets of programs, large assemblages of programs were being attempted. This ushered out the age of functional programming and ushered in the age of structure oriented programming. The problems created by attempting to control satellites or air traffic with these large systems highlighted the need for some sort of philosophical viewpoint which would enable software designers to create systems (not just single programs) that worked; systems whose construction was aided by the design and not encumbered by it. During this time many concepts were "discovered" in software such as the difference between logical design (i.e., abstract, conceptual) and physical design (i.e., blueprint, one-for-one correspondence to what will be built), philosophies of design (e.g., top-down), and the use of prototypes from which to learn and protect an investment. Much of this ferment peaked in the late 1960's and early 1970's with the advent of some specific methods and approaches to the problem of software design representations. These seemed to address structure problems and some were adopted quite widely. However, they have been found to be in need of some support in order to solve "real world" problems. This has resulted in yet another wave of approaches. This body of help now available to the software engineer falls into two classes—methods (strategies, recommendations, or guidelines based on a philosophical view) and techniques (tactics or well-advised "tricks of the trade"). Both classes are described in the remainder of this paper together with a look at where much of this may be leading.

III. Representing Software Designs

One of the earliest techniques to be developed is that of software design representation. The problem is a fundamental one. That is, how does the software designer depict his design so as to communicate it to fellow workers and the customer? Architects can use sketches and the customer's everyday experiences with physical surroundings to capture the overall characteristics of a proposed building. But software is not a directly experienced product.

There are several issues related to the portrayal of a software design [3]. These can be organized into the following categories.

1) Architectural—The depiction of the relationships which exist between major system elements and between the software system and the outside world (user, terminals, etc.).

2) Structural—The depiction of relationships between distinguishable system elements. This is a static view of the system.

3) Behavioral—The depiction of interactions among system elements. This is a dynamic view of the system.

4) Informational—The conceptual organization of the data used by the system.

The sensitivity of project success to these issues has increased as the complexity of the software system being attempted has increased. The role of software representation techniques has shifted from documenting what the design contains to that of playing an active part in the evolution of the design. This evolution is the result of increased communication among the designer, user, customer, and program implementation personnel. Some design representation techniques can be used to describe system level concepts while others are at a more local level, often closely tied to a specific subset of issues. However, all are directed at reducing and controlling the amount of complexity exhibited by these abstract models of the system. We will examine some of the approaches that are available for architectural, structural, and behavioral classes of information.

A. Representing Architectural Features

Only recently has this important aspect of software design representation been addressed in the literature [4]. Perhaps this is because it is easier to address local, simple, and comprehensible aspects of an overall system than to deal with these "global" issues. The prospect can be overwhelming.

The Leighton diagram is intended to address the problem of depicting software system architecture in an easy to read and understandable format. The approach taken tends to avoid many of the problems associated with employing hierarchy diagrams while displaying the sources of inputs, processing levels, precedence relationships, and destination of outputs.

Leighton diagrams were originally developed to serve a function analogous to that of the artist's concept in engineering. This diagrammatic form is intended to satisfy its author's perception of the need to be fulfilled

1) employ some of the characteristics of treelike structures,
2) clearly depict sources of input and destinations of output;
3) be easy to present and copy using $8\frac{1}{2} \times 11$ in. form, and allow sequential presentation to avoid repetitive backtracking;
4) clearly depict the complexity associated with what is being presented and the occurrence/use of common routines;
5) permit the incorporation of textual information;
6) be effective on a broad spectrum of system types;
7) be cost effectively automatable.

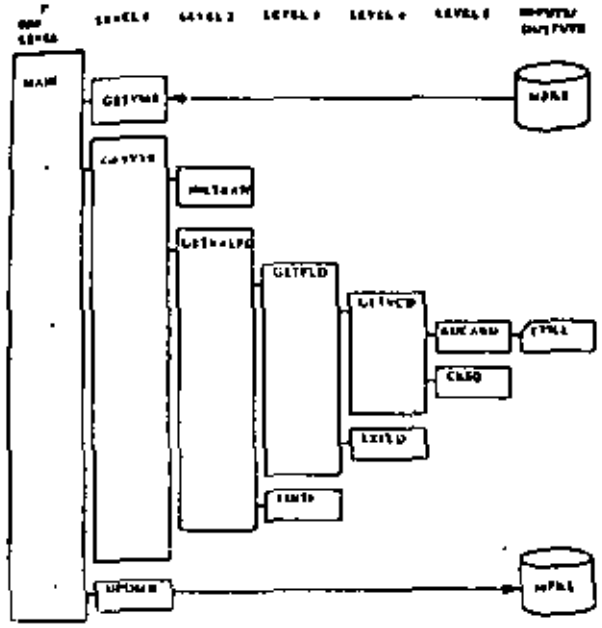


Fig. 1. An example of a Leighton diagram.

The primary goal in developing this diagrammatic technique was to enhance the communication between customers, users, software architects, and software development personnel concerning design (and development) issues.

This first of many different software design representation schemas for depicting system architecture is presented in Fig. 1. Note how the essential elements (e.g., external interfaces, interactions among major software components, and overall priority or precedence relationships) are all that is depicted. The flavor of what the system is like is retained without the introduction of unnecessary detail. In the sense that software design representation began with descriptions of individual programs and only recently turned to describing overall systems, this process has been bottom up in nature.

B. Depicting Structural Characteristics

Portraying the structure of a software system involves the taking of a "snapshot" of the relationships between internal system elements. It is a static view of what the system is like. We will first look at an approach which was common to engineering at least a couple of thousand years before it was "discovered" as part of top-down structured programming approaches. The other representation schemes (methods) in the remainder of this article utilize the design tree concept together with some other features.

1) The Design Tree: This approach [5] to depicting design structure is directed at externalizing the composition of a system. It is based on concepts in mathematics which have been used in top-down design.

Those utilizing the design tree concept employ one of two mutually exclusive concepts. The first, containment, refers to the fact that a concept includes (or contains) other concepts. For example, the concept of airplane includes the concepts of propulsion system, body, wing, and tail. The second, hierarchy, explicitly portrays concepts as being subordinate to one another. That is, not just including other lesser concepts, but being greater or higher on a scale in some way. This means of ordering systems has been used since the time of the Egyptians (and possibly earlier). It is inherent to the concept of top-down design and analysis. Thus the design tree supports top-down decomposition both conceptually and graphically (Fig. 2).

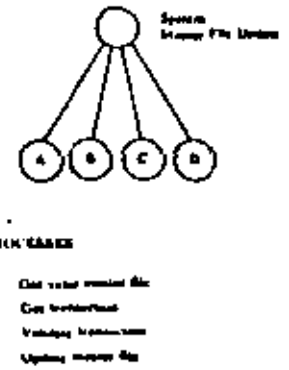


Fig. 2. An example of a design tree.

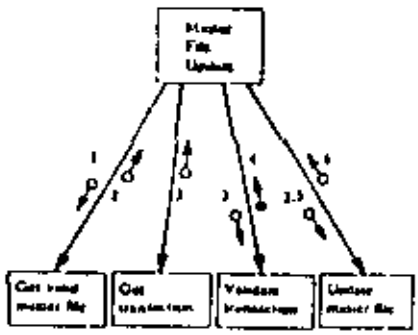


Fig. 3. A structure chart.

The design tree uses a simple flexible notational scheme. It is simple in that only two basic graphics elements are used—a straight line and a node symbol. Examples of commonly used node symbols include circles, squares, and rectangles. Each of these will contain textual information. The user of this scheme can employ any node symbol that is convenient or may choose not to enclose the text associated with each node. This scheme is also flexible in that it can be used to depict complex or simple systems to the desired level of decomposition.

2) Structure Charts: Structure charts were originally developed by Constantine *et al.* [6] to specify modular characteristics of software during design. They are an integral part of the structured design method [7], [8].

The basis for the structure chart is a treelike structure which depicts hierarchical relationships. However, the basic notation of the design tree has been enhanced to externalize module relationships. A module is defined as a set of lexically contiguous program statements which can be referred to by name. The relationships and interactions which are depicted include data flow, activation, and communication of control parameters. Note that the design tree can relate concepts or processes that comprise the system. This scheme specifically identifies modules that will comprise the software system.

Structure charts can be drawn in several different ways. The approach proposed by Constantine utilizes three basic

graphical forms

1) the rectangle, used to contain a module or module descriptor;
2) the vector, used to highlight interaction between modules (usually a call);
3) the arrow with circular tail, used to depict the transfer of data and control between modules.

Together, these comprise the structure chart as shown in Fig. 3. In this case, the system level is a master file update processor which has several subordinate functions. The master file is passed to the edit module which returns errors and valid entries and so on. Note that control elements are identified by having the circles on their tails filled in.

3) Structured Analysis and Design Technique: This approach is usually referred to as SADT©. It was originated and trademarked by SofTech Inc. [9]. It was derived from work done by Hori [10], and as a result of experience with computer-aided manufacturing studies.

This representation scheme utilizes two forms. One is a tree-like structure (a design tree) which acts as a roadmap to the system model. The other form is the activity chart. The system model can be used to describe a single program or group of programs. It is composed of one or more activity charts or more simply diagrams. The basic idea here is to provide the user of the technique with a means of graphically portraying whatever his analysis of an existing system reveals or his perception of a system under design.

This method also uses labeled rectangular boxes and arrows. veral distinctions are made both in what is represented and ow it is represented. For example, the basic distinction between data flow and activities is made but data flows are classified as being input, output, or control (Fig. 4). However, execution sequences are not explicitly shown.

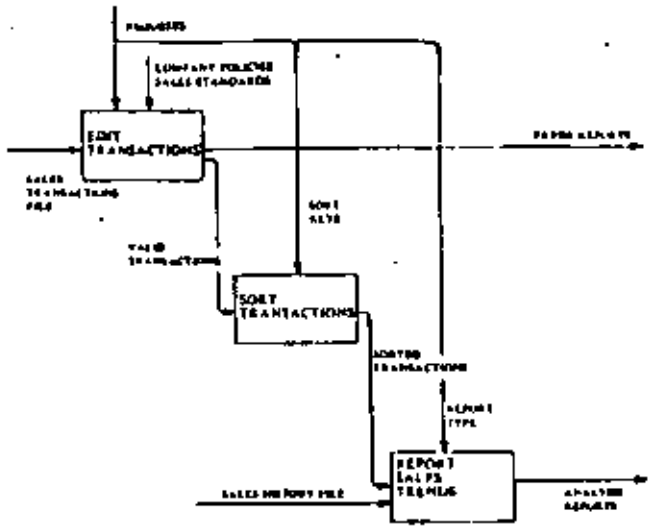


Fig. 4. An SADT© diagram.

C. Representing Behavioral Features

The issues related to the representation of the actual behavior or "execution" of the software are among those first addressed in the early days of software design. These involved the detailed and explicit documentation of precisely what the (eventual) program would do when it was constructed. One of the first of these schemes, the flowchart, typified the attitudes and concerns of the times. Namely, will the program work and if so, how?

Today, attitudes have changed. The concern is focused more on the organizational character of the program than on the details of execution. Problems with program maintenance and development have emphasized the need for some reordered priorities. The approaches presented here are a representative sampling of a much larger set of schemes from which to choose [11].

1) The Flowchart: The flowchart is probably the most widely used, misused, and misunderstood software representation schemes in use today. It has resulted in many derivatives. It was originally developed by von Neumann who intended it to be an accurate means of documenting a program after it as written not as a design representation scheme. The use of the flowchart has been somewhat standardized through the efforts of the American National Standards Institute (ANSI) [12]. What is presented here employs the ANSI format.

The ANSI flowchart depicts the details of control flow. This is accomplished by using different symbols for processes, decisions, and entry and termination points. There are n specific requirements that some specific set of constructs b employed. Hence, any type of control transfer supported by a programming language or algorithm may be represented via the flowchart.

Primarily this scheme uses three symbols: the rectangle, the rhombus (or diamond), and a directed line. Other symbols are used to depict manual operations and external/hardware interfaces. The rectangle is used to represent arithmetic operations or processes. Examples of these include "$n = n + 1$," "$x = ab$," and reset end of file flag, etc. The diamond shape is used to represent decision points. Examples of these include end of file flag set, "$n = 0$," and "$x = ab$." The directed line indicates the flow of control or precedence in a flow sequence. An example of an ANSI flowchart is presented in Fig. 5.

2) Decision Tables: The decision table has been used to analyze and describe deterministic systems and to sort out confusing decision making problems. Their use in programming is not new and dates back to the days of wired logic and telephone switching problems and beyond [12].

Decision tables can take many different forms [13]. They all stem from the same basic notion that for each of the possible combinations of situations that a system (or program) can encounter, the system's response is known. These situations are referred to as conditions while system responses are referred to as actions. For every condition or set of conditions which can occur, one and only one action or set of actions can occur. The response of the system is known with certainty.

The basic decision table consists of two portions—the condition stub and the action stub. Conditions are collected and (optionally) labeled into the condition stub while actions are collected and (optionally) labeled in the action stub. Conditions and actions are most often described horizontally. Vertical columns in the condition stub are used to identify which conditions apply in a given instance. A corresponding column in the action stub describes the system's response (Fig. 6).

3) Hamilton and Zeldin Approach: This scheme was originally devised in support of the National Aeronautics and Space Administration (NASA) space shuttle software development [14]. It is a response to the need to simplifying and clarifying the often overwhelming detail present in flowcharts

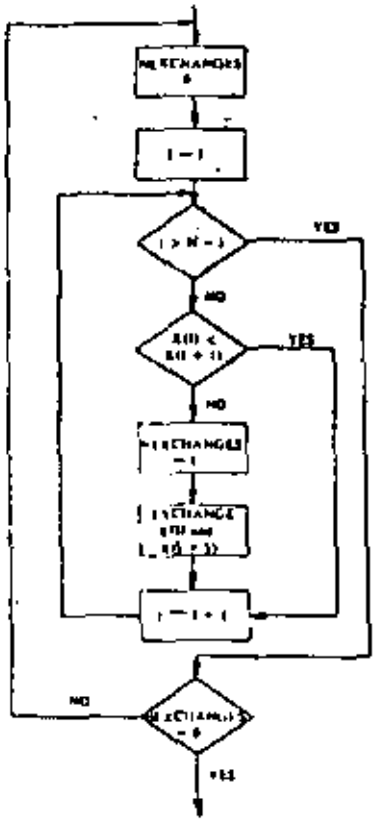


Fig. 5. An ANSI flowchart.

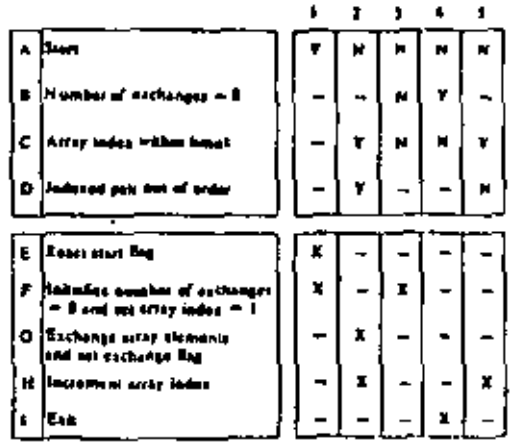


Fig. 6. A decision table representation of an algorithm.

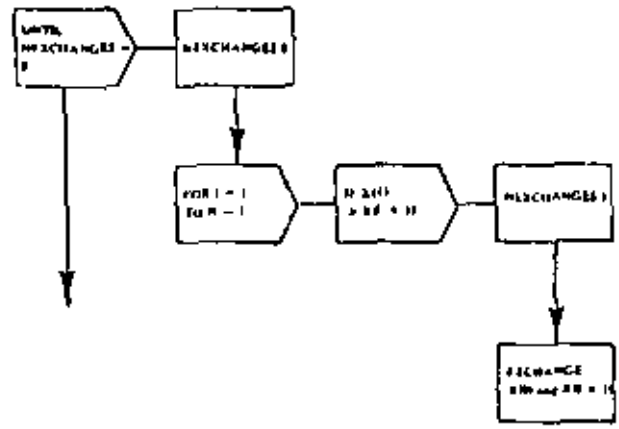


Fig. 7. The Hamilton and Zeldin design representation.

The Hamilton and Zeldin design representation scheme has been further refined and enhanced and is now supported by an automated flowcharting tool [15]. Diagrams of this type are called structured design diagrams (not to be confused with the "structured design method," Section IV).

The original intent of this scheme was to support the use of higher level languages. However, it could be employed in other applications. The objectives of this scheme include

1) explicitly depicting the levels of nesting (and hence, complexity) inherent in the design structure;
2) depicting all processes in an algorithmic manner (as opposed to employing textual material);
3) demonstrating the extent or scope of control/effect of all loops;
4) supporting and encouraging the use of the basic constructs and their alternative forms.

The basic effect of this scheme is that the software designer and customer can follow sequences of equations to determine whether or not they are appropriate while gaining an appreciation for the relationship(s) between different parts of a software system.

Structured design diagrams are composed of rectangles, directed lines, and a pentagonal combination of a rectangle and triangle. Execution is generally from top to bottom on the diagram with the exception of IF tests and DO loops which proceed from left to right with each occurrence. This clearly shows the level of nesting. Each of the graphic forms contains an algebraic statement or test, as appropriate. An example of this technique is presented in Fig. 7.

4) Nassi and Shneiderman Approach: This approach was introduced as a means of syntactically enforcing use of three basic program constructs: sequence, decision, and loop. Since its introduction [16] it has been modified [17], [18] and has been incorporated into an interactive graphics system to aid software design [19].

Nassi-Shneiderman diagrams are a departure from flowchart-based techniques. They aid in the adherence to the use of programming structures other than the GO TO (an unconditional transfer of control). A single process is completely contained within a rectangular box. The box is subdivided into sections. Each section denotes a specific subprocess (e.g., an assignment, decision). A hierarchy of such diagrams can be established and maintained via the use of routine calls or dummy processes which refer to other diagrams. The range of a loop, the basis and effect of a decision, and the general sequence of the process flow are all explicit in this scheme.

These diagrams consist of rectangles and triangles each containing a statement of the operation performed and are arranged so as to define process flow (Fig. 8).

D. Comments Regarding Software Design Representation

There are well over a dozen different approaches to software design representation. Many of these also have derivations making the total number in use today much higher [11]. In documenting a software design, three issues must be addressed regardless of what representation scheme is used

1) what information should be communicated?
2) who is the audience?
3) what are the concerns of the audience?

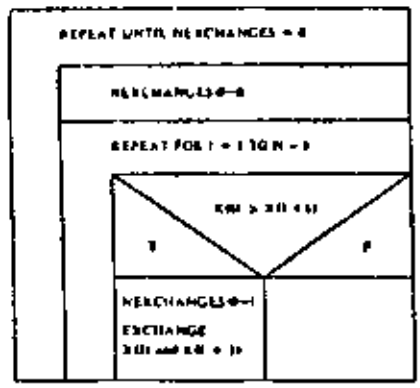


Fig. 8. The Nassi-Shneiderman representation approach.

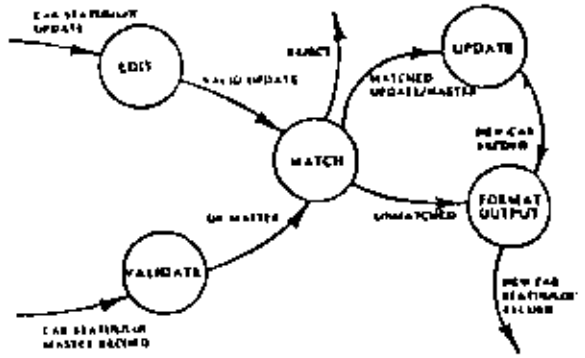


Fig. 9. A data-flow diagram.

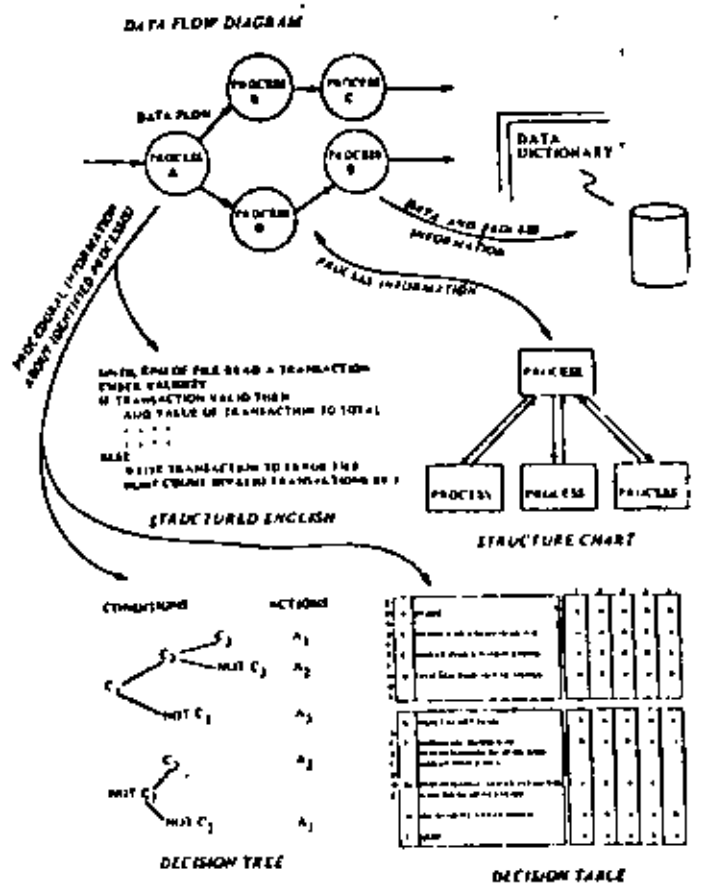


Fig. 10. An overview of the structured design method.

Since there are many different types of reviewers, it is unlikely that a single representation scheme will satisfy the needs of all of them. Rather, it is more likely that selection criteria and problem characterization parameters could be applied [11] in order to "engineer" a representation system composed of many different classes of representation schemes.

IV. Composition of Software

As challenging as the software design representation problem may seem, the problem of composing or creating the software in the first place is even more so. Many articles have been published which describe to the reader how software may be derived beginning with a few given pieces of information. Each method utilizes a design representation scheme or system of schemes in order to accomplish its goal(s). One approach [11] organizes these approaches into three categories

1) data-flow oriented methods
2) data structure methods
3) prescriptive methods.

The naming of these categories is related to the information they utilize as being given (e.g., a data-flow model) or the nature of the approach (e.g., prescriptive). Rather than describe the use of each of the more than one dozen major methods [11] currently documented in the literature, a representative example of each of the categories will by synopsized here.

A. Data-Flow Oriented Methods

Data-flow oriented methods provide the software engineer with guidance on how to define software given that he has a model of the data flow which will (eventually) occur in the system. A widely used approach is that of structured design [8].

This approach utilizes a network oriented design representation called a data-flow diagram and structure charts (see Section III). Together they form an integral part of a system of design representation directed at capitalizing on these depicted properties. Structured design's notational system depicts structural and behavioral characteristics separately and includes objective evaluations of design quality. These evaluations are aimed at identifying the level of system strength and flexibility (coupling) as well as the level of internal strength possessed by each model. These concepts can be applied to any software design and constitute what well may be a "natural law" of software composition. They need not be thought of as being strictly applicable to designs arrived at using the structured design method. The use of this method is complemented by the existence of a method for defining system specifications called structured analysis [20].

Structured design views systems in two complementary ways. One is the flow or movement of data while the other is the transformation(s) which such data flow undergoes to be transformed from input into output. Together they form a network model of a system. Data enter as input, undergo transformations, converge, diverge, get stored with other data, and become output. This view of software may sound simple and perhaps dull, but it generates several interesting side effects. Among these are the following.

1) Absence of time in the data-flow representation—Since movement and transformation of data are the only characteristics represented by the data-flow diagram, the concept of the

passage of time along any single or several data-flow path(s) is not present (Fig. 9). The software engineer is free to concentrate on the establishment of a clear understanding of what major/minor transformations must occur in order for the input data to be incrementally and correctly transformed into output.

2) Lack of classical functional decomposition—Top-down design has been described as showing only one path in a tree-like structure because it assumes that there is only one problem to be solved [5]. The use of the "data-flow viewpoint" reduces the effect(s) which the designer's experience and biases will have on the results. This allows the "shape" or structure of the system to be retained. Further on in the structured design process, these "natural aggregates" of transformations and data flows play a vital role in the identification and organization of modules in the construction plan or physical structure of the system.

A serious problem for software engineers is the control of complexity. Much of this complexity is caused by concern over solving the problem conceptually while allowing implementation issues to distort this abstract view. The software engineer finds himself pulled in two directions at once. Hence, his attention is focused alternatively on high level, abstract problem issues and low level, detail implementation ones. This shuttling between two somewhat incompatible sets of issues creates an unstable environment in which mistakes will be made. Some key issue or subtle nuance regarding the problem may be overlooked. Structured design recommends a distinction between abstract or logical design and physical design. This enables the software engineer to create such as the architect proceeds by first establishing a conceptual solution then adjusting it to the intended operating environment (making it practical).

The basic ploy used in structured design is to identify the data flow(s) apparent in the problem and to build in both detail and structure in an iterative fashion. It is assumed that a system specification exists before design begins. This specification identifies inputs, desired outputs, and a description of the functional aspects of the system. The specification is used as a basis for the graphical depiction of the inherent data flows and data transformations. This graphical depiction is called a data-flow diagram. From the data-flow diagrams, natural aggregates of these transformations and data flows are identified. This eventually leads to the definition and depiction of the modules and their relationship to one another and various system elements called a structure chart. The system specification is reexamined, errors and omissions remedied, and the process cycled through again. Not all steps are repeated, only those deemed necessary by the software engineer. An overview of this process is shown in Fig. 10.

B. Data Structure Oriented Methods

Data structure oriented methods give the software designer a means of proceeding to a design given that the structure of the data that will be processed by the software is known. In this case, structure refers to the logical relationship(s) which exist between data elements and not to the physical format of data. The basic premise in this case is that the structure of the data and the structure of the program must be compatible. The data structured approach we will examine here is that of Warnier [21]. It shares the data structure view with an approach authored by Jackson [22]. They are widely popular in Europe and growing in popularity in the U.S. but utilize fundamentally different approaches to design representation.

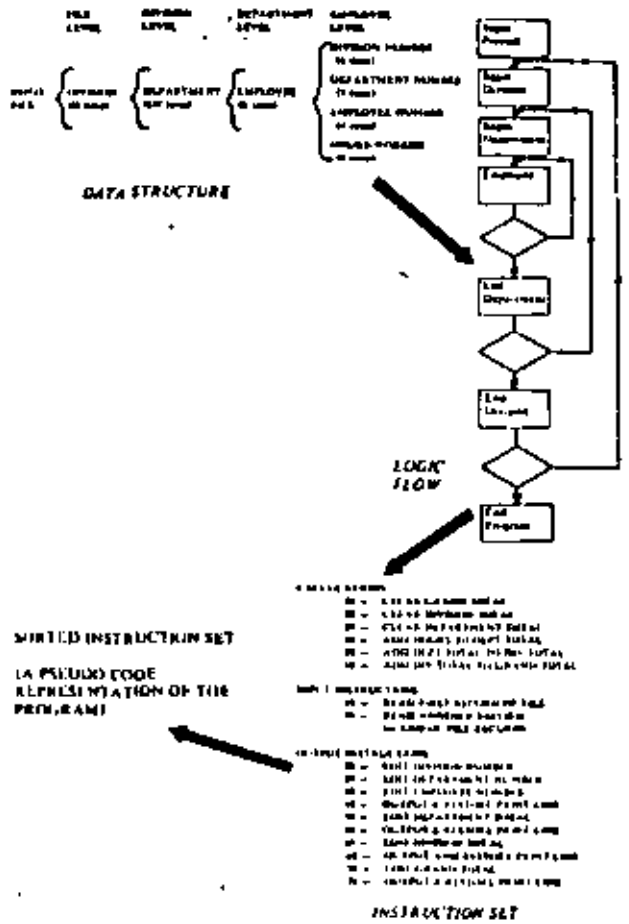


Fig. 11. An overview of the Warnier method.

Conceptually it is assumed that the inherent structure of input and output data is

1) discernible;
2) useful as a driving force in software development;
3) insurance that a prudently structured program will result.

This method employs the following four different design representation schemes:

1) data organization diagram;
2) logical sequence diagram;
3) instruction list;
4) pseudocode (a design oriented, codelike language with limited syntactic and semantic restrictive [23] or code.

The procedure for developing a software design using this method is relatively simple.

1) Identify all input data entities. Their format is a secondary issue to that of the relationship they have to one another.

2) Organize the input data into a hierarchy.

3) Assign a description of each entity in the input file and note the number of times it occurs (e.g., 1 employee file, n employees, each employee record contains 5 entries: social security number, hire date, employee number, company address/location, and home address).

4) Perform the same process as in 1)-3) above but for output data.

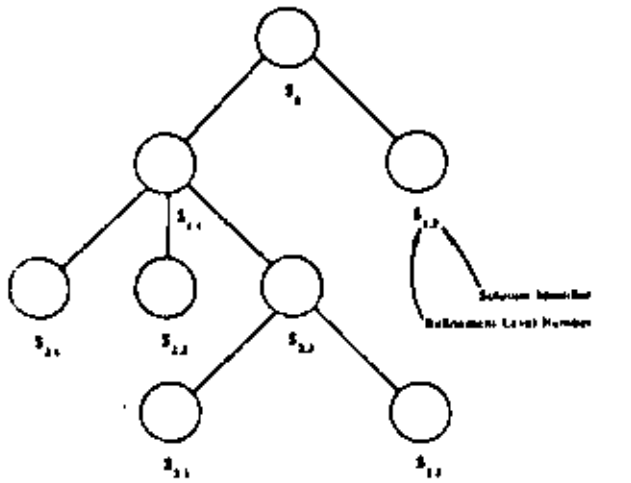


Fig. 12. An overview of iterative stepwise refinement.

5) Specify functional characteristics of the program by defining the instructions (by type) which will occur in the program. These are defined by type of instruction in the following order: read instructions, branch (and related) instructions, computations, outputs, and calls to subroutines.

6) Using a modified flowchart, display the logical sequence of instructions using symbols to represent begin process, end process, branching, and nesting.

7) Number each element of the logical sequence and expand it (where possible) using combinations of the basic set of instructions defined in 6) above.

Warnier provides a number of other more detailed instructions for the composition of programs using this method but these are not germane to our discussion. The development flow using this method is shown in Fig. 11.

C. Prescriptive Methods

This category contains most of the software composition methods currently available [11]. These methods are based on the experience(s) of their authors and lack some unifying conceptual framework. They are the result of valuable experience and a good deal of empiricism. Many of them are a collection of "good things to do" while others put forward an overall approach to solving software design problems. Most have adopted some specific set of design representation schemes. META stepwise refinement [24] does not contain a specific design representation scheme but does provide an overall approach to problem solving. It consists of three concepts: disciplined (or structured) programming, problem solving by iteration, and return on investment/cost effectiveness. The way in which each of these concepts manifests itself in this approach is described below.

1) Disciplined programming—This refers to both the process of decomposing the solution space in order to identify alternative solutions and to the breaking down of a given solution into its constituent attribute. It is not decomposition in the classic sense of the word, as we shall see.

2) Problem solving by iteration—Here, this refers to both the process of repetitively solving a design problem and to the practice of adding detail with each iteration.

3) Return on investment and cost effectiveness—The time and energy necessary to apply the other two concepts are brought into sharp focus via this one. In this case, further refinement is only done on the most promising of the alternative solutions. This makes for a more prudent and cost-effective expenditure of resources.

Finally, the refinement discipline is directed at postponing implementation decisions until the appropriate level of detail has been reached.

This technique begins with the (presumed) existence of an exact stable problem definition. Given that such an expression of the problem exists, the first step is to compose a simple statement of the solution to the problem. Next, several solutions are generated. Each of these is of the same level of detail as the others but each is more detailed than the initial one. Each set of equivalent solutions constitute a level of refinement. The "best" of these solutions is selected. It is now treated in the same manner as the initial solution. That is, several equivalent solutions of more detail are generated, and so forth. The process ends when a level of refinement is reached which can be directly implemented into the intended programming language (Fig. 12).

D. Comments Regarding Software Composition

Each of the methods available today relies heavily on a specific, somewhat inflexible model of software development. Often, the model is localized in character in that it addresses implementation rather than architectural problems. However the overall effect of the availability of methods in software design has been quite beneficial. Discipline, organization, and an openness in development are a few of the positive trends experienced thus far.

V. Conclusions

Software design is a branch of software engineering which is just beginning to show recognition of its commonality with other design efforts dating back to antiquity. The lack of a standardized approach to depicting software design can be both an asset and a liability. It is an asset in that the means of depicting a design can be tailored to the audience and the type of problems being considered. It is a liability in that much of this potential is lost due to local, factionalized views or requirements on how "best" to portray software—any software. Hopefully, the trend to expand local, program-oriented thinking into a global or systematized approach will accelerate progress toward viewing software design as a *system* of concepts and techniques, and not a secret.

References

[1] G. Mie, "Contributions to the optics of turbid media, especially colloidal metal solutions," *Ann. Phys.*, vol. 25, p. 377, 1908 (in German).

[2] L. W. Carrier, G. A. Cato, and K. J. von Essen, "The backscattering and extinction of visible and infrared radiation by selected major cloud models," *Appl. Opt.*, vol. 6, p. 1029, 1967.

[3] L. J. Peters and L. L. Tripp, "Design representation schemes," presented at the Microware Research Institute Symp. Computer Software Engineering, New York, Apr. 1976.

[4] J. R. Scott, "An engineering methodology for presenting software functional architecture," in *Proc. 3rd Int. Conf. Software Engineering* (Atlanta, GA), May 1978.

[5] L. C. Carpenter, A. S. Kawaguchi, L. J. Peters, and L. L. Tripp, "Systematic development of automated engineering systems," presented at the 2nd Japan-U.S.A. Symposium Automated Engineering Systems, Aug. 1975.

[6] L. L. Constantine, "Structure charts—A guide," course handout from "Structured design course," conducted Nov. 1974.

[7] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 2, pp. 115–138, 1974.

[8] E. N. Yourdon and L. L. Constantine, *Structured Design*. New York: Yourdon Inc., 1975.

[9] SofTech Inc., "How to increase programmer productivity through software engineering using PL/I" (course notes), SofTech, Wal-

Lano, R. J., "The N^2 Chart," TRW Software Series, TRW-SS-77-04, November 1977, 119 pp.

ISIOS Newsletter
Volume 10, No. 3
August 1978
Appendix 3, page 1

ABSTRACT

One of the increasingly significant problems facing the system designers and master planners of today is the timely and accurate definition of system element interfaces and task or activity interrelationships.

The classical approach to requirements and/or activity definition and allocation consists of the subdivision of large functions into multiple, understandable, discipline-oriented modules or blocks. Each of these blocks is then defined or specified in terms of its inputs, outputs and transfer functions. The emphasis, unfortunately, is usually placed on transfer functions, while the inputs and outputs (interfaces) are all but ignored. Transfer function definitions are more readily understood and communicated than interface definitions since these functions are quantifiable, structured, close to the implementation design, and can be easily related to past experience. A structured technique or discipline must therefore be acquired by the effective system designer and master planner, which places equal weight on all of the definition and allocation parameters. Interfaces and activity relationships must be dealt with in a top-down, hierarchial manner which is as structured and timely as today's transfer function definition process. The implementation of an effective interface definition methodology has become essential due to the increasing relationship complexity of today's tasks, organizations and systems.

This paper describes the N^2 Chart, which is an implementation tool and methodology for the tabulation, definition, analysis and description of functional interactions and interfaces. The tool presented is not limited to any particular field, discipline, market area or system type. The N^2 Chart is simple and easy to understand, structured and methodical, top-down in nature, communicative of the design, and forces a uniform level of design consistency. The N^2 Chart is an effective tool for the integration of all of the people, products, and paper that make up any given system.

An initial N^2 Chart definition is provided, together with discussions relating to its various usages, formats and applications. Applications covered include interface tabulation, definition, design and analysis activities. Other non-obvious applications are additionally discussed which are concerned with design description, operational procedure analysis and schedule analysis activities. Examples are presented of actual N^2 Chart implementations on numerous TRW studies and proposal activities. Two examples are presented in detail which illustrate the applications of the N^2 Chart in interface problem solving and in top-down system design definition.

6. CONCLUSIONS

The N^2 Chart technique has been used on over 50 individual studies and projects at TRW with a great deal of success. The primary attribute of the technique has been in its rapid communication of interface and relationship data between large numbers of people with varied interests, backgrounds, and needs. N^2 Charts have been used to describe and define large systems concepts, hardware component design details, software program detailed designs, organizational and operational relationships, and tasking and schedule interfaces. A box on an N^2 Chart has been used to define everything from an earth/spacecraft interface down to a connector pin or wire interface. One of the main values of the N^2 Chart has been its disciplined structure which by its nature forces the users to a more consistent and complete level of detail than other visual presentation techniques. Figure 6-1 summarizes the major features and applications of the N^2 Chart technique. The following paragraphs provide additional information on the use of N^2 Charts for the tabulation, definition, design, analysis, and description activities illustrated throughout this paper.

The tabulation task is probably the best fit for the N^2 Chart technique. Interface and relationship definition activities are greatly enhanced with the use of N^2 Charts. The interface accountability attributes of the technique, together with the change simplicity and reproducibility of the actual chart forms, make the N^2 Chart an ideal tool for large and small system tabulation tasks. This fact has been completely verified by all projects employing the N^2 Chart technique as the interface tabulation medium.

As a tool for interface and relationship definition tasks, the N^2 Chart provides the much needed "input" and "output" portions of the entire definition equation. The "transfer function" portion of the equation, however, cannot be accomplished totally with the N^2 Chart technique. The N^2 Chart must be used in conjunction with other techniques (e.g., block diagram, flowchart, etc.) in order to perform the total system definition task. However, the step sequence and loop N^2 Chart variations have aided the transfer function definition job to a greater extent than originally expected. Many projects have employed the step sequence N^2 Chart in preference to the flowchart for defining top level operational and data flows. The loop chart variation has proven very useful in defining the secondary system flows and dependencies (control, coordination, monitor, support, etc.).

The design and analysis activities involved with a system are aided by the N^2 Chart only through its tabulation, definition and description capabilities. The N^2 Chart technique does not provide mathematical proof or even formulation capability to automatically cope with the system design problems. Engineering experience, discipline, know-how, and judgment are still the most important ingredients in the system design process. The N^2 Chart does, however, provide a great tool for the description of the design process, both for the designers and the reviewers.

Lano, R. J., "The N^2 Chart," TRW Software Series, TRW-SS-77-04, November 1977, 119 pp.

ISDOS Newsletter
Volume 10, No. 3
August 1978
Appendix 3, page

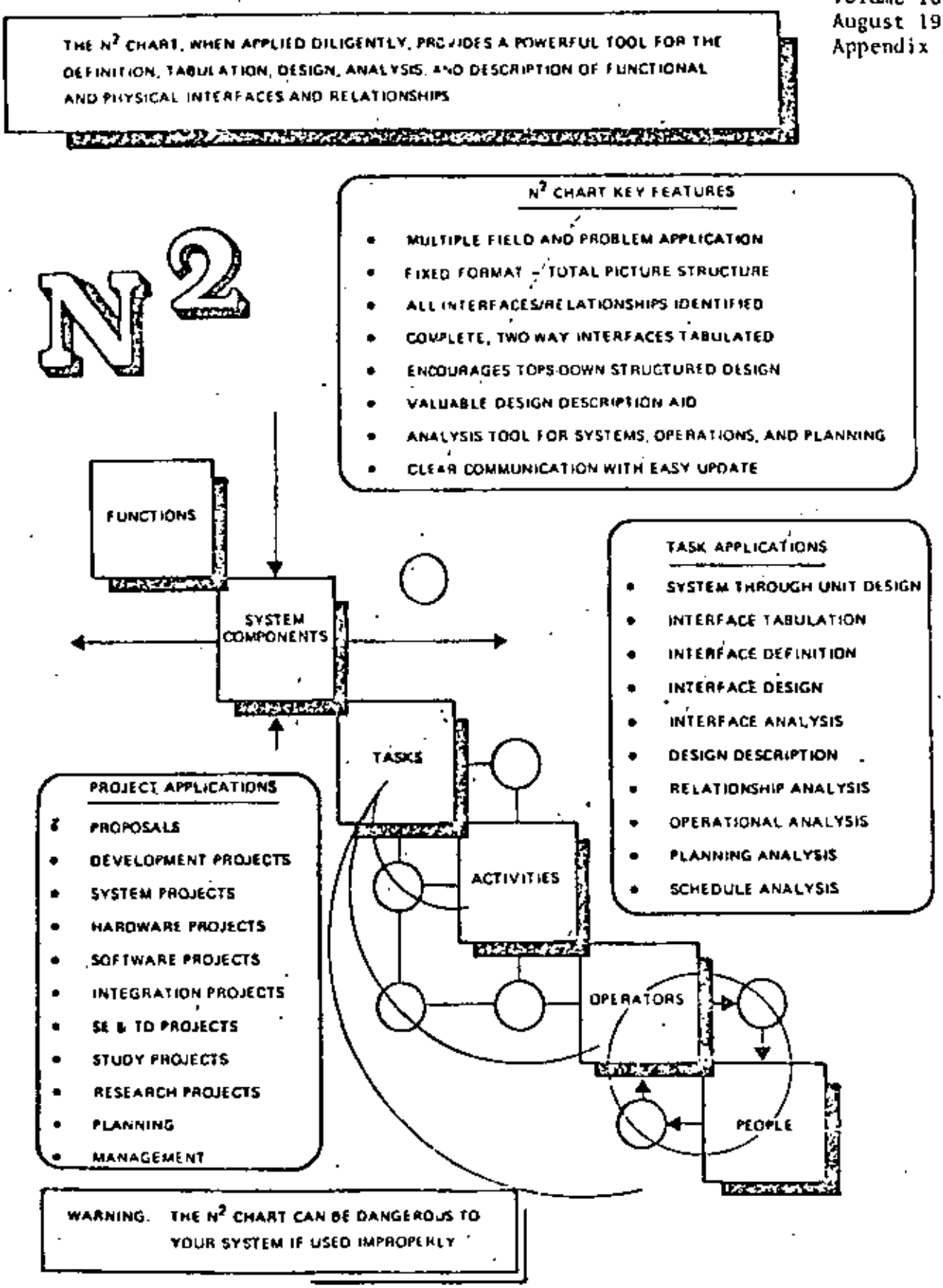


Figure 6-1. N^2 Chart Overview

3-5-11

Lano, R. J., "The N^2 Chart," TRW Software Series, TRW-SS-77-04, November 1977, 119 pp.

The N^2 Chart can, therefore, be thought of as a communication medium to coordinate the design and analysis processes rather than an equation or facility for accomplishing the task. A previous statement, which is most applicable to design and analysis activities, is repeated below to illustrate a possible short-fall of the N^2 Chart technique:

> *"The reader must be reminded that the N^2 Chart is only a tool for interface definition and not an automatic solution to the problems at hand. The usage of the N^2 Chart gains benefits only if the user is diligent and methodical in his definition effort. The N^2 Chart has the potential of being a dangerous crutch which, if used improperly, conveys the feeling of user understanding to the reviewer who does not spend the energy to comprehend and study in detail this 'lay it all out' visual presentation."*

The use of N^2 Charts as a description tool has proven extremely successful. A great quantity of complex information has been transferred to a large number of people in a short time period with the use of N^2 Chart techniques. The ease of production, simplicity of presentation, and the visually selective level of detail provided by this technique has proven to be invaluable as a communication medium among the designer, reviewers, consultants, users, and other participants in a system design effort. The descriptive attribute of the N^2 Chart technique has, through experience, become its primary attribute, even though it was not the original intent.

4.- INGENIERIA DE SOFTWARE.

Measurement and Experimentation in Software Engineering

BILL CURTIS, MEMBER, IEEE

Abstract—The contributions of measurement and experimentation to the state of the art in software engineering are reviewed. The role of measurement in developing theoretical models is discussed, and concerns for reliability and validity are stressed. Current approaches to measuring software characteristics are presented as examples. In particular, software complexity metrics related to control flow, module interconnectedness, and Halstead's Software Science are discussed. The use of experimental methods in evaluating cause-effect relationships is also discussed. Example programs of experimental research which investigated conditional statements and control flow are reviewed. The conclusion argues that many advances in software engineering will be related to improvements in the measurement and experimental evaluation of software techniques and practices.

I. Introduction

THE MAGNITUDE of costs involved in software development and maintenance magnify the need for a scientific foundation to support programming standards and management decisions. The argument for employing a particular software technique is more convincing if backed by experiments demonstrating its benefit. Rigorous scientific procedures must be applied to studying the development of software systems if we are to transform programming into an engineering discipline. At the core of these procedures is the development of measurement techniques and the determination of cause-effect relationships.

A commitment to measurement and experimentation hopefully begins by focusing on the phenomenon we are trying to explain. Rather than beginning by counting or experimentally manipulating various properties of software, we should first determine what software-related task we wish to understand. Modeling the processes underlying a software task helps identify properties of software that affect performance. Once the process is modeled, we can dissect it with all manner of scientific procedures.

The article on reliability by Musa [62] in this issue presents a rigorous approach to modeling a software phenomenon. He specifies a set of assumptions about software failures that guide his development of a quantitative measure. Yet, Musa does not stop with a description of his measure. He takes the critically important step of validating his equation with actual data. Further, he does not define his measure on the basis of a one-shot study, but continues to test and refine his model against new data sets.

Manuscript received April 9, 1980; revised June 9, 1980. This work was supported by the Office of Naval Research, Engineering Psychology Programs under Contract N00014-79-C-0595 to Information Systems Programs, General Electric Company, Arlington, VA. However, opinions expressed here are not necessarily those of the Department of the Navy or the General Electric Company.

The author was with the Software Management Research Unit, Information Systems Programs, General Electric Company, Arlington, VA. He is now with the Telecommunications Technology Center, International Telephone and Telegraph Company, Stratford, CT 06497.

Statements that a software product has a mean-time-between-failure of 48 h or satisfies specified timing constraints are grounded in the established measurement disciplines of reliability [22], [62], and performance evaluation. Other important attributes of software, such as its comprehensibility to programmers, have not been adequately defined. A model of how software characteristics affect programmer performance should underlie software engineering techniques which purport to make code more readable or reduce the mental load of the programmer. The empirical study of such a model requires a disciplined application of measurement and experimental methods.

Several important principles in measurement and experimentation will be discussed and their application in research on how certain software characteristics make a program difficult for a programmer to understand and work with will be reviewed. We will begin with a discussion of how measurement is fundamental to the development of a scientific discipline.

A. Science and Measurement

Margenau [53] argues that the various scientific disciplines can be classified by the degree to which their analytical approach is theoretical rather than correlational. The correlational approach explains phenomena by the degree of relationship among observable events. The theoretical approach attempts to explain these relationships with principles and constructs which are often several levels of abstraction removed from relationships among empirical data.

Torgerson [88] believes that "the sciences would order themselves in largely the same way (as Margenau's ordering) if they were classified on the . . . degree to which satisfactory measurement of their important variables has been achieved" [88, p. 2]. We know considerably more about measuring electricity or sound than we do about measuring the comprehensibility of software. Consequently, correlational studies are more characteristic of the behavioral than the physical sciences. According to Lord Kelvin [46]:

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

The development of scientific theory involves relating theoretical constructs to observable data. Fig. 1 illustrates two levels of theoretical modeling as discussed by Margenau and Torgerson. In a well-developed science constructs can be defined in terms of each other and are related by formal equations (e.g., force = mass X acceleration). A model of relationships among constructs becomes a theory when at least some

0018-9219/80/0900-1144$00.75

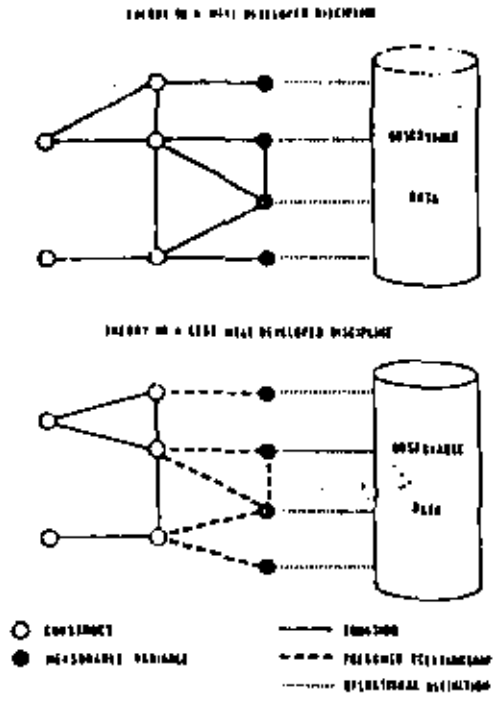


Fig. 1. The structure of theory in science.

constructs can be operationally defined in terms of observable data.

In a less well-developed science, relationships between theoretical and operationally defined constructs are not necessarily established on a formal mathematical basis, but are logically presumed to exist. Such relationships among operationally defined constructs are often described by correlation or regression coefficients, while their relationships to nonoperationally defined theoretical constructs are typically presented in verbal arguments. These presumed relationships are difficult to test, because negative results can be as easily attributed to a poor operational definition of the constructs as to an incorrect modeling of the relationships. In the next section, we will find presumed relationships existing between the hypothetical construct of program comprehensibility and its operational definition in software characteristics.

The development of an operational definition (i.e., the relating of a theoretical construct to observable data) requires a system of measurement. As described by Stevens [84, p. 24]:

> . . . the process of measurement is the process of mapping empirical properties or relations into a formal model. Measurement is possible because there is a kind of isomorphism between 1) the empirical relations among properties of objects and events and 2) the properties of the formal game in which numerals are the pawns and operators the moves.

Measurement does not define a construct, rather it quantifies a property of the construct. The "brightness" of light and the "intelligence" of programmers are represented with a number system. Numbers are that fortunate development which relieves us of reporting the size of a software system with several hundred thousand pebbles.

The value of any empirical study will depend on the reliability and validity of the data. *Reliability* concerns the extent to which measures are accurate and repeatable [66]. The less random error associated with a measurement, the more reliable it becomes. Two important factors underlying reliability are the internal consistency and the stability of the measure. If a measure is computed as the composite of several other measures, as in adding question scores to obtain an overall test score, then it is important to demonstrate that this composite is *internally consistent*. That is, all of the elementary measurements must be assessing the same construct and must be interrelated. If unrelated elements are added into a composite, then it is difficult to interpret the resulting score.

The other aspect of reliability, *stability*, implies that an equivalent score would be obtained on repeated collections of data under similar circumstances. The reliability of a measure limits the strength of its relationships to other measures. However, a reliable measure may not be a valid measure of a construct.

Validity has many interpretations, and all seem to concern whether a measure represents what it was designed to assess. Generally, three types of validity are identified which differ in their implications for the measure's ultimate use. Often validity will depend on the thoroughness with which a domain of interest has been covered. This concern for *content validity* is important for the software quality metrics to be discussed in the next section. Content validity requires an inclusive definition of the domain of interest, such as a definition of the phenomena covered under software complexity. A measure is often said to be "face valid" if it appears to broadly sample the content domain.

Predictive validity involves using the measure to predict the outcome of some event. For instance, does knowing something about software complexity allow one to predict how difficult a program will be to modify? Predictive validity is determined by a relationship between the measure and a criterion. A number of predictive validity studies will be reported in the next section.

In the less well-developed sciences the skill with which we operationally define constructs is critical in theory building. *Construct validity* concerns how closely an operational definition yields data related to an abstract construct. Construct validity is of immediate concern in developing software measurements, since many of our models do not rest on mathematical analysis.

Measurement consists of assigning numbers to represent the different states of a property belonging to the object under study. Relationships among these different states determine the type of measurement scale which should be employed in assigning numbers. Stevens [84] describes four types of scales which are presented in Table I. The important consideration with scales is that we only operate on numbers in a way which faithfully represents potential events among the properties they measure. That is, in considering jersey numbers (a nominal scale) we would not expect to add a fullback from Texas (#20) to a tackle from Harvard (#79) and end up with a Yugoslavian placekicker (#99). The operation of addition is limited to interval and ratio scales.

The most desirable scales are those which possess ratio properties, because of the broader range of mathematical transformations which can be legitimately applied to such data. The type of measurement scale also limits the type of statistical operations that can be sensibly applied in analyzing data. It makes little sense to add up all the jersey numbers, divide by the total number of players, and then claim that the average player is a center (#51). Since statistical techniques make no assumptions about the type of scale employed, this problem is one in measurement rather than statistical theory.

Improved measurement will result from concentration on what we really ought to measure rather than what properties

TABLE I
TYPES OF MEASUREMENT SCALES

SCALE	APPROPRIATE OPERATIONS*	DESCRIPTION	EXAMPLES
NOMINAL	$=, \neq$	CATEGORIES	SEX, RACE ASSIGN NUMBERS
ORDINAL	$<, >$	RANK ORDERINGS	HARDNESS OF MINERALS RANK IN CLASS
INTERVAL	$+, -$	EQUIVALENT INTERVALS BETWEEN NUMBERS	TEMPERATURE (F° AND C°) CALENDAR TIME
RATIO	$+$	EQUIVALENT INTERVALS AND ABSOLUTE ZERO	TEMPERATURE (K°), HEIGHT LINES OF CODE

* THE OPERATIONS LISTED FOR EACH SCALE ARE APPROPRIATE FOR ALL SCALES LISTED BENEATH IT.

are readily countable. The more rigorous our measurement techniques, the more thoroughly a theoretical model can be tested and calibrated. Thus progress in a scientific basis for software engineering depends on improved measurement of the fundamental constructs [45].

II. Measurement of Software Characteristics

A. Uses for Software Metrics

Measurements of software characteristics can provide valuable information throughout the software life cycle. During development measurements can be used to predict the resources which will be required in future phases of the project. For instance, metrics developed from the detailed design can be used to predict the amount of effort that will be required to implement and test the code. Metrics developed from the code can be used to predict the number of errors that may be found in subsequent testing or the difficulty involved in modifying a section of code. Because of their potential predictive value, software metrics can be used in at least three ways.

1) Management Information Tools: As a management tool, metrics provide several types of information. First, they can be used to predict future outcomes as discussed above. Measurements can be developed for costing and sizing at the project level, such as in the models proposed by Freiman and Park [33], Putnam [69], and Wolverton [93]. Other models have been developed for estimating productivity [32], [89]. Such metrics allow managers to assess progress, future problems, and resource requirements. If these metrics can be proven reliable and valid indicators of development processes, they provide an excellent source of management visibility into a software project.

2) Measures of Software Quality: Interest grows in creating quantifiable criteria against which a software product can be judged [60]. An example criterion would be the minimally acceptable mean time between failures. These criteria could be used as either acceptance standards by a software acquisition manager or as guidance to potential problems in the code during software validition and verification [90].

3) Feedback to Software Personnel: Elshoff [27] has used a software complexity metric to provide feedback to programmers about their code. When a section grows too complex they are instructed to redesign the code until the metric values are brought within acceptable limits.

The three uses described above suggest a difference between measures of process and product. Measures of process would include the resource estimation metrics described as potential management tools. Measures of cost and productivity quantify attributes of the development process. However, they convey little information about the actual state of the software product. Measures of the product represent software characteristics as they exist at a given time, but do not indicate how the software has evolved into this state. Measures used for feedback to programmers or as quality criteria fall within this second category.

Belady [5] argues that it will be difficult to develop a metric which can represent both process and product. Development of such a metric or set of metrics will require a model of how software evolves from a set of requirements into an operational program. Charting the sequential phases of the software life cycle will not provide a sufficient model. Some progress is being made on system evolution by Lehman and his colleagues at Imperial College in London [7], [91], [47], [49] and is discussed by Lehman [48] in this issue. In the remainder of this section, we will deal with measures of product rather than process.

B. Omnibus Approaches to Quantifying Software

There have been several attempts to quantify the elusive concept of software quality by developing an arsenal of metrics which quantify numerous factors underlying the concept. The most well known of these metric systems are those developed by Boehm, Brown *et al.* [11], Gilb [35], and McCall *et al.* [55]. The Boehm *et al.* and McCall *et al.* approaches are similar, although differing in some of the constructs and metrics they propose. Both of these systems have been developed from an intuitive clustering of software characteristics (Fig. 2).

The higher level constructs in each system represent 1) the current behavior of the software, 2) the ease of changing the software, and 3) the ease of converting or interfacing the system. From these primary concerns Boehm *et al.* develop seven intermediate constructs, while McCall *et al.* identify eleven quality factors. Beneath this second level Boehm *et al.* create twelve primitive constructs and McCall *et al.* define 23 criteria. For instance, at the level of a primitive construct or criterion both Boehm *et al.* and McCall *et al.* define a construct labeled "self-descriptiveness". For Boehm *et al.* this construct underlies the intermediate constructs of testability and understand-

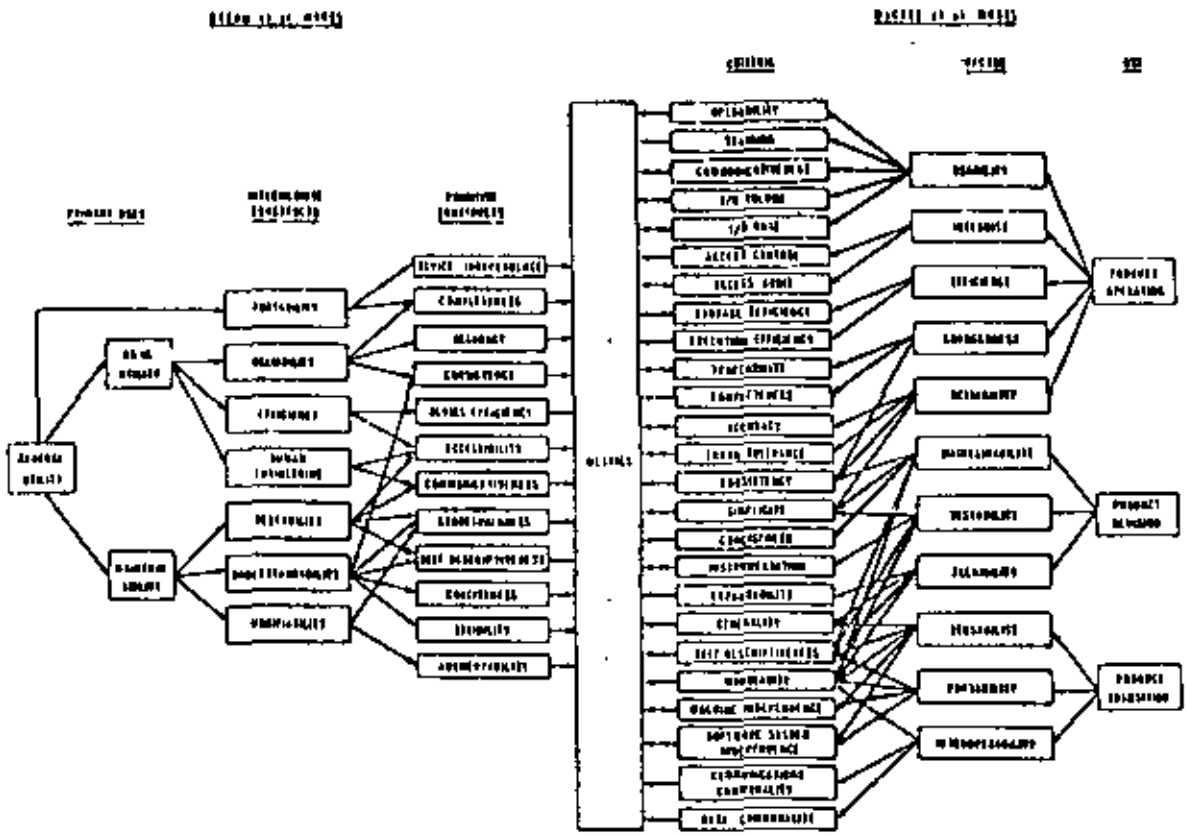


Fig. 3. The Boehm *et al.* and McCall *et al.* software quality models.

ability, both of which serve the primary use of measuring maintainability. For McCall *et al.* self-descriptiveness underlies a number of factors included under the domains of product revision and transition.

Primitive constructs and criteria are operationally defined by sets of metrics which provide the guidelines for collecting empirical data. The McCall *et al.* system defines 41 metrics consisting of 175 specific elements. Thus the metrics themselves represent composites of more elementary measures. This proliferation of measures should ultimately be reduced to a manageable set which can be automated. Reducing their number will require an empirical evaluation of which metrics carry the most information and how they cluster. There are a number of multivariate statistical techniques available for such analyses [61].

No software project can stay within a reasonable budget and maximize all of the quality factors. The nature of the system under development will determine the proper weighting of quality factors to be achieved in the delivered software. For instance, reliability was a critical concern for Apollo space flight software where human life was constantly at risk. For business systems, however, maintainability is typically of primary importance. In many real-time systems where space or time constraints are critical, efficiency takes precedence. However, optimizing code often lowers its quality as indexed by other factors such as maintainability and portability. Fig. 3 presents a tradeoff analysis among quality factors performed by McCall *et al.* [55].

The omnibus approach to metric development had its birth in the need for measures of software quality, particularly during system acquisition. However, the development of these metrics has not spawned explanatory theory concerning the processes affected by software characteristics. The value of these metric systems in focusing attention on quality issues is substantial. However, there is still a greater need for quantitative measures which emerge from the modeling of software phenomena. Much of the modeling of software characteristics has been performed in an attempt to understand software complexity.

C. Software Complexity

The measurement of software complexity is receiving increased attention, since software accounts for a growing proportion of total computer system costs [10]. Complexity has been a loosely defined term, and neither Boehm *et al.* nor McCall *et al.* included it among their constructs of software quality. Complexity is often considered synonymous with understandability or maintainability.

Two separate focuses have emerged in studying software complexity: computational and psychological complexity. *Computational complexity* relies on the formal mathematical analysis of such problems as algorithm efficiency and use of machine resources. Rabin [70, p. 625] defines this branch of complexity as "the quantitative aspects of the solutions to computational problems." In contrast to this formal analysis, the empirical study of *psychological complexity* has emerged from the understanding that software development and maintenance are largely human activities [91]. Psychological complexity is concerned with the characteristics of software which affect *programmer performance*.

The investigation of computational and psychological complexity has been carried on without a unifying definition for the construct of software complexity. There do, however, seem to be common threads running through the complexity

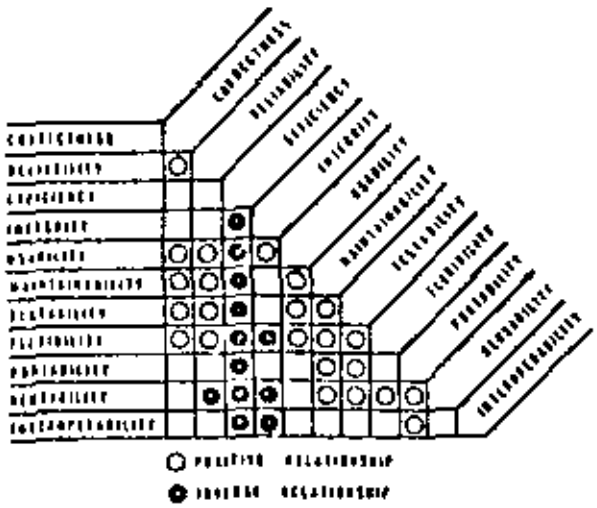


Fig. 3. McCall *et al.*'s tradeoff analysis among software quality factors.

literature [19, p. 102] which suggest the following definition:

> Complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software.

Several important points are implied by this definition. First, the focus of complexity is not merely on the software, but on the software's interactions with other systems. Complexity has little meaning in a vacuum; it requires a point of reference. This reference takes meaning only when developed from other systems such as machines, people, other software packages, etc. It is these systems that are affected by the "complexity" of a piece of software. Worrying about software characteristics in the absence of other systems has merit only in an artistic sense, and measures of "artistic" software are quite arbitrary. However, when there is an external reference (criterion) against which to compare software characteristics, it becomes possible to operationally define complexity.

Second, explicit criteria are not specified. This definition allows mathematicians and psychologists to become strange bedfellows since it does not specify the particular phenomena to be studied. Rather, this definition steps back a level of abstraction and describes the goal of complexity research and the reference against which complexity takes meaning. Complexity is an abstract construct, and operational definitions only capture specific aspects of it.

The second point suggests the third: complexity will have different operational definitions depending on the criterion under study. Operational definitions of complexity must be expressed in terms which are relevant to processes performed in other systems. Complexity is defined as a property of the software interface which affects the interaction between the software and another system. To assess this interaction, we must quantify software characteristics which are relevant to it. A model of software complexity implies not only a quantification of software characteristics, but also a theory of processes in other systems. Thus the starting point for developing a metric is not an ingenious parsing of software characteristics, but an understanding of how other systems function when they interact with software.

The following steps should be followed in modeling an aspect of software complexity:

1) define (and quantify) the criterion the metric will be developed to predict;
2) develop a model of processes in the interacting system which will affect this criterion;
3) identify the properties of software which affect the operation of these processes;
4) quantify these software characteristics;
5) validate this model with empirical research.

The importance of this last point cannot be overemphasized. Nice theories become even nicer when they work. Preparing for the rigors of empirical evaluation will probably result in fewer metrics and tighter theories. Results from validation studies make excellent report cards on the current state of the art.

Belady [6] has categorized much of the existing software complexity literature. First, he distinguishes different software characteristics which are measured as an index of complexity: algorithms, control structures, data, or composites of structures and data. In a second dimension he describes the type of measurement employed: informal concept, construct counts, probabilistic/statistical treatments, or relationships extracted from empirical data. Most research has concerned counts of software characteristics, particularly control structures and composites of control structures and data. We will review some of the complexity research in these two areas and compare them to a system level metric.

D. Control Structures

A number of metrics having a theoretical base in graph theory have been proposed to measure software complexity by assessing the control flow [8], [17], [36], [54], [71], [94]. Such metrics typically index the number of branches or paths created by the conditional expressions within a program. McCabe's metric will be described as an example of this approach, since it has received the most empirical attention.

McCabe [54] defined complexity in relation to the decision structure of a program. He attempted to assess complexity as it affects the testability and reliability of a module. McCabe's complexity metric $v(G)$ is the classical graph-theory cyclomatic number indicating the number of regions in a graph, or in the current usage, the number of linearly independent control paths comprising a program. When combined these paths generate the complete control structure of the program. McCabe's $v(G)$ can be computed as the number of predicate nodes plus 1, where a predicate node represents a decision point in the program. It can also be computed as the number of regions in a planar graph (a graph in regional form) of the control flow. This latter method is demonstrated in Fig. 4.

McCabe argues that his metric assesses the difficulty of testing a program, since it is a representation of the control paths which must be exercised during testing. From experience he believes that testing and reliability will become greater problems in a section of code whose $v(G)$ exceeds 10.

Basili and Reiter [4] and Myers [64] have developed different counting methods for computing cyclomatic complexity. These differences involved counting rules for CASE statements and compound predicates. Definitive data on the most effective counting rules have yet to be presented. Nevertheless, considering alternative counting schemes to those originally posed by the author of a metric is important in refining measurement techniques.

Evidence continues to mount that metrics developed from graphs of the control flow are related to important criteria such as the number of errors existing in a segment of code and the time to find and repair such errors [20], [28], [73]. Chen

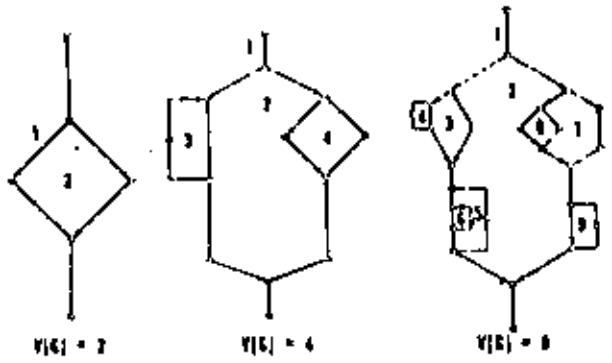


Fig. 4. Computation of McCabe's $v(G)$.

[17] developed a variation of the cyclomatic number which indexed the nesting of IF statements and related this to the information-theoretic notion of entropy within the control flow. He reported data from eight programmers indicating that productivity decreased as the value of his metric computed on their programs increased. Thus the number of control paths appears directly or indirectly related to psychological complexity.

E. Software Science

The best known and most thoroughly studied of what Belady [6] classifies as composite measures of complexity has emerged from Halstead's theory of Software Science [40], [42]. In 1972, Halstead argued that algorithms have measurable characteristics analogous to physical laws. Halstead proposed that a number of useful measures could be derived from simple counts of distinct operators and operands and the total frequencies of operators and operands. From these four quantities Halstead developed measures for the overall program length, potential smallest volume of an algorithm, actual volume of an algorithm in a particular language, program level (the difficulty of understanding a program), language level (a constant for a given language), programming effort (number of mental discriminations required to generate a program), program development time, and number of delivered bugs in a system. Two of the most frequently studied measures are calculated as follows:

$$V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$

$$E = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)}{2\eta_2}$$

where V is volume, E is effort, and

- η_1 number of unique operators
- η_2 number of unique operands
- N_1 total frequency of operators
- N_2 total frequency of operands.

Halstead's theory has been the subject of considerable evaluative research [31]. Correlations often greater than 0.90 have been reported between Halstead's metrics and such measures as the number of bugs in a program [8], [18], [30], [34], [67], programming time [39], [75], debugging time [20], [51], and algorithm purity [14], [26], [41].

We have evaluated the Halstead and McCabe metrics in a series of four experiments with professional programmers. In the first two experiments [21] problems in the experimental procedures, a limit on the size of programs studied, and substantial differences in performance among the 36 programmers

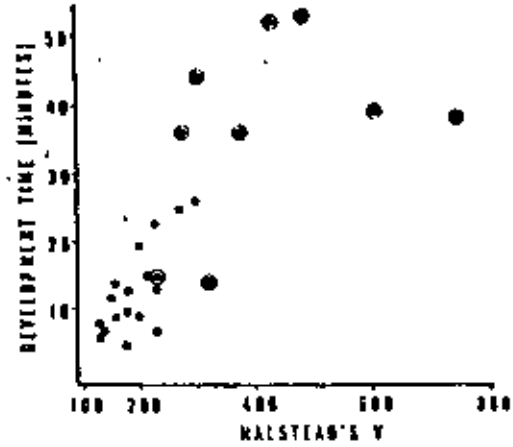


Fig. 5. Scatterplot of Halstead's V against development time from Sheppard *et al.*

involved in each, suppressed relationships between the metrics and task performance. In fact, it did not appear that the metrics were any better than the number of lines of code for predicting performance. However, in the third experiment [20] we used longer programs, increased the number of participants to 54, and eliminated earlier procedural problems. We found both the Halstead and McCabe metrics superior to lines of code for predicting the time to find and fix an error in the program.

In the final experiment [75], we asked nine programmers to each create three simple programs (e.g., find the maximum and minimum of a list of numbers) from a common specification of each program. The best predictor of the time required to develop and successfully run the program was Halstead's metric for program volume (Fig. 5). This relationship was slightly stronger than that for McCabe's $v(G)$, while lines of code exhibited almost no relationship.

The data points circled in Fig. 5 represent the data from a program whose specifications were less complete than those of the other two programs studied. The prediction of development time for this program was poor. We have observed in other studies that outcomes are more predictable on projects where a greater discipline regarding software standards and practices was observed [58], [59]. This experiment suggests that better prediction of outcomes may occur when more disciplined software development practices (e.g., more detailed program specifications) reduce the often dramatic performance differences among programmers.

In these experiments, we found Halstead's and McCabe's metrics to be valid measures of psychological complexity, regardless of whether the program they were computed on was developed by the programmer under study or by someone else. We concluded that there is considerable promise in using complexity metrics to predict the difficulty programmers will experience in working with software. Similar conclusions have been reached by Baker and Zweben [3] on an analytical rather than empirical evaluation of the Halstead and McCabe metrics.

Halstead's metrics have proven useful in actual practice. For instance, Elshoff [27] has used these metrics as feedback to programmers during development to indicate the complexity of their code. When metric values for their modules exceed a certain limit, programmers are instructed to consider ways of reducing module complexity. Bell and Sullivan [8] suggest that a reasonable limit on the Halstead value for length is 260, since they found that published algorithms with values above this figure typically contained an error.

Regardless of the empirical support for many of Halstead's predictions, the theoretical basis for his metrics needs considerable attention. Halstead, more than other researchers, tried to integrate theory from both computer science and psychology. Unfortunately, some of the psychological assumptions underlying his work are difficult to justify for the phenomena to which he applied them. In general, computer scientists would do well to immediately purge from their memories

1) the magic number 7 ± 2;
2) the Stroud number of 18 mental discriminations per second.

These numbers describe cognitive processes related to the perception or retention of simple stimuli, rather than the complex information processing tasks involved in programming. Broadbent [12] argues that for complicated tasks (such as understanding a program) the magic number is substantially less than seven. These numbers have been incorrectly applied in too many explanations and are too frequently cited by people who have never read the original articles [57], [85]. Regardless of the validity of his assumptions, Halstead was a pioneer in attempting to develop interdisciplinary theory, and his efforts have provided considerable grist for further investigation.

E. Interconnectedness

Since the modularization of software has become an increasingly important concept in software engineering [68], several metrics have been developed to assess the complexity of the interconnectedness among the parts comprising a software system [7], [56], [63], [65], [95]. For instance, Myers [63] models system complexity by developing a dependency matrix among pairs of modules based on whether there is an interface between them. Although his measure does not appear to have received much empirical attention, it does present two important considerations for modeling complexity at the system level [65]. The first consideration is the *strength* of a module; the nature of the relationships among the elements within a module. The stronger, more tightly bound a module, the more singular the purpose served by the processes performed within it. The second consideration is the *coupling* between modules; the relationship created between modules by the nature of the data and control that is passed between them.

A primary principle of modular design is to achieve as much independence among modules as possible. This independence helps to localize the impact of errors or modifications to within one or a few modules. Thus the complexity of the interface between modules may prove to be an excellent predictor of the difficulty experienced in developing and maintaining large systems. Myers' measure identifies data flow as a critical factor in maintainability. Nevertheless, his measure has not been completely operationally defined, and its current value is primarily heuristic. You and his associates [95] are currently working on validating a model of this generic type. Unfortunately, little empirical evidence is available to assess the predictive validity of such metrics.

The focus of metrics measuring the interconnectedness among parts of a system is quite different from those which measure elementary program constructs or control flow. Metrics measuring the latter phenomena take a microview of the program, while interconnectedness metrics speak to a macrolevel. An improved understanding of aggregating from the microlevel to the macrolevel needs to be achieved. For instance, summing the Halstead measures across modules leads to very different results than computing them once over the entire program [59].

Interconnectedness metrics may prove more appropriate parameters for macrolevel models such as those which predict maintenance costs and resources. Macrolevel metrics may prove better because factors to which microlevel metrics are more sensitive, such as individual differences among programmers, are balanced out at the macro- or project level. Macrolevel metrics are less perturbed by these factors, increasing their benefit to an overall understanding of system complexity and its impact on system costs and performance.

Although we can quantify a software characteristic and demonstrate that it correlates with some criterion, we have not demonstrated that it is a causal factor influencing that criterion. An argument for causality requires the support of rigorous experimentation. The experimental evaluation of software characteristics is a small but growing research area.

III. Experimental Evaluation of Software Characteristics

A. Cause-Effect Relationships

Most of the studies reported in the previous section do not demonstrate cause-effect relationships between software characteristics and programmer performance. That is, there were a number of uncontrolled factors in the data collection environment which could have influenced the observed data. These alternate explanations dilute any statement of cause and effect. Although structural equation techniques [25], [44] allow an investigation of whether the data are consistent with one or more theoretical models, a causal test of theory will require a rigorously controlled experiment. According to Cattell [16, p. 20]:

> An experiment is a recording of observations . . . made by defined and recorded operations and in defined conditions followed by examination of the data . . . for the existence of significant relations.

Two important characteristics of an experiment are that its data-collection procedures are repeatable and that each experimental event result in only one from among a defined set of possible outcomes [43]. An experiment does not prove an hypothesis. It does, however, allow for the rejection of competing alternative explanations of a phenomenon.

The confidence which can be placed in a cause-effect statement is determined by the control over extraneous variables exercised in the collection of data. For instance, Millman and Curtis [58] reported a field study in which a software development project guided by modern programming practices produced higher quality code with less effort and experienced fewer system test errors when compared to a sister project developing a similar system in the same environment which did not observe these practices. Although many of the environmental factors were controlled, an alternate explanation of the results was that the project guided by modern practices was performed by a programming team with more capable personnel.

An important characteristic of behavioral science experiments is the random assignment of participants to conditions [29]. By removing any systematic variation in the ability, motivation, etc., of participants across experimental conditions, this method supposedly eliminates the hypothesis that experimental effects are due to individual differences among participants. Assigning a morning class to one condition and an afternoon

class to another condition does not constitute random assignment, since students rarely choose class times on a random basis. However, if classes are the unit of study, the problem can be solved by randomly assigning a number of classes to each experimental condition. Random assignment has been a problem in testing causal relationships in field studies on actual software development projects.

There is often a conflict between what Campbell and Stanley [15] describe as the internal and external validity of an experiment. *Internal validity* concerns the rigor with which experimental controls are able to eliminate alternate explanations of the data. *External validity* concerns the degree to which the experimental situation resembles typical conditions surrounding the phenomena under study. Thus internal validity expresses the degree of faith in causal explanations, while external validity describes the generalizability of the results to actual situations.

In software engineering research, rigorous experimental controls are difficult to achieve on software projects and laboratory studies often seem contrived. External validity is probably a greater problem in studying process factors such as the organization of programming teams than in studying software characteristics. That is, the environmental conditions surrounding software development which are difficult to replicate in the laboratory would probably have a greater effect on the functioning of programming teams than on a programmer's comprehension of code.

Reviews of the experimental research in software engineering have been compiled by Atwood *et al.* [2] and Shneiderman [77]. Topics which have been submitted to experimental evaluation include batch versus interactive programming, programming style factors (e.g., indented listings, mnemonic variable names, and commenting), control structures, documentation formats, code review techniques, and programmer team organization. In discussing experimental methods, we will focus on the evaluation of conditional statements and control flow. These topics were not chosen because they were believed to be more important than other subjects. Rather, they were chosen because several programs of research have developed around them and because the conditional statement has been a focus of argument since it was originally assailed by Dijkstra [23] in 1968. Control statements have been a concern of the structured programming movement, and the results reported here evaluate their most effective implementation.

The usability of control statements is important since they account for a large proportion of the errors made by programmers [59], [96]. Control structures are closely related to some of the metrics discussed in a previous section, such as McCabe's cyclomatic number. Research will be described here in a tutorial fashion to demonstrate the depth and rigor with which experimental programs can investigate a problem.

B. Conditional Statements

Sime, Green, and their colleagues at Sheffield University have been studying the difficulty people experience in working with conditional statements. In their first experiment Sime *et al.* [81] compared the ability of nonprogrammers to develop a simple algorithm with either nested or branch-to-label conditionals. Nesting implies the embedding of a conditional statement within one of the branches of another conditional. Nested structures are designed to make this embedding more visible and comprehensible to a programmer. Branch-to-label structures obscure the visibility of embedded conditions, since the "true" branch of a conditional statement sends the control elsewhere in the program to a statement with a specified label.

The conditional for the nested language was an IF-THEN-OTHERWISE construct similar to conditionals used in Algol, PL/I, and Pascal. This conditional construct is written

```
IF [condition] THEN [process 1]
OTHERWISE [process 2].
```

The branch-to-label conditional was the IF-GO TO construct of Fortran and Basic which Dijkstra [23] considered harmful. This conditional was written

```
    IF [condition] GO TO L1
    [process 2]
L1  [process 1].
```

In both examples, if the condition is true, process 1 is executed; if it is not true, process 2 is executed. A "condition" might be an expression such as "$X > 0$," while a process might be one or more statements such as "$X = X + 1$." Participants used one of these microlanguages to build an algorithm which organized a set of cooking instructions depending on the attributes of the vegetable to be cooked.

Sime *et al.* found that participants using the GO TO construct finished fewer problems, took longer to complete them, and made more semantic (e.g., logic) errors in building their algorithms than participants using the IF-THEN-OTHERWISE construct. They concluded that the GO TO construct placed a greater cognitive load on programmers by requiring that they both set and remember the label to which a conditional statement might branch. Further, programmers had to remember the various conditions which could branch to a particular label. These conditions are potentially more numerous for a GO TO than for an OTHERWISE statement.

In a further study Sime *et al.* [82] questioned whether the superiority of nested over branch-to-label constructs would be maintained when multiple processes were performed under a single branch of a conditional. For instance, given a statement

IF condition THEN process 1 AND process 2,

there are two ways a programmer may interpret its execution

1) (IF condition THEN process 1) AND process 2, or
2) IF condition THEN (process 1 AND process 2).

In the first interpretation, process 2 is performed regardless of the state of the condition, while in the second it is performed only if the condition is true. Sime *et al.* believed it would be difficult for a programmer to retain the scope of the processes to be performed within each branch of a conditional statement. This difficulty would be especially acute when conditionals were nested within each other.

Sime *et al.*'s second experiment investigated different techniques for marking the scope of the processes subsumed under each branch of a conditional statement. In addition to the IF-GO TO conditional, they defined a nested BEGIN-END and a nested IF-NOT-END representing two different structures for marking the scope of each branch in nested conditionals (Table II). The BEGIN and END statements mark the scope of processes performed under one branch of a conditional statement, while the IF-NOT-END uses a more redundant scope marker by repeating the condition whose truth is being tested. In a strict sense, IF-NOT-END is the counterpart of the IF-THEN-ELSE construct, while the BEGIN-END markers could be used under either construct.

TABLE II
CONDITIONAL STRUCTURES INVESTIGATED BY SIME, GREEN, AND GUEST

IF GOTO	SELECT BEGIN END	SELECT IF-NOT END

IF (Condition 1) GOTO L1
IF (Condition 2) GOTO L2
(Process 4) STOP
L1 IF (Condition 3) GOTO L3
(Process 2) STOP
L2 (Process 3) STOP
L3 (Process 1) STOP
	IF (Condition 1) THEN
	BEGIN
	IF (Condition 2) THEN
	BEGIN (Process 1)
	END
	ELSE
	BEGIN (Process 2)
	END
	END
	ELSE
	BEGIN
	IF (Condition 3) THEN
	BEGIN (Process 3)
	END
	ELSE
	BEGIN (Process 4)
	END
	END
		IF (Condition 1)
		IF (Condition 2) (Process 1)
		NOT (Condition 1) (Process 2)
		END (Condition 2)
		NOT (Condition 1)
		IF (Condition 3) (Process 3)
		NOT (Condition 3) (Process 4)
		END (Condition 3)
		END (Condition 1)

As in the previous experiment, nonprogrammers were asked to develop an algorithm for each of five problems within 2 h total. Sime *et al.* found that more semantic (algorithmic) errors occurred in the IF-GO TO language, while errors in the nesting languages were primarily syntactic (grammatical). The BEGIN-END construct produced more syntactic errors and only half as many successful first runs as the other constructs. Errors were debugged ten times faster in the IF-NOT-END condition, which proved to be the most error free construct.

Results for semantic errors suggest that it is easier to keep track of the control flow in a nested language. However, when multiple processes are performed the use of scope markers often result in careless syntactic errors. Syntactic errors are more likely to occur with the BEGIN-END than in the IF-NOT-END constructs, because the redundancy of conditional expressions in the latter make the placement of markers more obvious.

Differential results for the two types of errors offer some validity for the syntactic/semantic model of programmer behavior developed by Shneiderman and Mayer [78]. This model distinguishes between information which is language specific (syntactic) and language independent (semantic). The correct design of the algorithm is a semantic issue, while the grammatically accurate expression of that algorithm in a language is a syntactic issue. The structure of a language may simplify the design of an algorithm, but make its expression more difficult. Obviously, a language design should seek to simplify both the design and expression of an algorithm.

Based on the results of this second experiment, Sime *et al.* [82] proposed that their memory load explanation be replaced with an explanation that information is easier to extract from some languages than others. They distinguished two types of information: sequence and taxon. *Sequence information* involves establishing or tracing the flow of control forward through a program. *Taxon* information involves the hierarchial arrangement of conditions and processes within a program. Such information is important when tracing backward through a program to determine what conditions must be satisfied for a process to be executed. Sime *et al.* hypothesized that sequence information is more easily obtained from a nested language, while taxon information is more easily extracted from a nested language which also contains the redundant conditional expressions. Conceptually, taxon information can be determined without fully understanding the sequence of processes within a program, since not all branches have to be examined in a backward tracing.

In two subsequent studies Green [37] validated these hypotheses in research with professional programmers. Sequence information was more easily determined from nested languages, although no differences were observed between the BEGIN-END and IF-NOT-END constructs. Backward tracing was performed much more easily with the IF-NOT-END construct. Green in his article [37], estimated that the time required for backward tracing was 15 percent less in IF-NOT-END than in IF-GO TO conditionals.

It is important to recognize that program comprehension is not a unidimensional cognitive process. Rather, different types of human information processing are required by different types of software tasks. Green demonstrated that certain constructs were more helpful for performing certain software tasks. Software engineering techniques may differ in the benefits they offer to different programming tasks, since they differ in the types of human information processing that they assist.

Since the IF-NOT-END construct is not implemented in existing languages, Sime *et al.* [79] investigated ways to improve the use of the BEGIN-END conditional markers. They developed a tool which would automatically build the syntactic portions of a conditional statement once the user chose the expression to be tested. In a second experimental condition, they developed an explicit writing procedure for helping participants develop the syntactic elements of a conditional statement. This procedure involved writing the syntax of the outermost conditional first, and then writing the syntax of conditionals nested within it. In the final condition participants were left to their own ways of using the conditional constructs.

Sime *et al.* found that participants solved more problems correctly on their first attempt using the automated tool, but that a writing procedure was almost as effective. The writing procedure reduced the number of syntactic errors, which had been the major problem with the BEGIN-END construct in earlier studies. Syntactic errors were not possible with the automated tool. The writing procedure and automated tools helped participants dispense with syntactic considerations quickly, so that they could spend more time concentrating on the semantic portion of the program (i.e., the function which was to be performed). However, once an error was made, it was equally difficult to correct regardless of the condition. Thus writing procedures and aids primarily increase the accuracy of the initial implementation.

Arblaster *et al.* [1] reported on two further experiments, which extended the results for writing rules to conditionals using GO TO's. Two groups of nonprogrammers, one of which had been taught the writing rules, constructed simple algorithms using the IF-GO TO conditional. The group trained in the writing procedures made fewer errors, semantic and syntactic, replicating the results obtained with the nested languages.

Arblaster *et al.* repeated this experiment using more sophisticated participants. They added two additional experimental conditions to those described above, both using an IF-IF NOT conditional structure. This conditional was written

```
    IF [condition 1] GO TO L1 IF NOT GO TO L2
L1  [process 1]
L2  [process 2].
```

One of the two groups using this conditional was also trained in the use of writing rules. Use of the IF-IF NOT procedure resulted in fewer syntactic errors than observed with the IF-GO TO conditional. No differences were found for semantic errors.

Arblaster *et al.* pointed out that an early version of Fortran had a conditional format consistent with those found to be most effective in their experiments. However, this format was lost when further revisions of the language opted for an arithmetic IF conditional.

Shneiderman [76] compared the use of the arithmetic IF

```
    IF [arithmetic condition] L1, L2, L3
L1  [process 1]
L2  [process 2]
L3  [process 3]
```

to the use of the logical IF

```
    IF [Boolean condition] GO TO L1
    [process 2]
L1  [process 1]
```

in Fortran. The arithmetic IF creates the possibility of a conditional with three branches (i.e., tests for >, =, and <). He found that novice programmers had more difficulty comprehending the arithmetic than the logical IF. However, no such differences were observed for more experienced programmers, who Shneiderman believed had adjusted to translating the more complex syntax of the arithmetic IF into their own semantic representation of the control logic.

In a recent experiment by Richards *et al.* quoted by Green [38], the ordinary nesting of IF-THEN-ELSE conditionals was compared to the nesting-of IF-NOT-END conditionals, and both were compared to a style of nesting in which an IF never directly follows a THEN. This last conditional would be written

```
IF [condition 1] THEN [process 1]
ELSE IF [condition 2] THEN [process 2]
ELSE IF [condition 3] THEN [process 3]
etc.
```

and has the appearance of a CASE statement. Although this arrangement is contrary to the tenets of structured coding [24], some have argued that it may be easier for programmers to understand [87].

Green reports that for different forms of comprehension questions, this IF-THEN-ELSE-IF conditional was "never much better and sometimes much worse" than the other forms of nested conditionals. He argues that it is important to design computer languages so that perceptual cues such as indenting the levels of nesting can visually display the structure of the code. These perceptual cues can relieve the programmer of searching through the program text, a task which is distracting, time consuming, and error prone.

The extensive program of research by Sime, Green, and their colleagues has answered important questions about the structure of conditional statements. Their conclusions should be heeded in the design of future languages. They demonstrated

1) the superiority of nested over branch-to-label conditionals;
2) the advantage of redundant expression of controlling conditions at the entrance to each conditional branch;
3) that the benefits of a software practice may vary with the nature of the task;
4) that a standard procedure for generating the syntax of a conditional statement can improve coding speed and accuracy.

Overall, these results indicate that the more visible and predictable the control flow of a program, the easier it is to work with.

Sime, Green, and their associates have demonstrated how a preference among conditional constructs can be reduced to an empirical question which provides alternatives that were previously unconsidered (e.g., the IF-NOT-END construct). Their research has investigated structured coding at the construct level [80]. The next step is to evaluate the reputed benefits of structured coding at the modular level, considering the various structured constructs in total.

C. Control Flow

Structured programming has become a catch-all term for programming practices related to system design, programming team organization, code reviews, configuration management, rules for program control flow, and myriad other procedures for software development and maintenance. This section will review only experiments related to structured coding: the enforcement of rules for program control flow. The control structures generally allowed under structured coding are displayed in Fig. 6.

To evaluate reputed problems with the GO TO statement, Lucas and Kaplan [52] instructed 32 students to develop a file update program in PL/C, and half were further instructed to avoid the use of GO TO's. However, programmers in the GO TO-less condition were not trained in using alternate conditional constructs. Not surprisingly then, the GO TO-less group required more runs to develop their programs. In a subsequent task, all participants were required to make a

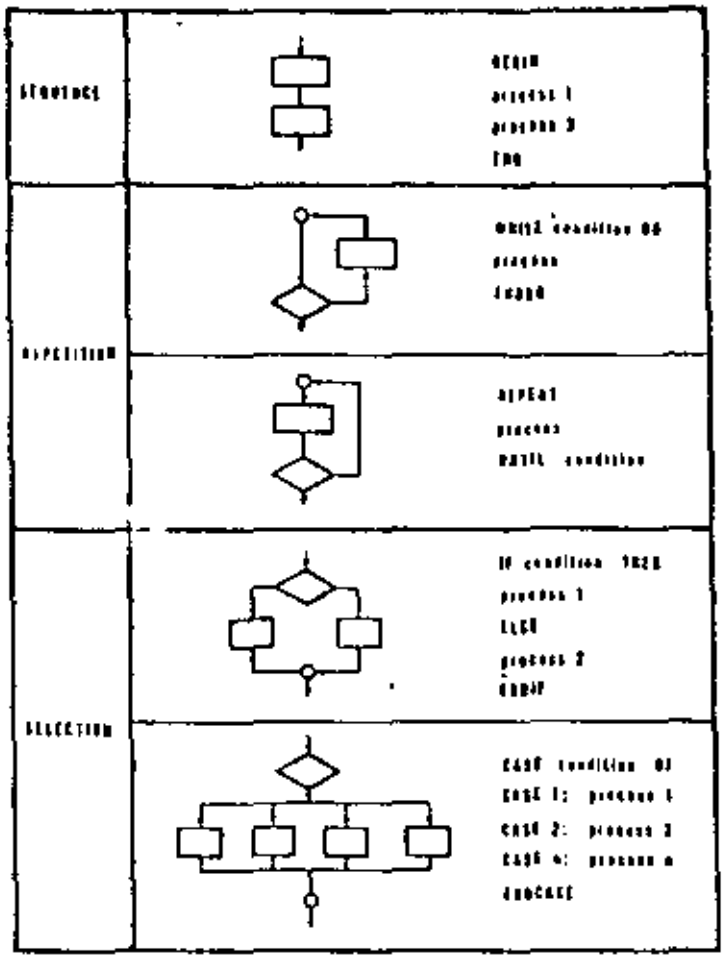


Fig. 6. Constructs in structured control flow.

specified modification to a structured program. Contrary to results on the earlier task, the group which had earlier struggled to write GO TO-less code made quicker modifications which required less compile time and storage space.

Weissman [92] also investigated the comprehension of PL/I programs written in versions whose control flow was either 1) structured, 2) unstructured but simple, or 3) unstructured and complex. Participants were given comprehension quizzes and required to make modifications to the programs. Higher performance scores were typically obtained on the structured rather than unstructured versions, and participants reported feeling more comfortable with structured code. Love [50] subsequently found that graduate students could comprehend programs with a simplified control flow more easily than programs with a more complex control flow.

Recently, a series of experiments evaluating the benefits of structured code for professional programmers was conducted in our research unit by Sheppard *et al.* [74]. In the first experiment participants were asked to study a modular-sized Fortran program for 20 min, and then reconstruct it from memory. Three versions of control flow performing identical functions were defined for each of nine programs. One version was structured to be consistent with the principles of structured coding described by Dijkstra [24] by allowing only three basic control constructs: linear sequence, structured selection, and structured iteration. Because structured constructs are sometimes awkward to implement in Fortran IV [86], a more naturally structured control flow was constructed which allowed limited deviations from strict structuring: multiple returns, judicious backward GO TO's and forward mid-loop exits from a DO. Finally, a deliberately convoluted version was developed which included constructs that had not been permitted in the structured or naturally structured versions, such as backward exits from DO loops, arithmetic IF's, and unrestricted use of GO TO's.

As expected, control flow did affect performance. The convoluted control flow was the most difficult to comprehend (Fig. 7). The difference in the average percent of reconstructed statements for naturally structured and convoluted control flows was statistically significant. Contrary to expectations, however, the strictly structured version did not produce the best performance. A slightly (although not significantly) greater percent of statements were recalled from naturally structured than from strictly structured programs.

In a second experiment Sheppard *et al.* instructed programmers to make specified modifications to three programs, each of which was written in the three versions of control flow described previously. A significantly higher percent of steps required to complete the modification was correctly implemented in the structured programs when compared to convoluted ones (Fig. 7). There were no differences in the times required. No statistically significant differences appeared between the two versions of structured control flow, although performance was slightly better on strictly rather than naturally structured code. These results suggested that the presence of a consistent structured discipline in the code, either strict or natural, was beneficial, and minor deviations

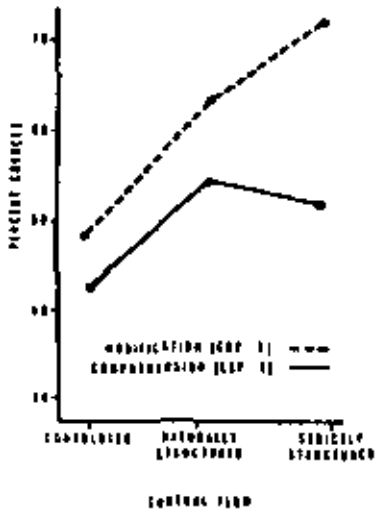


Fig. 7. Performance by type of control flow from Sheppard *et al.*

from strict structuring did not adversely affect performance.

In a third experiment Sheppard *et al.* decided to compare the two versions of structured Fortran IV to Fortran 77, which contains the IF-THEN-ELSE, DO-WHILE, and DO-UNTIL constructs. They measured how long a programmer took to find a simple error embedded in a program. No differences were attributable to the type of structured control flow, replicating similar results in the first two experiments. The advantage of structured coding appears to reside in the ability of the programmer to develop expectations about the flow of control—expectations which are not seriously violated by minor deviations from strict structuring.

The research reviewed here indicates that programs in which some form of structured coding is enforced will be easier to comprehend and modify than programs in which such coding discipline is not enforced. It is not clear that structured coding will improve the productivity of programmers during implementation. Some productivity improvements may be observed if less severe, more easily corrected errors are made using structured constructs, as suggested in the data of Sime and his colleagues. However, structured coding should reduce the costs of maintenance since such programs are less psychologically complex to work with. Experiments such as these can provide valuable guidance for decisions about an optimal mix of software standards and practices.

D. Problems in Experimental Research

It is important to recognize the benefits and limitations of controlled laboratory research. On the positive side, rigorous controls allow experiments to isolate the effects of experimentally manipulated factors and identify possible cause-effect relationships in the data. On the other hand, the limitations of controlled research restrict the generalizations which can be made from the data. Laboratory research has an air of artificiality, regardless of how realistic researchers make the tasks.

Several problems attendant to most current empirical validation studies severely limit the generalizability of conclusions which can be drawn from them [13]. For instance, program sizes have frequently been restricted because of limitations in the research situation. This problem is characteristic of experimental research where time limitations do not allow participants to perform experimental tasks such as the coding or design of large systems. Also, since new factors come into play in the development of large systems (e.g., team interactions), the magnitude of a technique's effect on project performance may differ markedly from its effect in the laboratory.

The nature of the applications studied are often limited by the environments from which the programs are drawn (e.g., military systems, commercial systems, real time, nonreal time, etc.). Further, there is frequently little assessment of whether results will hold up across programming languages. It is extremely difficult to perform evaluative research over a broad range of applications, especially when experimental procedures are used. Thus empirical results should be replicated over a series of studies on different types of programs in languages other than Fortran.

Another problem arises with what Sackman *et al.* [72] observed to be 25 or 30 to 1 differences in performance among programmers. This dramatic variation in performance scores can easily disguise relationships between software characteristics and associated criteria. That is, differences in the time or accuracy of performing some software task can often be attributed more easily to differences among programmers than to differences in software characteristics. Careful attention to experimental design is required to control this problem.

If generalizations are to be made about the performance of professional programmers, this is the population that should be studied rather than novices. As is true in most fields, there are qualitative differences in the problem-solving processes of experts and novices [83]. However, the advantage of some techniques is the ease with which they are learned, and novices are the appropriate population for studying such benefits. Attempts to generalize experimental results must also be tempered by an understanding of how real-world factors affect outcomes. Data should be collected in actual programming environments to both validate conclusions drawn from the laboratory and determine the influence of real-world factors.

IV. Summary

Software wizardry becomes an engineering discipline when scientific methods are applied to its development. The first step in applying these methods is modeling the important constructs and processes. When these constructs have been identified, the second step is to develop measurement techniques so that the language of mathematics can describe relationships among them. The testing of cause–effect relationships in a theoretical model requires the performance of critical experiments to eliminate alternative explanations of the phenomena. Even when possessed of supportive experimental evidence, our sermonizing should be cautious until we have established limits for the generalizability of our data.

There are four major points I have stressed, some by implication, in this review. First, measurement and experimentation are complementary processes. The results of an experiment can be no more valid than the measurement of the constructs investigated. The development of sound measurement techniques is a prerequisite of good experimentation. Many studies have elaborately defined the independent variables (e.g., the software practice to be varied) and hastily employed a handy but poorly developed dependent measure (criterion). Results from such experiments, whether significant or not, are difficult to explain.

Second, results are far more impressive when they emerge

from a program of research rather than from one-shot studies. Programs of research benefit from several advantages, one of the most important being the opportunity to replicate findings. When a basic finding (e.g., the benefit of structured coding) can be replicated over several different tasks (comprehension, modification, etc.) it becomes much more convincing. A series of studies also result in deeper explication of both the important factors governing a process and the limits of their effects. For instance, Sime, Green, and their colleagues identified the benefits and limitations of nested conditionals in an extensive program of research. Performing a series of studies also affords an opportunity to improve measurement and experimental methods. Thus the reliability and validity of results can be improved in succeeding studies.

Third, the rigors of measurement and experimentation require serious consideration of processes underlying software phenomena. Definitions should not be based on popular consensus. As energy is invested in defining constructs, a clearer picture of the process often emerges. Factors not thought to be a part of the process may present themselves as direct effects or limiting conditions. Alternate approaches to the techniques will also emerge, as in Sime *et al.*'s development of the IF-NOT-END nested conditional. The frustrations of scientific investigation are often the mothers of invention. Improved measurement techniques also provide better tools for management information systems. More reliable and valid measurement provides greater visibility and insight into project progress and product quality.

Finally, there is no substitute for sound experimental evidence in arguing the benefits of a particular software engineering practice or in comparing the relative merits of several practices. Managers often vacillate between their desire for proof and their impatience with the scientific approach. However, with understanding comes the possibility of greater control over outcomes, especially if causal factors and their limitations have been identified. The merchants of software elixirs can always quote case studies. Yet, such experiential evidence is no replacement for experimentation, and is frequently no better than a manager's intuition.

Measurement and experimentation are not intellectual diversions. They are the scientific foundations from which engineering disciplines continue to be built. Scientists must be sensitive to the most important questions they should tackle in software engineering, and should constantly reassess research priorities to keep pace with the state of the art. Software professionals need to encourage the scientific investigation of their business if real improvements are to be made in software productivity and quality. The scientific study of software engineering is young, and its rate of progress will improve as measurement techniques and experimental methods mature.

ACKNOWLEDGMENT

The author would like to thank Laszlo Belady, Sylvia Sheppard, Dr. Elizabeth Kruesi, and Dr. John O'Hare for their thoughts and comments. He would also like to thank Pat Belback, Noel Albert, Kathy Olmstead, and Lisa Grey for manuscript preparation and Lou Oliver for his support and encouragement.

REFERENCES

[1] A. T. Arblaster, M. E. Sime, and T. R. G. Green, "Jumping to some purpose," *Computer J.*, vol. 22, pp. 105-109, 1979.

[2] M. E. Atwood, H. R. Ramsey, J. N. Hooper, and D. A. Kollas, *Annotated Bibliography on Human Factors in Software Development* (Tech. Rep. TR-P-79-1). Alexandria, VA: Army Res. Inst., 1979. (NTIS no. AD A071 113.)

[3] A. L. Baker and S. H. Zweben, "A comparison of measures of control flow complexity," in *Proc. COMPSAC '79*. New York: IEEE, 1979, pp. 695-701.

[4] V. R. Basili and R. W. Reiter, "Evaluating automatable measures of software development," in *Proc. IEEE Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*, pp. 107-116, 1980.

[5] L. A. Belady, "Software complexity," in *Software Phenomenology*. Atlantic Aerospace Electronics, 1977, pp. 371-383.

[6] —, "Complexity of programming: A brief summary," in *Proc. IEEE Workshop on Quantitative Models of Software Reliability, Complexity, and Cost*, pp. 90-94, 1980.

[7] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 225-252, 1976.

[8] D. E. Bell and J. E. Sullivan, *Further Investigations into the Complexity of Software* (Tech. Rep. MTR-2874). Bedford, MA: MITRE, 1974.

[9] G. Benyon-Tinker, "Complexity measures in an evolving large system," in *Proc. IEEE Workshop Quantitative Software Models for Reliability, Complexity, and Cost*, pp. 117-127, 1980.

[10] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, vol. 19, no. 5, pp. 48-59, 1973.

[11] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, and M. J. Merrit, *Characteristics of Software Quality*. Amsterdam, The Netherlands: North Holland, 1978.

[12] D. E. Broadbent, "The magic number seven after fifteen years," A. Kennedy and A. Wilkes, Eds., in *Studies in Long Term Memory*. New York: Wiley, 1975, pp. 3-18.

[13] R. Brooks, "Studying programmer behavior experimentally: The problems of proper methodology," *Commun. ACM*, vol. 23, pp. 207-213, 1980.

[14] N. Bulut and M. H. Halstead, "Impurities found in algorithm implementation," *SIGPLAN Notices*, vol. 9, no. 3, pp. 9-10, 1974.

[15] D. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Chicago, IL: Rand-McNally, 1966.

[16] R. B. Cattell, "The principles of experimental design and analysis in relation to theory building," in R. B. Cattell, Ed., *Handbook of Multivariate Experimental Psychology*. Chicago, IL: Rand-McNally, 1966, 19-66.

[17] E. T. Chen, "Program complexity and programmer productivity," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 187-194, 1978.

[18] L. M. Cornell and M. H. Halstead, *Predicting the Number of Bugs Expected in a Program Module* (Tech. Rep. CSD-TR-205). West Lafayette, IN: Purdue Univ., Computer Science Dep., 1976.

[19] B. Curtis, "In search of software complexity," in *Proc. IEEE Workshop Quantitative Software Models for Reliability, Complexity, and Cost*, pp. 95-106, 1980.

[20] B. Curtis, S. B. Sheppard, and P. Milliman, "Third time charm: Stronger prediction of programmer performance by software complexity metrics," in *Proc. 4th IEEE Int. Conf. Software Engineering*, pp. 356-360, 1979.

[21] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 96-104, 1979.

[22] Data Analysis Center for Software, *Quantitative Software Models* (SRR-1). Griffiss AFB, NY: Rome Air Development Center, 1979.

[23] E. W. Dijkstra, "GO TO statement considered harmful," *Commun. ACM*, vol. 11, pp. 147-148, 1968.

[24] —, "Notes on structured programming," in O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. *Structured Programming*. New York: Academic, 1972, pp. 1-82.

[25] O. D. Duncan, *Introduction to Structural Equation Models*. New York: Academic, 1975.

[26] J. L. Elshoff, "Measuring commercial PL/I programs using Halstead's criteria," *SIGPLAN Notices*, vol. 11, pp. 38-46, 1976.

[27] J. L. Elshoff, "A review of software measurement studies at General Motors Research Laboratories," in *Proc. 2nd IEEE Software Life Cycle Management Workshop*, pp. 166-171, 1978.

[28] A. R. Feuer and E. G. Fowlkes, "Some results from an empirical study of computer software," in *Proc. 4th IEEE Int. Conf. Software Engineering*, pp. 351-355, 1979.

[29] R. A. Fisher, *The Design of Experiments*. London: Oliver & Boyd, 1935.

[30] A. B. Fitzsimmons, "Relating the presence of software errors to the theory of software science," in A. E. Wasserman and R. H. Sprague, Eds., *Proceedings of the Eleventh Hawaii International Conference on Systems Sciences*. Western Periodicals, 1978, pp. 40-46.

[31] A. B. Fitzsimmons and L. T. Love, "A review and evaluation of

software science," *ACM Computing Surveys*, vol. 10, pp. 3–18, 1978.

[32] K. Freburger and V. R. Basili, *The Software Engineering Laboratory: Relationship Equations* (Tech. Rep. TR-764). College Park, MD: Univ. of Maryland, Computer Science Dep., 1979.

[33] F. R. Freiman and R. E. Park "PRICE software model - Version 3 an overview," in *Proc. IEEE Workshop Quantitative Software Models for Reliability, Complexity, and Cost*, pp. 32–41, 1980.

[34] Y. Funami and M. H. Halstead, "A software physics analysis of Akiyama's debugging data," in *Proceedings of the MRI 14th International Symposium: Software Engineering*. New York: Polytechnic Press, 1976, pp. 133–138.

[35] T. Gilb, *Software Metrics*. Cambridge, MA: Winthrop, 1977.

[36] T. F. Green, N. F. Schneidewind, G. T. Howard and R. Parkeau, "Program structure, complexity and error characteristics," in *Proceedings of the Symposium on Computer Software Engineering*. New York: Polytechnic Press, 1976, pp. 139–154.

[37] T. R. G. Green, "Conditional program statements and their comprehensibility to professional programmers," *J. Occupational Psychology*, vol. 50, pp. 93–109, 1977.

[38] ——, "Ifs and thens: Is nesting just for the birds," *Software–Practice and Experience*, vol. 10, 1980, to be published.

[39] R. D. Gordon and M. H. Halstead, "An experiment comparing Fortran programming times with the software physics hypothesis," in *AFIPS Conf. Proc.*, vol. 45, pp. 935–937, 1976.

[40] M. H. Halstead, "Natural laws controlling algorithm structure," *SIGPLAN Notices*, vol. 7, no. 2, pp. 19–26, 1972.

[41] ——, "An experimental determination of the "purity" of a trivial algorithm," *ACM SIGME Performance Evaluation Rev.*, vol. 2, no. 1, pp. 10–15, 1973.

[42] ——, *Elements of Software Science*. New York: Elsevier, North-Holland, 1977.

[43] W. L. Hays, *Statistics*. New York: Holt, Rinehart and Winston, 1973.

[44] D. R. Heise, *Causal Analysis*. New York: Wiley, 1975.

[45] T. C. Jones, "Measuring programming quality and productivity," *IBM Syst. J.*, vol. 17, no. 1, pp. 39–63, 1978.

[46] W. T. Kelvin, "Popular lectures and addresses, 1891–1894," Quoted by M. L. Shooman in the Introduction to *Proc. IEEE Workshop Quantitative Software Models for Reliability, Complexity, and Cost*, 1980.

[47] M. M. Lehman "Programs, cities, students–Limits to growth," in D. Gries, Ed., *Programming Methodology*. New York: Springer-Verlag, 1978, pp. 42–69.

[48] ——, "Programs, programming and the software life cycle," this issue, pp. 1061–1076.

[49] M. M. Lehman and F. N. Parr, "Program evolution and its impact on software engineering," in *Proc. 2nd IEEE Int. Conf. Software Engineering*, pp. 350–357, 1976.

[50] T. Love, "An experimental investigation of the effect of program structure on program understanding," *SIGPLAN Notices*, vol. 12, no. 3, pp. 105–113, 1977.

[51] L. T. Love and A. Bowman, "An independent test of the theory of software physics," *SIGPLAN Notices*, vol. 11, pp. 42–49, 1976.

[52] H. C. Lucas and R. B. Kaplan, "A structured programming experiment," *Computer J.*, vol. 19, pp. 136–138, 1976.

[53] H. Margenau, *The Nature of Physical Reality*. New York: McGraw-Hill, 1950.

[54] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, 1976.

[55] J. A. McCall, P. K. Richards, and G. F. Walters, *Factors in Software Quality* (Tech. Rep. 77CIS02). Sunnyvale, CA: General Electric, Command and Information Systems, 1977.

[56] C. L. McClure, *Reducing COBOL Complexity through Structured Programming*. New York: Van Nostrand Reinhold, 1978.

[57] G. A. Miller, "The magic number seven, plus or minus two," *Psychol. Rev.*, vol. 63, pp. 81–97, 1956.

[58] P. Milliman and B. Curtis, "An evaluation of modern programming practices in an aerospace environment," in *Proc. 3rd IEEE Digital Avionics Systems Conf.*, 1979.

[59] ——, *A Matched Project Evaluation of Modern Programming Practices* (RADC-TR-80-6, 2 vols.). Griffiss AFB, NY: Rome Air Development Center, 1980.

[60] S. N. Mohanty, "Models and measurements for quality assessment of software," *ACM Computing Surveys*, vol. 11, pp. 251–275, 1979.

[61] D. F. Morrison, *Multivariate Statistical Methods*. New York: McGraw-Hill, 1967.

[62] J. Musa, "The measurement and management of software reliability," this issue, pp. 2031–2043.

[63] G. J. Myers, *Software Reliability*. New York: Wiley, 1976.

[64] ——, "An extension to the cyclomatic measure of program complexity," *SIGPLAN Notices*, vol. 12, no. 10, pp. 61–64, 1977.

[65] ——, *Composite/Structured Design*. New York: Van Nostrand Reinhold, 1978.

[66] J. C. Nunnally, *Psychometric Theory*. New York: McGraw-Hill, 1967.

[67] L. M. Ottenstein, "Quantitative estimates of debugging requirements," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 504–514, 1979.

[68] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, 1972.

[69] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 345–361, 1978.

[70] M. O. Rabin, "Complexity of computations," *Commun. ACM*, vol. 20, pp. 625–633, 1977.

[71] P. Richards and P. Chung, *Localization of Variables: A Measure of Complexity* (Tech. Rep. 75CIS01). Sunnyvale, CA: General Electric, Command and Information Systems, 1975.

[72] H. Sackman, W. J. Erickson, and E. E. Grant, "Exploratory and experimental studies comparing on-line and off-line programming performance," *Commun. ACM*, vol. 11, pp. 3–11, 1968.

[73] N. Schneidewind and H. M. Hoffman, "An experiment in software error data collection and analysis," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 276–286, 1979.

[74] S. B. Sheppard, B. Curtis, P. Milliman, and T. Love, "Modern coding practices and programmer performance," *Comput.*, vol. 12, pp. 41–49, 1979.

[75] S. B. Sheppard, P. Milliman and B. Curtis, *Experimental Evaluation of On-line Program Construction* (Tech. Rep. TR-79-388100-6). Arlington, VA: General Electric, Information Systems Programs, 1979.

[76] B. Shneiderman, "Exploratory experiments in programmer behavior," *Int. J. Comput. Inform. Sci.*, vol. 5, pp. 123–143, 1976.

[77] ——, *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, MA: Winthrop, 1980.

[78] B. Shneiderman and R. R. Mayer, "Syntactic/semantic interaction in programmer behavior: A model and experimental results," *Int. J. Comput. Inform. Sci.*, vol. 8, pp. 219–238, 1979.

[79] M. E. Sime, A. T. Arblaster, and T. R. G. Green, "Reducing programming errors in nested conditionals by prescribing a writing procedure," *Int. J. Man-Machine Studies*, vol. 9, pp. 119–126, 1977.

[80] ——, "Structuring the programmer's task," *J. Occupational Psychol.*, vol. 50, pp. 205–216, 1977.

[81] M. E. Sime, T. R. G. Green, and D. J. Guest, "Psychological evaluation of two conditional constructions used in computer languages," *Int. J. Man-Machine Studies*, vol. 5, pp. 105–113, 1973.

[82] M. E. Sime, T. R. G. Green, and D. J. Guest, "Scope marking in computer conditionals–A psychological evaluation," *Int. J. Man-Machine Studies*, vol. 9, pp. 107–118, 1977.

[83] H. A. Simon, "Information processing models of cognition," in M. R. Rosenzweig and L. W. Porter, Eds. *Annual Review of Psychology*, vol. 30. Palo Alto, CA: Annual Reviews, 1979, pp. 363–396.

[84] S. S. Stevens, "Measurement," in G. M. Maranell, Ed., *Scaling: A Sourcebook for Behavioral Scientists*. Chicago, IL: Aldine, 1974, pp. 22–41.

[85] J. M. Stroud, "The fine structure of psychological time," *N. Y. Acad. Sci. Ann.*, vol. 138, no. 2, pp. 623–631, 1967.

[86] T. Tenny, "Structured programming in Fortran," *Datamation*, vol. 20, no. 7, pp. 110–115, 1974.

[87] W. J. Tracz, "Computer programming and the human thought process," *Software–Practice and Experience*, vol. 9, pp. 127–137, 1979.

[88] W. S. Torgerson, *Theory and Methods of Scaling*. New York: Wiley, 1958.

[89] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, pp. 54–73, 1977.

[90] G. F. Walters, "Applications of metrics to a software quality management (QM) program," in J. D. Cooper and M. J. Fisher, Eds., *Software Quality Management*. New York: Petrocelli, 1979, pp. 143–157.

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

[92] L. M. Weissman, *A Method for Studying the Psychological Complexity of Computer Programs* (Tech. Rep. TR-CSRG-37). Toronto, Ontario, Canada: Univ. of Toronto, Dep. Computer Science, 1974.

[93] W. R. Wolverton, "The cost of developing large scale software," *IEEE Trans. Comput.*, vol. C-23, pp. 615–634, 1974.

[94] M. R. Woodward, M. A. Hennell, and D. Hedley, "A measure of control flow complexity in program text," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 45–50, 1979.

[95] S. S. Yau and J. S. Collofello, "Some stability measures for software maintenance," in *Proc. IEEE COMPSAC '79*, pp. 674–679, 1979.

[96] E. A. Youngs, "Human errors in programming," *Int. J. Man-Machine Studies*, vol. 6, pp. 361–376, 1974.

TUTORIAL SERIES 7

From prototype techniques presently used, new directions are emerging that lead to improved management and earlier detection of errors.

Software Project Verification and Validation

Michael S. Deutsch
Hughes Aircraft Company

Many opportunities exist for injecting human error into the series of activities involved in developing software systems. Errors may occur at the very inception of the process with erroneously or imperfectly specified system objectives as well as later in the process during design and development, when these objectives are mechanized. The basic goal for software in terms of quality is that it perform its functions in the manner intended by its architects. To achieve this goal, the final product must implement the architects' intentions with a minimum of mistakes as well as remain free of misconceptions regarding those intentions.

Because humans lack the ability to perform and communicate with perfection, software development is accompanied by a verification and validation activity. The importance of this activity to the software project continues to grow as data processing enters more areas, such as health and transportation, where software failures could have catastrophic results.

The terms "verification" and "validation" are used extensively throughout the literature of the software community with various definitions expressed or implied. A general understanding of these terms emerges from a survey of usage. *Verification* refers to an activity which assures that the results of each successive step in a software development cycle correctly encompass the intentions of the previous step. *Validation* has a more global connotation. Its goal is to establish that a product functions and contains the features prescribed by its requirements specification.

During the past 15 or 20 years, software projects have developed from rather small tasks involving a few people to enormously complex undertakings requiring the services of many individuals. Likewise, the concept and practice of verification and validation have changed. Previously, verification and validation was an informal, highly individualized activity, mostly consisting of the programmer exercising his code against a small set of ad hoc test cases. As the volume and complexity of software increased, the fallibilities of this "craftsman" approach became obvious in the increasing number of unreliable products. Clearly, to fully entrust the product developer with the responsibility for product verification created conflicting and overwhelming intellectual demands. As a result, verification and validation emerged as an activity requiring individual emphasis and technology within the software project.

Verification and validation technology, as it is applied to today's complex software projects, has three characteristics that sharply contrast with original practices. First of all, verification and validation methodologies are now applied over the entire software life cycle before software testing begins. These pretesting activities typically include requirements verification, design walk-throughs, design quality measurements, interface control definition, code inspections, and formal design reviews. This broad approach was inspired by the realization that mistakes discovered early in the life cycle are less costly to correct.

Secondly, the concept of testing has undergone significant evolutionary change. It has become a highly rigorous function involving formalized plans and procedures. The use of automated test tools has taken some error-prone operations out of human hands.

Thirdly, formal testing is now performed by an ITO—that is, an independent test organization—which reports

Adapted by permission of Prentice-Hall, Inc., Englewood Cliffs, New Jersey, from "Verification and Validation," Michael S. Deutsch, in *Software Engineering*, Randall W. Jensen and Charles C. Tonies, © 1979.

0018-9162/81/0400-0054$00.75 © 1981 IEEE

directly to the project manager. The ITO normally occupies a position in the project organization hierarchy on the same level as the software development organization. On very large projects, the customer may employ a verification and validation contractor who is independent of the project organization.

Although much maligned in recent years, testing remains the basic pragmatic verification and validation mechanism on the software project, despite certain deficiencies regarding formal verification and validation. This article emphasizes modern testing approaches which have ameliorated many of the earlier deficiencies that resulted in unreliable software products. The presentation begins with a rigorous software construction and test approach which validates that the software end item satisfies the specified requirements and verifies that the end item performs according to the design. The next section examines automated testing through an example application of an automated test tool and then considers the projected effect of automated testing on life cycle costs. The third section summarizes verification and validation activities over the entire software life cycle and leads into an exploration of future trends.

Control of software construction and test

We will examine a method of software construction and test that

- segments a complex software development into more manageable functional elements,
- demonstrates key functional capabilities early in the testing activity,
- maintains a visible connection between testing and software requirements, thus formalizing the testing process, and
- forms the basis of a very effective project planning and control strategy.

The approach described here was originally developed at Computer Sciences Corporation[1] and has been refined in its application at Hughes Aircraft Company. Because of its combined technical and management merits, this concept has a powerful effect on ordering the software construction/test process.

Basic approach. Figure 1 depicts a software test procedure that is visibly connected to the requirements. The process is driven by a tool called the *system verification diagram* or SVD derived directly from the software requirements specification. The SVD consists of stimulus/response elements called *threads*[2] that are associated with an identifiable function and specific requirements. Each stimulus/response element can be mapped into the design to identify the specific software modules which, when executed, perform the function of that thread. Thus, a highly granular relationship exists between the functional requirements (the threads) and the design (the modules), becoming the primary means by which the software development can be accurately defined and closely controlled.[3]

The next step is to allocate specific threads to specific builds. A *build* represents a significant partial functional capability of the system.[4] Each incremental build demonstration is a partial dry run of the final acceptance test. Each successive build demonstration is a regression test of the capabilities of the previous builds. Thus, a demonstration of the full system naturally follows from the integration of the final build into the accumulated previous builds.

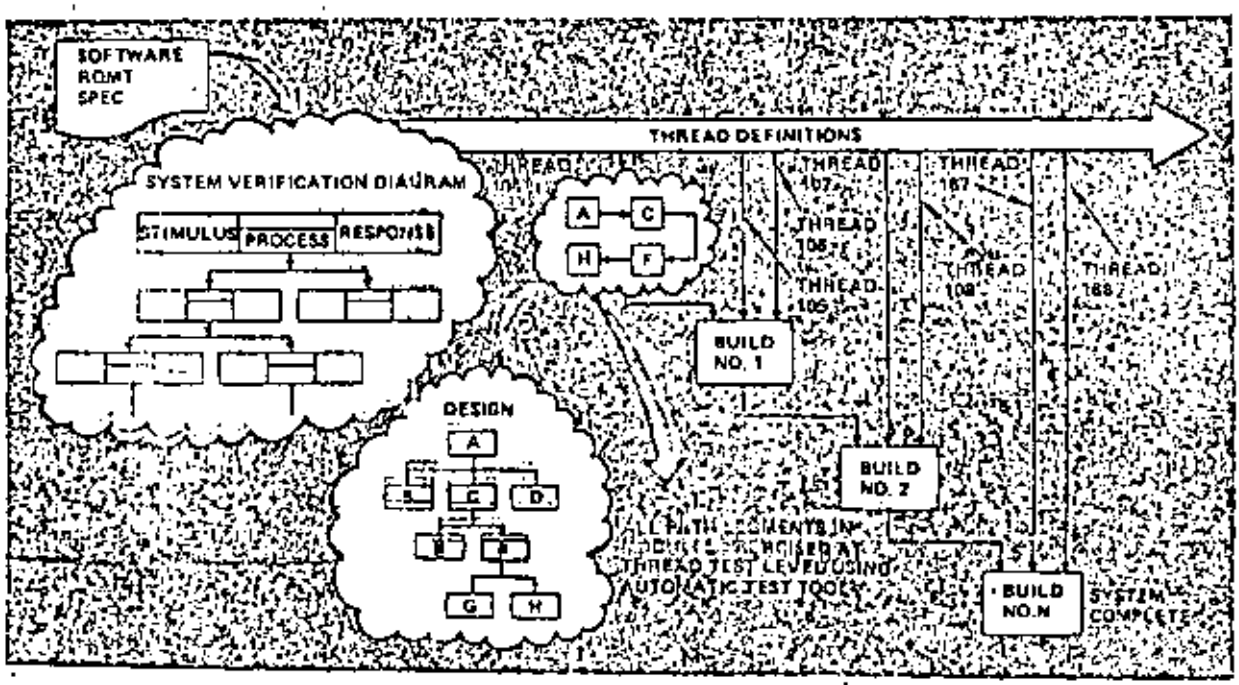


Figure 1. Software test/construction procedure visibly connected to requirements.

Defining the threads. The objective of the system verification diagram—the tool used for defining the threads—is to represent the software requirements in a complete, consistent, and testable manner. Since the SVD represents these requirements as a series of stimulus/response pairings, inconsistencies, redundancies, and omissions may be revealed while developing these pairings. Immediately drafting an SVD for each release of the soft-

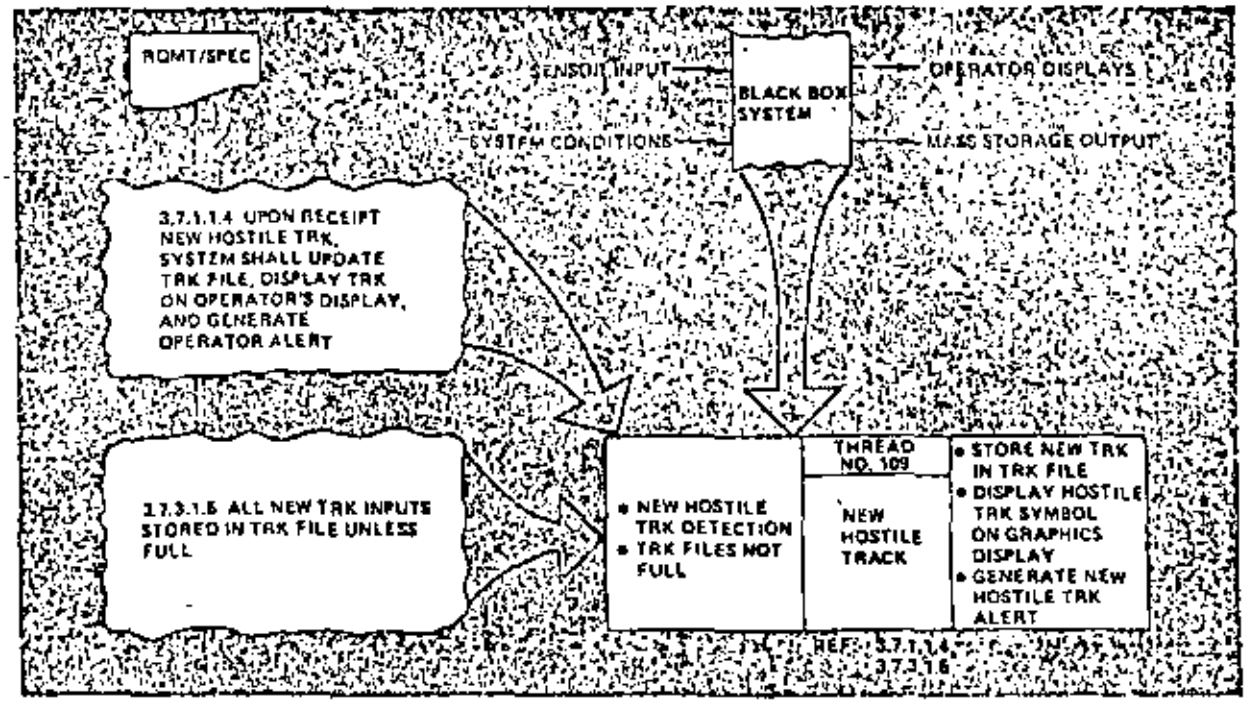


Figure 2. Identifying a thread from the requirements. Reprinted from "The Control of a Software Test Process," by Robert Carey and Marc Bendick, *Proceedings COMPSAC 77*, p. 328. Copyright © 1977 by The Institute of Electrical and Electronics Engineers, Inc.

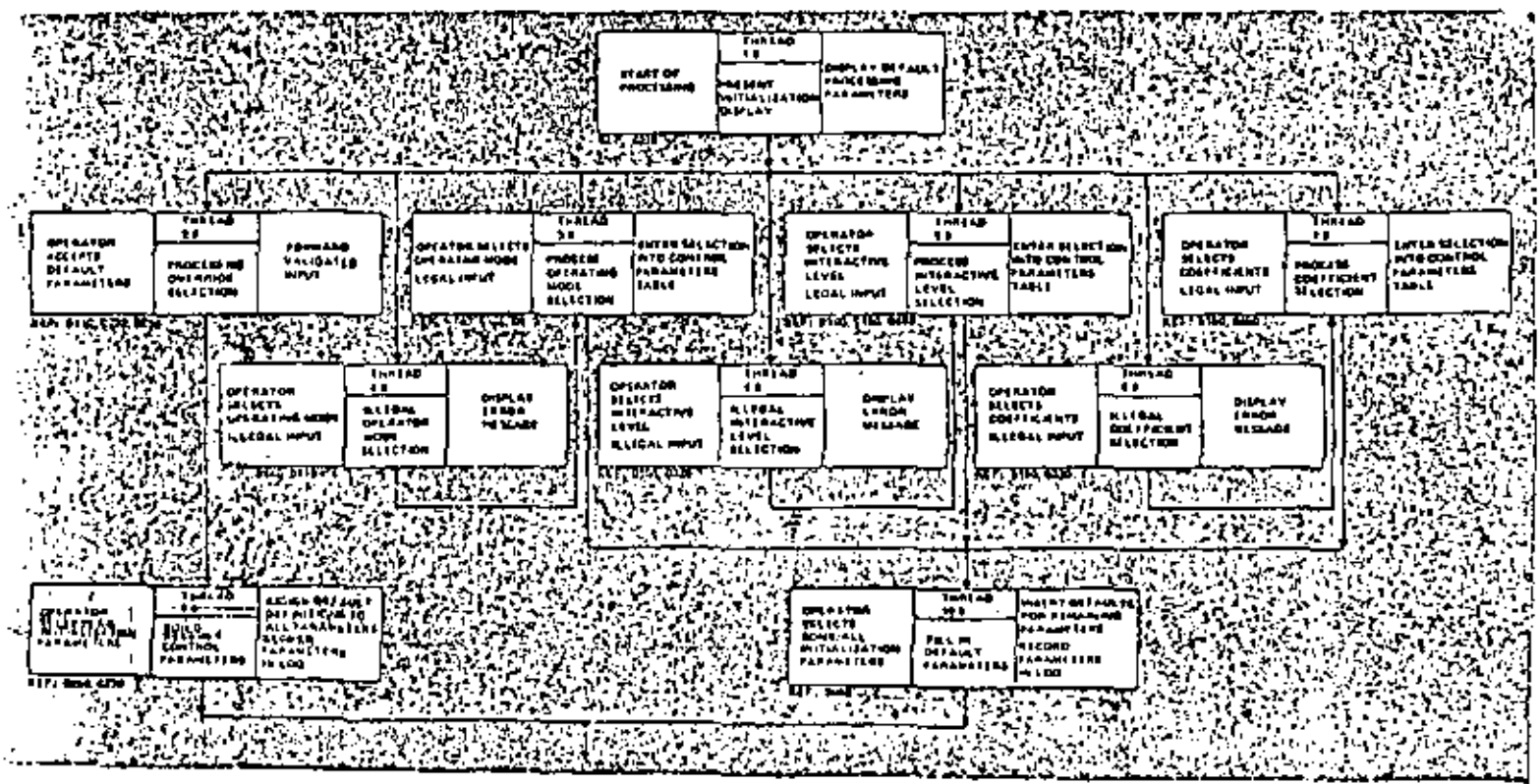


Figure 3. System verification diagram specimen.

ware requirements specification is an effective approach, providing maximum opportunity for feedback and for incorporating feedback findings into the next release. It may also reveal a need to consult the customer about heretofore unrecognized open requirements issues. Thus, the SVD is also an informal requirements verification tool.

Each stimulus consists of one or more inputs plus any conditional qualifiers; each response consists of one or more outputs plus conditional qualifiers generated as a result of the input event.[3] Each stimulus/response element or thread represents an identifiable function or micro-function. Each thread has a direct relationship to one or more software requirements (usually several) and can later be associated with the modules in the software design architecture which will implement the thread. The SVD is normally generated from the requirements specification before the software architecture is designed.

Figure 2 shows an example of the derivation of a thread from requirements specifications. An objective illustrated here is the definition of a transfer function which coalesces two or more functionally related requirements which may possibly reside in physically separate areas of the requirements specification. The test procedure which verifies the software modules that implement this thread also validates the requirements associated with the thread.

The entire requirements specification can be represented by a set of threads logically connected to each other by arrows denoting sequence. Quite often, a conditional qualifier associated with the input event of a thread is the successful completion of the function represented by the previous thread in the sequence. Complete functional testing of the software is attained if all paths through the SVD are traversed during the test program.[4] Figure 3 shows a portion of an SVD used for a recent software development.

Planning the test and buildup process. Detailed test planning can begin when the software architecture defining the structure of the modules is available. In software developments sponsored by the Department of Defense, this normally occurs by the time of the preliminary design review. This test/construction approach requires a modular software architecture developed from structured design precepts; otherwise, the positive effect is diluted. Experience shows that the technique works well with a structure granularity of approximately 100 HOL source lines per module, which may be subdivided later into subroutines.

In this approach, software test and construction are intertwined; they do not occur separately and sequentially. The order in which the software is coded, tested, and synthesized is essentially determined by the system verification diagram which defines the test procedure. The SVD has segmented the system into demonstrable functions called threads. The development of the threads is calendarized. The modules associated with each thread are coded and tested in an order commensurate with this calendarization. The threads are synthesized into higher order sections called builds; each build incrementally demonstrates a significant partial functional capability of the system. This process culminates in a demonstration of the full system with the integration of the last build to the accumulation of builds.

The thread functional allocation chart shown in Figure 4 is a convenient mechanism for recording the relationships among the thread, the requirements, and the modules implementing the thread. An estimation on a linear scale from 1 to 5 of the person-effort required to code, check out, test, and integrate each thread is entered in the column labeled "complexity units." A rating of "5" indicates the most complex and "1" the simplest. This subjective measure is based on the volume of code to be constructed, code complexity, and complexity of the interface with other threads already integrated. Application at Hughes Aircraft Company largely supports the Computer Science Corporation experience that this estimation parameter is useful and linear.[3]

If the system has not been completed by the scheduled date, there will still be the nucleus of a system that performs functions which are demonstrable and operationally useful.

Two considerations are paramount during the next step as threads are assigned to builds and completion points are designated within the schedule period:

- Early implementation of those threads comprising the highest technical risk would be desirable for most projects.
- A system that is logically complete, even though not functionally complete, should be the objective as early as possible in the buildup schedule.[5]

This is basically an implement-to-schedule strategy. If, despite the best of efforts, the entire system has not been completed by the scheduled date, there will still be the nucleus of a system that performs functions which are demonstrable and operationally useful.[5] The customer, undoubtedly, will be displeased. However, his displeasure most likely will be far less than it would be had a less organized approach been used to complete the same

Figure 4. Thread functional allocation chart. For a military standard software development, this chart will satisfy the information content requirements of Type C-5 Specification, Paragraph 3.1.1 (Functional Allocation).

volume of code consisting of diverse software units *not* orderly sequenced into operational functions.

Next, the results of the planning effort described above are displayed on a build plan. This diagram provides an overall calendarized view of the sequence of construction/test events, including the sequence of the builds, the relationship of the builds to each other, the allocation of threads to builds, and the sequence of the threads.

In the build plan from a recent software development shown as an example in Figure 5, implementation of an end-to-end data flow is achieved at Build 6.0, which occurs less than halfway into the scheduled implementation period. This constitutes a logically complete system for this project. Builds 7.0 to 15.0 add additional functions which supplement the basic data transformations and provide operator interactive control over the processing sequence. Each build adds new functions (threads) to the system and—to the extent indicated by the connecting arrows on the build plan—contains the cumulative capabilities of previous builds. Each build demonstration tests the new capabilities and is a regression test of the capabilities accumulated from previous builds. The final acceptance test of the total system occurring at Build 15.0 is merely a natural culminating event in this succession. Each thread is also tested informally. The build demonstration tests may be viewed as natural culminating events in a succession of thread tests. Because each thread is directly linked to software modules, the build plan also defines the order of construction for the modules.

Since by their nature threads represent useful operational functions, a measure of thread completion also represents a measure of project completion. Using the number of complexity units assigned to each thread, the build plan may be summarized into a thread production plan shown in Figure 6 for the same software development. The solid line represents the planned production rate against which the actual production rate may be plotted in dashed lines. Thus, the thread production plan provides an easily understood means of reporting actual progress versus planned progress.

Benefits. The thread testing approach we have discussed has a number of benefits. It

- allows testing and analysis in digestible quantities,
- provides early demonstration of key functional capabilities,
- forces the early availability of executable code,
- provides a meaningful measurement of project progress that is easily understood,
- defines module construction sequence and serves as a basis for allocating personnel resources,
- brings a degree of formality to software project verification and validation by validating that the end item satisfies requirements and by verifying that it implements the design, and
- allows mission analysts (due to the analogy between threads/builds and operational functions) to in-

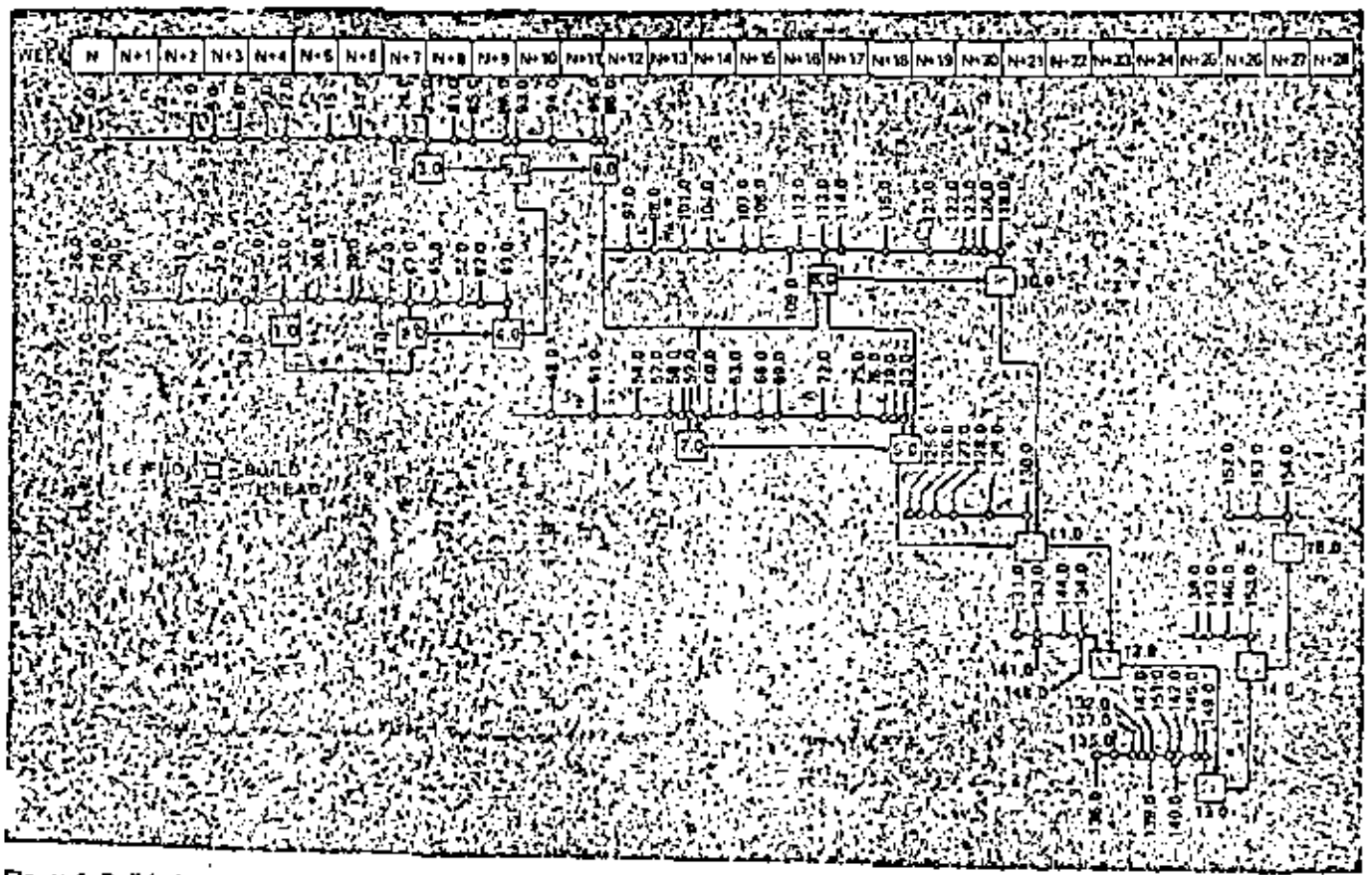


Figure 5. Build plan example.

ternet earlier in the testing process, scrutinizing performance results.

To some extent, our description of this approach has understated the complexity of modern software development projects by addressing a software system as a single development entity. Many software projects are too large and the schedules too short to be managed in this fashion. The software system is decomposed into structural elements that are developed in parallel and are eventually integrated to recompose the system. On military standard software developments, this element is referred to as the CPCI, the computer program configuration item. The software construction and test process described here is an effective approach at both the CPCI and software system levels. Procedures for testing and integration at the system level may be prescribed by defining system-level threads. These high-level threads would be correlated with CPCIs or subCPCIs in lieu of modules.

Automated testing

The basic approach to software construction and testing summarized in Figure 1 prescribes that the testing of each thread include the exercise of all path segments in the associated modules. This is an ambitious goal, perhaps costly to attain. In the discussion that follows, we explore the motivation for this objective and demonstrate a tool which helps to achieve this goal. In addition, from contemporary project experiences we present certain quantitative data which explain the benefit of this approach to software life cycle costs.

As software systems grew to immense proportions in both size and complexity, the effort needed to test these systems grew more than proportionately, requiring a mass application of human resources. This application has been costly and not particularly effective in terms of the reliability produced. Despite expenditures of up to half of software development budgets for testing, a significant number of errors remained in delivered software, often severely deterring normal system operations. This dilemma inspired the development of automatic test tools that assist in the production of effective tests and analyze the test results. In essence, they take much of the time-consuming, mechanical aspects of testing out of human hands.

Automatic test tools provide improved organization of testing through automation, measurement of testing coverage, and improved reliability—attributes not as easily attained by manual testing approaches. Automatic tools furnish machine amplification of human capability and relieve test personnel of routine time-consuming chores. Manual error-checking, for example, is an error-prone process; automated error-checking is more reliable. Budget and schedule considerations foreclose on the slow, tedious process of manual testing with the result that the volume of testing is often insufficient. Relief is available through utilization of automated tools.

The complex content of large software systems requires the application of many test cases to thoroughly exercise the code. Generating these test cases and analyzing the paths exercised to determine the extent of the resulting test coverage would quickly exceed human capacity. Testing aids can instrument the code, measure the coverage

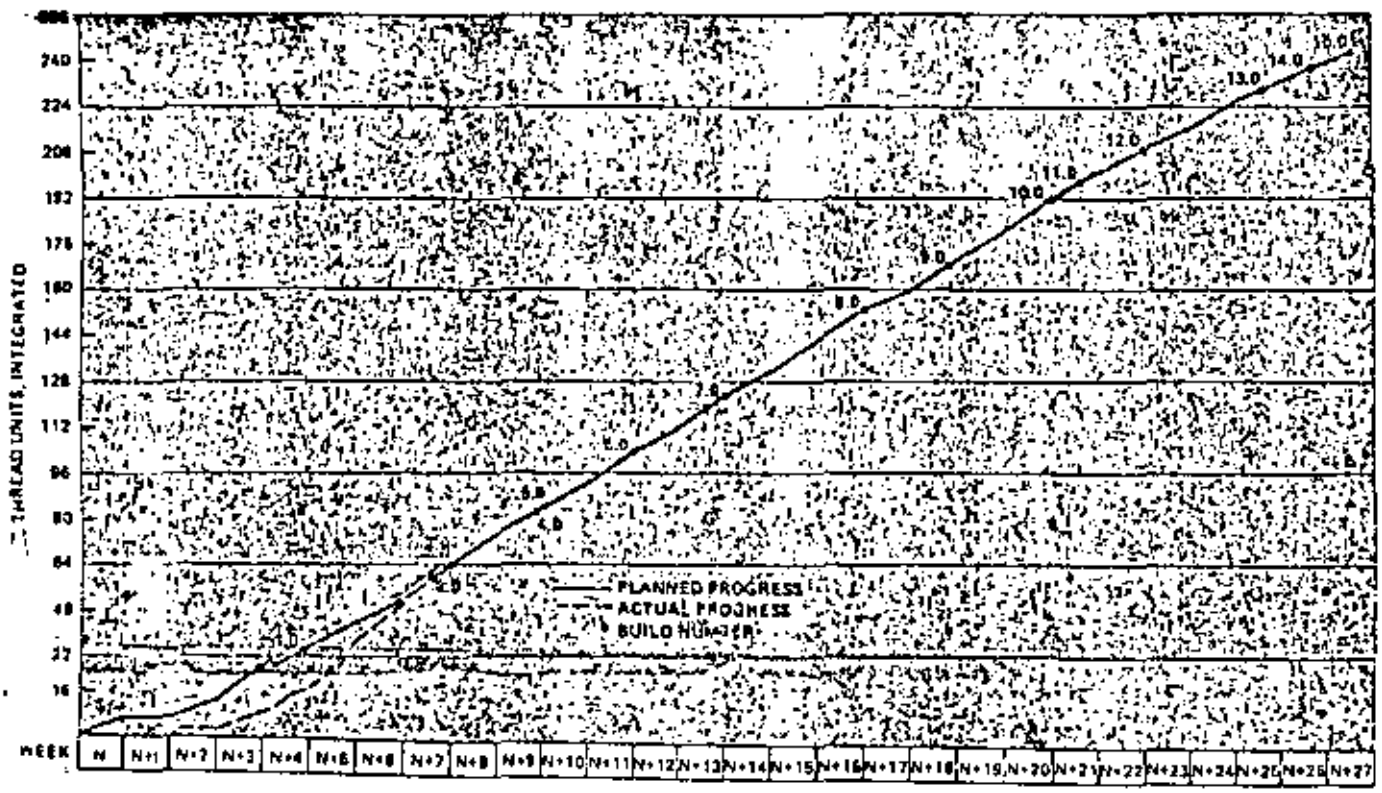


Figure 6. Thread production plan example.

provided by a test case, and furnish a report that shows the number of times each statement and sequence of statements were executed.

Improved reliability results when potential sources and avenues of errors are more closely investigated. This occurs as more experience in the use of the software is accumulated through carefully directed testing. Automatic tools enable a higher volume of testing than would be attainable manually for the same cost.

Almost all aspects of the software development process can be assisted by some form of automation. A spectrum of tools exists in various states of maturity. Particularly advanced, and of demonstrated utility to real-world industrial software projects, is a class of automated test assistance tools called source analyzer systems.

SAS. A source analyzer system, or SAS, provides facilities for measuring the performance of the software and the effectiveness of test cases. Such tools derive their information from the source text of the program being analyzed and tested.

Five basic functions are performed by the SAS:

- analysis of source code and creation of a data base,
- generation of reports, based on static analysis of the source code, that reveal existing or potential problems in the code and identify the software control and data structures,
- insertion of software probes into the source code that permit data collection on code segments executed and values computed and set in storage,
- analysis of test results and generation of reports, and
- generation of test assistance reports to aid in organizing the testing and deriving input sets for particular tests.

The elements of a typical SAS that performs these functions are diagrammed in Figure 7. The *static analysis module* analyzes the static structure of the code. Typically, this analysis would consist of partitioning each routine into path segments which support an evaluation of test thoroughness and analyzing the invocation structure of the program. This information is captured in a data-base file and formatted for output as a printed report.

The *instrumentation module* acts as a preprocessor by inserting additional statements within the original source code. During execution, these statements or "probes" intercept the flow of execution at key points and record program performance statistics and signals in an intermediate file. The instrumented source code is compiled, an executable image is built, and the image is executed.

The *analyzer module* functions as a postprocessor after the program execution. It formats and edits data recorded in an intermediate file during program execution and provides a printed report. This report furnishes information on test coverage, including

- statements executed and frequency, percentage of statements executed,
- statement sequences traversed, percentage of sequences traversed, listing of sequences not traversed,
- data ranges of variables, and
- assertion violations.

The analyzer module may also compare anticipated test-case output augmented by any user-supplied evaluation criteria (prestored in a file) with actual output and list resulting discrepancies.

The *test case assistance module* aids testing personnel in the selection of test inputs that will economically attain comprehensive testing goals. The module uses the static analysis data base and the coverage data recorded during

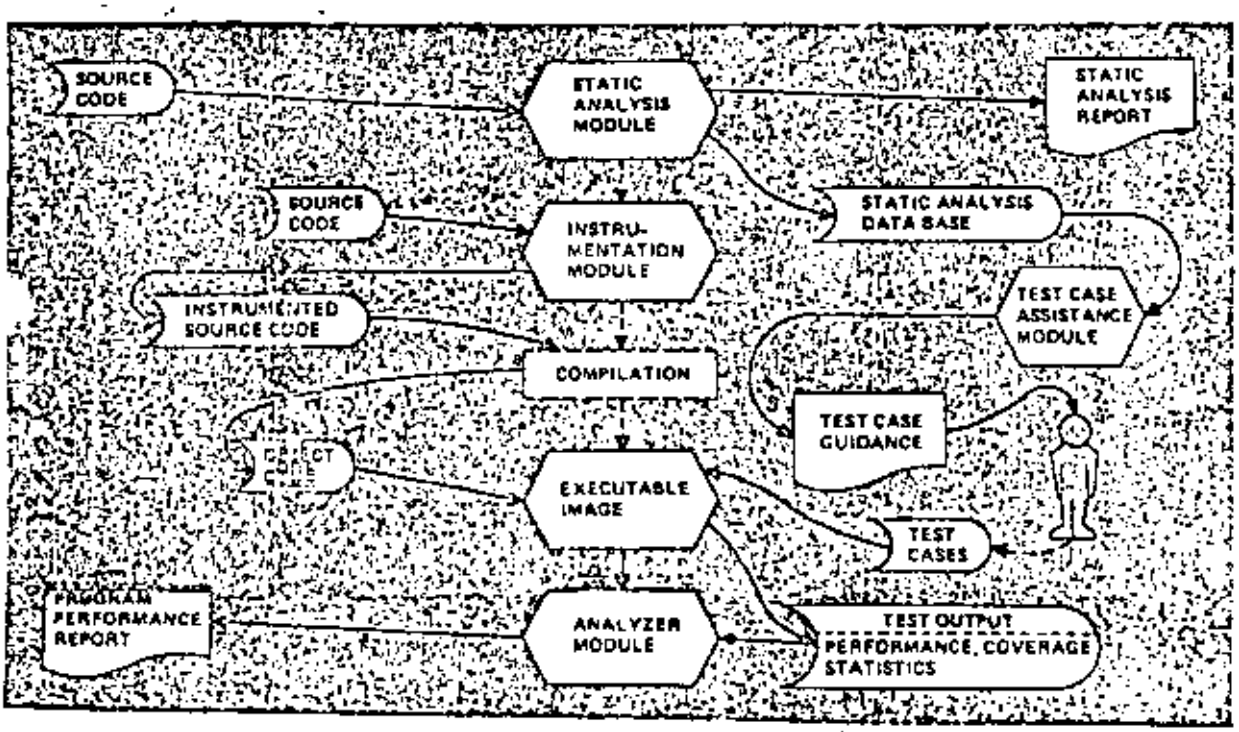


Figure 7. Typical elements of a source analyzer system.

4-2-7

executions to guide the testers in the preparation of additional test cases that exercise paths in the program not executed by previous test cases. An algorithm detects statements and branches not exercised and indicates conditions necessary to traverse that path.

Typical application. The sequence of events in the software testing procedure with the aid of an SAS is shown in Figure 8 as exemplified by the capabilities of the RXVP system.* The individual boxes define complete steps in the operation, each producing outputs that feed subsequent operations either directly or through updates to the central data base.

The complete operation shown in Figure 8 covers the five SAS functions listed earlier in this discussion. Below, each identified step is described in terms of some of the outputs generated and the purpose of those outputs in the testing process. In the description of these operations, the subroutine SOLVE is the subject of analysis.

*RXVP is a Fortran/Ifran source text analysis system built and marketed by General Research Corporation of Santa Barbara, California.

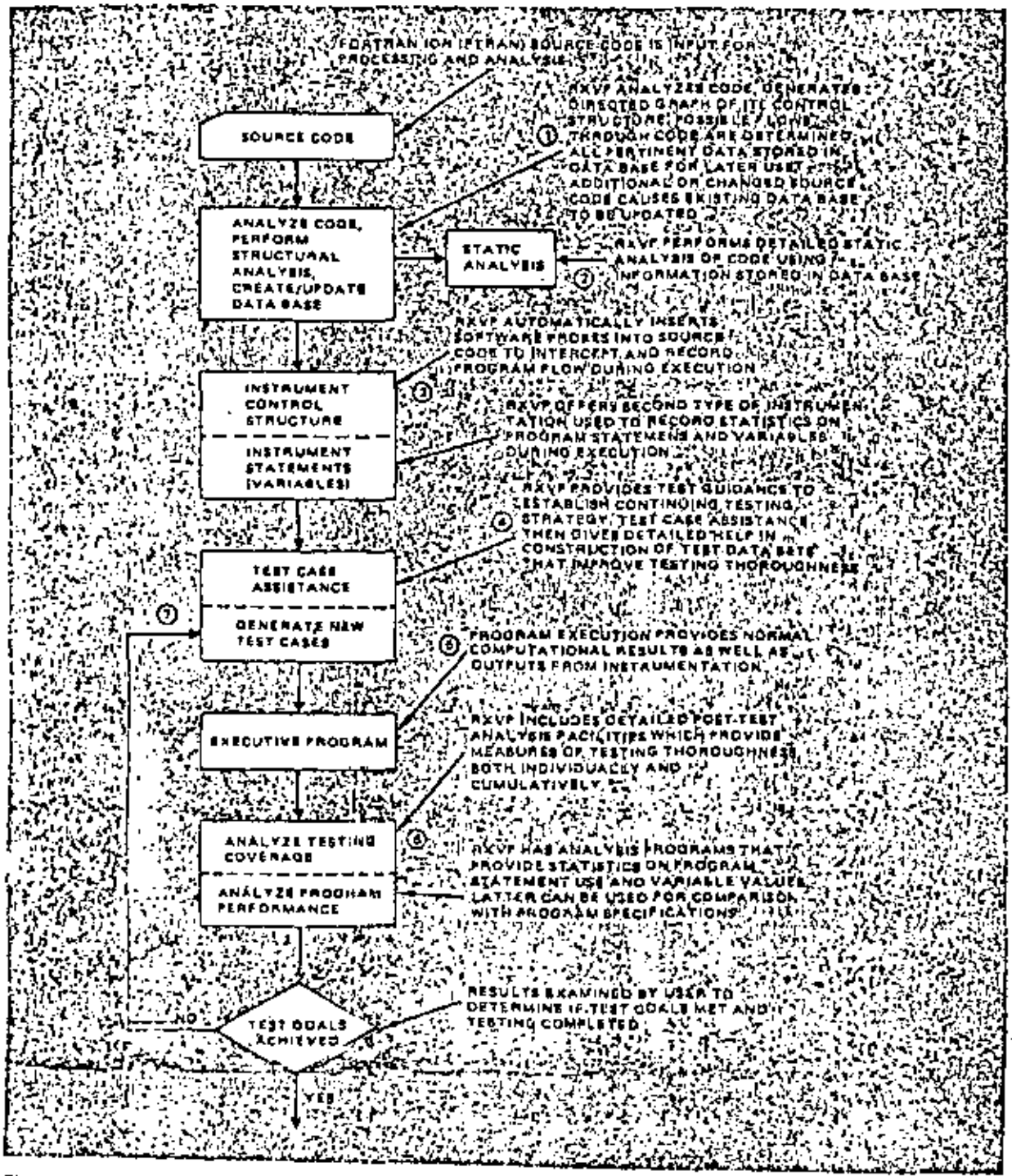


Figure 8. Application of RXVP in software testing.

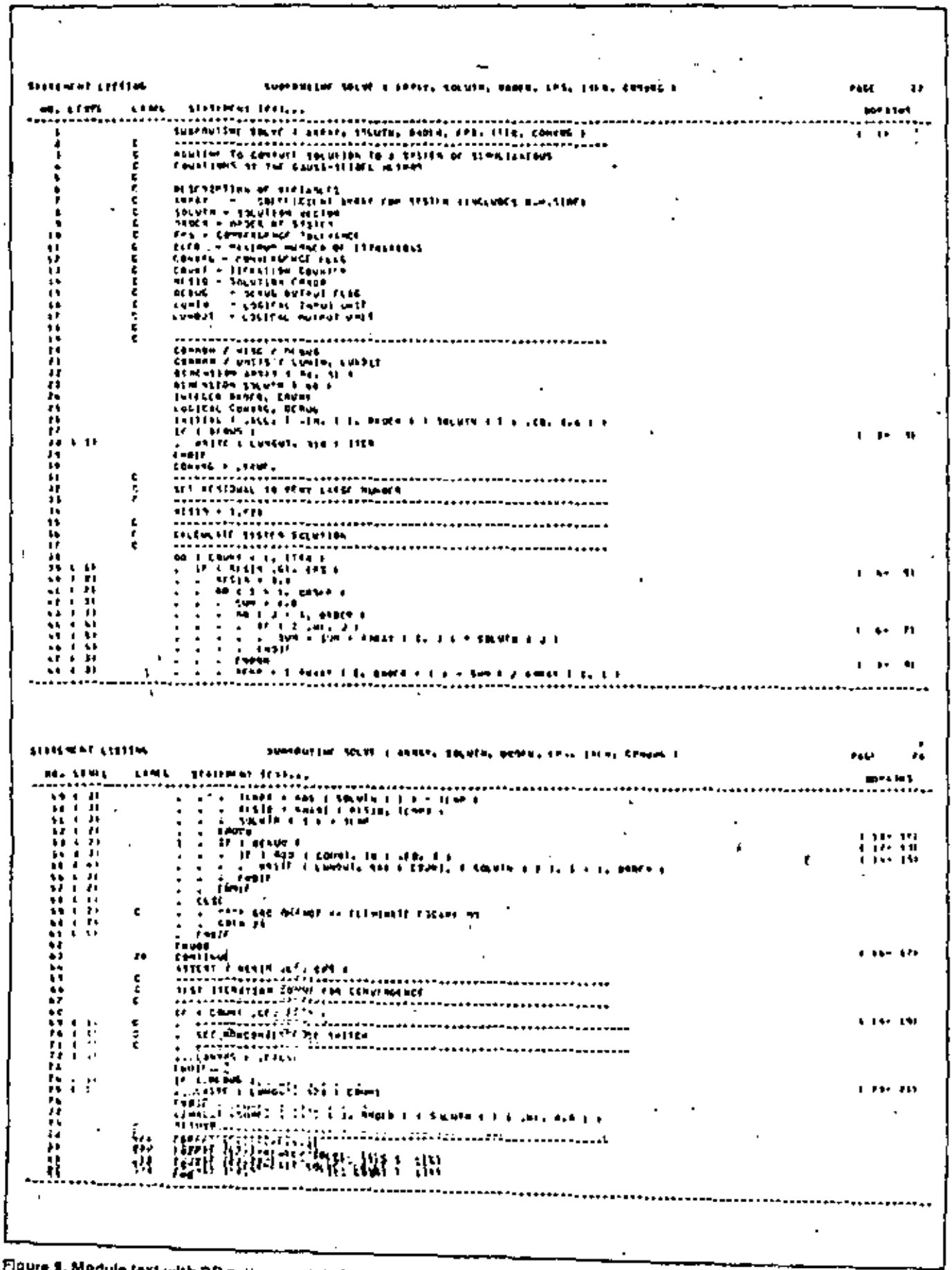


Figure 9. Module text with DD paths annotated.

Step 1: Building the data base. The system reads source code as its input. Simple commands are also supplied as a separate input file to direct the operations.

The system first derives its data base from the source code. Line numbers are assigned to each line of text; the control structure and symbols are identified; and a directed graph model of the control structure is built. The resulting information is stored in the data base that becomes the controlled copy of module text as well as the source of all outputs generated in later steps.

The control structure of a module is defined in terms of the sequence of statements from the outcome of a decision through the next decision. This sequence is immediately executable, once the initial decision outcome has been evaluated, and is the basic logical segment of the control structure. The sequence is called a decision-to-decision path, or DD path.

Step 2: Producing documentation reports. After the data base is created, the documentation reports are generated. These reports provide information that will be used to document the code and to aid the test group in understanding code organization. Included are several reports delineating the invocation hierarchy of the modules and the module text.

Figure 9 shows the module text. The text of a module as stored on the data base is the source of this report. The leftmost column contains the statement number. Adjacent to it, in parentheses, is a nesting level number that identifies nesting depth for the structured languages. Next, the statement is printed, indented according to nesting level. For each statement that starts a DD path, the path number is printed in the rightmost column. This module text printout is the basic module description report that gives the statement numbers referred to in other reports, lists the full module text, and identifies all DD paths.

Step 3: Module instrumentation. Collection of program flow statistics is facilitated by the instrumentation of the program control structure. The system examines the code and automatically inserts a call to a data collection routine that is invoked each time a control branch is taken. When the instrumented code is executed, the collection routine notes the module and code section executed and builds up a data file from which the test analyzer generates its reports. Typical controls for this step allow selection of modules and define the level of instrumentation. RXVP has the option of instrumenting either at the module entries only or at each control branch point, including module entries.

If more detailed performance statistics are desired, the next level of instrumentation provides data collection calls for each assignment statement, permitting a complete record of the values stored during execution. This level of instrumentation would normally be applied only to selected modules where the more detailed information is needed to ensure that testing covers a proper range of computational results.

Step 4: Test assistance. At this point, testing is ready for a set of test cases. It is assumed that some form of prior testing has produced a set of working test cases. At the first cycle of testing, these existing cases are executed with the instrumented code. After these initial tests are analyzed and the reports examined, this step will entail generation of new or modified test cases as needed to achieve testing goals.

Step 5: Test execution. While the instrumented code is running, it produces its normal outputs and invokes the data collection routine. In its simplest form, the data collection routine merely writes a trace file that records the sequence of DD paths executed or the sequence of variable values assigned. More extensive data collection routines can perform some analysis and summarizing during execution to reduce the amount of data collected. RXVP has three options: variable trace, DD-path trace, and DD-path summary by counting executions and recording only the counters on the trace file.

Step 6: Test analysis. The function of the posttest analyzer is to supply the reports that can be evaluated against testing goals. DD-path coverage analysis reports execution counts by module and DD path as well as summary data on testing progress. The variable trace analyzer reports the maximum, minimum, first, last, and average value of all variables traced.

The top-level view of testing progress is depicted on the *coverage summary* report (see Figure 10). For each module, the number of DD paths, the number of invocations, the number of DD paths traversed, and the percent

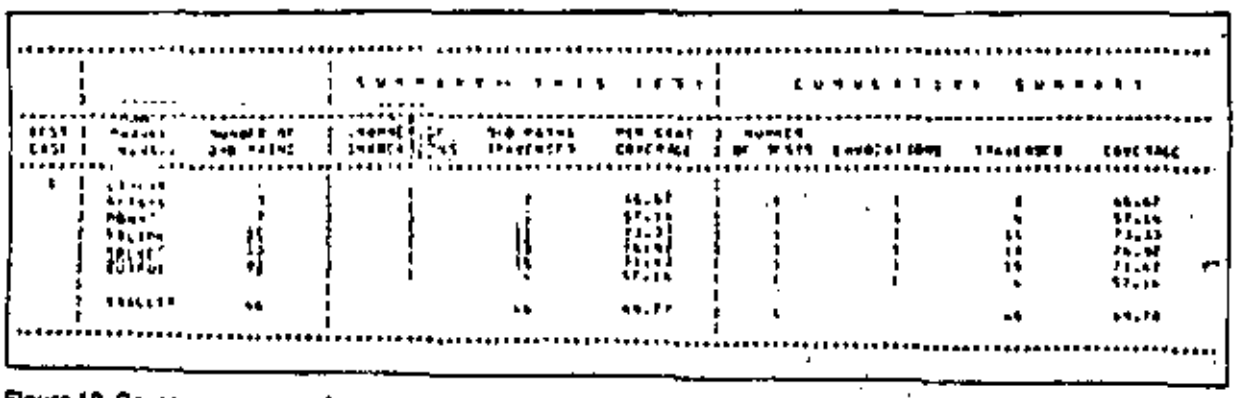


Figure 10. Coverage summary for one test case.

coverage are shown. These parameters are also depicted for the accumulation of all modules. This information is presented for each test case and for the accumulation of all test cases.

The *DD paths not hit* report (see Figure 11) tells the tester which DD paths were not executed by each test case and which remain untested after all tests are accumulated. The report also displays single-test and cumulative test information. This information guides the tester regarding potential targets for the next test case.

The *DD-path execution counts* report (see Figure 12) gives the finest detail on DD-path execution for each test case. For each DD path, an entry is made under "not executed" when execution count is zero. For execution counts greater than zero, the bar represents the count as a percent of the maximum number of executions for a single path. The exact number of path executions is entered under "number of executions." In addition to the module summary line at the bottom, this report shows where most of the execution time is being spent in a module. The report also graphically shows where there are large segments of code not being tested.

The more detailed variable trace analyzer produces a report, *performance test analysis*, which is keyed directly to the source statements. Three types of reports are given. If the statement has no value computed (e.g., a CONTINUE, WRITE, READ, CALL, or assignment of a constant), then only the execution count is printed. Statements that test logical expressions report times true, times false, and final value of the logical expression. Statements that compute a numeric value (e.g., assignment, arithmetic IF, or DO) report initial, final, maximum, and minimum values.

Step 7: Test assistance. Figure 11 shows that a number of DD paths remain untested. The objective of continued

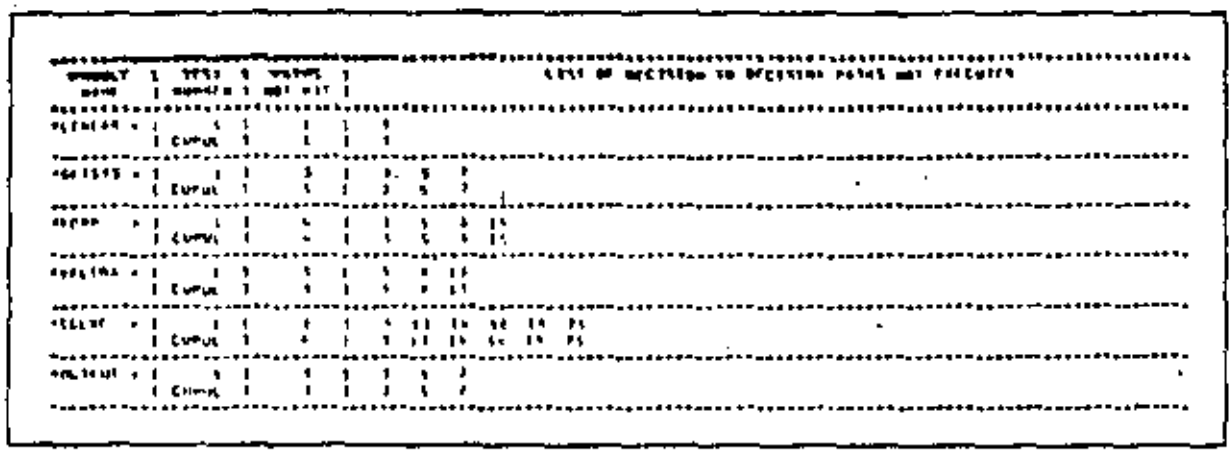


Figure 11. DD paths not hit (one test case).

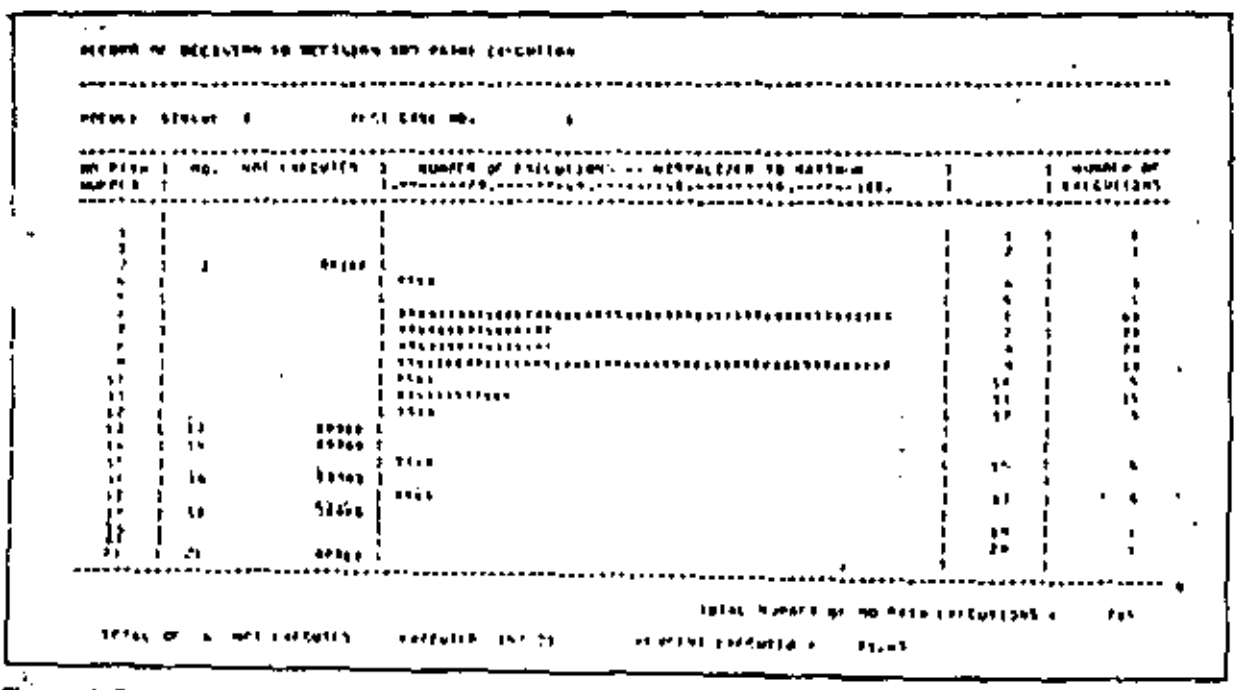


Figure 12. DD paths execution counts.

4-2-11

testing is to generate test cases that will include at least one execution of the untested DD paths. This testing goal should be considered a minimum level of testing in which all statements are executed at least once and each decision is exercised through all possible outcomes. Although this is far from a complete test that exercises all logical paths through the program, it is more comprehensive than is generally achieved by tests that sample only selected values in the input space. Furthermore, the process of generating new test cases focuses attention on the source code of the program and frequently reveals errors that would go undetected if only execution results were examined.

The first portion of test assistance draws on the *predicate analysis*. A "predicate" is a decision element statement. The predicate analysis lists the decision statements for the untested DD paths and gives information about the variables appearing in the predicate (see Figure 13). From this report we see that three of the paths are controlled by the variable DEBUG which is read in as an input in the module LINEAR (see Figure 14).

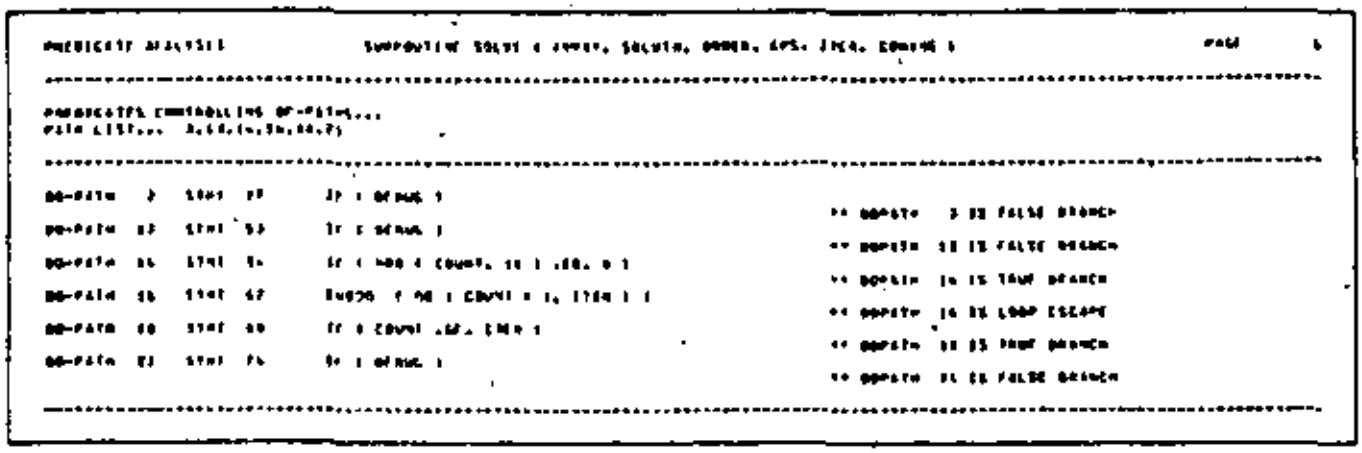


Figure 13. Predicate analysis for SOLVE.

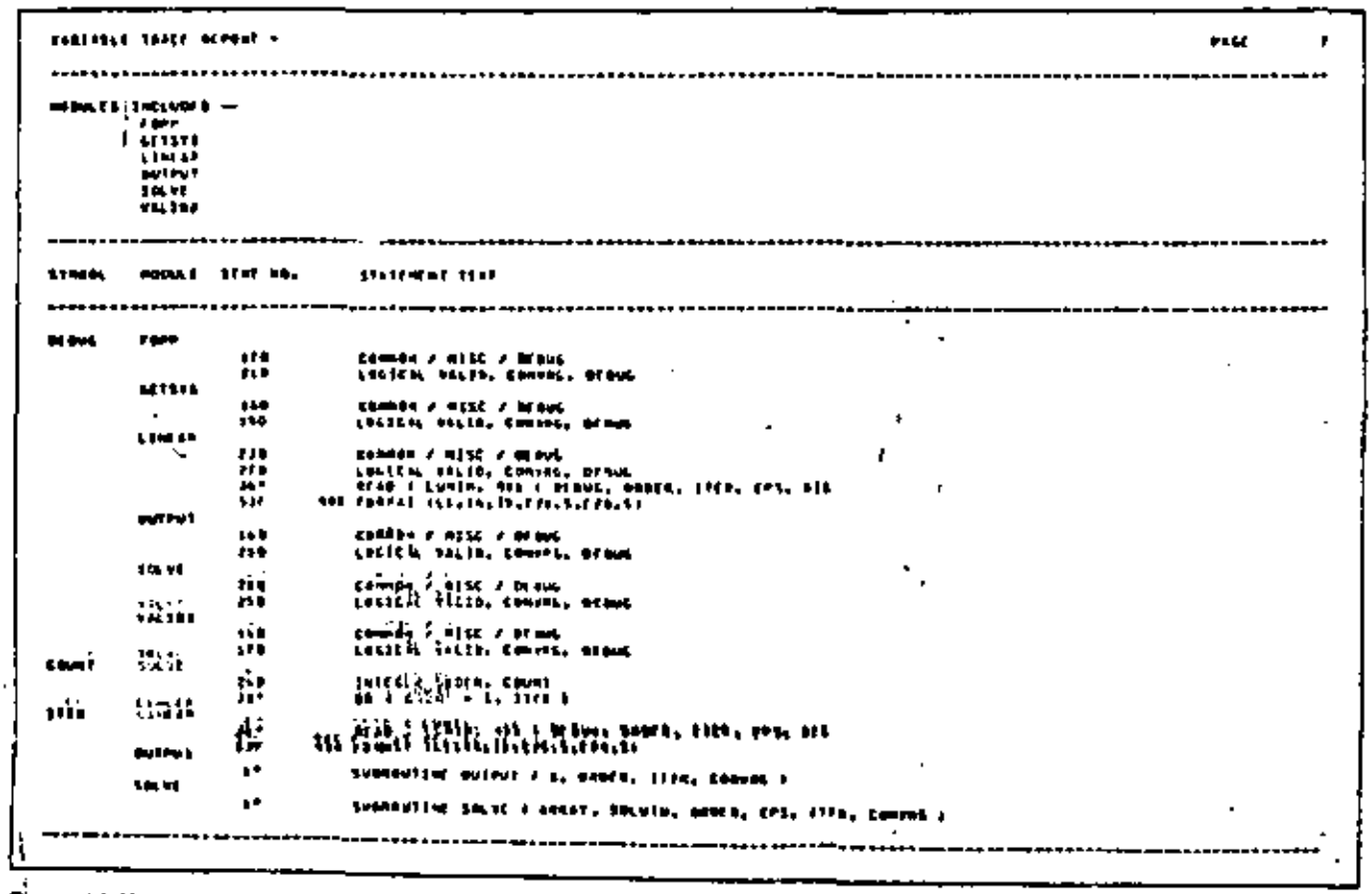


Figure 14. Variable trace for untested predicates of SOLVE.

Therefore, any test case can be simply changed to make DEBUG false.

We also see from this report that the other three missed paths are controlled by the variables COUNT and ITER. COUNT is a local variable in a module that is the loop variable of the DO loop. Thus, Path 16 will be reached if the DO loop exit is reached (as will Path 18). These conditions are achieved by a failure to converge within the allowable limits of ITER. To reach these paths, either a nonconvergent case must be supplied or the value of ITER reduced. ITER is also read in the module LINEAR and can be reduced to a small number by a simple data input change.

Figure 15 shows the summary report as a result of adding a new test case for module SOLVE to our test set. Additional test cases can be contrived by using the methodology we have just described until testing goals are attained.

Automated test tools are currently gaining widespread use, acceptance, and recognition as cost-effective devices providing the ability to exhaustively test software to a level that would be cost-prohibitive with manual means. Achieving the potential benefits of this approach, however, is contingent upon recognizing the proper scope of application. When applied to large amounts of code, this procedure can become overly involved, thus probably neutralizing potential positive effects. Recent project experiences indicate that maximum advantage is attained when this testing approach is applied at the individual thread level, where reasonably small "chunks" of code (100-300 source lines) are involved. Indeed, this scope of application is consistent with the overall objective of uncovering as many errors as possible early in the implementation period, rather than risk encountering these errors at higher levels of integration.

Effect on life cycle costs. It is possible to examine the quantitative benefits to software life cycle costs resulting from the test strategy described above. The expected benefit model is based on an accounting of additional errors detected from a recent software project at Hughes Aircraft Company which utilized this approach and on a profile of the escalating relative cost to correct errors when they are discovered later in the life cycle.

The general influence of this exhaustive testing strategy and its projected influence on life cycle costs for the Hughes project is depicted in Figure 16. This figure compares the effects of automatic tool testing and the standard test approach. The use of the automatic test tool results in higher error detection rates during the construct period. However, during subsequent periods including operations, the detection rate of latent errors remains higher when the test tool is not used. The area under both curves—i.e., the total number of errors—should be approximately equal. Of course, the cost of rectifying the same error later in the life cycle is higher than it would be had it been detected earlier.

The exhaustive testing approach on the Hughes software project consisted, first, of checking out each thread using a set of functionally oriented test cases. The traditional brand of testing would conclude at this point. Instead, additional test cases were contrived to achieve a DD-path coverage of close to 100 percent. The expectation was that this extra effort would expose additional errors.

The actual project experience supported this scenario. Because of these clearly demarcated testing phases, it was a simple matter to maintain a count of the errors detected by the extended testing. The accounting showed that an average of one additional error per thread was uncovered. The cost, in schedule time, of performing the extended testing ranged from one-half day to three days per thread; the average schedule cost was close to one day per thread. Normally, two persons were involved with the testing of the thread at this point. Hence, the incremental cost of the exhaustive testing effort was an average of two person-days per thread.

This project consists of about 400 threads. The incremental cost of finding and correcting the 400 addi-

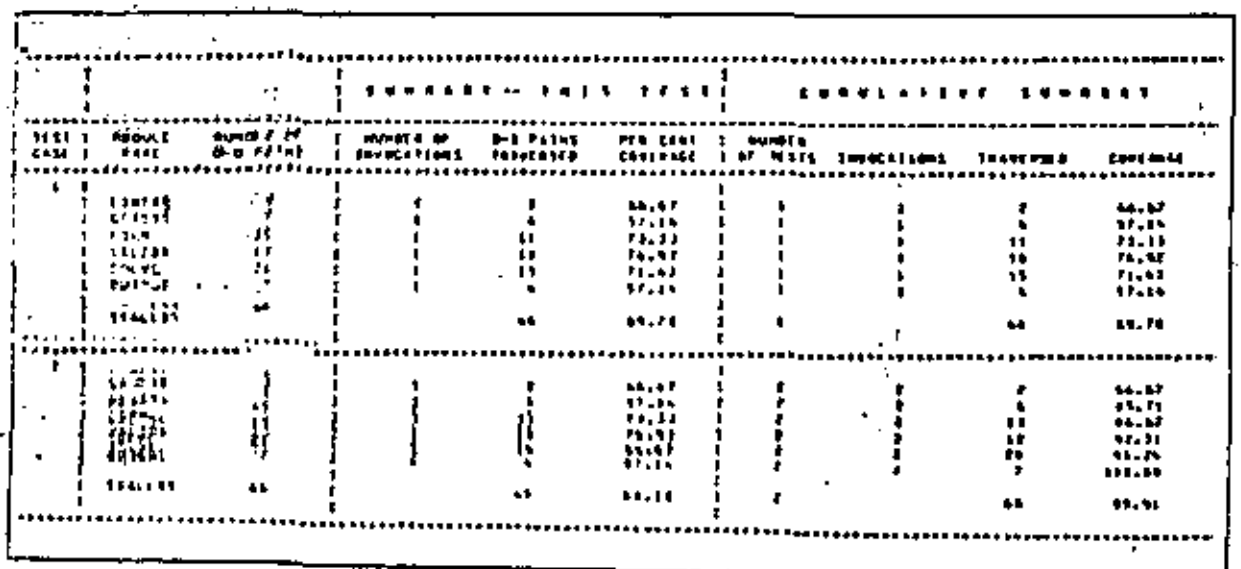


Figure 15. Summary report after additional test cases.

tional exposed errors is 800 person-days (400 errors × 2 person-day/error).

In a survey performed at TRW[9] which seems to have become an industry standard, a profile was composed to depict the relative cost of fixing errors as a function of the life cycle stage in which they are discovered. This profile was compiled from a number of software projects at TRW, IBM, and General Telephone and Electronics. According to this profile, the average relative cost to fix errors during the integration activity versus the construct activity is 4. On the other hand, the average relative cost to fix errors during the operations phase versus the construct activity is 9.

To model the life cycle cost benefits, we will assume that the discovery of these 400 errors would have otherwise been evenly distributed over the integration and operations periods. This is probably a conservative assumption since the type of errors overlooked during the construct activity are more likely to reappear in operations where the software would undergo its first thorough exercise. The 200 errors found during integration would cost 1600 person-days (200 errors × 2 person-days/error × 4) to correct under this model; the 200 errors found during operations would cost 3600 person-days (200 errors × 2 person-days/error × 9) to correct. Thus, the differential savings in life cycle cost achieved by the exhaustive testing strategy is speculated to be 4400 person-days (3600 + 1600 = 5200 person-days less 800 person-days) for this particular project.

Verification and validation over the entire life cycle

Verification and validation on modern software projects involves activities over the entire software life cycle, not just during testing. The merits of this practice include a more reliable product, lower life cycle costs, early exposure of requirements inconsistencies, and early exposure of design errors. Statistics have been compiled that depict the escalating costs to correct errors as later stages of the development cycle are traversed.[11] This profile reveals that it is 10 to 100 times more costly to correct an error after the software is operational than it would be if the error had been detected during the preliminary design period.

The responsibility for verification and validation is distributed among the three technical line organizations which comprise the software project—system engineering, software development, and the independent test organization.

System engineering is normally the smallest of the three groups. It is responsible for prescribing the overall architecture of the system, which includes definition of the system-level requirements, the system-level design, and the requirements for each computer program configuration item. Focusing the responsibility for definition and maintenance of a system's external attributes (requirements) in a single organization assures that the system architecture reflects a single philosophy and a

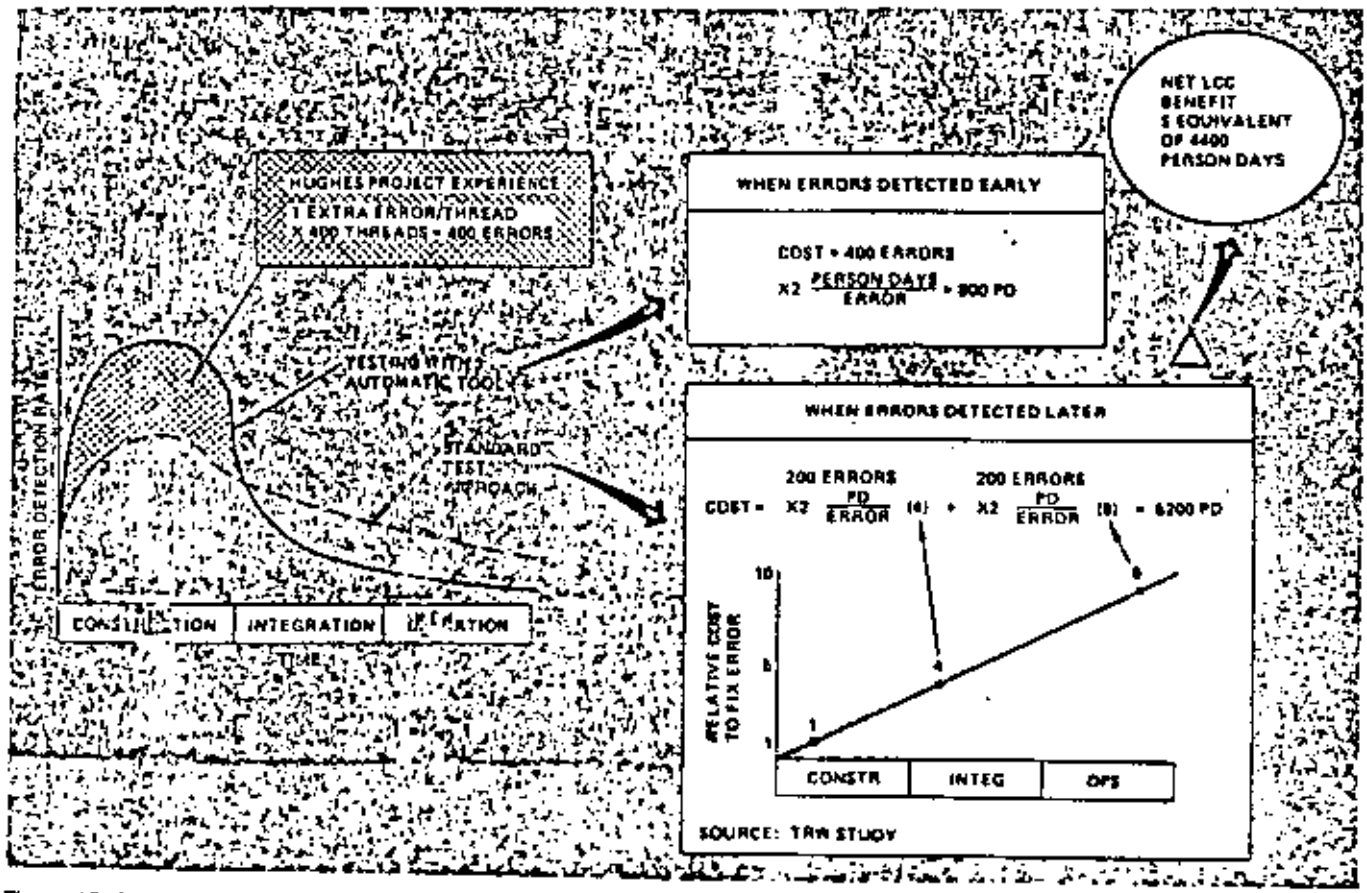


Figure 16. Automatic test tool cost-effectiveness in life cycle.

unified set of concepts. System engineering defines the system architecture at an abstract level and does not prescribe a specific implementation.

Software development designs and implements the software product in accordance with the intentions of the architects. On large projects, software development may be divided into several or many groups. Each group develops a "chunk" of the system, usually a CPCI. At the conclusion of thread/build testing, the development organization hands the software product over to the independent test organization for formal qualification testing.

Maintaining a healthy adversary relationship with software development, the independent test organization scrutinizes (by testing) the software product and identifies discrepancies where requirements have not been implemented correctly. The discrepancies are referred back to the developer for rectification. The independent test organization also integrates the individual CPCIs to form a unified system product.

Figure 17 summarizes the verification and validation activities which occur over the software life cycle, showing a typical life cycle and allocation to the three performing organizations. Small projects will involve a simpler, more compressed set of activities. Very large projects, on the other hand, will have a more complex succession of life cycle events. Nevertheless, all three organizations would be at least minimally involved with verification and validation at each life cycle stage. The intended effect is that there be a set of checks and balances among the organizations who work to produce a reliable software system product.

Future trends

The description in the foregoing sections of pragmatic verification and validation techniques presently in use on modern software projects suggests some trends for the future. Directions and advances seem to concentrate in three general categories—improved management approaches, improved implementation of proven technologies, and evolution of current experimental techniques.

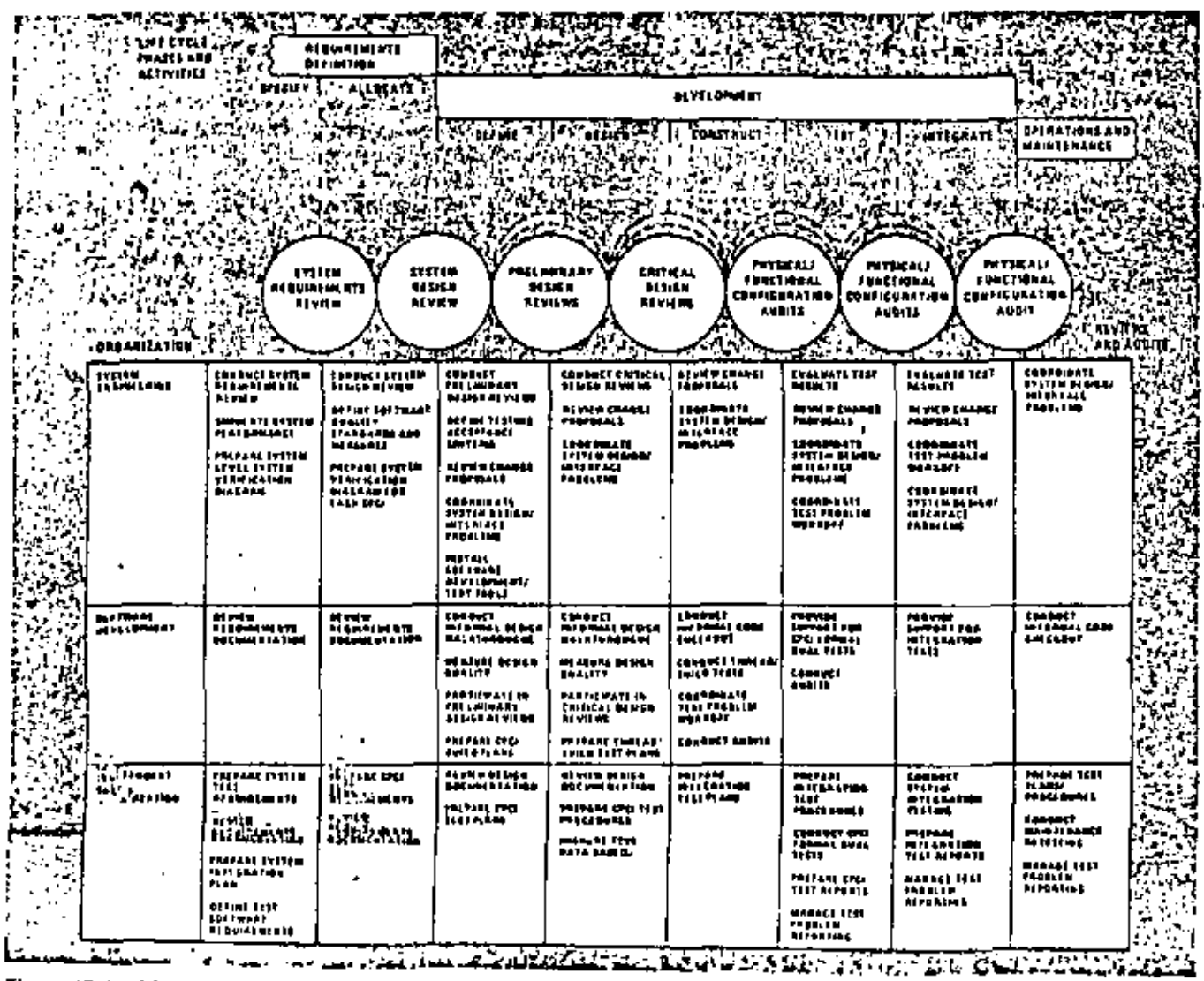


Figure 17. Verification and validation activities by organization over software life cycle.

These areas, discussed individually below, are not necessarily discrete.

During the historical development of the software project, the need developed for separating system engineering, software development, and testing by an independent organization. Unfortunately, in many instances, this separation resulted in the use of separate approaches and tools with marginal coordination. Economics and the resulting performance no longer support this arrangement.

A revised, more realistic approach to software development requires an overall coordinated package of tools and methodologies covering all phases of the software life cycle. Of course, more of a burden will fall on management to achieve these results. What is needed is the continuous evolution of a corps of software managers who are well-educated in the elements of "software physics," who are committed to modern methodologies, and who will be aggressive in overcoming inertia while assuring that these approaches are implemented.

From the compulsion to provide individual attention to them, verification and validation have become too detached from other activities on the software project. Management must review this separation and provide a more integrated role for verification and validation within each software engineering area, while maintaining the separate test organization.

The software development process is more accurately viewed when the technical engineering problem is considered in conjunction with the human engineering aspect of the process. The application of talented personnel should, theoretically, be a sufficient requisite on which to forecast success. However, basic human fallibilities open loopholes for errors to occur. Engineering, scientific, and management practices are frequently corrupted by human errors of inconsistency and omission. Computer automated tools have been and will continue to be developed to implement existing proven technologies that are most prone to inconsistency and omission errors if performed manually.

Although more detailed classifications are possible, automated tools may be partitioned into two basic groups—those tools that assist in the definition and design of the software, and those tools that help evaluate the software once it is built. The first group of tools performs the clerical functions of checking for completeness, consistency, and omission errors during requirements definition and design activities. The usefulness of such tools depends on the ability to express requirements and software design in machine-readable form. Thus, the future utility of these tools is, to a large degree, directly keyed to progress in the development and widespread use of specification languages, program design languages, and automated documentation systems.

The RXVP (Source Text Analysis System, discussed earlier, is typical of the evaluation tools. This technology is already in a highly developed state, although much necessary research continues to be directed toward the automatic generation of an optimal set of test cases. Widespread use and acceptance of these tools are presently hampered by two problems—the reluctance of management to forsake traditional manual methods that previously have been successful on small software proj-

ects, and the availability of tools only to companies able to finance their development.

Both problems are expected to slowly dissolve in the next few years. The first problem will be overcome through an educational process, particularly regarding the cost benefits of automated tools and the prospect of being denied contracts for large software projects if the tools are not used. The second problem is more acute. In the future, more companies may be willing to invest the funds necessary for developing automatic tools in order to protect future business. The purchase rights to some existing tools are presently being marketed by their developers. Another future possibility is that, for projects where a government agency is the procurer, automatic test tools may be available as government-furnished equipment.

As automated tools achieve more prolific use over the entire software life cycle, from requirements definition through testing, a more complete verification and validation of the system results. It can also be expected that errors would be detected earlier in the development cycle, leading to cost containment.

It has also been suggested that nearly error-free programs can be constructed by applying certain organizational techniques. The correctness of these programs could be certified in a static manner (without actual execution) by using mathematical proofs. The constructive approach to the development of programs is with us today in the form of "structured" design and programming. The static approach concerns "program proofs," a popular subject of current research. Practical application of program-proof techniques to programs of significant size may be at least a decade away.[11]

The proof concept uses the program source statements to prove mathematical theorems about program behavior. The expected program behavior is characterized by a set of assertions. The intentions of the designers are reflected in these assertions about values of variables at end or intermediate points of the programs. The program then can be converted into a theorem and the assertions proved.[11]

Program proofs are intended to place verification and validation on a theoretical foundation that is more sound than testing which checks the performance of programs on the limited basis of a set of sample data. Major obstacles to the wider use of proof techniques include difficulties in developing the assertions that are to be proved and in the resulting length of the proof computations. The economic feasibility of program proofs requires the further development of automatic theorem-proving tools.[12]

A related subtechnology of program proofs, symbolic execution is an operational technique more likely to see practical implementation in the near future. Symbolic execution is a way of executing a program [illegible] of symbolic inputs which are verified against a set of predicted conditions or assertions. Practical difficulties involve the generation of assertions and the economics of the required computations.

Because of these difficulties, symbolic execution does not now appear to be economically feasible for mass application to large software projects. However, selective application of symbolic execution to critical software units would be advantageous when these units could do irreparable harm if certain conditions were violated. An example might be spacecraft command and control where certain combinations of commands could injure the vehicle. Such software would require maximum verification.

Presently, a good environment does not exist for the introduction of new verification and validation approaches. A consolidation of present technology is needed to fill the gap between available technology and management's ability to efficiently apply it. This consolidation is necessary to provide a firm basis for introducing advanced methodologies. ■

References

1. Robert Carey and Marc Bendick, "The Control of a Software Test Process," *Proc. COMPSAC 77*, pp. 327-333.*
2. *Ibid.*, p. 327
3. *Ibid.*, p. 329.
4. *Ibid.*, pp. 327, 329.
5. *Ibid.*, p. 327.
6. *Ibid.*, p. 329.
7. *Ibid.*, p. 331.
8. *Ibid.*, p. 329.
9. *Ibid.*, p. 331.
10. Barry W. Boehm, "Government/Industry Software Management Initiatives—Status and Trends," *Documentation Software Management Conf.*, AIAA/TMSA/DPMA, 1978.
11. E. F. Miller, Jr., *Methodology for Comprehensive Software Testing*, Rome Air Development Center, Griffiss Air Force Base, N.Y., 1975, p. 17.
12. M. R. Paige and E. F. Miller, Jr., *Methodology for Software Validation—A Survey of the Literature*, RM 1549, General Research Corp., Santa Barbara, Calif., 1972, p. 37.
13. E. F. Miller, Jr., *A Survey of Major Techniques of Program Validation*, RM 1731, General Research Corp., Santa Barbara, Calif., 1972, p. 58.

*These proceedings are available from the Order Desk, IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.

Michael S. Deutsch is a project manager for the Data Processing Laboratories of Hughes Aircraft Company, Space Communications Group. Since joining Hughes in 1969, he has been involved with the definition and development of both large and small data processing systems. His experience has covered a full range of software engineering functions—system engineering, software development, verification and validation, human factors, and new business/proposal activities. His present interest is in the management of modern software engineering methodologies in project environments.

Deutsch holds a BS from Penn State and an MS from UCLA, both in meteorology.

4-2-17

tion in DEMOS," in *Proc. 6th ACM Symp. on Operating System Principles*, pp. 33-31, Nov. 1977.
[15] A. L. Scherr, *An Analysis of Time Shared Computer Systems*, Cambridge, MA: M.I.T. Press, 1967.
[16] E. R. Lesserite and A. L. Scherr, "Modelling the performance of the OS/360 Time Sharing Option (TSO)," in *Statistical Computer Performance Evaluation*, W. Freiberger, Ed. New York: Academic Press, 1972, pp. 57-72.
[17] B. W. Kernighan and P. J. Plauger, *Software Tools*, Reading, MA: Addison-Wesley, 1976.
[18] P. J. Denning, "Resource allocation in multiprocess computer systems," Ph.D. dissertation, Massachusetts Institute of Technology project MAC, MAC-TR-50, May 1968.
[19] —, "Guest Editorial: SOSAWKCAB 1975," *Software Practice Experience*, vol. 5, Oct. 1975.
[20] —, "Virtual memory," *Computing Surveys*, vol. 2, no. 3, pp. 153-189, Sept. 1970.
[21] —, "Third generation computer systems," *Computing Surveys*, vol. 3, no. 4, pp. 175-216, Dec. 1971.
[22] B. Randell and C. J. Kuehner, "Dynamic storage allocation systems," *Commun. Ass. Comput. Mach.*, vol. 11, no. 5, pp. 365-369, May 1968.
[23] P. J. Denning, "Working sets past and present," *IEEE Trans Software Eng*, vol. SE-6, pp. 64-84, Jan. 1980.
[24] J. Rodriguez-Rosell and J. P. Dupuy, "The design, implementation, and evaluation of a working set dispatcher," *Commun. Ass. Comput Mach.*, vol. 16, no. 4, pp. 247-253, Apr. 1973.
[25] M. C. Adams and G. E. Millard, "Performance measurements on the Edinburgh Multi Access System (EMAS)," *Proc. ICS 75*, June 1975.
[26] J. B. Morris, "Demand paging through the use of working sets on the MANIAC II," *Commun. Ass. Comput. Mach.*, vol. 15, no. 10, pp. 867-872, Oct. 1972.
[27] F. J. Corbato, "A paging experiment with the MULTICS system," in *In Honor of P. M. Morse*, K. U. Ingard, Ed. Cambridge, MA: M.I.T. Press, 1969, pp. 217-228.
[28] M. C. Easton and P. A. Franaszek, "Use bit scanning in replacement decisions," *IEEE Trans. Comput.*, vol. C-28, pp. 133-141, Feb. 1979.
[29] M. V. Wilkes, "Computers then and now," *J. Ass. Comput. Mach.*, vol. 15, no. 1, pp. 1-7, Jan. 1968.
[30] P. J. Denning and J. P. Buzen, "The operational analysis of queuing network models," *Computing Surveys*, vol. 10, no. 3, pp. 225-262, Sept. 1978.
[31] P. J. Courtois, "Decomposability, instabilities, and saturation in multiprogramming systems," *Commun. Ass. Comput. Mach.*, vol. 18, no. 7, pp. 371-377, July 1975.
[32] —, *Decomposability*, New York: Academic Press, 1977.
[33] K. M. Chandy and C. H. Sauer, "Approximate methods for analyzing queueing network models of computer systems," *Computing Surveys*, vol. 10, no. 3, pp. 281-318, Sept. 1978.
[34] G. Balbo, "Approximate solutions of queueing network models of computer systems," Ph.D. dissertation, Comput. Sci. Dep., Purdue Univ., W. Lafayette, IN, Sept. 1979.
[35] R. Marie, "An approximate analytical method for general queueing networks," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 530-538, Sept. 1979.
[36] L. A. Belady and B. Leavenworth, "Program modifiability," in *Proc. Workshop Software Engineering*, ed. H. Freeman, New York: Academic Press, 1980.

Software Management—A Survey of the Practice in 1980

JACK R. DISTASO, MEMBER, IEEE

Abstract—The primary thrust of this paper is to explore many of the problems currently plaguing software development activities and to propose how some of the recently developed management practices may be employed in dealing with them. The practices described range from people organizations (e.g., chief programmer teams) to fully automated engineering tools (e.g., software requirements engineering methodology). A number of techniques are presented. These include methods for coping with communications problems in the requirements definition activity, for evolving a facility which will help increase the productivity of software designers and programmers, and for maintaining a high degree of visibility during the elusive unit design code and test phase of a project.

A very practical view of the management problems and suggested approaches is presented. Key principles, potential "pitfalls," and unresolved concerns are addressed from a viewpoint of where the state of the art is today and where the industry appears to be heading in the mid-1980's.

Manuscript received March 24, 1980; revised May 6, 1980.
The author is with TRW Defense and Space Systems, One Space Park, Redondo Beach, CA 90278.

1. Introduction

DURING the 1970's, several major studies were accomplished toward the orderly evolution of a software management methodology. Advanced methods for defining requirements, achieving productivity improvements, and improving the control and visibility of the software development process were introduced and utilized in many places. This paper presents a summary of several of the approaches developed, with emphasis on how these practices may help address many of the real issues facing the software

0018-9219/80/0900-1103$00.75

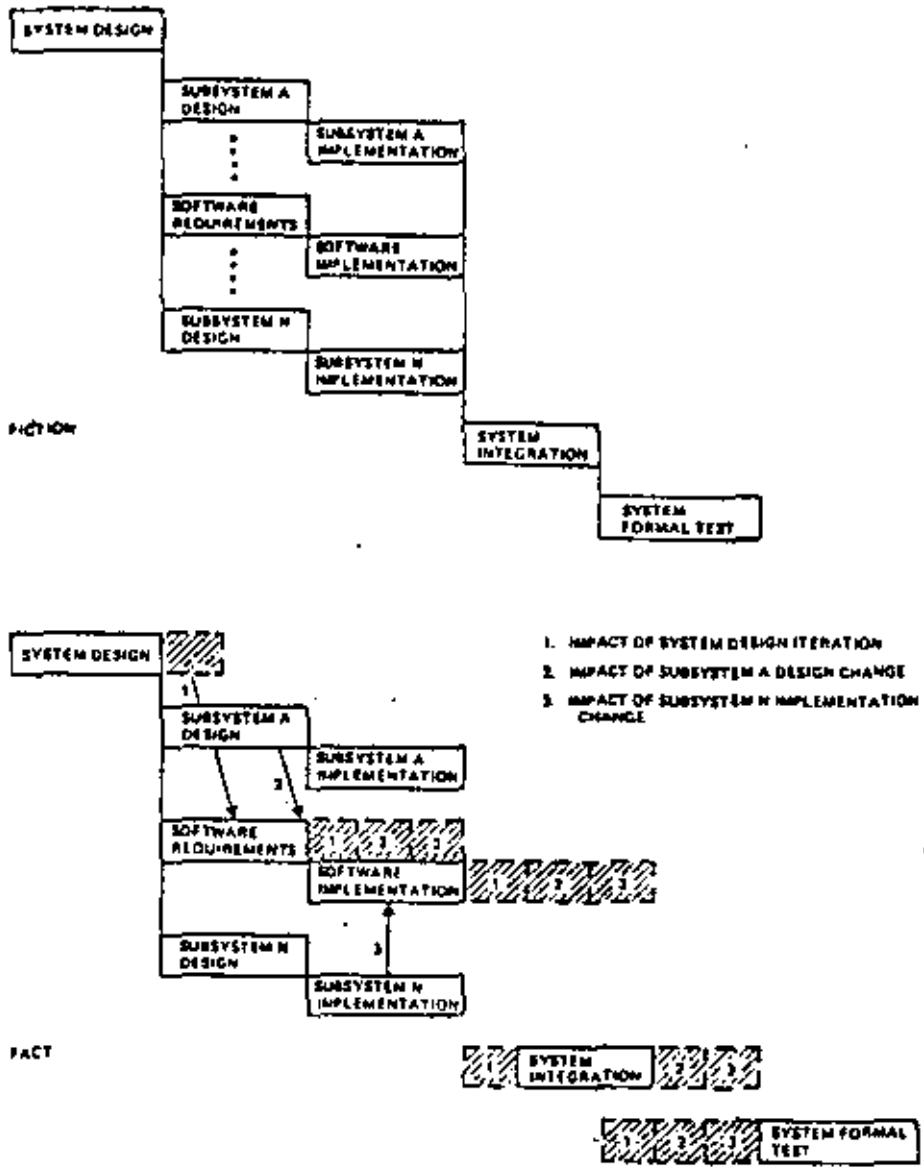


Fig. 1. System development schedule.

development manager of the 1980's. Four principal problem areas are addressed

1) obtaining satisfactory software requirements;
2) improving the art of software cost estimating;
3) achieving significant productivity improvements;
4) maintaining control and visibility of software developments.

Finally, a brief look is taken at some issues that are likely to become key problems in the 1980's.

II. Requirements

Obtaining satisfactory software requirements remains the single largest obstacle to software project success.

The problem of generating complete, consistent, unambiguous, testable software requirements continues to plague the software industry and remains management's most serious challenge to bring order to the software development process [1]. It is important to understand why this problem, which has been debated, analyzed, and researched for years, is invariably the critical factor in project success or failure for most projects of significant size. The key elements of this problem are the following:

1) continually changing user needs;
2) scheduling difficulties of major system developments;
3) communication barriers among users, system designers and software designers;
4) lack of use of new generalized methodologies;
5) misapplication of simulation.

A. User Needs

Few of us would be inclined to make a significant purchase for a product (car, house, etc.) that does not exist and for which there is not even an artist's conception available. Similarly, we would not likely invest in a product for which the only description that exists is a few block diagrams of its basic functions, plus perhaps some equations or other descriptions of its fundamental operating principles. On the other hand, a building contractor who was continuously expected to construct buildings without the square footage, the numbers of

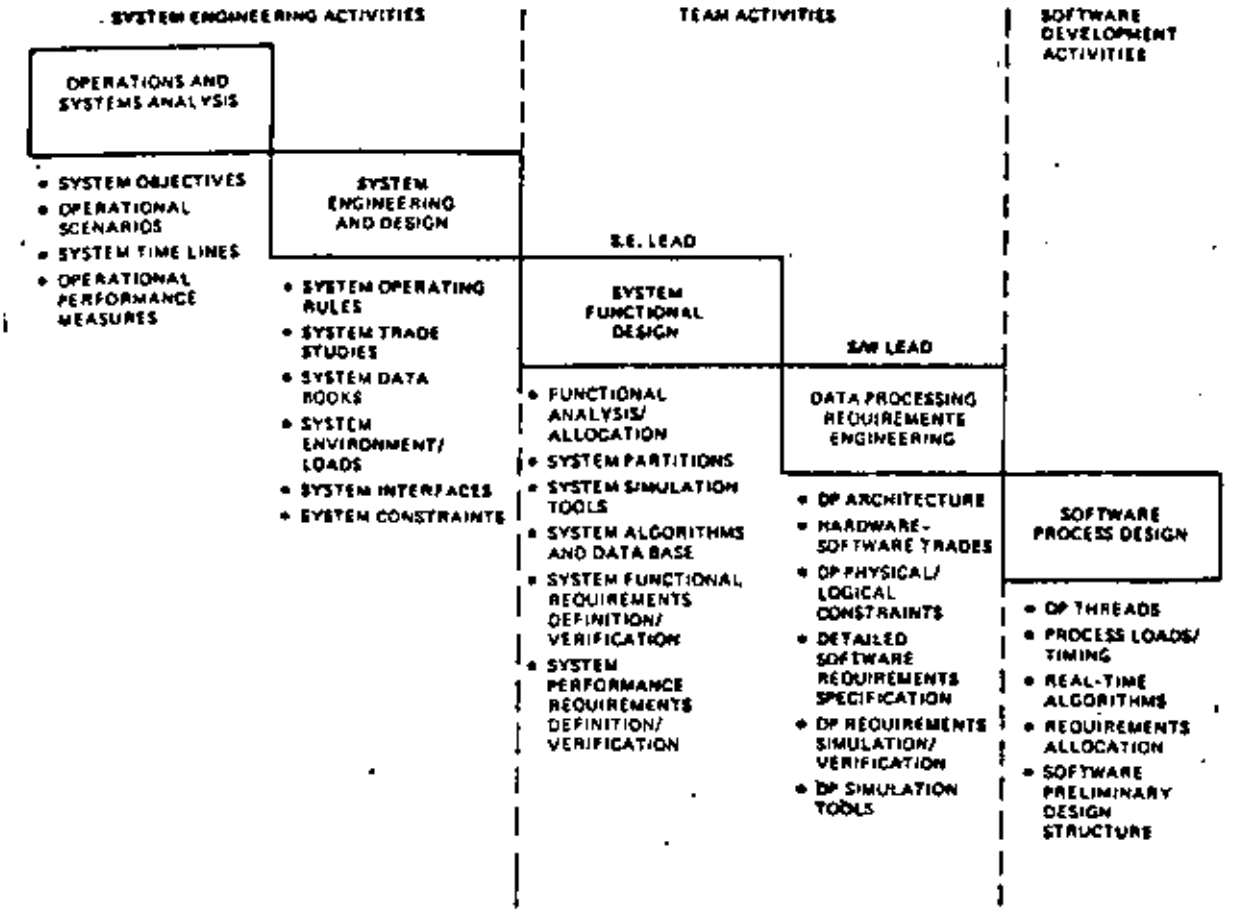


Fig. 2. Team approach to software requirements definition.

ors, and the locations for walls and windows specified, would ckly consider another line of business. Yet, these scenarios l reasonably reflect the situations which data processing rs and developers are commonly faced with at the start of a jor software development activity. J. B. Munson [2] stated t the software industry still has not come up with any "pic- es" of its products. Users of major new complex systems asked to generate precise specifications for a product which y can neither "see" nor fully understand. User's responses predictable. They want a specification that leaves them ne options, does *not* lock them forever to their initial ughts, and provides some freedom to better express their ds as they see the product evolve.

For the developer, the growing size and complexity of sys- ms, coupled with users' inability to express their needs, is reasingly going to impact the developer's ability to scope produce his system. The software industry is one of the remaining builders of one-of-a-kind systems, and until we provide a better prespective of our product, the problem software requirements will remain with us.

System Schedules

For major embedded computer systems, another phenomenon ves to make the software requirements definition task diffi- t. This is the fact that software requirements definition uires a stable overall system design—but schedules rarely mit adequate time for the systems engineering and subsys- design tasks to stabilize before software requirements are ded. For embedded systems, the software provides the control logic (i.e., the glue) that holds the system together and causes it to work as a system. Consequently, functional, control or interface changes in virtually every other subsystem typically have an impact (often major) on the system control software. This is particularly pronounced early in the program when the design of these subsystems are undergoing major fluctuations in attempting to meet their own specifications. It also continues at a lower level throughout the development such that the software continually must adjust to the changing requirements of the subsystems it controls (see Fig. 1).

Consequently, the software requirements are both initially delayed and changed frequently during this time of subsystem design iteration. Unfortunately, to support integration of these same subsystems, portions of the software must be developed and ready at the time of functional integration of these components into the system. The net effect is that the software development must be initiated early enough to support the integration schedule, but runs the risk of major requirements changes if it starts before the system design has settled down sufficiently.

C. Communication Barriers

There exist real communication barriers among typical systems users/operators, hardware systems engineering personnel and software designers/programmers. With the growing application of computer technology into many realms of endeavors, it is not feasible for the average software designer to learn the language and technology of all the potential applications of his product. Conversely, the current explosion of hardware/soft-

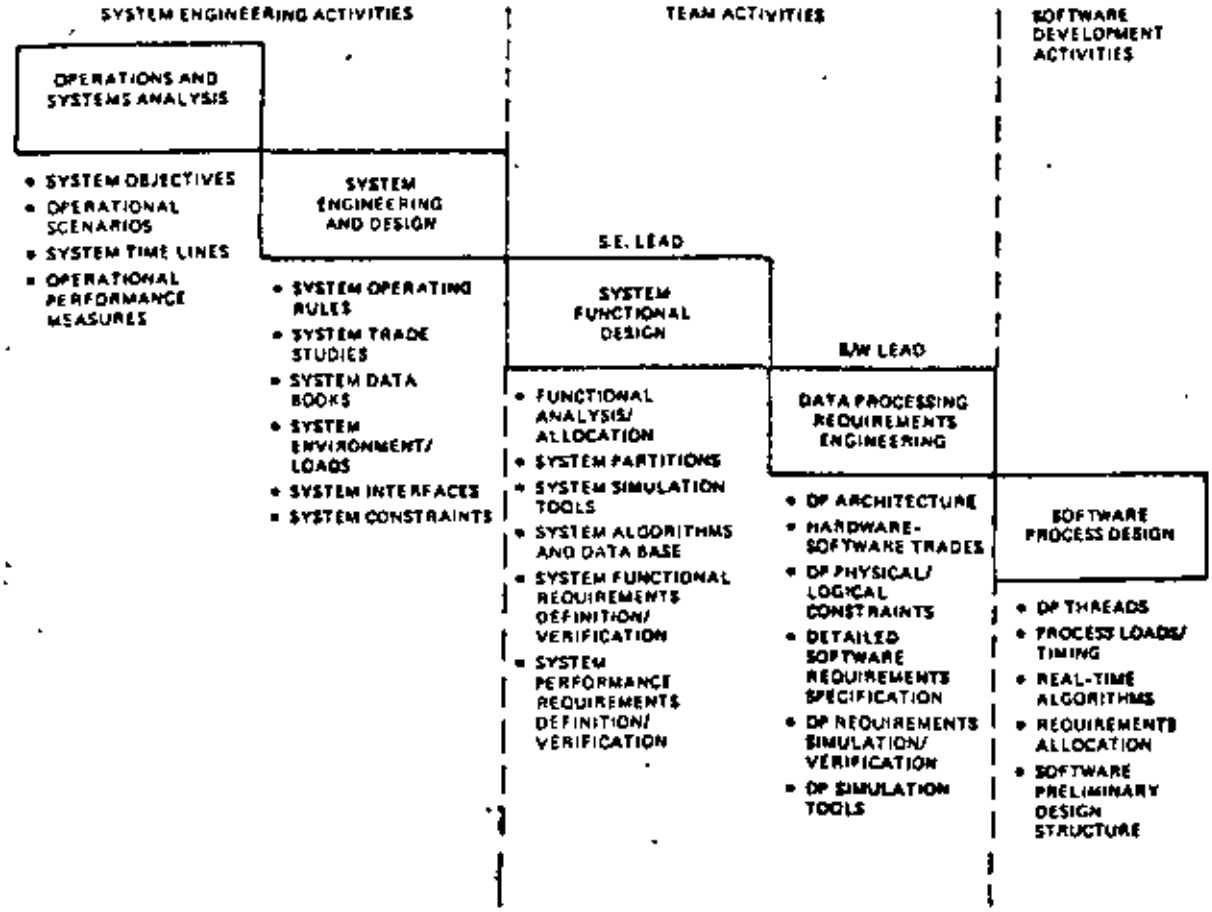


Fig. 2. Team approach to software requirements definition.

fluors, and the locations for walls and windows specified, would quickly consider another line of business. Yet, these scenarios still reasonably reflect the situations which data processing users and developers are commonly faced with at the start of a major software development activity. J. B. Munson [2] stated that the software industry still has not come up with any "pictures" of its products. Users of major new complex systems are asked to generate precise specifications for a product which they can neither "see" nor fully understand. User's responses are predictable. They want a specification that leaves them some options, does *not* lock them forever to their initial thoughts, and provides some freedom to better express their needs as they see the product evolve.

For the developer, the growing size and complexity of systems, coupled with users' inability to express their needs, is increasingly going to impact the developer's ability to scope and produce his system. The software industry is one of the few remaining builders of one-of-a-kind systems, and until we can provide a better prespective of our product, the problem of software requirements will remain with us.

B. System Schedules

For major embedded computer systems, another phenomenon serves to make the software requirements definition task difficult. This is the fact that software requirements definition requires a stable overall system design--but schedules rarely permit adequate time for the systems engineering and subsystem design tasks to stabilize before software requirements are needed. For embedded systems, the software provides the control logic (i.e., the glue) that holds the system together and causes it to work as a system. Consequently, functional, control or interface changes in virtually every other subsystem typically have an impact (often major) on the system control software. This is particularly pronounced early in the program when the design of these subsystems are undergoing major fluctuations in attempting to meet their own specifications. It also continues at a lower level throughout the development such that the software continually must adjust to the changing requirements of the subsystems it controls (see Fig. 1).

Consequently, the software requirements are both initially delayed and changed frequently during this time of subsystem design iteration. Unfortunately, to support integration of these same subsystems, portions of the software must be developed and ready at the time of functional integration of these components into the system. The net effect is that the software development must be initiated early enough to support the integration schedule, but runs the risk of major requirements changes if it starts before the system design has settled down sufficiently.

C. Communication Barriers

There exist real communication barriers among typical systems users/operators, hardware systems engineering personnel and software designers/programmers. With the growing application of computer technology into many realms of endeavors, it is not feasible for the average software designer to learn the language and technology of all the potential applications of his product. Conversely, the current explosion of hardware/soft-

ware computer technology has made it impossible for anyone not continually involved in this arena to keep up with the terminology, let alone the full range of technical implications. Consequently, even though much progress has been made in the education of society in uses and applications of the computer [3], a continually growing communication gap exists among the key organizations needed to configure a major new system. With the increasing specialization required to cope with the complexities in all facets of both the applications and the data processing technology, a new breed of specialists is required who can bridge the gap between the other specialists. The problem with this "generalized" specialist is that his useful life (~5 years) in this role, before he is outdated by the disciplines he serves, is very short relative to the time it takes to gain the experience needed to perform in that role. Currently, the best approach to overcoming these communication barriers is the assembly of a team of systems engineers and data processing designers working together closely to perform the system functional analyses and to prepare the detailed software requirements specifications. As shown in Fig. 2, the systems engineers should take the lead during the functional analysis, and the software designers should direct the detail requirements definition activity. From these teams, hopefully, the "generalized" specialists of the future will emerge.

D. Methodology

Until recently, methodologies for generation of software requirements have been quite informal. Requirements analyses using functional flow block diagrams, N^2-charts, processing flow analyses, and the like have been employed to understand the problem. In this mode, traditional manual free-form English language statements (i.e., the program shall . . .) are used to document the requirements. As Boehm [4] notes, these require virtually no training—but their ambiguity and lack of organization generally lead to serious problems with incompleteness, inconsistency, and misunderstanding among the people using them. While there are now other options available, manual specifications are still the overwhelming majority among forms of current specifications. Boehm [4] and Miller [5] describe four other categories of methodologies

1) forms based: special purpose forms are used to provide the basic structures of the specification;
2) automated assistance: automated interaction with the user is provided to help rapidly construct a specification that adheres to built-in consistency rules;
3) automated: integrated tools that help in generation and validation of complete and consistent specifications;
4) formal: specifications are expressed in a precise mathematical form, with both syntax and semantics rigorously defined.

Two major automated tools which have received probably the most usage the past few years are problem statement language (PSL)/problem statement analyzer (PSA) [6] and software requirements engineering methodology (SREM) [7]. PSL/PSA is a computer-aided structured documentation and analysis technique that was developed for analysis and documentation of requirements and preparation of functional processing specifications. As shown in Fig. 3, this system has the capability to describe an information system, record that description in a computerized data base, incrementally modify the data-base description and produce reports and documentation on the system. PSL expresses the system description in eight aspects: system input/output flow, s[illegible] structure, data structure, data derivation, system size, system dynamics, system properties, and project management. The software package PSA records this description in the data base, modifies it, performs analysis, and generates reports under direction of the user command language. PSL/PSA has continued to evolve and has added more powerful consistency and completeness checks, plus better user-oriented data entry and output reports.

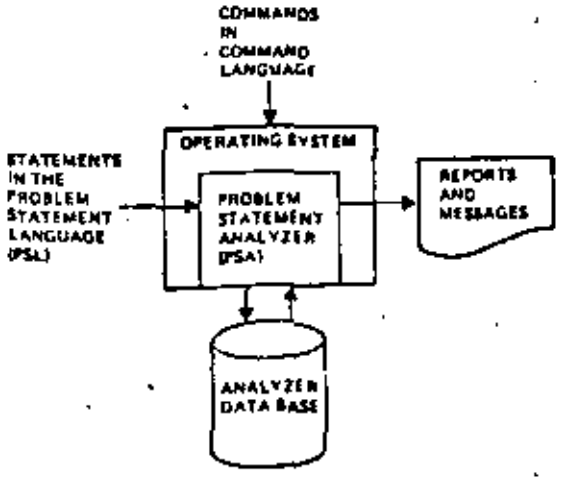


Fig. 3. PSL/PSA system [6].

SREM is a complete methodology for generation of software requirements for large real-time processing systems. As shown in Fig. 4, it employs a nomenclature called requirements networks (RNet's) to unambiguously represent all processing paths in a system while maintaining precedence/successor relationships but without imposing design constraints. It uses a formal requirements statement language (RSL) to reduce ambiguity and to facilitate the automation of the methodology. Automated tools, integrated into a requirements engineering and validation system (REVS) using RSL statements as input, check the requirements for completeness and consistency, maintain traceability to originating requirements and models and generate simulations to validate the correctness of the requirement. SREM has become available on a number of host computers and has been utilized on a number of defense, business, and distributed processing applications.

Several other tools such as SADT [8], SAMM [9], etc., are also in various stages of development and usage. The area of semiautomated and automated requirements generation tools represents one of the major accomplishments of software engineering in the last decade. Yet, at this time, the usage of these tools represents a very small percentage of their potential application. An obvious reason for this is their general lack of availability in many places and the time it takes to learn to use them effectively. Another reason, however, is that these tools generally require a data processing/computer science orientation. Yet, many of the large system software requirements activities are managed and staffed with system analysts/engineers who have relatively little of this orientation and often resist the learning required by these kinds of approaches. This resistance is somewhat akin to that experienced during the early days of structured programming, and a determined customer/management encouragement will probably be required for these tools to gain widespread acceptance.

E. Simulation

Simulation has been used for several years now in the requirements specification process to bring quantification and preci-

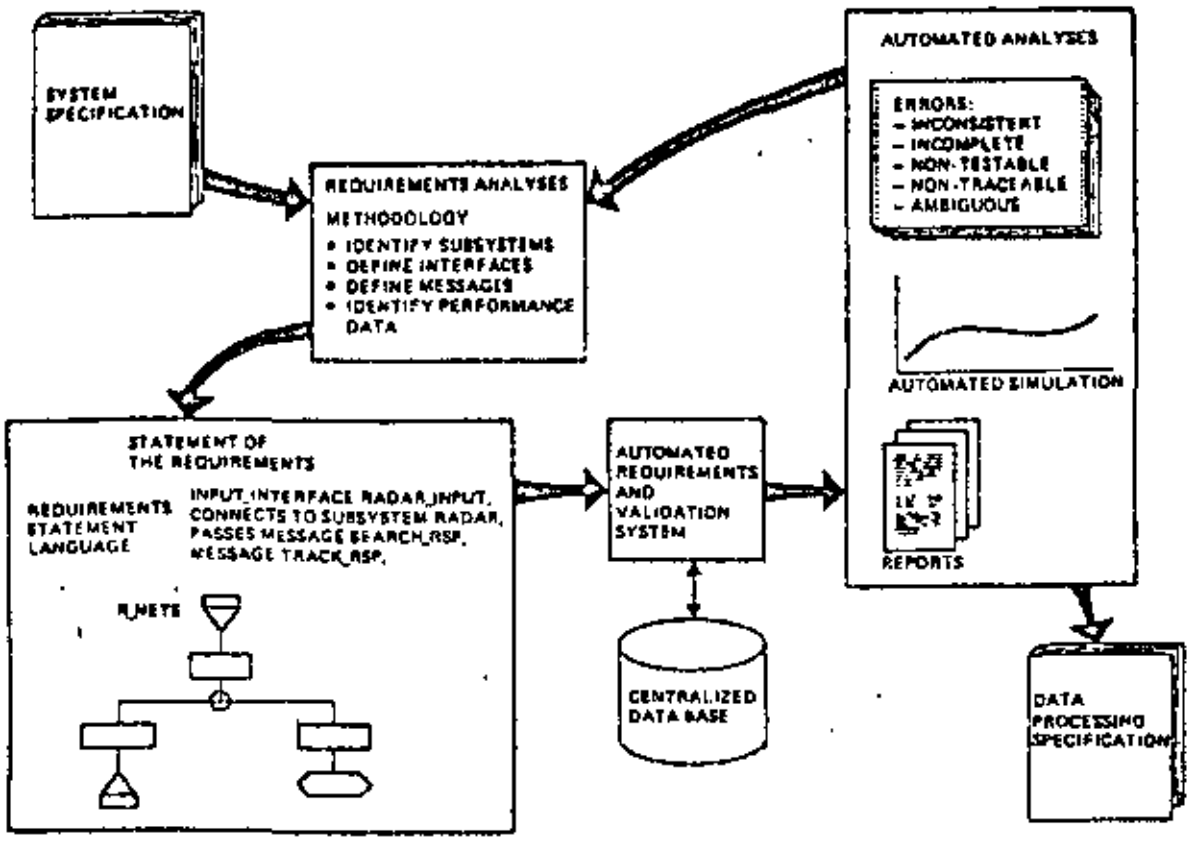


Fig. 4. SREM requirements engineering methodology.

on to the statement of performance requirements. However, experience has shown that simulation is just as likely to be improperly used as it is to be used effectively. Some questions which should be asked when deciding whether to employ simulation are the following.

1) Is the problem supportive of simulation? Systems which are essentially open loop, steady state, or deterministic in nature rarely require simulation to accurately define their performance characteristics. Conversely, closed-loop nonlinear systems or systems requiring rapid response to step events often do require modeling to adequately define their performance parameters. In particular, systems requiring responses to certain stimuli within a small time factor (<10) to the average module dispatch time or running time will usually require fairly detailed modeling of the data-processing hardware and software process.

2) Is there adequate schedule, cost, and data available to model the system to the required fidelity? This is probably the single area where most simulation efforts run into trouble. There is always a level of fidelity below which the simulation is essentially worthless for the task it is being called to do. Yet, frequently, there is either insufficient cost/schedule available to build the simulation or there is insufficient data on the system or environment to model it to the level of detail necessary. In either case, it is prudent to take a hard look at whether any simulation work should be performed, or whether simpler analytic techniques could achieve essentially equivalent results.

3) Is the system design stabilized sufficiently for simulation? Although simulation is frequently used in the requirements definition process, it is important to remember that it is design that must be modeled—not requirements, *per se*. If a system requires modeling in great detail to achieve the required simulation fidelity, then changes to that system d. will be devastating to the simulation effort. In these cases, the simulation often cannot be done in sufficient time to be useful to the requirements definition process, and again a simpler means of obtaining performance parameters must be utilized.

If the answer to all three of these questions is yes, then simulation is usually the best means to obtain precise, quantifiable system/software performance parameters. To obtain the data necessary to define data-processing requirements (e.g., timing, instruction rates, throughput, etc.) usually will require a functional (e.g., event driven) simulation of the data-processing hardware and software, an accurate representation of the process control algorithms, and a realistic model of the external input–output environment.

III. Costing

Software cost estimating is, at best, an imprecise art. Estimates are generally clouded by questions of instruction counts and complexity factors, and universally not believed.

Why are software cost estimates generally considered to be consistently poor? In most endeavors, the first time through a new experience is typically a time of difficulty and error. To improve one's probability of success, generally a period of intense planning, preliminary design and laboratory breadboarding is spent. Few building contractors would bid on a house without a set of plans showing the size, material, and layout required. Rarely is a new piece of engineering hardware (whether a jet engine or an electronic communications set) costed without a significant research and development effort which scopes the magnitude, performance, and environment of the equipment. Yet, major software procurements frequently occur without any research or even meaningful preliminary design of the products to be developed.

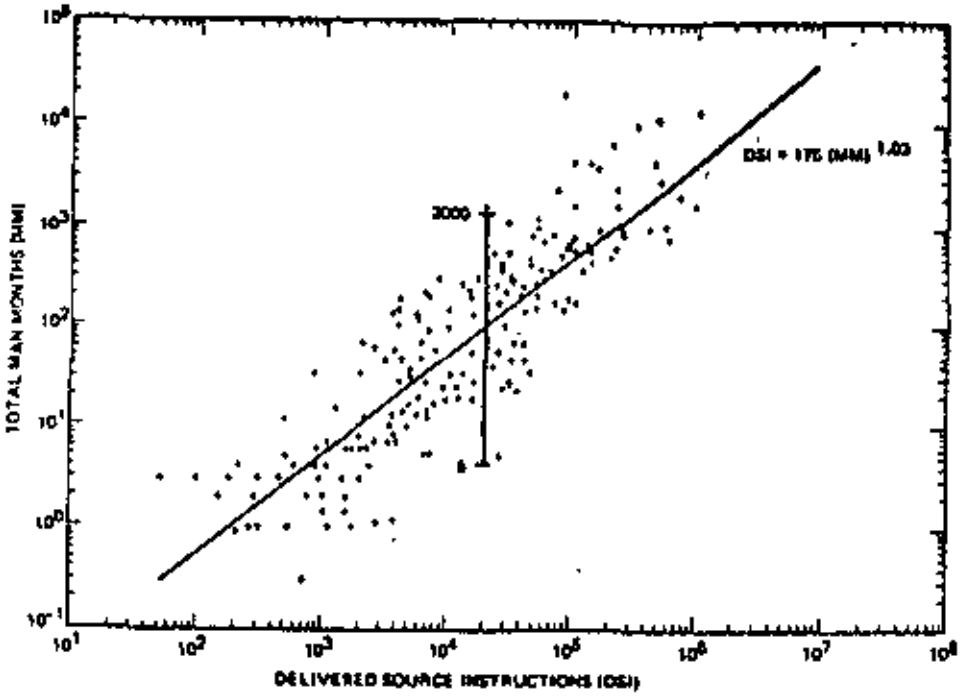


Fig. 5. Correlation of software effort with number of instructions.

A principal reason for this is that a customer or system designer can usually specify a few performance parameters (e.g., radar range, pulse rate, frequency, signal-to-noise, etc.) to characterize a piece of hardware sufficient for a hardware designer to design, scope and bid the equipment. For software, no such parameters currently exist. Software can only be characterized by the detailed functions it must perform to carry out the requirements of the system, and often these must all be specified by the system designer. Frequently major systems are defined and bid before the detailed functional sequences of these systems are specified. Consequently, the software estimates for these systems reflect the lack of design definition necessary for accuracy.

Costing of software is further complicated by the metrics involved. (See Curtis [35], this issue.) RADC [10] has shown that the number of machine instructions provides the best single correlation with program cost (Fig. 5). Yet, even from this data, errors of almost 1000:1 are possible using only this factor as an estimator. Thus it becomes necessary to include such qualitative factors as complexity, requirements volatility, experience, etc., to get a better picture of the true potential cost. This is in sharp contrast to most hardware developments where a few simpler metrics (e.g., numbers of components, drawers, etc.) can usually estimate the effort to a reasonable degree.

Unfortunately, even knowing the factors involved does not guarantee a credible estimate since both the relative value of a given factor and the sensitivity of a particular job to that factor must also be estimated. Several efforts have been made over the past fifteen years to quantify these factors and sensitivities in a form that can be related to a particular software task. Two models which are being maintained and updated are the RCA PRICE-S [11] and TRW SCEP [12]. Although most previous cost models have been generated by curve-fitting cost data from previous developments, both PRICE-S and SCEP attempt to construct the estimate by accounting for the effect of each sensitivity and factor and combining these into a total estimate. Both models have used previous developments for validation and refinement.

PRICE-S uses 42 representative factors (Fig. 6) to represent the software development environment. The principal inputs to the model are grouped into 7 categories

1) project magnitude—how much code to be produced;
2) program application—type of project (e.g., MIS, C^2, etc.);
3) level of new design and code—not available from existing inventory;
4) resources-experience/skill level of developers;
5) hardware limitations—timing/memory constraints;
6) customer specifications and reliability requirements—measure of reliability, testing and documentation required;
7) development environment—what complicating factors exist.

The basic metric for project magnitude is number of delivered executable machine instructions (DEMI's) to be developed. Program application is the model's approach to provide weights for the relative difficulty of different types of software (from 11.0 for operating systems to 0.9 for routine mathematical operations). Organizational capabilities and experience, including labor categories, productivity rates, computer operations, etc., are modeled by the resources variable. Other complexity parameters can be accounted for under development environment, where special factors such as parallel hardware developments, new languages, highly changing requirements, etc., may be addressed.

RCA PRICE-S is a supported model, accessed on commercial time sharing computers and available under monthly leasing agreements. Another similar commercial software cost estimation model is the Putman SLIM model [13].

TRW software cost estimating program (SCEP) is a model which uses 16 factors to encompass the cost estimating range. These are grouped into

1) program size—number of DEMI's and data items;
2) program attributes—complexity, type, language;
3) hardware attributes—storage/timing constraints, concurrent hardware developments;

PROJECT SIZE
PROJECT TYPE (MIS, RADAR, TELEMETRY, ETC.)
OPERATIONAL CUSTOMER ENVIRONMENT
HARDWARE CONSTRAINTS (SYSTEM LOADING)
EXISTING DESIGN
EXISTING CODE
EXTERNAL INTERFACES (TYPE AND QUANTITY)
HIERARCHICAL DESIGN FUNCTIONAL FLOW STRUCTURE
NUMBER OF FUNCTIONS PERFORMED
AMOUNT OF CODE PER FUNCTION
SCHEDULE CONSTRAINTS, LEAD TIMES AND OVERLAPS
RESOURCE CONSTRAINTS
ICN EFFECTS
ECONOMIC TRENDS
TECHNOLOGY GROWTH
FEE, PROFIT, AND G & A
COMPUTER OPERATION COSTS
OVERHEAD
ORGANIZATIONAL EFFICIENCY
SKILLS
PROJECT FAMILIARITY
INTENSITY OF EFFORT
CHANGING REQUIREMENTS
PROGRAMMING LANGUAGE
COMPILER POWER AND EFFICIENCY
DEVELOPMENT LOCATION (IN HOUSE OR ON-SITE)
PROJECT COMPLEXITY
ENGINEERING REQUIREMENTS
PROGRAMMING REQUIREMENTS
CONFIGURATION CONTROL
DOCUMENTATION
PROGRAM MANAGEMENT
DESIGN PHASE ACTIVITIES
IMPLEMENTATION ACTIVITIES
TEST AND INTEGRATION ACTIVITIES
INTEGRATION AND INDEPENDENT PROJECTS
VERIFICATION AND VALIDATION
MULTIPLE TEST BEDS INSTALLATIONS
GOVERNMENT FURNISHED SOFTWARE
PURCHASED SOFTWARE (E.G., SUBCONTRACTS)
DESIGN-TO-COST
RESOURCE ALLOCATION WITH RESPECT TO TIME

Fig. 6. Factors accounted for by price S.

FACTOR	RATING	RQTS. ANALYSES	PRELIM. DESIGN	DETAIL DESIGN	CODE & UNIT TEST	INTEG. & TEST
1 REQUIRED QUALITY	VERY LOW	• LITTLE DETAIL • MANY TBD'S • LITTLE VALID'N • MINIMAL PLANS (QA, CM, ACCEPT.) • MINIMAL SRR	• LITTLE DETAIL • MANY TBD'S • LITTLE VERIF'N • MINIMAL QA, CM, STDS • MINIMAL PDR • NO DRAFT USER MAN.	• BASIC DESIGN INFO. • MINIMAL UOF'S • MINIMAL QA, CM • MINIMAL CDR • NO DRAFT USER MAN.	• MINIMAL U.T. PLAN • MINIMAL FCL'S, PATH TEST, STDS CHECK • MINIMAL QA, CM • MINIMAL I/O AND OFF-NOMINAL TESTS • MINIMAL USER MAN.	• MINIMAL TEST PLANS • NO TEST PROCEDURES • MANY RQTS. UNTESTED • MINIMAL QA, CM • MINIMAL STRESS, OFF-NOMINAL TESTS • MINIMAL AS-BUILT DOC'N
	LOW	• BASIC INFO VALID'N • FREQUENT TBD'S • BASIC PLANS (QA, CM, ACCEPT.) • BASIC SRR	• BASIC INFO VERIF'N • FREQUENT TBD'S • BASIC QA, CM, STDS • BASIC PDR • MINIMAL USER MAN.	• MODERATE DETAIL • BASIC UOF'S, PA, CM, CDR • MINIMAL USER MAN	• BASIC U.T. PLAN • PARTIAL FCL'S, PATH TEST, STDS CHECK • BASIC QA, CM, UM • PARTIAL I/O AND OFF-NOMINAL TESTS	• BASIC TEST PLANS • MINIMAL TEST PROCED • FREQUENT RQTS UNTESTED • BASIC QA, CM, UM • PARTIAL STRESS, OFF-NOMINAL TESTS
	AVG	FULL TRW POLICIES	FULL TRW POLICIES	FULL TRW POLICIES	FULL TRW POLICIES	FULL TRW POLICIES
	HIGH	• DETAILED VALID'N PLANS, SRR	• DETAILED VERIF'N, QA, CM, STDS, PDR, DOC'N	• DETAILED VERIF'N, QA, CM, STDS, CDR, DOC'N	• DETAILED U.T. PLAN, FCL'S, QA, CM, DOC'N, CODE WALK THRUS • EXTENSIVE OFF-NOMINAL TESTS	• DETAILED TEST PLANS, PROCEDURES, QA, CM, DOC'N • EXTENSIVE STRESS, OFF-NOMINAL TESTS
	VERY HIGH	• DETAILED VALID'N, PLANS, SRR • IV&V INTERFACE (SUPPORT, RESPONSE)	• DETAILED VERIF'N, QA, CM, STDS, PDR, DOC'N • IV&V INTERFACE	• DETAILED VERIF'N, QA, CM, STDS, CDR, DOC'N • IV&V INTERFACE	• DETAILED U.T. PLAN, FCL'S, QA, CM, DOC'N, CODE WALK-THRUS • VERY EXTENSIVE OFF-NOMINAL TESTS • IV&V INTERFACE	• VERY DETAILED TEST PLANS, PROC'S, QA, CM, DOC'N • VERY EXTENSIVE STRESS, OFF-NOMINAL TESTS • IV&V INTERFACE

Fig. 7. Example of factor/rating definitions: TRW SCEP model.

4) project attributes—personnel quality, experience;
5) environmental attributes—requirements volatility, required quality, computer access.

A major thrust of the SCEP model was to define subjective parameters (e.g., complexity, required quality) in terms relatable to the problem being solved. Fig. 7 shows an example of the different rating values for required quality. In this table, quality is defined relative to "full TRW policies." This rating refers to a standard set of policies for TRW software development projects, which provide another foundation block for reliable software cost estimation. As in PRICE-S, the principal metric for SCEP is the number of DEMI's to be developed. The fidelity of the model was demonstrated by comparisons with twenty completed projects (see Fig. 8) and this validation is continuing as further project completion data is made available.

The principal considerations for someone using these or similar models for costing are: ability to accurately estimate program size, ability to define the project environment and characteristics (e.g., personnel quality, requirements volatility, relative complexity, etc.) and the ability of the model to accurately

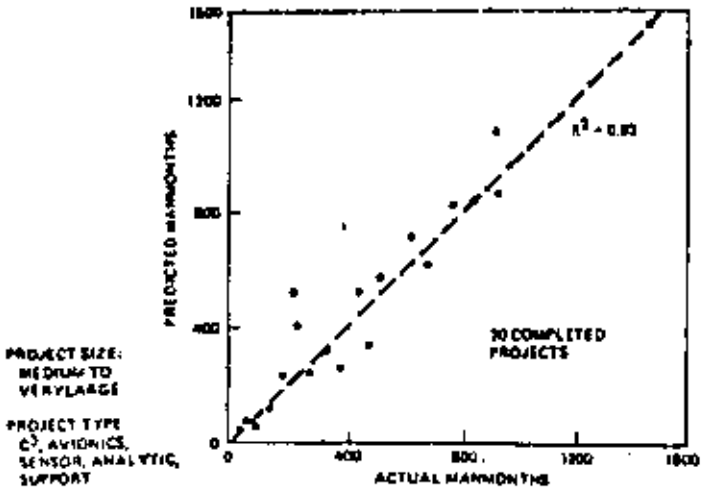


Fig. 8. Example of TRW software cost model performance.

estimate the cost. For a new development, errors in estimating program size of 100 percent are not unusual. Unless the program requirements are reasonably stable and a thorough preliminary design is performed (to a level where either algorithms can be scoped or unit sizes compared with previous experiences), the likelihood of significantly misestimating program size is quite high. One's ability to accurately assess the program environment will vary greatly from project to project. For an environment where the requirements are available or self-generated, the programming team defined and in place, and the task homogeneous and well understood, the environment should be readily defined. Many business systems projects fit this description. The opposite end of the spectrum would be a large new tactical command and control system where the man–machine interface and operational algorithms are only vaguely understood, the processor capacity limited by size/weight considerations, and a large increase of staff necessary to obtain the needed development personnel. For this latter class of projects, it will be necessary to continue the estimating process well into the development itself until many of the parameters have settled down. For many projects, it may be as late as the software preliminary design review (PDR) before an accurate assessment can be made. However, it is usually better to recognize the budget problems that exist at this time (before the main project expenditures are committed) then to jump into the main development effort unaware of the potential cost implications.

IV. Productivity

Techniques for obtaining the large software productivity improvements needed to meet the data processing demands of the 1980's are neither currently available nor even on the near horizon.

During the last decade, the industry experienced continually increasing productivity, primarily through the use of higher order languages and software implementation/test tools and the employment of modern programming practices. Estimates on the long-term productivity improvement rate of programmers range from 3 to 7 percent/year [14]–[16]. While this is not without merit, the demands of the 1980's is for 10, 50, or 100:1 improvement [1]. This demand for order of magnitude improvements is being driven by three forces: the advent of complex "super systems" created by netting of multiple distributed systems, the scarcity of trained software personnel, and the explosive growth of the computer hardware/microprocessor industry which has put data processing hardware within the cost reach of most organizations.

It can be said that the software industry has managed to automate virtually every field except its own. In fact, Tanaka [16] sees information processing becoming the most labor intensive of all industries by 1985 if better methods are not employed.

The greatest likelihood for obtaining substantial productivity improvements lies in better automated tooling for the software designer just as computer aided design/manufacturing (CAD/CAM) has significantly improved the productivity of hardware designers. The techniques most commonly discussed are the software production facility, libraries of primitives, new applications-oriented languages, integrated systems of development tools, etc. A problem with most of these approaches is that they focus on the implementation phase of the development, while much of the software cost and schedule is embedded in the requirements, design, and system test phases. Further research to tie requirements methodologies, design languages, and automated test generation/verification tools, together with the implementation tools will be necessary before the degrees of desired improvements can be approached.

An example of a specialized computing facility dedicated to support large software development projects is the programmers's workbench (PWB) [17] in use at Bell Laboratories. The PWB is based on the concept that program development and execution of the resulting programs are two radically different functions, and much can be gained by assigning each function to a computer best suited for it. The software development is done on a PWB computer dedicated to that task, while execution occurs on another computer—the "target" machine. Thus the PWB presents a single uniform interface to its users, even though it may be supporting several target systems. The Bell Labs PWB is currently implemented on a network of PDP-11's operating under the UNIX time-sharing systems. Some capabilities of the PWB are

1) convenient interactive computing services;
2) a file structure oriented to interactive use;
3) a set of tools for scanning and editing text files;
4) a flexible command language;
5) extensive document preparation facilities;
6) facilities to support small data base management problems.

The PWB is a reasonably good example of a program production facility which generally features interactive programming, automated tools and word processing. The range of possibilities for such a facility (Fig. 9) is a function of the tooling provided. This can range from a few text editing capabilities to a complete set of development, test, quality assurance, and configuration management tools to a set of tools plus an entire library of primitives which can be called and inserted into a program via the interactive programming features. The key to gaining significant productivity improvements from these techniques is the provision of those tools and primitives which the users are familiar with and believe are useful. This generally means starting the production facility simply, with the interactive features, and a few well chosen tools. Then additions to the tooling can be made as the demand for them increases (i.e., when the users become comfortable with the environment). This will also serve to spread out the capital investments needed to furnish this kind of a facility.

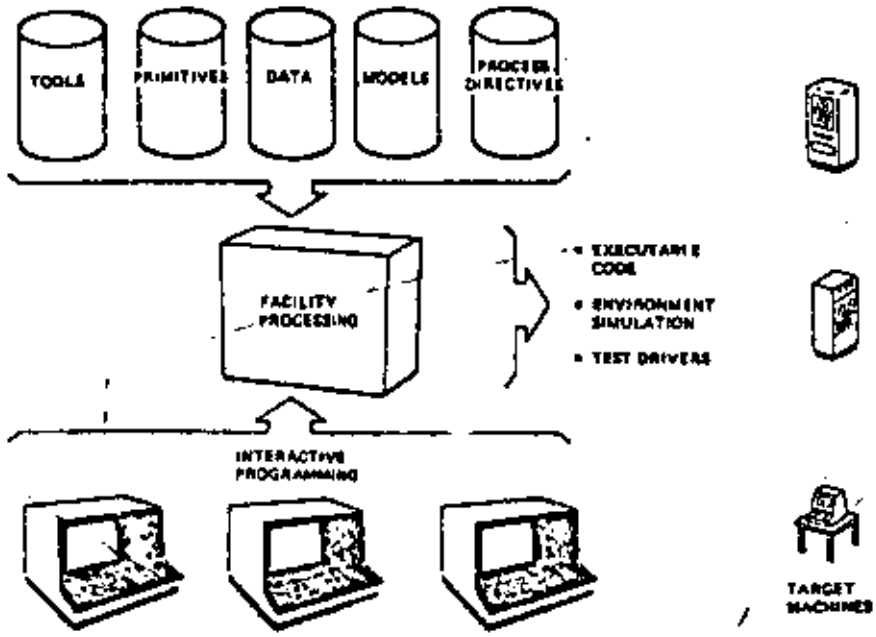


Fig. 9. Program production facility.

PAGE	DOCUMENT	SPECIFICATION PARAGRAPH	DESIGN SOLUTION			CHANGE STATUS	TEST PROCEDURE ALLOCATION			NOTES
			FUNCTION	TASK	ROUTINE		UNIT	SUBSYSTEM	SYSTEM	
153	SPEC #1438785	3.2.2.1.3.4.2.1a	JSCR1	TRD1	R173 R174				SP 2110	
163	SPEC #1438785	3.2.2.1.3.4.2.1b	JSCR1	TRD1	R173	ECP #49	173-6			*MAY BE IMPACTED BY ECP #846
163	SPEC #1438785	3.2.2.1.3.4.2.2	JSCR1	TRD1	R194 R196			IP 1003		
164	SPEC #1438785	3.2.2.1.3.4.2.3a	JSCR1	TRD1	R173	ECP #49			SP 2110	
164	SPEC #1438785	3.2.2.1.3.4.2.3b	JSCR1	TRD2 TBR3	R205 R207			IP 1003		
165	SPEC #1438785	3.2.2.1.3.5	JBD	TCB	R541	ECP #48	541-2			
165	SPEC #1438785	3.2.2.1.3.5.1	JBD	TCB	R541	ECP #48	541-1			
165	SPEC #1483785	3.2.2.1.3.5.1.1a	JBD	TCB	R547 R552 R538				SP 3479	
166	SPEC #1438785	3.2.2.1.3.6.1.2a	JBD	TCC	R012 R436	ECP #12		IP 1123		*SEE ICD #43178
166	SPEC #1438785	3.2.2.1.3.6.1.2b	JBD	TCC	R012 R673	ECP #12		IP 1123		*SEE ICD #43178
166	SPEC #1439785	3.2.2.1.3.6.2	JBDA	TCC	R018 R051 R018	ECP #12		IP 1124		
167	SPEC #1438785	3.2.2.1.3.6.3.4	JBDA	TAC2	R043 R047				SP 3217	
168	SPEC #1438785	3.2.2.1.3.6.3.4	JBDA	TAC1	R049		49-4 49-5			*REFERENCE OPR-AC1
169	SPEC #1438785	3.2.2.1.3.6.3.3	JBDA	TAJ6	R661			IP 1271		
169	SPEC #1438785	3.2.2.1.3.6.4	JBDA	TAJ6 TDJ3	R607 R247 R248 R249			IP 1271		

Fig. 10. Example of requirements traceability matrix.

V. Control

The key to effective control is to break up the development into a number of small measurable steps and then to rigorously audit the satisfactory completion of those steps.

The 1970's are noted for the introduction of several excellent approaches for bringing order into the software development process. The initial work which started this trend was the structured programming/top-down design concepts presented by Dijkstra [18]. Many others have since contributed elements of these approaches, generally referred to as "modern programming practices." This paper would like to discuss some of these techniques, which are believed necessary for controlling the development of the large software based systems of the 1980's.

A. Requirements Traceability

While it is often most difficult to obtain an adequate requirements specification for a large operational real-time system, it is mandatory that once obtained, those requirements be controlled, mapped to the design, and verified in some formal auditable fashion, in order to have any measure of confidence that the product will meet the intent for which it is being developed. While this is relatively straightforward for small

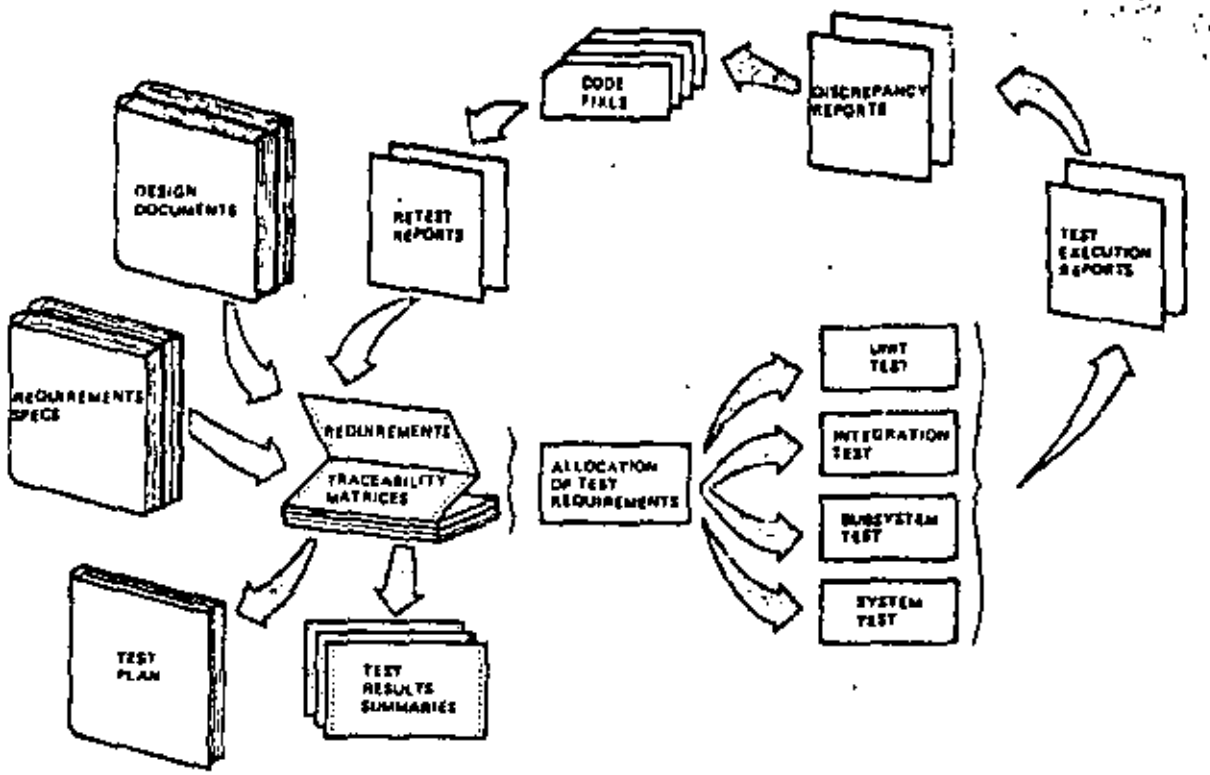


Fig. 11. Requirements traceability process.

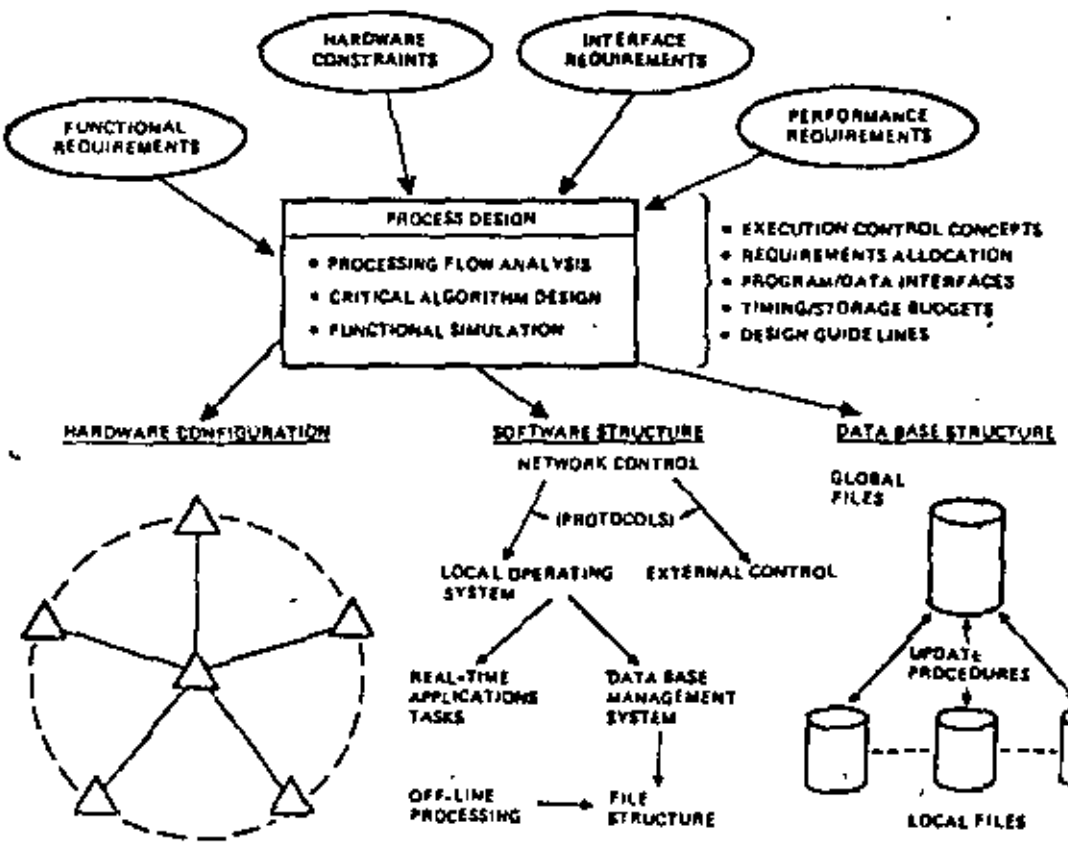


Fig. 12. Process design approach.

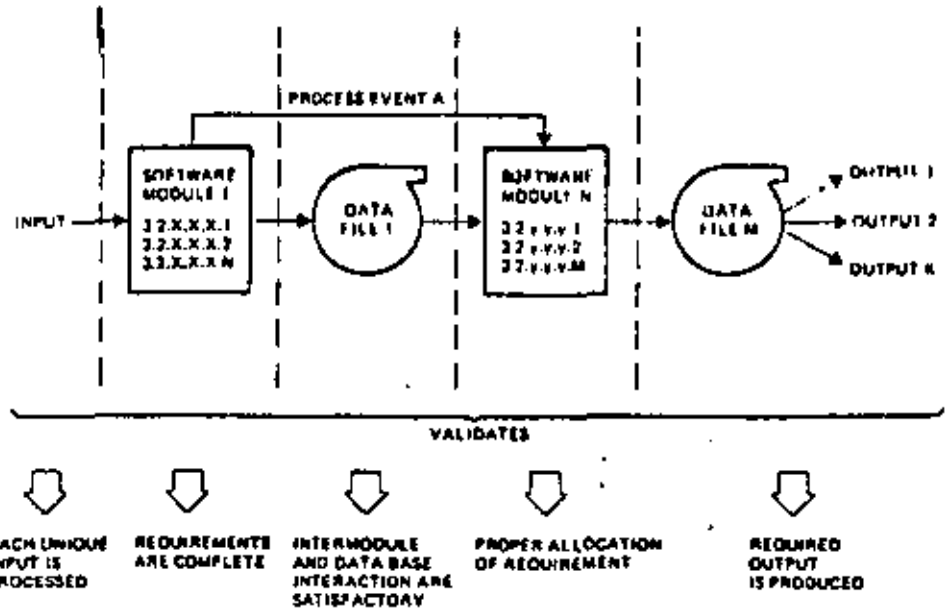


Fig. 13. Mapping of system inputs—requirements—outputs.

projects, it is often a major consideration for large projects with thousands of requirements, many of which are undergoing significant change throughout the development. To control this process, it is advisable to maintain a record for each requirement showing its origin, change status, design allocation, test allocation, verification method, and authority. The origin of an individual requirement can range from a uniquely identified paragraph in a requirements specification to an element in a table of an interface control document. It is mandatory to collect each real requirement (whatever its source), give it a unique identification (usually referenced to source document, paragraph, section, etc.) and establish a control record for it which is maintained throughout the project. Fig. 10 shows an example of the type of information needed in this process. In [19], Krause and Diamant discuss a specific tool to perform this task which was used successfully to control and monitor the testing of a program with over 10 000 requirements.

As shown in Fig. 11, the key elements in successfully tracking the requirements allocation and testing process are to

1) uniquely identify every individual processing requirement (including subparagraphs or sections);
2) allocate design solution to lowest level(s) possible;
3) maintain status of latest authority governing that requirement (specification, interface document, change notice, etc.);
4) allocate verification of that requirement to specific test procedure (at either unit, subsystem or system level);
5) maintain record of test/retest results;
6) continuously audit the process to be sure all data is up to date.

B. Process Design

The concept of process design grew out of the ballistic missile defense processing requirement for large scale, high reliability, ultra-high-speed real-time processing of complex algorithms, and the associated need to have some assurance that the problem was solvable during the early stages of the major development. Davis and Vick [20], Goulding and Lawson [21], Dreyfus and Karacsony [22] have each discussed different aspects of the approach. Yeh and Zave [36] present a view of process specification from the perspective of software requirements generation. The purpose of process design is to iterate the system requirements, hardware/software architecture, and data-base structure to evolve a baseline hardware configuration and software design structure. The tools of process design are processing flow analysis, functional simulation and critical algorithm design/benchmarking. Fig. 12 depicts the basic engineering approach. Conceptually, process design considers a real-time system as a set of transformations on the system stimuli to produce the proper system responses. These transformations may initially be represented in any of several different forms (RNet's, threads, Petri-net, etc.) but must eventually provide a direct mapping which shows how the allocated processing requirements operate on the system inputs to provide both the proper external outputs and internal state changes (see Fig. 13). Critical algorithm benchmarking is used to "size" (timing, memory) those key functions which have large uncertainties associated with them. Functional simulation is used to model the entire process, aid in iterating the potential design solutions, and then to test the baseline structure to demonstrate its predicted performance under varying loads and external environments.

C. Incremental Development

The purpose of incremental development (Fig. 14) is to reduce risk in the project by obtaining an early determination of the performance of key tools, procedures and software. Williams [23] discusses the need to "hear the process talk" many months before a complete set of capabilities would otherwise be available. The partitioning of a software implementation can significantly affect the entire system development process and needs to be considered as a fundamental step in the initial scheduling of a program. Some of the key factors which should be considered in defining an incremental development approach are as follows.

1) Preliminary design of the entire software package must be completed before initiation of any incremental development activity. Each increment must proceed in a straightforward manner from the preliminary design structure. Budgets for timing, storage, accuracy, etc., allocated at preliminary design must be rigorously adhered to and demonstrated in

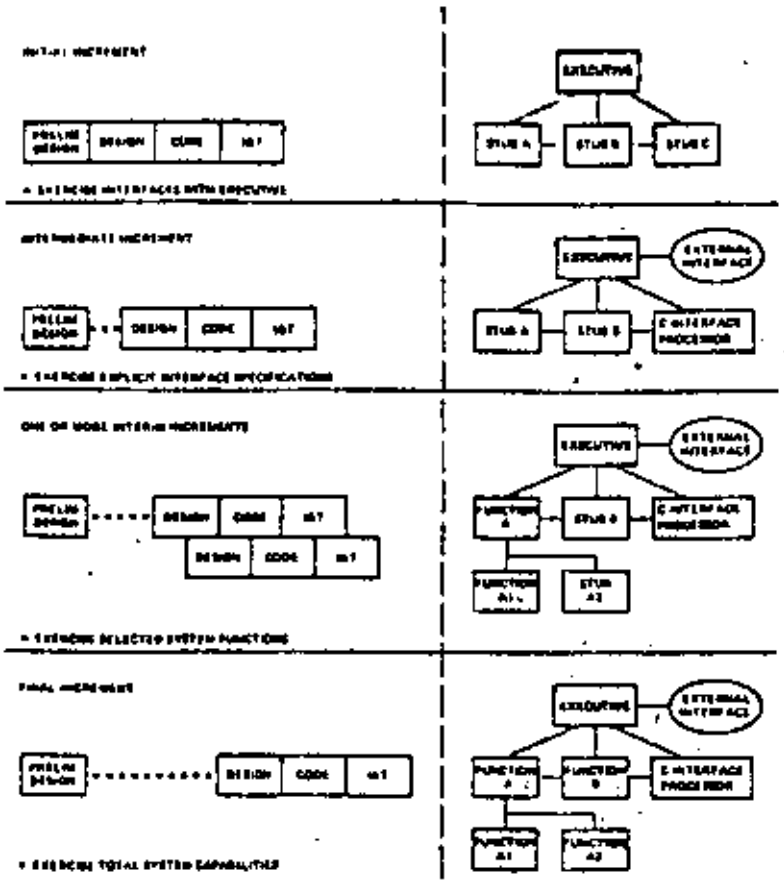


Fig. 14. Incremental development and top-down integration.

each increment, so that late increments do not have to "make-up" for the lack of control of earlier increments.

2) Functional capabilities which are well defined but considered high risk should be developed early.

3) System capabilities which are poorly defined or likely to change should be implemented as late as possible. This includes software functions supporting other subsystems which themselves are undergoing change.

4) Organize increments to minimize breakage. Where possible include complete functions with clear interface delineations.

McGowan [24] suggests that for large projects the following 4 increments should be provided in the indicated order.

1) "An *initial* increment to exercise all interfaces with executive software." This can be done using a "dummy" or "stub" version of the applications code to execute the operating system features in a realistic environment.

2) "An *intermediate* increment to exercise explicit interface specifications." This is exceptionally valuable when the software system must interface with newly built hardware or nonstandard interface protocols. Most interface bugs can usually be worked out during this increment.

3) One or more "*interim* increments to exercise selected system functions." This is where function selection based on potential requirements or software breakage should be carefully addressed.

4) "A *final* increment to exercise total system functions." By this time most of the system capabilities will have been demonstrated. Therefore, this increment can focus on demonstrating system performance under varying environments and loads.

As seen, top-down integration is readily served by incremental development, as the process can evolve from a first increment consisting of an operating system plus "dummy" application programs (i.e., application modules represented by stubs simulating the module interfaces) to a set of module control routines with either stubs or real functional software (depending on the specific increment) supporting them to a final capability where all stubs are replaced by operational software.

D. Structured Development

In early 1970's, Baker [25], Mills [26], Brooks [27] and others, reporting on the lessons learned at IBM, presented a software development methodology which included the concepts on top-down design, chief programmer teams, structured programming, and the like. These concepts received widespread acclaim in the literature, but did not as readily become accepted practices within the industry. Although a 1977 study by Holton [28] implied a lack of the widespread usage of these approaches at that time, Munson [2] demonstrated at a 1979 workshop on software management that at least most aerospace and defense system contractors now use some version of these practices as a matter of policy. A brief summary of the key concepts is given below.

Chief Programmer Team: A team of 4–8 software developers consisting of a chief programmer, backup programmer, librarian, and task programmers. The chief programmer is the team leader who is responsible for both the development of critical modules and the review of noncritical modules assigned to the task programmers. The backup programmer is also a senior programmer who supports the development of critical modules and is available to take over as chief programmer, if required.

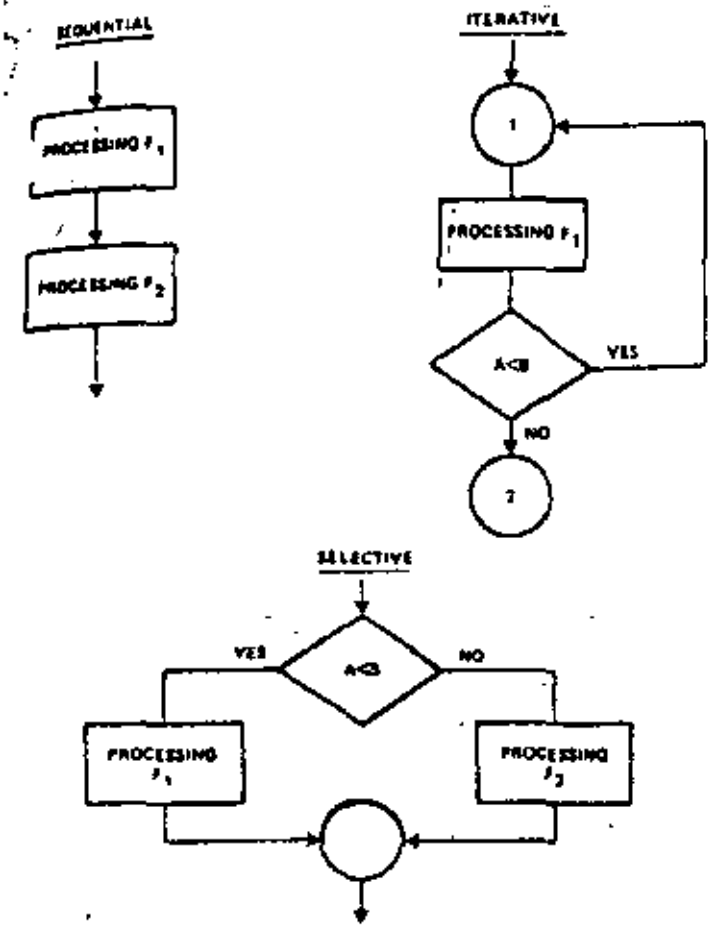


Fig. 15. Structured programming basic constructs.

The librarian maintains the support processes for updating code libraries and test data libraries, supports program compilation, binding and execution, and maintains status on program discrepancies and fixes.

Top-Down Design: Following allocation of requirements to individual modules, the system is further decomposed in a series of steps starting from the module control level to a set of simpler and simpler functions until the lowest level routines are defined. Program design and coding is then started at the top level. Testing occurs with stubs for lower level routines. The "stubs" are then replaced as each lower level routine is completed. Top-down integration proceeds similarly.

Structured Design: This approach enhances the modularity of a system by providing techniques for dividing large programs into modules. It provides a methodology for decomposition of a system into independent modules, such that changes to any given module will have minimal effect on other modules of the system.

Structured Programming: This concept serves to define a system by programs having only single points of entry and exit. This is accomplished by virtual elimination of branching statements, or conversely, by limitation of program constructs to sequential, iterative (DO-UNTIL), and selective (IF-THEN-ELSE) structures (Fig. 15). This process eliminates the source of many programming errors, provides a formal mechanism for reducing individual routine size to less than a single page of listing (thus complementing program modularity), and eliminates the possibility of generating complicated hard-to-understand (and debug) "spaghetti" code. This technique is usually implemented along with source listing logic indentation to improve readability. By expanding this concept into design language or flowcharts, every system can be represented by a series of one-page descriptions with single entry and exit.

Design Walk Through: This walk through is a peer-level design review with the emphasis on program error detection. It is conceptually a low-key nondefensive review where the participants work to help the reviewer find potential errors in the design and code. There has been much discussion on whether management could or should attend. The quality and intensity of the review is potentially higher with some management attendance—but the thrust of the review must still be to find errors, not to impress the boss. Thus any management participation must be limited to those capable of understanding and contributing to the detail review process.

E. Software Design Language

The application of tools to allow English-like languages to be used in designing and documenting software is becoming more widespread. One such automated tool is the program design language (PDL) of Caine *et al.* [29]. A design in PDL is written in structured English for processing by the PDL processor. The processor generates design documents which include formatted sequence listings, program calls and call cross references. The PDL output can be formatted so as to remove the need for separate program flowcharts. Updates are accommodated by changes to the structured English input.

PDL is commercially available and is currently supported on a number of different computers.

F. Unit Development Folders

Most software organizations use some sort of project notebook, development workbook, design folder, etc., to collect and maintain the multiplicity of data generated during a large development project. One of the most effective devices of this kind is the unit development folder (UDF) described by Ingrassia [30]. The UDF is a notebook associated with each unit of software in a development project. The notebook is organized by sections (shown in Fig. 16) and provides a uniform collection point for all documentation and code associated with that unit. The principle features of the UDF that differentiate it from many of the other notebooks are

1) it is both a working document and an audited controlled document;
2) it provides management with detailed visibility into the progress and performance of each unit on the project;
3) it aids individual discipline in the establishment and attainment of scheduled milestones;
4) it provides a consistent project-wide approach for unit development and status assessment;
5) it produces the deliverable documentation associated with a given unit on an as-you-go basis.

The UDF is "opened" upon allocation of specific requirements to units, usually at the completion of preliminary design. A unit can be any level of software from a routine to a collection of routines (performing a specific function) which can be developed by one individual. Upon initiation, the responsible manager or chief programmer assigns the allocated requirements and establishes the date for completion of testing for this unit in accordance with the project schedule. These are recorded on the UDF cover sheet (Fig. 17). The assigned programmer then defines his own intermediary milestones and initiates the design activity. When each section is completed,

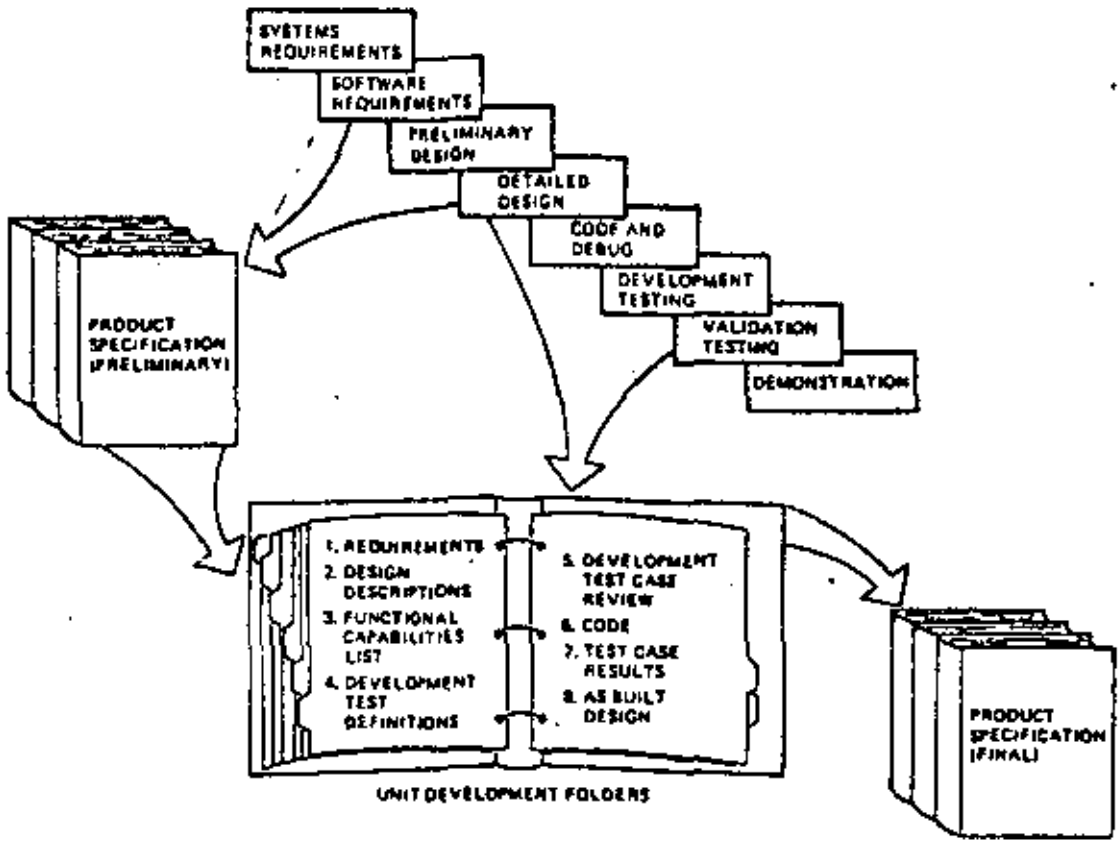


Fig. 16. Unit development folder organization.

the responsible designer signs and dates the section. The review authority (manager, chief programmer, or other reviewers) then reviews the section for completeness and accuracy and approves it. Thus each unit is reviewed frequently for adequacy of development and progress. Additionally, each UDF is continually accounted for and maintained in a location accessible by quality assurance personnel for standards compliance auditing as well as other designers for interface review. The as-built design description is maintained in the exact format specified for delivery to the customer. Thus the large majority of the deliverable documentation is made available by copying that section from all UDF's on the project. A separate "notes" section may be kept with the UDF to record memos, analyses, or other pertinent data relative to that unit.

G. Quality Assurance/Configuration Management

Quality assurance (QA)/configuration management (CM) play an important role for the software project manager in the auditing of on-going activities, status accounting of configured items and change specifications. The QA manager acts as the project manager's "hired gun" to verify that standards are adhered to, procedures are followed, documentation is kept up to date, etc. Generally, well-defined logical standards, rigorously enforced, are well accepted by most programmers. An excellent approach for accomplishing this is to automate as much of the audit as possible. Code standards, structured programming rules, commentary requirements, etc., are readily amenable to automated verification. Often programmers will use these tools themselves during development to assure their passing of the audit at code turnover. At this time, documentation must still be manually audited; however, use of the UDF allows much of this audit to be done during the development, rather than all just prior to documentation release. The role of configuration management is to maintain the current status of all configured items and documentation. For a large project, a requirements or interface change can have a significant ripple effect through much of the remaining documentation and code releases. Controlled updating of both upon incorporation of changes is mandatory for software development sanity. Inevitably, however, it will be necessary to get some "fix" incorporated before all the paperwork reviews and approval are complete. A streamlined quick fix procedure which provides fast test turnaround is mandatory to avoid having the project wrapped up in its own paper. However, this must be done within the framework of a release-based CM system; it must provide a mechanism for identification of all changes (quick or permanent) attached to a released baseline; and it must facilitate the incorporation (or replacement) of the "quick" fix as a permanent, fully documented change.

H. Life-Cycle Maintenance

As the investment in large software systems continues at its exponentially increasing rate, the interest in maintaining these systems at reasonable costs is also increasing. In a recent survey, Lientz-Swanson [31] showed that for business data-processing systems maintenance costs now exceed development costs. DeRoze [32] estimated in 1976 that 60–70 percent of the Department of Defense software dollars are being spent after the software has been tested and delivered by the development contractor. According to Swanson [33], maintenance resources are generally applied to three classes of efforts: corrective—correction of latent problems; perfective—program/documentation improvements; and adaptive—changes to the

MNEMONIC TFAST CUSTODIAN Bradley

SECTION NO.	DESCRIPTION	DUE DATE	DATE COMPLETED	ORIGINATOR	REVIEWER/ DATE
1	REQUIREMENTS	10 Nov 1975	12-25-75 RMB	Bradley	Norman 5 JA 1/6/76
2	DESIGN DESCRIPTION INITIAL CODE TO:	19 Dec 1975	12-29-75 RMB	Bradley	Norman BJ 1/9/76
3	FUNCTIONAL CAPABILITIES LIST	19 Dec 1975	12-29-75 RMB	Bradley	Norman BJ 1/9/76
4	DEVELOPMENT TEST DEFINITIONS	31 Dec 1975	1-23-76 RMB	Norman 1-22-76	Bradley 1-28-76
5	DEVELOPMENT TEST CASE REVIEW	6 Jan 1976	2-16-76	Bradley	Norman BM 2-16-76
6	UNIT CODE	19 Jan 76 / 1 Mar 76	1-16-76 RMB / 3-1-76	Bradley / Bradley	1-28-76 BM / 3-2-76 BM
7	TEST CASE RESULTS	1 Apr 1976	3-4-76 RMB	Jones	Norman 3-11-76 BM
8	AS BUILT DETAIL DESIGN DESCRIPTION	1 Jun 1976	5-4-76 BM	Bradley	Norman 5-4-76 BM

Fig. 17. Unit development folder cover sheet.

environment. Unlike hardware components, software repair is a relatively small part of the maintenance activity. According to the Lientz–Swanson survey, maintenance efforts were roughly distributed as follows:

1) correction of problems—22 percent;
2) Program/documentation improvements—51 percent;
3) changes to data, files, and environment—27 percent.

Consequently, unlike hardware, improving software reliability will not, of itself, drastically reduce software maintenance costs. Munson [34] provides some practical guidelines for consideration of life-cycle problems during the requirements and development activities. Some further thoughts for dealing with each of the classes of maintenance efforts are

Corrective:
1) test every branch and instruction during unit testing;
2) track and verify the demonstration of all requirements during some level of testing;
3) perform both nominal and stress testing whenever possible;
4) perform comprehensive regression tests after each significant change is incorporated.

Perfective:
1) many of these changes occur because the requirements specification was incomplete or unclear; look for potential holes in the specification;
2) provide feedback to the user as early as possible during development as to the capabilities and interface he will have with the systems; enhancements are frequently needed in the user interface area.

Adaptive:
1) build maintainable code; use coding and documentation standards, such as structured programming, to facilitate a maintenance programmer easily following the logic of the developed program;
2) build modular code; simplify interfaces; limit code and data interface functions to control routines;
3) keep documentation up to date; do not allow a new code release into the system until all documentation is complete;
4) maintain the same level of control on the data base as on the code; Use automated, centralized control whenever possible.

VI. Directions

The software industry is currently experiencing unparalleled growth, recognition and influence, but there are some worried concerns on how the train can be kept "on track."

As the 1970's were completed, the software industry could rightly point to having overcome many of the problems headlined the decade before. Many large projects are being completed on schedule and within cost; a methodology for controlling large developments is being employed in many places; and reliable cost models are being demonstrated and validated. Although examples to the contrary still abound, it is realistically possible today to estimate, develop and field a large operational software system with a high probability of success. Certainly, overly ambitious estimates in a competitive environment, continually changing requirements, poor management, lack of qualified personnel, etc., can still undermine any project, but given reasonable development conditions, successful developments of monolithic systems should start becoming the norm. Unfortunately, the processing demands of the 1980's will not likely allow this utopian condition to last for long. Some of the areas which software managers should watch for in the 1980's include the following.

Requirements Definition: All of the requirements methodologies in the world cannot overcome a customer (user or system designer) who continually changes his specification. While many software users have become educated as to the penalties of this process, the rapid growth of data processing into new fields of endeavor only means more potential customers who will have to learn the price the hard way.

Distributed Processing: Many of the procedures and techniques established for monolithic systems are not readily trans-

ferable or upgradable to distributed systems. The multiplication of possible paths, problems of intraprocess timing and deadlock, synchronization of data bases, and nondeterminism of the programming problem provide several dimensions of new complexity to an already difficult problem. Yet, the real hardware cost savings available will drive the industry in this direction at a continually accelerating rate.

Microprocessing: The advent of the very low cost, high performance microcomputer is causing the introduction of digital processing modules in areas previously limited to electromechanical or other pure hardware solutions. In many ways, the lessons learned in the 1960's regarding the need for higher level languages, structured developments, integration principles, etc., are being relearned. It appears we have become so complacent with our newly gained methodologies and techniques that we cannot see many of the same old problems when they reappear in a slightly different form.

Personnel: The data-processing industry is growing at such an accelerated pace into areas of such increased complexity that the availability of trained personnel has been almost completely exceeded. While software development remains one of the most labor intensive of industries (with no real change in sight) the capability of meeting the continually growing processing demands of the 1980's is greatly threatened. Additionally, the increased complexity of most new systems makes lowering of the qualifications for personnel a virtual impossibility. Greatly increased research into means for significantly improving productivity and for simplifying the development processes is mandatory to keep from falling too far behind the demand.

Super Systems: Very large data systems are being configured and then linked with other large systems to create supersystem networks to cope with the data problems (e.g., FET, communication, command, and control, etc.) in today's society. These systems, by their very size and nature, are beyond the scope of understanding of any person or team of persons. Yet, the data-processing industry must demonstrate a capability to rationally evolve this class of systems to provide the services demanded by society while respecting the concerns of society for privacy and protection against "big government/business."

Government Regulations: As software receives a higher and higher percentage of the costs of new systems, the attention it is receiving at all levels of government (e.g., House Government Operation Committee) is also increasing. The solution of government to this perceived "problem" is likely more regulation and standardization. While this is not necessarily bad (the industry itself is continually attempting to standardize on key approaches, languages, etc.), the potential effect of overregulation at a time when innovation is required could be devastating. Recent studies have shown that one impact of the "Brooks Bill" has been to impede the acquisition by the government of new computing equipment. Similar impediments on software by imposition of overly detailed standards or auditing processes could create an environment which is not supportive of increased productivity.

VII. Conclusions

The field of software engineering and management is still very young as an engineering discipline. However, the rapid expansion of the data-processing industry will not allow a quiet time to grow and bring to maturity the technologies and practices needed to smoothly evolve the large systems of the future. The processes and techniques needed to guide future developments must be developed and proven themselves right along with the systems they are being used to support. The ideas and approaches presented in this paper represent a snapshot in time of some of the key principles evolved during the last decade. The complexity of the systems of the 1980's will require significant advancements in most of these approaches to keep pace with the demands for improved productivity while dealing with problems of increased dimensionality.

Ten years ago, no one would have predicted that the art of software engineering and management would have achieved its present state of maturity this soon. Consequently, this author is not going to predict that the ingenuity and intensity of effort needed to meet the challenges of the 1980's will not be forthcoming. However, it will not be a smooth and easy road.

References

[1] J. R. Distasso, J. B. Munson, J. H. Manley, and L. G. Stucki, "Software technology—Key issues of the '80s," *Digest of Papers COMPCON '80*, Feb. 1980, pp. 387–389.

[2] J. B. Munson, "Lessons learned in the application of some modern programming practices," presented at the 1979 IEEE Lake Arrowhead Workshop, Lake Arrowhead, CA, Sept. 5–7, 1979.

[3] R. H. Austing and G. L. Engel, "Computers and society: Report of a workshop," in *Proc. COMPSAC '78*, pp. 301–313, Nov. 1978.

[4] B. W. Boehm, "Software engineering—As it is," in *Proc. 4th Int. Conf. Software Engineering*, pp. 11–21, Sept. 1979.

[5] E. Miller, "Tutorial: Automated tools for software engineering," in *Proc. COMPSAC 79*, pp. 29–31, Nov. 1979.

[6] D. Teichrow and E. A. Hershey III, "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 41–48, Jan. 1977.

[7] M. W. Alford, "A requirements engineering methodology for real-time processing requirements," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 60–69, Jan. 1977.

[8] D. T. Ross and K. E. Schoman, Jr., "Structured analysis for requirements definition," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 6–15, Jan. 1977.

[9] S. S. Lamb, V. G. Loch, L. J. Peters, and G. L. Smith, "SAMM: A modeling tool for requirements and design specifications," in *Proc. COMPSAC 78*, pp. 48–53, Nov. 1978.

[10] R. Nelson, "Software data collection and analysis," Draft Rep., RADC, Sept. 1978.

[11] P. R. Freiman and R. E. Park, "PRICE software model—Version 3, an overview," in *Proc. Workshop Quantitative Software Models*, pp. 32–41, Oct. 1979.

[12] B. W. Boehm and R. W. Wolverton, "Software cost modeling: Some lessons learned," in *Proc. 2nd Software Life Cycle Management Workshop*, pp. 129–132, Aug. 1978.

[13] L. H. Putnam and A. Fitzsimmons, "Estimating software costs," *Datamation*, Sept.–Nov. 1979.

[14] W. Myers, "The need for software engineering," *Comput.*, Feb. 1978.

[15] J. M. Heason, "Computer Applications: Applications, trends, and directions," presented at *COMPCON Fall 77*, Sept. 1977.

[16] N. French, "Programmer productivity rising too slowly: Tanaka," *Computerworld*, 1977, 11(32) p. 1.

[17] T. A. Dolotta and J. R. Mashey, "An introduction to the programmer's workbench," in *Proc. 2nd Int. Conf. Software Engineering*, pp. 164–168, Oct. 1976.

[18] E. W. Dijkstra, "Programming considered as a human activity," in *Proc. IFIP Congr. 1965*.

[19] K. W. Kruse and L. W. Dunant, "A management methodology for testing software requirements," in *Proc. COMPSAC 78*, Nov. 1978, pp. 749–754.

[20] C. G. Davis and C. R. Vick, "The software development system," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 69–84, Jan. 1977.

[21] S. N. Goulding and J. O. Lawson, "Process design engineering—A methodology for real-time software requirements," in *Proc. 2nd Int. Conf. Software Engineering*, pp. 80–85, Oct. 1976.

[22] J. M. Dreyfus and P. J. Karassony, "The preliminary design as a key to successful software development," in *Proc. 2nd Int. Conf. Software Engineering*, pp. 206–213, Oct. 1976.

[23] R. O. Williams, "Managing the development of reliable software," in *Proc. 1975 Int. Conf. Reliable Software*, pp. 3–8, Apr. 1975.

[24] C. L. McGowan, "Management planning for large software projects," in *Proc. COMPSAC 78*, pp. 90–92, Nov. 1978.

[25] F. T. Baker, "Chief programmer team management of production

Ilham, MA, 1973.
[10] S. Hori, "CAM-I long range planning final report for 1973," ITFT Res. Inst., Dec. 1973.
[11] L. J. Peters, *Handbook of Software Design*. New York: Yourdon Press, 1980.
[12] S. L. Pollack, H. T. Hick, Jr., and W. F. Harrison, *Decision Tables, Theory and Practice*. New York: Wiley-Interscience, 1971.
[13] M. Montalbano, *Decision Tables*. Palo Alto, CA: Science Research Associates, 1974.
[14] M. Hamilton and S. Zeldin, "Top-down, bottom-up, structured programming and program structuring," Charles Stark Draper Lab., Massachusetts Institute of Technology, Cambridge, MA, Document E-3238, Dec. 1972.
[15] J. Rood, T. To, and D. Harel, "A universal flowcharter," in *Proc. NASA/AIAA Workshop on Tools for Embedded Computer Systems Software* (Hampton, VA), pp. 41–44, Nov. 1973.
[16] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *SIGPLAN Notices*, vol. 8, no. 8, pp. 12–26, Aug. 1973.
[17] N. Chapin, R. House, N. McDaniel, and T. Wachtel, "Structured programming simplified," *Comput. Decisions*, pp. 28–31, June 1976.
[18] N. Chapin, "New format for flowcharts," *Software Practice Experience*, vol. 4, pp. 341–357, 1974.
[19] P. G. Hebalkar and S. N. Zilles, "TELL: A system for graphically representing software design," IBM Res. Rep. RJ 2351, Sept. 1978.
[20] T. Demarco, *Structured Analysis and System*. New York: Yourdon, 1978.
[21] J. D. Warnier, *Logical Construction of Programs*. New York: Van Nostrand-Rheinhold, 1977.
[22] M. A. Jackson, *Principles of Program Design*. London, England: Academic Press, 1975.
[23] R. C. Linger, H. D. Mills, and B. F. Witt, *Structured Programming Theory and Practice*. Reading, MA: Addison-Wesley, 1979.
[24] H. F. Ledgard, "The case for structured programming," *Nordisk Tidskrift for Informations Behandling, Sweden*, vol. 13, pp. 45–47, 1973.

Software Quality Assurance: Testing and Validation

JOHN B. GOODENOUGH AND CLEMENT L. McGOWAN

Abstract—There are many pitfalls for the unwary hardware engineer who must develop software. In particular, hardware quality control procedures and concepts can easily be misapplied. The purpose of this paper is to help hardware-oriented engineers apply some of the quality assurance lessons learned by software engineers. We first discuss how software is and is not analogous to hardware, and then outline recommended software engineering approaches for developing high-quality software products. We concentrate on methods for preventing and detecting software errors.

I. INTRODUCTION

A. The Software-Hardware Analogy

THERE ARE many similarities between software and hardware design and development, but unless one is careful, the similarities can be misleading. For example, hardware failures are sometimes considered analogous to software failures, but often the wrong analogies are drawn. In particular, component deterioration is the usual cause of hardware failure. In contrast, software failures are almost always *design errors* that show up only when the software is used under certain conditions.

For example, a hand calculator was once manufactured that did not correctly compute the sine function for all argument values. All units computed the same incorrect value for particular arguments. This was clearly a design error—the circuit for calculating sines was incorrectly designed. Since the calculator gave correct answers for most arguments, the problem was only discovered by a few users. This is typical of design errors, which are latent until the hardware or software is exercised under the appropriate conditions.

Because software errors are analogous to hardware design errors, software quality assurance focuses on techniques for getting the design right. Adapting hardware quality assurance procedures to software development means focusing on the procedures used to prevent and detect design errors. This is the proper analogy between software and hardware quality assurance. It illustrates our point that unless one makes the proper analogies, intuitions about how to ensure hardware quality will lead one astray when working with software.

A comparison of hardware and software life cycles is given in Fig. 1. This figure indicates the phases of developing a new hardware or software product. Although Fig. 1 provides an

Manuscript received February 13, 1980, revised April 14, 1980.
The authors are with SofTech, 460 Totten Pond Road, Waltham, MA 02154.

0018-9219/80/0900-1093$00.75 © 1980 IEEE

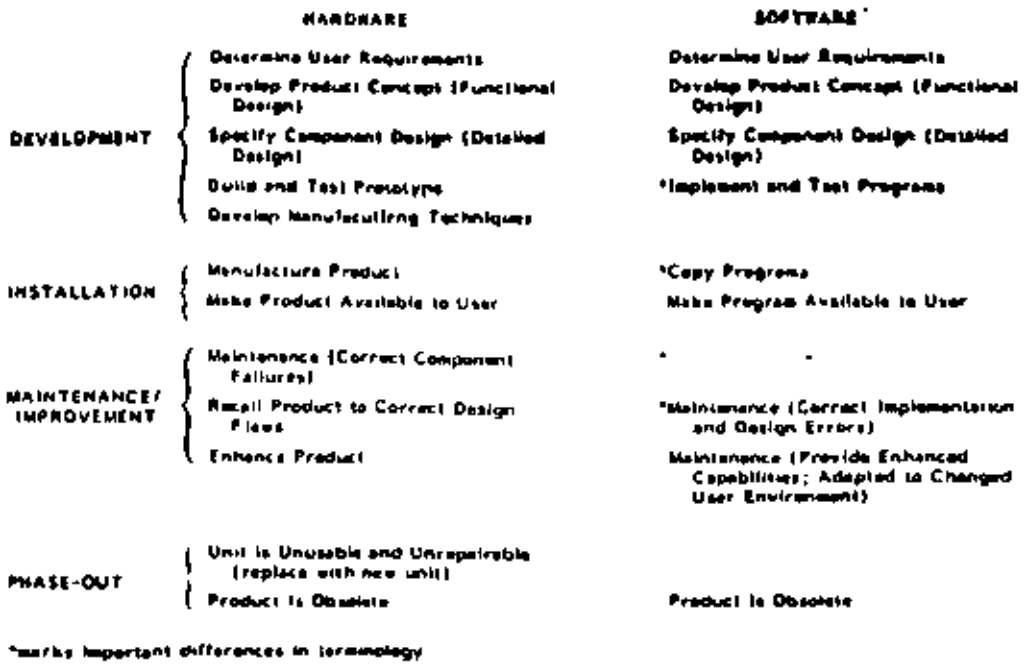


Fig. 1. Corresponding steps in software and hardware product life cycles.

oversimplified and idealized description, it shows clearly that similar terms in the two fields sometimes have radically different meanings. Perhaps the most important differences are

1) coding programs is not equivalent to manufacturing a product;
2) maintenance refers to quite different processes (see Lehman [16] for further discussion of this point);
3) program development and test is conceptually similar to developing and testing a hardware prototype, but in software, the "prototype" is the first system that gets delivered to users.

The focus of this paper is software testing and quality assurance. As Fig. 1 indicates, we will therefore be discussing procedures and concepts aimed at producing a high quality *design* (in the hardware sense). Testing and quality control procedures used in manufacturing or maintaining a hardware product are not directly relevant to software quality assurance.

Finding design errors is difficult for both hardware and software engineers, and finding such errors is more difficult as the complexity of the product increases. In general, software products are equivalent to very complex hardware products, so it is no wonder developing high-quality software is difficult and costly.

B. Sources of Software Errors

The basic steps in developing a software system are

1) defining user requirements;
2) deciding what functions and major components a system must provide to meet these requirements;
3) designing and specifying the intended behavior of individual software components;
4) implementing (i.e., coding) software components.

Each of these software development activities is subject to error

1) construction errors—failure of software components, as implemented, to satisfy their specifications;
2) specification errors—failure to accurately specify the intended behavior of a unit of software construction;
3) functional design errors—failure to establish an overall design able to meet identified requirements;
4) requirements errors—failure to identify user needs accurately, including failure to communicate these needs to software designers.

Different quality assurance techniques are required to deal with these errors, since they arise at different stages of software development. In this paper, we will concentrate on design, specification, and construction errors.

Software is always tested before being released for operational use and testing is typically the last activity before release. This gives testing high visibility to developers and users and often leads to an undue reliance on tests as the means of ensuring software quality.

Since tests are performed only after code is written, they are a costly method of detecting errors, especially specification and design errors. Correcting these errors may require discarding some software that has already been written and tested. A cost effective approach to reducing software errors must attempt to prevent and detect errors as soon as possible. Ideally, software design errors are detected during the design phase, before specifications of software components are written, and inconsistencies between specifications and the design are detected before code is written.

In Section II, we will discuss principles and practices underlying the development of software tests, and in Section III, we will discuss an integrated approach to software quality assurance. The integrated approach discusses what quality assurance procedures and principles should be applied at each phase of software development, rather than focusing solely on the development and evaluation of test cases.

II. Software Testing Principles

From an engineering point of view, software testing has traditionally been *ad hoc*, with few principles and no theoretical work indicating how to construct an adequate set of tests or how to measure the adequacy of a set of tests that someone

has constructed. In the last few years, however, some theoretical work has increased our understanding of how to construct tests. We will discuss this recent work to show how the basis for an engineering approach to software testing is developing.

Software testing involves the execution of programs with selected inputs, called test cases. The results of test executions are then used to decide whether the program is operating acceptably.

The most common objective in software testing is to determine whether a program is correct, i.e., whether the program produces specified outputs when presented with permitted inputs. Although correctness may at first seem to be the most important property a program can have, this is by no means the case, particularly for large software systems. (See [20] for a more detailed discussion.) Large programs are often so complex they never completely satisfy their specifications, and yet, they may be quite usable because failures are encountered infrequently in practice, and when they do occur, their impact on a user is acceptably small. Hence correctness is not necessary for a program to be usable and useful. Nor is correctness sufficient. A correct program may satisfy a narrowly drawn specification and yet not be suitable for operational use because in practice, inputs not satisfying the specification are presented to the program and the results of such incorrect usage are unacceptable to the user. If a program is correct with respect to an inadequate specification, its correctness is of little value.

Consequently, although testing for correctness is the most common and best understood testing goal, correctness is by no means the only important property of usable software—reliability, robustness, efficiency, and other properties (see [12]) are also of significant importance. But these properties are less commonly the focus of testing activities.

In designing correctness tests, it is important to keep a few principles in mind.

1) "Black box" tests (i.e., test cases chosen without knowledge of how a program has been implemented) cannot ensure that all correctness errors are detected unless the tests are exhaustive, i.e., consist of all inputs in the program's input domain. Exhaustive tests are almost never practicable, however. Input domains are just too large; often they are not finite.

2) Even "glass box" tests (i.e., tests chosen with full knowledge of how a program has been implemented) are not necessarily adequate to detect all correctness errors. For example, selecting tests so all branch conditions in a program or even all execution paths through a program are exercised will not detect an error if a branch or path that should be present in the program is missing (see [12]).

3) The most effective way to design tests is to hypothesize certain software errors and then to select test cases that will fail if the errors are present. We discuss aspects of this approach below.

Current research approaches for developing correctness tests fall in two categories—deterministic and probabilistic. Deterministic methods select tests that will fail if certain kinds of errors (and only those kinds of errors) are present in a program. Probabilistic methods provide estimates of the likelihood of undetected errors remaining in a program without ever fully guaranteeing that all errors (of certain kinds) have been eliminated. Probabilistic methods are aimed not so much at defining how to select test data but rather at evaluating the effectiveness of tests in uncovering errors. An ideal software testing method gives both a reliable measure of effectiveness and, if the measure indicates more testing is needed, shows a tester where further testing effort will be profitable.

For example, the deterministic approach is illustrated by Chow's [6] technique for testing communications software protocols and programs implementing them. Such protocols can be described in terms of finite state machines. Chow has shown that given an upper bound on the number of states in the correct finite state machine, a set of test cases can be generated that will detect all errors due to missing states and missing or incorrect state transitions. A program implementing the protocol can be deterministically tested with these inputs to determine if it has realized the intended design, i.e., successful execution implies the implementation is correct. West [31] describes a similar procedure for detecting system deadlocks, potential losses of messages, and other communications protocol errors. West also requires that the protocols be modeled as finite state machines.

Another example of the deterministic approach is the one developed by White and Cohen [32]. They attempt to detect only errors due to incorrectly written branch conditions in a program. They partition the input domain of a program into equivalence classes such that each element of a class causes execution of the same program control flow path. Their testing strategy describes how to select test data to determine if the boundaries defining a class have been shifted from their correct position. For example, given a boundary defined by an ordering relation, e.g., $3X + 2 > Y$, they show why in general three test cases are needed to show whether the boundary has been correctly specified—two test points on the boundary and one off the boundary, e.g., (3, 11), (−1, −1), and (1, 4). A more naive approach to test case selection would assume that two test cases would suffice—one in which the relation was satisfied, and one in which it was not. But Cohen and White's analysis shows why two cases are insufficient to detect errors in the coefficients. Showing how intuitive testing approaches can be inadequate is an important result of deterministic testing research.

The common elements in deterministic testing approaches are 1) their focus on detecting only certain kinds of errors in the absence of other kinds of errors, and 2) their development of methods for selecting test data that is guaranteed to detect these errors if they are present. A productive area for further research is defining test case selection methods for additional classes of errors. Such research is needed to provide a firm theoretical basis to guide software engineering practice.

In constrast to the deterministic approach to testing, in which one attempts to find test cases guaranteed to detect certain kinds of errors, DeMillo *et al.* [8] have developed a probabilistic method for assessing the quality of a test set without knowing precisely what errors the test set is able to detect. Their method requires "mutating" (i.e., modifying) the program to be tested by introducing small changes that are likely to be errors. The original program and the mutated programs are tested using the same set of test cases. The quality of the test set is determined by the number of mutants generated and by how many fail to pass the tests. Ideally, all mutants fail. When mutants are not eliminated by the tests, one must either attempt to find test data for which the mutants will fail or demonstrate that the mutants are equivalent to a correct program.

The principle underlying the mutation approach to testing is "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors" [8].

There is no certainty that all errors are discovered if the set of tests eliminates all incorrect mutants. The assumption, however, is that in practice any remaining errors are not important. Research is currently underway to see to what extent this assumption holds and to see what kinds of program mutations are most useful for validating test sets [1].

Software testing is not currently an engineering discipline. The theoretical basis needed for such a discipline simply does not yet exist. But our examples of new testing approaches show that some progress is being made.

While waiting for theory to catch up with the needs of software developers, today's software engineers must apply rules of thumb learned by experience. These rules are well explained by Myers [23], [24], whose books contain what is probably the best currently available collection of good practices for developing software tests.

Perhaps the most important thing to remember about software testing is that although it is necessary, it cannot be the sole means of ensuring software is reliable, robust, and useful. To obtain software having these properties in sufficient measure, one must use appropriate quality assurance techniques at each stage in the software development process. Testing is only one of these stages.

III. An Integrated Approach to Quality Assurance

As noted previously, software development is analogous to the building and testing of a hardware prototype. Accordingly, attempts to assure software quality should attend principally to the design process and should analyze its outputs. Indeed, H. Mills [18] recommends that

> Given double the budget and schedule (to test the sincerity of a requirement for ultrareliability), do not spend the extra on testing—spend it on design and inspection.

In this section, we review the software development process's major activities, namely design, construction, and testing, from the perspective of what proven quality assurance principles and procedures should be applied. Coverage of all the applicable techniques and tools is not possible here. Rather, the intention here is to survey the state of the art as practiced with suitable references supplied for reader follow up.

A. Reducing Design and Specification Errors

Software design can be characterized as allocating requirements to the components of an architecture. This characterization stresses that a design consists of parts (modules) and their interconnections (interfaces) for the purpose of realizing a given set of requirements. Clearly this presupposes that the software requirements are defined and analyzed prior to the design activity. Without first satisfying this important presupposition all subsequent efforts to assure a quality product are, at best, misguided.

Presently, there are quite a few competing "design methodologies" offering a software designer (or a design team) prescriptions of explicit steps to follow as well as a specific, graphic notation for representing the resultant design. We mention here some of the more prominent approaches: structured design [21], [22], [29], [36], the Jackson method [15], the Warnier–Orr approach [25], [30], the structured analysis and design technique (SADT) [9], [27], the requirements engineering validation system (REVS) [2], the systematic design methodology (SDM) [3], [14], the operational software concept [26], [35], higher order software (HOS) [13], and the architecture definition technique (ADT) [4].

Generally by using a top-down strategy, each of these methods develops a software system design as a group of communicating modules. Such top-down strategies proceed from the general to the particular by first defining the context, the function, and the interfaces for any module before specifying the internal details of that module. However, the various methods are fundamentally different in what they regard as important and in how they attack a design problem. Naturally the graphic notation associated with a particular method is well suited to representing "important" system aspects. That is, the very notation for representing a design architecture enforces abstraction by excluding or deferring less important aspects (as judged by a method's creators). Some of the system aspects that have been used by various design methods for creating a "good" design are: control flow, data flow, the input and output data structures, the internal data-base structure, stimulus/response threads through the system, hierarchical decomposition of system functions, system states (or modes), and the transitions between them, dependencies between functions, and major system features that should be easily changeable.

For a software quality assurance viewpoint, there are many advantages to adopting a particular design method and graphic notation for representing a design. Such a standard establishes a basis for training and for project communication. This eases the introduction of new personnel during development and provides a useful "mental framework" for the eventual maintainer. Once selected, design standards enhance the efficacy of all internal design reviews by providing some objective criteria that can be uniformly applied when assessing design "qualities" such as completeness, correctness, and clarity. Moreover, standard design documentation can—and should—be supported by software tools that store the design information for subsequent queries, that automatically produce a standard formatting, and that report on the consistency of the design information entered thus far. Some research and development efforts (based on particular design methods) have extended the design support tools to include simulations or "animations" of the currently stored design. In summary, with respect to using a design method experience indicates that any reasonably systematic strategy is better than none.

Besides design reviews internal to the development team, several formal design reviews for the benefit of senior management, the customer, and an independent quality assurance group are advisable. Established practice now holds two such formal exercises called the preliminary design review (PDR) and the critical design review (CDR). The PDR assesses the proposed design architecture for logical and technical feasibility. All of the software components (both procedural and data) should be functionally specified with interfaces identified in some detail. In addition, a plan scheduling subsequent design, implementation, testing, and integration tasks must be provided. Draft user and maintenance manuals and a proposed acceptance test plan are appropriate. The CDR focuses on implementation and performance feasibility. It typically requires the detailed algorithms, data structures, and interface formats along with memory and execution time budgets for each software elements. Usually satisfying the CDR "action items" is the prerequisite to initiating actual coding. With respect to internal and formal design reviews Glass [11] has said the following.

> Most important, the success or failure is dependent on people. The people who attend must be intelligent, skilled, knowledgeable

in the specific problem area, cooperative, and have the time available to dedicate themselves to the review. Only the interactions of capable people can make a design review successful.

The quality assurance role in a design review is to examine the design for completeness (are all requirements addressed?), for quality (using objective criteria derived from the selected design method), and for adherence to standards.

B. Reducing Construction Errors

Software construction is concerned with the design of module details and then with expressing these details in executable code. Algorithm design, data layout, and access considerations are central to this activity. Already there is a rich body of knowledge about specific algorithms and data structures as well as their relative performance tradeoffs. By demonstrably helping to improve the quality of software construction, two general practices have gained widespread acceptance—structured programming for detailed routine and data design and a higher order language (HOL) for coding programs.

Structured programming involves the hierarchical fabrication of programs and data structures by means of the repeated application of some basic composition rules. For example, in "structured coding" basic statements (such as arithmetic evaluation and assignment) may be combined into compound statements only by using statement sequencing, conditional selection of a group of statements to be executed, or conditional iteration of some statements. That is, the unconditional branch or GO TO statement is excluded from structured coding (sometimes erroneously characterized as GO TO-less programming). These restrictions generally simplify a routine's flow-of-control logic and have demonstrably reduced a significant source of coding errors. In less than a decade, structured programming has gone from a proposed approach [7], [19], [33] to an operational standard for most major software developers.

Coding standards incorporate aspects of structured programming with rules for indentation (to reflect nesting), header comments, identifier names, and the length of routines (to encourage short, functional procedures). A major premise of structured programming is that programs are to be read by people as well as by computers. Proceeding on this premise, quality assurance activity has recently included code reading. The traditional desk checking by the programmer is profitably augmented with a more formal peer code review [10] that is scheduled, conducted using checklists for guidance, and tracked with all the trappings of reports, action items and follow ups.

Using an HOL makes software construction much more reliable and productive than using assembly language. Good optimizing compilers can now produce code that is within 25 percent in both space and execution time of well-crafted assembly language. HOL code is so much easier to produce, checkout, maintain, and modify that it should be used for all but the most stringently constrained target environments. Indeed, Fred Brooks claims, "I cannot easily conceive a programming system I would build in assembly language" [5]. Interactive debugging in the symbolic terms of the HOL source program substantially speeds routine check out. Modern typed HOL's (such as Pascal [34]) catch many data interface and misuse errors during compilation. In some cases, detailed design can be better represented in a suitable source HOL (because of the automatic check out and cross referencing) than in flowcharts or in some program design language (PDL).

C. Extending Testing Principles to Software Systems

The burden of producing detailed design documentation and of unit testing various routines often obscures what the essential product of software development is. The real goal should be to place a system (including people, machines, and software) into satisfactory operation. Consequently system and software integration are prominently placed on the critical path. To better reduce and control the associated risk, software integration and testing efforts should be distributed over as much of the coding effort as possible. Indeed, a more accurate measure of current project development status is "what percentage of the total number of software modules identified during architectural design have been coded and integrated into an operational subsystem?" Such a measure is far superior to the more frequently used "how many lines have been coded?" or "how many modules have been unit tested?" Clearly a percentage-of-modules-integrated measure dictates using some variant of a top-down implementation and testing sequence [17]. A significant strategy is to identify some operational subsets (or "builds") of the intended system and to develop and deliver in succession these subsets [28]. This approach provides an early check out of the major software and user interfaces while helping to identify some important but nontechnical potential problems (e.g., user training, delivery format and mechanism, coordination of multiple contractors, etc.). In fact, testing should drive development in the sense that plans for testing software subsets should determine the implementation sequence.

Testing can be regarded as gathering information on the software's reliability. Thus viewed, test requirements, plans, and procedures as well as design reviews are as much a part of testing as is exercising software on a machine. It is beneficial to have a separate project test or quality assurance team address these issues. For example, a test procedure must clearly define the series of actions required to verify that a product meets its requirements. These actions may include

1) configuring—arranging the software, hardware, people, and logistics for the test;
2) conditioning—bringing the system to the initial state for testing;
3) introducing data—entering either "live" or simulated data;
4) starting the test—initiating and synchronizing where necessary;
5) collecting data—gathering snapshot, frequency, and resource utilization information;
6) displaying results—selecting and formatting results as directed;
7) analyzing results—comparing actual to expected.

A variety of software tools assist in conducting test procedures. These include environment simulators, test data generators, and test coverage analyzers.

In summary, software development and test is fairly analogous to the development and testing of a hardware prototype; design errors predominate. This insight helps in identifying some major sources of software errors. A serious software quality assurance effort that focuses on these error sources must be active from a project's inception. The issues of requirements testability, traceability, and cost must be resolved early. Design reviews must be conducted for the purpose of finding design errors sooner rather than later. Proven methods for designing, programming, making the evolving product

visible, and controlling different system configurations can be standardized and then supported by a suitable software project library with associated procedures. Of course, execution testing remains the principle means for determining the current status of a software product.

REFERENCES

[1] A. T. Acree, R. A. DeMillo, T. J. Budd, R. J. Lipton, and F. G. Sayward, "Mutation analysis," NTIS Rep. ADA-076575, Sept. 1979.

[2] M. Alford, "A requirements engineering methodology for real-time processing requirements," *IEEE Trans. Software Eng.*, vol. 3, pp. 60-68, Jan. 1977.

[3] R. Andrew, A systematic approach to the design and structuring of complex software systems, Ph.D. dissertation, Sloan School of Management, Massachusetts Institute of Technology, 1978.

[4] C. Bachman and J. Bouvard, "Architecture definition technique: its objectives, theory, process, facilities and practice," in *Proc. ACM SIGFIDEF Data Description Access and Control*, pp. 257-305, 1973.

[5] F. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.

[6] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Software Eng.*, vol. 4, pp. 178-186, May 1978.

[7] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. New York: Academic Press, 1972.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Comput.*, vol. 4, pp. 34-43, Apr. 1978.

[9] M. Dickover, C. McGowan, and D. T. Ross, "Software design using SADT," in *Proc. Nat. ACM Conf.*, pp. 125-137, 1977.

[10] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, pp. 182-211, 1976.

[11] R. Glass, *Software Reliability Guidebook*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[12] J. B. Goodenough, "A survey of program testing issues," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 316-340.

[13] M. Hamilton and S. Zeldin, "Higher order software—A methodology for defining software," *IEEE Trans. Software Eng.*, vol. 2, pp. 9-32, June 1976.

[14] S. Huff and S. Madnick, An extended model for a systematic approach to the design of complex systems, NTIS Rep. ADA-058585, 1978.

[15] M. A. Jackson, *Principles of Program Design*. New York: Academic Press, 1975.

[16] M. M. Lehman, "Programs, programming, and the software life cycle, this issue, pp. 1061-1078.

[17] C. L. McGowan and J. R. Kelly, *Top-Down Structured Programming Techniques*. New York: Van Nostrand, 1975.

[18] H. Mills, "Software development" in *Research Directions in Software Technology*, P. Wegner Ed. Cambridge, MA: M.I.T. Press, pp. 87-105.

[19] —, "Top-down programming in large systems," in *Debugging Techniques in Large Systems*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1971, pp. 41-55.

[20] J. D. Musa, "The measurement and management of software reliability," this issue, pp. 1131-1143.

[21] G. J. Myers, *Reliable Software Through Composite Design*. New York: Van Nostrand, 1975.

[22] —, *Composite/Structured Design*. New York: Van Nostrand, 1978.

[23] —, *Software Reliability: Principles and Practices*. New York, Wiley, 1976.

[24] —, *The Art of Software Testing*. New York: Wiley, 1979.

[25] K. Orr, *Structured Systems Development*. New York: Yourdon, Inc., 1977.

[26] J. Rodriguez and S. Greenspan, "Directed flowgraphs: The basis of a specification and construction methodology for real-time systems," *J. Syst. Software*, vol. 1, pp. 19-27, Jan. 1979.

[27] D. Ross, "Structured analysis: A language for communicating ideas," *IEEE Trans. Software Eng.*, vol. 3, pp. 16-34, Jan. 1977.

[28] D. Schultz, "A case study in system integration using the build approach," in *Proc. Annu. ACM Conf.*, pp. 143-151, Oct. 1979.

[29] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, pp. 114-139, 1974.

[30] J. Warnier, *Logical Construction of Programs*. Leiden, Germany: Stenfert Kroese, 1974.

[31] C. H. West, "General technique for communication protocol validation," *IBM J. Res. Develop.*, vol. 22, pp. 393-404, 1978.

[32] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. 6, pp. 247-257, May 1980.

[33] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221-227, Apr. 1971.

[34] —, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35-63, 1972.

[35] K. Wong and J. Engelland, "Operational software concept: A new approach to avionics software," in *Proc. AIAA Digital Avionics System Conf.*, 1975.

[36] E. Yourdon and L. Constantine, *Structured Design*. New York: Yourdon Inc., 1975.

Specifying Software Requirements

RAYMOND T. YEH, SENIOR MEMBER, IEEE, AND PAMELA ZAVE

Abstract—Many of the problems of software system development can be traced to poor understanding or specification of what the system is supposed to do. Much attention is now being given to producing requirements specifications that are understandable, formal, complete, and modifiable. The structure and content of conceptual models for problem understanding are discussed; such models can form the basis for requirements analysis. Techniques for "tuning" and formally specifying the conceptual model are surveyed, and the proper scope of a software requirements document—including nonfunctional requirements such as performance—is explained.

I. Introduction

IT IS BECOMING evident that the central problem in large data processing complexes is *continuity and change*, and *not* system development. By current estimates [6], maintenance cost is the predominate cost over the life cycle of a software system. If this trend continues, most organizations will eventually be strangled—most of their resources will be used for maintenance and few, if any, will be available for new development. Ironically, much of the maintenance cost is due to poor requirements specification, as pointed out by many authors [5], [7], [24].

The neglect suffered by the requirements phase in software engineering is not surprising in light of the fact that many engineering disciplines evolve in a bottom-up fashion. In the case of software engineering, programmers had to bend backwards to accomodate the computer during the early days of computing. The programmer's task was to code small algorithms in the machine language of a particular computer. It was considered good practice to use as many clever tricks as possible to "squeeze" as much time and memory out of the machine as possible. This rather restrictive environment left little freedom to individual programmers. As the complexity of problems has grown and high-level languages have emerged, individual programmers can no longer comprehend their problems entirely; hence the necessity for "design" has become evident. While "design" is an intermediate step between the perceived world and the actual product (the code), it assumes that the designer has an *a priori* understanding of the problem. This assumption is often not true, especially for large, complex problems. Thus evolution has finally lead us to study the problem of systematic analysis and derivation of software requirements.

Although many techniques and tools have been developed during the last few years [1], [26], [28], [32], to aid in the analysis and specification of software requirements, many problems remain in current practice. The quality of different requirements documents varies greatly, depending on the background of the user/proponent. Such documents are usually masses of detail, often comprising volumes of natural language statements. There are no standards about what should or should not be included in a software requirement document, nor is there any formal interaction between user and developer while the requirements are being developed.

We believe that part of the problem encountered in developing requirements is a lack of understanding of what should be the activities during this phase. In many techniques proposed, design as well as implementation is considered as part of the requirements document. In our opinion, three main activities should be pursued during the requirements phase

1) *problem identification*—identify and describe the needs of a system for certain purposes;
2) *problem understanding*—collect and analyze information about the system and its environment, as well as their interaction;
3) *problem specification*—describe the behavior of the system.

This paper is intended to be a tutorial introduction to a systematic approach for generating software requirements. As such, we will look at some of the basic issues in requirements generation: What information should be included in a requirements document? How should requirements be stated? What are the fundamental steps in the systematic derivation of requirements? Before we can proceed to some of these questions we need first come to a consensus as to what a requirements document is, and what it is for.

A "requirement" is something mandatory. It conveys an essential property or condition that the system must satisfy. For example, a sentence such as "Weights of up to 10 tons must be supported" would be part of the requirements for a bridge. A precise description of the requirements for a software system is called a software requirements document (SRD). We may thus define a software requirements document as "a set of precisely stated properties or constraints that a software system must satisfy."

While the SRD specifies *what* the system is to do, it must not constrain *how* it is to be done, for that is the province of design. If requirements analysts think in terms of a particular system structure, they will inevitably lock that design into the requirements, even if it is not the best solution to the real problem. The bridge requirements drawn up by a municipality should not specify a suspension bridge, for instance, because a good civil engineer may propose a much more cost-effective structure.

To put it another way, an SRD establishes boundaries on the "solution space" of the problem of developing a useful software system. These boundaries are made up of properties and constraints which can be used to test whether a proposed

Manuscript received April 9, 1980; revised May 5, 1980. This work was supported in part by the U.S. Air Force under Contract F 49620-79 C 0001 and by the U.S. Army under Contract DASG 60-80-C-0074.

The authors are with the Computer Science Department, University of Maryland, College Park, MD 20742.

0018-9219/80/0900-1077$00.75

solution (design or actual implementation) is indeed valid. An SRD is a behavioral description of a "family" of solutions, while a design is a structural description of a particular solution.

The SRD could be used by many different groups throughout the life cycle of the system it specifies. First of all, it is the basis for a contract between the customer and the developer. Additionally, the SRD is the basis for design, for developing the user's manual, and for controlling the evolution of the system. It is not surprising that lack of a good requirements methodology has contributed significantly to the cost of software. In the next section, we shall discuss certain requirements for SRD's. Subsequent sections present a systematic approach for developing SRD's as a synthesis of some of the existing approaches.

II. Requirements for Requirements

What is a set of criteria for a good SRD? We may derive such criteria by observing the uses of the SRD

1) as a means of communication among users, experts, analysts, and designers;
2) to support design validation;
3) to control the operations and evolution of the system it specifies.

Since the SRD must serve such a variety of people, perhaps its most important property is understandability. The greatest barrier to human understanding presented by today's systems problems is their complexity, and so the SRD must use any available means to decompose complexity. The two most familiar approaches to decomposition of complexity can be termed "partition" and "abstraction." Partition is describing a whole (the complete system) in terms of its parts. Abstraction is describing a complete system very abstractly, and then adding successive layers of detail to the description. This process of elaboration is often called "stepwise refinement," and the resulting description has a hierarchical structure. Another promising strategy is "projection" of the description onto "orthogonal" components, each component being a description of the complete system, but only mentioning a subset of its properties. An obvious analogy is that a two-dimensional architectural drawing is a projection of the description for a complete three-dimensional building onto a particular view. There is much to learn about how and when to use each decomposition technique, but surely they can help a person understand the SRD one piece at a time.

In order to be able to validate his design, the designer must be able to ascertain that it will lead to a system fulfilling the contract, by verifying that certain properties of the design satisfy the requirements. This requires that the SRD should have the property of formality. Furthermore, a formal specification can be incorporated into a design data base, and subjected to automated consistency checkers and other design tools. It may even be possible to make requirements specifications interpretable (i.e., "simulatable"). This would make testing available as a validation tool [36].

Two issues are involved in system evolution control: avoiding making changes, and making changes easily under the constraint that "small perturbations in the environment should cause correspondingly small changes in the system." To avoid unnecessary system changes, the SRD must be complete in that all constraints and assumptions are explicitly stated. This includes the often-neglected *nonfunctional* requirements, such as constraints on the performance, reliability, and cost of a system (see Section V). Systems undergo continuous evolution [4], however, and so the SRD itself must be easily updated. Thus the SRD must also have the properties of "completeness" and "modifiability" [3]. The work of Simon [30] suggests that both properties may be enhanced by modeling the environment of the system to be developed. This idea will be discussed in Section III.

To summarize, the criteria on which to judge an SRD are: understandability, formality, completeness, and modifiability.

III. Conceptual Modeling as a Basis for Requirements Specification

In order to write a good SRD, an analyst must first have a good understanding of the problem. Such an understanding is usually achieved through mental models that exist in the minds of analysts. However, for complex systems, not only is it impossible for any individual to comprehend the whole problem, but such mental models tend to be incomplete and ambiguous due to the inherent complexity of the system to be designed. It is, therefore, necessary to construct an explicit formal model from the very beginning and use it as a basis for analysis and hence aiding in the understanding of the problem.

A conceptual model is used for problem understanding, and therefore can be regarded as a complex "knowledge structure" which consists of a highly structured collection of concepts and their interrelationships. An analyst may gain understanding of the problem by "navigating" through such a knowledge structure. Thus the modeling process consists of a collection of activities involving information gathering, analysis, and structuring. It involves both bottom-up processes, e.g., information collection and aggregation, and top-down processes, e.g., classification and decomposition. The SRD is then a precise description of our understanding through modeling, presented in separate parts which are "orthogonal" to each other.

The question before us now is "what to model?" It is instructive to quote the following [30]: "A man, viewed as a behaving system, is quite simple. The apparent complexity of his behavior over time is largely a reflection of the complexity of the environment in which he finds himself." Large software systems are known to undergo continuous changes. These changes are primarily due to environmental perturbations: new machine being installed, new applications added, etc. In order to "design for change" so as to minimize the evolutionary impact on the system, the conceptual model not only needs to model the system to be designed, but also the environment in which the system is embedded.

Abstractly, we may regard the whole environment as a set of asynchronously interacting processes in which the system to be designed is one or several of the processes. We provide the basic structure of a simple conceptual model in Fig. 1.

The simple model depicted in Fig. 1 describes the system and the environment as two interacting subsystems. The features of the environment to be modeled are

1) *entities*—identifiable objects in the environment;
2) *events*—"happenings" in the environment.

The "states" in the system consist of information about relevant entities and their attributes in the environment. Consider for example, the "history of patients" in a patient monitoring system. This state may contain the following information: patient name, patient ID number, measurements made on a patient's health factors during the last six months. The "actions" of the system are responses of the system to the events in the environment; these actions may, in the

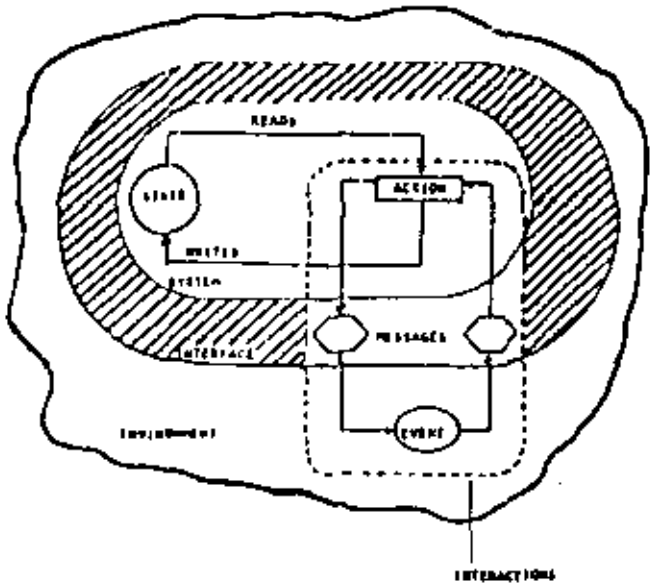


Fig. 1. Basic structure of a simple conceptual model.

trigger other events in the environment. Note that the actors or processors of actions could be either human or machine (which may not be specified at this level). The exchanges between the environment and the system are carried out through messages or some other such communication facility. Of course this information alone would not be sufficient to develop the model. To do so, we must include in the "knowledge base" constraints, e.g., relationships among entities such as part/whole, instance of, sequencing of events, etc.

Note that the conceptual model may be constructed by many people and hence is a result of consolidating many different views. In the process, a great deal of redundant information may have been collected in the model. While redundancy facilitates understanding, it may also restrict a designer's freedom of choice. Thus redundancies need to be filtered out of the conceptual model to form the SRD. We view then the SRD as the minimal set of essential information extracted from the conceptual model which can completely characterize the system to be developed.

The model presented in Fig. 1 has already partitioned the global environment into separate parts. This model can be further refined to model more sophisticated systems, such as operating systems, by partitioning both the environment and the computer system into sets of asynchronously interacting processes.

An Example

Let us consider a simple example by developing a patient-monitoring system. The initial "statement of needs" is given as follows.

"A patient-monitoring system is to be designed 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 those factors on a periodic basis specified for each patient, keeps a history of the factor values, compares the factor values with safe ranges specified for each patient, and notifies the nurse station if any factor is outside of the safe range."

In general, such statements of needs do not provide all the information needed for developing the system. Additional information should be collected and organized in order to model

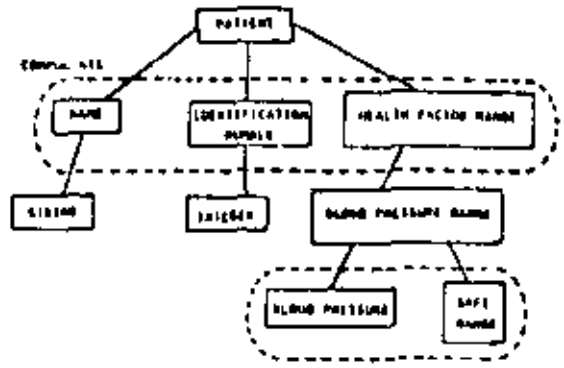


Fig. 2. An aggregation hierarchy.

the system and its environment. For the patient-monitoring system, for example, we may collect the following from the needs statement:

1) *entities*—hospital, patient, nurse, safe range, etc.;
2) *events*—patient-monitoring, factors-out-of-safe-range;
3) *system capabilities (actions)*—measure factors, keep history;
4) *states*—patient-factor-history, factor-range;
5) *anticipated changes*—frequencies of factors measured, factors.

In general, we may obtain additional relevant information for our model based on the following two assumptions [35].

1) The data-processing requirements of an organization can usually be determined from the collection of tasks required to be performed within an organization.

2) In each organization there is a common underlying language used for communication among tasks and/or the task performers. This language consists of reports, files, on-line inquires of existing data-base systems or query models, messages and verbal communication among task performers.

We shall not be concerned here about information gathering or forms for collecting tasks, data, etc. Interested readers are referred to [35].

The next step in modeling is to construct a model of the relevant part of the environment by structuring. For structuring entities (data objects), there are two general approaches: generalization and aggregation [31]. The generalization hierarchy organizes data objects into "categories" or types. In our example, a "person" falls into one of the two categories or subtypes "nurse" and "patient." Similarly, "health factors" has as categories "blood pressure," "skin resistance," "pulse," and "temperature."

The aggregation hierarchy organizes data objects into "components" so that several lower level concepts can be aggregated to form a higher level concept. For example, an aggregation hierarchy is given in Fig. 2. A "patient" data object consists of three components, a "name," an "identification number," and a "health factor range." A "blood pressure range" is one of the components of a "health factor range" (the other components are not shown), and it, in turn, has components "blood pressure" and "safe range."

The organization of data objects or entities allows the designer to reflect this structure in the system's state information. For example, the state component for each patient will contain information about name, ID number, and health factor measurement history and range. The structure of states can be represented as a relation or table as in Table I.

TABLE I
EXAMPLE OF A RELATION (THE "PATIENT-FACTOR-RANGE" RELATION) IN TABULAR FORM

Patient ID	Patient-name	BP range	Temp range	SR range	Pulse range
81039	Jones, R.	⟨-,150⟩	⟨-,105⟩	60	⟨50,120⟩
73084	Smith, J.	⟨-,160⟩	⟨-,104⟩	50	⟨50,140⟩
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
97405	Lynch, N.	⟨-,145⟩	⟨-,104⟩	70	⟨45,140⟩

The state information can also be obtained in a systematic fashion by observing what particular event occurs, what message it sends to the system and what action is triggered, and finally how the information content of the message is deposited in the system's states. Such a sequence of interactions is illustrated in Fig. 3 where the event "measure factors" sends message "factor-information" with content (Patient ID, blood pressure, temperature, . . .) (the brackets denote a sequence or tuple) to the system and triggers the action "insert factors" which deposits the message content into the state "patient-factor-history." If additional usage information such as a query model were collected, then the structure of the "patient-factor-history" state, which is represented by a relation, could be "normalized" [12]. For example, a query "From patient ID, get patient's factor range" would provide a "dependency" relation:

patient ID → (Blood pressure (1), . . . , Blood pressure (*n*); temp (1), . . . , temp (*n*); . . .)

where *n* indicates the history record, i.e., *n* measurements have been made.

Such sequences of interactions as illustrated above systematically establish major functions of the system (the actions) as well as the data flow (messages). Similarly, an analyst can establish the control flow, i.e., the sequencing of functions. This is done by first studying the sequencing of events in the environment. For example, the event "patient monitoring" can be split into two events: "initializing devices" and "initializing ranges." The states, actions, events and the flow systems then become the model. Analysis of the system can be done using this model through analytic methods, simulation, or interpretation depending on how the model is represented. However, we will not be concerned with these issues here.

IV. TUNING THE CONCEPTUAL MODEL

The model helps the analyst to understand the problem. However, it may contain redundancies resulting from distribution of some information into different states, or because information in certain states is not used at all. These redundancies should not be included in the SRD. Similarly, the description of actions (or tasks) may contain insufficient or too much information. For example, "use the *XYZ* timesharing system" is a description with insufficient information since it lacks information on the *XYZ* system, whereas "use the *XYZ* algorithm" contains too much information since it describes an implementation. Thus the conceptual model needs to be "tuned" before the SRD is finalized.

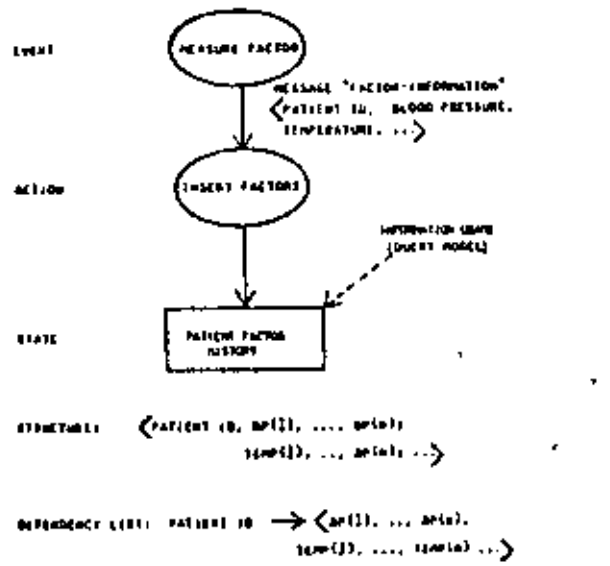


Fig. 3. An illustration of the modeling process.

The tuning process usually involves composition/decomposition of states as well as actions. In the previous example, two states "patient-factor-history" and "patient-factor-range" are combined into a new state called "patient-history" as shown in Fig. 4. On the other hand, we may have the situation illustrated in Fig. 5, that two processes (or actions) have access to a state but each only uses only part of the information. As shown in Fig. 5, Process A_1 uses S_1 whereas A_2 uses only S_2. In this case, S can be decomposed into two states S_1 and S_2 dedicated to A_1 and A_2, respectively, and recombined into a state $S' = S_1 \cup S_2$ so that the redundant information S-S' is eliminated. Similarly, the description of actions may be "tuned" so that it is the most *logical* description. We should note that in the tuning process, certain high-level tradeoffs between the amount of storage to be used and the amount of processing to be done are being made constantly, even though we would like to postpone such decisions as long as possible. Once the conceptual model is tuned, the analyst can then begin to write down the SRD.

But what are the components of a conceptual model in a SRD and how should they be described? The orthogonal components of a SRD should include a subset of the following, depending on the system to be developed. (Note that although the term "specification" is often associated with design, we regard requirements *and* designs as things that can and should be precisely specified.)

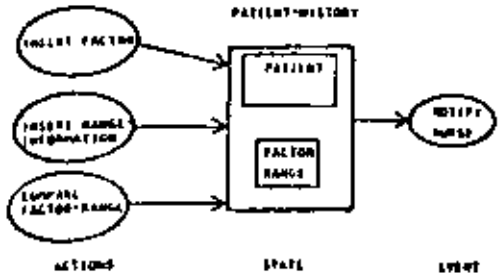


Fig. 4. Illustration of a combined state.

A. Data Specification

"Abstract data models" are used to express the relationships and constraints on entities (or data objects) of the "perceived" world. There are a large number of such models in existence, e.g., the relational model [10], the entity-relationship model [9], the role model [2], the object-role model [14], the simulation model [21], the semantic model [29]. These models, while different in philosophy, are used to model the primarily "static" parts of the perceived world as to entities, relationships among entities, attributes and values of entities, and constraints (such as data integrity). Transformation between different models has been studied by several authors [9], [18], [29]. The use of a particular model will depend on applications and the experience of the analysts.

We will discuss briefly the relational model here. It consists of a set of mathematical relations. Each relation consists of a set of n-tuples. In our example, we list a few entities as relations in the following:

Patient (patient ID, name, address, insurance plan, doctor assigned, nurse assigned)
Nurse (name, station, patients assigned, specialty)
Health Care Assignment (patient, nurse)
Health Factor (temperature, blood pressure, skin resistance, pulse)

The underlined domains are called *keys*. Keys in a relation can uniquely identify all the nonkey domains. In modeling, we may interpret relations with single keys to represent simple concepts or atomic facts whereas multikey relations are regarded as representing aggregations of concepts. Changes in the environment can be incorporated through algebraic operations on relations such as join, union, intersection, projection etc [10].

A specification of data via an abstract data model allows the designer/implementor to design a data base or file structure for the system.

B. Process Specification

Both events and actions in Fig. 1 are processes. While there are a number of ways to abstract functional specifications at the program level [20], forms of structured English such as PDL [19] or RSL [1] seem to be most widely used at the requirements level in general, whereas a decision table is a very useful specification tool in business applications with limited numbers of choices.

Basically, a specification language such as PDL is a restricted subset of English. Its syntax consists of an "outer" syntax and an "inner" syntax. The outer syntax is very similar to the syntax of the control structures of a high-level programming language. The inner syntax consists of selected special English words depending on the application.

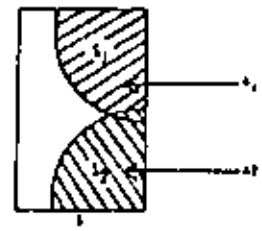


Fig. 5. Two processes, each one accessing only partial information in state S.

For example, consider the following specifications of an Air-Line Reservation System using RSL [35]:

R-net: AIR-LINE-RESERVATION-SYSTEM
Description: "Top Level Requirements for an Airline Reservation System"

```
STRUCTURE:
   ALPHA      VALID-REQ
   ALPHA      DET-REQ-TYPE
   DO         ALPHA RESV
          OR  ALPHA CANC
          OR  ALPHA CONF
          OR  ALPHA GET-LOAD-FACTOR
          OR  ALPHA CREATE-FLT
          OR  ALPHA DELETE-FLT
   END
   END
```

where each ALPHA is a functional processing step such as RESV (Reservation), CANC (Cancellation), CONF (Confirmation), etc., and can be further specified in more detail.

C. Specifications of Constraints

The two most important constraints are on data and control flows.

1) *Data Flow:* This specifies the relationship between data objects and processes. Such a relationship is usually expressed in terms of a data-flow diagram. For example, Fig. 6 is a top-level flow diagram of a business enterprise. We note that as the model develops, certain actions (or tasks) will be refined to have clearer logical descriptions. For example, the retailing function 2 can be further decomposed into more primitive functions: Enter new customer, receive payment, issue invoice, etc. at a lower level. The level that should be included in SRD is the one in which the logical behavior of each process can be described most clearly.

2) *Control Flow:* This specifies the "sequencing" relationships between processes. There are many mechanisms that can be used for precise specification of sequencing. We choose a modified Petri net notation [25] for our example as shown in Fig. 7. The "star" in Fig. 7 indicates that the particular operation can be performed as often as desired.

3) *Process Synchronization:* For systems such as real-time systems or operating systems, it is often necessary to specify the synchronization of processes competing for the same resource (data). While there are many mechanisms for concurrency control [13], [16], [27], most are at the implementation level instead of at the requirements level. We describe here a technique due to Conner [11] which can be used at the requirements level. Conner's method achieves synchronization by assigning controls both to the processes (called *rights controllers*) and to the resources (called *synchronizing controllers*)

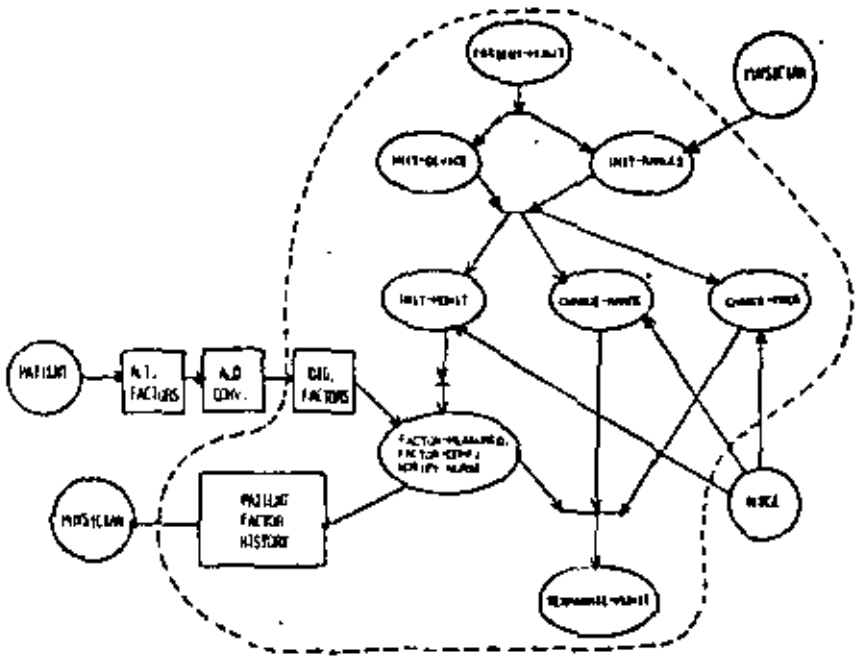


Fig. 6. Top-level data flow diagram of a business enterprise.

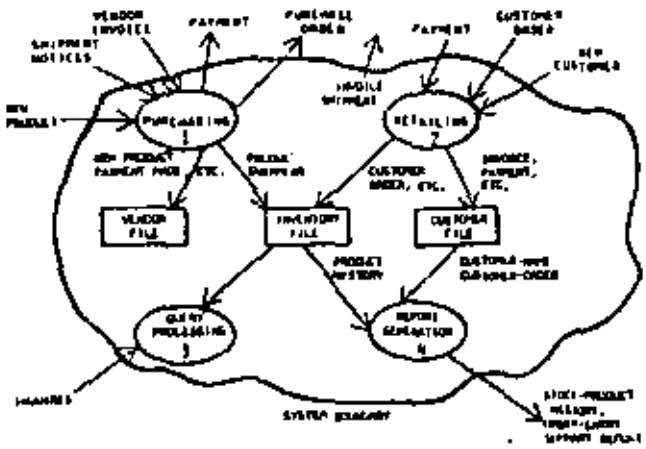


Fig. 7. The event net of patient-monitoring system example.

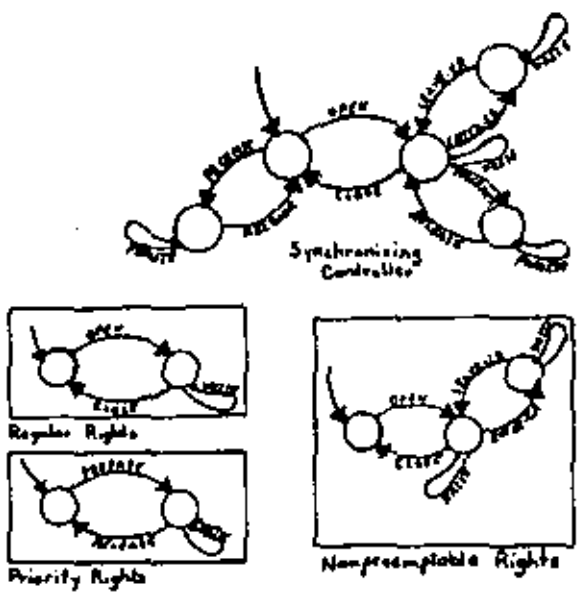


Fig. 8. Synchronizing and rights controllers of example.

Both kinds of controller are finite state graphs. The rights controller is a mechanism to enforce a constraint on the sequence of operations performed by a process through an access path to a data object. The synchronization controller is a mechanism to schedule the execution on shared data objects to achieve a specified sequencing constraint which is independent of the access path (and therefore processes) through which operations are performed. The advantage of Conner's approach is that specifications of sequencing constraints can be done on processes and data objects independently. Let us consider an example [11].

D. Example

A system consisting of several processes and a shared output device with the following properties:

1) proper use—before sending data to the output device, it must be readied;
2) there are processes with priority rights, i.e., they can preempt processes with regular rights;
3) there are processes with nonpreemptable rights.

The synchronizing controller and rights controllers of the processes and output device are given in Fig. 8 (the initial states are pointed to by arrows). It is easy to see from Fig. 8 that after a process has readied the output device, it can be preempted by a process with a priority right, but not vice versa. However, a process with a nonpreemptable right will take the synchronizing controller to a state (via an enter critical-region (CR) operation) which is not accessible by a process with priority or regular right and hence is not preemptable.

Note that Conner's method is easily extendable to specify process interaction in a distributed environment [22].

V. Nonfunctional Aspects of the SRD

A conceptual model can be used to specify all of the functional requirements for a system, but the SRD must not end there—additional nonfunctional constraints are a vital part

a complete and forward-looking requirements document. These additional constraints fall into three categories: constraints on the system to be developed, on the process of system development, and on the life of the system after it becomes operational.

The best-known constraints on the target system are performance requirements, notably constraints involving time and physical reliability [37]. Constraints on resources to be used by the target system include mandatory use of certain pieces of hardware or software, as well as constraints on the weight, volume, etc., of the finished package (for systems embedded in satellites or other tight places). A final category includes items that are, strictly speaking, functional requirements, but often so far from the main purpose of the system that decomposition of complexity is best served by keeping them out of the functional requirements. Examples are human factors and security requirements.

Requirements concerning the development of the system constrain the timetable, development resources (money, personnel, equipment, software development tools), and the methodology used. Methodological constraints can include programming or documentation standards, milestones, and acceptance criteria, to mention just a few.

So little attention has been devoted to anticipating the lifetime of a proposed system, and the maintenance and evolution it will undergo during that lifetime, that it is difficult to be specific about what kind of constraints concerning maintenance and evolution will be appropriate. Nevertheless, there can be little doubt that effort spent on anticipating changes during requirements analysis will be rewarded [23].

These concepts will be illustrated by an SRD for a simple process control system, taken from the description in [8]. The SRD follows a general outline we have developed for process control systems, although some sections are not applicable to this particular problem.

SRD for a Process Control System

0. *Introduction*

0.1. *Natural Language Description:*
A small process control system for an ammonia nitrate plant is needed. Its two purposes are to notify the operator of the plant if any dangerous conditions arise, and to monitor both the consumption of raw materials and the production of ammonia nitrate.

0.2. *Definitions:*
All real times are accurate to the nearest second.

1. *Objects in the Environment of the Computer System*

1.1. *Sensors:*
(a) Readings of more than 500 temperatures, pressures, and flow rates come through an analog/digital converter. In detail, ...
(b) There are about 150 digital inputs: some are single pulses from devices such as kilowatt-hour meters and bag fillers; others are the states of alarm contacts throughout the plant. Specifically, ...

1.2. *Actuators.*
This system has no automatic actuators, i.e. all of the actual process control is done manually, and the system's role in the feedback loop is to provide timely information to the operator.

1.3. *Input Devices:*
Operator console.

1.4. *Output Devices:*
(a) Light panel.
(b) Log printer.
(c) Alarm printer.

2. *Control Activities*

The following items must be specified for each condition or class of conditions to which a response is required. In this case there is only one such class.

(a) *Definition of Condition:*
Existence of an alarm condition is determined from the states of the alarm contacts, the analog values, and safe limits for the alarm values, as follows . . . .

(b) *Response to Condition:*
Notify operator through light panel and alarm printer, as follows . . . .

(c) *Real-Time Constraints:*
Alarms must appear within 5 min of the onset of any dangerous condition.

(d) *Scope, i.e., Possible Modifications:*
The real-time constraint is a default; the operator may stop the system's checking for alarm conditions, restart it for any given value of wall clock time, or change the response time value.

3. *Information-Gathering Activities*

The following lettered items must be specified for each class of output.

3.1. *Log Reports:*
(a) *Stimulus for output:*
Every hour.
(b) *Contents of output:*
A snapshot of how the plant operates, including . . . .
(c) *Scope:*
The hourly time is a default; the operator may stop this activity, restart it for any given value of wall clock time, or change its period.

3.2. *Flow Reports:*
(a) *Stimulus for output:*
Every 8 hours.
(b) *Contents of output:*
Consumption of electricity, production of ammonia nitrate, total flow of materials such as natural gas, steam, ammonia, and nitric acid.
(c) *Scope:*
See 3.1.(c).

4. *System Constraints*

4.1. *Reliability:*
The system uptime should be 95 percent over a 3-year period of 24-h operation.

4.2. *Security:*
No requirements.

4.3. *Human Factors:*
The operator command formats must be as simple as possible, although it is permissible to require the operator to press a special key before typing in a command. It is also permissible to have the operator set the system's real-time clock, but all other activities must be carried out autonomously by the system.

Programs, Life Cycles, and Laws of Software Evolution

MEIR M. LEHMAN, SENIOR MEMBER, IEEE

Abstract—By classifying programs according to their relationship to the environment in which they are executed, the paper identifies the sources of evolutionary pressure on computer applications and programs and shows why this results in a process of never ending maintenance activity. The resultant life cycle processes are then briefly discussed. The paper then introduces laws of Program Evolution that have been formulated following quantitative studies of the evolution of a number of different systems. Finally an example is provided of the application of Evolution Dynamics models to program release planning.

I. Background

A. The Nature of the Problem

THE TOTAL U.S expenditure on programming in 1977 is estimated to have exceeded $50 billion, and may have been as high as $100 billion. This figure, which represents more than 3 percent of the U.S. GNP for that year, is already an awesome figure. It has increased ever since in real terms and will continue to do so as the microprocessor finds ever wider application. Programming effectiveness is clearly a significant component of national economic health. Even small percentage improvements in productivity can make significant financial impact. The potential for saving is large.

Economic considerations are, however, not necessarily the main cause of widespread concern. As computers play an ever larger role in society and the life of the individual, it becomes more and more critical to be able to create and maintain effective, cost-effective, and timely software. For more than two decades, however, the programming fraternity, and through them the computer-user community, has faced serious problems in achieving this [1]. As the application of microprocessors extends ever deeper into the fabric of society the problems will be compounded unless very basic solutions are found and developed.

B. Programming

The early 1950's had been a pioneering period in programming. The sheer ecstasy of instructing a machine step by step to achieve automatic computation at speeds previously undreamed of, completely hid the intellectually unsatisfying aspects of programming; the lack of a *guiding theory* and *discipline*; the largely *hit or miss* nature of the process through which an acceptable program was finally achieved; the ever present uncertainty about the *accuracy*, even the *validity*, of the final result.

More immediately, the gradual penetration of the computer into the academic, industrial, and commercial worlds led to serious problems in the provision and upkeep of satisfactory programs. It also yielded new insights. Programming as then practiced required the breakdown of the problem to be solved into steps far more detailed than those in terms of which people thought about it and its solution. The manual generation of programs at this low level was tedious and error prone for those whose primary concern was the result; for whom programming was a means to an end and not an end in itself. This could not be the basis for widespread computer application.

Thus there was born the concept of *high-level*, problem-oriented, *languages* created to simplify the development of computer applications. These languages did not just raise the level of detail to which programmers had to develop their view of the automated problem-solving process. They also removed at least some of the burdens of procedural organization, resource allocation and scheduling, burdens which were further reduced through the development of operating systems and their associated job-control languages. Above all, however, the high-level language trend permitted a fundamental shift in attitude. To the discerning, at least, it became clear that it was not the programmer's main responsibility to instruct a machine by defining a step-by-step computational process. His task was to state an algorithm that correctly and unambiguously defines a mechanical procedure for obtaining a solution to a given problem [2], [3]. The transformation of this into executable and efficient code sequences could be more safely entrusted to automatic mechanisms. The objective of language design was to facilitate that task.

Languages had become a major tool in the hands of the programmer. Like all tools, they sought to reduce the manual effort of the worker and at the same time improve the quality of his work. They permitted and encouraged concentration on the intellectual tasks which are the real province of the human mind and skill. Thus, ever since, the search for better languages and for improving methodologies for their use, has continued [4].

There are those who believe that the development of *programming methodology*, high-level languages and associated concepts, is by far the most important step for successful computer usage. That may well be, but it is by no means sufficient. There exists a clear need for additional methodologies and tools, a need that arises primarily from program maintenance.

C. Program Maintenance

The sheer level of programming and programming-related activity makes its disciplining important. But a second statistic carries an equally significant message. Of the total U.S. expenditure for 1977, some 70 percent was spent on program *maintenance* and only about 30 percent on program *develop-*

Manuscript received February 27, 1980, revised May 23, 1980.
The author is with the Department of Computing, Imperial College of Science and Technology, 180 Queen's Gate, London SW7 2BZ, England

0018-9219/80/0900-1060$00.75

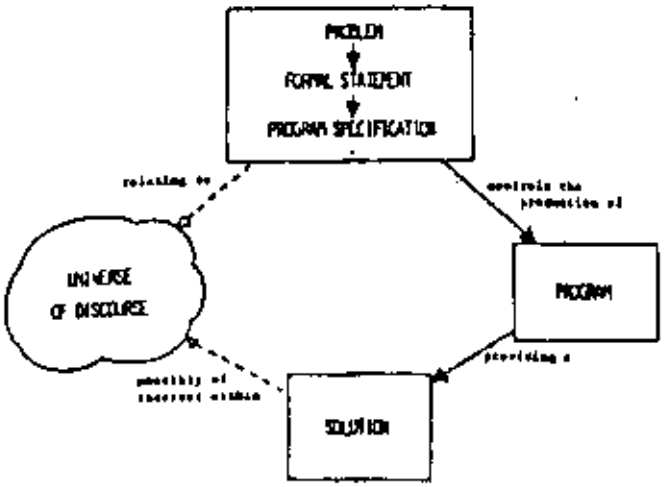


Fig. 1. *S*-programs.

ment. This ratio is generally accepted by the software community as characteristic of the state of the art.

Some clarification is, however, necessary. For software the term *maintenance* is generally used to describe *all* changes made to a program after its first installation. It therefore differs significantly from the more general concept that describes the *restoration* of a system or system component to its *former* state. Deterioration that has occurred as a result of usage or the passage of time, is corrected by repair or replacement. But software does not deteriorate spontaneously or by interaction with its operational environment. Programs do *not* suffer from wear, tear, corrosion, or pollution. They do not change unless and until *people* change them, and this is done whenever the current behavior of a program in execution is found to be wrong, inappropriate, or too restricted. *Repair* actually involves changes *away* from the previous implementation. Faults being corrected during maintenance can originate in any phase of the program *life cycle* (Section III).

Moreover, in hardware systems, major changes to a product are achieved by *redesign*, retooling, and the construction of a new model. With programs *improvements* and *adaptations* to a changing environment are achieved by alterations, deletions, and extensions to existing code. New capability, often not recognized during the earlier life of the system, is superimposed on an existing structure without redesign of the system as a whole.

Since the term software maintenance covers such a wide range of activities, the very high ratio of maintenance to development cost does not necessarily have to be deprecated. We shall, in fact, argue that the need for continuing change is *intrinsic* to the nature of computer usage. Thus the question raised by the high cost of maintenance is not exclusively how to control and reduce that cost by avoiding errors or by detecting them earlier in the development and usage cycle. The *unit cost of change* must initially be made as low as possible and its growth, as the system ages, minimized. Programs must be made more *alterable*, and the alterability *maintained* throughout their lifetime. The change process itself must be *planned* and *controlled*. Assessments of the economic viability of a program must include *total lifetime costs* and their life cycle *distribution*, and not be based exclusively on the initial development costs. We must be concerned with the cost and effectiveness of the life-cycle process itself and not just that of its product.

The opening paragraph highlighted the high cost of software and software maintenance. The economic benefit and potential of the application of computers is, however, so high that present expenditure levels may well be acceptable, at least for certain classes of programs. But we must be concerned with the fact that *performance*, *capability*, *quality in general*, cannot at present be designed and built into a program *ab initio*. Rather they are gradually achieved by evolutionary change and refinement. Moreover, when desirable changes are identified and authorized they can usually not be implemented on a time scale fixed by external need. *Responsiveness* is poor. And as mankind relies more and more on the software that controls the computers that in turn guide society, it becomes crucial that people control absolutely the programs and the processes by which they are produced, throughout the useful life of the program. To achieve this requires *insight*, *theory*, *models*, *methodologies*, *techniques*, *tools*: a *discipline*. That is what software engineering is all about [5]–[8].

II. Programs as Models

A. Programs

Program evolution dynamics [9 and its bibliography] and the laws [2], [3], [10], [11] discussed in the next section, have always been associated with a concept of *largeness*, implying a classification into large and nonlarge programs. Great difficulty has, however, been experienced in defining these classes. Recent discussions [12] have produced a more satisfying classification. This is based on a recognition of the fact that, at the very least, any program is *a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse*. The classification categorizes programs into three classes, *S*, *P*, and *E*. Since programs considered large by our previous definition will generally be of class *P* or *E*, the new classification represents a broadening and firming of the previous viewpoint.

B. S-Programs

S-programs are programs whose function is formally defined by and derivable from a *specification*. It is the programming form from which most advanced programming methodology and related techniques derive, and to which they directly relate. We shall suggest that as programming methodology evolves still further, all large programs (software systems) will be constructed as structures of *S*-programs.

A specific problem is stated: lowest common multiple of two integers; function evaluation in a specified domain; eight queens; dining philosophers; generation of a rectangle of a size within given limits on a specific type of visual display unit (VDU). Each such problem relates to its universe of discourse. It may also relate directly and primarily to the external world, but be completely defined, e.g., the *classical* travelling salesman problem.

As suggested by Fig. 1 the specification, as a formal definition of the problem, directs and controls the programmer in his creation of the program that defines the desired solution. Correct solution of the problem as *stated*, in terms of the programming language being used, becomes the programmer's sole concern. Almost questions of elegance or efficiency may also creep in.

The problem statement, the program and the solution when obtained may relate to an external world. But it is a casual, noncausal relationship. Even, when it exists we are free to

change our interest by redefining the problem. But then it has a *new program* for its solution. It may be possible and time-saving to derive the new program from the old. But it is a *different* program that defines a solution to a *different* problem.

When this view can be legitimately taken the resultant program is conceptually static. One may change it to improve its clarity or its elegance, to decrease resource usage when the program is executed, even to increase confidence in its correctness. But any such changes must not effect the mapping between input and output that the program defines and that it achieves in execution. Whenever program text has been changed or transformed [13], [14] it must be shown that either the input–output relationship remains unchanged, or that the new program satisfies a new specification defining a solution to a new problem. We return to the problem of correctness proving in Section II-E.

C. P-Programs

Consider a program to play chess. The program is completely specified by the rules of chess plus procedure rules. The latter must indicate how the program is to analyze the state of the game and determine possible moves. It must also provide a decision rule to select a next move. The procedure might, for example, be to form the tree of all games that may develop from any current state and adopt a minimax evaluation strategy to select the next move. Such a definition, while complete, is naive, since it is not implementable as an executing program. The tree structure at any given stage is simply too large, by many orders of magnitude, to be developed or to be scanned in feasible time. Thus the chess program must introduce approximation to achieve practicality, judged as it begins to be used, by its performance in actual games.

A further example of a problem that can be precisely formulated but whose solution must inevitably reflect an approximation of the real world is found in weather prediction. In theory, global weather can be modeled as accurately as desired by a set of hydrodynamic equations. In the actual world of weather prediction, approximate solutions of modified equations are compared with the weather patterns that occur. The results of such comparisons are interpreted and used to improve the technology of prediction, to yield ever more usable programs, whose outputs, however, always retain some degree of uncertainty.

Finally consider the travelling salesman problem as it arises in practice, for example from a desire to optimize continuously in some vaguely defined fashion, the travel schedule of salesmen picking up goods from warehouses and visiting clients. The required solution can be based on known approaches and solutions to the classical problem. But it must also involve considerations of cost, time, work schedules, timetables, value judgments, and even salesmens' idiosyncracies.

The problem statement can now, in general, no longer be precise. It is a model of an abstraction of a real-world situation, containing uncertainties, unknowns, arbitrary criteria, continuous variables. To some extent it must reflect the personal viewpoint of the analyst. Both the problem statement and its solution approximate the real-world situation.

Programs such as these are termed *P*-programs (real world *problem* solution). The process of creating such programs is modeled by Fig. 2 which shows the intrinsic feedback loop that is present in the *P*-situation. Despite the fact that the problem to be solved can be precisely defined, the acceptability of a solution is determined by the environment in which it is

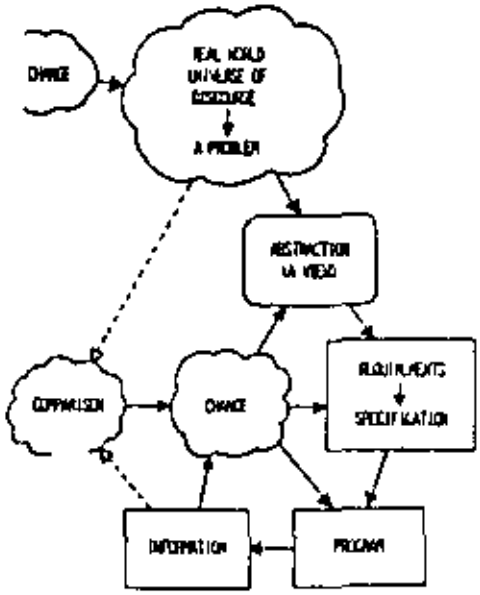


Fig. 2. *P*-programs.

embedded. The solution obtained will be evaluated by comparison with the real environment. That is, the critical difference between *S* and *P*-programs is expressed by the comparison cloud in Fig. 2. In *S*-programs, judgments about the correctness, and therefore the value, of the programs relate by definition *only to its specification*, the problem statement that the latter reflects. In *P*-programs, the concern is not centered on the problem statement but on the *value* and *validity* of the solution obtained *in its real-world context*. Differences between data derived from observation and from computation may cause changes in the world view, the problem perception, its formulation, the model, the program specification and/or the program implementation. Whatever the source of the difference, ultimately it causes the program, its documentation or both to be changed. And the effect or impact of such change cannot be eliminated by declaring the problem a *new* problem, for the real problem has always been as now perceived. It is the perception of users, analysts and/or programmers that has changed.

There is also another fact of life that needs to be considered. Dissatisfaction will arise not only because information received from the program is incomplete or incorrect, or because the original model was less than perfect. These are imperfections that can be overcome given time and care. But *the world too changes* and such changes result in additional pressure for change. Thus *P*-programs are very likely to undergo never-ending change or to become steadily less and less effective and cost effective.

D. E-Programs

The third class, *E*-programs, are inherently even more change prone. They are programs that mechanize a human or societal activity.

Consider again the travelling salesman problem but in a situation where several persons are continuously en route, carrying products that change rapidly in value as a function of both time and location, and with the pattern of demand also changing continuously. One will inevitably be tempted to see this situation as an application in which the system is to act as a continuous dispatcher, dynamically controlling the journeys and calls of each individual. The objective will be to maximize

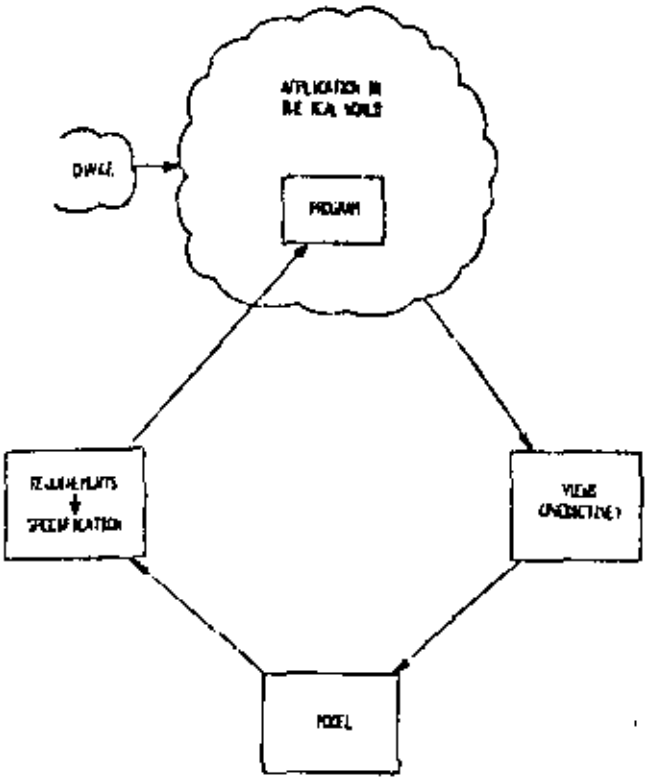


Fig. 3. *E*-programs-The basic cycle.

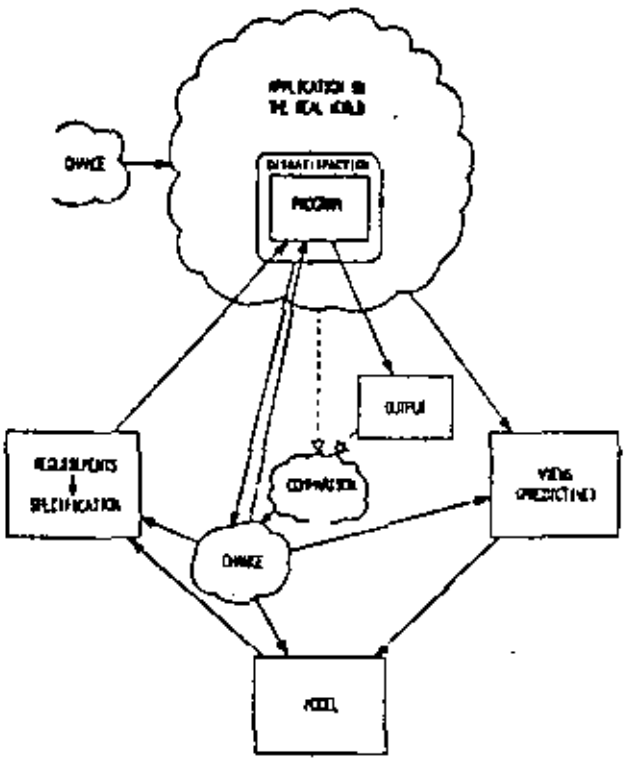


Fig. 4. *E*-programs.

profit, minimize loss, expedite deliveries, maintain customer satisfaction or achieve some optimum combination of the factors that are accepted as the criteria for success. How does this situation differ from that discussed in the previous sections?

The installation of the program together with its associated system—radio links to the salesmen, for example—change the very nature of the problem to be solved. *The program has become a part of the world it models*, it is *embedded* in it. Conceptually at least the program as a model contains elements that model itself, the consequences of its execution.

The situation is depicted in Figs. 3 and 4. Even without considering program execution and evaluation of its output in the operational environment, the *E*-situation contains an intrinsic feedback loop as in Fig. 3. Analysis of the application to determine requirements, specification, design, implementation now all involve extrapolation and prediction of the consequences of system introduction and the resultant potential for application and system evolution. This prediction must inevitably involve opinion and judgment. In general, several views of the situation will be combined to yield the *model*, the system specification and, ultimately, *a* program. Once the program is completed and begins to be used, questions of correctness, appropriateness and satisfaction arise as in Fig. 4 and inevitably lead to additional pressure for change.

Examples of *E*-programs abound: computer operating systems, air-traffic control, stock control. In all cases, the behavior of the application system, the demands on the user, and the support required will depend on program characteristics as experienced by the users. As they become familiar with a system whose design and attributes depend at least in part on users' attitudes and practice before system installation, users will modify their behavior to minimize effort or maximize effectiveness. Inevitably this leads to pressure for system change. In addition, system exogenous pressures will also cause changes in the application environment within which the system operates and the program executes. New hardware will be introduced, traffic patterns and demand change, technology advance and society itself evolve. Moreover the nature and rate of this evolution will be markedly influenced by program characteristics, with a new release at intervals ranging from one month to two years, say. Unlike other artificial systems [15] where, relative to the life cycle of process participants, change is occasional, here it appears continually. The pressure for change is built in. It is intrinsic to the nature of computing systems and the way they are developed and used. *P* and *E* programs are clearly closely related. They differ from *S*-programs in that they represent a computer *application* in the real world. We shall refer to members of the union of the *P* and *E* classes as *A*-type programs.

E. Program Correctness

The first consequence of the SPE program classification is a clarification of the concepts of program correctness and program proving. The meaning, reality, and significance of these concepts have recently been examined at great length [16], [17]. Many of the viewpoints and differences expressed by the participants in that discussion become reconcilable or irrelevant under an adequate program classification scheme.

For the SPE scheme, the concept of verification takes on significantly different meanings for the *S* and the *A* classes. If a completely specified problem is computable, its specification may be taken as the starting point for the creation of an *S*-program. In principle a logically connected sequence of statements can always be found, that demonstrates the validity of the program as a solution of the specified problem. Detailed inspection of and reasoning about the code may itself produce the conviction that the program satisfies the specification completely. A true proof must satisfy the accepted standards of mathematics. Even when the correctness argument is

expressed in mathematical terms, a lengthy or complex chain of reasoning may be difficult to understand, the proof sequence may even contain an error. But this does not invalidate the concept of program correctness proving, merely this instance of its application.

We cannot discuss here the range of *S*-programs for which proving is a practical or a valuable technique, the range of applicability of constructive methods for simultaneous construction of a program and its proof [18], [19]; whether confidence in the validity of an *S*-program can always be increased by a proof. We simply note that since, by definition, the sole criterion of correctness of an *S*-program is the satisfaction of its specification, (correct) *S*-programs are always *provably* correct.

This is not purely a philosophical observation. Many important components of a large program, mathematical procedures for example, in conjunction with specified interface rules (calling and output), are certainly *S*-type. It becomes part of the design process to recognize such potential constituents during the partitioning process and to specify and implement them accordingly. In fact it will be postulated in the next section that an *A*-program may always be partitioned and structured so that *all* its elements are *S*-programs. If this is indeed true, no *individual* programmer should *ever* be permitted to begin programming until his task has been defined and delimited by a complete specification against which his completed program can be validated.

For an *E*-program as an entity on the other hand, validity depends on human assessment of its effectiveness in the intended application. Correctness and proof of correctness of the program as a whole are, in general, irrelevant in that a program may be formally correct but useless, or incorrect in that it does not satisfy some stated specification, yet quite usable, even satisfactory. Formal techniques of representation and proof have a place in the universe of *A*-programs but their role changes. It is the detailed *behavior* of the program *under operational conditions* that is of concern.

Parts of the program that can be completely specified should be demonstrably correct. But the environment cannot be completely described without abstraction and, therefore, approximation. Hence absolute correctness of the program as a whole is not the real issue. It is the *usability* of the program and the *relevance* of its output in a changing world that must be the main concern.

F. Program Structures and Structural Elements

The classification created above relates to program entities. Any such program will, in general, consist of many parts variously referred to as subsystems, components, modules, procedures, routines. The terms are, of course, not used synonymously but carry imputations of functional identity, level, size, and so on.

The literature discusses criteria [20] and techniques [21]–[23] for partitioning systems into such elements. Related design methodologies and techniques seek to achieve optimum assignment, in some sense, of element content and overall system structure. In the present context we consider only one aspect of partitioning using the term module for convenience. The discussion completes the presentation of the SPE classification and provides a link to other current methodological thinking [24].

Consider the end result of the design process for an *A*-program to be constructed of primitive elements we term modules. The analysis and partitioning process will identify some functional elements that can be fully specified and therefore developed as *S*-program modules. Any specification may of course be less than fully satisfactory. It may even prove to be wrong in relation to what the system purpose demands, in itself or in relation to the remainder of the design. For example the specification may not mention input validity checks, the specified output accuracy may be insufficient or the specified range of an input variable may be wrong. But each of these represents an omission from or an error in the *specification*. Thus it is rectified by first correcting the *specification* and then creating, by one means or another, a new program that satisfies the new specification.

The remainder of the system is required to implement functions that are at least partly heuristic or behavioral in nature and therefore define *A*-elements. Nevertheless, we suggest that it is *always* possible to continue the system partitioning process until *all* modules are implementable as *S*-programs. That is, any imprecision or uncertainty emanating from model reflections of incomplete world views will be implicit or, if recognized when the specification is formulated, explicit in the specification statement. The final modules will all be derived from and associated with precise specifications, which *for the moment*, may be treated as complete and correct.

The design may now be viewed and constructed as a data-flow structure with the inputs of one module being the outputs of others (unless emanating from outside the system). Each module will be defined as an abstract data type [25]–[27] defining, in turn, one or more input-to-output transformations. Module specifications include those of the individual interfaces, but for the system as a whole, the latter should, in some sense be standardized [28]. Moreover, given appropriate system and interface architecture and module design, each module could be implemented as a program running on its own microprocessor and the system implemented as a distributed system [9], [24], [28], [92]. The potential advantages for both execution (parallelism) and maintainability (localization of change) cannot be discussed here.

Many problems in connection with the design and construction of such systems need still to be solved. Adequate solutions will represent a major advance in the development of a process methodology (Section III-C). We observe, however, that the concepts presented follow directly from our brief analysis and classification of program types. Interestingly, the conclusions are completely compatible with those of the programming methodologists [24], [29], [30].

III. The Life Cycle

A. The General Case

The dynamic evolutionary nature of computer applications, of the software that implements them and of the process that produces both, has in recent years given rise to a concept of a program *life cycle* and to techniques for *life-cycle management*. The need for such management has, in fact, been recognized in far wider spheres, particularly by national defense agencies and other organizations concerned with the management of complex artificial systems. In pursuing their responsibilities, these must ensure continuing effectiveness of systems whose elements may involve many different and fast developing technologies. Often they must guarantee utterly reliable operation under harsh, hostile, and unforgiving conditions. The outcome is an ever increasing financial commitment. Only lifetime-orientated management techniques applied from project

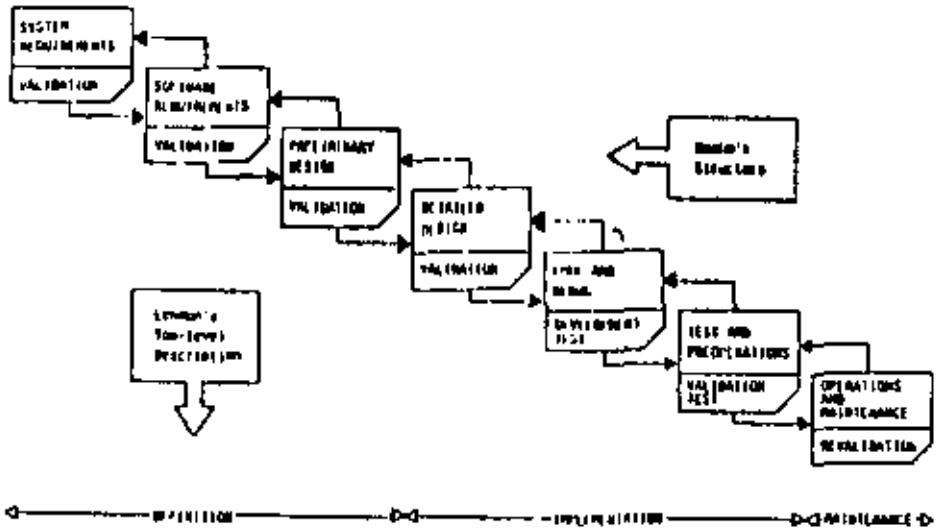


Fig. 5. The software life cycle according to Boehm.

initiation can permit the attainment of lifetime effectiveness and cost effectiveness.

The problems in the more general situation are essentially those we have already explored, except that the time interval between generations is perhaps an order of magnitude greater than in the case of pure software systems. In briefly examining the nature of the life cycle and its management in this section, we use the terminology of programming and software engineering. The reader will be able to generalize and to interpret the remarks in his own area of interest.

B. Software Life Cycles

In studying program evolution, repetitive phenomena that define a life cycle can be observed on different time scales representing various levels of abstraction. The highest level concerns *successive generations* of system sequences. Each generation is represented by a sequence of system releases. This level corresponds most closely to that found in the more general systems situation, with each generation having a life span of from, say, five to twenty years. Because of the relatively slow rate of change it is difficult for any individual to observe this evolution phenomenon, measure its dynamics and model it as a life-cycle process since in the relevant portion of his professional career he will not observe more than two or three generations. It might therefore be argued that this level should not be treated as an instance of the life-cycle phenomenon. The present author has, however, had at least one opportunity to examine program evolution at this level and to make meaningful and significant observations [31]. These indicated that much could be gained in cost effectiveness in the software industry if more attention were paid to the easier creation of replacement generations, something that can be achieved effectively only if the appropriate predictive models are available.

The second level is concerned with a *sequence* of *releases*. The latter term is also appropriate when a concept of continuous release is followed, that is when each change is made, validated, and immediately installed in user instances of the system.

Fig. 5 shows one view [6] of the *sequence* of *activities* or life-cycle phases that constitute the lowest level, the development of an individual release, if it is assumed that "maintenance" in the seventh box refers to on-site *fixes* and repairs implemented as the system is used. If maintenance is taken to refer to permanent changes, effected through new releases by the system originator, then the structure becomes *recursive* with each maintenance phase comprised of all seven indicated phases. With this interpretation the single recursive model reflects the composite life-cycle structure of all the above levels.

The remainder of this paper is chiefly concerned with the intermediate level, the life cycle of a generation as represented by a sequence of releases. It is at this level that analysis in terms of the *S* and *A* classification is particularly relevant and enlightening.

C. Assembly Line Processes

An assembly line manufacturing process is possible when a system can be partitioned into subsystems that are simply coupled and without invisible links. Moreover, the process must be divisible into separate phases without significant feedback control over phases and with relatively little opportunity for tradeoff between them.

Unfortunately, present day programming is not like that. It is constituted of tightly coupled activities that interact in many ways. For example, at least some aspects of the specification and design processes are left over, usually implicitly, to the implementation (coding) phase. Fault detection through inspection [90] is not yet universal practice and by default is often delayed till a system integration or system testing phase. One of the main concerns of life-cycle process methodology research must be to develop techniques, tools, new system architectures (Section II-F) and programming support environments [32]–[34] that permit partitioning of the program development and maintenance process into separated activities.

D. The Significance of the Life-Cycle Concept

For assembly line processes the life-cycle concept is not, generally, of prime importance. For software and other highly complex systems it becomes critical if effectiveness, cost effectiveness, and long life are to be achieved. At each moment in time, a manager's concern concentrates on the successful completion of his current assignment. His success will be assessed by immediately observable product attributes, quality, cost, timeliness, and so on. It is his success in areas such as these that determine the furtherance of his career. Managerial strategy will inevitably be dominated by a desire to achieve maxi-

mum local payoff with visible short-term benefit. It will not often take into account long-term penalties, that cannot be precisely predicted and whose cost cannot be assessed. Top-level managerial pressure to apply life-cycle evaluation is therefore essential if a development and maintenance process is to be attained that continuously achieves, say, desired overall balance between the short- and long-term objectives of the organization. Neglect will inevitably result in a lifetime expenditure on the system that exceeds many times the assessed development cost on the basis of which the system or project was initially authorized.

To overcome long time lags and the high cost of software, one may also seek to extend the useful lifetime of a system. The decision to replace a system is taken when maintenance has become too expensive, reliability too low, change responsiveness too sluggish, performance unacceptable, functionality too limiting; in short, when it is economically more satisfactory to replace the system than to maintain it. But its expected life time to that point is determined primarily in its conception, design and initial implementation stages. Hence management planning and control during the formative period of system life, based on lifetime projections and assessment, can be critical in achieving long life software and lifetime cost effectiveness [1].

E. Life-Cycle Phases

1) The Major Activity Classes: At its grossest level a life cycle consists of three phases: definition, implementation and maintenance. As indicated in Fig. 5, these three phases correspond approximately to the activities described in the first three, the second three and the seventh box respectively of Boehm's model. In practice, however, many of these activities are overlapped, interwoven, and repeated iteratively.

2) System Definition: For *E*-class systems in particular, the development process begins with a pragmatic analysis leading into a systematic *systems analysis* to determine total system and program *requirements* [35]–[38]. The analysis must first establish the *real* need and objectives and may examine the manual techniques whereby the same purpose is currently achieved. Where appropriate, it may be based on mathematical or other formal analysis. Whatever the approach, it has now been recognized that the analysis must be *disciplined* and *structured* [29], [30], the term *structured analysis* now being widely used [9], [41], [42].

By their very nature initial requirements, being an expression of the user's view of his needs, are likely to include incompatibilities or even contradictions. Thus the analysis and the negotiation process by and between analysts and potential users that produces the final *requirements specification*, must identify a balanced set that, in some sense, provides the optimum compromise between conflicting desires.

The requirements set will be expressed in the concepts and language of the application and its users. It must then be transformed into a *technical specification*. The specification process [43], [44] must aim to produce a correct technical statement, *complete* in its coverage of the requirements and *consistent* in its definition of the implementation. It may include additional determinations or constraints that follow from a technical evaluation of the requirements in relation to what is feasible, available and appropriate in the judgment of the analyst and designer in agreement with the user.

It has long been the aim of computer scientists to provide formal languages for the expression of specifications so as to permit mechanical checking of completeness and consistency [45]–[49], [91], but a widely accepted language does not yet exist. Given a machinable specification it is conceptually possible to reduce it mechanically to executable [50] and even efficient [14] code but these technologies too are not yet ready for general exploitation.

Thus, for the time being, the specification process will be followed by a *design* phase [49], [51]. The prime objective of this activity is to identify and structure data, data transformation and data flow [23]. It must also achieve, in some defined sense, optimal partitioning of system function [20], select computational algorithms and procedures, and identify system components, and the relationships between them. It is now generally accepted that iterative *top-down* [52] analysis and partitioning processes are required to achieve *successive refinement* [21] of the system design to the point where the identified objects, procedures, and transformations can be directly implemented.

3) Implementation: Following the completion of the design, system *implementation* may begin. In practice, however, design and implementation overlap. Thus, as the hierarchical partitioning process proceeds, analysis of certain aspects of the system may be considered sufficient for implementation, while others require further analysis. In a software project, time always appears to be at a premium. A work force comprising many different abilities is available and must be kept busy. Thus, regrettably, implementation of subsystems, components, procedures, or modules will be initiated despite the fact that the overall, or even the local design, is not yet complete.

As the implementation proceeds code must be *validated* [53], [54]. Present day procedures concentrate primarily on *testing* [55], though in recent years increasing use has been made of design *walkthrough* and code *inspection* [90]. These latter procedures are intended to disclose both design and implementation errors before their consequences become hidden in the program code. The ratio of costs of removing a fault discovered in usage as against the cost of removing the same fault if discovered during the design or first implementation phase is sometimes two or three orders of magnitude. Clearly, it pays to find faults early in the process.

In any case, testing by means of program execution is carried out, generally bottom up, first at the unit (module or procedural) level, then functionally, component by component. As tested components become available they are then assembled into a system in an *integration* process and *system test* is initiated. Finally, after some degree of independent certification of system function and performance, the system is designated ready for *release*.

The above very brief summary has identified some of the activities that are typically undertaken in a system creation process. Individual activities as described may overlap, be iterated, merged, or not undertaken at all. Design of an element, for example, may be followed immediately by a test implementation and preliminary performance evaluation to ensure feasibility of a design before its implications spread to other parts of the system. Clearly, there should be a set of overall controlled procedures to take a concept from the first pragmatic evaluation of the potential of an application for mechanization to the final program product executing in defined hardware or software and hardware environment(s).

4) Maintenance: Once the system has been released, the maintenance process begins. Faults will be observed, reported, and corrected. If user progress is blocked because of a fault, a temporary bypass of the faulty code may be authorized. In other circumstances a temporary or permanent fix may be

applied in some or all user locations. The permanent repair or change to the program can then be held over for a new release of the system. In other cases, a permanent change will be prepared for immediate installation by all those running the system. The particular strategy adopted in any instance will depend on the nature and severity of the fault, the size and difficulty of the change required, the number and nature of program installations and user organizations, and so on. The aggregate strategy will have a profound impact on the rate of system complexity growth, on its life-cycle costs, and on its life expectancy.

The faults that are fixed in the maintenance process may be due to changes external to the system, incorrect or incomplete specification, design or implementation errors, hardware changes or to some combination of these. Since each user exposes the system in different ways, all installations do not experience all faults, nor do they automatically apply all manufacturer-supplied fixes or changes. On the other hand, installations having their own programming staff may very well develop and install localized changes or system modifications to suit their specific needs. These patches, insertions, or deletions may in turn cause new difficulties when further incremental changes are received from the manufacturer, or at a later date when a new release is received. The inevitable consequences of the maintenance process applied to systems installed for more than one user, is that the system drifts apart. Multiple versions of system elements develop to encompass the variations and combinations [56]. System *configuration* management becomes a major task. *Support environments* [33]–[35] that automatically collect and maintain total activity records become an essential tool in programming process management.

F. Life-Cycle Planning and Management

The preceding discussion, while presenting a simplified view of the life cycle, will have made clear the difficulty associated with cycle planning. In recent years this problem has received much attention [57], [58]. A variety of techniques have been developed to improve estimation of cost, time, and other resources required for software development and maintenance [59]–[64]. These techniques are based on extrapolation of past experience and tend to produce results in the nature of self-fulfilling prophecies. In general, it has not yet proved possible to develop techniques that estimate project requirements on the basis of objective measurement of such attributes as application complexity and size and the work required to create a satisfactory system. Techniques such as software science [65], [66] seek to do just this but to date lack substantiation [67] and interpretation. Major research and advances are required if software engineering is to become as manageable as are other engineering disciplines, though fundamentally the peculiar nature of software systems [28] will always leave software engineering in a class of its own.

IV. Laws of Program Evolution

A. Evolution

The analysis of Section II associated with the life-cycle description of Section III, has indicated that evolution is an intrinsic, feedback driven, property of software. The metasystem within which a program evolves contains many more feedback relationships than those identified above. Primitive instincts of survival and growth result in the evolution of stabilizing mechanisms in response to needs, events and changing objectives. The resulting pseudohierarchical structure of self-stabilizing systems includes the products, the processes, the environments and the organizations involved. The interactions between and within the various constituents, and the overall pattern of behavior must be understood if a program product and its usage are to be effectively planned and maintained.

The organizational and environmental feedback, links, focuses, and transmits the evolutionary pressure to yield the continuing change process. A similar situation holds, of course, for any human organized activity, any artificial system. But some significant differences are operative in the case of software. In the first instance there is no room in programming for imprecision, no malleability to accommodate uncertainty or error. Programming is a mathematical discipline. In relation to a *specific* objective, a program is either right or wrong. Once an instruction sequence has been fixed and unless and until it is manually changed, its behavior in execution on a given machine is determined solely by its inputs.

Secondly, a software system is soft. Changes can be implemented using a pencil, paper, and/or a keyboard. Moreover, once a change has been designed and implemented on a development system it can be applied mechanically to any number of instances of the same system without further significant physical or intellectual effort using only computing resources. Thus the temptation is to implement changes in the existing system, change upon change upon change, rather than to collect changes into groups and implement them in a totally new instance. As the number of superimposed changes increases, the system and the metasystem become more complex, stiffer, more resistant to change. The cost, the time required, and the probability of an erroneous or unsatisfactory change all increase.

Thirdly, the rate at which a program executes, the frequency of usage, usage interaction with the operating environment, economic and social dependence of external process on program execution, all cause deficiencies to be exposed. The resultant pressure for correction and improvement leads to a system rate of change with a time scale measured in days and months rather than in the years and decades that separate hardware generations.

B. Dynamics and Laws of Program Evolution

The resultant evolution of software appears to be driven and controlled by human decision, managerial edict, and programmer judgment. Yet as shown by extended studies [68]–[76], measures of its evolution display patterns, regularity and trends that suggest an underlying dynamics that may be modeled and used for planning, for process control, and for process improvement.

Once observed the reasons for this unexpected regularity is easily understood. Individual decisions in the life cycle of a software system generally *appear* localized in the system and in time. The considerations on which they are based *appear* independent. Managerial decisions are largely taken in relative isolation, concerned to achieve local control and optimization, concentrated on some aspect of the process, some phase of system evolution. But their aggregation, moderated by the many feedback relationships, produces overall systems response which is regular and often normally distributed.

In its early stages of development a system is more or less under the control of those involved in its analysis, design, and implementation. As it ages, those working on or with the system become increasingly constrained by earlier decisions, by existing code, by established practices and habits of users and

TABLE I
LAWS OF PROGRAM EVOLUTION

I. *Continuing Change*

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version.

II. *Increasing Complexity*

As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.

III. *The Fundamental Law of Program Evolution*

Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances.

IV. *Conservation of Organizational Stability (Invariant Work Rate)*

During the active life of a program the global activity rate in a programming project is statistically invariant.

V. *Conservation of Familiarity (Perceived Complexity)*

During the active life of a program the release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

implementors alike. Local control remains with people. But process and system-internal links, dependencies, and interactions cause the global characteristics of system evolution to be determined by organization, process and system parameters. At the global level the metasystem dynamics have largely taken over.

Since the original observation [63], studies of program evolution have continued, based on measurements obtained from a variety of systems. Typical examples of the resultant models have been reported [69]-[72], [74], [76] including also one detailed example of their application to release planning [77].

It was repeated observation of phenomenologically similar behavior and the common interpretation of independent phenomena, that led to a set of five laws, that have themselves evolved as insight and understanding have increased. The laws, as currently formulated to include the new viewpoint emerging from the SPE classification, are given in Table I. Their early development can be followed in [9], [10], [72]. We note that the laws are abstractions of observed behavior based on statistical models. They have no meaning until a system, a project and the organizational metasystem are well established. More detailed discussion of their nature and of their technical and managerial implications will be found in [11], [77], [78] and [77], [79], [80], respectively.

The first law, *continuing change*, originally [3], [10], [79] expressed the universally observed fact that large programs are never completed. They just continue to evolve. Following our new insight, however, reference to *largeness* is now replaced by the phrase . . . "that reflect some other reality . . ."

The second law, *increasing complexity*, could be seen as an instance of the second law of thermodynamics. It would seem more reasonable to regard both as instances of some more fundamental natural truth. But from either viewpoint its message is clear.

The third law, the *fundamental law of program evolution*, is in the nature of an existence rule. It abstracts the observed fact that the number of decisions driving the process of evolution, the many feedback paths the checks and balances of organizations, human interactions in the process, reactions to usage, the rigidity of program code, all combine to yield statistically regular behavior such as that observed and measured in the systems studied.

The fourth law, *conservation of organizational stability*, and the fifth, *conservation of familiarity*, represent instances of the observations whose generalization led to the third law. The fourth reflects the steadiness of multiloop self-stabilizing systems. It is believed to arise from organizational striving for stability. The managements of well-established organizations avoid dramatic change and particularly discontinuities in growth rates. Moreover, the number of people and the investments involved, the unions, the time delays in implementing decisions, all operate together to prevent sudden or drastic change. Wide fluctuations may in fact lead to instability and the breakup of an organization.

The reader may find it difficult to accept the implication that the work output of a project is independent of the amount of resources employed, though the same observation has also been recorded by others [81]. The underlying truth is that activities of the type considered, though initiated with minimal resources, rapidly attract more and more as commitment to the project, and therefore the consequences of success or failure, increase. Our observations as formalized in the fourth law imply that the resources that can be productively applied becomes limited as a software project ages. The magnitude of the limit depends on many factors including attributes of the total environment. But the pressure for success leads to investment to the point where it is exceeded. The project reaches the stage of resource saturation and further changes have no visible effect on real overall output.

While the fourth law springs from a pattern of organizational behavior, the fifth reflects the collective consequences of the characteristics of the many individuals within the organization. It is discussed at length in [11]. Suffice it to say here that the law arises from the nonlinear relationship between the magnitude of a system change and the intellectual effort and time required to absorb that change.

TABLE II
SYSTEM X STATISTICS

Release 19 Statistics	
Size	4800 Modules 1.3 Assembly *M*-statements
Incremental growth	410 Modules
Modules changed[a]	2650 Modules
Fraction of modules changed	0.55
Release Interval	275 Days
System Statistics	
Age	4.3 Years
Change rate	10.7 Modules/day
Average incremental growth	200 Modules/release
Maximum safe growth rate	400 Modules/release

Most Recent Releases

Release	15	16	17	18	19
Incremental growth (Δ Mod)	135	171	183	354	410
Fraction changed	0.33	0.43	0.48	0.50	0.56
Change rate	12.5	0.12	9.6	9.9	9.6
Interval (Days)	96	137	201	221	275
Old mods. Changed/ Mod	7.9	8.6	10.0	5.1	5.4

[a]Modules that are changed in any way in release $l+1$ relative to release l are counted as one changed module, independently of the number of changes or of their magnitude.

V. APPLIED DYNAMICS

A. Introduction

The previous sections have emphasized the phenomenological basis for the laws of program evolution, indicating how they are rooted in phenomena underlying the activity of programming itself.

The origin of the laws in individual and societal behavior makes their impact on the construction and maintenance of software more than just descriptions of the evolutionary process. The laws represent *principles* in software engineering. They are, however, clearly not immutable, as for example, are the laws of physics or chemistry. Since they arise from the habits and practices of people and organizations, their modification or change requires one to go outside the discipline of computer science into the realms of sociology, economics and management. The laws therefore form an environment within which the effectiveness of programming methodologies and management strategies and techniques can be evaluated, a backdrop against which better methodologies and techniques can be developed.

Their implications, technical and managerial, have been previously discussed in the literature [3], [9], [11], [79], [80]. In the present paper, we restrict the discussion to outlining an example of the application of evolution dynamics models to release planning.

B. A Case Study—System X

1) The System and its Characteristics: System X is a general purpose batch operating system running on a range of machines. The eighteenth release (R18) of the system is operational in some tens of installations running a variety of workloads. The nineteenth release (R19) is about to be shipped.

Table II and Fig. 6(a)–(g) present the system and release data and models available for the purposes of the present exercise. We cannot, however, provide here the details of statistical analysis and model validation [76], based on th and that from other systems that gives us confidence in our conclusions and predictions.

Examining the system dynamics as implied by models derived from the data and as illustrated by the figures, Fig. 6(a) shows the continuing growth of the system (first law) albeit at a declining rate (demonstrably due to increasing difficulty of change, growing complexity—second law).

Fig. 6(b) indicates that as a function of release sequence number (RSN) the system growth (measured in modules) has been linear but with a superimposed ripple (a strong indicator of feedback stabilization).

Fig. 6(c) shows the net incremental growth per release (fifth law).

For system architectures such as that of system X, the fraction of system modules that are changed during a release may be taken as a gross indicator of system complexity. Fig. 6(d) shows that system X complexity, as measured in this way, shows an increasing trend (second law).

Fig. 6(e) is an example of the repeatedly observed constant average work rate (fourth law).

Fig. 6(f) illustrates how the average work rate achieved in individual releases, as measured by the rate of module change (changed modules per release interval day (m/d) oscillates, a period of high rate activity being followed by one or more in which the activity rate is much lower (third law).

Finally, Fig. 6(g) plots the release interval against release sequence number. It has been argued that release interval depends purely on management decision that is itself based on market considerations and technical aspects of the release content and environment. Data such as that of Fig. 6(g) indicates, however, that the feedback mechanisms that, amongst other process attributes, also control the release interval, while including human decision taking processes, are apparently not dominated by them. As a consequence, the release interval pattern is sufficiently regular to be modelable, and is statistically predictable once enough data points have been established.

2) The Problem: Already prior to the completion (and re-

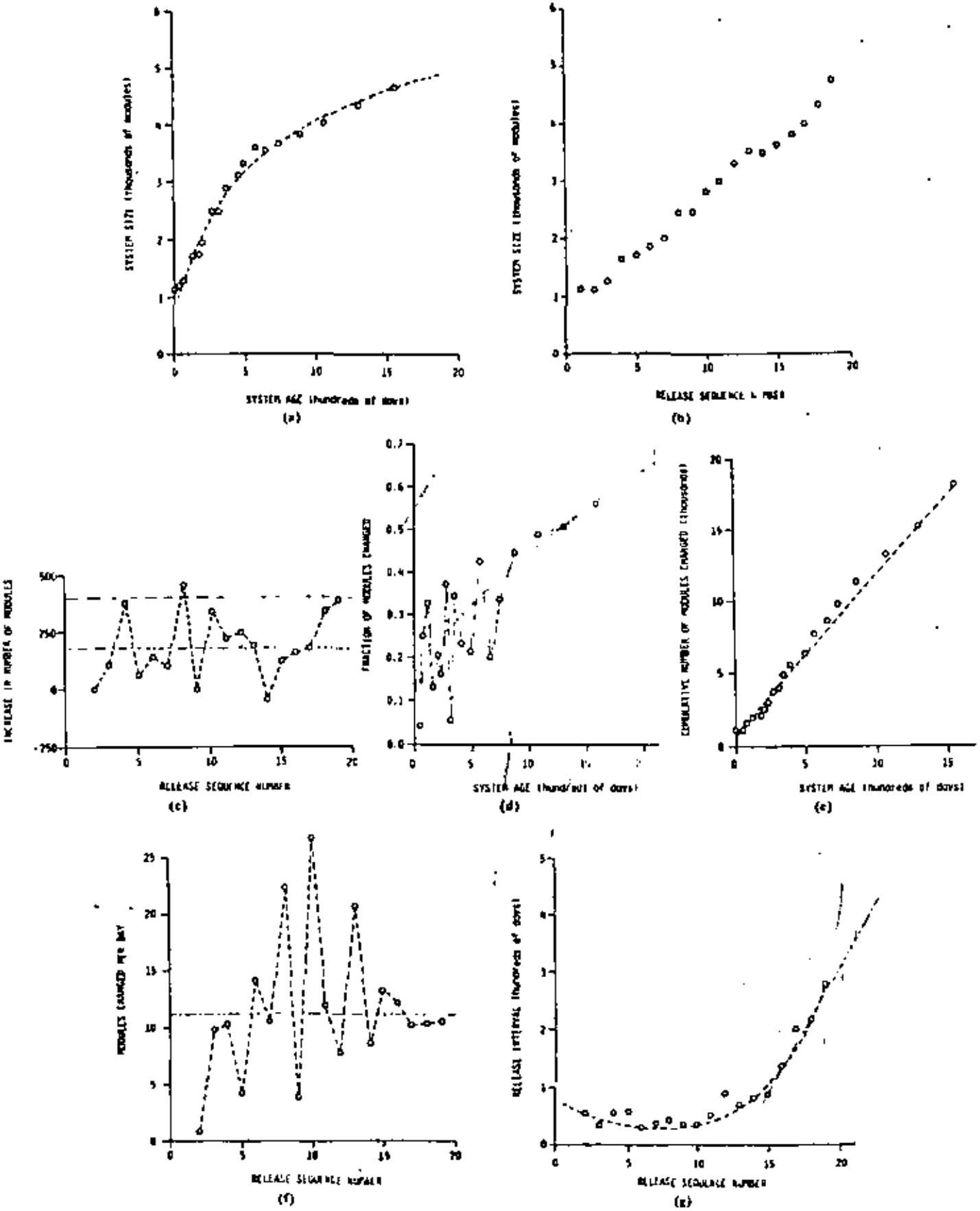


Fig. 6. System characteristics

TABLE III
RELEASE 20 PLANNED CONTENT

Functional Enhancement

No.	Description	New Mods (NM)	Old Mods Chgd (OMC)	Mods Chgd (NM + OMC)	OMC/NM (IR)
1.	Identified faults (PRE. RLS. 19)	2	380	382	–
2.	Expected faults (RLS. 19)	0	600	600	–
3.	Interactive terminal support (ITS)	750	1783	2533	2.4
4.	Dynamic storage management (DSM)	170	1500	1670	8.8
5.	Remote job entry (RJE)	57	462	519	8.1
6.	New disk support (NDS)	17	124	141	7.3
7.	Batch scheduler improvements (BSI)	3	29	32	9.7
8.	File access system (FAS)	8	74	82	9.3
9.	Paper tape support (PTS)	12	80	92	6.7
10.	Performance improvements	2	157	159	–
		1021	5189	6210	

Detail of Interactive Terminal Support (ITS)

No.	Description	New Mods	Old Mods Chgd	OMC/NM
3A	Terminal support	444	1032	2.3
3B	Scheduling	127	293	2.3
3C	Telecom support	58	232	4.0
3D	Misc.	121	226	1.9
		750	1783	

lease) of R19, work has begun on a further version R20, whose main component is to be the addition of interactive access to complement current batch facilities. This new facility "ITS" together with other changes and additions summarized in Table III, are to be made available eighteen months after first customer installation of R19.

For each major planned functional change, the table lists the number of new modules to be added (NM), the number of R19 modules that are to be changed in the course of creating R20 (OMC), the total number of modules changed (NM + OMC), and the ratio of OMC to NM (the interconnectivity ratio (IR), an indicator of complexity). No modules are planned for removal in the creation of R20 hence the planned net system growth is 1021 modules.

Management has also accepted that a further release R21, will follow twelve months after R20, to include any leftovers from R20. It may also include additional changes for which a demand develops over the next two years. The current exercise is to endorse the overall plan, or if it can be shown to be defective, to prepare an alternative recommendation.

3) Process Dynamics:

a) Work rate: From Fig. 6(e) the work rate has averaged 10.4 m/d[1] over the lifetime of the system. Fig. 6(f) indicates that the maximum rate achieved so far has been 27 m/d. Evidence that cannot be detailed here reveals, however, that that data point is misleading and that a peak rate of about 20 m/d is a better indicator of the maximum achievable with current methodology and tools. Moreover, there is strong circumstantial evidence that releases achieved with such high work rates were extremely troublesome and had to be followed by considerable clean-up in a follow-up release, as also implied by Fig. 6(c). Thus, if R20 is planned so as to require a work rate in the region of 20 m/d, it would be wise to limit R21 to at most 10 m/d, the system average. If on the other hand, the process is further stabilized by working on R20 at near average rate, one could then, with a high degree of confidence, approach R21 with a higher work rate plan.

b) Incremental growth: The maintained average incremental growth for system X has been around 200 modules/release. Once again circumstantial evidence indicates that releases (for which, in this case, the growth rate (incremental growth per release) has exceeded twice the average) have slipped delivery dates, a poor quality record and a subsequent need for drastic corrective activity. Fig. 6(c) and Table II indicate that R19 will lie in this region and that R18 had high incremental growth. That is, R19, once released, is likely to prove a poor quality base. The first evidence emerges that maybe R20 should be a clean-up release.

c) Growth rate in modules for release: The same indication follows from Figs. 6(a) and (b) where the ripple periods are seen to be three, four, and five intervals, respectively over the first three cycles. In the fourth cycle, six intervals of increasing growth rate have passed with the R18–R19 growth the largest ever. Without even considering the planned growth to R20 (Point X'), it seems apparent that a clean-up release is due.

4) R20 Plan Analysis:

a) Initial analysis: The first observation on the plan as summarized by Table III stems from the column (6) of IR

[1] Modules per day = $\frac{\text{number of modules changed in release}}{\text{release interval in days}}$

factors. It has not been calculated for items 1, 2, and 10 since these represent activities that only *rarely* require the provision of entirely new (nonreplacement) modules. For items 4–9 the ratio lies in the range 8.2 ± 1.5, a remarkably small range for widely varying functional changes. Yet the predicted ratio for ITS is only 2.4. One must ask whether it is reasonable to suppose that the code implementing an interactive facility is far more loosely coupled to the remaining system than, for example, a specialist facility such as paper tape support? Is it not far more likely that ITS has been inadequately designed; viewed perhaps as an independent facility that requires only loose coupling into the existing system? Thus, when it is integrated with the remainder of the system to form R20, will it not require many more changes to obtain correct and adequate performance? From the evidence before us, the question is undecidable. Experience based intuition, however, suggests that it is rather likely that the number of changes required elsewhere in the system has been underestimated. Thus a high-priority design reappraisal is appropriate. If the suspicion of incomplete planning proves to be correct, it would suggest delaying R20, so that the planning and design processes may be completed. An alternative strategy of delaying at least ITS to R21 should also be evaluated.

b) Number of modules to be changed: The situation may of course not be quite as bad as direct comparison of the present estimate of the ITS interconnection ratio (IR) with that of the other items, suggests. In view of the 750 new modules involved, its IR factor could not exceed 6.4 even if all 4800 modules of R19 were effected by the ITS addition. Such a 100 percent change is, in fact, very unlikely, but the IR factor of 2.4 remains very suspect.

Moreover, even with the low ratio for ITS the sum of the individual OMC estimates for the entire plan exceeds the number of modules in R19. This suggests a new situation. Multiple changes applied to the same module must have become a significant occurrence. Even ignoring the fact that even independent changes applied in the same release to the *same* module generally demand significantly more effort than similar changes applied to independent modules, the total effort and time required must clearly increase with both the number of changes implemented and the number of modules changed. The presently defined measure "modules *changed*" is inadequate. The new situation demands consideration of more sensitive measures such as "number of module *changes*" and "average number of changes per module."

These cannot be derived from the available data. We may, however, proceed by considering a model based on the data of Fig. 6(d). Extrapolating the fraction changed trend, reveals that R20 may be expected to require a change of, say, 64 percent, or 3725 changed modules.[2] Comparing this estimate with the total of 6210 obtained if the estimates for individual items are summed, it appears that the average number of changes to be applied to R19 modules according to the present plan is at least of order two. We have already observed that multiple changes cause additional complications. Hence any prognosis made under the implied assumption of single changes (or of a somewhat lower interconnection ratio) will lead to an optimistic assessment.

[2] *Historical Note:* In the system on which this example is based the release including the interactive facility ultimately involved some 58 percent of modules changed. Moreover the first release was significantly delayed, and was of limited quality and performance. More than 70 percent of its modules had subsequently to be changed again to attain an acceptable product. Our estimate is clearly good.

c) Rate of work: The current plan calls for R20 with its 3725 module changes to be available in 18 months, that is 548 days. This implies a change rate of less than 6.8 *m/d*. This relatively low rate, following a period of average rate activity suggests that *work rate* pressures are unlikely to prove a source of trouble, even with multiple changes to many of the modules.

d) Growth rate: In Figs. 6(a) and (b), we have indicated the position of R20 as per plan, with an *X*. Both modules indicate that the planned growth represents a major deviation from the previous history. Thus confirmation that the plan is realistic requires a demonstration that the special nature of the release, or changes in methodology, makes it reasonable to expect a significant change in the system dynamics. In the absence of such a demonstration, the suspicion that all is not well is strengthened.

e) Incremental growth: The current R20 plan calls for system growth of over 1000 modules. This figure which is five times the average and two and a half times the recommended maximum, must be interpreted as a danger signal.

We have already suggested that the low interconnection ratio for ITS suggests that the planners saw the new component as a stand alone mechanism that interfaces with the remainder of the system via a narrow and restricted interface. If this view proves justified, the large incremental growth need not be disturbing. But it seems reasonable to question it. With the architecture and structure that system *X* is known to have, such a relatively narrow interface is unlikely to be able to provide the communication and control bandwidth that safe, effective, and high capacity operation must demand. This is apparent from comparisons with, say, the paper tape or disk support changes or the RJE addition. The onus must be put onto the ITS designers to demonstrate the completeness of their analysis, design, and implementation.

Without such a demonstration one must conclude that the present plan is not technically viable. *Marketing* or other considerations may, of course, make it desirable to stay with the present plan even if this implies slipped delivery dates, poor and unreliable performance of the new release, limited facilities, and so on. But if such considerations force adoption of the plan, the implications must be noted, and corrective action planned. Ways and means will have to be created to enable users to cope with the resultant system and usage problems and the inevitable need for a major clean-up release. It might, for example, be wise to set up specialized customer support teams to assist in the installation, local adaptation and tuning of the system.

f) Release interval: Fig. 6(g) indicates two possible model for the prediction of the most likely (desirable?) release interval for R20 and R21. Linear extrapolation suggests a release period of under one year for each of the two releases. If this is valid, the apparent desire for a release after the 18 months is of itself unlikely to prove a source of problems. On the basis of evidence not reproduced here, however, the exponential extrapolation is likely to be more realistic and this yields an R20 release interval forecast of about 15 months and an R21 interval of some 3 years.

g) Recommendation–Summary: On the basis of the available data we have concluded that

1) to proceed with the plan as it stands is courting delivery and quality problems for R20;
2) a clean-up release appears due in any case;
3) failure to provide it will leave a weak base for the next

TABLE IV
MODIFIED RELEASE 20 CONTENT

Class	Reason	Items
Fault Repair	Clean-up of base	1, 2.
Hardware Support	Revenue Producing	6, 9.
Performance Improvement	Install—but do not announce. Will be available to counteract ITS performance deterioration in R21'	7, 10
ITS Related Components	To receive early user exposure	3c, 5, 8.

TABLE V
MODIFIED RELEASE 20 STATISTICS (FROM TABLE III) IN ORDER OF PRIORITY

Item	New Mods.	Running Total	Changes	Running Total
1	2	2	382	382
2	0	2	600	982
6	17	19	141	1123
9	12	31	92	1215
7	3	34	32	1247
10	2	36	159	1406
8	8	44	82	1488
5	57	101	519	2007
3c	58	159	522	2529

release; at the very least the number of expected faults (Table III, item 2) is likely to prove an underestimate;

4) the absolute size of the ITS component and the related incremental system growth would represent a major challenge even on a clean base;
5) there are indications that the ITS aspect of the release design is incomplete;
6) change rate needs for R20 are not likely to prove a source of problems;
7) nor is the demand for attainment of a next release in eighteen months.

The following recommendations follow:

8) initiate immediately an intensive and detailed reexamination of the ITS design and its interaction with the remainder of System X;
9) from the integration records of R19 and by comparison with the records of earlier releases, make quality and error rate models and obtain a prognosis for R19 and an improved estimate for R20 correction activity; integration and error rate models have not been considered in the present paper but have been extensively studied by the present author and by others [85];
10) assess the business consequences of, on the one hand, a slippage of one or two years in the release of ITS and on the other, a poor quality, poor performance release with a slippage of, say, some months (due to acceptable work rate but excessive growth);
11) in the absence of positive indication of a potential for major deviations from previous dynamic characteristics or the existence of a genuine business need that is more pressing than the losses that could arise from a poor quality product, abandon the present plan;
12) instead redesign release 20 to yield R20'; a clean, well-structured, base on which to build an ITS release, R21';
13) tentatively release intervals of 9 months and 15 months are proposed for R20' and R21', respectively;
14) R21' should be a restricted release for installation in selected sites;
15) it would be followed after 1 year by a general release R22'.

a) Recommendations—Details: Assuming that the further investigation as per paragraphs 8 to 10 of Section V-B4g reinforces the conclusions reached, three releases would have to be defined. We outline here proposals for R20' and R21'. The third, R22' will be a clean-up but its content cannot be identified in detail until a feel for the performance and general quality of R21' has developed. The detailed analysis is left as an exercise to the reader.

The inherent problem in the design of the ITS release is the fact that the component has a size almost twice the maximum recommended incremental growth. Moreover, with the possible exception of its telecommunications support (Table III, item 3c), none of the component subsystems would receive usage exposure in the absence of the others. Thus a clean ITS release cannot be achieved except by releasing the component in one fell swoop. Similarly, dynamic storage management (DSM) is exposed to user testing only when the ITS facility is operational. We may, however, investigate whether the telecommunication facility (3c) will be usable in conjunction with the RJE facility, item 5. If it is, there will be some advantage to be gained by releasing 3c and 5 before the remainder of ITS and DSM.

Strictly speaking, Fig. 6(c) suggests that R20' should be a very low content release dedicated to system clean-up and restructuring. But the six preceding releases were achieved with average change rates and, from that point of view, did not stress the process. Thus, if R20' is also an average rate release, it should not cause problems, and it would seem a low risk strategy to include R20' in all those items as in Table IV, that will simplify the subsequent creation and integration of the excessively large ITS release.

The list, in priority order, of the new proposal shows a maximum incremental growth (159) well under average. It is a matter of some judgment and experience whether it would be wiser to delay item 3c with 58 new modules and item 5 with 57 to R21' thereby achieving the very low content release mentioned above. With the information before the reader it is not possible to resolve this question since additional information, at the very least answers to the questions raised in Section V-B4g, would be required. However, the desire to minimize R21' problems suggests the adoption of the complete plan as in Tables IV and V.

In assessing achievable release intervals for these releases, we base our estimates only on the module change count and change rate. The constraints on the present example do not permit the full analysis which would consider models based on Fig. 6(g), and take into account additional data. At 10 m/d change rate, implementation of the complete plan appears to require 253 days, say 9 months, whereas exclusion of 3c and 5 would reduce the predicted time required to some seven months. This recommendation cannot be taken further without more information of both a technical and a marketing nature, and an examination of other interval models. But the need for a clean base for R21' suggests adoption of the

maximum acceptable release interval. R21' will now include, at the very least, ITS (except 3c) and DSM. This involves at least 920 new modules, an excessive growth that cannot usefully be further split between two or more releases. Assuming a change fraction of, say, 70 percent (Fig. 6(d)), of a system that is expected to contain 5911 modules, we estimate a total of 4200 changed modules in the release, many with multiple changes. Since there will now have been seven near average change-rate releases, it seems possible to plan for a change rate of 15–20 m/d, yielding a potential release interval of under 9 months. That is, it would appear that, by adopting the new strategy, all of the original changes and additions could be achieved in about the same time, but much more reliably. More complete analysis, however, based on additional data, other models and taking into account the special nature of the releases might well lead to a recommendation to increase the combined release interval to, say, two years.

A further qualification must also be added. As proposed in the revised plan, R21' will still be a release with excessive incremental growth and is therefore likely to yield significant problems. The additional fact that the evidence indicates incomplete planning, reinforces concern and expectation of trouble ahead. It is therefore also recommended that R21' be announced as an experimental release for exposure to usage by selected users in a variety of environments. It would be followed after an interval of perhaps one year by an R22', a cleaned up system, suitable for further evolution.

i) Final comments: The preceding section has presented a critique of a plan, and outlined an alternative which is believed technically more sound. The case considered is based on a real situation, though in the absence of complete information details have had to be invented. But the details are not important since the objective has been to demonstrate a methodology. Software planning can and should be based on process and system measures and models, obtained and maintained as a continuing process activity. Plans must be related to dynamic characteristics of the process and system, and to the statistics of change. By rooting the planning process in *facts*, *figures* and *models*, alternatives can be quantitatively compared, decisions can be related to reality and risks can be evaluated. Software planning must no longer be based solely on apparent business needs and market considerations; on management's local perspective and intuition.

VI. Conclusion

This paper rationalizes the widely held view, first expressed in Garmisch [82], that there is an urgent need for a discipline of software engineering. This should facilitate the cost-effective planning, design, construction, and maintenance of effective programs that provide, and then continue to provide, valid solutions to stated (possibly changing) problems, or satisfactory implementations of (possibly changing) computer applications.

Following a brief discussion of the nature of computer usage and of the programs, the paper introduced the new SPE classification that addresses the essential evolutionary nature of various types of programs and establishes the existence of a determining specification as the criterion for nonevolution.

In the subsequent discussion of the concepts, significance and phases of the program life cycle, no details of life-cycle planning and management models, as such, have been included. In particular, we have [illegible] discussed cost, resource, and reliability models [83]–[85]. Approaches to process modeling based on continuous models [73], [75] have also not been included, nor has the vital topic of software complexity [86]–[89].

Recognizing the intrinsic nature of program change, the laws that appear to govern the dynamics of the evolution process were introduced. Among their other implications, the laws indicate that project plans must be related to dynamic characteristics of the process and system, and to the statistics of change. By rooting the planning process in facts, figures, and models, alternatives can be quantitatively compared, decisions can be related to reality and risks can be evaluated. Software planning must no longer be based solely on apparent business needs and market considerations; on management's local perspective and intuition. To illustrate this, we have included a brief example of the application of evolution dynamics models to release planning.

Many of the concepts and techniques presented in this paper could find wide applications outside the specific area of software systems, in other industries, and to social and economic systems. Unfortunately that theme cannot be pursued here.

Acknowledgment

First and foremost, thanks must be extended to L. A. Belady, a close collaborator for almost ten years. Many others, particularly colleagues and associates at ALMSA, IBM, Imperial College, and WG 2.3 have contributed through their comments, questions, critique, and original thoughts. All of them deserve and receive the author's grateful acknowledgments and thanks for their individual and collective contributions. The author would like to single out Prof. W. M. Turski for the major contribution he made on his recent visit to London. Also, sincere thanks to Dr. G. Benyon-Tinker, Dr. P. G. Harrison, and Dr. C. Jones for their detailed and constructive criticism of an early draft of this paper and R. Bailey for his artistic support. Finally the author would like to acknowledge the constant support of his wife, without which neither the work itself nor this paper would have been possible.

References

[1] J. Goldberg, Ed., in *Proc. Symp. High Cost of Software* (Naval Post-grad. School, Monterey, CA). Menlo Park, CA: SRI, 1973, 138 pp.

[2] M. M. Lehman, "The environment of design methodology," Key note Address, in *Proc. Symp. Formal Design Methodology*, T. A. Cox, Ed. (Cambridge, England), Apr. 1979. Harlow, England: STL Ltd., 1980, pp. 18–38.

[3] ——, "The software engineering environment," Infotech State of the Art Rep., "Structured software development," P. J. L. Wallis Ed., vol. 2, pp. 147–163, 1979.

[4] W. A. Wulf, "Languages and structured programs," in *Current Trends in Programming Methodology*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 33–60.

[5] L. A. Belady, Ed., *Proc. IEEE Special Issue on Software Engineering*, vol. 68, Sept. 1980.

[6] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, pp. 1226–1241, Dec. 1976.

[7] W. M. Turski, *Computer Programming Methodology*. London, England: Heyden, 1978, 208 pp.

[8] B. W. Boehm, "Software engineering—As it is," in *Proc. 4th Int. Conf. Software Engineering* (Munich, Germany), pp. 11–21, Sept. 1979. (IEEE Cat. no. 79CH1479-5C.)

[9] L. A. Belady and M. M. Lehman, "Characteristics of large systems," in *Research Directions in Software Technology*, P. Wegner Ed. Cambridge, MA: M.I.T. Press, 1979, part 1, ch. 3, pp. 106–142 (sponsored by the Tri-Services Committee of DoD), and *Proc. Conf. Research Directions in Software Technology* (Brown University, Providence, RI), Oct. 10–13, 1977.

[10] M. M. Lehman, "Programs, cities, students—Limits to growth?" Inaugural Lecture, May 14, 1974, *ICST Inaugural Lecture Ser.*

vol. 9, pp. 213–229, 1970–1974; and in *Programming Methodology*, D. Gries, Ed. New York: Springer-Verlag, 1979, pp. 42–69.
[41] ——, "On understanding laws, evolution and conservation in the large program life-cycle," *J. Syst. Software*, vol. 1, no. 3, pp. 213–232, 1980.
[42] W. M. Turski, "Report on an SRC-sponsored visit to Imperial College," Dep. Computing, Imperial College of Science and Technology, Univ. of London, London, England, Oct. 1979, 2 pp.
[43] F. L. Bauer, H. Partsch, P. Pepper, and H. Wessner, "Notes on the project-CIP: An outline of a transformation system," TUM-INFO-7729, Tech. Univ. Munich, 61 pp, 1977.
[44] J. Darlington, "Program synthesis transformation: An introduction and survey," *Comput. Bull.*, ser. 2, no. 23, pp. 22–24, Dec. 1979.
[45] H. A. Simon, *The Sciences of the Artificial*. Cambridge, MA: M.I.T. Press, 1969, 123 pp.
[46] R. A. Demillo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," *Commun. Ass. Comput. Mach.*, vol. 22, no. 5, pp. 271–280, May 1979; and no. 11, pp. 628–630, Nov. 1979.
[47] C. A. R. Hoare, "Review of a paper by Demillo, Lipton and Perlis: 'Social processes and proofs of theorems and programs,'" *ACM Comput. Rev.*, vol. 22, no. 8, rev. no. 34897, p. 324, Aug. 1979.
[48] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *Nordisk Tidsrift for Informations. Behandling, Sweden*, vol. 8, pp. 174–186, 1969.
[49] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, no. 10, pp. 576–583, Oct. 1969.
[50] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.
[51] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, no. 4, Apr. 1971, pp. 221–227.
[52] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. New York: Academic Press, 1972, pp. 1–81.
[53] M. A. Jackson, *Principles of Program Design*. London, England: Academic Press, 1975, 299 pp.
[54] N. Wirth, "The module: A system structuring facility in high-level programming languages," in *Proc. Symp. Programming Languages and Programming Methods* (Sydney, Austral.), J. Tobias, Ed. Lucas Hts., New South Wales: AAEC, 1979.
[55] B. Liskov and S. Zilles, "An introduction to formal specification of data abstraction," in *Current Trends in Programming Methodology*, vol. 1, *Software Specification and Design*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 1–32.
[56] C. Jones, *Software Development—A Rigorous Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1980, 400 pp.
[57] M. Shaw, "The impact of abstraction concerns on modern programming languages," this issue, pp. 1119–1130.
[58] M. M. Lehman, "The funnel—A functional channel," Imperial College, Dep. Computing, Univ. of London, Res. Rep. 77/29, July 1977, 14 pp.; and *IBM Tech. Disclosure Bull.*, 1976.
[59] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. New York: Academic Press, 1972, 220 pp.
[60] R. C. Linger, and H. D. Mills, "On the development of large, reliable programs," in *Current Trends in Programming Methodology*, vol. 1, *Software Specification and Design*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 120–139.
[61] M. M. Lehman, "OS/VS2-MVS long range prognosis," Private communication, MML-104, 13 pp. Apr. 15, 1975.
[62] T. A. DeLotta and J. R. Mashey, "An introduction to the programmer's workbench," in *Proc. 2nd Int. Conf. Software Engineering* (San Francisco, CA) Oct. 1976. (IEEE Cat. no. 76CH1125-4C, Oct. 1976, pp. 164–168.)
[63] A. F. Hutchings, R. W. McGuffin, A. E. Ellison, B. R. Trauter, and P. N. Westmacott, "CADES—Software engineering in practice," *Proc. 4th Int. Conf. Software Engineering* (Munich, Germany), Sept. 1979. (IEEE Cat. no. 79CH1479-5C, Sept. 1979, pp. 156–157.)
[64] J. N. Buxton, "Requirements for ADA programming support environment—STONEMAN," U.S. Dep. of Defense, Washington, DC, 44 pp., Feb. 1980.
[65] T. E. Bell, D. C. Bixler, and M. E. Dyer, "An extendable approach to computer-aided software requirements engineering," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 49–59, Jan. 1977.
[66] M. W. Alford, "Software requirements engineering methodology (SREM) at the age of two," in *Proc. COMPSAC 78*, pp. 331–339, Nov. 1978. (IEEE Cat. no. 78CH1338-3C.)
[67] K. Heninger, "Specifying requirements for complex systems: New techniques and their application," in *Proc. Specification of Reliable Software Conf.*, pp. 1–14, Mar. 1979. (IEEE Cat. no. 79CH1401-9C.)
[68] R. T. Yeh, and P. Zave, "Specifying software requirements," this issue, pp. 1077–1085.
[69] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, 1974.
[70] G. J. Myers, *Composite/Structured Design*. New York: Van Nostrand Reinhold, 1978, 134 pp.
[71] D. T. Ross, and K. E. Schuman, "Structuring analysis for requirements definition," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 6–15, Jan. 1977.
[72] ——, "Structured analysis (SA): A language for communicating ideas," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 16–33, Jan. 1977.
[73] T. Demarco, *Structured Analysis and System Specification*. New York: Yourdon Press, 1978, 352 pp.
[74] B. W. Likov and V. Berzins, "An appraisal of program specifications," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, Part 2.1, ch. 7, pp. 406–142 (sponsored by the Tri-Services Committee of DoD); and in *Proc. Conf. Research Directions in Software Technology* (Brown University, Providence, RI), pp. 276–301, Oct. 30–32, 1977.
[75] J. N. Buxton, and E. Randell, Eds., "Software engineering techniques," Rep. Conf. sponsored by the NATO Science Committee (Rome, Italy), Oct. 1969. (Brussels, 164 pp. 1970.)
[76] P. Van Leer, "Top-down development using a program design language," *IBM Syst. J.*, vol. 15, no. 2, pp. 155–170, 1976.
[77] Teichroew and E. A. Hershey III, "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 41–48, Jan. 1977.
[78] R. T. Yeh, "Current trends in programming methodology," vol. 1, *Software Specification and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1977, 275 pp.
[79] T. A. Cox, Ed., *Proc. Symp. Formal Design Methodology* (Cambridge, England), 1977. Harlow, England: STL Ltd., 1980, 350 pp.
[80] F. W. Zurcher and B. Randell, "Iterative multi-level modelling—A methodology for computer system design," in *Proc. IFIP Congr. 1968* (Edinburgh, Scotland), pp. D138–142, Aug. 1968.
[81] L. Peters, "Software design engineering," this issue, pp. 1085–1093.
[82] G. H. Swaum, *Top-Down Structured Design Techniques*. New York: Petrocelli Books, 1978, 140 pp.
[83] E. Miller and W. E. Howden, Eds., "Tutorial: Software testing and validation technique," *IEEE Comput. Soc.*, 423 pp., 1978. (IEEE Cat. no. EHO-138-8.)
[84] J. B. Goodenough and L. M. Clement, "Software quality assurance testing and validation," this issue, pp. 1093–1098.
[85] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156–173, June 1975.
[86] L. A. Belady and P. M. Merlin, "Evolving parts and relations—A model of system families," IBM Res. Rep. RC8677, 14 pp, Aug. 1977.
[87] M. M. Lehman and L. H. Putnam, Eds., "Software phenomenology, working papers of the (first) software life cycle management workshop (Airlie, VA), Aug. 1977. Fort Belvoir, VA: ISRAD/AIRMICS, Computer Systems Command, U.S. Army, Dec. 1977, 683 pp.
[88] V. R. Basili, E. Ely, and D. Young, Eds., "Second software life-cycle management workshop, 21–22 Aug. 1978 (Atlanta, GA)," 220 pp., Dec. 1978. (IEEE Publ. no. 78CH1390-4C.)
[89] C. P. Idris and C. E. Walston, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54–73, 1977.
[90] L. H. Putnam and R. W. Wolverton, "Quantitative management—Software cost estimating," in *Proc. Comp. Soc. 77, IEEE Computer Software and Applications Conf. (Tutorial)*, 326 pp., Nov. 1977. (IEEE cat. no. EHO129-7.)
[91] L. H. Putnam, "The influence of the time-difficulty factor in large scale development" in *Proc. Software Phenomenology Working Papers of the (first) Software Life-cycle Management Workshop* (Airlie VA), Aug. 1977. Fort Belvoir, VA: ISRAD/AIRMICS, Computer Systems Command, U.S. Army, Dec. 1977, pp. 303–312.
[92] B. W. Boehm, and R. W. Wolverton, "Software cost modelling—Some lessons learned," in *Proc. 2nd Software Life-cycle Management Workshop*, Aug. 21–22, 1978 (Atlanta, GA) pp. 139–132, Dec. 1978. (IEEE Publ. no. 78CH1390-4C.)
[93] F. N. Parr, "An alternative to the Rayleigh curve model for software development effort," *IEEE Trans. Software Eng.*, vol. SE-6, May 1980, pp. 291–296.
[94] S. C. Aron, *The Program Development Process, Part II The Programming Team*. Reading, MA: Addison-Wesley, 1980.
[95] M. Halstead, *Elements of Software Science*. New York: Elsevier, 1977, 127 pp.
[96] A. Fitzsimmons and T. Love, "A review and evaluation of software science," *Computing Surveys*, vol. 10, no. 1, pp. 3–18, Mar.

5.- BIBLIOTECA DE SOPORTE DE PROGRAMAS

5-1-0

INTRODUCCION

EL PROPOSITO DE ESTE DOCUMENTO ES EL PRESENTAR A INVESTIGADOR DEL DEPARTAMENTO DE ANALISIS DE REDES EL USO DE LA BIBLIOTECA DE SOPORTE DE PROGRAS, ASI COMO EL PAPEL QUE JUEGA ESTA EN EL DESARROLLO DE LOS PROGRAMAS Y SISTEMAS.

5-1-1

ASPECTOS GENERALES.

EN GENERAL Y DURANTE MUCHO TIEMPO HA SIDO RESPONSABILIDAD DE LOS PROGRAMADORES EL DISEÑAR, PROGRAMAR Y MANTENER LOS PROGRAMAS Y EL CODIGO GENERADO. DADO QUE USUALMENTE EL TAMAÑO DE LOS PROYECTOS TIENDE A AUMENTAR ASI COMO SU COMPLEJIDAD E INTERFAZ CON OTROS SISTEMAS, EL PROBLEMA DE MANIPULAR Y MANTENER PROGRAMAS SE TORNA EN UNA TAREA DIFICIL. LOS ARCHIVOS DE ENTRADA Y SUS MODIFICACIONES NO SON FACILES DE MANTENER: MODULOS FUENTE OBJETO Y DE CARGA PUEDEN NO SER EQUIVALENTES, CAMBIOS PEQUEÑOS EN TAREAS PUEDEN TENER EFECTOS CATASTROFICOS EN EL SISTEMA, DEMAS RESULTA "RARO" QUE EL MISMO PROGRAMADOR SE PRUEBE SUS PROPIOS "ERRORES". ES DECIR SER JUEZ Y PARTE

POR LOS MOTIVOS ANTERIORES SE HACE DE VITAL IMPORTANCIA CONTAR CON UN SISTEMA DE CONTROL CENTRALIZADO EN EL CUAL SE LLEVE LA ADMINISTRACION DE LOS PROGRAMAS Y SISTEMAS DESARROLLADOS.
SE ASIGNA UN "BIBLIOTECARIO" EN LA TAREA DE MANTENER LOS PROGRAMAS DAR RESPALDO A LOS MISMOS. ASI COMO LLEVAR UN CONTROL DE LOS ELEMENTOS Y PROGRAMAS QUE CONFIGURAN EL SISTEMA.

5-1-2

LOS DOCUMENTOS QUE FORMAN LA BIBLIOTECA DE SOPORTE DE PROGRAMAS SON:

PROGRAMAS FUENTE
PROGRAMAS OBJETO
PLAN DE PRUEBAS
NORMAS DE CODIFICACION
NORMAS DE ANALISIS
NORMAS DE DISEÑO
NORMAS DEL ROL
ARCHIVO DE ENTRADA
ARCHIVOS DE SALIDA
REPORTES
MANUALES
ESPECIFICACIONES
RPC
DDPC
Y FORMAS DE LA BIBLIOTECA

5-1-3

PROPOSITO DE LA BIBLIOTECA

EL PROPOSITO DE LA BIBLIOTECA DE SOPORTE DE PROGRAMAS (BSP) ES EL DE PROVEER UNA METODOLOGIA SISTEMATICA Y CONSISTENTE PARA:

. ASEGURAR EL CONTROL TECNICO Y LA EVOLUCION DE CADA ELEMENTO DEL SISTEMA.

. PERMITIR UNA FORMA DE ADMINISTRACION Y CONTROL DE LOS PROGRAMAS

.USO, MANTENIMIENTO Y RESPALDO DE LOS ELEMENTOS DEL PROGRAMA ASI COMO TENER SIEMPRE LA ULTIMA VERSION DE PROGAMA Y DOCUMENTO A DISPONIBILIDAD DEL USUARIO.

LA BSP SE COMPONE DE DOS PARTES, LA BSP EXTERNA Y LA BSP INTERNA.

LA BSP EXTERNA SE COMPONE DE:

1.- DOCUMENTOS RELACIONADOS CON EL PROYECTO.
2.- DOCUMENTOS RELACIONADOS CON EL CONTROL "INTERNO DE LA BSP.

LAS RESPONSABILIDADES DEL BIBLIOTECARIO SON LAS DE MANTENER EL CONTENIDO DE LA BSP EN UNA FORMA ORDENADA Y SISTEMATIZADA Y LAS LABORES QUE TIENE QUE DESARROLLAR SON LAS SIGUIENTES:

-CATALOGAR TODOS LOS DOCUMENTOS, LISTADOS Y MANUALES. PROVEER A LOS USUARIOS PERIODICAMENTE DE UNA LISTA DE PRODUCTOS DENTRO DE LA BIBLIOTECA.

-DISTRIBUIR COPIAS DE LOS DOCUMENTOS Y PROGRAMAS DE ACUERDO A LISTAS DE DISTRIBUCION

-TENER LISTAS DE LOS PROGRAMAS "TERMINADOS" EN PODER DE LA BIBLIOTECA.

DISTRIBUIR Y GENERAR LISTAS DE ESTATUS DEL SOFTWARE.

LA LIBRERIA INTERNA CONSISTE EN TODOS LOS ARCHIVOS Y DOCUMENTOS QUE FUERON GENERADOS Y QUE ESTAN BAJO LA CUSTODIA DEL BIBLIOTECARIO.. EL BIBLIOTECARIO ES LA UNICA PERSONA AUTORIZADA, PARA MODIFICAR LOS ARCHIVOS, EL RESTO DE LOS INVESTIGADORES SOLO PODRAN LEERLOS, MAS NO MODIFICARLOS SIN LA PREVIA AUTORIZACION DE LA BIBLIOTECA Y SIGUIENDO LAS REGLAS PREESTABLECIDAS Y LAS QUE SE EXPLICAN EN ESTE DOCUMENTO. LOS ARCHIVOS INTERNOS DE LA BSP SON:

PROGRAMAS Y CODIGO FUENTE
PROGRAMAS O ELEMENTOS DE PROGRAMA EN CODIGO OBJETO.
CAMBIOS.-TODOS LOS CAMBIOS QUE FUERON LLEVADOS A CABO EN ARCHIVOS PROGRAMAS MODULOS MANUALES ETC., ASI COMO LA PAPELERIA QUE DIO LUGAR A DICHOS CAMBIOS.
PLAN DE PRUEBAS Y RESULTADO DE LAS MISMAS.

PROCEDIMIENTOS DE LA BSP INTERNA

LAS SIGUIENTES FUNCIONES SON LLEVADAS A CABO COMO PARTE DE LA OPERACION CON EL OBJETIVO DE MANTENER Y GENERAR EL CONTENIDO INTERNO DE LA BSP.

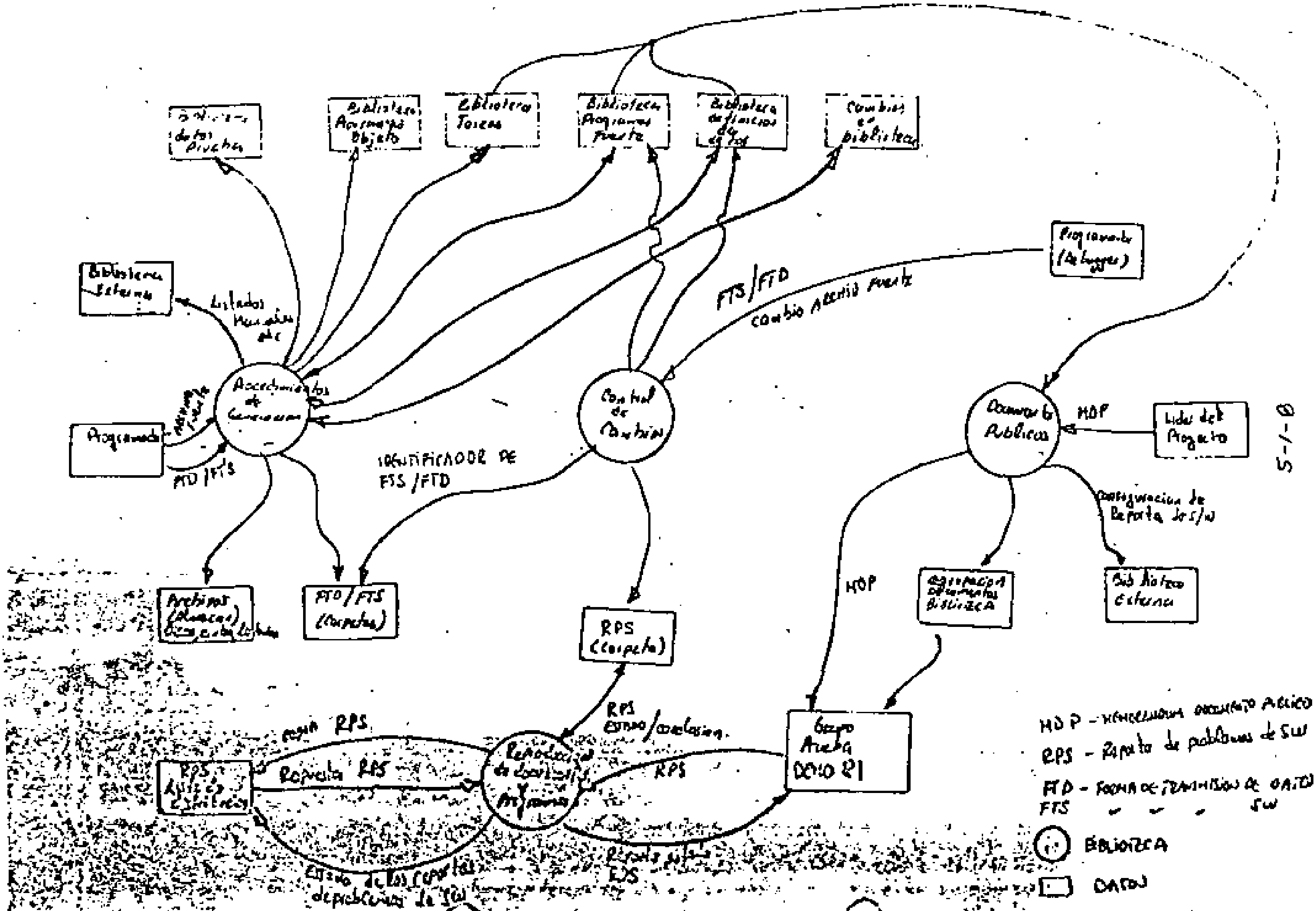
1.- GENERACION INICIAL DE ARCHIVOS.- CUALQUIER PROGRAMA, TAREA, SUBRUTINA MANUALES ETC. HAY QUE HACER NOTAR QUE LOS PROGRAMAS QUE HAN SIDO ENTREGA A LA BSP SON Y HAN SIDO PROGRAMADOS Y ADEMAS CUMPLEN CON LAS NORMAS DE CALIDAD, DISEÑO Y EL LENGUAJE DESCRIPTOR DENTRO DEL ENCABEZADO DE CADA MODULO O PROGRAMA. CUALQUIER ELEMENTO DENTRO DE LA BSP ESTA CONGELADO SIN EMBARGO CUALQUIER CAMBIO DE ESTE PRODUCTO "CONGELADO" SE DEBERA HACER A TRAVES DE LOS PROCEDIMIENTOS DE CAMBIO/CORRECCION/ ALTERACION DE LA BIBLIOTECA DE SOPORTE DE PROGRAMAS. LOS ARCHIBOS DE LA BSP ESTAN A DISPOSICION DE LOS INVESTIGADORES DEL GRUPO DE AR.

2.- MANTENIMIENTO DE ARCHIVOS.- CUALQUIER CAMBIO DE MODULOS BAJO EL CONTROL DE LA BSP SE DEBERA HACER BAJO LOS LIMITAMIENTOS DE CONTROL DE CAMBIO. Y ESTOS DEBERAN TENER LA AUTORIZACION DE MAURICIO MIER MUTH, O DE ROLANDO NIEVA, ADEMAS SE DEBERAN DE LLEVAR LOS PROCEDIMIENTOS QUE INDIQUEN SI DICHO CAMBIO SE DEBE A ERROR EN EL PROGRAMA, NUEVO PEDIDO DE CFF-CENACE. ERROR EN LA ENTRADA, ERROR EN EL RPC, ETC. LA BIBLIOTECA SE HARA CARGO DE REGRASAR EL MODULO AL GRUPO QUE LE HARA LA CORRECCION ASI COMO LLEVAR A LOS PROCEDIMIENTOS PARA ACTUALIZAR CUALQUIER OTRO MODULO QUE RESULTE AFECTADO. LA BSP TENDRA LISTADOS "OFICIALES" DE LA VERSION A SER DISTRIBUIDA.

3.- CONSTRUCTOR DE MODULOS DE CARGA.- LAS VERSIONES ACTUALIZADAS DE LOS MODULOS DE PROGRAMAS DE COMPUTADORA SON ENCADENADOS POR EL GRUPO DE DESARROLLO DE PROGRAMAS, TENIENDOSE UN MODULO EJECUTABLE. ES RESPONSABILIDAD DEL BIBLIOTECARIO EL VERIFICAR QUE DICHOS MODULOS CORREN, ESTAN COMPLETOS, PASAN EL FILTRO DEL VERIFICADOR DE CODIGO, ASI COMO EXISTEN LOS DOCUMENTOS DE RESPALDO. UN ARCHIVO O MODULO GENERADO ES PROPIEDAD DE LA BIBLIOTECA Y POR TANTO SUJETO A SUS REGLAS Y PROCEDIMIENTOS.

4.- REPORTE DE ESTADO DEL SISTEMA.- CADA VEZ QUE SE LLEVA A CABO ALGUN CAMBIO EN LA CONFIGURACION DEL SISTEMA (ACTUALIZACION, CONSTRUCCION DE MODULOS) LA BIBLIOTECA PROVEERA REPORTES PERIODICOS SOBTE ESTADO DEL SISTEMA Y SU CONFIGURACION. ESTE REPORTE CONTIENE EL MODULO ASI COMO TODOS LOS COMPONENTES DE QUE SE COMPONE. SE GENERAN TAMBIEN REPORTES DE CAMBIO EN LA CONFIGURACION.

5.- REPORTE EN PROBLEMAS DE SOFTWARE,- ERRORES Y DEFICIENCIAS ENCONTRADAS EN LOS PROGRAMAS O ARCHIVOS DE DATOS DEBEN DE SER REPORTADOS A TRAVES DE LAS FORMAS QUE YA EXISTENTES, TANTO POR LOS PROGRAMADORES, COMO POR LOS USUARIOS A LA BSP. ES LA RESPONSABILIDAD DEL BIBLIOTECARIO LA DE UN NUMERO DE ENTRADA A LA FORMA Y SEGUIR LOS PROCEDIMIENTOS ADECUADOS PARA QUE EL ERROR SEA CORREGIDO. SE DARAN TAMBIEN REPORTES DE PROBLEMAS DE CERRADOS QUE SON LOS QUE YA HAN SIDO CORREGIDOS. SE DAN TAMBIEN PRIORIDADES EN LA CORRECCION DE PROGRAMAS.


Biblioteca Tareas
Biblioteca Programas Fuente
Cambios en biblioteca
Biblioteca Externa
Listados Mensajes etc
Programador
FTD / FTS
Programador (debugger)
FTS / FTD
cambio Archivo Fuente
Control de Cambios
Lider del Proyecto
HDP
IDENTIFICADOR DE FTS / FTD
Archivos
FTD / FTS (Carpeta)
RPS (Carpeta)
Biblioteca Externa
RPS
Respuesta RPS
RPS - Reporte de problemas de SW
BIBLIOTECA
DATOS

EN ALGUNOS DOCUMENTOS DE LA ASP EXISTEN PRIORIDADES RECOMENDADAS BASADAS EN EL IMPACTO DE ERROR O DEL CAMBIO EN EL SISTEMA. ESTA PRIORIDAD ES FIJADA POR EL LIDER DEL PROYECTO, O JEFE DEL GRUPO ENCONJUNTO CON EL EL GRUPO DE DEMOSTRACION. EXISTEN TRES CLASES DE PRIORIDADES:

CRITICA...ESTA CLASE TIENE UN IMPACTO SEVERO EN EL SISTEMA O RESULTADOS A OBTENER , Y SE DEBE ACTUAR INMEDIATAMENTE EN EL REPORTE CALIFICADO COMO DE CRITICO.

MEDIANO... ESTE PROBLEMA TENDERA UN IMPACTO DEL TIPO MEDIANO, LO CUAL QUIERE DECIR QUE SE ESTA PROCESANDO ALGUNA INFORMACION ERRONEA SIN EMBARGO EL PROGRAMA CORRE EN CONDICIONES "NORMALES". ESTE ERROR SE DEBE CORREGIR ANTES DE ENTREGAR EL PRODUCTO.

BAJO...ESTE TIPO DE PROBLEMAS SE CATALOGAN LO QUE CASI NO TIENEN IMPACTO EN EL EL SISTEMA, NO SE REQUIERE UNA SOLUCION INMEDIATA.

5-1-10

F O R M A S Y E J E M P L O S

EL DEPARTAMENTO DE ANALISIS DE REDES HA GENERADO UNA SERIE DE FORMAS QUE FORMAN PARTE DE LOS DOCUMENTOS EXTERNOS DE LA RSP . SE ANEXAN ESTAS FORMAS A CONTINUACION ASI COMO EL USO DE LA MISMAS .

GENERACION DE ARCHIVOS. LOS INVESTIGADORES DEL GRUPO DE DESARROLLO DE PROGRAMAS DE ACUERDO AL RPC Y AL ODPC PROGRAMARAN MODULOS Y SISTEMAS, LOS CUALES DEBERAN ADEMAS CONFORMAR LA INDICACIONES DEL PLAN DE PRUEBAS. ES LA RESPONSABILIDAD DEL PROGRAMADOR ENTREGAR MODULOS COMPLETOS A LA BIBLIOTECA DE SOPORTE DE PROGRAMAS, ESTOS MODULOS DEBEN CUMPLIR CON TODAS LAS NORMAS YA PREESTABLECIDAS.

ENCABEZADO DE MANUALES

ESTA FORMA ES LA PRIMERA PAGINA DE CADA MANUAL, Y SE ACTUALIZA DE ACUERDO CON LAS REVISIONES QUE TENGA EL MANUAL/DOCUMENTO/PROGRAMA, DE ACUERDO CON LAS VERSIONES LIBERADAS POR LA RSP. VEASE LA FORMA Y EL EJEMPLO.

FORMA DE TRANSMISION DE SOFTWARE.- LOS PROGRAMAS FUENTE SON ENTREGADOS A LA BIBLIOTECA VIA UNA FORMA DE TRANSMISION DE SOFTWARE. (FTS) SE DEBE TENER UNA FORMA PARA CADA VERSION DE CADA MODULO QUE SE ENTREGADO A LA BIBLIOTECA YA SEAN PRIMERAS VERSIONES O PROGRAMAS CORRECTORES, O ACTUALIZACIONES. SE DEBE HACER REFERENCIA PRECISA AL RPC Y AL ODPC. ESTA FORMA DEBE TENER UN NUMERO DE CONTROL UNICO ASIGNADO POR LA BIBLIOTECA DE SOPORTE DE PROGRAMAS.
DENTRO DE BIBLIOTECA ELTOCA SE DEBE DIFERENCIAR SI LA ENTREGA QUE SE HACE ES UNA PRIMERA VERSION PARA CREAR LA PRIMERA VERSION OFICIAL DE ESE MODULO EN LA BIBLIOTECA O ES UN SISTEMA NUEVO, LO CUAL QUIERE DECIR QUE ES UNA MODIFICACION DE UN SUBSISTEMA YA ENTREGADO Y QUE SE USA PARA CONCEPTO DE "SISTEMA NUEVO". EN EL RENGLON DE RPS SE DEBE ANOTAR SI SE TRATA DE UNA VERSION GENERADA POR UN PROBLEMA EN SOFTWARE O POR UN CAMBIO EN INGENIERIA (SCI).

APLICACION REVISION

TIPO	EQUIPO	DESCRIPCION	FECHA	ELABORADO POR
		CAMBIO POR	APROBADO POR	

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

NOMBRE DEL DOCUMENTO

INDICE REVISIONES	PAGINA																		
	REVISION																		

OBSERVACIONES:

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

REPORTE DE PROBLEMAS EN SOFTWARE

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

FECHA REPORTADO _____ POR _____

FECHA CORREGIDO _____ POR _____

PROGRAMA _____

MODULO O SUBRUTINA _____

PRUEBA EFECTUADA _____

PRIORIDAD DE CORRECCION [ ] CRITICA [ ] IMPORTANTE [ ] POCO IMPORTANTE

DESCRIPCION DEL ERROR

[ ] SE INCLUYE LISTADO

PROGRAMADOR ASIGNADO _____ FECHA _____

REFERENCIA RPC _____ REPORTE PROBLEMA DE SOFTWARE

[ ] ERROR EN DOCUMENTO
[ ] ERROR PRUEBA
[ ] OTRO
[ ] VERSION YA CORREGIDA
[ ] ERROR EN RPS
[ ] MISMO ERROR REPORTE # _____
[ ] PRIORIDAD

OBSERVACIONES

5-2-2

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

RESUMEN REPORTE ERROR EN PROGRAMA

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

REP #	DESCRIPCION	STATUS	OBSERVACIONES

STATUS

P - REPORTE EN PREPARACION

E - REPORTE ENTREGADO AL COMITE DE CAMBIOS A LA CONFIGURACION (CCC)

A - REPORTE APROBADO POR CCC

R - REPORTE RECHAZADO POR CCC

I - REPORTE IMPLEMENTADO

X - REPORTE REFERIDO A OTRA PERSONAL

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

FORMAS DE TRANSMISION DE SOFTWARE

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

NUMERO DE CONTROL DE BIBLIOTECA: _____

RPC _____ DDPC _____

☐ PRIMERA ENTREGA ☐ SISTEMA NUEVO

RPS #'S _____ NOMBRE PROGRAMADOR _____

CAMBIO DE DISEÑO SCI/RPS #'S _____

ARCHIVOS ENTRADA

FUENTE	CAMBIOS

ANEXO

DESCRIPCION DE LA ENTREGA: _____

OTROS ARCHIVOS QUE REQUIEREN CAMBIO: _____

DOCUMENTOS QUE SE AFECTAN: _____

☐ SE ANEXAN PRUEBAS ☐ ARCHIVOS PARA PRUEBAS

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

AVISO CAMBIO ESPECIFICACIONES

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO : _____

RPC _____ DDPC _____ SCI _____

ESTE AVISO INFORMA QUE LAS ESPECIFICACIONES INDETIFICADA POR EL NUMERO (Y DOCUMENTOS DE REVISION) HAN SUFRIDO CAMBIOS, LAS PAGINAS QUE SE HAN ALTERNADO SE ANEXAN

ACE No.	PAGINAS CAMBIADA (ENCERSION/ELIMINAR	S	I	FECHA

S - TANTALIZAD PAGINA ANTERIOR

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

EVALUACION RPC/SCI

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

SCI #

SI	NO	
○	○	ESTE CAMBIO CORRIGIO LA DEFICIENCIA EN EL DISEÑO (VALORES O REQUERIMIENTOS)
○	○	SE AFECTAN INTERFASES O CAMPATIBILIDAD CON EL SISTEMA
○	○	SE AFECTA LA MODULARIDAD, INTERCAMBIABILIDA ETC.
○	○	SE AFECTA LA SEGURIDAD DEL SISTEMA
○	○	SE REQUIERE RETROALIMENTACION
○	○	SE AFECTAN LAS PRUEBAS, OPERACIONES DEL SISTEMA O MANTENIMIENTO DE PROGRAMAS
○	○	SE AFECTAN MANUALES ○ TECNICO ○ USUARIO
○	○	SE AFECTAN LOS PROCEDIMIENTOS DE PRUEBA
○	○	SE AFECTA EL ENTRENAMIENTO DEL PERSONAL

SI CONTESTO SI A CUALQUIER PREGUNTA ANTERIOR, ENTOCES LA SCI ES DE CLASE I

SE AUTORIZA CAMBIO DE INGENIERIA ☐ SI ☐ NO

RIZO CAMBIO: _____ FECHA: _____

OBSERVACIONES.

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

REPORTE CORRECCION DE ERROR

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

PROGRMA: _____

PROGRAMADOR: _____

REFERENCIA RPC: _____

VERSION # : _____

DESCRIPCION ACCION CORRECTIVA

LISTA CAMBIOS EN CODIGO

DIRECCION DEL CAMBIO	CODIGO PRESENTE	CODIGO CORREGIDO

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

MATRIZ CORRESPONDECIA PRUEBA

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

METODO DE VERIFICACION
NA - NO APLICABLE
I - INSPECCION
A - ANALISIS
D - DEMOSTRACION
P - PRUEBA

TIPO DE PRUEBA
NIVEL 1 - PUEBLA COMPONENTE
NIVEL 2 - PRUEBA RPC
NIVEL 3 - PRUEBA SISTEMA/SUBSISTEMA

No. DE PARRAFO REQUERIDO	METODO DE VERIFICACION					NIVEL PRUEBA			VERIFICACION DEL No. DE PARRAFO REQUERIDO
	NA	I	A	D	P	1	2	3	

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

FORMA DE TRANSMISION DE DATOS

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

No. DE CONTROL DE BIBLIOTECA: _____ FECHA PROCESADO: _____

IDENTIFICADOR: _____ REFERENCIA DDPC: _____

☐ DEFINICION DE DATOS ☐ DEFINICION ENSAMBLADOR ☐ DATOS PRUEBA

PROGRAMADOR: _____ APROBRADO: _____

RAZON DE LA FORMA

☐ ENTREGA NUEVA ☐ SISTEMA NUEVO

REPORTE DE PROBLEMA DE SOFTWARE: _____

SOLICITUD DE CAMBIO DE INGENIERIA _____

ARCHIVO ENTRADA CAMBIO

_____ _____

_____ _____

DESCRIPCION DE LA ENTREGA: _____

.RA USO DE LA BIBLIOTECA:

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

REPORTE DE ERROR EN PROGRAMA

TITULO DEL PROYECYO. FECHA.

NUMERO DE PROYECTO. PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

PROGRAMA: _____

REFERENCIA PAG. # _____ RPC: _____

OTRA REFERENCIA: _____

REPORTADO POR: _____

DESCRIPCION DEL PROBLEMA.:

ERROR DE OPERADOR
ERROR DE COMPUTO
ERROR DOCUMENTO _____
ERROR PROGRAMADO
ERROR INFORMACION

ERROR DATOS E/S
PROBLEMA DE EQUIPO

SISTEMA DE SOPORTE
LIBRERIAS
OTROS

OBSERVACIONES:

MODULO DE PROGRAMA	FECHA CARGADO	CINTA O DISCO
_____	_____	_____

PROGRAMADO POR: _____

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

CATALOGO PROVISIONAL DE PROGRAMAS EXISTENTES

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: AUTORIZO:

NOMBRE DEL PROGRAMA:

PROGRAMADOR (ES)

FECHA ULTIMA VERSION

UBICACION

COMPUTADORA: MEDIO:

NOMBRE DEL ARCHIVO: OTRO IDENTIFICADOR:

Última versión efectuada por:

OBJETIVO DEL PROGRAMA:

BREVE DESCRIPCION DEL PROGRAMA.(Algoritmo) Utilice otras hojas si se requiere

FORMA DE CORRERLO.(INCLUIR DATOS DE ENTRADA QUE REQUIERE EL PROGRAMA y datos importantes). Utilice otras hojas si requiere.

5-2-12

☐ SE ANEXA EJEMPLO ☐ SE ANEXA LISTADO ☐ SE ANEXA [illegible]

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

REQUERIMIENTO SALIDA DE RESULTADOS

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: AUTORIZO:

NOMBRE DEL REPORTE	DONDE SE GENERA FUENTE	DONDE SE RECIBE DESTINO	No. DE COPIAS	LINEAS LONGITUD MAGNITUD	MEDIO	FRECUENCIA	O B S E R V A C I O N E S

5-2-13

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

REQUERIMIENTOS DE DATOS

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: AUTORIZO:

NOMBRE DEL ARCHIVO (O VARIANTE)	ENTRADA			FRECUENCIA USO
	No. RECORDS	TAMAÑO RECORD	MEDIO	

S-2-14

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

SOLICITUD DE PRODUCTO

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

SISTEMA QUE SOLICITA: _____

SISTEMA AL QUE SE LE SOLICITA: _____

PERSONA QUE SOLICITA: _____ AUTORIZO: _____

PERSONA ASIGNADA: _____

FECHA QUE DEBE ENTREGARSE: _____

DESCRIPCION DEL PRODUCTO QUE SE SOLICITA

REFERENCIA RPC: _____ DDPC: _____

MOTIVO DE LA SOLICITUD

RESULTADOS QUE DEBAN OBTENERSE (FORMATOS ER)

PRUEBAS QUE DEBERA CUMPLIR

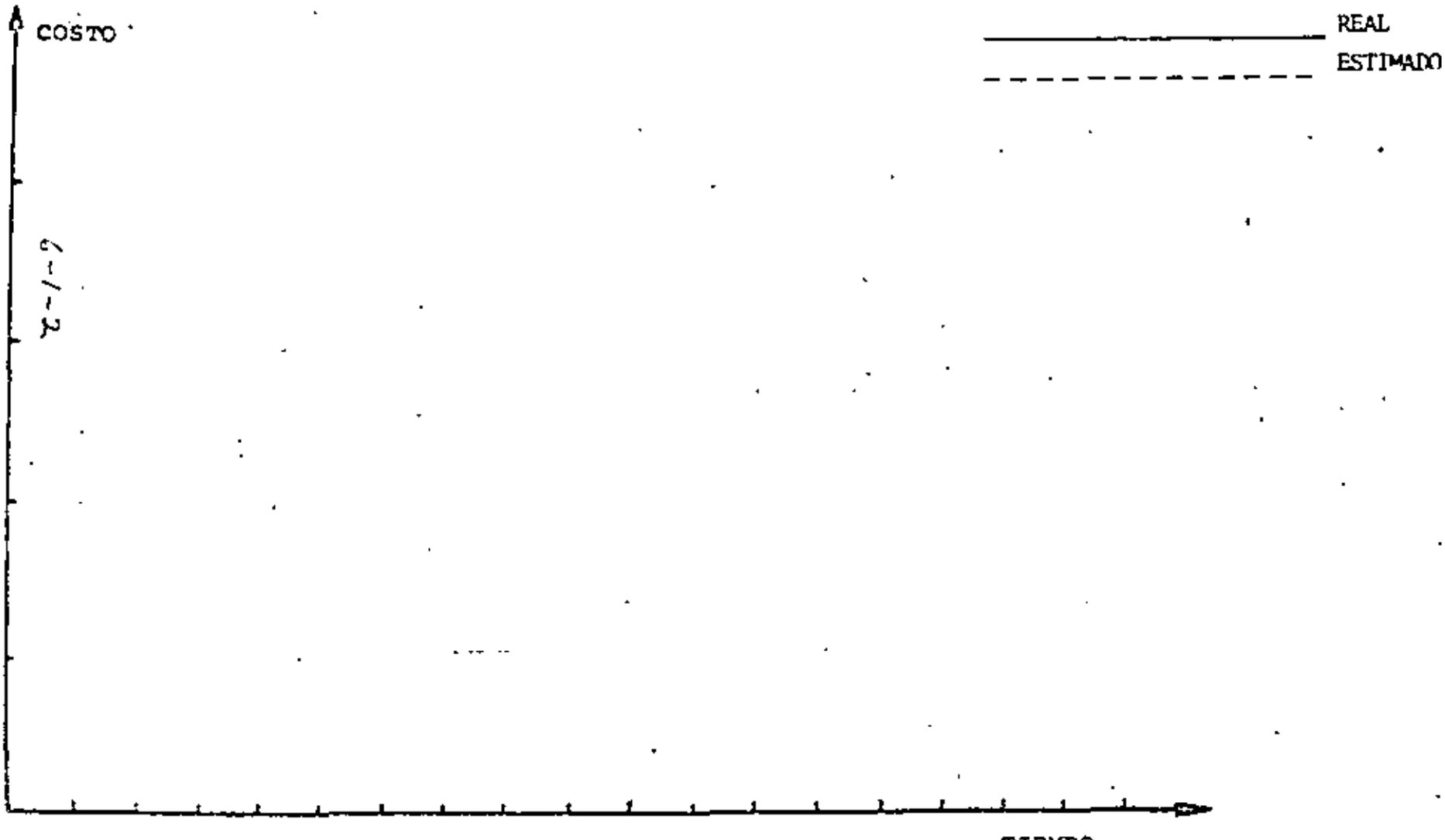
FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

REPORTE GLOBAL DE PROGRESO (COSTO)

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO : AUTORIZO:



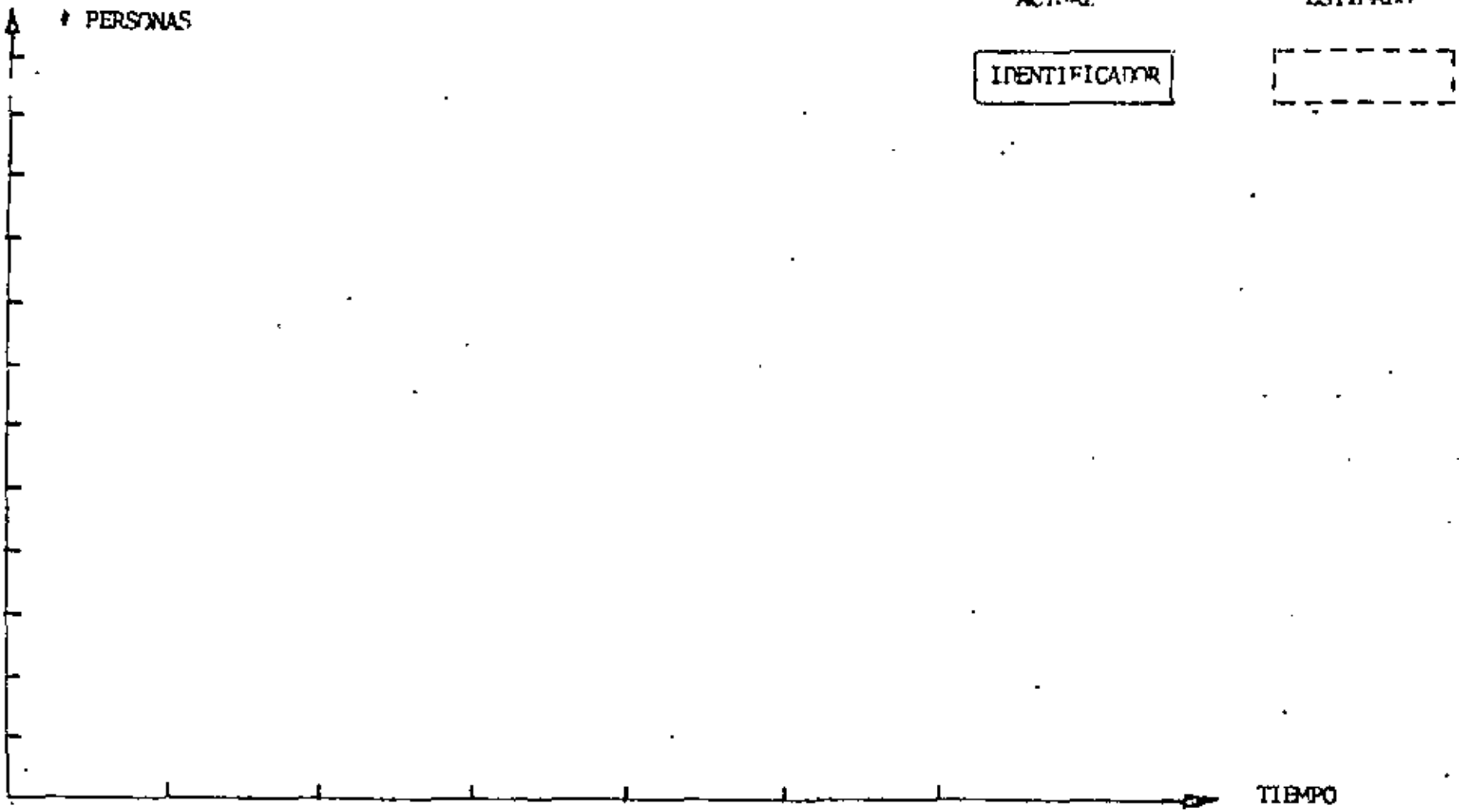
FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

CONTROL DE PERSONAL

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____



6-1-3

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: AUTORIZO:

FECHA DE LA REUNION	COMPROMISOS CONTRAIDOS	FECHA DE REALIZACION	COMPROMISOS CUMPLIDOS			RESPONSABLE
			SI	NO	FECHA	

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

REPORTE DE ACTIVIDADES (PERSONAL/SEMANAL)

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

DESCRIPCION DE LAS ACTIVIDADES DESARROLLADAS:

RESULTADOS OBTENIDOS:

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

PROGRAMA DE ACTIVIDADES (PERSONAL/SEMANAL).

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

HORAS ESTIMADAS MAX: MIN:

EQUIPO O INFORMACION NECESARIA

DESCRIPCION DE LAS ACTIVIDADES A REALIZAR EN LA SEMANA: _____

RESULTADOS QUE SE DEBEN OBTENER:

ESTIMACION DE RECURSOS 6-1-7

DIAS CALENDARIOS _____

DIAS PERSONA _____

HORAS PERSONA _____ PERFIL _____

COSTO DIRECTO _____ $ _____

_____ $ _____

_____ $ _____

RECURSOS CRITICOS _____

_____

_____

ACTIVIDADES CRITICAS _____

_____

_____

6-1-8

FUNDACION ARTURO ROSENBLUETH
Para el Avance de la Ciencia, A.C.

TARJETA PARA EL CONTROL DE ACTIVIDADES

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA SUBPROGRAMA

ELABORO: _____ AUTORIZO: _____

PREDECESOR DEFINICION

\ DE

\ DE

\ DE

SUCESOR DEFINICION

\

\

\

FUNDACI) ROSENBLUETH
Para el [illegible] la Ciencia, A.C.

DIAGRAMA DE BARRAS

TITULO DEL PROYECTO: FECHA

NUMERO DE PROYECTO: PROGRAMA: SUBPROGRAMA:

ACTIVIDADES PRINCIPALES	198												198												ANTECEDENTES
	ENE.	FEB.	MAR.	ABR.	MAY.	JUN.	JUL.	AGO.	SEP.	OCT.	NOV.	DIC.	ENE.	FEB.	MAR.	ABR.	MAY.	JUN.	JUL.	AGO.	SEP.	OCT.	NOV.	DIC.	
1																									
2																									
3																									
4																									
5																									
6																									
7																									
8																									
9																									
10																									
11																									
12																									
13																									

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

ADMINISTRACION DE PROYECTOS DE SOFTWARE

M. EN C. MARCIAL PORTILLA ROBERTSON

FEBRERO, 1982

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

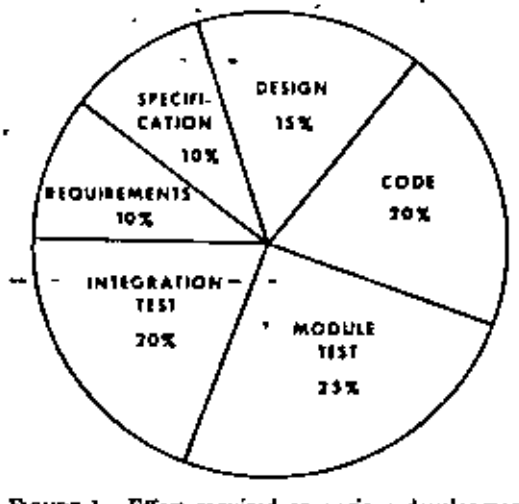


FIGURE 1. Effort required on various development activities (excluding maintenance).

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* [FIPE77]) 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.

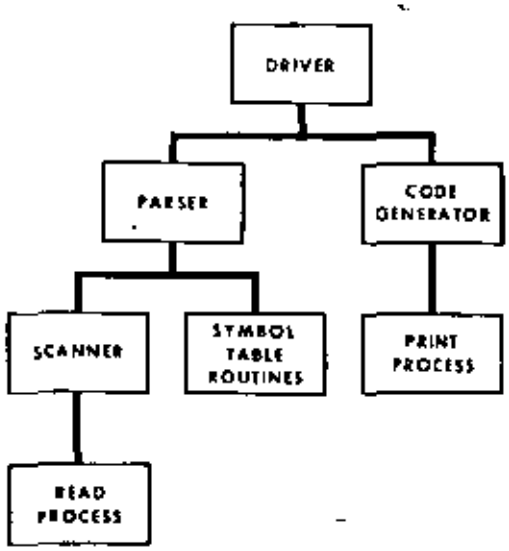


FIGURE 2. Sample baseline diagram for a compiler.

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

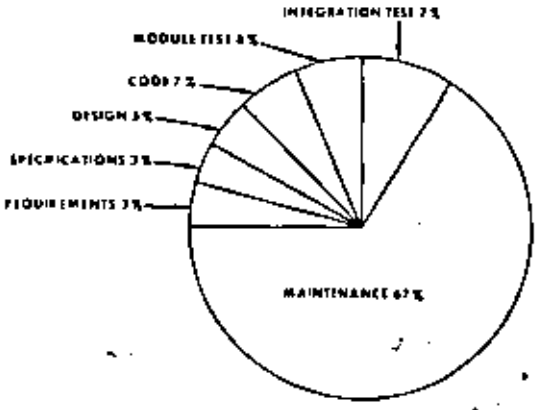


FIGURE 3. True effort on many large-scale software systems.

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).

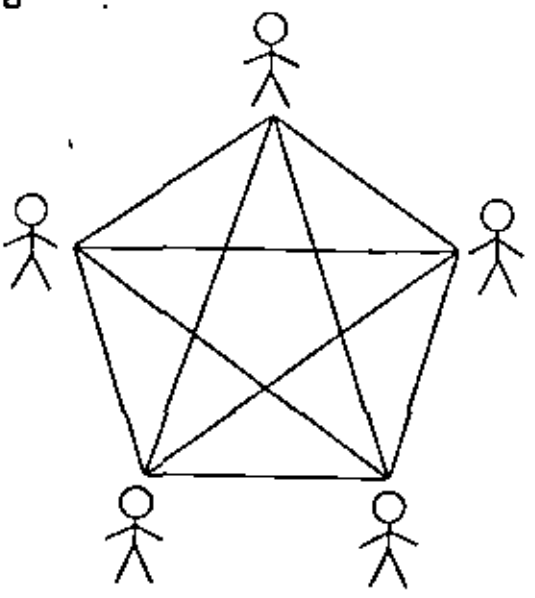


FIGURE 4(b). Five-member group: 4,000 lines per year = 40,000 lines in two years (ten communication pairs).

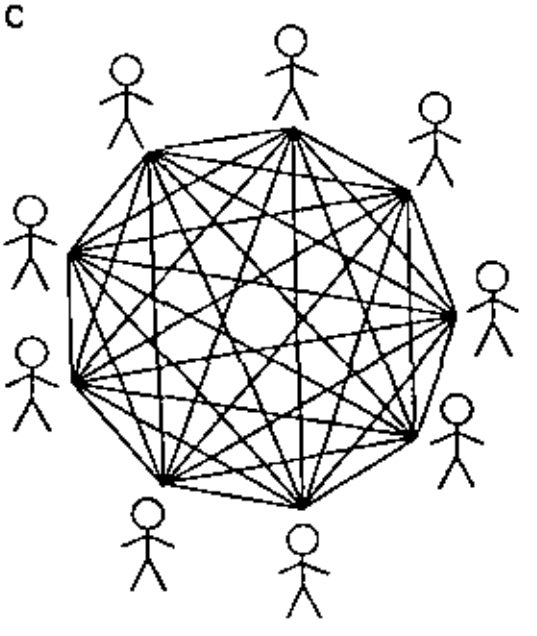


FIGURE 4(c). Nine-member team: 3,000 lines per year = 50,000 lines in two years (36 communication pairs).

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

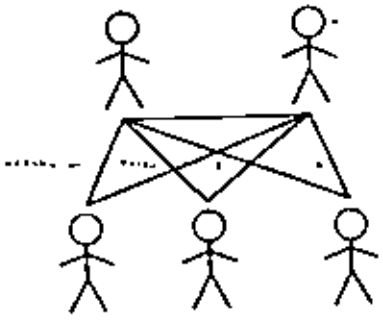


FIGURE 5. Fewer communications paths in a chief programmer team.

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.

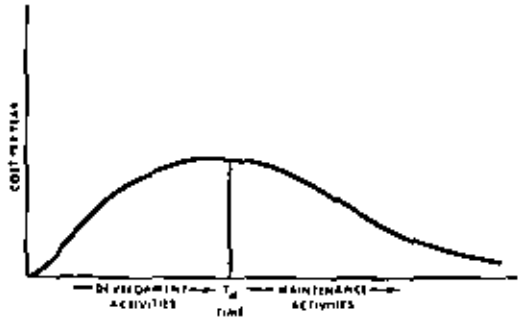


FIGURE 6. Yearly rate of expenditures approximates the Rayleigh curve. Total cost (area under curve) = K; $a = 1/T_d^2$; rate = $2Kate^{-at^2}$.

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 and

debugging problems are apparent to every programmer; these tools are the oldest and most advanced. Techniques for improving coding were developed next. The most recent developments have related to requirements and specifications. Although many technical problems have not been solved, an effective methodology is emerging. Some of these techniques are presented in the following subsections.

Verification and Validation

Verification and validation (module and integration testing) of a system occupy about half of the development time of a project. Many debugging aids have been developed to facilitate this effort; most are implemented as programs to test some feature of a system.

Automated Tools

The earliest and most primitive debugging tools were the dump and the trace. A *dump* is a listing of the contents of the machine's memory. This listing can often reveal unintelligible data or errors. Unfortunately, a dump may not be taken until long "after the fact" and the cause of the error may not then be apparent. A *trace* is a printout showing the values of selected variables after each statement is executed. It may help a programmer to discover errors.

These techniques are not usually very effective because they supply much data with little or no interpretation. More advanced methods are needed to reduce this data to an intelligible form.

Flowgraph analyzers are capable of detecting references to variables which are never initialized or never reused after receiving a value; these usually indicate errors. Test data generators are also available. Assertion checkers validate that given conditions are true at indicated points of a program. Automatic verification systems have been implemented for small languages [KING69] and symbolic execution has been proposed as a practical means for validating programs in a more complex language. The 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 [DURH76].

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' 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_i 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_i and A_j 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_2 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/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 [RUBY76]. 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 one should not be led to infer that actual program construction 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, *revised presentation* of a program serves admirably well to keep the individual building blocks sufficiently 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

$$f(x) = \text{if } p(x) \text{ then } g(x) \text{ else } h(x)$$

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 those 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 wave-form 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 $O(N)$ [ALLI72]. 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., x53-1976, 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/I," *Commun. ACM* 16, 3 (March 1973), 169–179.

[COOL65] COOLEY, J. W.; AND TUKEY, J. W. "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19, 90 (1965), 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 MASLEN, 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.

[LOWE69] LOWRY, E. S.; AND MEDLOCK, C. W. "Object code optimization," *Commun. ACM* **12**, 1 (Jan. 1969), 13–22.

[MILL70] 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; *IACM 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. 253–258.

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!"

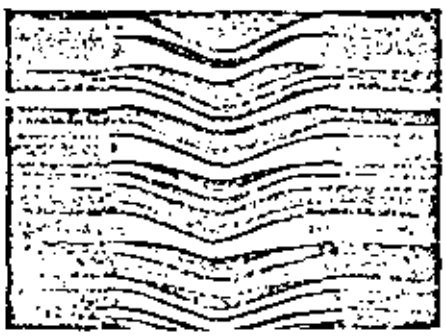
For structured programming to have the proper opportunity to show its strengths and not be rejected outright by an organization, consideration should be given to these questions:

1. What will the specific objections to it? Are there any potential disadvantages that may be experienced as a result of using the techniques?
2. Which of the techniques should be attempted first, assuming that you cannot use them all? For example, should you attempt to use structured programming first, and then try top-down design on a later project?
3. What kind of programming project should you begin with as an experiment to demonstrate the benefits of the structured programming techniques?
4. How should you evaluate the success of the experiment?

Common objections

It would be unrealistic to assume that structured programming, top-down design, chief programmer teams, structured walkthroughs (one programmer explains his code to others), program librarians, and structured design would be accepted without argument in any organization. Here are some of the more common objections:

1. Many managers and programmers point out that the structured programming techniques are primarily intended for new development projects; for maintenance of existing unstructured programs, they seem to be of limited use (though the librarian concept and the structured walkthrough concept would still be quite useful). This point is basically valid, though it is usually possible to add new



sections of code in a top-down structured manner, e.g., when completely new features are being added to a system at the request of a user.

This problem may be solved eventually with the aid of "structuring engines" that will automatically convert unstructured logic into structured form; while such an "engine" cannot magically transform "bad" code into "good" code, it will at least foster some standardization.

2. Some managers point out that their programming projects are typically too small to require a team of programmers; therefore, they argue, they don't need any of the new "programming productivity" techniques. Since most managers apparently are not prepared to fire most of their mediocre programmers and replace them with one highly competent chief programmer, we must accept this objection as a fairly valid one—but only for the chief programmer team concept.

There is no reason why the existing programmers in the organization, even working by themselves, could not use structured programming and top-down design.

3. Still other managers object to the cost of training their programmers in the techniques of structured programming. This training exercise is admittedly nontrivial, though it depends on the programmer's experience. (Junior programmers learn the techniques more easily than senior programmers; the author's group trained 120 programmers in an Australian government agency prior to beginning our payroll project, and of the 20 who scored at the top of the class, eight were novice programmers with less than six months experience.)

Ease of training also depends on the programming language; the techniques are generally easiest in PL/1, reasonably easy in COBOL, and more difficult in FORTRAN and assembly language. In general, we found that programmers require three to five days of classroom training to learn the techniques, and approximately one month of programming (when they are at least as productive as they were previously) to become comfortable with the new techniques. The investment in training is thus relatively small compared to the benefits that were discussed in the preceding section.

4. The manager often has to overcome technical objections raised by the programmers; these objections most frequently come from senior programmers, many of whom are now project leaders, who still fondly recall the "good old days" of the 1401 and the 650. The common programmer-oriented objections are: it is awkward and inconvenient at first; the programming language is inadequate for the strict discipline imposed by structured programming; it is not obvious that the new approach will actually reap the benefits discussed above; and finally, there is a concern that the top-down structured approach will lead to tremendously inefficient programs.

The awkwardness and inconvenience is largely a matter of training. The question of language adequacy can be a relevant one, and one answer might be to convince the programmers (and their managers!) to begin using PL/1 and the other ALGOL-like languages in preference to the primitive COBOL-like and FORTRAN-like languages.

The objection about efficiency can only be answered by appealing to the programmer's common sense; the battle for efficiency is generally won or lost at the system or program design level (e.g., by making sure the system is running on an efficient hardware configuration, and that it isn't doing things it wasn't intended to do), and not at the bit-fiddling level (i.e., where the programmer "wastes" a microsecond or two indulging in a subroutine call).

Obviously, there are some exceptions to this, in some real-time systems, for example; but as Professor Bill Wulf of Carnegie-Mellon Univ. points out, "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity."

5. In addition to programmer retraining costs, many managers object to the cost of developing new standards to conform to the new programming techniques. Some organizations may have only recently finished developing standards for testing—and they tend to be "bottom-up" standards, in direct contrast to the top-down methodology currently being advocated. While the cost of developing new standards may well be substantial, it is difficult to think of any way of avoiding it. Indeed, even if structured programming had not appeared on the scene, surely some new techniques would eventually appear, forcing the voluminous standards manuals to be rewritten . . . it seems to be an inevitable fact of life.

This may be an academic point, but I know of one large bank and one insurance company that have apparently rejected structured programming for this reason alone—all of which seems rather sad.

Some managers have expressed the fear that the structured programming concepts might not work (or that their programmers might not be sufficiently familiar with the new techniques) on a significant project whose failure would have disastrous consequences. There is a very simple solution to this problem: if you have not tried structured programming before, you probably should not use a critical project as a "guinea pig."

I have seen three projects in early 1975 fail in their attempt to use various aspects of structured programming. One midwestern company used top-down testing at the program level, but then integrated the programs in a bottom-up fashion to build a system—and they couldn't understand why the "top-down" approach had failed to give them miraculous results.

Another company in New England made an unsuccessful attempt at using the program librarian concept, but it appears that the librarian was a secretary who was required to spend six hours a day typing envelopes and the other two hours supporting the program.

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/I 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/I 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 teeshirts 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 on 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 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 [illegible] 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?

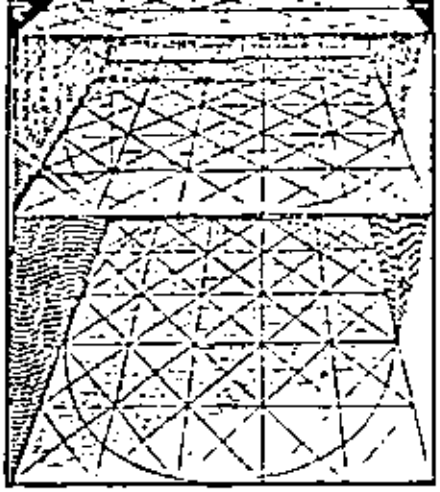
I find that a reviewing audience can easily spot five to six bugs in a 200 statement program within 15 minutes. It is important to recognize that the programmer who wrote the code was usually convinced that his code was correct, and his test data would usually be a self-serving attempt to confirm that 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-435. 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 DL/I.
6. Machine—IBM 370/145, with TSO.
7. Data base environment.
8. Programmer coding productivity @ 200 hours per month—that is, approximately 11 lines *per hour*.
9. Number of programs and modules—50
10. Median module size — 5.4 pages.
11. Type of application—financial.
12. Technical complexity—mild.
13. Programming logic complexity —moderate.
14. Design considerations—structured programming, structured design.

A group of programmers was organized into a "chief programmer team." The chief programmer was responsible for the design of all programs. He did the actual coding of some critical programs, and organized and coordinated the programming of the other programs in the system. Two programmers were on the team full time and five other programmers contributed to the team effort. Programs were designed adhering to the principles of structured design. Principally, programs were kept small and designed "functionally." Large programs were divided into modules. The coding of programs was structured. No GOTO statements were allowed, and other conventions of structured programming were followed.

The overriding constraint on the system development was time. Because of contractual obligations program design, programming, debugging, and implementation had to be completed in four months. The deadline was not extendable.

The structuring of the data base reflected a strong user orientation and was not designed for easy program manipulation. At the outset of the project, only two programmers had extensive data base experience. In preparation for future on-line considerations, the system was to be updated by transactions. Reports were generated directly from the data base. In terms of programming logic complexity, the system was straightforward except for several internal relationships. Technically, the complexity of the system was not beyond the ability of the chief programmer team.

Design goals

The design goals of the system were greatly influenced by these program development priorities:

1. Speed of development
2. Program and algorithmic accuracy
3. Maintainability of programs
4. System flexibility

STRUCTURED DESIGN

The traditional consideration of production run efficiency was only of inc[illegible]tal importance. It was not ig[illegible]nc[illegible] simply gave way to the higher pr[illegible]s. 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 de[illegible]s[illegible] for general usage, so that it w[illegible]oe 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 a[illegible]ng to the contents of the transaction.

Another way to view an updating program is from the traditional actions of adding, deleting, or replacing data in the data base. In this case, as a transaction enters the program, it is categorized into one of the three major functions, and is handled along those functional lines.

Program flexibility and modularization

The flexibility of the programs in the system was difficult to assess. However two characteristics of the system point to the fact that structured design produces some flexibility. One is that changes in it were made with a minimum of effort. The largest change required four days, and the next largest, less than a day. The second characteristic is the fact that errors were quickly located and corrected. It appeared that the major factor contributing to system flexibility was the isolation by function that was achieved by structured design. When a problem occurs, it is easy to eliminate many modules from consideration since their functions may have nothing to do with the problem.

Data coupling was extensively used in the interfacing of modules with each other. (Data coupling occurs when all input and output to and from the called module are passed as parameters or arguments—i.e., as data elements). The independence gained here enhanced both the reusability of a module and the isolation of a module by function from other modules.

In other ways, the modular approach accelerated the development of the system. The physical separation of modules meant that more than one person could simultaneously contribute to a program. In fact at one time, five programmers were actively working on modules of the same program.

A byproduct of using a modular approach was the lessening of the total learning time involved in the translation of programming specifications into code. This occurred because, as a programmer completed a module, he was assigned another similar one. He still had the previous coding fresh in mind and could therefore quickly grasp the new programming specifications.

Another feature of adopting a modular approach occurred when a programmer completed a module and was then free to work on other systems. In a conventional system, a programmer is tied to a program until he finishes it; and if a problem in his area of expertise arises while writing the program, something must suffer. However, if a program is broken into small modules, many breaking points result naturally, and the problems involved in conventional methods of programming are minimized.

Despite advantages in using a modular approach, there were also some disadvantages. One problem arose in the interfacing of modules. In spite of careful attention given to the flow of data to and from other modules, there still were some errors. The number and order of parameters, their characteristics, and the interpretation of their values were in some cases misunderstood. Another problem arose in the organization necessary for the direction of several people working on the same problem simultaneously. Modules were being developed so rapidly that coordination of testing, linking, and integrating them became a large task.

Structured walkthrough

One of the techniques that led to the quick completion of the project was that of the structured walkthrough. A walkthrough, or mental execution of the program by the programming team, was done for each program.

After a programmer had compiled a program and scrutinized it for obvious errors, he sent the chief programmer and several other programmers involved copies of his source text. After time was allowed for the group to examine the text (usually a few hours), the team met and collectively performed a mental execution of the program. A list of errors was made as each logical path was followed. The interface with modules either calling or called by the module being examined was carefully checked as well. At the end of the walkthough, the list of errors was given to the author.

In this manner most errors were caught before any testing had occurred. Also, the team members who reviewed the text became familiar with a part of the system other than their own. This helped to establish a common base of understanding of all components of the system.

Coincidentally, while the team reviewed the interface with other modules, errors (usually from miscommunication) were also spotted in other modules. It was also not uncommon to have a program or module execute correctly the first time it was tested. One of the major factors in the success of the system was the shortness of the time spent in testing, and the major contributor to it was structured walkthroughs.

Program testing

One reason why testing and debugging went smoothly was the fact that the design of the system included programming specifications for debugging. Not only did programming specifications define the function of a program or module, but they also required variables to be inserted and manipulated solely for the purpose of debugging.

The most effective technique was using an activity variable in each

module. Initially the activity variable is set up as inactive. As a module was invoked, the value of the activity variable was changed to indicate that it was active. As control was relinquished, the variable was returned to the inactive state. When a dump occurred, it was easy to trace the path of active modules, arriving at the final

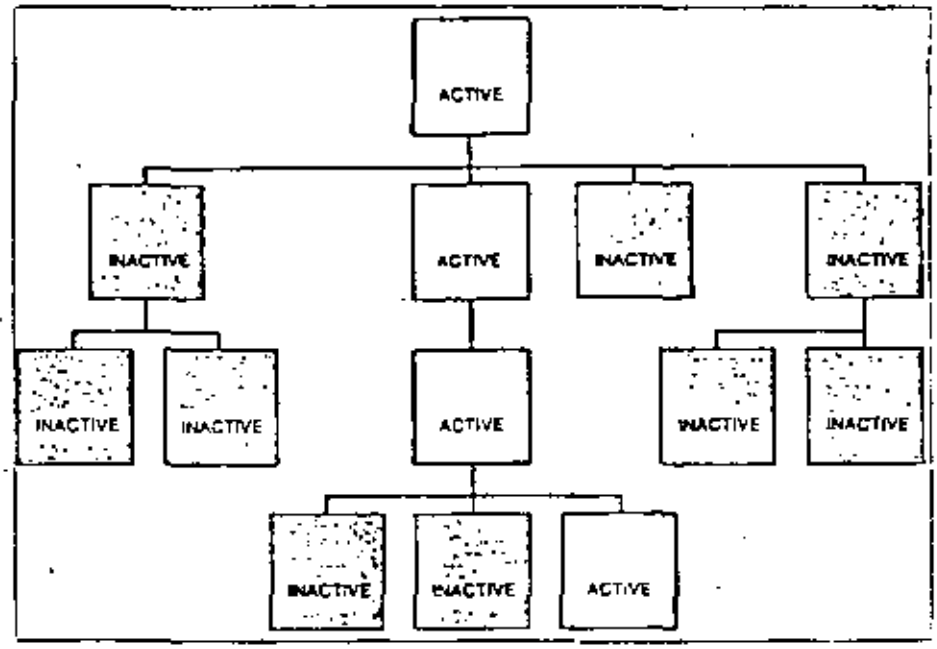


Fig. 1. By using an "activity" variable in each module, the flow of control through the program can be reconstructed and non-active modules ignored in debugging.

one in which activity had occurred (see Fig. 1).

Another useful technique was to display the parameters of a module at the time of invocation. Because of data coupling, the complete set of relevant data was available for determining the state of the module.

An additional technique that proved to be useful was the gathering of statistics internally by each module. Typically, a module would keep track of how many times it had been invoked, the parameters it had been called with, what type of internal results it had generated, etc. Data of this type made reconstruction and analysis of program activity an easy task. They also helped to identify areas of code that had never been tested. Also, it helped to locate areas of code that were critical in the efficient running of the program.

Summary

The most pleasing and surprising aspect of the project was the smoothness with which debugging and testing were accomplished. This smoothness is reflected by the high programmer productivity rate of approximately 11 lines per hour. We felt that several factors contributed to this success—structured walkthroughs, the breakdown of large programs into modules, and structured programming.

The major criterion, that of speed of development, was certainly enhanced by structured design. A modular programming approach alone would not have made it possible to write the large programs needed in the amount of time allotted.

The other design goals—flexibility and maintainability—cannot yet be effectively evaluated since the system has remained relatively stable in its short lifetime. The few changes made do point to a high degree of maintainability. Structured design, used in conjunction with the complementary techniques of structured programming and walkthroughs, did make possible a level of speed and accuracy in system development not previously attainable. Based upon our experience with developing this system, structured design is as powerful in practice as has been predicted in theory. ❁

Mr. Inmon is a chief programmer for GTE Sylvania Electronics Systems Group in Mountain View, Calif. He received an M.S. in computer science from New Mexico State Univ.

ADMINISTRACION DE PROYECTOS DE SOFTWARE

ANEXO

SISTEMA DE BANCO DE SANGRE

FEBRERO DE 1982

I N D I C E

INTRODUCCION

INTRODUCCION

La finalidad del presente documento de análisis es la de proporcionar la visión conceptual del

SISTEMA DE BANCO DE SANGRE

En él se pretende controlar de una manera efectiva todos los problemas relacionados con el manejo de información en un banco de sangre. Entre estos podemos nombrar aquellos de distribución de sangre, manejo de los donadores activos, resultado de las campañas de donación.

Nuestro principal objetivo será entonces llevar el mejor control posible de todos los factores que afectan a un banco de sangre lográndose así un mejor servicio a la sociedad en algo tan importante como es la capacidad de poder salvar la vida de un ser humano.

A continuación se presentan 3 secciones, las cuales informan de la filosofía que se ha seguido en el desarrollo del sistema.

1.- BASE DE DATOS. DIAGRAMA DE LOS PRINCIPALES PROCESOS DEL BANCO DE SANGRE.

COMENTARIOS

Este sistema está enfocado para instituciones como el ISSSTE, IMSS, SSA, que deben llevar un estricto control sobre sus clínicas y hospitales.

El sistema es idealista, ya que no se maneja información acerca de la calidad de la sangre, como número de plaquetas o leucocitos. No se incluye este tipo de información ya que no se conoce realmente las necesidades de una banco de sangre; todo lo expuesto a continuación son suposiciones de la información mínima que se debería manejar.

BASE DE DATOS

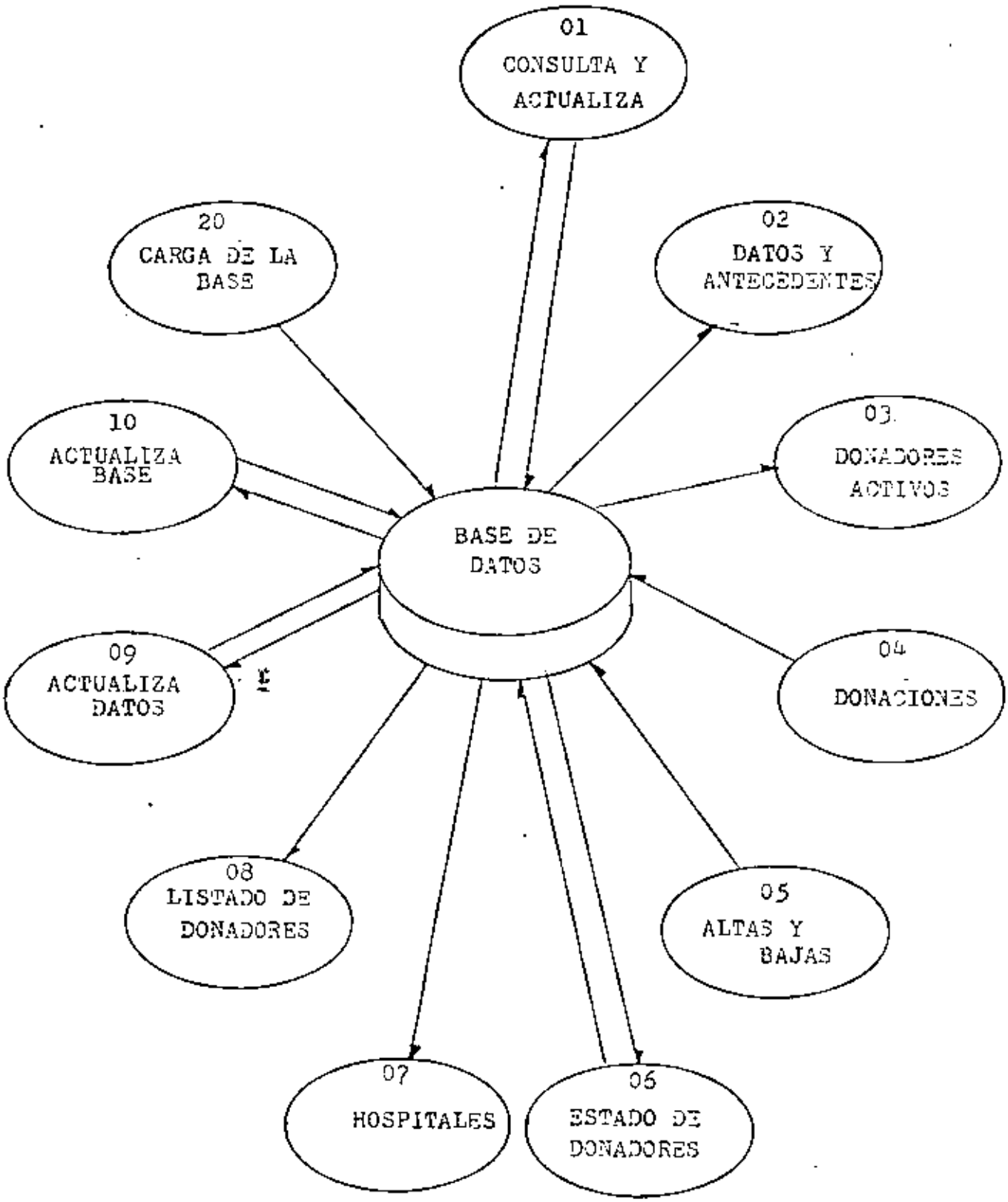


DIAGRAMA DE PROCESOS (PRIMERA PARTE)

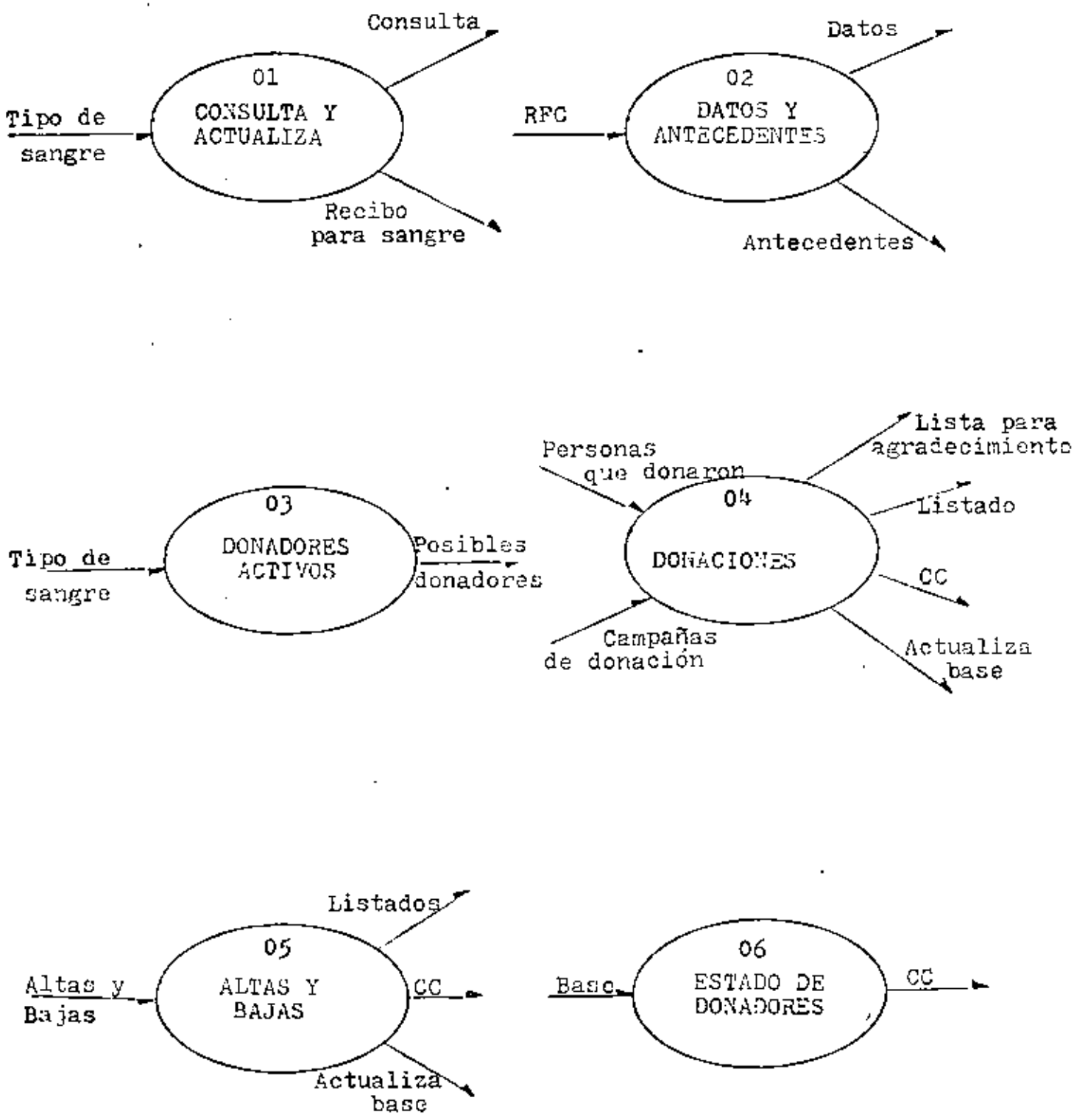
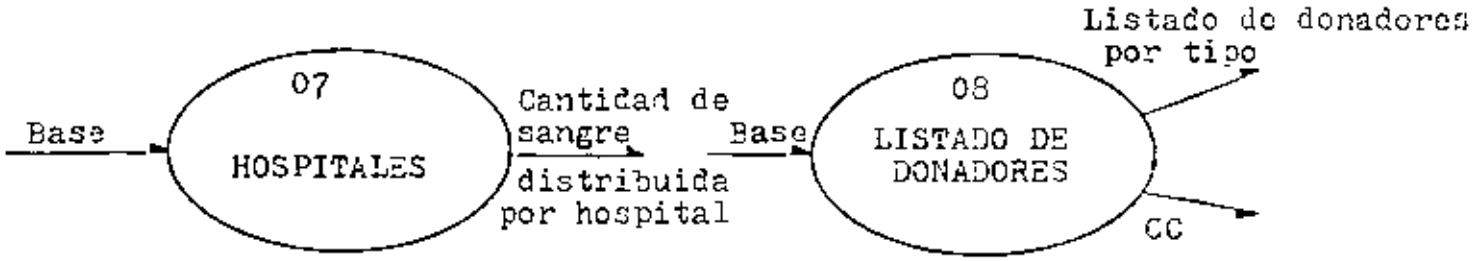
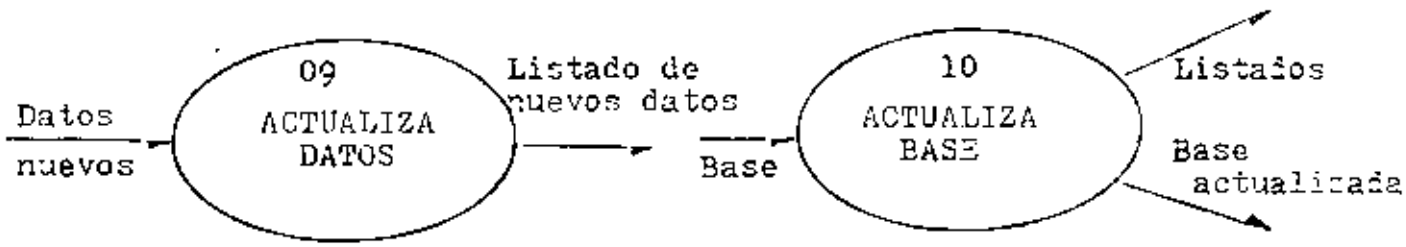
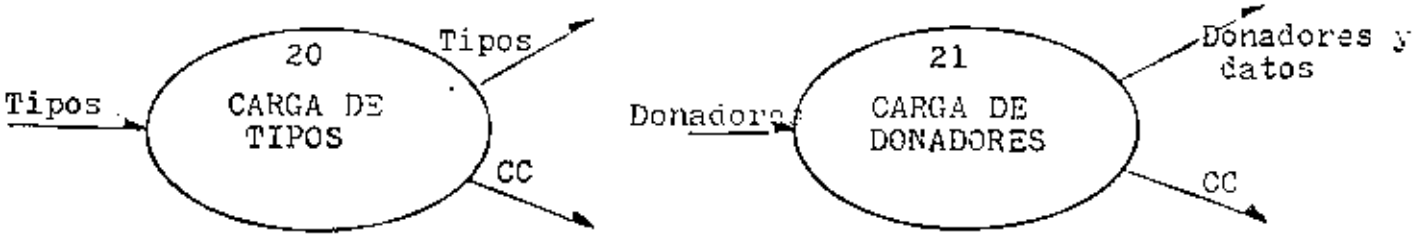


DIAGRAMA DE PROCESOS (SEGUNDA PARTE)







2.- MOVIMIENTOS DEL BANCO DE SANGRE. REFERENCIA CRUZADA DE PROCESOS Y ARCHIVOS DEL SISTEMA. DESARROLLO DE LOS MISMOS.

PROCESO: 01 CONSULTA Y ACTUALIZA.

ENTRADA: Tipo de sangre.

PROCESO: Da consulta o modifica la cantidad existente.

SALIDA: Consulta de: a)Cantidad existente; b)Instituciones; c)Donadores.

Genera: Recibo para recoger del almacén la cantidad de sangre autorizada.

DESTINO: Administrativo ó persona que lo solicite.

FRECUENCIA: Cuando se requiera.

OBSERVACIONES: Por pantalla.

PROCESO: 02 DATOS Y ANTECEDENTES.

ENTRADA: RFC.

PROCESO: Dar los datos y antecedentes de un donador.

SALIDA: Datos como dirección, sexo, teléfono etc.

Antecedentes como la fecha de la última donación, cantidad donada, peso etc.

DESTINO: Doctor o enfermera que va a llevar a cabo la extracción de sangre.

FRECUENCIA: Cuando se lleve a cabo una donación.

OBSERVACIONES: Por pantalla.

PROCESO: 03 DONADORES ACTIVOS

ENTRADA: Tipo de sangre.

PROCESO: Generar un listado de donadores activos.

SALIDA: Listado de posibles donadores.

DESTINO: Administrativo.

FRECUENCIA: Mensualmente o cuando la cantidad límite llegue a su mínimo.

OBSERVACIONES: Límite mínimo el 40% del promedio.

PROCESO: 04 DONACIONES

ENTRADA: Archivo de personas que donaron. Resultado de campañas de donación.

PROCESO: Añadirá las cantidades donadas a la base.

SALIDA: Listado de personas que donaron con sus cifras de control. Cifras de control sobre resultados de las campañas de donación.

Listado para cartas de agradecimiento.

DESTINO: Administrativo.

FRECUENCIA: Diario.

PROCESO: 05 ALTAS Y BAJAS:

ENTRADA: Archivo de altas y bajas.

PROCESO: Maneja altas y bajas de los donadores en la base.

SALIDA: Listados de altas y bajas. Cifras de control.

DESTINO: Administrativo.

FRECUENCIA: Diario.

PROCESO: 06 ESTADO DE DONADORES.

ENTRADA: Base.

PROCESO: Cambiará el status de los donadores.

SALIDA: Cifras de control.

DESTINO: Administrativo.

FRECUENCIA: Cuando el listado de posibles donadores sea menor que el 25% del promedio total de donadores.

OBSERVACIONES: No se considerarán como activos a aquellas personas que hayan donado en los últimos 45 días.

PROCESO: 07 HOSPITALES.

ENTRADA: Base.

PROCESO: Genera informa de cantidad distribuída por hospital.

SALIDA: Listado.

DESTINO: Administrativo.

FRECUENCIA: Mensual.

PROCESO: 08 LISTADO DE DONADORES.

ENTRADA: Base.

PROCESO: Listado general de donadores.

SALIDA: Listado de todos los donadores (activos y pasivos) por tipo. Incluye fecha última de donación.

Cifras de control.

DESTINO: Administrativo y proceso 05.

FRECUENCIA: Mensual.

OBSERVACIONES: Incluye a todos los donadores aunque hayan donado el día anterior.

Donadores con más de seis meses sin donar serán dados de baja.

PROCESO: 09 ACTUALIZA DATOS.

ENTRADA: Datos nuevos.

PROCESO: Actualiza los datos de la base.

SALIDA: Listado de datos viejos y nuevos.

DESTINO: Administrativo.

FRECUENCIA: Diaria.

OBSERVACIONES: Por pantalla.

PROCESO: 10 ACTUALIZA BASE.

ENTRADA: Base.

PROCESO: Depuración de la base.

SALIDA: Listado general para hacer un archivo histórico.

Base actualizada.

DESTINO: Administrativo.

FRECUENCIA: Anual.

OBSERVACIONES: Toda la información de más de un año de antiguedad será guardada en cinta.

PROCESO: 20 CARGA DE TIPOS.

ENTRADA: Archivo de tipos.

PROCESO: Carga de tipos de sangre.

SALIDA: Listado de tipos de sangre cargados.

Cifras de control.

DESTINO: Jefe del proyecto.

Administrativo.

FRECUENCIA: Unica.

PROCESO: 21 CRAGA DE DONADORES.

ENTRADA: Archivo de donadores.

PROCESO: Carga de donadores.

SALIDA: Listado de donadores con sus datos personales cargados.

Cifras de control.

DESTINO: Jefe del proyecto.

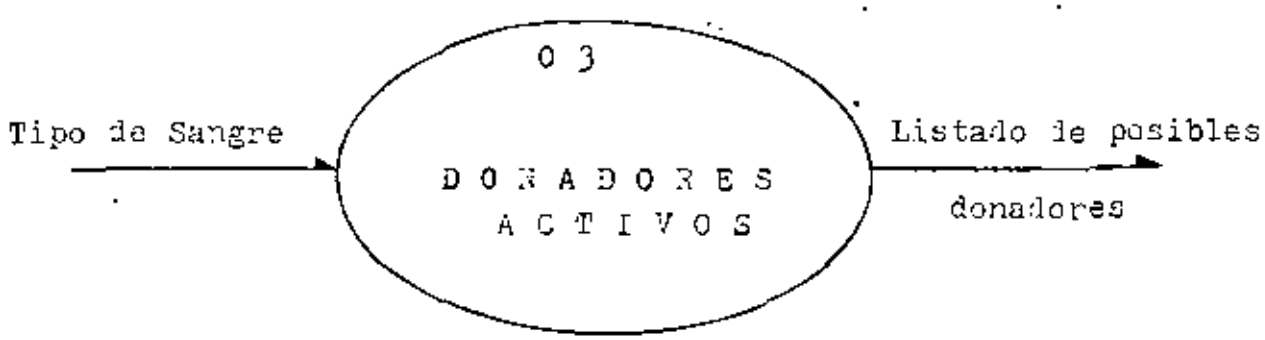
Administrativo.

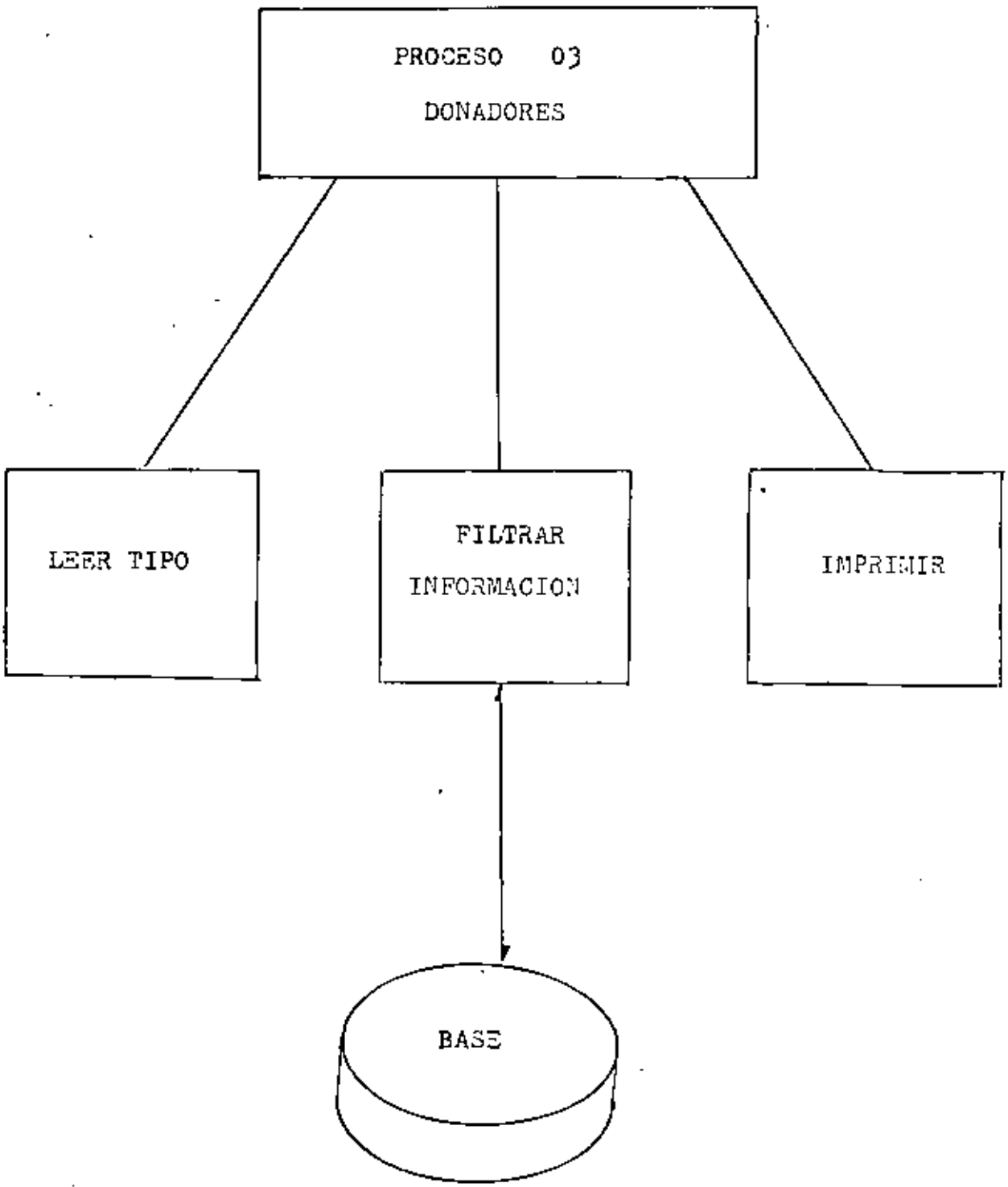
FRECUENCIA: Unica.

OBSERVACIONES: Se aplicarán filtros de entrada.

P R O C E S O 0 3

Dado un tipo de sangre nos da una lista de los posibles donadores con sus datos de localización.




PROCESO 03
DONADORES
LEER TIPO
FILTRAR
INFORMACION
IMPRIMIR
BASE

PROCEDIMIENTO:

Se leerá de terminal el tipo de sangre deseado. Se navegará por el set DONA-TIP y se comparará la primera instancia del registro tipo donadores. Si su estatus es 0 se navegará por el set DATOS y se obtendrá su dirección, nombre, teléfono y sexo. La edad se calculará por medio del RFC. En el caso de que no se encuentren estos datos se mandará un mensaje de error con la base.

Se escribirán estos datos en un archivo de salida, se incrementará el contador de activos. En el caso de que el estatus sea 1 se pasará a la siguiente instancia del tipo de registro donadores, incrementando el contador de donadores pasivos y se volverá a comparar el estatus repitiéndose esta secuencia hasta el fin de las instancias del registro tipo donadores.

Al finalizar se escribirán los contadores de activos y pasivos.

Adicionalmente se generará un proceso lanzador (03-L) que cargará las áreas y archivos necesarios así como el esquema de la base y su subesquema correspondiente.

FRECUENCIA:

Mensualmente o cuando la cantidad límite llegue a su mínimo.

FILTROS:

El tipo de sangre debe estar entre los que se manejan.

El estatus tiene que ser 0 ó 1.

Debe existir información del donador.

PRUEBAS:

Un tipo de sangre no existente.

Estatus mayor de uno ó menor que cero.

Un donador sin datos de localización.

Que hace si existe una área vacía.

Veracidad de los contadores (Información).

OBSERVACIONES:

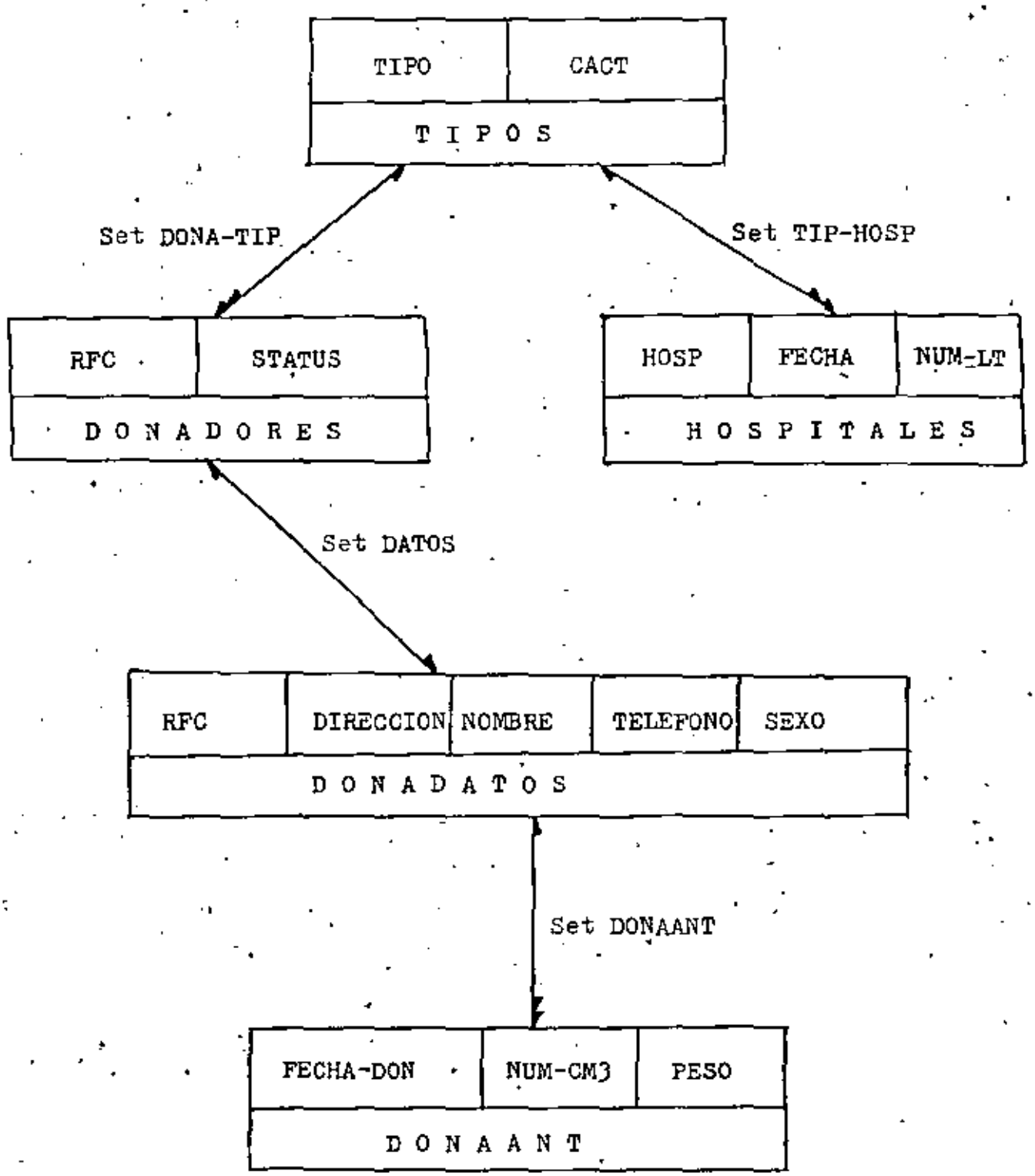
Se debe manejar una rutina de errores.con la base de datos.

Debe escribir el encabezado correspondiente en cada hoja de impresión.

Estatus 0 implica que el donador es activo.

Estatus 1 implica que el donador es pasivo.

3.- DESCRIPCION DE LOS ELEMENTOS DE LA BASE DE DATOS.



ORGANIZACION DE LA BASE DE DATOS

TABLA DE REGISTRO

DESCRIPCION	NEMONICO
1.- Tipos de sangre	TIPOS
2.- Clave de donadores	DONADORES
3.- Hospitales	HOSPITALES
4.- Datos de las donadores	DONADATOS
5.- Antecedentes de los donadores	DONAANT

TABLA DE CAMPOS

NEMONICO	FORMATO	DESCRIPCION	OBSERVACIONES
1.- REGISTRO TIPOS:			
TIPO	X(03)	Tipo de sangre	
CACT	9(06)	Cantidad actual	En litros
2.- REGISTRO DONADORES.			
RFC	X(12)	Registro Federal Causantes	
STATUS	9(01)	Estado	0 - Activo
			1 - Pasivo
3.- HOSPITALES.			
HOSP	X(30)	Nombre de la institución	
FECHA	9(06)	Fecha de entrega	año/mes/día
NUM-LT	9(04)	Cantidad Autorizada	En litros
4.- REGISTRO DONADATOS.			
RFC	X(12)	Registro Federal de Causantes	
DIRECCION	X(40)	Calle, Colonia, Ciúdad	
NOMBRE	X(30)	Nombre del donador	
TELEFONO	9(07)		
SEXO	9(01)		1 - Masculino
			0 - Femenino
5.- REGISTRO DONAANT.			
FECHA_DON	9(06)	Fecha de la donación	año/mes/día
NUM-CM3	9(05)	Cantidad donada	En cm^3
PESO	9(03)V9	Peso a la fecha	En Kg

Directorio de asistentes al curso: Administración de Proyectos de Software 15 al 20 de Febrero de 1982.

1. Juan V. Almaguer Meave
Cía. de Luz y Fza. del Centro, S.A.
Analista Programador
Melchor Ocampo 171-610
México 17, D.F.
566 09 20

Playa Guitarrón 419
Marte
México 13, D.F.

2. Ignacio Carbajal Carbajal
I M P
Jefe del Depto. Automatización
Av. L. Cárdenas 152
México 14, D.F.
567 66 00 Ext. 2527

Fdo. Rmez. 53
Obrera
D. Cuauhtémoc
México 8, D.F.
578 52 45

3. Orlando Esponda
TELEVISA, S.A.
Supervisor de Sistemas
Arcos de Belem 58-4° Piso
Centro
México, D.F.
588 08 18

Dalias Mz. 148 L. 16
Fracc. Ojo de Agua
Mpio. Tecamac, Edo. de Méx.

4. Alejandro Flores Alvarez
Secretaría de Programación y Presupuesto
Jefe del Depto.
Ecuador 77-4°
Centro
D. Cuauhtémoc
06010 México, D.F.
529 56 52

Valle de Grijalva No. 25
Valle de Aragón
57100 Estado de México

5. Rodolfo Gabriel Méndez
Secretaría de Gobernación
Subdirector de Ana. y Eva.
Galeana No. 5
Tlalpán
México, D.F.
573 56 62

Prolg. de Obsidiana No. 14
La Estrella
Gustavo A. Madero
México, D.F.
537 94 28

6. Juan Carlos Guerrero Ortiz
Constructora del Altiplano S.A. de C.V.
Ing. deSistemas
Prolg. Muñoz 790
Los Reyes
78170 San Luis Potosí, S.L.P.
30030

Av. Himno Nacional 405
Fracc. Avenida
San Luis Potosí, S.L.P.
30269

7. Jorge Gutiérrez Meraz
Consultores en Planeación Integral, S.C.
Subgerente de Informática
Artículo 123 No. 90
Centro
México 1, D.F.
510 48 60

Norte 54 No. 3643
Emiliano Zapata
D. G. A. Madero
07850 México, D.F.
517 83 85

8. Manuel Hernández Trejo
Secretaría de Programación y Presupuesto
Analista de Sistemas
Rep. de Ecuador 77
Centro
D. Cuauhtémoc
06010 México, D.F.
529 52 97

Miramón 138
Col. M Carrera
D. Cuauhtémoc
07070 México, D.F.
781 57 54

9. Artemio Martínez Ramos
PEMEX
Coordinador del Area
Gerencia de Informática
Marina Nal. 329 Edif. 1938 4° Piso
Col. Anahúac
México 17, D.F.
531 66 92

Dr. Atl 229-14
Sta. Ma. la Ribera
D. Cuauhtémoc
06400 México, D.F.
547 42 35

10. Miguel Angel Mora Espinosa
I P N
Profesor e Investigador
Unidad Profesional de Zacatenco
Col. Lindavista
D. G. A. Madero
México 14, D.F.
754 42 51

L. Cárdenas 490-1301
Tlatelolco
D. Cuauhtémoc
06900 México, D.F.
597 54 96

11. Pedro Moreno Sánchez
Industrias Realstorl, S.A.
Gte. de Proy. de Sist.
Ciruelos 99
Bosque de Las Lomas
D. M. Hgo.
México, D.F.
596 35 88

Varsovia 133
Valle Dorado
Tlanepantla, Edo. de Méx.
379 56 12

12. Juan Antonio Navarro Martínez
Fac. de Est. Sup. de Cuautitlán
Jefe de Lab. Microprocesadores
Cuautitlán, Edo. de México

Impacto 286
Col. Atlanta
Cuautitlán Izcalli, Edo. de Méx.

13. Eduardo Ortiz Velazquez
Analista de Sistema
Ecuador No. 77-4°Piso
Centro
México, D.F.
Tel. 5275297

Mar de Irlanda No. 51
Col. Popotla
D. Miguel Hidalgo
11400 México, D.F.
Tel. 399 08 47

14. Mario A. Palomar Alcibar
Asesor en la Sala de Computo
Ciudad Universitaria
Copilco
04360 México, D.F.

Av. Cuauhtémoc No. 877-1
Col. Narvarte
D. Benito Juárez
03020 México, D.F.
Tel. 543-78 95

15. Sergio Pérez Uscanga
Jefe de Sección
Moras No. 850
Col. del Valle

Paseo de Italia No. 101
Col. Lomas Verdes
Naucalpan Edo. de Méx.
53120, EDO. DE MEX.
6 58 45 10

16. Pablo Pinedo Navarro
Coordinador de Proyectos
Marina Nacional No. 329
Col. Anahuac
11320 México, D.F.
Tel. 531 66 92

Margaritas No. 13
San Angel Inn
D. Alvaro Obregón
01060 México, D.F.
Tel.

17. Ernesto Posadas Alba
Aeromexico

18. José Alejandro Ramírez Flores
Ingeniero
Melchor Ocampo No. 171
D. F.

Canarias No. 926-303
Col. Portales
D. F. Juárez
03300 Mexico, D.F.
Tel. 6 72 01 58

19. Ricardo Ramírez Verdeja
Jefe de Sección
Cuautitlan
Rancho Almaraz
Edo. de Méx.

Valle del Agua No. 27
Ixcalli del Valle
Edo. de Méx.
Tel. 12 03 45 55 - 98

20. Raymundo A. Rangel Gutiérrez
Facultad de Ingeniaria U.N.A.M.
Profesor
Ciudad Universitaria
Tel. 550 52 15 Ext. 3750

Av. Copilco No. 300
Copilco Universidad
D. Coyoacán
04360 México, D.F.
Tel.

21. Rubén Romero Ruiz
Secretaria de Gobernacion
Responsable Area de Informatica
Galeana No. 5
Tlalpan
14000 México, D.F.
Tel. 573 98 66

Paseo de Italia No. 69
Lomas Verdes
53120 Edo. de Mex.
Tel. 562 98 41

22. Clemente Juan Pablo
D.E.P.F.I. UNAM
Ciudad Universitaria
Tel. 5 50 52 15 Ext. 4493

José Rodríguez Gonzáles 9
Constitución de 1917
D. Iztapalapa
09260 México, D.F.
Tel. 6 91 08 48

23. Sergio Francisco Ruiz Palacios
D.E.F.F.I.
Ciudad Universitaria
04360 México, D.F.
Tel. 550 52 15 Ext. 4486

Avenida Nueve No. 37-1
Col. Independencia
D. Benito Juárez
03610 México, D.F.
Tel.

24. Gregorio Raúl Saldaña Tea
Aeronaves de Mexico
Analista de Sistemas
Paseo de la Reforma
Col. Cuauhtémoc
06500 México, D.F.
Tel. 2 86 44 22 Ext. 268

Durazno No. SA P.B.
Col. San Rafael
54120 Edo. de México
Tel. 5 65 57 43

25. Luis Manuel Téllez Lascano
Aeronaves de México
Paseo de La Reforma 445 7-8
México, D.F.
Tel. 525-43-45

Campeche No. 59
Col. Roma Sur
D. Cuauhtémoc
México, D.F.
Tel. 5 64 46 32

26. Roberto Vázquez Mendoza
Grupo Profesionales en Informatica, S.C.
San Antonio Abad No. 164-1er. Piso
Col. Tránsito
06820 México, D.F.
Tel. 5 78 97 11

Rosa Damasco No. 56
Col. Molino de Rosas
D. Alvaro Obregón
01470 México, D.F.
Tel. 6 51 28 60

27. José Othón Joaquín Vázquez Zepeda
I.T.E.S.M.
Tel. 3 36 00 Ext. 119

Leandro Valle No. 28
Edo. de Mex.
Tel. 54 58 58

28. Francisco J. Zavala Morales
Compañia Mexicana Aerofoto
11 de Abril No. 338
Col. Escandón
D. Miguel Hidalgo
Tel. 5 16 07 40 Ext. 144

Manuel M. Flores No. 149-A
Col. Chavez
D. Cuauhtémoc
06800 México, D.F.
Tel. 5 78 00 01