DIRECTORIO DE PROFESORES DEL CURSO:

TECNICAS MODERNAS DE DESARROLLO Y ADMINISTRACION DE PROGRAMAS MARZO 1983

1. DR. VICTOR GEREZ GREISER (COORDINADOR)
   INSTITUTO DE INVESTIGACIONES ELECTRICAS
   DIRECTOR DE SISTEMAS DE POTENCIA
   DIVISION DE SISTEMAS DE POTENCIA
   INTERIOR INTERNADO PALMIRA
   COL. PALMIRA
   APDO. POSTAL 475
   CUERNAVACA, MOR.
   91731 43811 Ext. 2158

2. DR. ROLANDO NIEVA GOMEZ
   INSTITUTO DE INVESTIGACIONES ELECTRICAS
   INTERIOR INTERNADO PALMIRA
   COL. PALMIRA
   APDO.POSTAL 475
   CUERNAVACA, MOR.
   91731 43811 Ext.2129

3. DR. GUILLERMO RODRIGUEZ ORTIZ
   INSTITUTO DE INVESTIGACIONES ELECTRICAS
   INTERIOR INTERNADO PALMIRA
   COL. PALMIRA
   CUERNAVACA, MOR.
   91731 43811 Ext.3209

4. M. EN C. MAURICIO MIER MUTH
   JEFE DEL DEPARTAMENTO DE ANALISIS DE REDES
   DIVISION DE SISTEMAS DE POTENCIA
   INSTITUTO DE INVESTIGACIONES ELECTRICAS
   C. F. E.
   CALLE DE DON MANUELITO S/N ESQ. AV. TOLUCA
   COL. OLIVAR DE LOS PADRES
   A.OBREGON
   01780 MEXICO,D.F.
   595 55 33 EXT.240

5. M. EN C. BENITO ZCHILINZKY ZCHILINZKY
   Subdirector de Análisis e Intercambio de Sistemas
   Dirección General de Política Informática
   S. P. P.
   Izazága No. 29-3°Piso
   Centro
   Cuauhtémoc
   México, D.F.
   76150 53 y 761 30 66 Ext.229

1. EVOLUCION DE LA COMPUTACION

   1.1. Evolución de los Sistemas de Cómputo

      1.1.1. Orígenes de las Computadoras
      1.1.2. Las Generaciones de las Computadoras

   1.2. Evolución de la Ingeniería de Programación

2. EL PROCESO DE PROGRAMACION

   2.1. El Enfoque Sistémico
   2.2. Fase de Definición de Requerimientos
   2.3. Fase de Diseño
   2.4. Fase de Desarrollo
   2.5. Fase de Operación y Mantenimiento

3. TECNICAS CONTEMPORANEAS DE DESARROLLO DE PROGRAMACION

   3.1. Introducción
   3.2. Técnicas de Especificación

      3.2.1. Planteamiento de Objetivos
      3.2.2. Análisis de Requerimientos
      3.2.3. Especificación de Diseño

   3.3. Técnicas de Diseño

      3.3.1. Arquitectura
      3.3.2. Diagrama de Estructura
      3.3.3. Programación Estructurada
      3.3.4. Detalle de Módulos

   3.4. Técnicas de Desarrollo

      3.4.1. Codificación
      3.4.2. Integración
      3.4.3. Planes y Programación Prueba

   3.5. Técnicas de Revisión

4. CONTROL DE CALIDAD

   4.1. Introducción
   4.2. Normas

      4.2.1. Normas de Análisis
      4.2.2. Normas de Diseño

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

TECNICAS MODERNAS DE DESARROLLO, ADMINISTRACION Y PROGRAMACION

EVOLUCION DE LOS SISTEMAS DE COMPUTO

MARZO, 1983

Palacio de Minería Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

INSTITUTO DE INVESTIGACIONES ELECTRICAS

17 Feb 83

CAPITULO 1

EVOLUCION DE LOS SISTEMAS DE COMPUTO

Desde su introducción al mercado, los computadores digitales han ido ganando terreno rápidamente, como muestra la tabla 1. A pesar de ello, el potencial de crecimiento de la industria de la computación es todavía enorme, más aún, por ejemplo, en los Estados Unidos en el año de 1980 solamente participó con menos del 1% del producto nacional bruto, a pesar de que se considera que en su país industrializado el 48% de las personas económicamente activas trabajan en actividades relacionadas con el manejo de información digital. Una de las razones por las cuales el mercado de computadoras digitales todavía no se ha

extendido más, son los problemas que representa el desarrollo de programación.

Estos problemas, que algunos autores afirman han alcanzado niveles de crisis, tienen su origen en el alto costo de desarrollo y actualización (mantenimiento) de la programación y en una creciente desconfianza del usuario (FAG81), ya que con frecuencia la programación no solo no cumple con sus objetivos, sino es además de baja calidad.

En parte se atribuye este problema a que durante el rápido avance del equipo (hardware) durante la década de los 70's, no se detectó la necesidad de desarrollar técnicas de programación a igual velocidad (FAG81).

Estudios recientes indican [BERB0A]:

a) El desarrollo cuesta alrededor de $,10 Dlls por línea de código documentada, es decir, varias veces el costo de un "chip" de circuito integrado.

b) Alrededor del 70% del costo total de la programación se ejerce después de la entrega del producto, y es empleado para su mantenimiento y eliminación de errores.

c) El mantenimiento de la programación puede consumir del 50 al 80% del presupuesto destinado al desarrollo de los programas.

La causa principal de estas deficiencias es el empleo todavía frecuente de técnicas inadecuadas de desarrollo de programación.

El objetivo de este libro es precisamente dar a conocer técnicas modernas de desarrollo y administración de proyectos de programación que ayuden a resolverlos. A pesar de estos obstáculos, la computadora se ha convertido en la máquina de manejo de información de nuestra era y, como afirma Mc. Luhan "La computadora se ha convertido en el sistema nervioso central de nuestra sociedad." En este capítulo bosquejaremos el desarrollo tanto de los equipos de cómputo (hardware) como de la programación (software).

1.1 Organización De Esta Obra

INSTITUTO DE INVESTIGACIONES ELECTRICAS

1.2 Equipo

En general es más conocida la historia del desarrollo del equipo que la de la evolución de la programación. Los medios de difusión se encargan de dar a conocer los últimos avances: mayor capacidad de memoria, mayor velocidad de operación, minimización del equipo, etc. Los siguientes párrafos permitirán resumir la evolución del equipo así como sus tendencias futuras.

1.2.1 Orígenes De Las Computadoras -

El origen de estos dispositivos puede remontarse a la antigüedad, ya que el hombre siempre ha tratado de diseñar mecanismos y dispositivos que le faciliten el trabajo.

Pronto descubrieron nuestros ancestros que los engranes y ruedas podían usarse para 'contar' revoluciones; se afirma que el griego Hero construyó un primer odómetro, antecesor del velocímetro actual.

Desde luego existió equipo para 'contar' desde antes de la introducción del odómetro. En primer lugar pueden usarse objetos para contar, piedras por ejemplo. Posiblemente algún pastor primitivo colocaría una pequeña piedra en su

bolsa por cada animal de su rebaño que dejara salir en la mañana; en la noche retiraría una de estas piedras por cada animal que regresara al establo. Y si hubiese una piedra, el pastor sabría que había animales que no habían regresado. Pronto también surgió la idea de usar el "principio de posición" para indicar diferentes cantidades. Poniendo piedras en diferentes lugares podían representarse grupos de uno, de cinco, de diez, elementos; éste es el principio del ábaco. En América, los Incas usaban un sistema para contar llamado "quipos". Eran una serie de cuerdas donde la posición de los nudos indicaba la cantidad de cada una de las cosas que se registraban en ellas.

Con el descubrimiento de los logaritmos, por Napier (1550-1617) pronto aparecieron los primeros dispositivos de cálculo analógico: las reglas de cálculo, cuyo funcionamiento está basado en escalas logarítmicas.

Pascal (1623-1662) construyó una máquina para sumar (basada en engranes) sin embargo, por problemas mecánicos nunca llegó a trabajar correctamente. Posiblemente la contribución más importante de ese siglo al desarrollo de las computadoras modernas la haya hecho el matemático inglés Charles Babbage (1790-1881) quien estableció los conceptos básicos de una máquina que llamaba diferencial. Estaba diseñada para calcular números, almacenar información y

INSTITUTO DE INVESTIGACIONES ELECTRICAS

seleccionar diferentes maneras de resolver problemas de acuerdo con el método más eficiente. Preveía ya el empleo de instrucciones operacionales y el de variables. Sin embargo, la ciencia mecánica en esa época no estaba suficientemente avanzada como para construir una máquina de la complejidad concebida por este científico. Sus trabajos fueron poco conocidos, al grado de que la mayoría de los investigadores que trabajaron durante la segunda guerra mundial en el desarrollo de computadoras con frecuencia atacaron problemas que ya habían sido resueltos por este científico inglés.

El francés Jacquard (1752-1834) empleó tarjetas perforadas para controlar telares. El americano Herman Hollerith (1860-1929) adaptó esta idea a las necesidades especiales de los cálculos censales. En 1944, como resultado de los trabajos de Aiken (1900-1973) apareció el computador Harvard Mark I movido electrónicamente, y cuyas instrucciones y datos se le alimentaban mediante cinta perforada y tenía componentes eléctricas, electrónicas y mecánicas. Fué la primera máquina que tenía las características de una computadora actual.

Durante la guerra en los países del eje también se trabajó en el desarrollo de computadoras. Konrad Zuse (1910-) construyó el calculador controlado por programa ZUSE-Z3, que al haberse puesto en servicio en 1941 antecedió a la máquina

INSTITUTO DE INVESTIGACIONES ELECTRICAS

de Aiken.

La primer computadora electrónica denominada ENIAC fue construida en 1946 por J. P. Eckert (1919-) y J. W. Mauchly (1907-) en la Universidad de Pennsylvania. Esta era la primer máquina totalmente electrónica, capaz de multiplicar dos números de diez dígitos en tres milésimas de segundo (comparado con los tres segundos que tardaba la máquina Mark I). Sin embargo, contenía 19000 tubos de vacío, consumía 200 KW y su uso estaba limitado al cálculo de trayectorias balísticas.

A mediados de la década de los 40's, aplicando las ideas del álgebra binaria desarrollada por el inglés George Boole (1815-1864), Von Neumann (1903-1957) de la Universidad de Princeton demostró cómo emplear lógica y aritmética binarias para estructurar programas "almacenados". Comprobó que con el mismo lenguaje empleado para codificar un programa se pueden codificar los datos.

1.2.2 Las Generaciones De Computadoras

A partir de estos primeros sistemas, el desarrollo de las computadoras se ha acelerado y una forma conveniente de clasificarlas es en generaciones.

INSTITUTO DE INVESTIGACIONES ELECTRICAS

La primera generación de computadoras estaba caracterizada por el uso de tubos de vacio y el empleo de diferentes medios para almacenar información. Entre éstos señalar la línea de atraso de mercurio empleada en la máquina UNIVAC-1. Usando desarrollos realizados en la Universidad de Manchester se construyeron los sistemas de almacenamiento electrostático que fueron empleados en los sistemas IBM-701 (International Business Machines). Posteriormente la serie IBM-700 y la serie UNIVAC-1103 usaron sistemas de almacenamiento de tambor magnético que tenía gran capacidad de almacenamiento, aunque resultaban un tanto cuanto lentos. Muchos otros fabricantes fueron adoptando este tipo de memoria. Hacia 1953 fue introducida la memoria de núcleo magnético desarrollada tanto en los laboratorios de la Radio Corporation of America (RCA) como del Instituto Tecnológico de Massachusets (MIT). Esta tecnología fue rápidamente adoptada por todos los fabricantes importantes en sistemas de cómputo.

La tabla 1.2.2 resume las principales características de esta primer generación de equipos de cómputo así como ejemplos típicos de máquinas.

El invento del transistor en 1948, abrió la puerta al desarrollo de una nueva generación de computadoras. Hubo sin embargo que resolver problemas tecnológicos y de

INSTITUTO DE INVESTIGACIONES ELECTRICAS

fabricación antes de que este elemento de estado sólido pudiera ser incorporado a los sistemas de cómputo. Hasta 1959 comenzaron a aparecer en el mercado computadoras transistorizadas en cantidades importantes. Todas estas computadoras de segunda generación usaban sistemas de núcleo magnético para almacenamiento de la información aunque también empleaban discos magnéticos y cintas magnéticas como dispositivos auxiliares de almacenamiento.

Philco introdujo la primer computadora transistorizada. Sin embargo, no pudo penetrar este mercado y se retiró del campo en 1964. Esta época también vió el nacimiento de otro de los gigantes de esta industria: la CDC, Control Data Corporation, que pronto entró al mercado con la máquina 1604 y posteriormente con modelos más poderosos como lo fueron el 300, 3660 y 3800.

En esta época bien aparecieron en el mercado las primeras computadoras diseñadas específicamente para el procesamiento de grandes volumenes de información como fue por ejemplo la máquina IBM 7070.

La tercera generación de computadoras está caracterizada no solamente por la introducción de circuitos integrados y la tecnología de integración a gran escala, sino también por características arquitectónicas importantes como el empleo

de bytes de 8 bits para representar caracteres. Con la introducción del sistema IBM 360 en 1964 puede considerarse que se inició esta generación. Este sistema fué sucedido por un sistema todavía más poderoso, el 370. UNIVAC sacó al mercado el sistema 9000 siguiendo los patrones fijados por el sistema 360 de IBM. La CDC introdujo el sistema 6600 en 1964. Esta fué por muchos años la computadora más rápida y poderosa del mercado y fué sucedido después por el sistema 7600 y finalmente por la serie Cyber.

Otra característica importante de esta tercera generación es la aparición de las llamadas minicomputadoras representada sobre todo por la serie PDP de Digital Equipment Corporation (DEC). Con el desarrollo de la integración a gran escala fué posible desarrollar computadoras bastante completas en uno o dos chips semiconductores apareciendo con ella las microcomputadoras.

Como hemos resumido en los párrafos anteriores, a partir de la introducción de la primera computadora y con la llegada de los elementos de estado sólido y después los circuitos integrados, el tamaño de las computadoras se ha reducido drásticamente tal como muestra la figura 1.2.1.

Pero no solamente ha disminuido su tamaño, sino ha aumentado simultáneamente su capacidad como muestra la figura 1.2.2.

reduciéndose al mismo tiempo el precio (figura 1.2.3). Por tanto no es de extrañarse que la industria de la computación sea una de las de más rápido crecimiento, cuyo futuro solamente se ve oscurecido por el aumento en costo que ha sufrido el desarrollo de programación comparada con la disminución en el costo del equipo (figura 1.2.4).

INSTITUTO DE INVESTIGACIONES ELECTRICAS

1.2.3 Futuro De Las Computadoras. -

Tanto avances en los componentes actuales de las computadoras, como pueden ser las memorias, o nuevas arquitecturas, caracterizan las nuevas generaciones de estos equipos.

Se preveen avances importantes en dispositivos de memoria. Lo más probable es que veamos aplicaciones masivas de las burbujas magnéticas. Dentro de una substancia magnetizada pueden existir burbujas microscópicas; varias decenas de miles de ellas. Realmente son pequeñas partes de substancia que han sido magnetizadas en una forma diferente al resto. La presencia o ausencia de una burbuja de estas es empleada para representar un bit de información. Variando rápidamente el campo magnético que rodea un "chip", las burbujas se mueven alrededor de éste a gran velocidad y hay un lugar donde su presencia se detecta. Para leer el bit que una burbuja determinada representa, hay que esperar que

ésta llegue al lugar donde se puede leer. Esto es más lento que leer información de memoria principal, pero mucho más rápido que emplear discos o cintas. Probablemente tendrán a mediados de la década de los 80's "chips" con 100 MB de capacidad de memoria aunque hay quien afirma que se alcanzarán hasta 256 MB. Esto, desde luego, es un gran avance si se compara con las burbujas actualmente disponibles que solamente alcanzan entre 64 KB a 1 MB.

Almacenamiento en burbuja magnética nunca será tan rápido como en memoria principal. Siempre será más lento debido al tiempo que tarda la memoria en llegar al punto donde se detecta; sin embargo, tienen una importante ventaja: si se corta el suministro de energía eléctrica las burbujas se conservan de manera que la información no se pierde.

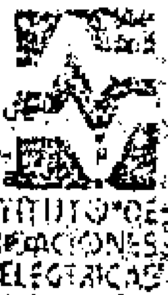
Desde luego, los discos también van a aumentar su capacidad. Posiblemente veamos diskettes con hasta 20 MB a fines de la década [BERR80].

Los dispositivos anteriores representan, respecto a velocidad de acceso, una situación intermedia entre almacenamiento principal y los discos y cintas. Por lo tanto, es necesario cubrir ese hueco, desarrollando dispositivos con velocidad de acceso intermedia. Una de las técnicas que se explora es el empleo de discos ópticos; éstos tienen

pequeños agujeros quemados en su material reflectivo que son detectados con rayos Lasser. La presencia o ausencia de uno de éstos representa la presencia de un bit 0 o al revés. Estos dispositivos pueden ser muy rápidos, pero una vez escritos sobre ellos, no pueden ser borrados.

Los dispositivos de almacenamiento mencionados anteriormente no harán desaparecer la memoria de disco. Ya hace dos décadas por lo menos que había personas que predecían que su potencial había sido agotado. Sin embargo, la densidad de bits en éstos sistemas ha aumentado varios cientos de veces en la última década, y es posible que siga incrementándose a igual velocidad en la presente. No obstante, la tecnología actual sí parece estar llegando a sus límites para un medio de grabado en particular, la cubierta de óxido de hierro, aunque los límites magnéticos teóricos todavía están lejos en varios órdenes de magnitud.

Avances en el desarrollo de circuitos integrados harán seguramente posible, que en esta década empiecen a comercializarse grandes computadoras construidas de solamente uno o dos circuitos integrados a gran escala. Estos nuevos circuitos presentan al diseñador de microprocesadores dos caminos:



1. Continuar con la tendencia actual en la construcción de circuitos integrados a gran escala, es decir diseñar sistemas cada vez más complejos donde el (hardware) realiza funciones que antes realizaba la programación (software).

2. Puede proceder, como recomienda otro grupo, en la dirección opuesta y construir procesadores más sencillos donde un número mayor de funciones sean realmente realizadas por programación.

Una mayor complejidad dejará al diseñador usar circuitos integrados cada vez más baratos pero complejos para substituir la cada vez más cara programación. En teoría, estos sistemas tan complejos reducirían el costo del desarrollo de la programación, tendrían un mayor número de funciones integradas al equipo y por lo tanto el costo total capitalizado de los sistemas disminuiría.

Sin embargo hay especialistas tanto en el medio académico como en los laboratorios de desarrollo que no están de acuerdo y afirman que estas máquinas más complejas ofrecerían poca ganancia tanto en características como en reducción de costo. Ellos proponen al contrario, que se empleen sistemas más simples y por lo tanto más baratos y que se desarrollen compiladores que realizarían con

eficiencia la función de optimizar el sistema. Consideran una solución más efectiva en costo el desarrollo de mejores compiladores que vayan simplificando el trabajo del programador.

INSTITUTO DE INVESTIGACIONES ELECTRICAS

Desafortunadamente todavía no existen modelos para evaluar el beneficio de una solución respecto a la otra [BER81].

Entre las nuevas arquitecturas de computadoras se cuenta ya con procesadores de arreglos cuyo uso se extenderá [BER82A]. Estos procesadores de arreglos, son ya de 10 a 100 veces más rápidos que grandes computadoras. Estas computadoras alcanzan velocidades de hasta 11 millones de operaciones de punto flotante, gracias a su arquitectura particular pudiendo realizar varias funciones diferentes simultáneamente. Son diseñadas específicamente para procesar vectores, matrices y otros arreglos numéricos. Aplicadas adecuadamente pueden obtenerse grandes aumentos en la velocidad de procesamiento, por ejemplo: en estudios de flujos de carga [POT82] se han reportado aumentos de hasta 5 veces en velocidad empleando procesadores de arreglos en lugar de grandes computadoras.

Como son en realidad, más que unidades aritméticas, memorias de alta velocidad, requieren de una computadora para proveer soporte de entrada/salida y un sistema operativo.

Los circuitos integrados a muy gran escala y nuevas arquitecturas de las máquinas harán posible espectaculares avances en la velocidad de las máquinas. Predicciones optimistas estiman que se llegarán a alcanzar velocidades de mil millones de operaciones de punto flotante por segundo (FLOPS) mientras que las máquinas más rápidas actuales alcanzan velocidades de veinticinco millones de FLOPS, llegando a veces a ciento sesenta millones. Sin embargo, para alcanzar estas predicciones habrá que resolver algunos problemas. No existe todavía concenso sobre cuál será la mejor arquitectura para estas nuevas máquinas. Las dos arquitecturas más promisorias parecen ser: máquinas de multiproceso y sistemas de flujo de datos. [BER92].

Las primeras son las más flexibles. En ellas un problema es dividido por el programador o compilador de tal manera que el procesador puede realizar los cálculos operando concurrentemente diferentes flujos de datos con diferentes instrucciones.

En los sistemas de flujo de datos, el algoritmo para realizar los cálculos es escrito primero en un lenguaje especial de programación diseñado para aplicaciones de flujos de datos. El programa adquiere con ello la apariencia de una gráfica dirigida que se implementa después directamente en una serie de unidades de equipo (hardware)

INSTITUTO DE
INVESTIGACIONES
ELECTRICAS

interconectadas. Cada unidad realiza una sola operación cada vez que los datos llegan a ella.

Otra incógnita es la velocidad máxima que realmente pueden alcanzar estas nuevas máquinas. En cualquiera de estas arquitecturas siempre habrá procesadores que no están operando debido a conflictos sobre memoria o canales de comunicación. La velocidad máxima no será, por lo tanto, la teórica, sino bastante más baja. Las estimaciones sobre su valor todavía fluctúan grandemente. Para explotar los posibles aumentos en velocidad será necesario:

1. Que todos los procesadores inicien su operación sincronamente.

2. Desarrollar algoritmos para organizar la memoria de tal manera que se eviten conflictos.

Cabe preguntar, desde luego, por qué se necesitan estas mas altas velocidades. Básicamente para resolver problemas de simulación en tres dimensiones, sobre todo en aquellas aplicaciones que requieren de solución de ecuaciones en derivadas parciales con valores en la frontera, como son los de flujos y los que se presentan en predicción meteorológica, [LFR82], modelado de sistemas complejos, [BUS80] y procesamiento de imágenes [SEH82], entre otras.

INSTITUTO DE INVESTIGACIONES ELECTRICAS

1.3 Programación

Menos conocida es la historia del desarrollo de las técnicas de programación. En los primeros sistemas el equipo fijaba los límites a la capacidad del sistema. Pronto sin embargo empezaron a surgir cuellos de botella en el desarrollo de la programación que limitaban el potencial de estos sistemas. Aunque menos conocidos que los avances en el equipo, los desarrollos en la ingeniería de la programación han sido igualmente importantes.

1.3.1 Evolución De La Ingeniería De Programación.

Los primeros sistemas generalmente se diseñaban e implementaban no tomando en cuenta su interdependencia con otros. Este procedimiento desde luego minimizaba los costos de diseño pero complicaba futuros cambios a los programas cuando se identificaban nuevas funciones que podían ser computarizadas o que se deseaba integrar al sistema.

Esta metodología producía programas aislados, que resultaba imposible ensamblar en un sistema general de manejo de información, dando margen a la necesidad de reprogramar todos los sistemas, que ahora, dentro de un contexto integral, podrían clasificarse más correctamente como

subsistemas.

En estos primeros esfuerzos se dedicaban los mayores a las fases de codificación y prueba, como se muestra en la figura 1.3.1. Generalmente, a raíz de este proceder, se hacían observaciones como: ¡¡Pero eso no es lo que yo deseaba!! ¿¿No me habías informado que estabas por terminar??, enfatizando que, por un lado, los requerimientos del programa de computadora eran distintos de los objetivos logrados por el programador y, por otra parte, el tiempo de desarrollo de programación se extendía más allá de lo planeado.

A medida que fueron evolucionando los sistemas de cómputo y ampliándose los objetivos que deberían de satisfacer, fueron cambiando las técnicas de diseño.

Como principal herramienta de diseño durante este periodo inicial, anterior a la década de los 60's, pueden mencionarse los diagramas de flujo, una forma esquemática de registrar desde su fuente, la lógica y la operación del programa, pasando por los diferentes procesos de operación hasta llegar al producto final. Estos diagramas de flujo dan una imagén gráfica de las operaciones que hay que realizar y al nivel de programación representan detalladamente los pasos lógicos que tiene que realizar la

computadora con objeto de llegar al resultado deseado. Sin embargo, estos diagramas nos ayudan a darle a los programas la estructura y la flexibilidad que se requie[illegible] actualidad. Satisfacían posiblemente los requerimientos de información solicitados por la administración, pero el empleo irrestricto de ramificación controlada (GO TO) producía programas con una lógica imposible de descifrar.

INSTITUTO DE INVESTIGACIONES ELECTRICAS

Siguiendo esta metodología tradicional pronto se llegó a lo que se conoce actualmente como "La crisis en el desarrollo de programación", que afecta no solamente programación antigua sino también, a nueva programación. Diseños erróneos, errores tipográficos, técnicas tradicionales y codificación no normalizada, crearon sistemas que durante toda su vida útil presentaban serios problemas en su mantenimiento y actualización. Los usuarios, ante programas mal estructurados y por lo tanto difíciles de mantener, frecuentemente caían en la tentación de rediseñarlos y reprogramarlos totalmente. No era por lo tanto de extrañarse, que los costos de mantenimiento del programa, llegasen a ser del orden del 90% del costo total de su ciclo de vida útil.

También se encontraban serios problemas en la administración de proyectos de programación. La base de la administración es planear, organizar y controlar. Planear y organizar son

conceptos fáciles de entender, pero el problema de controlar el desarrollo de un programa de computación escapaba a la metodología de la época. Lo clásico era el "Síndrome del 90%": los programadores informaban a los administradores que un sistema estaba terminado en un 90% y de ahí en adelante resultaba que todavía había que emplear, por lo menos, otro tanto de los recursos ya usados para terminar la programación.

Como muestra la figura 1.1.2, durante la década de los 60's el equipo avanzó rápidamente empezando a surgir la necesidad de contar con metodología más avanzada para el desarrollo de programación. Entre estas técnicas podemos mencionar el Algebra de Información que permite especificar los datos relevantes y después determinar las relaciones y reglas asociadas a su manipulación. Con esta metodología se pensaba o se esperaba optimizar el proceso de desarrollo de programación; sin embargo tuvo muy poco impacto sobre trabajos actuales. No obstante, mostraba que los investigadores empezaban a preocuparse por métodos de diseño más formales de la programación.

A principios de esa década surgió lo que se llamó "Plan para el Estudio de Organización". Este esquema definía tres conjuntos básicos: datos generales, información estructural y elementos operacionales. Los primeros contenían

INSTITUTO DE INVESTIGACIONES ELECTRICAS

básicamente información histórica de la empresa. El conjunto de información estructural incluía detalles sobre sus productos materiales, mercados, situación financiera, personal, entre otros. Finalmente los elementos operacionales proveían los datos que describían la operación del sistema, como diagramas de flujo y distribución total de los recursos dentro de las actividades operativas de la empresa. Esta técnica muestra claramente la preocupación por el flujo de información dentro de un sistema, característica, como se verá mas adelante, de las técnicas modernas de programación.

En esta misma dirección se ideó lo que se denomina "Sistemas exactamente desarrollados". Esta técnica distingue varios pasos formales: inicialmente se definen las salidas y las entradas necesarias para crearlas; a continuación se identifican las necesidades de cómputo para convertir las entradas en las salidas necesarias, incluyendo desde luego, restricciones y relaciones significativas entre los diferentes procesos de cómputo; posteriormente se determinaba qué información debería de ser guardada para pasos subsecuentes, y finalmente se requería una definición formal de relaciones lógicas en forma de una tabla de decisión. Además se seguía una técnica "exacta" para referir los datos: cada vez que un dato era empleado se le ligaba a su referencia o fuente previa, a fin de crear una

cadena para cada elemento, indicando claramente el flujo desde la entrada a la salida. Podemos considerar que estas ideas fueron base de las técnicas que surgieron en la década siguiente y que se conocen como diseño de arriba hacia abajo.

INSTITUTO DE INVESTIGACIONES ELECTRICAS

1.3.2 Panorama Futuro -

Las tendencias futuras en el desarrollo de programación pueden clasificarse en evolutivas y revolucionarias. Entre las primeras se tendrá el empleo cada vez mayor de la computadora para auxiliar en la construcción de programas convencionales y el desarrollo de lenguajes de un mayor nivel que el actual. Los llamados lenguajes "no asociados a un procedimiento" que permiten al usuario especificar QUE operaciones desea hacer y no COMO las desea hacer, y otros lenguajes con apoyo gráfico, son otros ejemplos de las tendencias evolutivas.

Se ha cuestionado la posibilidad de resolver a fondo la crisis de la programación, a menos que se introduzcan cambios revolucionarios en la filosofía de la programación que afectarían también en forma radical la arquitectura de las computadoras.

Para ayudar a resolver la llamada "crisis de desarrollo de programación" se verá sin duda un incremento en el empleo de

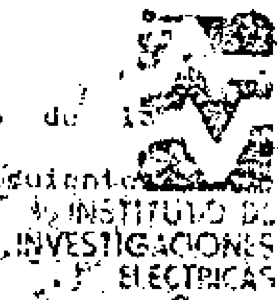
herramientas que usen la computadora como auxiliar en esta tarea. Una forma en que la computadora ayudará al desarrollo de programación es mediante el empleo de herramientas que auxilien en tareas específicas como ensambladores, editores de texto y programas para chequeo de código, entre otros (véase 4.3).

La segunda posibilidad, mucho más ambiciosa, es ir substituyendo la metodología que se emplea actualmente fundamentada básicamente en la experiencia por un enfoque sistémico. Este estará basado en una filosofía de desarrollo que deberá incluir un modelo teórico del proceso de desarrollo.

Herramientas de programación implementadas en la computadora soportarán el desarrollo de otros sistemas. Esta metodología, en combinación con sus herramientas, constituirán un sistema de soporte de desarrollo [LAU82]. El sistema consistirá de:

1. Un enfoque del proceso de desarrollo, que dará lugar a un modelo que lo represente y una metodología para proceder con el desarrollo del sistema.

2. Herramientas de programación basadas en este modelo, que soportarán el proceso de desarrollo.



Estas herramientas que utilizarán en el desarrollo de la programación deberán de tener las siguientes características:

1. Emplear la computadora como una herramienta de soporte durante todo el ciclo de desarrollo de la programación.

2. Apoyar la comunicación entre los integrantes del grupo: administradores, programadores y usuarios, con objeto que todos ellos tengan un adecuado conocimiento del sistema.

3. Facilitar el conocimiento del sistema para promover su aceptación y aplicación por parte del usuario.

4. Apoyar el desarrollo, no solamente del sistema de programación, sino también del equipo que lo soportará.

5. Facilitar la administración del proyecto y el control de calidad de la programación.

Actualmente la mayoría de las herramientas de soporte son de uso restringido. No se emplean en todo el ciclo de desarrollo de la programación y solo auxilian en forma limitada en la comunicación entre desarrolladores y usuarios. Su uso es restringido a aplicaciones fuera de línea e implantables solamente en grandes computadoras. Con el incremento en el empleo de minicomputadoras probablemente

sea necesario, además, desarrollar herramientas de soporte que puedan correrse en este tipo de equipo.

Otros enfoques evolutivos llevarán básicamente a la automatización de la programación; así como hace dos décadas el lenguaje Fortran y otros de alto nivel fueron desarrollados para permitir al programador generar muchas instrucciones de máquina a partir de un número de instrucciones de alto nivel, los esfuerzos actuales tienen el mismo objetivo: producir, a partir de una indicación simple del objetivo o comportamiento de un programa, algoritmos enteros de alto nivel para llevar a cabo la operación [LERR83].

Estos lenguajes, algunos de los cuales están a punto de salir al mercado, como el lenguaje Model, son lenguajes llamados de flujo de datos; el compilador del programa usa las relaciones entre los datos para determinar la secuencia en que debe de operarse sobre ellos, en contraste con los lenguajes actuales de flujo de control, donde la secuencia en que deben de llevarse a cabo las operaciones está explícitamente indicada en el programa.

Estos nuevos lenguajes permiten al usuario indicar qué operaciones desea realizar. En lenguajes convencionales hay que indicar cómo se desea realizar las operaciones. Desde luego todos estos nuevos lenguajes de flujo de datos tendrán capacidad interactiva. Si surgen dudas en su

interpretación, interactuarán con el usuario para aclaración. Un ejemplo de este lenguaje es el lenguaje Model, es empleado para modelado econométrico, otro es el lenguaje Moral, diseñado para la programación automática de equipo de prueba. Para enfatizar la importancia de estas herramientas se puede mencionar, que la programación para equipo de prueba automática de componentes y sistemas electrónicos cuesta actualmente diez veces más que el equipo (hardware).

Para ser empleados por personas que no están profesionalmente entrenadas para comunicarse con computadoras, como puede ser el usuario clásico de las minicomputadoras, se desarrollan nuevos lenguajes para la automatización del proceso de programación [LER 82]. Entre estos destacan "query-by-example", es decir 'búsqueda por ejemplo' desarrollado por la IBM, su variante 'procedimientos de oficina por ejemplo', y "small talk", plática pequeña, desarrollado por la compañía Xerox.

El lenguaje de 'búsqueda por ejemplo' de la IBM, es un lenguaje que permite dar instrucciones a la computadora en forma de tablas. Por ejemplo, si queremos construir un recibo, construiremos primero una tabla que incluya todos los datos de esa factura, y luego en otra tabla, la relación que existirá entre los datos de esa factura. El lenguaje, construye automáticamente el programa para relizar las

operaciones indicadas. La misma idea está detrás del lenguaje "procedimientos de oficina por ejemplo" que expande los conceptos del lenguaje anterior para permitir actividades de oficina como son el procesamiento de palabras, la escritura de reportes, el trazo de gráficas, y del correo electrónico entre otros. Todos estos nuevos lenguajes tienen una importante capacidad gráfica y una de sus principales ventajas es que la automatización de la oficina puede llevarse a cabo en una forma progresiva.

Si bien herramientas auxiliares de diseño y desarrollo ya se encuentran en un estado avanzado de desarrollo, y lenguajes como el MODEL pronto saldrán al mercado, algunos expertos como John Backus objetan este enfoque evolutivo y dudan que la arquitectura actual de las máquinas sea la adecuada. La traducción de especificaciones, primero a un lenguaje de alto nivel y después al nivel de lenguaje de máquina es demasiado lenta. Propone un enfoque revolucionario para resolver la crisis de producción. Como se mencionó en la introducción a este capítulo el equipo de los sistemas de Cómputo ha avanzado muchísimo en las últimas décadas, sin embargo, en el mismo período, ni la arquitectura básica de las computadoras ni los lenguajes de programación empleados para controlar el sistema han cambiado en forma significativa. Como resultado de esto no solamente se tienen elevados costos en el desarrollo de programación, sino tampoco se puede explotar totalmente el potencial de

los circuitos integrados a gran escala. Para aprovechar este potencial se inicia el desarrollo de un nuevo estilo de programación llamado programación funcional que permitirá el desarrollo de programas mucho más baratos. Estos programas requerirán de máquinas con nuevas arquitecturas que permitirán explotar plenamente el potencial de los circuitos integrados a gran escala.

La arquitectura clásica de las computadoras está basada en el modelo del matemático John Von Neumann (véase 1.2.1) y consiste básicamente de un procesador central, una memoria y conecciones entre ambos que transmiten unidades de información. Basados en esta arquitectura, los lenguajes actuales de programación resultan complejos ya que tiene que transmitir secuencialmente las unidades de información. Además es muy difícil emplear estos programas como bloques de programas más amplios.

En las máquinas convencionales la información debe pasar palabra por palabra de la memoria al CPU haciéndolas lentas. Los programas alteran los datos en memoria palabra por palabra. Las variables en el programa se usan para designar lugares de almacenamiento y una instrucción entera se necesita para alterar el dato de cada variable. De esta manera se forman programas que son una secuencia repetitiva de instrucciones que controlan la manera y las condiciones en que éstos se llevarán a cabo.

El hecho de que los programas actuales operen de palabra en palabra es otro factor importante que limita su aplicación universal, ya que conjuntamente con el programa se [illegible] el mapa de almacenamiento para que éste pueda operar [illegible]

Los programas actuales básicamente mapean lugares de almacenamiento a otros lugares de almacenamiento. Sin embargo, la finalidad de un programa es realmente mapear objetos a otros objetos; por ejemplo, un programa convencional para producir el inverso de una matriz no tiene por objeto almacenar datos en memoria y mapear lugares de memoria a otros lugares de memoria. Sin embargo las máquinas de Von Neumann requieren que se haga esta operación. El programador tiene que producir un programa que precisamente produzca este mapeo. Si por ejemplo un programador ha desarrollado un programa para invertir una matriz y otro programador desarrolla un programa para trasponer estos arreglos, si no se conoce el mapa de memoria de estos dos programas lo más probable es que debido a que ambos tienen diferente plano de memoria, no se puedan concatenar estas operaciones. Teniendo por una parte el programa para resolver estos problemas desde hace dos décadas se desarrolló el programa LISP, que sin embargo todavía conserva algunas características de los lenguajes convencionales. Más radicales son los cambios que introducirá la llamada programación a nivel funcional [BAC

cuyo énfasis se concentrará no en la combinación de objetos, sino en la combinación de programas. En este estilo de programación, los programas existentes se agruparán agregando operaciones llamadas de formación, y estos programas a su vez podrán ser usados como bloques para construir programas todavía mayores. Este sistema permite expresar fácilmente operaciones en paralelo y sugiere la construcción de máquinas formadas por grandes cantidades de unidades de procesamiento idénticas, para realizar operaciones simultáneamente aprovechando, de esa manera, el potencial de los circuitos integrados a gran escala. Aunque estas máquinas, como las llamadas de flujo de datos, todavía no han rebasado la fase experimental, su potencial futuro parece promisorio. Así como hace 25 años con la introducción del Fortran, que reemplazó la necesidad de programar en lenguaje de máquina se obtuvo un enorme incremento en la productividad, se espera que con estos nuevos lenguajes también se obtenga un beneficio similar.

1.3.3 El Ingeniero En Programación.

En la introducción a este capítulo se mencionó la denominada "Crisis de la programación" que tardíamente se ha empezado a resolver con el desarrollo de nuevas metodologías de programación, algunas de las cuales que describimos en esta obra. El ya muy extenso acervo de técnicas que se conocen

para desarrollar sistemas complejos permite definir el perfil de un nuevo tipo de profesionista: el Ingeniero de programación [FAGB1].

INSTITUTO DE INVESTIGACIONES ELECTRICAS

Medida que se extiende el uso de computadoras (en general, los microprocesadores se encuentran cada vez en mayor número de equipos) el problema de encontrar recursos humanos adecuados, para diseñar, construir, seleccionar y mantener estos equipos se hace más agudo.

Este ingeniero de programación deberá ser, en parte programador, en parte especialista en equipos y diseñador de programación. La demanda de este tipo de profesionistas deberá ser, en parte, cubierta por profesionistas con entrenamiento formal en otras ramas, quizá electrónica o matemáticas, que con la asistencia a cursos especializados y la práctica suplementen la preparación formal en computación. Pero aún con estas improvisaciones, la demanda en la industria y en el sector público de especialistas en computación excede a la oferta. Todavía agrava más este problema el hecho que la programación, en general, no es transportable de un sistema a otro, ya que gran parte del código está diseñado en lenguajes dependientes de máquina.

Este profesionista deberá entender tanto el equipo como la programación con el objeto de diseñar, seleccionar o poder

mantener estos sistemas. Su demanda será grande. Estimaciones conservadoras señalan que para 1988 se van a requerir el doble de ingenieros en programación que actualmente existen. El ingeniero de programación tendrá que conocer el sistema con su equipo operativo y sus problemas de programación como un todo. Tendrá que estar familiarizado con especificaciones electrónicas, técnicas de definición de requerimientos y diseño de programación, normas de codificación y control de calidad. Es decir, el ingeniero de programación debe ver el panorama amplio, abarcando su campo de acción desde la selección de microprocesadores hasta la concepción de grandes sistemas de cómputo. Deberá estar familiarizado con el equipo, ya que sin el conocimiento de la relación entre la arquitectura de la computadora y la estructura del sistema operativo y la programación no es posible seleccionar adecuadamente un equipo o desarrollar programación eficiente. Como debe poder administrar proyectos de computación tendrá que estar familiarizado con técnicas de administración e ingeniería industrial, con habilidad para organizar y estimar los recursos que se requieren para alcanzar un determinado objetivo. Para sus actividades de administración deberá tener conocimientos de estimación de costos, presupuestación y análisis de requerimientos. Para resolver problemas específicos, el ingeniero de programación deberá poder usar análisis numérico, optimización matemática, simulación

INSTITUTO DE INVESTIGACIONES ELECTRICAS

modulado, teoría de información y otras técnicas. En general, un ingeniero en sistemas tiene los conocimientos anteriores. Pero además de ellos deberá estar familiarizado con el campo dela computación, ya que por ejemplo:

El diseño de programas requiere de conocimientos de teoría de colas, programación estructurada, sistemas operativos y lenguajes de programación. El ingeniero en programación deberá poder interaccionar con los especialistas de diseño de equipo desde la fase inicial de desarrollo del sistema. Sus funciones no terminan con el diseño, continúan durante la construcción del sistema. Durante esta fase pueden surgir problemas que requieran de interacción mutuamente los grupos de desarrollo de equipo y de programación para resolverlos.

Los siguientes capítulos de este libro introducen al lector en el área de la ingeniería de programación, comenzando por una identificación de las diferentes fases del proceso de programación y una definición y descripción de cada una de las actividades asociadas.

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

TECNICAS MODERNAS DE DESARROLLO, ADMINISTRACION Y PROGRAMACION

EL PROCESO DE PROGRAMACION

MARZO, 1983

Palacio de Minería Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

INSTITUTO DE
INVESTIGACIONES
ELECTRICAS

17 Feb 83

CAPITULO 2

INSTITUTO DE INVESTIGACIONES ELECTRICAS

2.0 EL PROCESO DE PROGRAMACION

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

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

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

Esta metodología prevee la división del proceso de desarrollo de programación en varias fases que siguen una

secuencia temporal y se describen, incluyendo sus objetivos en las secciones de este capítulo. A saber:

Definición de Requerimientos

Diseño

Desarrollo

Operación y Mantenimiento

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

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

2.1 Introducción: El Enfoque Sistémico

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

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

INSTITUTO DE INVESTIGACIONES ELECTRICAS

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

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

2.1.1 Dos Sistemas -

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

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

negativos.

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

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

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

distintas localidades, de su teléfono deberían salir [illegible] hilos.

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

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

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

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

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

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

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

2.1.2 El Enfoque Sistémico Y La Programación.

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

Cuál es el problema?.

Cuáles son las alternativas?.

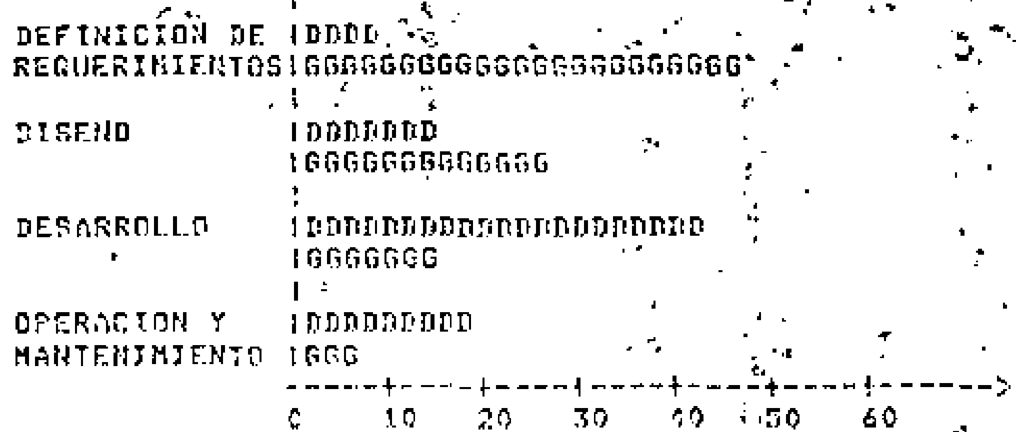
Cuál es la mejor?.

El objetivo de la metodología de sistemas es precisamente desarrollar programación con la misma filosofía con la que se diseñó el sistema telefónico. El costo cada vez creciente de programación hace necesario seguir un método sistémico para su construcción y no uno de prueba y error.

como venía realizándose hasta la fecha. La figura 2.1.1 muestra que el 55% de los errores en un programa son causados durante la fase de definición de requerimientos y el 50% de los errores aproximadamente son detectados durante el desarrollo.

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

FASES

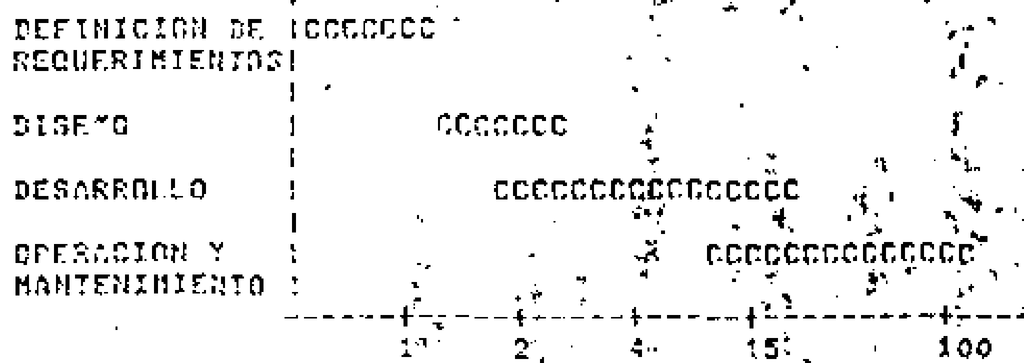


% DE PROBLEMAS

D.- DETECCION
G.- GENERACION

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

FASES



COSTO RELATIVO DE CORRECCION DE ERRORES

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

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

2.2 Definición de Requerimientos

2.3 Diseño

2.4 Desarrollo

2.5 Operación y Mantenimiento

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

2.2 Fase De Definición De Requerimientos.

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

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

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

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

1. Planteamiento de Objetivos

2. Análisis

3. Especificación de Requerimientos.

2.2.1 Planteamiento De Objetivos -

Esta etapa consiste en identificar y describir las necesidades del usuario, a fin de proponer un conjunto de

objetivos, que de lograrse, implicarían la satisfacción de las necesidades.

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

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

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

INSTITUTO DE INVESTIGACIONES ELECTRICAS

2.2.2 Análisis.-

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

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

Las actividades de esta etapa caen dentro del campo de acción de los analistas de sistemas. (véase capítulo 5). El grado de dificultad de esta etapa y el tiempo requerido para realizarla dependen directamente de la complejidad del proyecto o problema a resolver. El caso más sencillo lo representan aquellos problemas que no requieren de aproximaciones para obtener su solución, y el extremo más complejo lo representan aquellos problemas que, además de exigir algún tipo de aproximación o simulación, están relacionados con la mecanización de alguna actividad humana (LFH80). Algunas de las técnicas disponibles para realizar este análisis se presentan en la sección 3.2

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

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

2.2.3 Especificación De Requerimientos

Al terminar la etapa de Análisis se conocen las funciones del sistema y los métodos de solución que deberán ser empleados para realizarlas y satisfacer las necesidades del usuario. Esta información, reportada en los documentos DDP y RA, se redacta en un el lenguaje especializado, familiar a los analistas de sistemas (ingenieros electricistas, químicos, investigadores de operaciones, etc.) y no

necesariamente claro e inteligible a los especialistas en computación (diseñadores y programadores) que eventualmente diseñarán y desarrollarán el sistema de cómputo. Los resultados de la etapa de especificación de requerimientos se resumen en un documento de especificaciones, que denominaremos "Documento de Requerimientos de Programas de Computadora", (RPC), en el cual se traducen los resultados del Análisis a un idioma que permite la transferencia de ideas entre personal de distintas disciplinas. Este lenguaje está constituido por una combinación de prosa en lenguaje natural, con dibujos en algún lenguaje gráfico (diagramas de flujo de procesos, flujo de información, flujo de control), y notación matemática).

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

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

INSTITUTO DE INVESTIGACIONES ELECTRICAS

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

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

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

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

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

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

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

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

Requerimientos sólida y completa es una acción altamente recomendable.

INSTITUTO DE INVESTIGACIONES ELECTRICAS

2.3 La Fase De Diseño

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

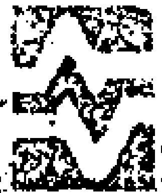
Los lenguajes de computadora son tan estrictos en su especificación que en lugar de afirmar que un programa esta escrito en un lenguaje se señala que está escrito en código. Los lenguajes naturales son muy flexibles, expresivamente poderosos y por lo tanto pueden ser ambiguos en el significado de sus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

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

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

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

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



1. Arquitectura.

2. Diagrama de estructura.

3. Detalle de módulos.

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

En la literatura, a las etapas del Diseño se les conoce con diferentes nombres; por ejemplo en [JET79], a la preparación de la arquitectura se le llama diseño del sistema, a la elaboración del diagrama de estructura se le conoce como diseño de la programación y al detalle de los módulos se le denomina diseño detallado.

2.3.1 Arquitectura -

Según el lineamiento fundamental del diseño de programas de arriba hacia abajo, los requerimientos tienen que dividirse en partes que resulten fáciles de entender, por lo tanto, es importante identificar las principales funciones que se

encuentran implícitos en los requerimientos a fin de diseñar un programa de computadora para cada una de las funciones. La arquitectura de programas de computadoras precisamente identifica esas funciones y las asocia a un programa.

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

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

2.3.2 El Diagrama De Estructura -

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

2.7.3 Detalle De Módulos. -

En esta etapa del diseño, se detalla el proceso o función que representa cada uno de los módulos del diagrama de estructura mediante lógica estructurada. La especificación del proceso de cada módulo es descrita, utilizando las técnicas de programación estructurada en un lenguaje de alto nivel denominado pseudo - código estructurado [LIN79], [DYC81], que permite independizar el diseño del lenguaje final de cómputo empleado por el sistema de explotación. La documentación del Diagrama de Estructura y el Detallado de los Módulos se denomina Diseño del Programa de Computadora (DPC).

2.8.1 Selección Del Lenguaje De Programación. -

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

Esta práctica debería abandonarse en favor de métodos más

formales que tomaran en cuenta las ventajas y desventajas técnicas de los posibles lenguajes disponibles. Desde luego, habrá situaciones en donde sólo exista la posibilidad de seleccionar un lenguaje, pero en muchas otras la lista de alternativas es amplia. En estos casos hay que tomar en cuenta varios factores para la selección del lenguaje. La literatura, en apoyo a la metodología de selección, es escasa. Sin embargo, existen algunas referencias interesantes. [ANDB2].

Anderson, Fujitson y Shumate describen los factores a tomar en cuenta en la selección de un lenguaje, y aunque los citan para el caso de lenguajes para microprocesadores, la metodología es, sin embargo, aplicable a la selección de un lenguaje para un tipo de aplicación menos restringida. El factor más importante a considerar es la minimización del costo de vida de la programación, incluyendo tanto el costo de desarrollo como el de mantenimiento, de acuerdo con la aplicación de la metodología de sistemas, citada al inicio de este Capítulo. Los autores mencionados recomiendan formar un grupo de estudio con personas que representen un amplio rango de experiencia en campos relacionados con la selección de un lenguaje. Estos campos de experiencia podrían ser: Programación de Sistemas, Mantenimiento de Programación, Equipo, Administración de Programación de aplicación, entre otros. Investigación de Operaciones, Definición de requerimientos. La formación de un grupo

INSTITUTO DE INVESTIGACIONES ELECTRICAS

evitar) en la selección de un lenguaje, los sesgos individuales que pudieran existir si la selección la hiciera una sola individuo y además permite incluir una gama más amplia de características técnicas, como resultado de las recomendaciones dadas por el grupo.

[ANDB[illegible]] propone una metodología de tres etapas para la selección del lenguaje más adecuado:

- Inicial
- Intermedia
- De análisis formal

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

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

Si ya se tiene seleccionado el procesador en que se va a

operar. La no disponibilidad de un compilador para un lenguaje específico, será un criterio lógico para su eliminación.

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

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

a) Tiempos de ejecución de la programación.

b) Diversidad de proveedores.

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

c) Similitud entre lenguajes.

Si existen varios lenguajes similares, otro criterio sería eliminar aquellos que tienen menos apoyo por parte

del vendedor o son más recientes, y por lo posiblemente todavía existen problemas con ellos.

d) Versatilidad

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

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

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

Entre los criterios mencionados en RAND 823 para la última etapa del proceso de selección pueden distinguirse criterios de tipo administrativos y de tipo técnico.

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

a) Tiempo y costo de desarrollo

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

b) Mantenimiento durante el ciclo de la vida.

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

Los factores técnicos pueden clasificarse como de:

- Primer orden
- Segundo orden

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

a) Entre los factores de primer orden destacan:

- Representación de datos.

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

- Estructuras de control.

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

- Programación de sistemas.

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

b) Entre los factores de segundo orden se cuentan:

- Transportabilidad:

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

- Facilidad de aprendizaje.

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

- Documentación.

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

- Eficiencia en tiempo.

  Esto se mide en el tiempo de ejecución de programas "tipo" (BENCH MARK)

- Eficiencia en espacio.

  Esto se mide por el tamaño del código objeto generado al compilar un programa "tipo".

- Facilidad de ligado con módulos de ensamblador.

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

Inteligibilidad.

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

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

2.4 La Fase De Desarrollo

Los objetivos de la fase de desarrollo son:

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

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

La fase de desarrollo se divide en tres etapas:

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

Los errores que se llegan a presentar en productos de

programación pueden ser clasificados conforme al siguiente criterio:

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

Este criterio de errores es útil para describir los tipos de pruebas que se practican en ingeniería de programación. Los errores de Codificación son identificados durante la etapa de integración [2.4.2.3]. Los restantes tipos de error son tema de la sección denominada Pruebas de Alto Nivel [2.4.3].

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

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

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

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

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

de Diseño de Casos de Prueba.

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

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

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

2.4.1 Codificación

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

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

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

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

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

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

- Conocimiento del usuario.

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

- Presiones de tiempo

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

- Calendarios precisos

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

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

- Codificación descendente (TOP-DOWN PROGRAMMING)

- Codificación ascendente (BOTTOM-UP PROGRAMMING)

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

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

como "Auditor de código". En la sección 4.3 se describe la función de un auditor de código estático.

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

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

Los productos de esta etapa del Proceso de Programación quedan plasmados en la documentación que se obtiene del código fuente generado (COD).

2.4.2 Integración.-

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

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

Pueden seguirse dos procedimientos:

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

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

Sobre estos procedimientos puede comentarse:

INSTITUTO DE INVESTIGACIONES ELECTRICAS

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

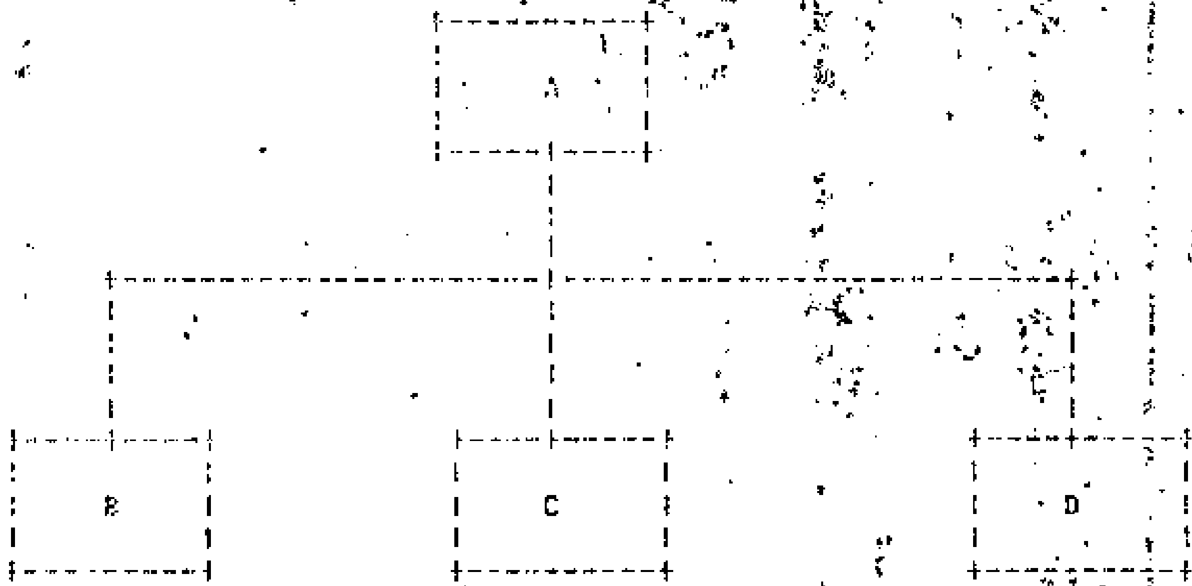


Fig. 2.4.1.
ESTRUCTURA DE UN SISTEMA EN PROCESO DE INTEGRACION

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

Al utilizar la técnica incremental,

INSTITUTO DE INVESTIGACIONES ELECTRICAS

esfuerzo, ya que por ejemplo, una vez probados individualmente los módulos "B", "C" y "D", podrán ser utilizados en la validación del módulo principal "A", evitando así el trabajo de elaborar los módulos ficticios "B", "C", y "D".

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

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

Dadas las tendencias en los sistemas de cómputo (costos decrecientes del equipo y crecientes en la programación) los

errores de programación contribuyen a engrosar los costos de programación; detectarlos oportunamente coadyuva para disminuir el precio de la programación. La variante no incremental favorece precisamente esta detección oportuna de errores, por lo cual sugerimos preferentemente su empleo.

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

2.4.3 Pruebas De Alto Nivel -

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

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

del usuario.

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

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

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

Se denominan 'Pruebas de alto nivel', a aquellas que tienen como objetivo identificar, no los errores de codificación ya encontrados en la etapa de integración, sino los de

análisis, especificación y diseño.

INSTITUTO DE INVESTIGACIONES

Entre estas pruebas las más importantes son las siguientes:

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

3.1.3.1 Pruebas Funcionales.-

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

Los casos para este tipo de prueba son generalmente producidos mediante técnicas de análisis de entrada-salida (sección 3.4). Para este tipo de técnicas la lógica interna del programa es irrelevante.

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

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

Durante esta etapa se producen basados en las normas descritas en 4.2.4 y 4.2.5, los documentos de planes y procedimientos de prueba (PLP) y (PRP).

2.1.3.7 Pruebas De Implantación.-

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

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

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

2.1.3.3 Pruebas De Sistema.-

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

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

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

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

veces, inclusive, su aceptación, por gentes escépticas al empleo de estos medios de manejo de información.

2.4.3.1 Pruebas De Aceptación :

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

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

2.5 Fase De Operación Y Mantenimiento

Esta última de las fases del proceso de programación es donde se reflejan los aciertos o los errores de las fases previas. Se identificarán aciertos en la medida en la que los requerimientos de los programas de computadora satisfacen las necesidades del usuario, la arquitectura y

los diseños se asocian a las características específicas del sistema de explotación de cómputo y en general, la disciplina que hubiere sido empleada para la construcción del código sea formal. Asimismo, gran parte del éxito de la programación radica en la metodología que se emplée para controlar los cambios que se tengan que realizar a los programas después de la primera instalación; controlar las diferentes versiones del código redunda en un beneficio inmediato para el usuario.

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

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

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

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

Todas estas observaciones señalan que la metodología utilizada en el desarrollo de esa programación fue deficiente; iniciando con los requerimientos, estos no satisfacen las necesidades del usuario, el no establecimiento de una documentación de interfaz entre los

diferentes programas del sistema de computo originan desajustes que eviten el éxito en la integración de la programación; una documentación deficiente y una administración no apropiada de la programación obligan a invertir grandes cantidades de recursos humanos para restaurar la programación.

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

2.5.1 Mantenimiento -

Dentro del campo de programación es necesario aclarar el término mantenimiento, que es generalmente utilizado para describir todos aquellos cambios hechos a la programación

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

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

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

a) Corrección de problemas latentes

b) Perfeccionamiento de programas y mejoras en la documentación.

c) Cambios adaptivos al medio ambiente de la programación.

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

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

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

2.3.1.1 Acciones Correctivas -

1. Probar cada instrucción y cada rama de programación durante la integración de los módulos (sección 3.4.2.).

2. Rastrear y verificar la demostración de todos los requerimientos en alguna etapa de las pruebas. Esto puede lograrse mediante una "Matriz de Requerimientos", (3.2.4) que válida dentro de los Planes de Prueba cada

uno de los requerimientos señalados en el documento.

2.5.1.2 Acciones Perfectivas.-

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

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

2.5.1.3 Acciones Adapti

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

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

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

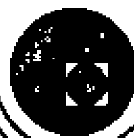
4. Mantener el mismo nivel de control en la base de datos y en el código; utilizar control automático y centralizado cuando ésto sea posible.

2.6 Resumen

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

Los párrafos iniciales resaltaron los altos costos de desarrollo y mantenimiento de la programación que el enfoque expuesto permite minimizar. Aún lográndolos abatir, éstos seguirán siendo altos, pero los beneficios económicos de las aplicaciones potenciales de las computadoras son tan altos, que estos niveles de costo son en muchos casos perfectamente aceptables en programación. Es de esperarse, sin embargo, que la investigación en el campo de desarrollo de programación (I.D.P.) siga no solo disminuyendo el costo relativo de su construcción, sino también permita que cada

vez satisfaga en mayor grado las características enunciadas al principio de este capítulo. A medida que la humanidad depende más y más de la programación que controla las computadoras y que como consecuencia controla la sociedad, se vuelve indispensable que los usuarios controlen absolutamente la programación y los procesos a través de los cuales se genera el código. Para lograr esto se requiere toda una teoría, modelos, metodologías, técnicas, herramientas; en fin una disciplina, vertida en el concepto de la ingeniería del proceso de programación.



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

TECNICAS MODERNAS DE DESARROLLO, ADMINISTRACION Y PROGRAMACION

TECNICAS PARA EL DESARROLLO DE PROGRAMAS DE COMPUTADORA

MARZO, 1983

Palacio de Minería Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

16 Feb 83

CAPITULO 3

3.0 TECNICAS PARA EL DESARROLLO DE PROGRAMAS DE COMPUTADORA

En el capítulo anterior de esta obra se señalo la necesidad de organizar el desarrollo de programas de computadora dividiendo el proceso de programación en cuatro fases. Este ordenamiento de actividades permite un refinamiento sistemático y progresivo de los productos de programación y una adecuada administración de los recursos humanos y financieros asociados.

En este capítulo se exponen las diferentes técnicas que emplean durante las diversas fases en el desarrollo sistémico del proceso de programación.

3.1 Introducción

Primeramente se exponen las técnicas sugeridas para una especificación técnica correcta, que cubra completamente los requerimientos y sea consistente con la definición de sus objetivos [3.2]. A continuación se proponen guías para abordar con éxito la fase de Diseño de los Porgramas de Computadora, permitiendo una subdivisión óptima de las diferentes funciones que debe realizar sistema de programación y una estructura adecuada de la información [3.3].

En [3.4] se exponen las técnicas recomendadas para desarrollar y construir programación. A medida que el proceso de programación avanza, se vuelve indispensable revisar los diseños [3.5], a fin de evitar al máximo incurrir en errores en las fases de desarrollo y operación de la programación, en donde el costo relativo de correción de errores es muy superior. Finalmente, en [3.6] se presentan las técnicas que permiten planear las pruebas a que serán sometidos los productos de programación a fin de demostrar que los requerimientos señalados por el usuario fueron satisfechos.

3.2 Técnicas De Definición De Requerimientos

En esta sección empezamos presentando las características que debe tener una adecuada especificación de requerimientos. Posteriormente describiremos como el empleo de modelos conceptuales adecuadas ayuda a establecer requerimientos en las características ya citadas. Continuamos señalando que estructura debe tener una documento de requerimientos y las tendencias que se observan en el desarrollo de lenguajes que faciliten el establecimiento de los requerimientos de un programa.

Son cuatro las principales características que una especificación de requerimientos debe poseer:

1. Debe indicar claramente las tareas que el sistema de cómputo propuesto realizará; incluyendo la interacción con otros sistemas (humanos y equipo), en términos tales que el usuario, el analista de sistema y el especialista en cómputo pueden comprender.

2. Debe definir el sistema propuesto en términos, lo suficientemente precisos y formales, que permitan verificar la correcta realización del producto una vez que éste haya sido terminado.

3. Debe definir el sistema en forma completa para propósitos de su desarrollo, cubriendo todos aquellos aspectos que no deben dejarse al libre albedrio del diseñador.

Por estas razones se dice que una especificación debe ser **inteligible, formal y completa** (YEH 80). A estas 3 caracteristicas es necesario agregar dos adicionales a saber:

4. El documento de especificaciones RPC debe ser estructurado a manera de permitir el uso de las partes terminadas mientras se desarrollan otras, tratándose además de minimizar el impacto de modificaciones sobre el documento.

1. El documento debe ser mantenible. En el caso de sistemas grandes, cuyos documentos RFC contienen más de 1000 páginas, el proceso de mantener la documentación actualizada, ante cambios que ocurren durante el desarrollo del sistema o durante su operación, se convierte en una actividad crítica. Es por esto que otra de las principales características del documento deberá ser su mantenibilidad.

Entre las principales deficiencias que con mayor frecuencia se presenta en documentos típicos de especificaciones se encuentran las siguientes (LEV 82):

a) Especificaciones infactibles ó erróneas.

b) Omisión de información clave.

c) Información ambigüa.

d) Información inintelegible.

e) Pobre presentación: distintos estilos a lo largo del documento.

f) Información obsoleta.

El empleo de una buena metodología durante la fase de especificación, asi como la adopción de una norma adecuada a la naturaleza del proyecto, ayuda a eliminar las deficiencias mencionadas.

La metodología de especificación puede definirse como un conjunto de técnicas apropiadas para la adquisición, análisis y presentación de información acerca de un sistema. Esta metodología debe cubrir los siguientes aspectos:

a) Identificación de los pasos que se deben seguir durante la fase de definición.

b) La forma de adquirir, analizar y verificar la información requerida sobre el sistema.

c) La forma de presentar información a traves de textos, notaciones y lenguajes especializados.

Una buena metodología aumenta la probabilidad de producir un RPC con las características resumidas en la tabla 3.2.1

1. Inteligible
2. Preciso y Formal
3. Completo
4. Actualizable y Modificable
5. Mantenible

TABLA 3.2.1 CARACTERISTICAS DE UNA ESPECIFICACION DE REQUERIMIENTOS

La norma de especificación puede definirse como un conjunto de reglas que gobiernan la elaboración del documento RPC y definen el criterio de aprobación de los documentos terminados.

La norma debe establecer la terminología, el grado de detalle, la estructura y el tipo de lenguaje permitido en el documento. La norma debe definir también el procedimiento para actualizar las partes terminadas del RPC. Es a través de la norma que se puede lograr un producto actualizado con buena presentación.

Trataremos a continuación los aspectos principales de la metodología deespecificación. Las normas de especificación de requerimientos serán objeto de la sección 4.2.1.

3.2.1 Modelos Conceptuales -

Definamos como un modelo conceptual de un sistema a un compendio de información sobre el sistema, que es colectado con el propósito de estudiarlo. Puesto que el propósito del estudio determina la naturaleza de la información que se colecta, no existe un modelo único por sistema. Se pueden elaborar diferentes modelos de un mismo sistema con el fin de estudiar distintos aspectos de éste.

En términos generales todo modelo conceptual de un sistema comprende los siguientes conceptos y las relaciones que

entre ellos existen (GOR 69):

a) Entidades y Atributos:

El término entidad es utilizado como sinónimo de componente en un sistema. El término atributo representa una propiedad o característica de entidades. Una entidad puede poseer varios atributos.

b) Proceso:

Proceso es sinónimo de toda acción que origine cambios en el sistema.

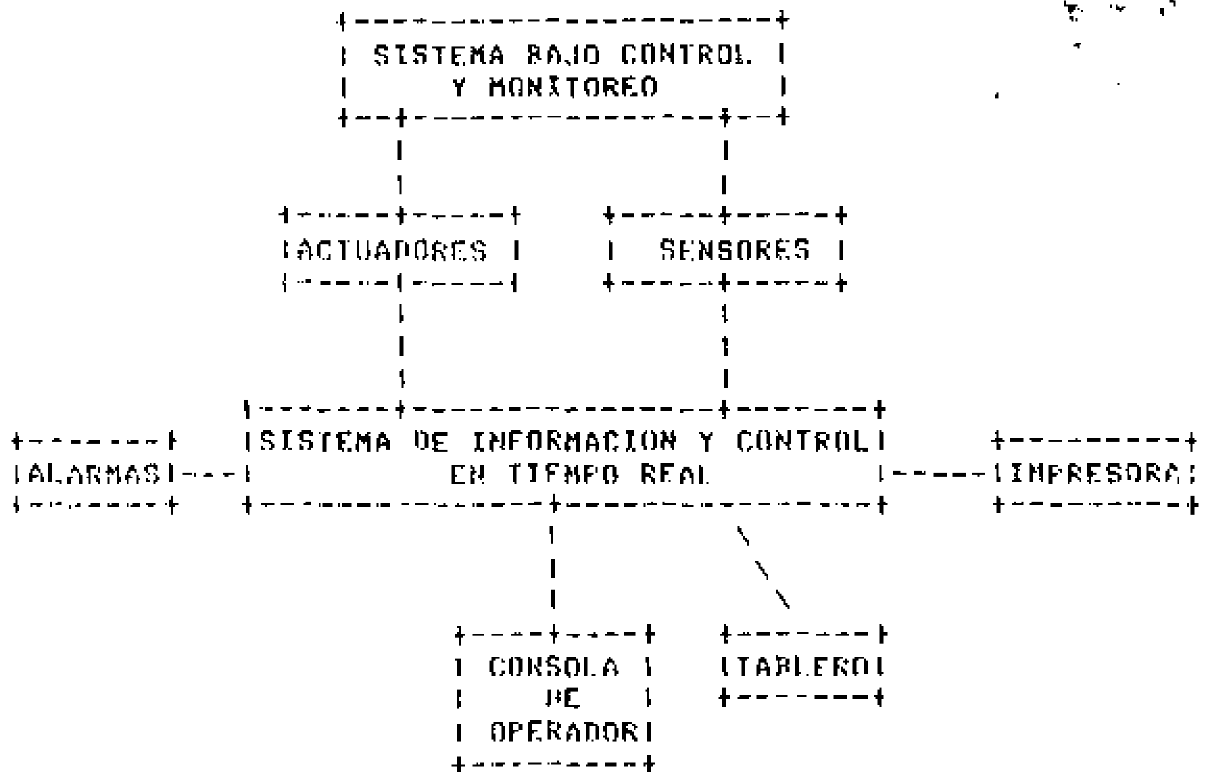


FIGURA 3.2.1 MODELO CONCEPTUAL DE UN SISTEMA DE INFORMACION Y CONTROL EN TIEMPO REAL.

c) Frontera y ambiente:

La frontera separa al sistema del ambiente que le rodea. La interacción entre el sistema y su ambiente ocurre a través de la frontera. Pueden existir procesos dentro del sistema que originan también cambios en el ambiente y viceversa.

Considérese por ejemplo el modelo conceptual de un sistema de control e información en tiempo real de la figura 3.2.1

Este modelo es general y podría representar, por ejemplo, un sistema de control de procesos en una refinería ó un sistema de control de sistemas eléctricos de potencia. En este caso el ambiente está formado por el sistema bajo control y por el operador del sistema. El ambiente interactúa con el sistema de control a través de la frontera, que se compone de las alarmas, la impresora, el tablero, la consola, los actuadores y los sensores.

Los sistemas que nos interesan en esta obra son dinámicos, su estado cambia, bajo la acción de diversos estímulos. A continuación definiremos estos términos:

1. Estímulos y Respuestas:

   Los estímulos representan la forma como un sistema es afectado por su ambiente, a través de la frontera, mientras que la respuesta, es la forma como el sistema afecta á su medio ambiente en su comportamiento observable. Estas acciones también se conocen con los nombres de factores exógenos y endógenos.

2. Estado

   El estado de un sistema es la descripción de todas sus entidades, atributos y procesos en un momento dado. Los

cambios que un sistema sufre durante un periodo dado se describen definiendo la evolución de los estados del sistema a lo largo del tiempo. Existen sistemas que poséen un número finito de estados (autómatas), pero también los hay que poséen un número infinito de ellos (sistemas continuos). El ejemplo más simple de los primeros es tal vez el caso de un interruptor, que en un momento dado puede encontrarse únicamente en uno de dos posiciones: encendido ó apagado. Un ejemplo de sistemas continuos, es el del termómetro común, cuya columna de mercurio puede prolongarse y tomar cualquier valor en un rango de varios centimetros dependiendo de la temperatura del medio ambiente.

Los procesos describen como cambia de estado y por lo tanto de respuesta un sistema ante el efecto de los estimulos. (veáse Fig. 3.2.2.) En el caso de sistemas de estados finitos los procesos se describen indicando, para cada estado, la lista completa de transiciones hacia otros estados, que pueden ocurrir ante todas las posibles combinaciones de distintos estimulos. En el caso de sistemas continuos, los procesos se describen con ecuaciones diferenciales.

Las respuestas son el comportamiento observable que un sistema exhibe cuando ha recibido y procesado un estimulo. No necesariamente, todos los elementos de un estado del

sistema son observables; las respuestas constituyen los componentes del estado que si lo son.

El formalismo de crear el modelo conceptual de un sistema, impone una disciplina en la manera de adquirir y analizar información relativa al sistema. Esta disciplina permite identificar los aspectos oscuros o desconocidos del sistema.

El modelo conceptual ayuda a entender la problemática del sistema y permite comprender las necesidades del usuario con respecto a éste.

El estudio del sistema a través de su modelo conceptual permite interpretar los objetivos que el usuario establece para el proyecto y es una herramienta necesaria para el análisis de su factibilidad.

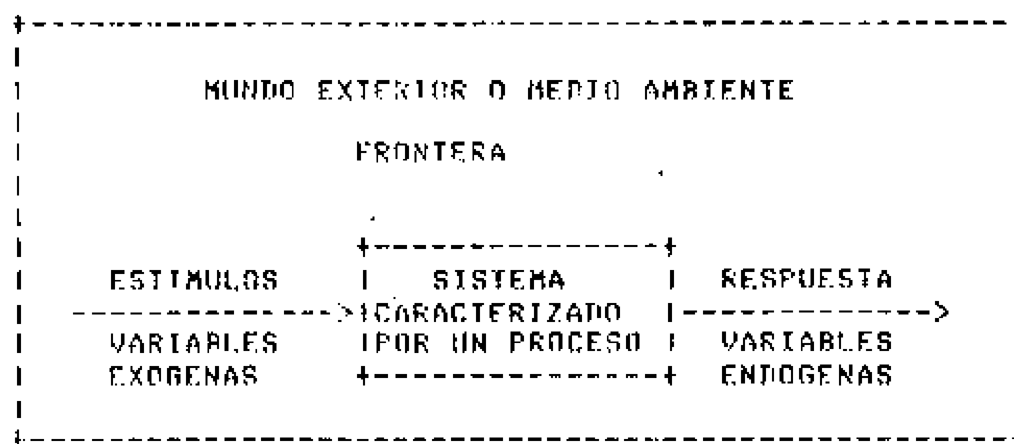


FIGURA 3.2.2. UN SISTEMA Y SU MEDIO AMBIENTE

El trabajo de crear un modelo conceptual consiste en establecer su estructura y suministrar sus datos. Estableciendo su estructura se dan a conocer sus entidades, atributos, procesos, frontera y ambiente. Suministrando sus datos se dan a conocer los valores de sus atributos y las relaciones que pueden ocurrir entre estímulos, procesos y respuestas.

3.2.2 Creación Y Refinamiento De Modelos Conceptuales -

La clave para cear modelos conceptuales de sistemas complejos radica en el empleo de técnicas de reducción y control de complejidad.

Los conceptos más utilizados para reducir complejidad son:

1. Abstracción:

   Consiste en estudiar el sistema en etapas sucesivas, incorporando más detalle y reduciendo el nivel de abstracción en cada nueva etapa. A este proceso también se le conoce como jerarquización o descomposición de arriba hacia abajo.

2. Partición:

Consiste en descomponer el sistema en un conjunto de subsistemas, permitiendo el estudio del sistema completo a través del estudio de sus partes y de las interacciones que ocurren entre ellas.

3. Relevancia:

Consiste en incluir en el modelo unicamente aquellos aspectos del sistema que son relevantes para los objetivos del estudio deseado.

4. Compactación:

consiste en agrupar entidades del sistema con atributos similares para formar entidades de carácter más general y reducir el número de entidades en el modelo.

5. Proyección:

Consiste en estudiar un sistema desde distintos puntos de vista. Cada punto de vista da origen a un modelo conceptual del sistema completo. Para ilustrar estos conceptos consideramos un ejemplo, se podría estudiar a través de 3 modelos distintos: uno desarrollado desde el punto de vista del usuario; para propósitos de especificación de requerimientos, otro modelo apropiado

para simulación de condiciones de operación, útil para el estudio de factibilidad, y otro con el punto de vista del diseño del sistema, útil para esclarecer las propiedades funcionales del sistema. Para ilustrar, considérese el diagrama jerárquico de la figura 3.2.3. El diagrama presenta la descomposición del sistema de información y control que es utilizado como ejemplo de modelo conceptual en la figura 3.2.1. La descomposición mostrada es el resultado de la aplicación sucesiva de los conceptos de abstracción y partición, habiéndose adoptado el punto de vista del usuario.

Decimos que se adopta el punto de vista del usuario cuando la estructura del modelo es apropiada para que el usuario exprese sus ideas, objetivos, necesidades y restricciones acerca del sistema que desea.

En nuestro ejemplo, el usuario desea un sistema que:

a) Adquiera información del sistema a controlar,

b) Ejecute acciones de control sobre el sistema, y

c) Le mantenga informado de la evolución del sistema, niveles de producción, consumo de recursos, etc.

Estos objetivos del usuario sientan las bases para el modelo conceptual de la figura 3.2.3.

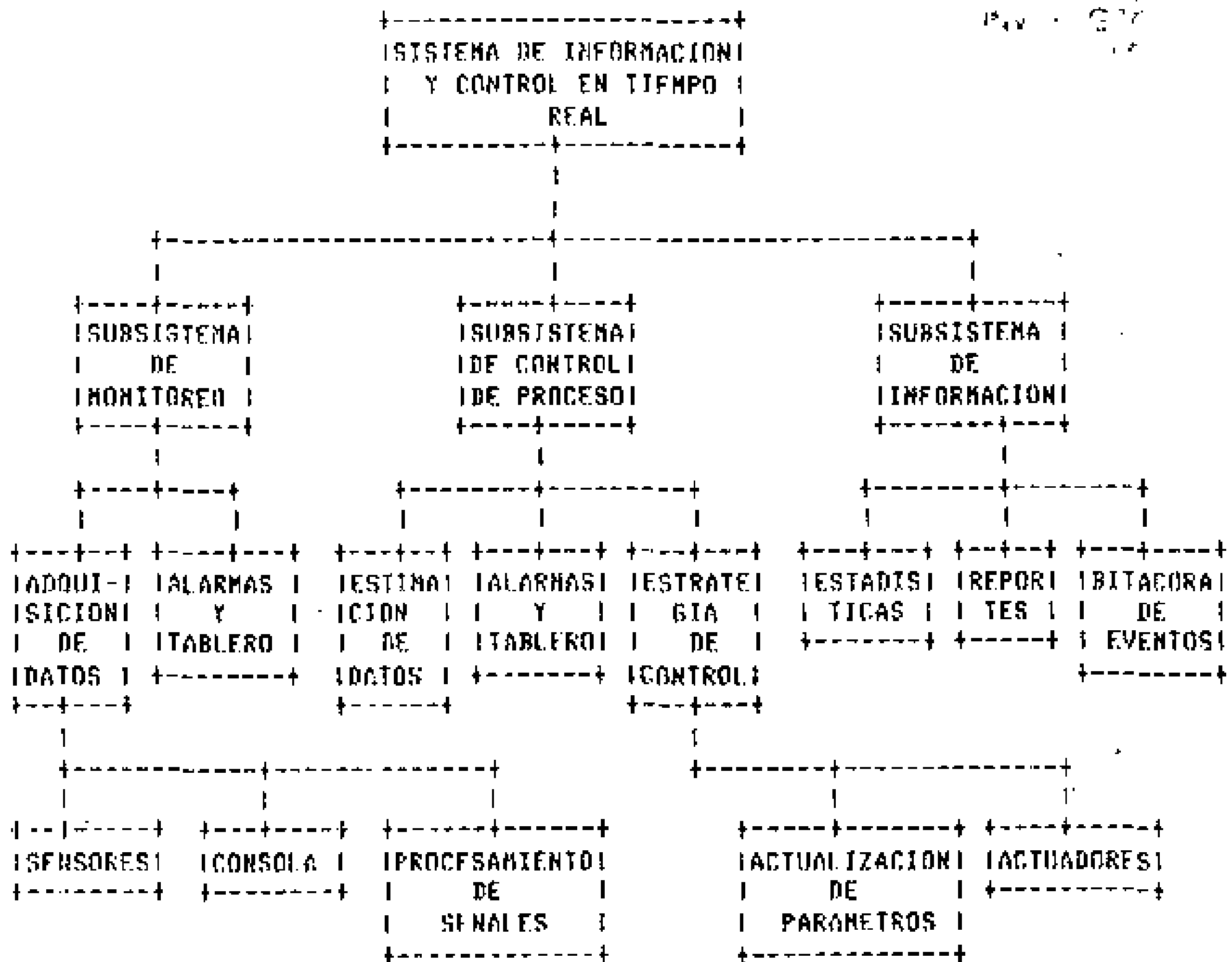


FIGURA 3.2.3 DESCOMPOSICION DE MODELO CONCEPTUAL DE UN SISTEMA DE INFORMACION Y CONTROL EN TIEMPO REAL: ENFOQUE DE USUARIO.

La figura 3.2.4 muestra un modelo conceptual del mismo sistema que resulta de la aplicación sucesiva de abstracción y partición, pero habiéndose adoptado el punto de vista del diseñador.

Decimos que se ha adoptado el punto de vista del diseñador cuando la estructura del modelo es apropiada para que el diseñador identifique las propiedades funcionales del sistema y explore las maneras de satisfacer los objetivos del proyecto.

En nuestro ejemplo, el diseñador está conciente que para satisfacer las necesidades del usuario el sistema deberá ser dotado de las siguientes entidades:

a) Base de datos

b) Monitor de adquisición de datos,

c) Procesador de alarmas y eventos,

d) Estimador de estados,

e) Controlador de proceso,

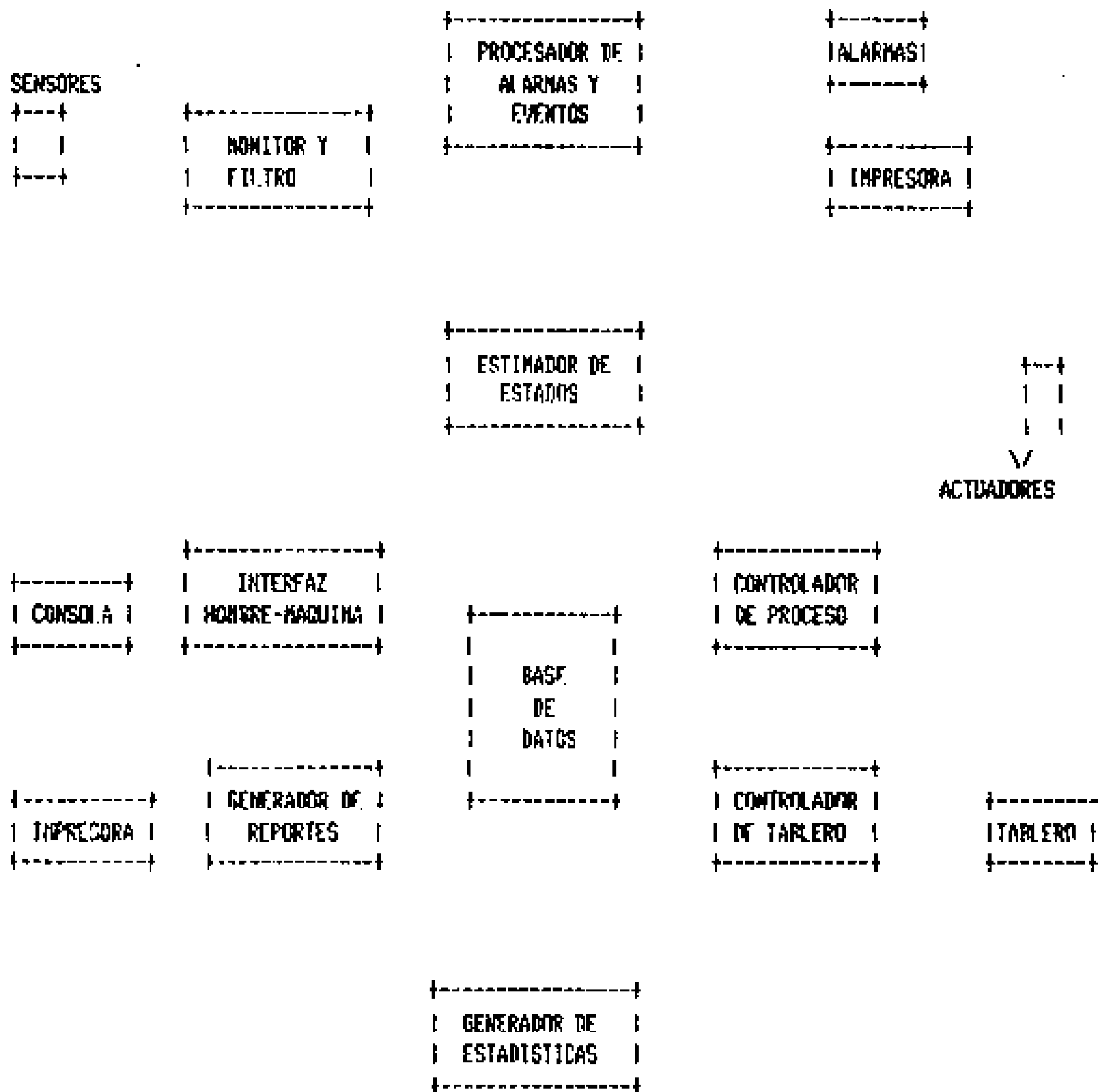
f) Generador de estadísticas,

g) Generador de reportes, y

h) Controlador del tablero.

Estas entidades fundamentan la estructura del modelo conceptual de la figura 3.2.4. La figura 3.2.5. nos muestra una versión más detallada del modelo, donde se clasifica la información de base de datos y se indica la

forma como ésta es compartida entre las distintas entidades.



FLUJO DE INFORMACION
SENAL DE ACTIVACION

FIGURA 3.2.4 MODELO CONCEPTUAL DE SISTEMA DE INFORMACION Y CONTROL EN TIEMPO REAL: ENFOQUE DE DISENADOR

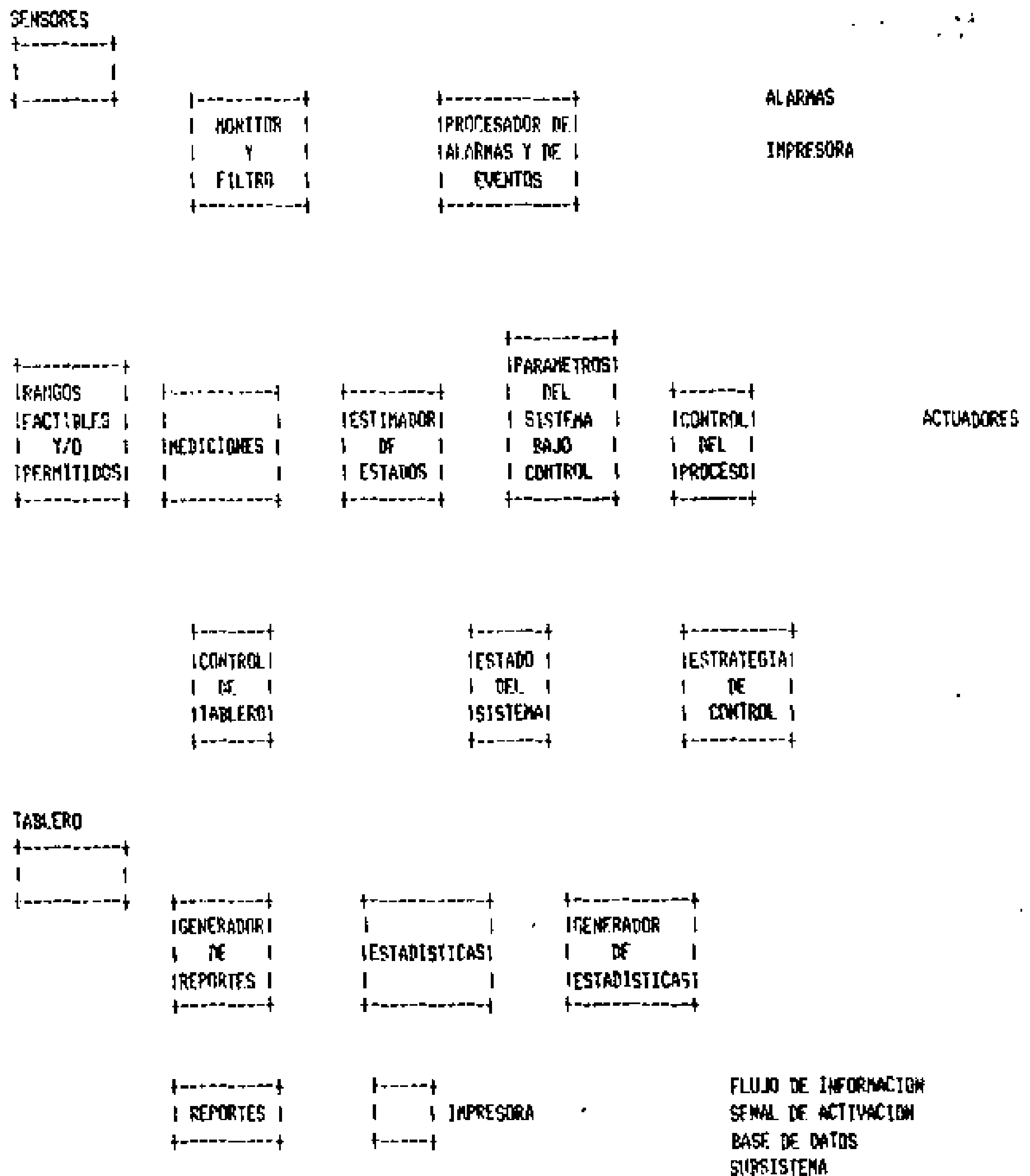


FIGURA 3.2.5 MODELO CONCEPTUAL DE SISTEMA DE INFORMACION Y CONTROL EN TIEMPO REAL: ENFOQUE DE DISENADOR

Los ejemplos que hemos mostrado en esta sección son concepciones muy abstractas de un sistema. Es necesario incorporar mucho más detalle a los modelos para que éstos lleguen a describir el sistema de información y control en forma completa. El beneficio que aportan estos modelos de alto nivel de abstracción es que nos permiten organizar la manera de adquirir más información sobre el sistema; nos ayudan a identificar los pasos necesarios para añadir más detalle y así continuar con la creación del modelo conceptual hasta completarlo.

3.2.3 Modelos Conceptuales En La Fase De Definición - de Requerimientos.

Hemos dicho ya que la fase de definición de requerimientos se realiza en tres etapas:

a. Planteamiento de objetivos

b. Análisis

c. Especificación de requerimientos.

Consideramos conveniente ahora comentar sobre el enfoque que se da a los modelos conceptuales en las distintas etapas de esta fase.

El punto de vista del usuario es especialmente apropiado para la etapa de planteamiento de objetivos. Un modelo conceptual con este enfoque permite la comunicación entre el analista y el usuario en los términos que el usuario conoce. La estructura del modelo sirve de guía al analista en su labor de adquisición de información sobre el sistema. Tomemos el caso del sistema de información y control en tiempo real; el analista en este caso podría, por ejemplo, explorar las necesidades y plantear los objetivos del usuario con respecto al tema de monitoreo siguiendo la rama de la izquierda en la estructura de la figura 3.2.3. Al concentrarse el usuario en éste aspecto del problema, ignorando temporalmente el resto, le permite dominar la complejidad. Sin embargo, este enfoque no es necesariamente útil para el análisis de factibilidad de objetivos o para establecer prioridades entre objetivos conflictivos, que son metas de la etapa de análisis. Consideremos el siguiente escenario. Supongamos que el usuario ha establecido para nuestro ejemplo los siguientes objetivos:

Objetivo 1. "El sistema de control deberá efectuar la

estrategia óptima de control para manejar la situación X en menos de T segundos. El criterio de optimalidad es Y las restricciones que deberán tomarse en cuenta son W".

Objetivo 2. "El sistema de información y control deberá ser integrado en un computador tipo Z".

Es evidente que estos objetivos son potencialmente conflictivos. Para esclarecerlo el analista sintetizaría la estrategia óptima de control, apoyándose en un modelo de enfoque matemático capaz de reproducir el comportamiento del sistema a controlar. Puesto que varias funciones del sistema de control competirán simultaneamente por los recursos del computador, el analista crearía un modelo con enfoque de diseñador para identificar todas las funciones que pudieran llegarse a ejecutar en un momento dado. Finalmente, para establecer la factibilidad de ejecutar la acción de control del objetivo 1, en menos de T segundos, y con los recursos del computador Z, el analista realizaría simulaciones apoyándose en un modelo conceptual distinto del sistema, posiblemente de naturaleza probabilística.

La moraleja de este simple escenario es que el concepto de proyección se aplica en su máxima expresión en la etapa de análisis. El análisis se apoyará en distintos modelos del mismo sistema. Resulta obvio que el número de modelos y los distintos enfoques que se adopten en esta etapa dependerán de la naturaleza del proyecto.

El análisis de un sistema de información gerencial se realiza con enfoques que difieren de los puntos de vista adoptados para el análisis de un sistema de control de reactores nucleares. Cabe enfatizar sin embargo, que el análisis de ambos sistemas se realiza en términos de modelos conceptuales, que son creados con los mismos conceptos de reducción de complejidad.

ETAPA	TIPO DE ENFOQUE EN EL MODELO
PLANTEAMIENTO DE OBJETIVOS	CON ENFOQUE DE USUARIOS
ANALISIS	CON ENFOQUE MATEMATICO Y/O DE DISENADOR

LA TABLA 3.2.2 RESUME LOS COMENTARIOS ANTERIORES

Tabla 3.2.2 Modelos más frecuentemente empleados en las etapas que preceden a la especificación de requerimientos.

3.2.4 Estructura De Los Documentos De La Fase De - Definición de Requerimientos.

La organización de los documentos de objetivos (DO) y documento de requerimientos (RPC) se deriva de la estructura de los modelos conceptuales utilizados para estudiar al sistema.

Si los modelos conceptuales constituyen una herramienta útil para que los análistas estudien y comprendan las características de un sistema, es evidente que también apoyan a los analistas en su labor de describir las características del sistema a otras personas, en particular

a aquellos que diseñarán y desarrollarán los programas del sistema.

Así por ejemplo, un documento de objetivos del sistema de información y control en tiempo real podría ser estructurado de acuerdo al modelo conceptual con el enfoque de usuario de la figura 3.2.3. Se podría describir cada uno de los bloques del modelo en secciones o subsecciones distintas del documento.

Ambos documentos, DO y RPC cubren en su contenido los siguientes aspectos:

a) El uso que se pretende dar al sistema por desarrollar

b) La descripción del sistema y su ambiente en base a uno o varios modelos conceptuales.

c) Consideraciones sobre tiempo de entrega y costo de desarrollo, y

d) Consideraciones sobre criterios de aceptación del producto.

La diferencia entre los documentos DO y RPC radica en el grado de detalle, en el enfoque de los modelos conceptuales y en el hecho de que los requerimientos del DO no pasan de

ser meros objetivos mientras que en el RPC los requerimientos representan características que deberán ser plasmadas en el producto final.

Así por ejemplo las consideraciones de costo y tiempo de entrega en el DO podrán señalar los límites superiores permitidos, mientras que en el RPC las mismas consideraciones establecen un compromiso, costo y tiempo de entrega, que forzosamente deberá satisfacerse.

El DO representa la iniciación de un proyecto, mientras que el RPC permite tomar la decisión de continuarlo y seguir con el diseño del sistema.

Desde el punto de vista del contenido, podría decirse que el RPC constituye una versión corregida, actualizada y aumentada del DO. Es importante aclarar que no siempre es necesario elaborar un RPC basándose en modelos cuyo enfoque sea distinto al que sirve de base para el DO. Podrá darse el caso de que una nueva versión del DO, resultado de haberse corregido los objetivos infactibles y de haber asignados las prioridades a objetivos antagónicos, pudiera servir de RPC sin ser indispensable recurrir a una estructura y enfoque distintos. Este es particularmente el caso cuando tanto el usuario, como los analistas y los diseñadores dominan el tema del proyecto y se comunican

fácilmente en los mismos términos. El caso contrario ocurre cuando el usuario es exponente de una disciplina cuyos fundamentos, conocimientos y metodologías difieren mucho de las áreas del conocimiento que dominan quienes desarrollarán el sistema, como podría suceder en un ambiente científico o de aplicación especializado de las ramas de la ingeniería.

La organización del RPC en base a un modelo con enfoque de diseñador permite al analista describir un sistema tratando de contestar las preguntas que un diseñador se haría con respecto a las especificaciones del sistema. Es por esta razón que el RPC es también conocido como "ESPECIFICACION FUNCIONAL" o "ESPECIFICACION DEL SISTEMA".

La adopción de este enfoque para elaborar el RPC tiene la ventaja de producir especificaciones del sistema de una manera estructurada, que es para el diseñador fácil de entender. Quien realice el diseño puede, con relativa facilidad, juzgar la completez del RPC, verificando si sus dudas respecto a aquello que se debe diseñar son resueltas.

Sin embargo, quien adopte este enfoque debe considerar que necesariamente su RPC ejercerá un cierto grado de influencia sobre el diseñador del sistema y deberá evitar restringir demasiado el campo de acción de éste. Podría ser que una especificación bien estructurada y clara, con enfoque de diseñador, permitirá la producción de un diseño correcto en

corto tiempo, lo cual representará evidentemente una ventaja.

Sin embargo, se corre el peligro de escribir el RPC de una manera que confunde aquello que el sistema deberá ser capaz de realizar, con la forma de implementar el sistema.

El alcance del RPC debe limitarse a establecer los requerimientos del sistema que deberán diseñarse.

Los requerimientos establecen las características que el sistema deberá poseer y la descripción de los aspectos, las tareas o funciones que el sistema deberá realizar.

Se prohibe establecer en el RPC el número y características de los programas que constituirán al sistema.

Es en la etapa de arquitectura de la fase de Diseño en donde, con el fin de aprovechar bién los recursos de cómputo, se identifican los programas necesarios para realizar las tareas del sistema. Bien pudiera suceder que en una instancia varias tareas del sistema sean realizadas por un mismo programa, mientras que en otros casos puede resultar conveniente diseñar varios programas para que en conjunto realicen una sola tarea del sistema.

3.2.5 Formalismo Y Lenguajes De Especificación -

La forma tradicional de especificar requerimientos consiste en describir las características del sistema a desarrollar, en un lenguaje natural y apoyándose en el uso de diagramas de proceso y de flujo de información.

Se ha generado recientemente una tendencia al uso de lenguajes formales de especificación, dotados de una sintaxis y una semántica bién definida, que con su notación precisa y restrictiva eliminan ambigüedad y permiten la verificación automática de consistencia en las especificaciones. El gran beneficio que aporta el formalismo de estos lenguajes es que permiten el uso de herramientas automáticas para almacenar y procesar la información clave de las especificaciones. Las herramientas automáticas son utilizadas para las siguientes funciones:

a) Asegurar consistencia en las especificaciones, verificando por ejemplo, que no exista tarea alguna cuya información de entrada no sea generada por otra tarea o por el ambiente.

b) Asegurar la completéz del diseño, verificando al terminar la fase de diseño, que todos los requerimientos hayan sido considerados.

c) Asegurar la correcta realización del sistema, generando automáticamente casos de prueba que permitan verificar la satisfacción de requerimientos.

Son muchos los lenguajes de especificación que se han desarrollado y son variados los enfoques utilizados en su diseño. Algunos fueron diseñados para uso especializado en proyectos de sistemas de información, como es el caso del PSL(TEI 77). Otros fueron desarrollados para uso especializado en sistemas de tiempo real como lo son EXPRESO (LUD82) y EPOS-R (LAV82).

Si bién los lenguajes de especificación han sido utilizados por varios años en proyectos de gran envergadura en, la vasta mayoría de los proyectos de computación no se emplean todavía, debido a que la escritura y lectura de especificaciones desarrolladas con estos lenguajes exige una orientación y preparación en ciencias computacionales, que los usuarios y los analistas de sistemas generalmente no poseen.

La desventaja de un formalismo acentuado es la dificultad que impone a la comunicación entre personal de distintas disciplinas. El formalismo reduce legibilidad en favor de precisión.

La tendencia actual en el desarrollo de especificaciones de

requerimientos pretende lograr un compromiso entre un formalismo acentuado, que es difícil de usar y leer, y un lenguaje natural que es fácil de usar y leer pero que da lugar a ambigüedad e inconsistencia.

Se pretende que la notación sea suficientemente rigurosa para que permita el procesamiento automático de especificaciones, y a la vez suficientemente legible, para que permita la comunicación entre usuario, analista y diseñador (CAS81).

3.3 Técnicas De Diseño

En esta sección se describen las técnicas de diseño más conocidas, documentadas en la literatura y aplicadas en la práctica. Inicialmente se discutirán los conceptos usados en el diseño de la Arquitectura, para después presentar las principales técnicas de Diseño de Diagramas de Estructura. Por último, se incluye la etapa del detalle de los Módulos.

3.3.1 La Arquitectura -

Como se mencionó en la sección 2.3, la arquitectura es conocida también por otros nombres. En [JEN79] se le llama

'diseño del sistema' y en [TEI77] se identificó como 'requerimientos del diseño del sistema'. Sin embargo, todos coinciden en cuáles conceptos deberán quedar reflejados en una arquitectura correcta y en el nivel de detalle con que deben ser descritos.

Existen herramientas, paquetes de computadora, útiles para el diseño de programas. Entre estos paquetes están el PSL/PSA [TEI77]. Conservar actualizado todo el diseño a través de uno de estos paquetes permite la producción de reportes actualizados del estado del diseño y el acceso en forma interactiva al diseño, facilitando la toma de decisiones. Si no se dispone de un paquete para automatizar el mantenimiento de la Arquitectura, se deberán producir los documentos en forma de reportes formales a la administración del proyecto.

La parte medular del diseño de la arquitectura es un diagrama de flujo del sistema que indica por medio de gráficos la interacción de las diferentes partes de un sistema: la base de datos y sus archivos, los programas y los operadores, (ver por ejemplo [GAN79]).

El diseño de la Arquitectura puede dividirse en las siguientes actividades:

1. Definición de conjuntos de información

2. Identificación de los procesos o programas a desarrollar

3. Definición del comportamiento dinámico.

4. Definición de las interacciónes organizacionales

5. Definición de las responsabilidades

A continuación se describen estas actividades

3.3.1.1 Conjuntos De Información -

Los conjuntos de información se clasifican en tres tipos:

1. De entrada

Los conjuntos de datos de entrada son todos aquellos elementos de información necesarios para la correcta operación de la programación y que usualmente son capturados a través de los diferentes dispositivos de lectura de los sistemas de cómputo, como unidades de cinta magnética, terminales de video, lectoras de tarjetas, etc.

2. De salida

Entre los conjuntos de datos de salida se incluye todo tipo de información producida por el sistema como cheques impresos, reportes, sobres con direcciones, gráficas, etc.

3. De base de datos

La base datos incluye toda la información que es internamente producida, mantenida y usada por el sistema.

Para la apertura de datos de entrada y el despliegue de los de salida se emplea en general paquetes de diseño de formas de menús, despliegues de fantasía, generadores de reportes y lenguajes de interrogación. Estos paquetes en general vienen con el sistema de cómputo. Siendo la base de datos, el lugar de concentración de la información que maneja el sistema, su diseño adecuado es uno de los factores más importante en el desarrollo de un sistema de cómputo. Ha sido ampliamente documentado en la literatura [ATR80] por ello solo resumiremos los principales pasos:

1. Diseño conceptual

El objetivo del diseño conceptual de la base de datos es obtener un modelo de las necesidades de información, independientemente de las características del sistema de cómputo. El modelo conceptual de una base de datos es un enunciado estructurado de las entidades que deben almacenarse y de las relaciones que existen entre ellas [ROD81 y ROD83].

2. Diseño lógico

En el diseño lógico de la base de datos, las entidades y relaciones del modelo conceptual se representan utilizando las estructuras de un sistema particular de administración de datos (por ejemplo: el IMS, el TOTAL, el DMSII, etc.) o las estructuras de un sistema de archivos.

3. Diseño físico

En el diseño físico de la base de datos se hacen estudios de los tiempos de acceso, para ver si corresponden a los requerimientos de tiempo de respuesta especificados por el usuario. Para mejorar los tiempos de acceso se pueden utilizar

estructuras de más bajo nivel que los que se utilizan en el nivel lógico, o bien, estructuras lógicas alternativas con redundancia [ATR80].

Las dos primeras etapas son siempre necesarias, siendo el diseño físico a veces no necesario, dependiendo del tipo de sistema de administración de base de datos o del tipo de sistema de archivos con que se cuente.

3.3.1.2 La Identificación De Programas -

La siguiente actividad durante el desarrollo de la arquitectura de un sistema consiste en identificar el número de programas o procesos a desarrollar, indicando las interfaces con las que interaccionan. El resultado de la definición de estos programas se puede documentar utilizando diagramas de flujo de información; en ellos se especifica, para cada programa, los conjuntos de información con que interactuará: las entradas, las salidas y la base de datos.

También, y como parte del diseño de la arquitectura, se debe incluir una explicación breve de las funciones de cada programa, sin especificar la descomposición del programa en módulos, ya que el diseño del diagrama de estructura forma parte de otra etapa del diseño.

3.3.1.3 El Comportamiento Dinámico Del Sistema -

La definición del comportamiento dinámico consiste en especificar los eventos y condiciones que afectan al sistema. Los eventos describen hechos que pueden ocurrir durante la operación del sistema. Las condiciones son enunciados que al cambiar su estado de falso a verdadero o viceversa, causan que un evento suceda. Por ejemplo, la condición: 'tarjetas listas' especifica que las tarjetas pueden ser procesadas. El evento: 'inicio del reporte de contabilidad' inicia o activa el programa que producirá el reporte de contabilidad; es necesario indicar si el evento es periódico o si es producido por algún otro programa o directamente por interacción humana.

3.3.1.4 Las Interacciones Organizacionales. -

Las interacciones organizacionales con el sistema describen las responsabilidades del medio ambiente con respecto al sistema. Se define quién proporciona la información de entrada y a donde va la salida del sistema; por ejemplo: el departamento de Personal será el responsable de proporcionar la información de los empleados al sistema utilizando los menús apropiados; el departamento de Contabilidad será el responsable de recibir los reportes de contabilidad producidos por el sistema; etc.

3.3.1.5 Las Responsabilidades -

Por último, la Arquitectura de un sistema deberá incluir la definición de los responsables de diversas actividades del diseño, los nombres de las personas y su responsabilidad. Por ejemplo, se deberá indicar quien es responsable del diseño conceptual de la base de datos, quien es responsable del diseño de cada uno de los programas, etc. con el objeto de poder dirigir preguntas a la persona adecuada.

3.3.2 El Diagrama De Estructura -

Para cada programa indentificado al establecer la arquitectura se deberá construir un Diagrama de Estructura, que representa la subdivisión jerárquica y por funciones de un programa. A cada elemento del Diagrama de Estructura le llamaremos Módulo.

La descomposición de un programa es una aplicación del principio clásico de solución de problemas: "divide y conquista". Pero inmediatamente surge la pregunta ¿Qué tan fina debe ser la subdivisión ?. Es práctica común, documentada en la literatura [JEN79], que un módulo resulte aproximadamente de no más de 50 líneas de código ejecutable. Esto quiere decir que un buen diagrama de estructura debe proponer módulos que sean relativamente fáciles de

desarrollar.Módulos con gran cantidad de líneas de código obscurecen los objetivos y complican el mantenimiento.'

Antes de exponer las técnicas para romper o descomponer un programa en módulos, presentamos a continuación una serie de criterios para evaluar si un diagrama de estructura ha sido bien diseñado, a saber:

1. Tamaño

2. Acoplamiento

3. Cohesión

4. Parsimonia

3.3.2.1 Tamaño -

Debe ser la meta de un buen diseño del diagrama de estructura [JEN79] lograr que un módulo no tenga mas de 50 líneas de código ejecutable, ni menos de 30. Si algún módulo excede 50 líneas, probablemente debería de descomponerse en más módulos. Sin embargo se pueden tener módulos con mas de 50 líneas pero con una estructura muy fácil de entender. Si un módulo es muy pequeño probablemente su función pueda incorporarse a otro. Cabe aclarar que no pueden establecerse normas que definan el

tamaño de los módulos. Este se debe dejar a juicio del diseñador y de los que revisan el diseño.

3.3.2.2 Acoplamiento -

Acoplamiento es la medida de interdependencia entre módulos; alta interdependencia de módulos refleja que los módulos son poco independientes, y por lo tanto, cuando un módulo es modificado, los módulos vecinos también tendrán que cambiar volviendo difícil el mantenimiento de un programa. Un módulo fuertemente acoplado a otros no es transportable, es decir, es más difícil de usar en otros programas ya que, para ser empleado requerirá de cambios en su código.

Un módulo es independiente cuando se puede usar en otros programas sin necesidad de cambiar su codificación. Un ejemplo clásico de módulos independientes son las subrutinas que existen en la mayoría de los lenguajes de programación como son las rutinas para calcular la raíz cuadrada ó el seno de un ángulo, entre otros.

Aún cuando la independencia de los módulos es altamente deseable, a veces no es posible lograrlo debido a la influencia de otros factores en el diseño o por la misma naturaleza de los módulos. Por eso, Yourdon y Constantine [YOU75], han creado una clasificación del acoplamiento entre

módulos para poder evaluar el diseño de Diagramas de Estructura.

1. Acoplamiento por argumentos

Cuando se tiene este tipo de acoplamiento, la comunicación entre módulos se logra exclusivamente a través de argumentos. Este tipo de acoplamiento es inevitable puesto que casi siempre los módulos tienen argumentos. Sin embargo la idea es mantener el número de argumentos al mínimo posible.

Ejemplo.

El siguiente módulo tiene cinco argumentos:

```
SUBROUTINE MOD023( A, B, M, N, L)
```

Si el número de argumentos puede ser menor, el acoplamiento de este módulo con el programa que lo use se reducirá.

2. Acoplamiento por estampado

Este acoplamiento resulta cuando uno o más módulos usan como argumento la misma estructura de datos. Por ejemplo, se tienen varios módulos: un módulo que calcula impuestos y usa como argumento el registro de un empleado; otro módulo

cálcula la prima de vacaciones y también usa como argumento el registro del empleado. Cuando la estructura del registro de empleados cambie, todos los módulos que usan el registro de empleados tendrán que revisarse para posibles cambios.

Por ejemplo, el registro de empleados puede especificarse en el lenguaje PL/I como:

```
DECLARE 1 EMPLEADO
          2 NUMERO CHAR (5)
          2 NOMBRE CHAR (20)
          2 SALARIO FIXED DECIMAL (7,2)
          2 FECHA-INGRESO
            3 MM      FIXED DECIMAL (2)
            3 DD      FIXED DECIMAL (2)
            3 AA      FIXED DECIMAL (2)
```

Los siguentes módulos usan como argumento el registro EMPLEADO de la siguiente forma,

```
MODULO32:  PROCEDURE (EMPLEADO, IMPUESTO)

MODULO45:  PROCEDURE (EMPLEADO, PRIMA)
```

Sin embargo, se observa que MODULO32 sólo necesita el salario y que MODULO45 sólo necesita la fecha, por lo tanto, para evitar acoplamiento por estampado, los argumentos de estos módulos deberían de ser como sigue:

MODULO32: PROCEDURE(SALARIO, IMPUESTO)

MODULO45: PROCEDURE(MM, DD, AA, PRIMA)

Para evitar el acoplamiento por estampado se recomienda no pasar como argumento estructuras completas de datos, a menos que el módulo las use completamente. A los módulos habrá que transferirles solamente los datos indispensables para efectuar su función, ni más, ni menos.

3. Acoplamiento por control

El acoplamiento por control ocurre cuando un módulo influye en la ejecución de otro módulo a través de un código o de una bandera. Esto quiere decir que el módulo que pasa el control (padre) dirige la ejecución del módulo llamado (hijo). Para evitar este tipo de acoplamiento se tiene que dividir el módulo controlado en varios módulos: uno para cada función. De esta forma, cuando haya cambios en el número de funciones, los cambios afectarán solo el módulo "padre" y no los módulos "hijos".

Una forma de acoplamiento híbrido que combina el acoplamiento por control y el acoplamiento por datos se tiene cuando el control es parte del dato. Por ejemplo, el caso del número del empleado, donde los dos primeros dígitos

pudierán representar el número del departamento, etc., Esto provoca que la estructura de los datos se limite en su evolución y por otro lado, se produce un acoplamiento por estampado junto con un acoplamiento por control.

4. Acoplamiento por datos externos

Este acoplamiento existe cuando dos o más módulos usan la misma estructura de datos: uno para leer, otro para escribir y otro para modificar una base de datos o un archivo. Un cambio en la estructura de los datos externos va a provocar que el código de los módulos también cambie. El acoplamiento por datos externos es de los más difíciles de evitar. El diseñador busca por un lado, módulos que realicen una sola función; por el otro tiene el problema de que esos módulos emplean una estructura común de datos externos. Como no se puede evitar que el acoplamiento por datos externos se presente, se recomienda tener un buen diccionario de datos para saber qué módulos se ven afectados cuando se modifica una de las estructuras de datos externos.

5. Acoplamiento por datos globales

El acoplamiento por datos globales se presenta cuando un grupo de módulos en un programa comparten una área global de memoria, por ejemplo, cuando se usa el 'COMMON' del lenguaje

FORTRAN, el 'EXTERNAL'de PL/I y el'DATA DIVISION' del COBOL. Un problema con este tipo de acoplamiento es que todos los módulos tienen que usar los nombres que aparecen en los 'COMMONS' y por lo tanto, el uso de éstos módulos es limitado en otros programas. Un cambio en esta región común obliga al menos a re-compilar los módulos.

En FORTRAN por ejemplo, la posición de los datos en el COMMON es importante. Fuerza la aparición de una extraña versión de acoplamiento por estampado haciendo más difícil encontrar errores, ya que los datos en al área global no pertenecen a un módulo en particular. En este caso se recomienda evitar el uso de variables globales. Si esto no se puede evitar se sugiere no usar COMMON's 'blancos' y emplear COMMON's 'etiquetados' con una sola variable por etiqueta. Esto permitirá que el acoplamiento entre módulos se limite a las variables que cada módulo use, por ejemplo, es más conveniente.

```
COMMON / A      / A ( N )
COMMON / B      / B ( M )
COMMON / C      / C ( N )
```

que declarar:

```
COMMON /ABC/ A(N),B(M),C(N)
```

Puesto que los módulos o subrutinas que usan la variable A, solo tienen que declarar 'COMMON /A / A(N)', cuando ocurra un cambio en las variables B y C, estos módulos no serán afectados ni tendrán que ser re-compilados.

6. Acoplamiento por contenido

El acoplamiento por contenido no se puede lograr en lenguajes como el FORTRAN o el COBOL. Solo en lenguajes como el Ensamblador, en donde un módulo puede modificar la secuencia de instrucciones de otro módulo. Si se presenta la necesidad de programar en Ensamblador se recomienda evitar esta práctica.

3.3.2.3 Cohesión -

El concepto de acoplamiento ayuda a evaluar las interfaces entre módulos y la forma como éstas afectan el mantenimiento y la reutilización de los mismos. La cohesión, también conocida con los nombres de: robustez, solidez, o coherencia de un módulo, es una medida de la fuerza con que los elementos dentro de un módulo están unidos. Los elementos o componentes de un módulo son las instrucciones, los grupos de instrucciones o las llamadas a otros módulos. Entre más robusto es un módulo es más independiente y necesita menos acoplamiento con otros.

1. Cohesión por función

Un módulo es funcional si todos sus elementos (incluyendo llamadas a otros módulos) contribuyen a ejecutar una y solo una función. Este es el tipo de cohesión ideal.

2. Cohesión por secuencia

Un módulo tiene rigidez o robustez secuencial cuando sus elementos ejecutan tareas donde los datos que produce una función son la información que recibe la otra. No es el peor tipo de cohesión y responde a una forma de pensar natural: 'haga ésto, después lo otro, después aquello' sin embargo, 'ésto, lo otro, y aquello' representan funciones que pueden ser realizadas por módulos funcionales.

Por ejemplo, el módulo siguiente claramente realiza dos funciones que pueden separarse en dos módulos independientes.

MODULO XYZ

SUMAR TOTAL DE INGRESOS Y

CALCULAR EL IMPUESTO SOBRE LA RENTA

3. Cohesión por comunicación

Esta estructura se presenta cuando un módulo realiza varias funciones que usan la misma estructura de datos. El problema con éste tipo de cohesión, es que puede suceder que una de las funciones se necesite ejecutar en otra parte del programa sin la presencia de las otras funciones. Por lo tanto, es mejor separar y crear un módulo para cada función.

Por ejemplo, el módulo siguiente contiene dos funciones que actuan sobre una misma estructura de datos, sin embargo, estas dos funciones pueden realizarse en dos módulos separados.

MODULO XYZ

FORME EL VECTOR DE PRIORIDADES

ORDENE EL VECTOR DE PRIORIDADES EN FORMA ASCENDENTE

4. Cohesión por procedimiento

Aqui, varias funciones se han ensamblado en un módulo único, porque el control (no los datos como en la cohesión por secuencia) fluye de una función a otra. Nuevamente, lo mejor es separar las funciones en módulos independientes.

Por ejemplo,

MODULO XYZ

ORDENE EL ARCHIVO DE PERSONAL

CALCULE EL TOTAL DEL INVENTARIO

.END LITEREAL

.b2

.lm-4

.le;Cohesi'on por coincidencia temporal

.b

.lm+4

En este caso, varias funciones aparecen juntas porque sus elementos est'an relacionados en el tiempo, por ejemplo, las funciones que se tienen que ejecutar al principio o al final de un proceso. Sin embargo, estas funciones deben de separarse 'o ser parte de otras.

.b2

Por ejemplo,

.tp 8

.literal

MODULO XYZ

INICIE A CEROS LOS APUNTADORES

EMBOBINE LA CINTA

VERIFIQUE LAS BANDERAS DE ESTADO

5. Cohesión por clase

Varias tareas son realizadas en un módulo porque se cree que pertenecen a una misma clase o categoría. Por ejemplo, un módulo de entrada y salida que puede ejecutar todas las funciones de entrada y salida de un programa es un módulo con poca cohesión. Por el contrario, un módulo robusto es muy específico, realiza solo una función.

6. Cohesión por conicidencia

Esta es la peor de las cohesiones; simplemente se encuentran varias funciones en un módulo sin justificación alguna.

3.3.2.4 Parsimonia -

Este criterio se usa para evitar que se diseñen módulos de propósito general. Cuando se diseñan, sobre todo programas grandes, las funciones de los módulos, deberán ejecutar solo lo necesario para cumplir con los Requerimentos del Programa de Computo (RPC). Las tendencias proféticas de algunos diseñadores

a concebir módulos con capacidades más amplias de lo necesario a fin de atender futuros requerimientos deben de manejarse con reservas, ya que diseñar funciones generalizadas puede elevar el costo del proyecto y retrasarlo sin ninguna necesidad.

Hay otros criterios para evaluar diseños de diagramas de estructura, pero los más importantes son los que ya se mencionaron. El lector interesado puede consultar [BER81], [JEN79], [MYE75] y [YOU75].

A continuación presentamos las técnicas más conocidas para el diseño de diagramas de estructura que aunadas a los criterios de evaluación arriba expuestos, forman un conjunto de herramientas que ayudan a un buen diseño de módulos.

3.3.2.5 Descomposición Funcional -

Varias formas de descomposición funcional han sido propuestas ([PAR72], [BAK75], [DIJ76] y [LIN79]). La técnica de descomposición funcional es la forma más simple de usar el principio de "divide y conquista", y consiste en llevar a cabo, recurrentemente, los siguientes pasos:

1. Redactar en lenguaje natural una instrucción que describa la función a desarrollar.

2. Si la función se puede desarrollar en una página (aproximadamente 50 lineas de código ejecutable), terminar la descomposición de esta función. En caso contrario, dividir la función en subfunciones y verificar que el conjunto de estas subfunciones es equivalente a la función original.

3. Repetir para cada subfunción el proceso de descomposición.

Este método para diseñar diagramas de estructura usa el concepto de una máquina abstracta o imaginaria. El diseñador debe de pretender que cada vez que escribe una instrucción en lenguaje natural, existe una computadora que es capaz de entender y ejecutar la instrucción. Una vez que una instrucción ha sido escrita, el diseñador deberá preguntarse a sí mismo si la instrucción puede traducirse a lenguaje de computadora sin mucho esfuerzo, esto es, en aproximadamente una página de código. Si la traducción requiere más de una página de código, reflejará que la instrucción todavía es muy abstracta y que deberá dividirse en más sub-instrucciones.

Este proceso se conoce también con el nombre de refinamiento progresivo. El diseñador empieza escribiendo una instrucción a

muy alto nivel. Estas órdenes o instrucciones son escuetas, claras y con mucho contenido. Este contenido es refinado o aclarado por el diseñador al tratar de traducirlo a lenguaje de computadora. Si una instrucción en el nivel X es fácilmente traducible a código, el proceso de refinamiento termina; sino, deberán escribirse, otras instrucciones que realicen subfunciones en el nivel X+1. Las funciones en el nivel X+1 representan subfunciones del nivel X que aclaran o hacen más fácil la traducción a código.

El método de descomposición funcional tiene como objetivo dividir el problema de tal forma que los bloques o módulos de los niveles más bajos sean facilmente traducibles a código. La desventaja radica en que la descomposición funcional es realizada casi exclusivamente utilizando como criterio el tamaño de los módulos y la capacidad del diseñador para identificar funciones que sean independientes, pero que integradas cumplan la función del programa.

3.3.2.6 Descomposición Usando Flujo De Datos -

Este método se basa en el uso de gráficas para representar el flujo de datos ([BER80], [PET80], [BAN79], [MAR67], [YOU75], [MYE75]). El método es también conocido con los nombres de "diseño centrado en la transformación", y "diseño compuesto", y

en realidad, no es otra cosa que una descomposición funcional con respecto al flujo de datos. Cada bloque del diagrama de estructura se obtiene usando repetidamente el concepto de "caja negra" que transforma un conjunto de datos de entrada en un conjunto de datos de salida.

El método consiste en dibujar un Diagrama del Flujo de Datos (figura 3.3.1) y en base a éste construir el diagrama de estructura. Como toda técnica de diseño, el proceso se repite hasta que los módulos resultantes sean fáciles de traducir a código.

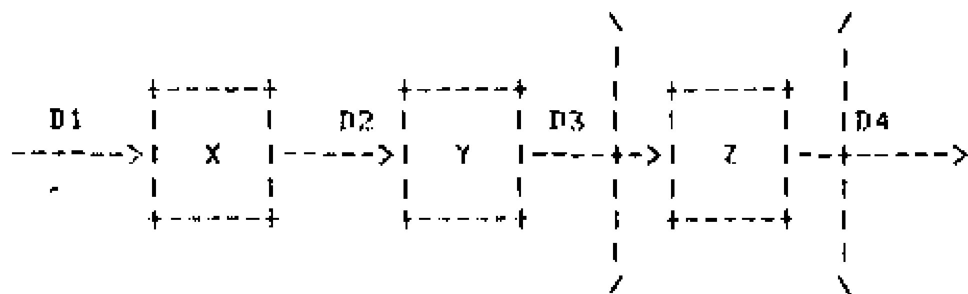


FIGURA 3.3.1 DIAGRAMA DE FLUJO DE DATOS

En un diagrama de flujo de datos un bloque representa un proceso de transformación y una flecha representa un conjunto de datos que entra para ser transformado o que sale ya procesado. Si el diagrama de flujo de datos se usara como diagrama de estructura de módulos, el resultado sería una red de módulos en lugar de una jerarquía de módulos. Para convertir el Diagrama de Flujo de Datos (DFD) en un diagrama de estructura jerárquico se selecciona uno o más de los bloques centrales. A continuación

se 'levantan' los bloques seleccionados dejando los bloques de los extremos 'colgando'; de esta forma el [DFD] queda dividido en tres partes que corresponden a los tres módulos del primer nivel del diagrama de estructura:

1. El módulo de entrada o lectura, llamado módulo aferente
2. El módulo de transformación central y
3. El módulo de salida o escritura, llamado módulo eferente

Por ejemplo, si en la figura 3.3.1 se toma como módulo de transformación central el bloque 'Z', entonces, el primer nivel del diagrama de estructura quedará definido como se muestra en la figura 3.3.2. El módulo 1.1, que obtiene el conjunto de datos D3 para ser transformado por el módulo central en el conjunto de datos D4, tendrá que realizar las transformaciones 'X' y 'Y'. Esto se logra con dos módulos en el segundo nivel: uno que obtiene los datos D2 y otro que los transforma en D3.

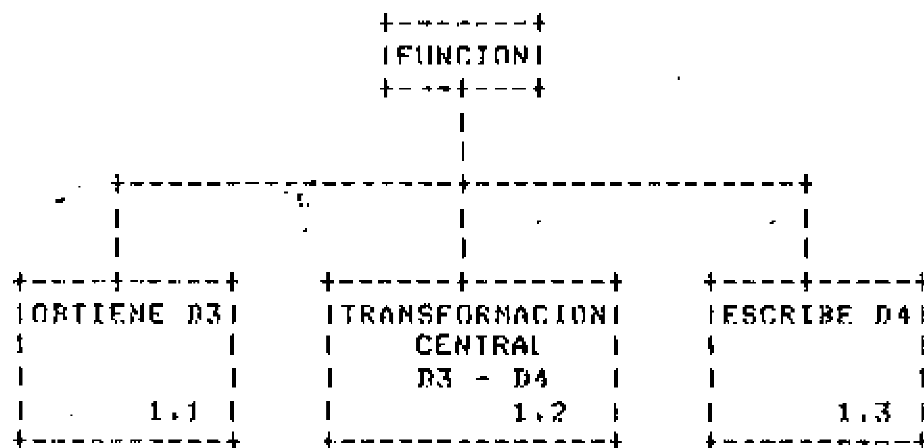


FIGURA 3.3.2 PRIMER NIVEL DEL DISE~O DEL DIAGRAMA DE ESTRUCTURA APARTIR DEL DIAGRAMA DE FLUJO DE DATOS

Ahora bien, el módulo 1.1.1 de la Figura 3.3.3. tiene como objetivo producir los datos D2 para que el módulo 1.1 llame al módulo 1.1.2 y éste los transforme en D3. Entonces, se necesitan dos funciones en el tercer nivel jerárquico: una que obtenga los datos D1 y otra que transforme D1 en D2.

El resultado del proceso de convertir el diagrama de flujo de datos de la figura 3.3.1 a diagrama de estructura se muestra en la figura 3.3.3.

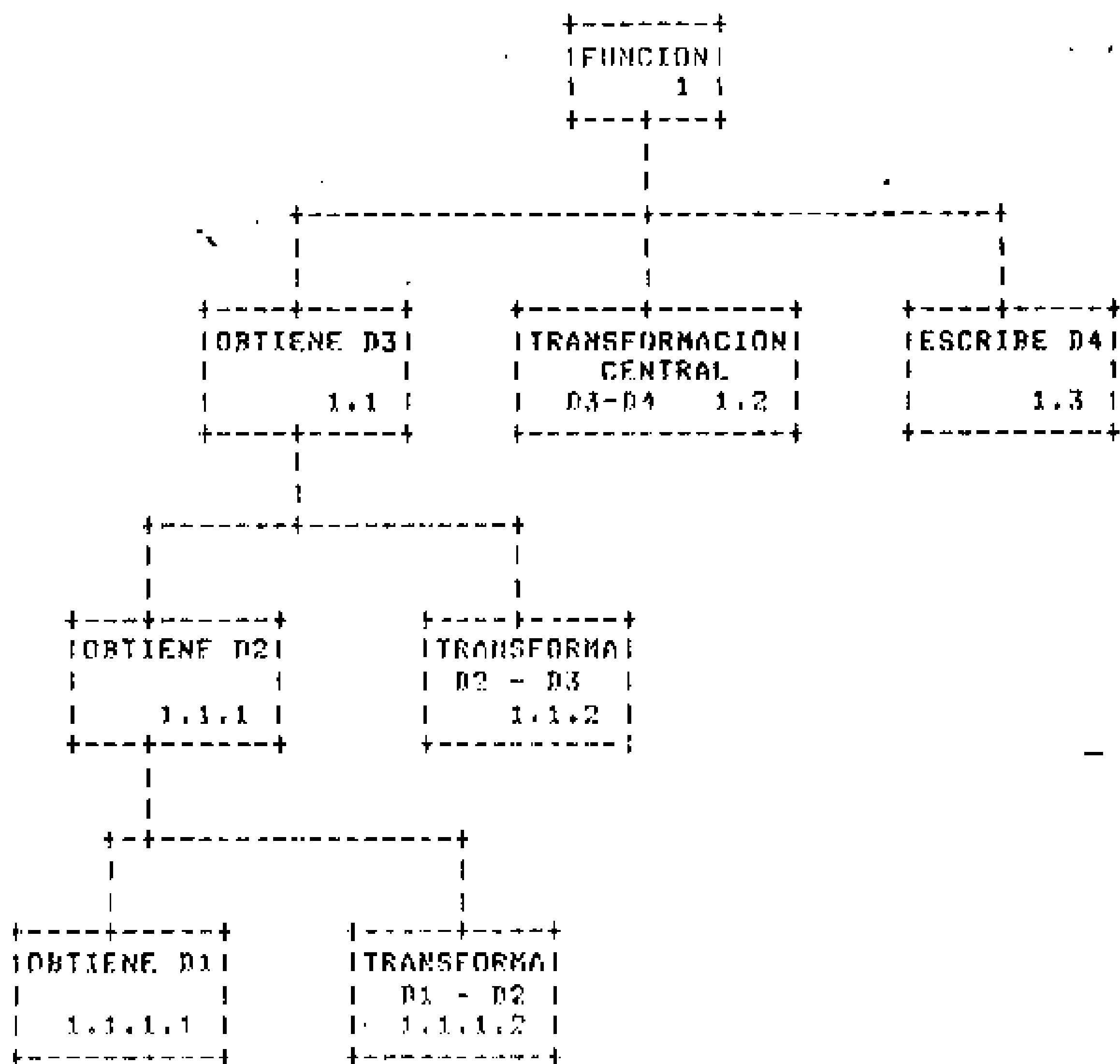
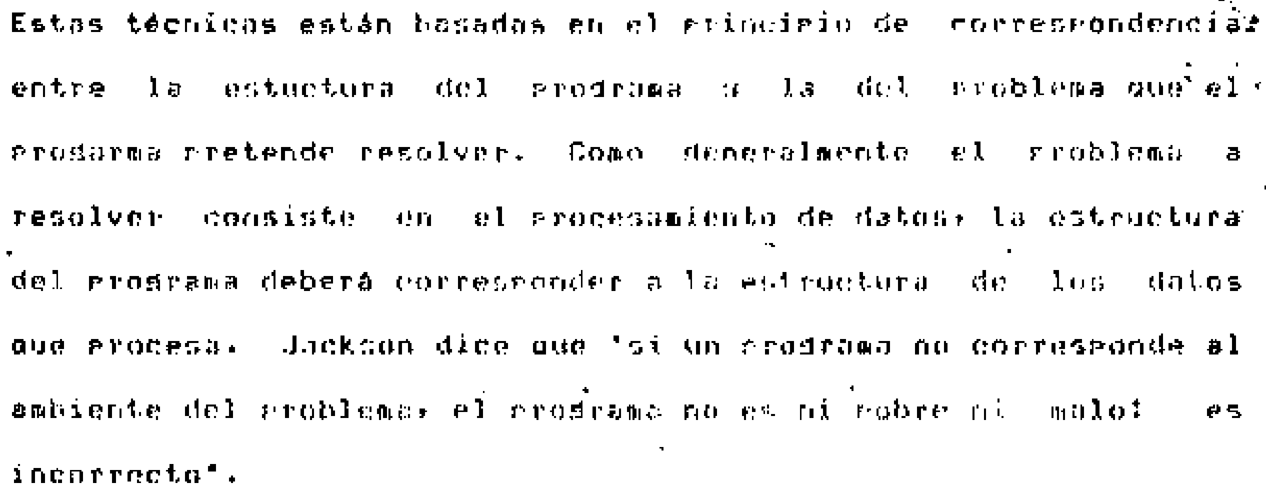


FIGURA 3.3.3 DIAGRAMA DE ESTRUCTURA DE DISEÑADO A PARTIR DEL DIAGRAMA DE FLUJO DE DATOS

3.3.2.7 Descomposición Usando Estructura De Datos -

Dos técnicas de descomposición similares fueron desarrolladas en forma independiente y prácticamente al mismo tiempo por Jackson [JAC75] en Inglaterra y Warnier [WAR74] en Francia.


Estas técnicas están basadas en el principio de correspondencia entre la estuctura del programa y la del problema que el programa pretende resolver. Como generalmente el problema a resolver consiste en el procesamiento de datos, la estructura del programa deberá corresponder a la estructura de los datos que procesa. Jackson dice que "si un programa no corresponde al ambiente del problema, el programa no es ni pobre ni malo: es incorrecto".

Supongamos:

En un almacén de una fábrica se requiere producir un reporte que presente el movimiento neto de productos (artículos, piezas, partes, etc.) que el almacén compra y vende. Se cuenta con un sistema de cómputo que tiene un archivo maestro de transacciones donde se registra cada venta y cada compra. Los registros del archivo contienen, entre otras variables, el número de identificador del artículo, el tipo de movimiento (venta o compra) y la cantidad. El archivo se encuentra clasificado por número de artículo de tal forma que todos los registros que representan los movimientos que ha tenido un artículo se encuentran secuencialmente ligados. El programa deberá producir el movimiento neto de cada artículo y al final deberá resumir el total de artículos que sufrieron movimientos. El reporte deberá tener el siguiente formato

TITULO

A5/13672   MOVIMIENTO NETO = - 490

A5/17924   MOVIMIENTO NETO =   1500.

B31/8200   MOVIMIENTO NETO =   123

TOTAL DE ARTICULOS CON MOVIMIENTOS = 3

La estructura del programa deberá derivarse de la estructura de datos. Puesto que se desea obtener una estructura jerárquica, los datos serán analizados jerárquicamente. El diagrama de estructura de los datos, así como el diagrama de estructura del programa representan la relación 'es compuesto de'. Por ejemplo, como entrada al programa tenemos el archivo de transacciones que 'es compuesto de' grupos de registros y cada grupo de registros 'es compuesto de' registros individuales y cada registro es de uno de dos tipos: venta o compra. (ver figura 3.3.4) (A)).

```
+-----------+
|           |
|           |
|           |                                 +-----------+
|  ARCHIVO  |-------->                        |  PRODUCE  |
|           |                                 |  REPORTE  |
|           |                                 +-----+-----+
|           |                                       |
|           |                                       |
|  MAESTRO  |                    +------------------+-----------+
|           |                    |                  |           |
|           |                    |                  |           |
|           |              +----+-----+   +-----+----+   +---+---+
|           |              | PRODUCE  |   |PRODUCE EL|   |PRODUCE|
+-----+-----+              |          |   |CUERPO DEL|   |       |
      |                    |ENCABEZADO|   |  REPORTE |   |RESUMEN|
      |                    +----------+   +-----+----+   +-------+
      |                                         |
  +---+----+                                    |
  | GRUPO  |                              +----+-----+
  |   DE   |-------->                     | CONSUME  |
  |REGISTROS|                             |  GRUPO   |
  +---+----+                              +----+-----+
      |                                        |
      |                                        |
      |                                        |
   +--+----+                              +----+-----+
   |       |                              | CONSUME  |
   |REGISTRO|-------->                    |REGISTRO  |
   |       |                              +----+-----+
   +--+----+                                   |
      |                                        |
  +---+------+                       +---------+---------+
  |          |                       |                   |
+--+--+    +--+---+              +---+----+         +---+----+
|VENTA|    |COMPRA|-------->     | CONSUME|         | CONSUME|
+-----+    +------+              |REGISTRO|         |REGISTRO|
                                 | VENTA  |         | COMPRA |
                                 +--------+         +--------+
         (A)                                 (B)
DIAGRAMA DE ESTRUCTURA              DIAGRAMA DE ESTRUCTURA
     DE DATOS.                          DEL PROGRAMA
```

FIGURA 3.3.4 DISENO DEL DIAGRAMA DE ESTRUCTURA DEL PROGRAMA A PARTIR DEL DIAGRAMA DE ESTRUCTURA DE DATOS.

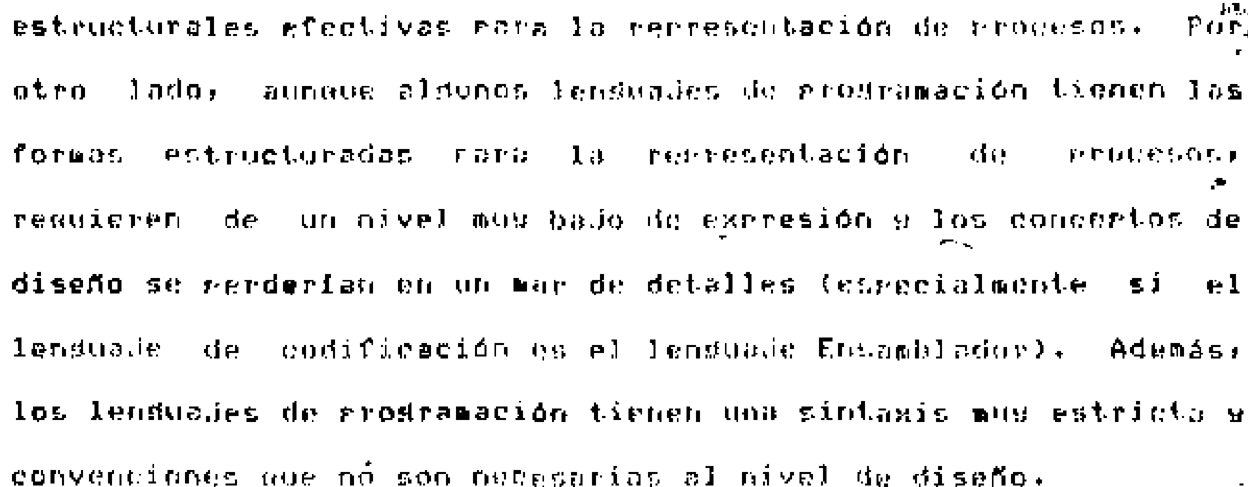
Por otro lado, a la salida del programa tenemos que la

estructura del reporte es la siguiente: el reporte "es compuesto de" un encabezado, un cuerpo (todos los movimientos netos) y el resumen. La estructura del programa se deriva de la estructura del reporte y de la estructura de los datos del programa: el programa que produce el reporte "es compuesto de" un módulo que produce el encabezado, de un módulo que produce el cuerpo y de un módulo que produce el resumen. A su vez, el módulo que produce el cuerpo contiene un módulo que consume un grupo de registros requiriendo a su vez de un módulo que consume un registro y como los registros son de dos tipos se necesitarán dos módulos: uno para consumir registros de tipo venta y otro para consumir registros de tipo compra. (ver figura 3.3.4 (B)).

Como puede verse en la figura 3.3.4 una parte del diagrama de estructura del programa que produce el reporte corresponde a la estructura de los datos de entrada (el archivo de transacciones), y otra parte corresponde a la estructura de los datos de salida (el reporte).

3.3.2.8 El Detallado De Módulos -

En esta etapa del diseño de un programa de computadora, se procede a la especificación detallada del proceso o función que un módulo realize. El uso de lenguaje natural para esta especificación podría ser suficiente, pero no cuenta con formas



estructurales efectivas para la representación de procesos. Por otro lado, aunque algunos lenguajes de programación tienen las formas estructuradas para la representación de procesos, requieren de un nivel muy bajo de expresión y los conceptos de diseño se perderían en un mar de detalles (especialmente si el lenguaje de codificación es el lenguaje Ensamblador). Además, los lenguajes de programación tienen una sintaxis muy estricta y convenciones que no son necesarias al nivel de diseño.

Por lo tanto, la técnica usada para el detallado de los módulos consiste en especificar el contenido de cada módulo usando un lenguaje intermedio entre el lenguaje natural y el código de programación. A este nivel se requiere usar frases en lenguaje natural y/o instrucciones semejantes a las del código, enmarcadas en un "seudo-código" que usa las estructuras de proceso de la programación estructurada: la secuencia, la decisión y la repetición. El objetivo es obtener un nivel de descripción del proceso tal que, cuando los módulos se codifiquen, una línea de detalle corresponda de una a cinco líneas de código.

En esta etapa deberán especificarse todos los datos con que un módulo interactúa. Para ello son necesarios:

1. La especificación de todas las estructuras de datos internos a cada módulo (y las globales a varios módulos, en caso de que se haya decidido usar módulos acoplados por datos globales) y como ya se mencionó.

2. Las estructuras de los datos externos que ya deberán estar diseñadas, por ser parte del diseño de la Arquitectura.

Dadas las características intermedias del detallado de procesos, se ha creado un Lenguaje para Diseño de Procesos (LDP) [ZYC81] que permite la especificación del contenido de los módulos a base de lenguaje natural y estructuras de proceso. Sin embargo, el uso del lenguaje natural deberá ser restringido: las frases o instrucciones en lenguaje natural deberán mencionar datos por su nombre y ser traducibles a código de una forma trivial.

El uso de LDP's es una práctica común en la industria y se recomienda en varias referencias, por ejemplo: [GAN79], [LIN79] y [JEN79].

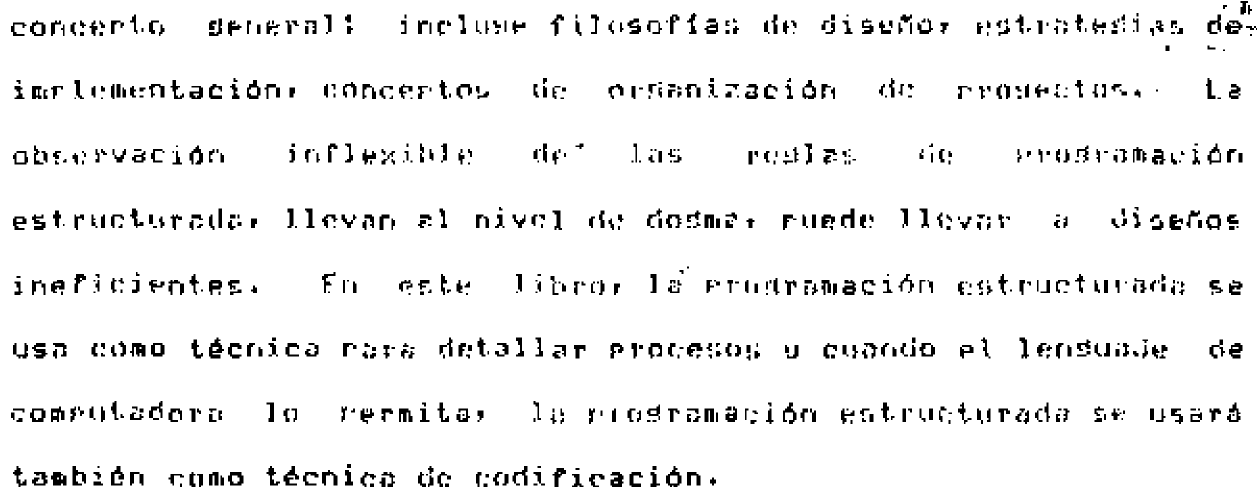
El lenguaje para diseño de programas consiste en usar el pseudo-código de estructuras de proceso presentadas en la sección 3.3.3

3.3.3 Programación Estructurada -

Es bien sabido que la "Programación Estructurada" fué la primera técnica estructurada discutida y practicada ampliamente. Está basada fundamentalmente en los estudios realizados en 1960 por DIJKSTRA, WIRTH, BOHM y JACOBINI.

El término Programación Estructurada ha aparecido en innumerables ocasiones en la literatura y con diferentes significados. Por ejemplo, Donelson [DON 73] define la programación estructurada como la forma de organizar y codificar programas para que éstos sean facilmente entendidos y modificados. Por otra parte, muchos otros autores consideran que un programa bien estructurado es aquel que no contiene instrucciones de ramificación o GO TO's. Las definiciones de éste último tipo fueron sustentadas principalmente por Dijkstra [DIJ68]. Afortunadamente, aparecieron otras definiciones posteriores más realistas. Mills [MIL 72], por ejemplo, establece que un programa estructurado es aquel que no simplemente no contiene GO TO's, sino que presenta una estructura. Wirth [WIR 74] por su parte presenta la siguiente definición: "programación estructurada es la formulación jerárquica, de estructuras de control de flujo anidadas e instrucciones computables, para formar programas".

Para muchos de nosotros la programación estructurada es un


concepto general: incluye filosofías de diseño, estrategias de implementación, conceptos de organización de proyectos. La observación inflexible de las reglas de programación estructurada, llevan al nivel de dogma, puede llevar a diseños ineficientes. En este libro, la programación estructurada se usa como técnica para detallar procesos y cuando el lenguaje de computadora lo permita, la programación estructurada se usará también como técnica de codificación.

De todos estos definiciones se observa que la programación estructurada es la aplicación de métodos básicos de descomposición de problemas para establecer una estructura jerárquica fácilmente manejable, a través del proceso llamado refinamiento progresivo. Esto es, cuando hablamos de la estructura del programa nos referimos a la manera en que un algoritmo complejo es construido a partir de procesos sucesivamente más simples.

Para mantener esta estructura jerárquica manejable, es crucial que las operaciones que constituyen cada nivel de abstracción estén relacionadas por medio de estructuras de control de flujo suficientemente simples, y que cada operación sea descrita como una parte del programa con un punto de inicio y un punto de terminación.

La programación estructurada está basada en un teorema

matemático probado, el cual establece que cualquier programa ...
puede ser escrito utilizando solamente tres estructuras lógicas de control simples [BOH 66], [IBM 74], [JEN 79]:

1. La secuencia

2. La decisión o alternativa y

3. La repetición

Estas estructuras básicas de especificación de procesos se explican a continuación:

La secuencia

La secuencia es una estructura de proceso que consiste en enunciar, una después de otra, una serie de instrucciones. Una instrucción es: una frase en lenguaje natural, trivialmente traducible a código, o una estructura de proceso. Las frases en lenguaje natural son imperativas: idealmente consisten de un verbo activo (calcule, lea, escriba, verifique, etc.) seguido de una cláusula objeto lo más sencilla posible, por ejemplo:

Verifique el crédito del cliente.
Calcule el deducible del salario
Encuentre el mínimo de los números en la lista
Asigne a la variable interés el valor de .15
Llame a la subrutina VERINI

La decisión

La decisión es una estructura de proceso que permite especificar alternativas en la ejecución de instrucciones dependiendo de una condición. Por facilidad de exposición hemos dividido la decisión en tres tipos: simple, doble y múltiple. La forma clásica de representar la decisión es a través de un diagrama de flujo. Sin embargo no lo usaremos en esta exposición puesto que consideramos que el uso de 'palabras reservadas' y el 'sangrado de instrucciones' es más que apropiado, simplificado el mantenimiento y obteniéndose claridad aceptable.

- La decisión simple

Esta estructura de proceso tiene la siguiente forma:

```
SI condición ENTONCES
    instrucción - 1
FIN (SI)
    instrucción - 2
```

Aquí se especifica que si la condición es verdadera, la instrucción-1 deberá de realizarse, y si la condición es falsa, entonces la instrucción-2 es la que deberá ejecutarse. Sólo una de las dos instrucciones se ejecuta como resultado de la ejecución de la decisión doble.

- La decisión múltiple

Esta estructura de proceso es una generalización de la alternativa doble. La estructura tiene la siguiente forma:

```
SI      condición-1   ENTONCES
            instrucción-1
SINO SI condición-2   ENTONCES
            instrucción-2
SINO SI condición-3   ENTONCES
            instrucción-3
SINO SI condición-4   ENTONCES
            instrucción-4
        .
        .
        .
SINO
            instrucción-N
FIN(SI)
```

Al igual que en la alternativa simple y doble, solamente una de las instrucciones se ejecuta como resultado de la ejecución de la estructura. Por lo tanto, las condiciones deberán ser mutuamente exclusivas. Debido a ésta última propiedad, es común encontrarse como parte del lenguaje de diseño y en algunos lenguajes de computadora la siguiente forma equivalente de la decisión múltiple que se conoce como 'CASO':

```
CASO
 condición-1
        instrucción-1
 condición-2
        instrucción-2
 condición-3
        instrucción-3
    .
    .
    .
 SINO
        instrucción-N
 FIN(CASO)
```

En ambas formas el último "SINO" es opcional, por ejemplo:

```
SI          (mes=1,3,5,7,8,10,12)   ENTONCES
            asigne 31 al número de días
SINO SI     (mes=4,6,9,11)          ENTONCES
            asigne 30 al número de días
SINO SI     (año es bisiesto)       ENTONCES
            asigne 29 al número de días
SINO
            asigne 28 al número de días
FIN(SI)
```

Esta especificación de la parte de un proceso puede hacerse también usando el caso:

```
CASO
            (mes=1,3,5,7,8,10,12)
            asigne el 31 al número de días
            (mes=4,6,9,11)
            asigne 30 al número de días
            (año es bisiesto)
            asigne 29 al número de días
SINO
            asigne 28 al número de días
FIN(CASO)
```

Es también común encontrar la siguiente forma de decisión o alternativa múltiple:

```
CASO        'variable o expresión'
            (Lista de valores-1)
                  instrucción-1
            (Lista de valores-2)
                  instrucción-2
            (Lista de valores-3)
                  instrucción-3
              .
              .
              .
```

```
SINO
        instrucción-N
FIN(CASO)
```

El significado de esta estructura es intuitivo: si la variable o expresión tiene un valor de la lista de valores-1, la instrucción-1 es ejecutada y así sucesivamente. Si la variable o expresión tiene un valor que no está en alguna de las listas de valores la instrucción-N es ejecutada. Por ejemplo:

```
CASO  estado-civil
   (CASADO)
         procese empleado casado
   (SOLTERO)
         procese empleado soltero
   (DIVORCIADO)
         procese empleado divorciado
   (VIUDO)
         procese empleado viudo
   (SEPARADO)
         procese empleado separado
   SINO
         reporte error de estado civil
FIN(CASO)
```

Es posible que esta estructura de alternativa múltiple se encuentre en algún lenguaje de computadora de alto nivel, por

ejemplo: el Fortran 77 de algunas máquinas tiene la estructura del tipo 'SINO SI' ya mencionada.

La repetición

La repetición es una estructura de proceso que permite la especificación de la ejecución iterativa de una serie de instrucciónes.

Presentaremos los siguientes tipos de estructuras de repetición:

La estructura MIENTRAS
La estructura EJECUTA-HASTA
LA estructura REPITE
La estructura CICLO

MIENTRAS

Esta estructura para el detallado de procesos tiene la siguiente forma:

```
MIENTRAS condición
      instrucción
FIN (MIENTRAS)
```

El significado de esta estructura es el siguiente: si la condición es veradadera, la instrucción es ejecutada y después de ser ejecutada, la condición vuelve a ser evaluada. si

resulta verdadera la instrucción se vuelve a ejecutar. Este proceso continúa hasta que la condición sea falsa, en cuyo caso, la instrucción no se ejecuta y el proceso ya no se repite.

EJECUTA-HASTA

La estructura del ejecuta-hasta es la siguiente:

```
EJECUTA
     instrucción
HASTA condición
```

Esta estructura de proceso señala que, al ejecutarse la estructura, la instrucción debe de realizarse. Una vez que la instrucción ha sido ejecutada, la condición es evaluada y si su valor es falso, entonces la instrucción deberá ejecutarse de nuevo. Esto se repetirá hasta que la condición sea verdadera. En este momento, la instrucción no se volverá a ejecutar y se dice que la estructura de repetición se ha terminado.

Una diferencia entre la estructura 'MIENTRAS' y la estructura 'EJECUTA-HASTA' es que la última ejecuta la instrucción al menos una vez.

REPITE

Esta estructura de proceso es una de las estructuras más conocidas para los conocedores del lenguaje Fortran. Tiene la siguiente forma:

```
REPITE variable= E-inicio,E-fin,E-incremento
    instrucción
FIN(REPITE)
```

El significado de esta estructua es el siguiente: al empezar la ejecución dela estructura, la variable toma el valor de la expresión E-inicio, si este valor es mayor que el valor de la expresión E-fin, la instrucción no se ejecuta y la estructura se termina. Sin embargo, si el valor de la variable es menor que el valor de E-fin entonces la instrucción se ejecuta. Una vez que la instrucción ha sido ejecutada, la variable se incrementa en un valor igual al de la expresión E-incremento. Nuevamente si el valor de la variable es menor que el valor de E-fin la instrucción se volverá a ejecutar; en caso contrario, la ejecución de la estructura se termina.

La variable deberá ser de tipo entera, y las expresiones E-fin y E-incremento deberán representar números enteros.

CICLO

La forma de esta estructura es la siguiente:

```
CICLO expresión
    instrucción
FIN(CICLO)
```

Con esta estructura se especifica lo siguiente: al inicio de la ejecución del ciclo, se calcula el valor de la expresión que, debe de resultar en un número entero. Este número entero representa el número de veces que deberá repetirse la ejecución de la instrucción. Posteriormente la ejecución de la estructura se da por terminada. Si el valor de la expresión es cero o negativo, la instrucción no se ejecuta y se termina la estructura.

En la siguiente sección se indica como los conceptos de programación estructurada se usan en el detallado de procesos en la fase de Diseño de un Programa de Computadora. Al uso directo de estas estructuras en la forma presentada en esta sección mediante palabras reservadas e instrucciones en español se le conoce como especificación de procesos en "pseudo-código". Se entiende por pseudo-código un lenguaje estructurado, más no ejecutable por una máquina computadora, ya que parte de las instrucciones todavía están en lenguaje natural.

Las palabras reservadas son las palabras clave que nos indican la estructura del proceso de un programa, por ejemplo, SI, ENTONCES, SINO, FIN (SI), etc.

Como veremos más adelante, las estructura básicas de especificación de proceso forman parte del repertorio de instrucciones de algunos lenguajes de computadora. En este

caso, la traducción de un diseño en pseudo-código al lenguaje de computadora es una tarea trivial.

En la presentación de las estructuras de proceso, como se indicó anteriormente, hemos usado el concepto de "instrucción" de una manera genérica para indicar, ya sea una frase en lenguaje natural o una estructura de proceso. Cuando se usa una estructura de proceso dentro de otra estructura de proceso, en la parte o partes donde se indica que debe ir una instrucción, se dice que hay un anidamiento, por ejemplo:

```
MIENTRAS condición
    SI condición-2 ENTONCES
        frase en lenguaje natural-1
    SINO
        CICLO expresión
            frase en lenguaje natural-2
        FIN(CICLO)
    FIN SI
      frase en lenguaje natural-3
FIN(MIENTRAS)
```

En este ejemplo, tenemos una estructura MIENTRAS cuya instrucción es una estructura de secuencia. La secuencia contiene dos instrucciones, la decisión doble y la frase natural-3. La decisión tiene a su vez dos intrucciones

alternativas, la frase en lenguaje natural-1 y la estructura ciclo. Por último la instrucción de la estructura ciclo es la frase natural-2. Note que el anidamiento de estructuras de proceso se representa usando el sangrado de las estructuras y frases en lenguaje natural. Al terminar el detalle de un proceso en la fase de diseño solo se deberá tener frases en lenguaje natural y estructuras de proceso anidadas. Cabe recordar que estas frases en lenguaje natural deberán ser trivialmente traducibles a código de computadora.

En la práctica, la programación estructurada es una herramienta que permite escribir programas mas claramente concebidos, más legibles y por lo tanto, con menos errores; sin embargo, no puede afirmarse que el mero uso de las estructuras de control básicas lleve natural y automáticamente a escribir programas estructurados; una serie de reglas mecánicas no puede ser un sustituto de la claridad del pensamiento. Sí es cierto, sin embargo, que un programa bien estructurado será concebido típicamente usando dichas estructuras como sus bloques de construcción [KER74].

La claridad de un programa depende en gran medida del estilo del programador. Si bien los principios de un buen estilo de programación, no pueden ser expresados como reglas mecánicas, la experiencia nos indica algunas normas que pueden servir como guía de estilo. Estas se detallarán en la sección de normas de

codificación (4.2.3).

Desde el punto de vista de la programación estructurada, la mejor documentación de un programa la constituye la claridad de su estructura; además, la única documentación confiable de un programa es el programa mismo, pues sólo leyendo el código puede el programador dar por hecho lo que hace el programa. De allí el énfasis en la legibilidad del código, base indiscutible de la programación estructurada.

3.4 Técnicas De Desarrollo

El objetivo de las técnicas de desarrollo es proporcionar una guía para la codificación, integración y verificación de los módulos de un programa.

En esta sección se presentan las técnicas empleadas con mayor frecuencia en el desarrollo de programas de computadora.

3.4.1 Tecnicas De Codificación -

Las técnicas de codificación tienen como objetivo facilitar la traducción del diseño detallado de los módulos que componen un programa a un lenguaje de programación específico, manteniendo

la estructura jerárquica definida en la fase de diseño.

La descripción detallada de los módulos sirve como base para la codificación. Cualquier módulo, aún el más complicado, ha sido detallado utilizando las reglas básicas de programación estructurada: los módulos tienen una sola entrada y una sola salida y pueden ser facilmente traducidos, revisados y probados en forma independiente.

El detalle de los módulos se traduce en instrucciones de máquina (computadora), dependiendo de las características del propio lenguaje de programación. Mientras algunos lenguajes incluyen las estructuras de control básicas entre sus instrucciones normales(ALGOL, PASCAL, PL/1, FORTRAN 77), en los lenguajes más populares (COBOL, BASIC, FORTRAN 66), estas facilidades no solo no existen, sino que además presentan algunas dificultades para su implementación.

La situación ideal se presenta cuando la instalación de cómputo cuenta con un lenguaje de programación estructurado, con lo cual la función de traducir el detalle de los modulos se convierte en una tarea trivial como se muestra a continuación:

ESTRUCTURA	IMPLEMENTACION FORTRAN 77
Mientras (bandera="S")	WHILE(BANDERA='S')
Llama actualiza	CALL ACTUALIZA
Limpiar pantalla	REWIND 3
Escribir (pantalla) 'cambios (S,N)?'	WRITE(3,5000)
"CAMBIOS(S,N)?"	
Leer (pantalla) bandera	READ(3,5100) BANDERA
Fin (mientras)	END WHILE

Sin embargo, esto no sucede comunmente con los lenguajes utilizados en la mayoría de las instalaciones de cómputo. En el caso del lenguaje de programación COBOL, por ejemplo, la proposición, "Si-entonces-sino" no puede ser implementada directamente, pues no permite el uso adecuado de proposiciones anidadas como se muestra en la figura 3.4.1. La desventaja de éste hecho es que el código de un módulo no permanece unido, sino que hay que dividirlo en secciones y lugares físicos diferentes.

ESTRUCTURA

```
Si P entonces
    Si Q entonces
        Instrucción-1
    Sino
        Instrucción-2
    Fin (si)
    Instrucción-3
Sino
    Instrucción-4
Fin (s).
```

IMPLEMENTACION COBOL

```
IF P THEN
    PERFORM IF-INTERNO
  INSTRUCCION-3
ELSE
  INSTRUCCION-4

IF INTERNO SECTION
    IF Q THEN
  INSTRUCCION-1
ELSE
    INSTRUCCION-2
```

FIGURA. 3.4.1 PROBLEMA DEL USO DE LA ESTRUCTURA SI-ENTONCES-SINO ANIDADAS EN COBOL.

En el caso de los compiladores FORTRAN 1966 [FOR66], el mayor problema consiste en que solo reconocen instrucciones simples y por lo tanto la estructura lógica del programa deber ser construida utilizando la instrucción de ramificación incondicionada ('GO TO') la cual, si se utiliza indisciplinadamente obscurece la lógica del programa. En cualquier lenguaje de programación deberemos restringirnos a usar instrucciones que, aunque no sean traducciones directas del pseudocódigo, al menos si sean implementaciones ordenadas o "emulaciones" de las estructuras básicas.

A continuación se presentan dos técnicas de codificación: Las emulaciones y los precompiladores, especialmente útiles cuando se utiliza un lenguaje de programación que no cuenta entre sus instrucciones normales: con las estructuras básicas, definidas en la sección de Técnicas de Diseño, específicamente en el punto de programación estructurada (2.3.3. Para terminar se menciona una tercera técnica que puede aplicarse a cualquier tipo de lenguaje y se conoce como estilo de programación.

3.4.1.1 Emulaciones -

A continuación se presentan, en FORTAN 66, las emulaciones de las principales estructuras básicas definidas en la programación estructurada. (Veáse figura 3.4.2)

ESTRUCTURA (SI-ENTONCES-SINO)

Si P entonces	IF (.NOT.P) GO TO N1
Instrucción-1	Instrucción-1
	GO TO N2
Sino	N1 CONTINUE
Instrucción-2	Instrucción-2
Fin(si)	N2 CONTINUE

ESTRUCTURA (MIENTRAS)

```
Mientras (condición)            N1  IF (.NOT.condicion) GO TO
N2
   Instrucción-1                       Instrucción-1
                                        GO TO  N1
Fin(mientras)                    N2  CONTINUE
```

ESTRUCTURA (EJECUTA)

```
Ejecuta                               GO TO N2
                                 N1  IF (condicion) GO TO N3
                                 N2  CONTINUE
   instrucción-1                       Instrucción-1
                                        GO TO N1
Hasta (condición)               N3  CONTINUE
                                      END IF
                                 N2  CONTINUE
```

ESTRUCTURA (REPITE)

Repite inicial, final, incr.		INDICE= INICIAL
	N1	CONTINUE
Instrucción-1		Instrucción-1
		INDICE=INDICE+INCR
Fin(repite)		IF(INDICE.LE.FINAL)GO TO N1

FIGURA 3.4.2 EMULACIONES EN FORTRAN 66 DE ESTRUCTURAS BASICAS

Las desventajas de la emulación de las estructuras básicas son: primero, que el número de pasos de descomposición necesarios para llevar el planteamiento original del programa a su implementación es mayor [CIM 81], y segundo, que la claridad de un programa escrito en pseudocódigo se pierde al traducirlo a código [JEN 79].

Tomando en consideración los puntos expuestos anteriormente, la solución más popular para la emulación de las estructuras básicas es el uso de un precompilador.

3.4.1.2 Precompiladores -

Un precompilador es un programa de computadora que extiende la sintaxis de un lenguaje de programación específico permitiendo al programador codificar sus programas utilizando las estructuas básicas (si-entonces-sino, mientas, etc.) en combinación con las instrucciones propias del lenguaje.

El uso de precompiladores presenta varias ventajas sobre el intento de programar en forma estructurada en lenguajes no estructurados:

a. Portobilidad
b. Menores violaciones a las reglas de programación estructurada.
c. Mayor aproximación entre los procesos de diseño y codificación.
d. Mayor confiabilidad.
e. Mayor mantenibilidad.

Algunos de los precompiladores más populares son el META COBOL [WEI 77], WATFOR [KER 76] y HIFTRAN [HIF 76].

En la práctica la programación estructurada es una herramienta que permite escribir programas más claramente concebidos, más legibles y por lo tanto, con menos errores; sin embargo, no puede afirmarse que el mero uso o emulación de las estructuras

de control básicas lleve automáticamente a escribir programas estructurados.

3.4.1.3 Estilo Del Programador -

La claridad de un programa depende en gran medida del estilo del programador. Si bien los principios de lo que constituye un buen estilo de programación no pueden tampoco ser expresados como reglas mecánicas de la buena programación, la experiencia nos indica algunas normas que pueden servir como "guía de estilo" y las cuales se detallarán en la sección de Normas de Codificación (4.2.3.).

Resumiendo, la mejor documentación de un programa la constituye la claridad de su estructura; además como ya se mencionó la documentación más confiable de un programa es el programa mismo, pues solo leyendo el código se puede detectar lo que hace el programa. Por lo tanto se recomienda la selección de lenguajes que estimulen una programación legible y estructurada, evitando la necesidad de instrucciones de flujo arbitrariamente complicadas.

3.4.2 Técnicas De Integración -

En esta sección se presentan algunos tipos de pruebas recomendadas para identificar los errores de cómputo que se originan en la fase de integración del proceso de programación

En la descripción de la fase de integración se describieron dos alternativas para validar módulos de programación: integración incremental y no incremental.

Asociadas a la integración incremental existen dos variantes en el desarrollo de las pruebas:

Pruebas de arriba hacia abajo (PARAB)

Pruebas de abajo hacia arriba (PABAR)

Antes de pasar a explicar las diferencias entre éstos dos tipos de pruebas en el contexto de un esquema de integración incremental, es conveniente señalar algunos términos que son utilizados comúnmente en forma errónea:

Los términos pruebas de arriba hacia abajo (PARAB), desarrollo de arriba hacia abajo y diseño de arriba hacia abajo son utilizados como sinónimos.

Pruebas (PARAB) y desarrollo de arriba hacia abajo si son efectivamente sinónimos, ya que representan un ordenamiento en la secuencia de codificación y pruebas modulares.

Diseño de arriba hacia abajo es algo distinto. Un programa de computadora originalmente diseñado de arriba hacia abajo puede ser ensamblado incrementalmente de arriba hacia abajo o de abajo hacia arriba.

Pruebas de abajo hacia (PABAR) se interpretan usualmente en forma equivocada como pruebas no incrementales, debido a que éstas comienzan en forma identica (cuando los módulos del nivel jerárquico más bajo son validados). Pero como aquí se señala, pruebas de abajo hacia arriba (PABAR) es una alternativa de la estrategia incremental de la fase de integración.

La estrategia 'PABAR' comienza con los módulos del nivel jerárquico más bajo dentro de un diagrama de estructura es decir, con aquellos módulos terminales que no requieren llamar otros modulos. (módulos 'B','C', y 'D' de la figura 2.4.1.). Una vez validados éstos módulos terminales no existe una estrategia óptima que permita definir el camino a seguir. El único requerimiento de las pruebas en el nivel jerárquico superior es la obligación de haber validado todos y cada uno de los módulos subordinados al módulo seleccionado.

Para validar 'A' será necesario haber probado 'B', 'C' y 'D' en el diagrama de estructura de la figura 2.4.1, que con lleva la necesidad de elaborar módulos 'direccionadores' que contengan casos de prueba que especificamente validen entradas y salidas

de cada módulo.

La estrategia 'PABAR' comienza con el módulo principal del programa de computadora. Para probar este primer módulo será necesario diseñar módulos 'ciegos' que simulen las funciones de los módulos subordinados. (para el caso de la figura 2.4.1 hubiese sido necesario diseñar módulos 'ciegos' para 'B', 'C', y 'D'), una tarea que en general resulta más complicada que la generación de módulos 'direccionadores'.

La función de los módulos 'ciegos' es generalmente mal interpretada, ya que no debe ser su función alarmar 'llegue a la subrutina B', o simplemente regresar el control al nivel jerárquico superior, sin antes simular su funcionamiento.

Una ventaja asociada con 'PABAR' es evitar la posibilidad de traslapar las fases de diseño y pruebas, ya que no debe comenzar la etapa de pruebas hasta no haber diseñado el último de los módulos del programa.

Asimismo, el problema de dejar inconclusas las pruebas de un módulo, comenzando las de otro, debido a las diferentes funciones por simular y codificar en los módulos 'ciegos', desaparece en 'PABAR'.

Los problemas asociados con la imposibilidad o dificultad de

generar casos de prueba dentro de una estrategia 'PARAR', no existen en las pruebas 'PABAR', ya que se asocia un caso de prueba a cada módulo 'direccionador'. Las pruebas se imponen directamente a los módulos subordinados, evitando la necesidad de contemplar otro módulos del programa de computadora.

En conclusión, podríamos señalar que la complejidad en el diseño de los casos de prueba de la estrategia 'PARAR' y la disponibilidad de herramientas de prueba para simplificar los módulos de prueba 'direccionadores' orillan a respaldar la técnica 'PABAR' para la integración incremental de programas de computadora.

3.5 Técnicas De Revisión

El proceso de desarrollo de programas en sus diferentes fases, requiere de revisiones formales al final de cada etapa del desarrollo. Estas revisiones se conducen con el objeto de permitir al cliente, usuarios, diseñadores y administradores del proyecto la evaluación del progreso logrado.

Una de las razones que justifican el proceso de 'revisión' a lo largo del desarrollo de programas es que se ha comprobado que las revisiones aumentan la confiabilidad del producto final. La confiabilidad de un programa de computadora se puede definir como la probabilidad de que el programa opere por un periodo de

tiempo sin errores de programación, en la máquina para la cual fue diseñado [JEN79]. En otras palabras, podemos decir que las técnicas de revisión permiten disminuir el número de errores en un programa de computadora.

Las revisiones tienen como objetivo principal la verificación y la validación del producto que se obtiene en cada una de las fases del proceso de programación.

Validar un producto de programación consiste fundamentalmente en evaluar su apego a los estandares y guías que se tienen para la elaboración de un producto determinado: RPC, DPC, codigo, etc., y verificarlo consiste en evaluar su contenido para que sea consistentente con los objetivos o requerimientos que motivaron su realización.

La evaluación de cualquier producto es difícil. Las revisiones son técnicas que permiten a la administración del proyecto una visibilidad adecuada del avance del desarrollo y por otro lado, permiten a los mismos diseñadores encontrar fallas.

Es muy importante mencionar que los programas, aparte de ser escritos para una máquina computadora, deben ser escritos y documentados para humanos [WEI71]. Es particularmente importante que el código generado sea susceptible de ser revisado, y no sólo ésto, sino que más delante sea mantenible, transferible, y modificable. Además, todo documento generado en el proceso de desarrollo de programas deberá ser formal para que

su revisión se facilite.

Las revisiones son actividades humanas y deben de interpretarse como pruebas que los productos del desarrollo de la programación deben de satisfacer. Las tecnicas de revisión más usadas son la 'inspección o auditoría' y las 'caminatas' (del ingles walkthroughs).

El objetivo de las inspecciones y de las caminatas es validar y verificar cada producto que se genere en el proceso de la programación. Cabe notar que el objetivo de una revisión es encontrar errores y no definir la forma de resolverlos. Las soluciones deberan proponerse despues de que se hayan detectado los errores y su implantación después de la revisión y no durante la misma.

El material a revisar deberá distribuirse con suficiente antelación a los participantes a fin de permitirles la lectura y comprensión del mismo.

Las revisiones se pueden realizar en las distintas etapas del proyecto, pero especialmente se deberán revisar los siguientes documentos:

- El planteamiento de objetivos (PO)

- El analisis de requerimientos (RA)

- Los requerimientos del programa de computadora (RPC)

- Los documentos de arquitectura: base de datos, flujo del sistema, etc. (ARQ)

- El diseño del programa de computadora (DPC)

- El código de los programas (COD)

- Los planes de prueba (PLP)

- Los procedimientos de pruebas (PRP)

- Los manuales de usuario (MUS)

Para cada revisión, el autor o autores del documento, seleccionan y distribuyen una lista de temas a revisar y una vez en la reunión el autor narra, línea por línea o resumiendo, el contenido del documento.

Los comentarios de los participantes deberán ser precisos y claros, por ejemplo, "el contador debe incrementarse en 2 y no en 1". se evitarán los comentarios vagos o picaros como por ejemplo, "falta explicar más" o "este párrafo está muy largo".

En la etapa de codificación, las inspecciones son complementarias a las pruebas realizadas en la computadora. Es en esta etapa y en la del detallado de los módulos donde MYERS recomienda la "caminata". Es aquí donde los participantes en la reunión "juegan a la computadora": con uno o dos casos de

prueba, los revisores ejecutan mentalmente el programa. Los valores de las variables se anotan en un pizarrón o en un papel y los participantes hacen preguntas al diseñador o codificador del programa en caso de duda.

3.5.1 Clasificación De Errores.-

MYERS [MYE79] clasifica los errores que se pueden encontrar en varias clases:

1. Errores por referencia de datos
2. Errores por declaración de datos
3. Errores de computación
4. Errores de comparación
5. Errores de control de proceso
6. Erroes en las interfaces
7. Errores de entrada-salida
8. Otros errores

A continuación desglosamos las listas de posibles errores que MYERS recomienda analizar y que sin ser exhaustivas, nos proporcionan un punto de referencia

Cada clase de errores viene seguida de una serie de preguntas que el revisor se formula para descubrir la existencia de un error de esa clase.

1. Errores por referencia de datos

Han sido asignadas las variables usadas?

Est'an entre sus l'imites los indices de arreglos?

Son enteros los 'indices de los arreglos?

Existen apuntadores colgados?

Es correcta la equivalencia de datos?

Coinciden las estructuras internas usadas en la lectura o escritura con las estructuras externas?

Es correcto el c'alculo de la posici'on de un bit?

Es la estructura de los datos la misma para todos los m'odulos?

2. Errores de declaración de datos

Est'an declaradas todas las variables?

Hay variables impl'icitamente declaradas?

Hay variables con el mismo nombre?

Han sido iniciados los arreglos y cuerdas de caracteres?

Son apropiados al tipo de variable los valores iniciales?

Son las longitudes y tipo de variables apropiados para resolver el problema?

3. Errores de computación

Se hacen c'alculos con variables no-aritm'eticas?

Hay c'alculos entre variables de diferente tipo?

Hay operaciones que usan variables de diferentes longitudes?

Las variables que reciben valores de una expresi'on son de la longitud (presici'on) apropiada?

Hay resultados intermedios que produzcan 'overflow' o 'underflow'?

Hay divisiones entre cero?

Hay errores de presicion debido a base 2?

Puede caer fuera de un rango significativo el valor de una varible?

Se usa la precedencia en las operaciones correctamente?

Son correctas las divisiones con n'umeros enteros?

4. Errores de comparación

Son consistentes las comparaciones entre variables?

Son del mismo tipo las variables que se comparan?

Son correctas las relaciones de comparaci'on?

Son correctas las expresiones 'booleanas'?

Hay mezcla de comparaciones y expresiones 'booleanas'?

Hay comparaciones de fracciones en base 2?

Se usa correctamente la precedencia de los operadores relacionales?

5. Errores en el control del proceso

Terminar'a cada una de las repeticiones?

Terminar'a el programa?

Se ejecutan las repeticiones al menos una vez?

Hay repeticiones que se ejecutan una vez m'as o una vez menos?

Est'an bien marcados el principio y el fin de las repeticiones?

Hay decisiones no exhaustivas?

6. Errores en las interfaces

Se usa el n'umero adecuado de argumentos al llamar a una subrutina?

Son los argumentos del mismo tipo tanto en la llamada como en la Subrutina?

Hay constantes que se pasan como argumentos de subrutinas?

Es consistente la definicion de las variables globales en cada uno de los m'odulos que las usan?

7. Errores de entrada/salida

Son correctos los atributos de los archivos?

Son correctas las instrucciones de apertura de archivos?

Est'an de acuerdo las especificaciones de formatos con las instrucciones de lectura y escritura?

Est'an de acuerdo las variables que reciben valores

con el tamaño de los registros externos?

Se prevee las condiciones de fin de archivo?

Se prevee los errores de entrada/salida?

Hay alg'un error en el texto de los reportes o mensajes?

8. Otros errores

Hay variables no referidas en el programa pero que est'an en las listas cruzadas?

Es esperada la lista de los atributos?

Son importantes los mensajes de aviso o advertencia en caso de que se produzcan?

Se omiti'o alguna funci'on en la codificaci'on?

Se verifican las entradas como legales?

En algunos casos, el proceso de revisión también sirve para observar las habilidades del programador o diseñador. Sin embargo, los adiministradores de un proyecto deberán de tener cuidado de no usar las revisiones como herramientas de evaluación de personal.

3.6 Técnicas De Diseño De Casos De Prueba

Los métodos de diseño de casos de prueba pueden ser clasificados en dos categorías:

1. Métodos de análisis de entrada-salida (caja negra)

Los métodos de análisis de entrada-salida son los que permiten diseñar casos de prueba a partir de la especificación del programa, sin requerir del conocimiento de la forma en que el programa ha sido realizado (GOO80 y MYE 79).

2. Métodos de cobertura de lógica (caja transparente).

Los métodos de cobertura de lógica (GOO80 y MYE79) son los que permiten diseñar casos de prueba a partir del diseño y código del programa.

Ninguno de los dos tipos de métodos puede ser considerado completo o ideal. Ambos tienen sus propias ventajas y deficiencias.

La práctica actual recomienda probar programas con casos de los dos tipos. El uso de estos métodos no garantiza que los casos de prueba diseñados sean necesariamente capaces de identificar todos los posibles errores de un programa. Sin embargo, de ser empleados correctamente permiten obtener un número reducido de casos de prueba, con alta probabilidad de identificar errores.

3.6.1 Análisis De Entrada-Salida -

La metodología de análisis de entrada-salida es un conjunto de principios y prácticas utilizados en ingeniería de computación para diseñar casos de prueba de programas (y sistemas) de cómputo.

Esta metodología permite diseñar casos de prueba a partir de la especificación escrita del programa. Se visualiza el programa desde un punto de vista externo y no se requiere conocer el diseño interno del mismo.

Se ha demostrado en la práctica que el uso apropiado de esta metodología permite reducir el número de errores de computación. La metodología de análisis de entrada-salida se fundamenta en los siguientes 3 principios:

PRINCIPIO 1. Un programa sin errores arroja el resultado correcto para todas y cada una de las entradas posibles.

En base a este principio se puede pretender encontrar errores en un programa sometiendole a casos de prueba diseñados para probar que la salida es incorrecta ante al menos una de las posibles entradas.

El número de entradas posibles para un programa es, por lo

general, elevado y frecuentemente infinito. Raramente es factible probar todas las entradas posibles.

PRINCIPIO 2. El costo y el tiempo necesario para probar un programa se incrementa conforme aumenta el número de casos de prueba.

En base a este principio se establece el objetivo del análisis de entrada-salida.

El objetivo del análisis de entrada-salida es el de obtener un número reducido de casos de prueba, que a pesar de ser reducido sea capaz de encontrar todos (o la mayoría de) los errores. El conjunto de casos debe poseer una alta probabilidad de encontrar errores.

PRINCIPIO 3. Es factible agrupar todas las posibles entradas al programa en un número finito de subconjuntos (comunmente llamados clases), agrupando elementos con características similares, de tal manera que la existencia de una salida incorrecta ante una entrada en la clase implique que el resto de las entradas en la clase también serán procesadas incorrectamente por el programa.

En base a este principio, se puede pretender seleccionar

entradas típicas de cada clase para formar un conjunto de casos de prueba que, al menos en el contexto de clases, "cubra", todas las entradas posibles.

Es posible asociar las entradas a un programa en una variedad de agrupamientos distintos. Y no todos los agrupamientos (conjunto de conjuntos) tienen la misma probabilidad de arrojar casos con alta probabilidad de encontrar errores.

La efectividad de un agrupamiento dado depende de la habilidad e ingenio del analista.

Existen sin embargo algunas prácticas y recomendaciones que pueden ser de utilidad al analista en su esfuerzo por agrupar en clases las entradas.

El uso del análisis de entrada-salida como metodología para diseñar casos de prueba ofrece dos notables beneficios.

1. El hecho de no ser necesario el conocimiento del código o diseño del programa para obtener los casos, permite desarrollar la actividad de planeación de pruebas en paralelo con las actividades de diseño y codificación de programas.

2. El hecho de ser necesario un análisis minucioso de la especificación del programa a probar permite frecuentemente identificar deficiencias en la

especificación misma (errores conceptuales, claridad, etc).

La desventaja principal de esta metodología es que el éxito de los casos de prueba es sensiblemente dependiente de la habilidad y creatividad del analista.

3.6.2 Procedimientos Para El Análisis De Entrada-salida. -

El análisis de entrada-salida para el diseño de casos de prueba puede realizarse en cuatro etapas:

a) Análisis conceptual.

b) Caracterización de la salida.

c) Caracterización de la entrada.

d) Selección de casos de prueba.

3.6.2.1 Análisis Conceptual. -

El análisis conceptual de la especificación del programa a probar es el paso más importante. Un buen o mal análisis es la clave del éxito de los casos de prueba y puede significar la

diferencia entre seleccionar casos de prueba al azar o diseñar casos con alta probabilidad de encontrar errores.

Este paso consiste en encontrar respuestas a las siguientes preguntas:

1. Cuál es el objetivo del programa y cuál es el problema a resolver?

2. Cuál es el concepto clave sobre el cual se fundamenta la solución del problema?

3. Qué condiciones existen que pudieran incrementar la susceptibilidad del diseño y código a errores de computación?

La primer pregunta es sencilla, puesto que su respuesta se encuentra en la especificación del programa.

La segunda pregunta es generalmente la más difícil. Raramente se encuentra su respuesta en la especificación. Se deben aclarar los siguientes puntos:

a) Condiciones que aseguren la existencia de soluciones (factibilidad).

b) Dependencia funcional entre entrada y salida del programa.

La tercer pregunta es también difícil. La respuesta depende de la habilidad del analista para identificar aquellas condiciones que son susceptibles a errores de programación. La recomendación es identificar condiciones que, de cumplirse o no, implicarían en cada caso un procedimiento o camino diferente para llegar a la solución. La existencia de estas condiciones implica a su vez la existencia de elementos de toma de decisiones en el programa. La experiencia indica que son éstos los elementos del programa que poseen una mayor probabilidad de ser codificados en forma equivocada (WHI80).

3.6.2.2 Caracterización De Entradas Y Salidas. -

La caracterización de entradas y salidas consiste en agrupar en clases, las entradas y las salidas con características similares.

El proceso de agrupar las entradas y salidas en clases es un procedimiento heurístico; sin embargo, una caracterización exitosa de las entradas y las salidas requiere de un análisis minucioso de la especificación y de un completo entendimiento de los resultados del análisis conceptual.

La entrada a un programa es el conjunto de datos procesados al

ser ejecutado el programa para producir un conjunto de datos de salida.

Una entrada arbitraria puede ser considerada como elemento del conjunto de todas las posibles entradas. Puede también suponerse que toda entrada está constituida por componentes, donde los componentes son a su vez conjuntos de datos relacionados de alguna manera por las condiciones establecidas en la especificación del programa y por las condiciones descubiertas en el análisis conceptual.

Para ilustrar, considérese el caso de una entrada, que según la especificación consta de: una bandera con dos posibles estados, una matriz cuadrada y una secuencia de caracteres alfanuméricos que puede escogerse entre N secuencias válidas. En este ejemplo se puede decir que la entrada consta de 3 componentes: la bandera, la matriz y la secuencia.

De la misma manera, una salida arbitraria del programa puede considerarse como elemento del conjunto de todas las salidas posibles. Cada salida puede además ser dividida en componentes.

Esta visualización abstracta de las entradas (y las salidas) de un programa permite definir las clases, en términos de las condiciones establecidas sobre los componentes de las entradas (y salidas).

El agrupamiento de las entradas en clases puede lograrse mediante 2 criterios distintos.

CRITERIO 1.-

Se consideran una a una todas las condiciones que la especificación y el análisis imponen sobre los componentes de entrada y se definen las clases de entrada en términos de éstos. Si existen razones para suponer que en una clase, un subconjunto de las entradas serán tratadas por el programa de una manera substancialmente diferente, entonces se subdivide la clase mencionada en subclases. El agrupamiento deberá incluir todas las posibles entradas al programa.

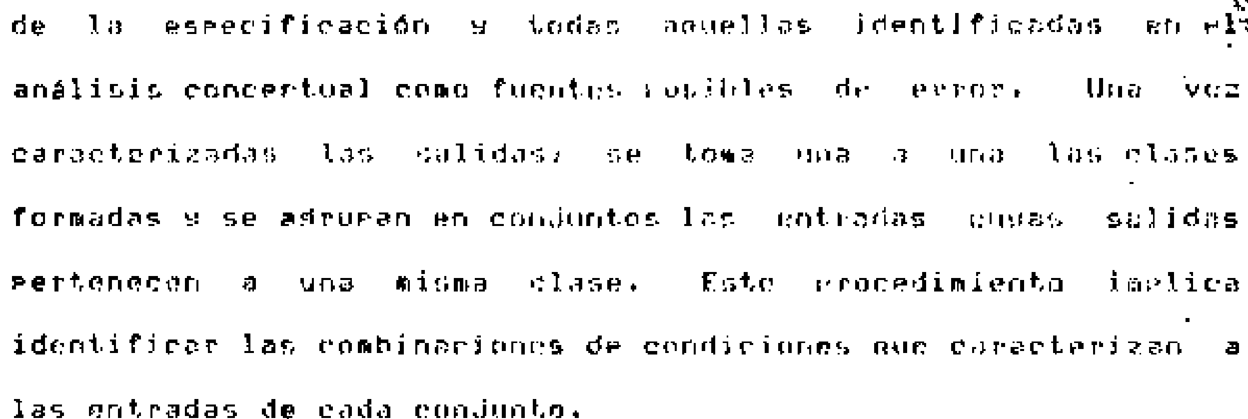
Este criterio tiene la desventaja de no considerar el conjunto de salidas posibles como un factor para realizar el agrupamiento de las entradas.

Quién utilice este criterio no puede mas que confiar en que el agrupamiento resultante es lo suficientemente fino, que le permitirá lograr con los casos de prueba un muestreo razonable del conjunto de salidas.

Se corre el riesgo de que un conjunto importante de salidas nunca sea puesto a prueba por los casos seleccionados. El siguiente criterio no sufre de esta desventaja.

CRITERIO 2.-

Se consideran una a una todas las condiciones impuestas sobre las componentes de salida y se definen las clases de salida en términos de éstas. Deberán considerarse todas las condiciones


de la especificación y todas aquellas identificadas en el análisis conceptual como fuentes posibles de error. Una vez caracterizadas las salidas, se toma una a una las clases formadas y se agrupan en conjuntos las entradas cuyas salidas pertenecen a una misma clase. Este procedimiento implica identificar las combinaciones de condiciones que caracterizan a las entradas de cada conjunto.

Una vez identificados los conjuntos de entradas asociados a cada clase de salida, se procede a particionar estos en agrupamientos mas finos. Conviene particionar un conjunto dado, si existen razones para creer que algunas de las entradas en este serán tratadas por el programa en una manera diferente y a la vez susceptible a error. El agrupamiento resultante constituye el conjunto de clases de entrada.

Este criterio es superior al criterio. 1. Sin embargo, su aplicación es también considerablemente mas laboriosa.

La caracterización de entradas y salidas debe permitir discriminar entre las que son válidas y las que no lo son.

Una entrada inválida es aquella que, de no existir errores en el programa, origina un código de respuesta asociado a datos inválidos.

Una salida inválida es aquella, que de ocurrir, implica la existencia de error, independientemente de cuál sea la entrada que le dá origen.

Una salida válida es aquella que no es inválida.

3.6.2.3 Selección De Casos De Prueba. -

El conjunto de casos de prueba debe estar compuesto por casos que cubren todas las clases de entrada (con elementos válidos o inválidos) y todas las clases de salida cuyos elementos son válidos.

La experiencia ha demostrado que no basta con seleccionar un caso típico de prueba por clase de entrada. Es conveniente seleccionar también casos en las fronteras de las clases de entrada y salida (MYE79).

Las fronteras de una clase están definidas por condiciones que establecen un límite entre distintas clases.

Para ilustrar considérese el ejemplo de una entrada compuesta por 3 números reales: a, b y c. La clase de entradas tales que la suma de a y b es menor o igual que c posee como frontera el conjunto de entradas tales que la suma de a y b es igual a c.

Es evidente que la presencia de estas condiciones de frontera implica la existencia de elementos de toma de decisiones en el programa. El hecho de seleccionar casos de prueba cuyas entradas y/o salidas yacen en las fronteras, asegura que los

elementos de decisión en el programa serán verificados al ejecutar la prueba.

La experiencia ha demostrado que es conveniente seleccionar un caso típico por cada condición que dé origen a una entrada inválida. Se recomienda no probar todas estas condiciones simultáneamente en un solo caso de prueba.

La justificación de esta práctica radica en el hecho de que el programa debe contar con distintas medidas de protección contra datos inválidos, y todas y cada una de ellas debe ser probada.

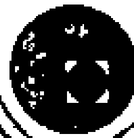
Ante una entrada caracterizada por varias condiciones de datos inválidos, es posible que un mecanismo de protección sea dominante, y por ende puede ser éste el único puesto a prueba.

Otra práctica recomendable para seleccionar casos de prueba consiste en identificar casos asociados con suposiciones que, a juicio del analista encargado del plan de pruebas, el programador pudiera haber hecho al leer una especificación incompleta (omisiones por accidente o por que quien la escribió pensó que se trataba de algo obvio).

Una vez seleccionado el conjunto completo de casos de prueba, conviene investigar la presencia de redundancias para eliminarlas en caso de que existan. De esta manera se reducirá aún más el conjunto de casos, lo cual reduce los costos asociados a esta actividad.

Cabe mencionar una vez más que el proceso de diseñar casos de prueba es difícil y laborioso; se requiere de un esfuerzo considerable de análisis, ingenio y creatividad.

En este capítulo se han descrito una serie de técnicas útiles durante las diferentes fases del desarrollo de programas de computadora. En el siguiente capítulo, se sugiere, utilizando normas o guías, la forma de documentar cada uno de los productos de las fases del proceso de programación y se presentan algunas herramientas automáticas muy útiles en el desarrollo de programas.



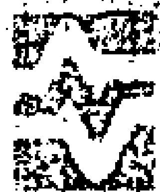
DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

TECNICAS MODERNAS DE DESARROLLO, ADMINISTRACION Y PROGRAMACION

CONTROL DE CALIDAD

MARZO, 1983

Palacio de Minería Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285



4.0 CONTROL DE CALIDAD

4.1 Introducción

Control de calidad es la verificación de la aplicación de un conjunto de normas específicas que permiten desarrollar programas de características específicas y que cumplen con los requerimientos del usuario.

En el capítulo 2 se señaló que conviene subdividir el proceso de programación en una serie de fases para facilitar e inclusive hacer posible su realización y administración. Aplicando esta metodología surge la siguiente secuencia de actividades:

a. Definición inicial de objetivos

b. Establecimiento formal de requerimientos.

c. Diseño del programa de computadora

d. Codificación de módulos

e. Validación de requerimientos a partir de Planes de Procedimientos de Pruebas.

Diversos factores, algunos asociados al grupo de desarrollo de la programación y otros al usuario y a su equipo de cómputo, dificultan el proceso de elaboración de la programación y no favorecen que el producto sea homogéneo; entre éstos se pueden citar:

a) Personal con grados de capacidad muy diferentes.

b) Pequeños grupos de desarrollo sin la especialización suficiente.

c) Clientes interesados exclusivamente en resultados y sin experiencia en un completo proceso de definición de requerimientos.

d) Especificaciones y requerimientos pobremente definidos pero a menudo asociados con objetivos complejos del cliente.

e) Alto grado de deserción de personal, que ocasiona no solo una continua necesidad de actualizar la programación existente, sino también, altos costos de entrenamiento.

f) Restricciones externas e internas, tales como disponibilidad de recursos, tiempos de desarrollo reducidos y limitaciones de personal.

(a) Carencia de funciones de apoyo en el computador que ocasionalmente obliga a los programadores de aplicación a operar como programadores de sistemas, desviando recursos a actividades no directamente relacionadas con el desarrollo de un programa.

(b) Necesidad de tener que emplear programación existente de pobre calidad producida sin normas ni documentación adecuada para su empleo y que requiere de largos periodos de estudio y prueba para entender su funcionamiento. Estos programas podrían ser por ejemplo rutinas de acceso del usuario a su base de datos que se tengan que emplear para la operación del programa en desarrollo.

El objetivo de control de calidad es garantizar que todos los productos de programación satisfagan o excedan los requerimientos que han sido acordados mutuamente entre el comprador y el vendedor. El control de calidad en el contexto de Desarrollo de Programación, es un proceso muy complejo debido a la característica intangible de los programas de computadora, la dificultad de mantener visibilidad en un programa de computadora grande y complejo, la dificultad de especificar adecuadamente los requerimientos, y la ambigüedad inherente de las especificaciones y otra documentación de programación. Un programa de control de calidad (PCC), debe incluir:

INSTITUTO DE INVESTIGACIONES ELECTRICAS

1. Cuidadosa planeación del proyecto, incluyendo difusión y mantenimiento de dichos planes.

2. Especificación detallada de procedimientos para el Control de Calidad.

3. Programación adecuada de actividades y productos, incluyendo el establecimiento claro y preciso de las metas parciales en el proyecto.

4. Documentación adecuada de:

   - Normas
   - Planes de trabajo
   - Metas
   - Procedimientos
   - Productos

5. Pruebas adecuadas de aceptación de los productos de programación

6. Auditorías internas y revisiones periódicas por expertos calificados.

7. Coordinación y participación adecuadas del cliente durante el desarrollo de los productos de programación.

8. Reuniones periódicas entre el personal administrativo y técnico involucrado en el proyecto.

9. Organización efectiva del proyecto con asignaciones específicas de responsabilidad.

10. Un grupo de desarrollo motivado para desarrollar productos de calidad.

La función de Control de Calidad debe recibir información de los productos elaborados durante cada una de las distintas fases en el desarrollo de programación, (RA, DOP, RPC, ARQ, DPC, etc).

A fin de permitir un desarrollo coordinado del proyecto sin tiempos muertos, es esencial que en cada una de las fases del proceso de programación, Control de Calidad inicie sus actividades conduciendo revisiones formales o informales frecuentes que involucren al comprador. Todos los productos principales del proceso de programación (RPC, DPC, etc), deberán de ser auditados en cada fase a fin de constatar:

1. Su integridad con el plan trazado para el proyecto.

2. Un nivel de detalle consistente con la fase de desarrollo del producto.

3. El cumplimiento de las convenciones elaboradas para la documentación.

En las siguientes secciones se presentan un conjunto de normas que deben cumplirse durante las distintas fases del proceso de programación y forman parte de un Programa de Control de Calidad (PCC):

4.2.1 Normas de Especificación de Requerimientos

4.2.2 Normas de Diseño

4.2.3 Normas de Codificación

4.2.4 Normas para Planes de Prueba

4.2.5 Normas para Procedimientos de Prueba

El empleo de herramientas automáticas facilita que el producto cumpla con las normas de control de calidad. Por ello, en la sección 4.3 se introducen algunas herramientas automáticas utilizadas en las diferentes fases del proceso de programación. En particular se describen:

4.3.1 Biblioteca de Soporte de Programas

4.3.2 Checador Estático de Código

4.2.1 Normas De Especificación De Requerimientos

La norma de especificación es un conjunto de reglas establecidas por el líder (analista de un proyecto), que sirve para guiar y controlar el desarrollo del documento de requerimientos de programas (RPC).

Las bases para dotar al RPC con las características deseadas

INSTITUTO DE INVESTIGACIONES ELECTRICAS

de legibilidad, cobertura, alcance, mantenibilidad y [illegible] (ver sección 3.2) se deben contar estableciendo las normas, antes de iniciar el desarrollo del RFC.

La norma es particularmente necesaria en proyectos de gran escala, donde el análisis y el desarrollo de los requerimientos es realizado por un número elevado de analistas. Los distintos hábitos de redacción, presentación de ideas y organización de documentación que generalmente se encuentran en un grupo de personas, de no ser normalizados, pudieran dar origen a un RFC mal presentado, difícil de mantener, ilegible y con distintos grados de detalle.

No existe una norma generalizada para la definición de requerimientos que sea aplicable en todos y cada uno de los diferentes proyectos de computación.

La norma, en general, debe ser establecida para cada caso, considerando las condiciones particulares que influyen en su desarrollo, como son: la naturaleza técnica del proyecto, entrenamiento del personal secretarial, número de analistas, facilidades de procesamiento de textos, etc.

A continuación presentamos una lista de tópicos que en forma general deben ser incluidos en una norma de especificación de requerimientos:

4.2.1.1 Criterio De Revisión Y Aceptación Del documento RPC

La norma deberá definir quién o quienes revisarán y aprobarán el documento RPC. Asimismo deberá establecer los enfoques que se emplearán durante las revisiones, por ejemplo: estilo, solidez técnica, apego a las normas, etc.

4.2.1.2 Herramientas De Elaboración Y Almacenamiento del RPC

La norma deberá definir el medio que será utilizado para producir el documento RPC, por ejemplo, procesador de textos, paquete de soporte de desarrollo de sistemas, máquina de escribir convencional, etc. Es también recomendable que se defina el medio que se empleará para almacenar la información, que como ejemplo pueden ser disco magnético, cinta o expedientes. Es recomendable que se defina la forma de accesar la información almacenada, citando una regla para nombrar o bautizar los expedientes o archivos a fin de facilitar la consulta de material de capítulos o secciones del RPC durante su desarrollo y edición.

INVESTIGACIONES ELECTRICAS

4.2.1.3 Organización Jerárquica De Requerimientos -

La norma deberá señalar la manera de organizar los requerimientos del sistema en distintos niveles jerárquicos a fin de simplificar el proceso de descripción para sistemas complejos.

La organización de los requerimientos es dictada por la estructura y el enfoque de los modelos conceptuales del sistema, que son desarrollados y utilizados durante la etapa de análisis (véase sección 3.2.4).

Así por ejemplo, para un sistema complejo, se puede optar por definir los requerimientos agrupándolos en conjuntos de requerimientos: por sistema, por subsistemas, por tareas y por funciones donde cada uno de estos términos denota un nivel jerárquico distinto.

Este tipo de descomposición es particularmente importante puesto que permite identificar aspectos del sistema que son relativamente independientes entre sí, permitiendo a los analistas describir sus requerimientos en forma simultánea.

4.2.1.4 Estructura Del Documento RPC -

La norma deberá establecer los lineamientos que permitan desarrollar en paralelo varias secciones del RPC. Para esto es necesario definir la lista del contenido del RPC,

presentando "a grosso modo" la cobertura del documento, definiendo la forma como se deberán identificar los distintos capítulos y secciones.

Si se pretende utilizar las partes terminadas del RPC mientras otras continúan desarrollándose, entonces la norma deberá asegurar que existe un cierto grado de redundancia en el documento a fin de permitir que cada sección o capítulo sea autocontenido.

Así por ejemplo, si se desea que cada capítulo pueda ser leído en forma independiente, es recomendable establecer, por norma, que cada capítulo del RPC sea dotado de una introducción, lista de referencia, glosario y que las páginas sean numeradas consecutivamente por capítulo.

4.2.1.5 Sintaxis Y Palabras Restringidas

La facilidad de identificación rápida de requerimientos a lo largo del texto del RPC es necesaria, tanto durante la fase de definición de requerimientos como durante las fases de diseño y mantenimiento. En la primera de estas fases es esencial supervisar el desarrollo del RPC, para lo cual es necesario elaborar reportes periódicos con listas de requerimientos terminados, en proceso de descripción y por terminarse. Durante las fases de diseño y operación es generalmente necesario realizar cambios al RPC a fin de

corregir requerimientos erróneos o incluir nuevos requerimientos o requerimientos faltantes.

La posibilidad de localizar rápidamente los requerimientos a lo largo del texto se puede lograr utilizando palabras de uso restringido en combinación con una "sintaxis" bien definida. Así por ejemplo se puede normalizar el uso de un identificador de requerimientos como el que se describe a continuación.

Cada requerimiento deberá ser bautizado con un nombre con la siguiente estructura:

REQUERIMIENTO X.Y.Z.(a): NOMBRE.

Donde:

Z - Indica que el requerimiento "NOMBRE" es el Z-ésimo requerimiento de la Y-ésima tarea del X-ésimo subsistema.

a - Es una clave que indica el estado y que puede ser cualquiera de las letras: O,N,C,D,M

Donde:

O - Indica que el requerimiento ha

permanecido sin cambios desde que terminó en su forma original.

N - Indica que se trata de un nuevo requerimiento, incertado en el RPC después de su aprobación formal.

C - Indica que el requerimiento ha sufrido cambios.

D - Indica que el requerimiento ha sido dado de baja.

NOMBRE - Es el nombre del requerimiento y puede ser de hasta 20 caracteres.

Cuando el texto del RPC se almacena en disco magnético, esta convención permite fácilmente localizar los requerimientos con un programa sencillo o con un editor de textos convencional. De esta manera, la elaboración de un reporte de requerimientos dados de baja o con cambios, es materia sencilla, aun cuando el documento RPC sea voluminoso.

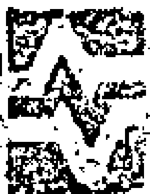
Así como es necesario normalizar el uso de palabras restringidas (REQUERIMIENTO, por ejemplo), también es recomendable prohibir el uso de palabras que puedan crear

confusión en un RPC. En particular, las palabras "Programa" y "Módulo" crean confusión en la fase de requerimientos, pues es en la etapa de arquitectura [3.3.1] cuando se definen los programas y en la etapa de detallado de módulos [3.3.2.8] cuando se definen los módulos requeridos para satisfacer los requerimientos de las funciones tareas y subsistemas en que, para propósitos de análisis y descripción, se ha particionado el sistema.

La norma debe listar todas las palabras de uso prohibido en el RPC.

4.2.1.5 Procedimiento De Actualización Del RPC. -

A fin de controlar el proceso de actualización del RPC, es siempre recomendable que exista un procedimiento formal de actualización, que permita hacer los cambios con un mínimo de burocracia, pero que a la vez, asegure que no se realice un solo cambio sin la autorización de los analistas responsables. La norma deberá establecer dicho procedimiento.



4.2.2 Normas De Diseño -

El objetivo de esta sección es presentar al diseñador un conjunto de normas y lineamientos que le permitan producir un documento donde se reporten los resultados de la etapa de diseño de un programa de computadora.

En esta guía se entiende como "el diseño del programa de computadora" o simplemente "DPC" al reporte de la fase de diseño de un programa, en las etapas del diseño del diagrama de estructura y del detalle de los módulos.

El DPC de un programa deberá presentar la siguiente información en su portada:

- Nombre del Programa
- Fecha y nombre(s) del investigador(es) que elaboró el DPC.
- Fecha y nombre(s) del revisor del DPC
- Fecha y nombre(s) de la persona que aprobó el DPC

(Véase Fig. 4.2.2)

4.2.2.1 Lista Del Contenido De Un DFC

El índice de un DFC deberá contener las siguientes secciones:

1. Panorama general.
2. Referencias.
3. Diagrama de estructura.
4. Diagrama de flujo de información.
5. Descripción del proceso del programa.
   - 5.1 Módulos del programa.

     Para cada componente del programa que aparece en el diagrama de estructura:

     - 5.1.1 Reseña del proceso.
     - 5.1.2 Detalle del proceso.
     - 5.1.3 Variables externas.
     - 5.1.4 Variables globales.
     - 5.1.5 Variables locales.
     - 5.1.6 Parámetros
     - 5.1.7 Estructuras de alto nivel de los datos locales

   - 5.2 Total de líneas de pseudocódigo.

6. Interfaces externas.
7. Variables y parámetros globales.
8. Equipo y lenguaje.

9. Recuperación de fallas.

10. Certificación de calidad.

11. Glosario.

4.2.2.2 Contenido del DPC -

A continuación se describe cada una de las secciones de un DPC:

4.2.2.2.1 Panorama General. -

En esta sección deberá incluirse un resumen del alcance del documento, iniciando con una Introducción, en donde se mencionan brevemente el alcance del programa y el subsistema al que pertenece. A continuación se establece el Objetivo del Documento de Diseño para concluir con un Resumen Funcional, en donde se presentan los componentes de los primeros niveles del Diagrama de Estructura.

4.2.2.2.2. Referencias -

Aquí se presenta la lista de referencias bibliográficas utilizadas en el diseño del programa, incluyendo el documento de Requerimientos del Programa de Computadora (RPC)

INSTITUTO DE INVESTIGACIONES ... FINANCIERAS

4.2.2.2.3 Diagrama De Estructura

Esta sección presenta mediante un Diagrama de Estructura, el ... [illegible] jerárquico y por funciones del programa. Los componentes ó módulos se representan en el Diagrama como un rectángulo o bloque interconectado con otros bloques en forma jerárquica.

Cada bloque en el diagrama de estructura deberá tener un nombre o identificador de módulo y un número de referencia. Los nombres de módulo son asignados por el diseñador del programa, teniendo usualmente en cuenta las restricciones del lenguaje o sistema de cómputo donde se va a implantar el programa.

Para asignar los números de referencia de los bloques, en los diferentes niveles jerárquicos, se utiliza el siguiente método:

1. El bloque superior representa al programa y se le asigna el número 1.

2. A los sucesores del bloque (niveles jerárquicos inferiores) se les asigna los números 1,2,...,N (donde N es el número de sucesores) precedidos de un punto y del número del bloque.

3. Para cada bloque no numerado que tenga sucesores numerados, se repetirá el punto 2 hasta que todos los bloques del diagrama estén numerados.

Las siguientes excepciones deberán ser observadas:

- Cuando un bloque se requiere como sucesor de otro bloque y éste sucesor ya aparece en el diagrama de estructura, entonces no se le asigna número de acuerdo al método expuesto sino que su nombre se repite y su número original se escribe entre paréntesis.

- Cuando un bloque representa un módulo reusable [4.3.2] ya desarrollado y documentado en alguna referencia bibliográfica, no se le asigna número; en este caso particular se escribe el nombre del módulo y entre paréntesis se indica la referencia bibliográfica.

- Las rutinas Fortran o de servicios del sistema no se muestran en el diagrama de estructura.

La Figura 4.2.2.1 muestra un ejemplo de un diagrama de estructura.

4.2.2.2.4 Diagrama De Flujo De Información. -

Este gráfico ilustra las relaciones de control y flujo de información entre el programa y sus principales procesos, bancos de datos, archivos, terminales de entrada y salida, etc.

El diagrama de flujo de información indica, mediante con bloques, el flujo de datos a través del programa. Se tendrá un bloque que represente al programa y varios bloques que representen los datos que se encuentren en disco, en cinta, en memoria compartida, o que provienen de las terminales de video. Para estos bloques se recomienda usar las convenciones estandard: los datos que estan en disco se representan con un cilindro o barril, el programa se representa con un rectángulo, etc. Cada bloque del diagrama deberá tener un nombre.

4.2.2.2.5 Descripción Del Proceso Del Programa. -

La subsección 5.1 (Módulos del Programa) contendrá la descripción de cada uno de los módulos que aparecen en el diagrama de estructura de la sección 3 del DPC. La sección 5.1 se dividirá en subsecciones numeradas de acuerdo a la numeración del diagrama de estructura en la siguiente forma:

- Habrá una subsección por cada bloque del diagrama de estructura.

- El número de cada subsección será el número del bloque, precedido de un 5, por ejemplo, la subsección 5.1 corresponde a la descripción del módulo 1, la subsección 5.2.2.1 corresponde a la descripción del módulo 2.2.1.

- Cada subsección tendrá como título el nombre del módulo. El nombre del módulo 1, será el nombre del programa.

Cada subsección deberá contener la descripción del módulo dividida de la siguiente forma: reseña del proceso, detalle del proceso, variables externas, variables globales y variables locales.

Subsección 5.1.1) Reseña del Proceso.

Usando prosa se describirá desde el punto de vista del usuario y de los posibles usuarios del módulo el proceso que se desarrolla en cada bloque. Deberá incluirse en que condiciones se ejecuta este bloque y una breve explicación de la función de las variables.

Es importante hacer hincapié en el uso y objetivo del bloque y no explicar el detalle del proceso que será presentado en pseudocódigo en la sección de detalle del

proceso.

Con el objeto de poder contrastar la documentación de un programa (RPC, DPC, Código, etc.). Se indicará en esta reseña las secciones o párrafos del RPC que se satisfacen con este módulo.

Subsección 3.1.2) Detalle del proceso.

El detallado de los módulos se cubrió ampliamente en la sección 3.3.3 de este libro. Recordando brevemente, el detalle de un proceso se deberá realizar utilizando el pseudocódigo o LPP. El detalle deberá utilizar única y exclusivamente las estructuras básicas de la programación estructurada: secuencia, decisión y repetición.

El objetivo del detallado de un módulo es documentar en forma estructurada el proceso que un módulo ejecutará. Si el lenguaje de computadora en el cual se va a codificar finalmente el módulo, es un lenguaje estructurado se puede optar por usar directamente el lenguaje de codificación sin incluir las declaraciones de las estructuras de datos que pertenecen a otras secciones del DPC. De esta forma, el ciclo de desarrollo se agilizará considerablemente.

Si el lenguaje de computadora a utilizar en la

codificación no es estructurado, el detalle del proceso deberá especificarse utilizando pseudocódigo, para después traducirse a código como se indica en la sección de Técnicas de Codificación (3.4.1).

Se recomienda que el detalle del proceso de un módulo ocupe aproximadamente una hoja tamaño carta.

Las variables son los identificadores o nombres de datos que un bloque usa durante su ejecución. Estas pueden ser de varios tipos: Globales (comunes a todos los módulos), locales o externas (variables que indirectamente usa un módulo, como por ejemplo los nombres de archivos o los nombres de los campos dentro de los registros). A continuación se aclaran algunos conceptos.

Subsección 5.1.3) Variables externas.

Las variables externas son almacenadas en algún medio (disco, cinta, memoria compartida, etc.) y existen antes, después y durante la ejecución de un programa. Las variables globales y locales solamente existen mientras el programa está en ejecución. Las variables externas incluyen los nombres de programas externos, por ejemplo: subrutinas intrínsecas y de servicios del

sistema.

Usando la lista de variables externas de la subsección 6.1 del DPC y los nombres de programas externos de la subsección 6.3, se listan en esta subsección los identificadores de las variables que el bloque usa como sigue:

- Variables externas de entrada. Son las que el módulo LEE pero no altera.
- Variables externas de entrada-salida. Son las que el módulo (o componente) LEE y escribe.
- Variables externas de salida. Son las que el módulo escribe.
- Programas externos. Son los nombres de subrutinas estandard del lenguaje o del sistema operativo y rutinas o programas documentados fuera del DPC del programa.

Subsección 5.3.4: Variables globales.

Las variables globales son la información común a dos o mas bloques que aparecen en el diagrama de estructura. En Fortran, estas variables se situan en las estructuras

INSTITUTO DE
ELECTRICAS

"COMMONS" programas y se usan para pasar información entre módulos sin hacer uso de argumentos.

→ El uso de COMMONS o variables globales eleva el acoplamiento entre módulos disminuyendo su independencia

3.3.2.- Para usar módulos acoplados por COMMON en otros programas, es necesario que las variables en el COMMON sean declaradas en el programa o módulo que vaya a usar.

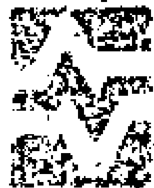
Usando la lista de variables globales de la sección 7 del DPC, se listan los identificadores de las variables que el bloque usa ordenándolas como sigue:

- Variables globales de entrada.

  Son las variables que aparecen en COMMONS de programa y que la subrutina o módulo usa sin modificar.

- Variables globales de entrada-salida.

  Son las variables en COMMONS de programa que la subrutina usa y modifica.



- Variables globales de salida.

  Son variables que estan en COMMONS de programa y que la subrutina crea o define en el COMMONS sin importar el valor de la variable al inicio de la subrutina.

Subsección 5.4.2) Variables locales.

Las variables locales son los identificadores o nombres de datos locales al bloque y que no aparecen en los COMMONS. Para cada variable local se lista la siguiente información:

- Identificador.
- Nombre completo sin abreviaciones.
- Descripcion que explica la variable, si el nombre completo no es suficiente.
- Tipo.
- Dimension.

- Unidades.
- Rango.
- Comentarios.

Las variables locales se agrupan como sigue:

* Argumentos. Los argumentos son variables locales al módulo y pueden ser de tres tipos:

  - Argumentos de entrada. Son los argumentos que el módulo o subrutina recibe y no modifica.

  - Argumentos de entrada-salida. Son los argumentos que la subrutina o módulo recibe y modifica.

  - Argumentos de salida. Son los argumentos que la subrutina genera y su valor es irrelevante al inicio del módulo.

* Internas. Son las variables locales que no son argumentos. El caso típico de una variable local es la variable que se utiliza en una repetición. Sin embargo, las variables locales internas pueden recibir, generar o modificar valores de variables externas en el medio externo.

* Parámetros. Los parámetros son nombres de variables que permanecen constantes durante la ejecución del programa. El valor de los parámetros se define, en Fortran con la instrucción "PARAMETER", y tiene su equivalente en otros lenguajes de computadora.

No hay que confundir el concepto de un parámetro con el valor inicial de una variable, que se define utilizando una instrucción de asignación o una declaración del tipo "DATA", ya que estos valores pueden ser alterados durante la ejecución del programa. Para alterar el valor de los parámetros de un programa es necesario compilarlo nuevamente.

Los parámetros pueden ser definidos a nivel global o a nivel local.

- Parámetros Globales. Los parámetros globales son constantes del programa. El objetivo de usar parámetros es la centralización de constantes al asignarles un nombre y localizándolas en un área global como es el "BLOCK DATA" de Fortran. Los parámetros globales representan datos que permanecerán relativamente fijos en la vida de un programa; por ejemplo: un programa de aplicación avanzado en Sistemas Eléctricos de Potencia que tiene como parámetro el número de centrales termoeléctricas en la

red nacional eléctrica; si suponemos que este número no cambia por algún tiempo, es un buen candidato para ser un parámetro. Si el número cambia continuamente (en cada ocasión que se ejecuta el programa), entonces el número de centrales termoeléctricas no será un parámetro, sino una variable.

La descripción completa de los parámetros globales aparecerá en la sección 7.2 de un TPC.

- Parámetros Locales: Estos parámetros son constantes locales de un módulo y no se usan en alguna otra sección del programa. La siguiente información deberá incluirse para cada uno de los parámetros locales:

- Identificador.
- Nombre completo sin abreviaciones.
- Descripción de la variable si el nombre completo no es suficiente.
- Tipo

- Unidades.

Rango.

- Valor.

Comentarios.

Subsección 5.1.7) Estructuras de alto nivel de los datos locales.

En esta subsección se indica la forma en que las variables locales forman estructuras de datos de alto nivel, por ejemplo: listas encadenadas, árboles, etc.

Ejemplo:

La siguiente figura representa un programa y un archivo. El nombre del programa es P y el nombre del archivo es A.

El programa P contiene, entre otras instrucciones, una declaración de variables:

Declara V1, V2, V3

y una de lectura de variables:

Lee V1, V2, V3.

La declaración de variables se logra en Fortran utilizando las instrucciones de "DIMENSION", "INTEGER" etc., y la instrucción de lectura se logra utilizando un "READ".

```
        programa P                          archivo A
+------------------------------+    +-------------------------------+
|                              |    | c1  |  c2  |  c3              |
|                              |    |-------------------------------|
| Declara v1,v2,v3             |    |                               |
|                              |    |-------------------------------|
|                              |    |                               |
| Lee (archivo A) v1,v2,v3     |    |                               |
|                              |    |                               |
+------------------------------+    +-------------------------------+
```

A los campos dentro del archivo A se les ha asignado nombres; por ejemplo, c1, c2 y c3. Estos nombres de campos son los nombres de las variables externas al programa P.

La clasificación de variables en este ejemplo es como sigue:

* Variables Externas.

- Entrada: c1, c2, c3 (puesto que estamos suponiendo que los campos c1, c2 y c3 serán almacenados en las variables v1, v2 y v3 respectivamente al efectuar la lectura.

- Entrada-salida: ninguna (puesto que el programa P no escribe en el archivo A).

- Salida: ninguna (por la misma razón).

*- Variables Globales.

- Entrada: ninguna

- Entrada- salida: ninguna (note que a estas variables el módulo P les asigna un valor que obtuvo del archivo A).

- Salida: v1, v2, v3 (solamente en el caso de que la declaración de v1, v2 y v3 se haya hecho en un COMMON de programa).

* Variables Locales.

- Argumentos: ninguno

- Internas: v1, v2, v3 (solamente que la declaración de v1, v2, y v3 se haya hecho en un "INTEGER", "DIMENSION" etc. que no estén relacionados con algun COMMON).

Es práctica común utilizar dentro del programa el mismo nombre de la variable en el archivo (nombre del campo), por ejemplo: en lugar de usar "declare v1, v2, v3" es más común usar "declare c1,c2,c3". Sin embargo, es util aclarar que las variables externas al programa no aparecen dentro del programa; aparecen únicamente las variables (globales o locales) que reciben los valores de las variables externas.

Fin del ejemplo.

5.2) Total de Lineas de Pseudocódigo.

Indica el número de líneas de pseudocódigo (con o sin comentarios) usadas en el detalle del proceso de cada bloque.

4.2.2.2.6 Interfaces Externas. -

Usando el Diagrama de Flujo de Información como referencia, se construye la lista de nombres externos al programa.

Subsección 6.1) Variables Externas.

Las variables externas son los identificadores o nombres de los datos que un programa usa (lee, escribe o actualiza) durante su ejecución y que permanecen almacenados en algún medio (disco, cinta, memoria compartida, etc.) antes y/o después de que el programa termine su ejecución.

En el caso de existir un documento de descripción de la base de datos o archivos, se lista únicamente los nombres de las variables usadas en el programa y la referencia bibliográfica. En caso de no existir esta referencia cada variable deberá incluir la siguiente información:

Identificador.

- Nombre completo sin abreviaciones.

- Descripción que explica la variable si el nombre completo no es suficiente.

- Residencia
- Dimensión: cantidad y número por registro.
- Tipo (entero, real, etc.)
- De entrada, salida o ambos.
- Condiciones iniciales.
- Comentarios.

Subsección 6.2) Matriz de Variables Externas.

Esta matriz está construida de la siguiente forma:

Renglones. Nombre de las variables externas, residencia y tipo (entrada, salida, entrada-salida)

Columnas. Nombre de los módulos del programa.

Subsección 6.3) Programas Externos.

Lista los nombres de los programas o rutinas externas llamadas por el programa, acompañadas con una breve descripción del mismo y referencias. Asimismo indica el bloque o bloques donde se usa el programa externo.

Subsección 6.4) Mensajes y Alarmas.

Para cada mensaje o alarma indique su contenido, el destino y el bloque donde se genera.

Subsección 6.5) Condiciones de Activación y Terminación.

Señale por un lado la estrategia utilizada para activar el programa (reloj, eventos, programas, consola, etc) y por el otro, las condiciones que pueden afectar la terminación normal o anormal del programa.

Subsección 6.6) Estructuras de Alto Nivel de los Datos Externos.

Esta sección indica la forma como las variables externas integran estructuras de datos de alto nivel (por ejemplo: listas encadenadas, arboles, archivos indexados, etc.). Si las estructuras aparecen en otros documentos, como el diseño de la base de datos o archivos, deberá indicarse la referencia.

INSTITUTO DE INVESTIGACIONES ELECTRICAS

4.2.2.2.7 Variables y Parámetros Globales.-

Subsección 7.1) Variables Globales.

Para cada variable global se incluirá la siguiente información:

- Identificador.
- Nombre completo sin abreviaciones.
- Descripción que explica las variables si su nombre completo no es suficiente.
- Tipo.
- Dimensión.
- Unidades.
- Rango.
- Etiqueta del COMMON (Si es Fortran).
- Comentarios.

Subsección 7.2) Parámetros Globales.

Para cada parámetro global el programa se incluye:

- Identificador.
- Nombre completo sin abreviaciones.
- Descripción que explica la variable si el nombre completo no es suficiente.
- Tipo.
- Unidades.
- Rango.
- Valor.
- Comentarios.

Subsección 7.3) Matriz de Variables y Parámetros Globales.

Cuando se usan COMMONS, se construirá una matriz de nombres de variables y parámetros que los ubique en los diferentes bloques.

Subsección 7.4) Estructuras de Alto Nivel de los Datos Globales.

Si tanto las variables globales (en COMMONS de programas) forman estructuras de datos de alto nivel (p. ejemplo: listas encadenadas, árboles, apuntadores entre

Y arreglos etc.

4.2.2.2.8 Equipo Y Lenguaje.-

Esta sección indica la computadora en donde se va a ejecutar el programa, y una lista del equipo periférico necesario para la operación del programa.

Indicar el lenguaje de computadora (Fortran 77, Fortran IV, Macro, Basic, etc.), en el que eventualmente se va a codificar el diseño. Y referencias al manual del lenguaje.

4.2.2.2.9 Recuperación De Fallas.-

Esta sección presenta una parte medular en la Fase de Operación del programa de computadora. Aquí deberán indicarse las diferentes acciones que requerirán ser realizadas en casos fortuitos en que por ejemplo: el computador principal falle y utilizando un procesador de respaldo se deba transferir el programa, o bien, los discos en donde reside la información no están disponibles, o finalmente en caso muy común, cuando el programa actualiza erróneamente los archivos del usuario de la programación.

4.2.2.2.10 Certificación De Calidad.-

Esta sección complementa la matriz de requerimientos inicializada en el RPD para mostrar que módulos o bloques cubren los requerimientos del programa de computadora.

4.2.2.2.11 Glosario.-

Toda la terminología técnica usada en el programa es listada, de tal forma que se pueda identificar el significado de los mismos en cualquier parte del RPD.

4.2.3 Normas De Codificación.-

Los estándares o normas de programación son un conjunto de reglas de estilo que permiten que los programas sean fáciles de leer, revisar, modificar y por lo consiguiente mantener.

Este conjunto de normas generales de programación tiene como objetivo unificar el "estilo de programación", sin limitar la creatividad del programador. Se entiende por estilo, la selección adecuada de hábitos de programación que favorecen la claridad y eficiencia del programa.

El empleo unificado de estos hábitos conduce a la producción de buenos programas. Estos deben de trabajar, eso es lo más

importante. La velocidad, la utilización del tiempo máquina, el número de líneas de código u otras variables cuantificables no tienen significado si el programa no trabaja. Podemos tener en cuenta los siguientes compromisos:

a) El programa debe ser correcto antes de ser rápido.

b) Debe ser factible antes de ser rápido.

c) Debe ser claro antes de ser rápido.

d) Debe tener bajos costos de desarrollo.

e) Debe tener bajos costos de prueba.

f) Debe tener costos mínimos de mantenimiento. Organizaciones que emplean masivamente procesamiento de información gastan entre el 50 a 90% de su presupuesto anual en el mantenimiento de sistemas existentes. ( )

g) Debe ser fácilmente modificable. Dentro del dinámico ambiente de una empresa moderna los requerimientos de los programas cambian continuamente, hasta el módulo más pequeño. Un buen programa deberá estar diseñado tomando en cuenta las eventuales necesidades de revisión o cambios.

h) Debe tener un diseño poco complicado. Siempre deberá diseñarse un programa con la idea de que alguna otra persona lo va a mantener.

1) Eficiente. Debe recordarse que en general cerca de [illegible]% del tiempo de ejecución de un programa se gasta solamente en el 3% de las instrucciones. ( ). Para obtener mayor eficiencia el programa deberá tener una estructura lógica, con énfasis en simplicidad y confiabilidad. Después de que el programa esté trabajando se recomienda reescribirlo y optimizar aquellos códigos que consuman más tiempo.

Para lograr programación con estas características, es necesario establecer ciertas reglas para la elaboración de código.

1. Una entrada, una salida, una sola función.

   Es conveniente que todo módulo de programación obtenga una y sólo una entrada, una y sólo una salida y que realice una sola función.

   Algunos lenguajes de programación no ofrecen la facilidad de modularización. En este caso todas las funciones de un programa se realizan en un sólo módulo tornando el programa complicado y por lo tanto difícilmente mantenible.

2. Convención de Nombramiento.

   El objetivo de una convención para nombrar archivos, variables, subrutinas, etc. es producir una documentación

de programas consistente. Como es sabido en cualquier instalación varios analistas, diseñadores o programadores pueden estar trabajando en el desarrollo o mantenimiento de un mismo sistema. Así mismo un sólo analista, diseñador o programador puede estar trabajando en varios subsistemas. Además, varias personas con diferentes formaciones pueden participar en la revisión, modificación o actualización de programación.

Por lo consiguiente, el uso consistente de una técnica de documentación dentro de una instalación mejora notablemente su utilidad, y asegura que los productos sean únicos y por lo tanto fáciles de identificar y catalogar.

Los nombres de símbolos utilizados por el programador para representar variables, archivos o nombres de módulos deben de ser representativos del contenido del mismo.

Un método sencillo para asignar nombres es: escribir una frase significativa del contenido de la variable o función con una, dos o más palabras, comprimir la frase eliminando sucesivamente (de derecha a izquierda) las vocales hasta obtener un nombre significativo o "mnemónico" de longitud adecuado. Además, es importante prestar atención en la pronunciación del mnemónico, ya que éste puede reducir la comprensión o legibilidad del programa.

Pronóstico de Carga - PRNKSA

Apuntador - APNTDR

3. Longitud de los programas y módulos.

El cuerpo de los módulos (programa principal y subrutinas) no deberá exceder de 300 instrucciones ejecutables y excluyendo comentarios explicativos.

4. Asignación y localización de banderas de compilación condicional.

Para la depuración o rastreo del contenido de las variables durante la ejecución de un módulo se sugiere el empleo de la compilación condicional. Esta función se lleva a cabo por medio de banderas de control, las cuales son activadas por el compilador.

Toda subrutina o programa principal deberá incluir ayudas correspondientes de rastreo. Estas ayudas son clasificadas de acuerdo a los tipos de variables o cualquier otra división.

Ejemplo:

```
C          ----------------------
C          * AYUDAS DE RASTREO*
C          ----------------------
IOKEZ   N

        instrucciones ejecutables

IESKP
```

en donde: N, : representa la bandera que se desea activar.

: SKFZ representa la inclusión de la ayuda permanente de rastreo.

: ESKP representa la terminación de la ayuda permanente de rastreo.

5. Definición explícita de variables.

Algunos lenguajes de programación, son limitados y no permiten el uso de identificadores (nombres de variables, archivos y módulos en control) de longitud mayor de 2 caracteres. Otros lenguajes tales como Fortran, Cobol, Algol y Pascal permiten el uso de identificadores de longitud variable.

Los nombres de estas varibles deberán ser explícitamente declaradas empleando las instrucciones correspondientes a cada compilador. No deberá dejarse que el sistema asuma, como en Fortran, que las variables que empiezan con las letras A-H y O-Z sean del tipo real y las que comienzan con I-N sean de tipo entero. Esta práctica, además de que puede alterar el significado en el contenido de la variable, reduce en uno el número de caracteres disponibles para el identificador.

Ejemplo FORTRAN:

REAL* 6 AAAAAA, BBBBBB

```
INTEGER* 2                  CCCCCC, DDDDDD, EEEEEE
DOUBLE PRECISION* 12        FFFFFF
LOGICAL                     GGGGGG, HHHHHH
CHARACTER* 1                IIIIII, JJJJJJ, KKKKKK
```

5. Autodocumentación

Los comentarios sirven para explicar y/o aclarar qué sucede en el flujo de control del programa. Existen dos tipos de comentarios: comentarios de prólogo o encabezado del módulo y comentarios explicativos del cuerpo del módulo.

5.1 Comentarios de Prólogo.

Todo módulo (programa, subrutina o función), debe contener una descripción breve de la función, uso, variables que maneja y aclaraciones pertinentes sobre el uso del mismo.

Estos comentarios deben aparecer inmediatamente después del nombre del módulo y pueden estar delimitados por un marco de asteriscos.

Los comentarios que se pueden incluir en la carátula de todo módulo, por orden de aparición, son:

a) Programa Fuente.

Referencia del área en disco que contiene el módulo en cuestión.

b) Lanzador.

Referencia del área en disco que contiene el conjunto de instrucciones en lenguaje de control del sistema (Job Control Language), para la ejecución de dicho módulo.

c) Propósito.

Describe en una o dos líneas la función que realiza el módulo y, en caso de ser relevante, indica el método de solución empleado.

d) Referencias.

Menciona las referencias en los distintos documentos RPC, DPC, etc., en donde el requerimiento que se está resolviendo por medio de este módulo queda definido.

e) Bibliografía.

Menciona las fuentes bibliográficas utilizadas como base en el desarrollo de este módulo.

f) Limitaciones.

Sirven para indicar en forma breve las restricciones en el volúmen de información y el rango de valores permisibles para ciertas variables.

g) Aclaraciones.

Esta sección sirve para indicar las condiciones de terminación de la ejecución, las condiciones anormales y las suposiciones.

h) Nombre y fecha de implementación.

Identifica el autor y la fecha de terminación asociados a este módulo.

i) Nombres y fecha de revisión(es)

Identifica los revisores y la fecha en que éstas fueron realizadas.

En la figura 4.3.2.1 se muestra un ejemplo ilustrativo.

3.2 Comentarios Explicativos.

Estos comentarios se insertan en el código a fin de indicar cual es el propósito de cada sección, conjunto de comandos o estructuras lógicas dentro del cuerpo del módulo.

El comentario deberá tener un sangrado igual al del nivel de anidamiento del código que describe. Deberá estar delimitado por líneas punteadas y comenzar y terminar con un asterisco.

Ejemplo:

INSTITUTO DE INVESTIGACIONES ELECTRICAS

```
C     ----------------------------------------
C     * IDENTIFICAR EL NUMERO DE SUBSISTEMA *
C     * ASOCIADO A CADA INTERMACIDIO         *
C     * PROGRAMADO                           *
C     ----------------------------------------
```

7. Indentación o Sangrado.

Lo anterior es con el objeto de hacer visible la estructura global del módulo, así como el rango de control que tiene cada estructura lógica. Menos de 3 espacios reduce la legibilidad del módulo, y más de tres es innecesario.

Algunos compiladores tienen la opción de indentar automática (por ejemplo FO.I), con lo cual ya no es necesario el sangrado en el código fuente.

Ejemplo:

```
C     ------------------------------
C     * PROCESO PARA IDENTIFICAR EL*
C     * ARCHIVO DE SALIDA SEGUN EL *
C     * MODO DE PROCESO            *
C     ------------------------------
      IF  (MODO.EQ.0) THEN
C     ------------------------------
C    * MODO DE PROCESO BAJO CONTROL*
C     ------------------------------
```

```
        LFN = LFSUD4 (SBTEMP)
      ELSE
C     -------------------------
C     * MODO DE PROCESO MANUAL *
C     -------------------------
        LFN = LFMAC4 (ARTEMP, CITEMP)
      END IF
```

3. Especificaciones de error de Operaciones I/O Entrada/Salida

Con el fin de codificar módulos que respondan ante un error en dispositivos de entrada/salida, las instrucciones de abrir, cerrar, leer y escribir archivos, deberán contener especificadores de error en la operación o variables que contengan el número de error asociado. Estos especificadores deben de ser utilizados en forma organizada como se muestra en el ejemplo a continuación.

Ejemplo:

```
C  ---------------------------------
C  * ABRIR ARCHIVO DE ENTRADA *
C  ---------------------------------
C  OPEN(UNIT=IM FILE=C:RIESGA, IOSTAT=IOS, ERR=3000)
C  ---------------------------------------
C  * LEER NUMERO DE DIAS A PRONOSTICAR  *
C  ---------------------------------------
```

```
   C  READ(IM,5000, IOSTAT=IOS, ERR=3000) NPDIA
   C  ------------------
   C  *AYUDAS DE RASTREO*
   C  ------------------
:SKPZ 1
      Write(6,5000)'NPDIA=     ', NPDIA
:ESKP
      NPDIA=NPDIA+1
   C  ------------------
   C  * CERRAR ARCHIVO *
   C  ------------------
      CLOSE(IM, STATUS='KEEP', IOSTAT=IOS, ERR=3000)
      IOS=OK/
3000 IF(IOS.EQ. OK/) THEN
   C  ------------------
   C  * ERROR EN ARCHIVO *
   C  ------------------
      WRITE(3,*) '*ERROR*',IOS, 'EN LFN= ', IM
      END IF
      ------------------
      * Formato de lectura *
      ------------------
5000 FORMAT (10.2)
      CALL EXIT
```

```
      END
```

7. Números mágicos.

Deberá prohibirse el empleo de números mágicos dentro del código del módulo. Estos números son constantes que aparecen en el código sin ninguna explicación.

Ejemplo:

```
      IF (.NOT. SW) THEN
      ENTDIF=12.21/0.20 * FLUJO
      ELSE
         DRVENT=(ENTSAL+ENDTIF*ENTPER)/0.433
      END IF
```

Existen equipos que permiten definir un área de parámetros e incluirla en el módulo en el momento de ejecución.

Todos aquellos parámetros deberán declararse explícitamente según las condiciones de cada compilador.

Ejemplo:

```
   INTEGER LU, MA, MIE, JUE, VIE, SAB, DOM
   DATA   LU, MA, MIE, JUE, VIE, SAB, DOM
>        / 1,  2,   3,   4,   5,   6,   7/
```

```
C ------------------------------------------
C  * REALIZAR EL PROCESO PARA LOS SIETE*
C  * DIAS DE LA SEMANA
C ------------------------------------------
  FOR    I=LU, DOM
       -----------------------------
       -----------------------------
       -----------------------------
  END FOR
```

10. Instrucciones prohibidas.

Todo módulo (programa principal o subrutina) deberá ser escrito utilizando comandos secuenciales y las estructuras lógicas de control definidas en la sección 3.3.3 ya que el uso de otro tipo de comandos o estructuras sólo obscurece el contenido y degrada el producto.

Es por ello que se quiere eliminar, hasta donde sea posible, el uso de las estructuras e instrucciones que a continuación se mencionan.

10.1 GO TO

Es el comando más dañino de todos. Debe ser evitado en todas sus ascepciones, excepto en compiladores o interpretes

4.2.4.1.1 La Lista De Contenido Del Plan De Pruebas.

La lista de contenido del Plan de Pruebas se presentará después de la portada. El contenido presentará la lista de secciones que integran el capítulo. No se incluirán las subsecciones en esta lista.

Plan de Pruebas para el Subsistema: Pronóstico

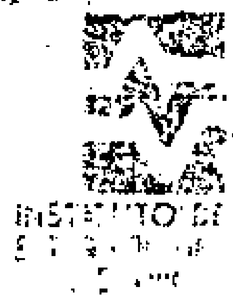
Responsable :

Colaboradores :

Autorizado por :

Fecha de última revisión: Julio 15, 1982.

FIGURA 4.2.4.1. PORTADA DE CAPITULO.



4.2.4.2 La Introducción Al Plan De Pruebas

La introducción al Plan de Pruebas deberá constar dos subsecciones que son:

- Panorama General
- Resumen

4.2.4.2.1 Panorama General -

La primera subsección de la introducción tiene por objetivo presentar un panorama general de la estructura y organización del capítulo. La subsección deberá contener siempre los siguientes puntos:

- Número total de secciones

  Número de secciones dedicadas a cada uno de los distintos aspectos del subsistema como son: interfaz hombre-máquina y tareas.

Cuando sea aplicable, la sección deberá contener los siguientes puntos:

- Secciones que describen pruebas realizables en paralelo

- Secciones que describen pruebas que forzosamente deberán ser realizadas en una secuencia dada.

4.2.4.2.2 Resumen Del Plan De Pruebas.

En esta subsección se deberá presentar la siguiente información por cada prueba propuesta:

- Clave de la prueba (dos números separados por un punto)

- Descripción breve.

- Número de casos que constituyen la prueba.

- Secciones del RPC cuyos requerimientos son cubiertos por la prueba.

Esta información será presentada en forma tabular. (Véase figura 4.2.4.2.2.)

RESUMEN DEL PLAN DE PRUEBAS
DEL SUBSISTEMA DE CONTROL DE PROCESO

CLAVE DE PRUEBA	DESCRIPCION	NUMERO DE CASOS	REFERENCIA RPC
2.1	PRUEBA DEL DESPLEGADO DE ESTADO	15	4.2.3
...	....	..	..
...	....	..	..
...	....	..	..
...	....	..	..
...	....	..	..

FIGURA 4.2.4.2.2. RESUMEN DE PLAN DE PRUEBAS

4.2.4.7 La Lista De Referencias

Esta sección deberá incluir todas las referencias mencionadas en el Plan de Pruebas. La lista deberá contener las referencias que:

a) Describen los requerimientos o especificaciones del programa a probar.
b) Describen los fundamentos de análisis sobre los cuales se basan las pruebas propuestas.

c) Describen las normas a seguir para documentar los planes de prueba.

d) Describen las técnicas para diseñar casos de prueba

A cada referencia se le asignará un número consecutivo.

1.2.4.4 La Documentación De Una Prueba. -

La documentación de una prueba deberá constar de una portada y de seis subsecciones:

Identificación

Análisis

- Caracterización de salidas
- Caracterización de entradas
- Reseña de la prueba
- Casos de prueba.

Las subsecciones identifica, Reseña de la Prueba y Casos de Pruebas serán necesarias para elaborar los Procedimientos de Prueba (PRP) que a su vez serán necesarios para poder ejecutar la prueba.

Se requieren las seis subsecciones para documentar el diseño

de la prueba. La documentación deberá permitir a terceras personas revisar y juzgar dicho diseño.

4.2.1.4.1 La Portada De Una Sección De Prueba. -

La portada de una sección dedicada a documentar una prueba deberá incluir (Ver figura 4.2.1.4.1):

- Número de página: Dos números separados por un guión.

  Clave de la sección: Dos enteros separados por un punto.

  Clave de la prueba: Dos enteros separados por un punto.

- Título de la prueba. Deberá ser conciso, pero a la vez, descriptivo del objetivo de la prueba.

- Referencia a la sección o secciones principales del RFC donde se reportan los requerimientos a probar.

- Nombre y firma del autor

- Fecha de última revisión

SECCION X.Y: SUBSISTEMA XXXXX

PRUEBA X.W

TITULO:

REFERENCIA RPC:

ELABORADO POR:

FECHA DE ULTIMA REVISION

PRUEBA X.Y	PRUEBA FUNCIONAL DE:	
REFERENCIA RPC (SECCIONES):		
REQ. NUMERO	DESCRIPCION DE REQUERIMIENTOS	REFERENCIA RPC (PARRAFOS).

FIGURA 4.2.4.4.2 TABLA DE IDENTIFICACION DE PRUEBA.

4.2.4.4.3 El Análisis.

En esta subsección se deberá documentar el resultado del análisis conceptual, primer paso en el diseño de casos de prueba.

La subsección deberá contener los siguientes puntos:

- Conceptos claves sobre los cuales se fundamenta el diseño de los casos de prueba.

- Lista de condiciones que pueden, a criterio del analista, incrementar la susceptibilidad del código o diseño a errores de computación.

Véase ejemplo 4.2.6.

4.2.4.4.4 Caracterización De Salidas. -

En esta subsección se deberá documentar el resultado de la caracterización de salidas que es el segundo paso en el diseño de casos de prueba.

La subsección deberá contener los siguientes puntos:

- Descripción de la salida, haciendo referencia a las secciones correspondientes del rec.

- Agrupamiento de las salidas posibles en clases.

Se recomienda el uso de tablas para presentar la información en forma concisa. Véase ejemplo 4.2.6.

4.2.4.4.5 Caracterización De Entradas.

En esta sección se deberá documentar el resultado de la caracterización de entradas, tercer paso en el diseño de casos de prueba

La subsección deberá contener los siguientes puntos:

- Descripción de la entrada, haciendo referencia a las secciones correspondientes del RPC.

- Agrupamientos de las entradas en clases.

Se recomienda el uso de tablas. Véase ejemplo 4.2.6.

4.2.4.4.6 Reseña De La Prueba.

La reseña de la prueba deberá ofrecer un panorama general de los casos de prueba propuestos, haciendo resaltar la estrategia utilizada para seleccionar los casos.

Esta subsección deberá contener los siguientes puntos:

- Número total de casos de prueba.

- Reseña de casos o grupos de casos de prueba con características similares.

- Método o métodos que deberán ser utilizados para verificar los resultados de la prueba (ejemplos, inspección de desplegados, inspección de reportes, inspección de listados, vaciado de base de datos, etc).

- Criterio de aceptación de resultados de la prueba.

Véase ejemplo 4.2.6.

4.2.4.4.7 Los Casos De Prueba.

En esta subsección se deberán presentar todos y cada uno de los casos de prueba propuestos.

A cada caso de prueba se le asignará una clave compuesta por 3 números separados por puntos. Los dos primeros números corresponderán a la clave de la prueba. El tercero será un número consecutivo por prueba.

Todo caso de prueba deberá estar compuesto por un conjunto de datos de entrada y un correspondiente conjunto de resultados de salida.

Se recomienda el uso de tablas para presentar información en forma concisa.

El responsable de la prueba decidirá a discreción si la subsección se divide en tantas partes como casos de prueba existan. Cuando la entrada y la salida no sean voluminosas la división de la subsección no será necesaria.

Véase ejemplo 4.2.6.

4.2.5 Normas Para Procedimientos De Prueba -

El objetivo de esta guía es dar un conjunto de normas y lineamientos que permitan documentar de una manera uniforme los diferentes procedimientos de pruebas de aceptación para programas de computadora (PRP).

4.2.5.1 Objetivo De Las Pruebas De Aceptación -

El objetivo de una prueba de aceptación es mostrar formalmente al cliente que un programa cumple con los requerimientos que se establecieron al inicio del desarrollo del programa.

El documento PLP contiene el enunciado de las pruebas, cada prueba con casos y cada caso con la especificación de las entradas y de las salidas. Esta especificación es abstracta,

es decir, no se hace referencia al equipo ni a la forma precisa como se ejecutará la prueba. La forma concreta como se ejecutará la prueba deberá de ser establecida en un documento de Procedimientos de Prueba (PRP) que describimos a continuación.

4.2.5.2 Requisitos Para La Construcción De PRP's -

Los Documentos de Procedimientos de Pruebas se preparan una vez que los programas han sido codificados y probados exhaustivamente y que los analistas de requerimientos, diseñadores y codificadores estén satisfechos con el funcionamiento del programa. El documento permitirá al cliente, en una forma ordenada y cabal, realizar la prueba de aceptación, siguiendo los pasos indicados.

El Documento de Procedimientos de Prueba (PRP) contendrá en detalle, paso a paso, las acciones que el ejecutante o ejecutantes de la prueba deberán de realizar, así como los resultados que deberán obtener como resultado de las acciones.

4.2.5.3 El Documento De Procedimientos De Prueba De Aceptación.

La organización de la documentación de los procedimientos de pruebas (PRP's) deberá realizarse en paralelo a la de los

planes de pruebas (PLP's).

Los procedimientos de pruebas de aceptación reflejan los planes de prueba elaborados a partir del RPC/U, son por lo tanto, procedimientos de pruebas funcionales. Esto quiere decir que, entre uno de los programas identificados en la arquitectura y un procedimiento de prueba determinado, puede o no, haber una relación uno a uno. En algunos casos al terminar de desarrollar un programa se podrán completar los procedimientos de prueba, mientras que en otros casos habrá que esperar a tener desarrollados varios programas para poder completar un capítulo de pruebas funcionales.

Sin embargo, los programas individuales que aparecen en el documento de arquitectura, requieren de pruebas modulares y de construcción, y quizá funcionales. No obstante, se requiere la elaboración de un documento de procedimientos de pruebas sólo cuando se le va a mostrar al cliente el avance de un producto o cuando se va a exponer un producto terminado para fines de aceptación.

Un documento de procedimientos de prueba deberá tener una portada con la información que se muestra en la Fig. 4.2.5.3.

a) Título. Por ejemplo:

PROCEDIMIENTOS DE PRUEBAS DE ACEPTACION

PARA EL PROGRAMA XXXX

DEL SUBSISTEMA ZZZZ, etc.

b) Nombres de las personas que elaboraron el PRP y la fecha de terminación.

c) Nombres de las personas que revisaron el PRP y la fecha de revisión.

d) Nombres, fechas y firmas de las personas que aceptan los productos de programación y son testigos de las pruebas. La firma del PRP constituye la aceptación formal y legal de un programa de computador.

4.2.5.4 El Indice De Un Documento De Procedimientos de Pruebas.

Un PRP deberá contener los siguientes encabezados de sección:

1. Panorama General.
2. Referencias.
3. Requerimientos de Operación.
   3.1. Equipo.
   3.2. Programas.
   3.3. Banco de Datos.
   3.5. Sistema Operativo.
   3.6. Entradas.
   3.7. Salidas.
   3.8. Métodos.
   3.9. Personal.
4. Procedimientos de Pruebas de Aceptación.

Para cada prueba contendrá los siguientes subsecciones:

4.1. Nombre de la prueba.

4.2. Reseña.

4.3. Condiciones.

4.4. Requerimientos Satisfechos.

4.5. Procedimientos de prueba.

4.2.5.5 Contenido Del PRP.

4.2.5.5.1 Panorama General. -

En esta sección se presenta el documento, señalando el alcance de las pruebas y la situación del programa dentro del sistema global de programación.

4.2.5.5.2 Referencias. -

Aquí se listan las referencias bibliográficas y documentos que se usaron para elaborar los procedimientos de prueba. Se incluyen referencias al RPC, DPC, PLP, etc., correspondientes al programa, así como los manuales usados, los reportes empleados, internos, minutas de reuniones con el cliente, etc.

4.2.5.5.3 Requerimientos De Operación. -

Esta sección se divide en diferentes subsecciones a fin de especificar: el equipo, el personal, los programas, el sistema operativo, los archivos, la base de datos, etc. necesarios para efectuar las pruebas.

* Equipo. En esta subsección se indica el equipo (hardware) necesario para hacer las pruebas. Se debe señalar el local o localidades donde se encuentra el equipo, tanto para equipo temporal como permanente, indicando quien es el responsable de que el equipo se encuentre funcionando el día de la prueba.

* Programa. Aquí se presenta la lista con los nombres de los programas que se van a probar o aceptar, incluyendo los nombres de los archivos donde se encuentran los fuentes de los programas y subrutinas que lo conforman, así como los procedimientos (lanzadores o job streams) necesarios para compilaciónes en caso necesario.

* Bases de Datos. Esta subsección enumera los nombres de los archivos que forman la base de datos necesarios para realizar la(s) prueba(s) de aceptación e indica en que condiciones deberán de estar los archivos al iniciar la prueba y la forma de generarlos en caso necesario. Se deberá indicar quien es el responsable de proporcionar los datos para realizar la

prueba.

* Sistema Operativo. Indica el nombre y la versión del sistema operativo bajo el cual se realizará la prueba.

* Entradas. En esta subsección se indican los tipos de entradas con las que se realizan las pruebas. Los tipos de entrada pueden ser, por ejemplo, sensores, teclados, tarjetas, señales analógicas, etc. Se especificará para cada una de las entradas si son simuladas o reales.

* Salidas. Lista la forma de las salidas, por ejemplo: Desplegados, reportes, alarmas, mensajes en la bitácora, etc.

* Métodos. Los métodos de prueba se definen en la sección de técnicas de desarrollo (3.1), por ejemplo: demostración visual y análisis.

* Personal. Se recomienda dividir al personal involucrado en las pruebas en dos grupos uno por parte del cliente, y otro para el personal de la compañía productora del programa. Es importante señalar las responsabilidades de cada una de las personas involucradas: quien monta la cinta, quien inicia el programa, quien interactua con la terminal, quienes son testigos, etc.

4.2.5.5.4 Procedimientos De Pruebas De Aceptación. -

Como ya se mencionó, esta sección será dividida en N subsecciones: una para cada prueba descrita en el Documento de Planes de Prueba (PLP).

Subsección 4.N Nombre de la Prueba.

Subsección 4.N.1 Reseña.

Usando prosa, describa brevemente el objetivo de la prueba.

Subsección 4.N.2 Condiciones.

Seleccione, de las listas de equipo y de personal de las secciones anteriores, el equipo y personal correspondientes a esta prueba.

Subsección 4.N.3 Requerimientos Satisfechos.

Liste las referencias (números de sección, páginas, párrafos, etc.) al documento de Requerimientos de Programa de Computadora (RPC), donde se especifican los requerimientos que en esta prueba se van demostrar.

Subsección 4.N.4 Procedimientos de Prueba.

Para cada caso de prueba indique el nombre del caso, y liste y numere, uno por uno, los pasos que el ejecutante o los ejecutantes de la prueba deben de realizar. Cada paso de la

prueba está integrado por una acción, un resultado y un requerimiento satisfecho.

La acción le indica al ejecutante exactamente qué debe realizar; por ejemplo, encender una terminal, oprimir una tecla, escribir un comando, montar una cinta, etc. El resultado, muestra lo que el programa realiza como resultado de la acción; por ejemplo, un mensaje aparece en la pantalla, se produce un reporte en la impresora, se activa una alarma, etc. Y el requerimiento satisfecho es una referencia a una sección, página y/o línea del documento RPC, donde se especifica el requerimiento que se está cumpliendo.

Esta lista de pasos que detallan el procedimiento a seguir durante pruebas de aceptación pueden presentarse en forma de matriz: una columna para la acción, otra para el resultado, y la última columna para indicar el requerimiento satisfecho.

Como se podrá dar cuenta el lector, esta sección es la más importante y extensa del documento de procedimientos de pruebas de aceptación (PPA) y es el objetivo propiamente dicho del mismo.

1.2.6 Ejemplo. -

A continuación se presenta un ejemplo cuya idea aparece originalmente [...] El objetivo es presentar en forma condensada los requerimientos de un programa de computador a TRIANO y sus planes

y procedimientos de prueba.

Los requerimientos del programa de computadora (RPC) son:

------------------------ RPC TRIANG ------------------------

1. El programa lee 3 valores enteros.

2. Los 3 valores son interpretados como las longitudes de los lados de un triángulo.

3. El programa imprime un mensaje indicando el tipo de triángulo a que corresponden los datos de entrada:

    a) Escaleno

    b) Isósceles

    c) Equilátero

4. Cuando los datos no corresponden a triángulo alguno el programa imprime un mensaje indicando que los datos son inválidos.

---------------- FIN del RPC de TRIANG ----------------

Con este RPC, se desarrolló un documento de diseño (DPC) y codificación del programa. Los siguientes casos de prueba forman el documento de planes de prueba (PLP)

------------------------------ PLP TRIANO ------------------------------

NUMERO DE CASO	ENTRADA			SALIDA
	A	B	C	
1	3	5	4	ESCALENO
2	4	5	5	ISOSCELES
3	5	4	5	ISOSCELES
4	5	5	4	ISOSCELES
5	5	5	5	EQUILATERO
6	5	4	-5	DATOS INVALIDOS
7	5	0	5	.
8	4	3.1	5	.
9	3	4	5	.
10	4	-	5	.
11	5	5	10	.
12	5	5	11	.
13	5	11	5	.
14	5	12	5	.
15	11	5	5	.
16	10	5	5	.

------------------------------ FIN del PLP de TRIANO ------------------------------

Note que es una sola prueba con 16 casos. En la práctica habrá planes de pruebas que incluyan más pruebas o cada prueba con varios casos.

A continuación se presenta el documento de procedimientos de pruebas de aceptación para este ejemplo.

----------- PRP TRIANG -----------

Prueba 1: Triángulos Válidos e Inválidos. 1.1 Reseña.

Verificar que el programa responde con los diferentes mensajes señalados en el RPG.

1.2 Procedimientos de Prueba.

1.2.1 Caso 1: Ejemplo de Triángulo Escaleno.

1. Acción: Ordene la ejecución del programa TRIANG escribiendo lo siguiente:

   RUN [illegible]1687,1427]TRIANG

   Resultado: Aparecen los siguientes mensajes:

   "SEAN A, B, C, LOS LADOS DE UN TRIANGULO."

   "ESCRIBA EL VALOR DE A:"

2. Acción: escriba el número 3

   Resultado: Aparece el siguiente mensaje:

   "ESCRIBA EL VALOR DE B:"

3. Acción: escriba el número 3

Resultado: Aparece el siguiente mensaje:

"ESCRIBA EL VALOR DE C:"

4. Acción: escriba el número 4

Resultado: Aparece el siguiente mensaje:

"ESCALENO"

Requerimientos Satisfechos: con la ejecución de estos últimos cuatro pasos se cumplen los requerimientos 1.2 y 3.a del RPC.

1.2.2 Caso 2: Ejemplo de Triángulo Isósceles.

Los procedimientos para esta prueba son los mismos de la sección 1.2.1, con la diferencia de que las acciones de los pasos 2, 3 y 4 se cambian de acuerdo al caso de prueba 2 que se indica en el PLP de este ejemplo.

Requerimientos Satisfechos: al término de este caso de procedimientos se cumple adicionalmente el requerimiento 3.b del RPC.

1.2.3 Casos 3 al 16.

Los procedimientos de los casos 3 al 16, tienen una forma similar

a los casos de Prueba 1 y 2. Usando el PLP del ejemplo, los procedimientos del Caso 1, se pueden demostrar el resto de los 16 casos de prueba especificados en el PLP.

Requerimientos Satisfechos: al término de los 16 procedimientos de casos de pruebas el total del RPD será satisfecho.

---------------- FIN del PRP de TRIANG ----------------

1.3 Herramientas Automáticas

Los sistemas de programación han alcanzado proporciones enormes, tanto en tamaño como en complejidad. El control de calidad en la programación es un proceso complejo, dada la intangibilidad de los programas de computadora y la dificultad de establecer bases con qué compararlos. Por ello, para controlar adecuadamente la calidad de estos sistemas difíciles de coordinar, es necesario emplear un gran número de recursos humanos, lo cual resulta muy costoso y no particularmente eficiente. Para mejorar esta situación se han desarrollado herramientas automáticas, que no solamente facilitan la labor descrita, sino que aseguran que los distintos programas desarrollados sean homogéneos.

El empleo de herramientas automáticas provee los siguientes beneficios:

1. Permite incrementar el volumen de pruebas a través de su automatización

2. Permite aplicar un mayor vigor en el proceso de auditoría

3. Tiene mayor aceptación por parte de los integrantes del grupo que métodos manuales.

4. Reducen el costo de control de calidad.

Prácticamente cualquiera de las fases del desarrollo de programación puede hacer uso de alguna herramienta automática. Una clasificación funcional de herramientas automáticas ha sido propuesta por Ramamoorthy y Ho (RAM75) de la siguiente manera:

1. Herramientas para el diseño del sistema, las cuales proveen una metodología rigurosa para establecer los requerimientos de diseño.

2. Herramientas para el análisis estático de programas fuente, que auditan la sintaxis del código, generan diagramas jerárquicos de flujo, detectan inconsistencias en la declaración de estructuras de datos e interfaz entre módulos.

3. Herramientas para el análisis dinámico de programas fuente, los cuales colectan estadísticas de comportamiento del programa en ejecución o generan casos de prueba para probar el código.

4. Herramientas para el mantenimiento del sistema, las cuales almacenan los programas que cumplen con las normas de control de calidad y registran los cambios realizados a éstos, ofreciendo siempre la última versión actualizada.

5. Herramientas para verificar el óptimo funcionamiento del sistema, tales como reestructuración de programas o validación de operaciones que podrían llevarse a cabo en paralelo.

La clasificación anterior muestra el espectro de aplicaciones para el cual se pueden escribir herramientas automáticas. A continuación se describen dos herramientas automáticas que asisten en el desarrollo de programación, concretamente en la auditoría estática del código fuente y en el registro y mantenimiento de programas. Estas herramientas son:

1. Auditor de Código

2. Biblioteca de Soporte de Programas.

4.3.1 Auditor De Código -

En virtud de que los programas que se desarrollan en una organización no pertenecen a un individuo o programador, sino que serán utilizados por personas ajenas a éste, es necesario que la programación mantenga un estilo uniforme guiándose por ciertos

normas de programación (sección 4.2.3) de [illegible] forma que [illegible] códigos [illegible] revisen, modifiquen o mantengan aquellas personas directamente interesadas en su aplicación.

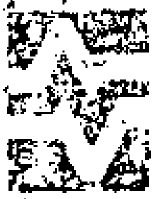
El objetivo principal del auditor de código es controlar la calidad de los programas fuente desarrollados identificando aquellos aspectos que se encuentran fuera de las normas establecidas.

El auditor de código no tiene como objetivo optimizar, ni limitar la creatividad del programador, sólo se desea mediante la unificación del estilo, obtener programas fuente que presenten una estructura coherente y tengan una mayor claridad.

4.2.7.2 Biblioteca De Soporte De Programas

Los programas, si son de buena calidad, una vez escritos son recursos reusables. Es común dedicar tiempo a desarrollar programas que ya existen, dado que su disponibilidad no es conocida por todos. Si existen programas de calidad y son reusables, éstos deben de agruparse en bibliotecas.

El establecimiento de bibliotecas de módulos reusables no sólo reduce el costo o duración de un proyecto al evitar duplicación de [illegible] dentro de una empresa de desarrollo de programación, sino también favorece el control de calidad, empleando módulos de programas ya probados. En algunos proyectos no es posible



emplear módulos reusables, pero aún en estos casos el establecimiento de una Biblioteca de Soporte de Programas es muy efectivo. Aún cuando los módulos hayan sido realizados para satisfacer una necesidad muy específica, su introducción a una BSP permite el control del avance del proyecto, da acceso a la última versión actualizada de los módulos y permite sistematizar el registro de los cambios realizados.

Solamente el empleo de herramientas automáticas permite establecer y mantener bibliotecas de módulos.

La BSP es una herramienta automática necesaria para apoyar el desarrollo de sistemas. El objetivo principal de la BSP es proveer continuamente programas fuente actualizados y legibles, tanto para el ser humano como para la computadora. Estos programas, para formar parte de la BSP, deben ser debidamente checados y deberán de cumplir con las siguientes normas de control de calidad establecidas:

a) Pasar satisfactoriamente la prueba de auditación estática de código fuente, donde se verifican las normas de documentación de encabezados y normas de codificación.

b) Pasar satisfactoriamente las pruebas de compilación.

Para poder introducir un programa a la BSP es necesario que el

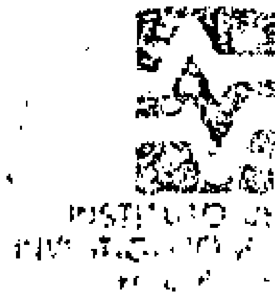
usuario llene una forma de "Forma de Transmisión de Información" (FTI), la cual se muestra en la figura (4.3.2.1.).

```
F O R M A   D E   T R A N S M I S I O N   D E   I N F O R M A C I O N
--------------------------------------------------------------------------
FECHA DE EMISION :                 *   FECHA DE ATENCION :
                     830104        *                          830105
                                   *
SOLICITADO POR    :                *   ATENDIDO POR       :
              VICTOR SEVILLA       *                     RAFAEL SIERRA
                                   *
AUTORIZADA POR IIE :               *   AUTORIZADA POR CFE :
               MAURICIO MIER       *                     ING. CALDERON
                                   *
REFERENCIA EXTERNA :
MINUTA 123, 12 DICIEMBRE 1982

DESCRIPCION DEL ARCHIVO :
                          COMPONENTE   LEIPRG
REALIZA LA LECTURA DE LOS INTERCAMBIOS PROGRAMADOS DE ARCHIVOS ESTRUCT
--------------------------------------------------------------------------

               A S I G N A D O   P O R   D S P G

--------------------------------------------------------------------------
FTI             SUBSISTEMA :                 TI      TIPO DE ARCHIVO :

                PRONOSTICO DE CARGA         (PC)     PROGRAMA FUENTE (1)
                COORDINACION HIDROTERMICA   (CH)     LANZADOR        (2)
                DESPACHO ECONOMICO          (DE)
                TABULADOR DE INTERCAMBIOS   (TI)
                CALCULO DE INTERCAMBIOS     (CI)

     NOMBRE ARCHIVO ORIGINAL         NOMBRE ARCHIVO ULTIMA VERSION

     DSPG*S.TIPS03                   DSPG*S.TIPS03
--------------------------------------------------------------------------
```

FIGURA 4.3.2.1. FORMA DE TRANSMISION DE INFORMACION

Diversas causas pueden dar origen a la necesidad de realizar cambios a los programas de la PSP. Durante el desarrollo del sistema pueden aparecer cambios a los requerimientos, mejoras al producto o errores en el código. Cualquiera que sea la causa para realizar un cambio, será necesario preparar una "Forma de orden de cambio" (FOC), la cual se muestra en la figura 4.3.2.2.

```
*----------------------------------------------------------------------------
*
*          F O R M A   D E   O R D E N   D E   C A M B I O
*
*----------------------------------------------------------------------------
*                                   *
*  FECHA DE EMISION :               *     FECHA DE ATENCION :
*                      830104       *                          830105
*                                   *
*  SOLICITADO POR    :              *     ATENDIDO POR        :
*                 ADRIAN INDA       *                   RAFAEL SIERRA
*                                   *
*  AUTORIZADA POR IIE :             *     AUTORIZADA POR CFE :
*              MAURICIO NIEN        *                  ING. CALDERON
*                                   *
*  REFERENCIA EXTERNA :
*  MINUTA 24. 01 NOVIEMBRE 1982
*
*  RAZON DEL CAMBIO :
*                                          MP
*
*                 MEJORAS EN PRODUCTO        (MP)
*                 NUEVAS ESPECIFICACIONES    (NE)
*                 ERROR EN PRODUCTO          (EP)
*
*----------------------------------------------------------------------------
*
*                 A S I G N A D O   P O R   D S P C
*
*----------------------------------------------------------------------------
*
*  FOC            SUBSISTEMA :                  TI        TIPO DE ARCHIVO :
*       120                                                              1
*
*                 PRONOSTICO DE CARGA           (PC)      PROGRAMA FUENTE (1)
*                 COORDINACION HIDROTERMICA     (CH)      LANZADOR        (2)
*                 DESPACHO ECONOMICO            (DE)
*                 TABULADOR DE INTERCAMBIOS     (TI)
```



```
*                CALCULO DE INTERCAMBIOS   (CI)
*
*
*        NOMBRE ARCHIVO ORIGINAL        NOMBRE ARCHIVO ULTIMA VERSION
*
*        DSPO*S.TIPO03                  DSPO*S:TIPO03
*          --------                        -------
*
*-----------------------------------------------------------------------
```

FIGURA 4.3.2.2. FORMA DE ORDEN DE CAMBIO

El código generado durante todo el proyecto se lleva en listados junto con la historia de las modificaciones. Varios reportes pueden ser obtenidos de la BSP, tales como el "Catálogo de programas fuente en biblioteca", y "Formas de transmisión de información y orden de cambio", los cuales se muestran en las figuras 4.3.2.3, 4.3.2.4 y 4.3.2.5, respectivamente.

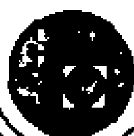
El uso de una Biblioteca de Soporte de Programas, combina con la programación estructurada y diagramas jerárquicos modulares, facilitan el control administrativo del proyecto de desarrollo de programación, al proporcionar continuamente visibilidad a su avance.

Algunos factores han dificultado actualmente el uso y aceptación de estas herramientas automáticas. Entre ellos pueden citarse: La costumbre de utilizar métodos manuales tradicionales, los cuales han sido empleados con éxito en pequeños proyectos de programación y la dificultad de financiar el desarrollo de estas herramientas.

Investigaciones recientes muestran que la tendencia en validación y verificación de la calidad en la programación es la construcción de herramientas automáticas que reduzcan los posibles errores que por omisión se cometen comunmente en los por procesos manuales, tanto aquellas que ayudan en la definición y el diseño de sistemas, como, los que ayudan a evaluar la programación desarrollada.

Mucho énfasis se está poniendo en el desarrollo automático de código a partir de lenguajes para el diseño de programas o pseudo-código (LDP), en la documentación automática de sistemas y en la generación de casos de prueba.

En México, el desarrollo de herramientas automáticas para el control de calidad en la programación esta en su infancia, pero nuestra industria de la computación también está en crecimiento, lo que crea un campo fértil para su desarrollo.



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

TECNICAS MODERNAS DE DESARROLLO, ADMINISTRACION Y PROGRAMACION

ADMINISTRACION DEL DESARROLLO DE LA PROGRAMACION

MARZO, 1983

Palacio de Minería Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

17 Feb 83

CAPITULO 5

INSTITUTO DE INVESTIGACIONES ELECTRICAS

3.0 ADMINISTRACION DEL DESARROLLO DE LA PROGRAMACION

Como se señaló en los capítulos 1 y 2, la planeación, el diseño e implementación de un complejo sistema de programación es un proceso largo y difícil.

Para llevarlo a cabo se requiere del empleo de recursos humanos, financieros y materiales, con el objetivo de alcanzar las metas del proyecto a tiempo y sin incurrir en sobrecostos, entregando programas de buena calidad, que cumplan con todos los requisitos que se le hayan impuesto durante la Fase de Análisis de Requerimientos (2.2).

Administración se define como la ciencia o el arte que permite organizar el trabajo de otros para alcanzar un objetivo. Planear, organizar y controlar son la esencia del proceso de administración. En el desarrollo de programación, al igual que en el de otros sistemas complejos, en la práctica se observa que si bien cada una de las actividades particulares del proyecto no parece ser muy difícil, todas ellas tomadas en conjunto parecen generar un efecto sinergético y generan un problema mucho mayor que el de la suma de sus partes. Por esto, es necesario planificar adecuadamente estos proyectos, máxime que las fallas [illegible] y insatisfacción del cliente observadas en el desarrollo de los primeros grandes sistemas de

INSTITUTO DE INVESTIGACIONES ELECTRICAS

programación, precisamente serán, en general, atribuibles a fallas en la administración.

En este capítulo se revisarán las bases para administrar el desarrollo de programación así como la relación que guarda la complejidad de un proyecto con el esfuerzo de administración [5.7] y la organización de los recursos humanos [5.3]. Siendo con mucha frecuencia un problema severo el sobregiro en los costos de un proyecto de este tipo el capítulo termina con comentarios sobre su estimación y distribución en un proyecto de este tipo, con el objeto de tener una guía para presupuestarlo y para posteriormente vigilar el ejercicio del presupuesto durante el desarrollo del mismo.

5.1 Introducción

Las bases para administrar un proyecto de programación no difieren de las que se emplean para administrar cualquier otro proyecto complejo; deben seguirse diferentes pasos, a saber:

1. Establecimiento de los objetivos del proyecto.
2. Subdivisión del proyecto en subproyectos y fijación de objetivos de cada uno de ellos.

3. Establecimiento de un plan de asignación de recursos y funciones.

4. Asignación de funciones a cada subproyecto.

5.- Asignación de recursos a cada subproyecto.

6.- Monitoreo de resultados parciales.

7. Implementación de acciones correctivas.

Se trata, por lo tanto de un proceso de retroalimentación, tal como muestra la figura 5.1.1

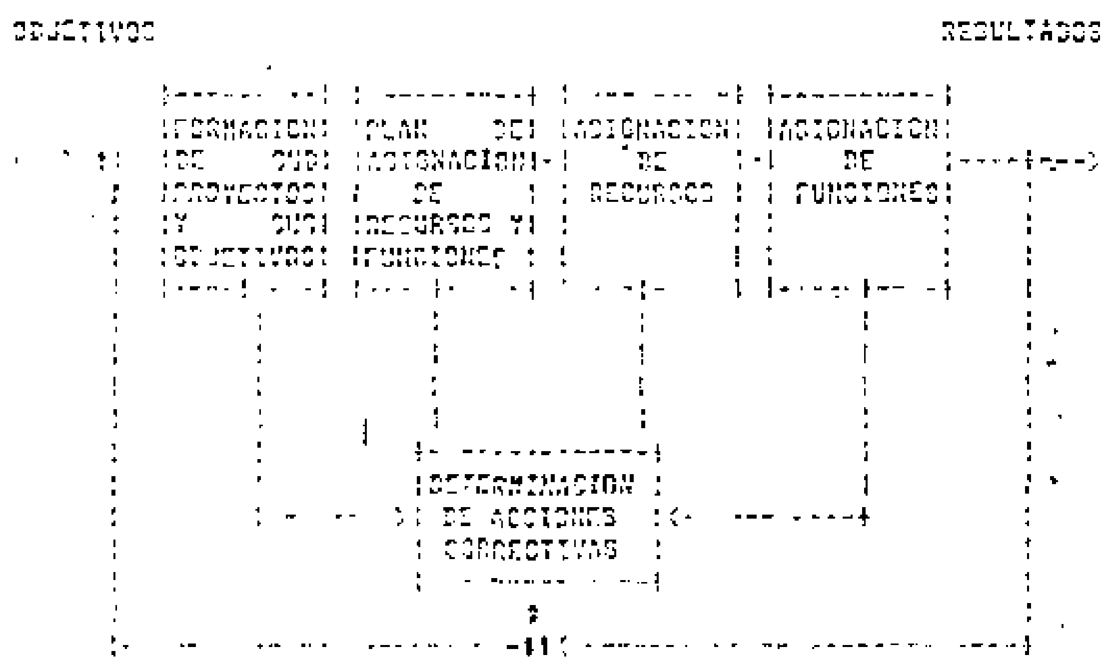


FIG.5.1.1 PASOS DEL PROCESO ADMINISTRATIVO INCLUYENDO UNA RETROALIMENTACION.

Como los proyectos que nos ocupan en esta obra son de un tamaño y complejidad que hacen imposible que una sola

persona pueda realizarlos, es necesario subdividir el proyecto en una serie de subproyectos más pequeños; en función de los objetivos del proyecto deberán asignarse objetivos específicos a cada una de éstas partes; las técnicas contemporáneas de diseño de programación, como del [illegible] y recientemente permiten esta subdivisión en partes cuyo desarrollo puede administrarse más fácilmente.

A continuación se deben establecer las necesidades materiales para alcanzar el objetivo de cada uno de estos subproyectos y las funciones que deben realizar las personas asignadas al mismo; con esta información se establecerá un plan de asignación de recursos y funciones.

Ninguna actividad de administración podrá considerarse completa sino se tiene un sistema para revisar continuamente los resultados que se van obteniendo durante el desarrollo del proyecto y para implementar acciones correctivas si estos resultados difieren de los esperados. Por ello es necesario revisar continuamente los resultados y realizar modificaciones y correcciones cuando resulten necesarias.

Como todos los proyectos deberán realizarse dentro de un marco de referencia temporal y empleando recursos limitados es muy importante fijar el tiempo disponible para la

realización de las diferentes actividades.

Además en alguna de estas será necesario realizar inversiones, es decir, aplicar recursos materiales que tienen cierto valor. Como los beneficios del proyecto, en general, no se empiezan a percibir antes de su terminación, cualquier atraso en la ejecución del mismo implicará demoras en la recuperación de la inversión. Por ello los tiempos de realización de cada actividad del proyecto, que desde luego afectan el tiempo de desarrollo del conjunto, son de gran importancia.

Para el control de estos proyectos existe un modelo matemático llamado Ruta Crítica [illegible] que permite programar las actividades que deben de realizarse y vigilar cuales atrasos en la realización de actividades afectan el desarrollo global del proyecto.

Desde luego el esfuerzo de administración debe estar en función de la complejidad del proyecto. Algunos comentarios sobre el [illegible].

3.2 Complejidad De Los Proyectos

Los costos de mantenimiento de un programa pueden llegar a ser del orden del 40 al 70% de los costos totales de programación, [CHAR82] por esta razón cualquier medida que se tome para disminuirlos tendrá un efecto muy importante sobre el costo de la programación. Una adecuada métrica de complejidad de un programa debe dar un paso importante al factor "mantenibilidad".

Los factores que más afectan la "mantenibilidad" de un programa son: su comprensibilidad, su modificabilidad y la facilidad con que se le pueda probar.

Para medir un atributo existen varios tipos de escalas:

1. Escalas nominales.- Estas emplean calificativos como: es difícil de entender, moderadamente difícil de entender, pero ni llegan a ordenarlos por grado de dificultad.

2. Escalas ordinales. Permiten ordenar por ejemplo programas en función de su atributo, complejidad precisando que el programa A es más complejo que el B, y éste último a su vez más que el C, pero sin señalar cual es la diferencia en el atributo.

INSTITUTO DE INVESTIGACIONES ELECTRICAS

3. Escalas de Intervalos.- Estas no solo nos permiten ordenar una serie de objetos según un atributo, sino también señalan cuantas unidades del atributo separan un objeto del otro.

4. Escalas de Relación.- Estas tienen además de las características de las anteriores la propiedad de índice, cuanto dista métrica de un objeto en particular de una carencia total de la propiedad por medir.

El estado del arte actual en la determinación de la métrica de la complejidad del programa, no permite ir más allá de establecer una métrica de tipo ordinal.

Se ha sugerido que sean cuatro CHAROCI las características para medir la complejidad de un programa: tamaño del programa, estructura de datos, flujo de datos; flujo u flujo de control.

1. Tamaño del programa. Esta característica es la más fácil de calcular y la que se aplica frecuencia, sin embargo, aún aquí existe discrepancia sobre si solo deben contarse las líneas de código ejecutables ó si deben incluirse también las de las estructuras de datos.

En la referencia citada anteriormente se describen metodologías particulares para combinar estos dos

factores en una sola medida para el atributo "tamaño del programa". Puede decirse que, en general, la métrica basada en el tamaño del programa ha sido la más exitosa. La evidencia experimental indica que los programas más grandes son los que tienen los costos más elevados de mantenimiento, sin embargo, si se tienen programas del mismo tamaño se tienen que tomar en cuenta los otros factores (estructura de datos, flujo de datos y el flujo de control) para distinguir entre la complejidad de los mismos.

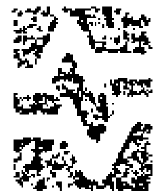
2. Estructura de datos y flujo de datos.- Basados en la forma en que se organizan y se asignan datos, se han propuesto varios métodos para medir la complejidad de un programa en función de la estructura y el flujo de datos.

Uno de estos métodos está basado en la determinación de la distancia entre dos referencias sucesivas al mismo identificador.

Se ha propuesto, por lo menos, otras dos técnicas para evaluar la influencia de éstos factores (estructura de datos y flujo de datos) para medir la complejidad de un programa, pero la evidencia experimental para justificar su uso ha sido bastante pobre.

3. Flujo de control.- Los trabajos más recientes sobre la complejidad de la programación han estudiado el efecto que sobre ésta tiene la estructura de control del programa; un programa de 50 líneas de código, con 25 (if-then, else's) presenta millones de posibles trayectorias de control; una estructura de este tipo es, desde luego, muy difícil de comprender. Todas las nuevas técnicas reportadas en la literatura [HAR82] para estimar la complejidad del programa, basada en la estructura del programa de control, parten de establecer una gráfica dirigida que ilustre el flujo de control dentro del programa. Desafortunadamente para determinar la densidad de transferencias de control y su interrelación en un programa [illegible] y para encontrar métricas adecuadas para medir la complejidad de un programa.

Si bien como hemos señalado en los párrafos anteriores la complejidad de productos de programación no es sólo función del número de líneas de código, éste sí es un buen índice. Podrían clasificarse los productos empleando este parámetro de la manera que muestra la tabla 5.2.1.



LINEAS DE CODIGO	COMPLEJIDAD
a) Hasta 1000	Trivial
b) Entre 1000 y 10,000	Simple
c) Entre 10,000 y 100,000	Difícil
d) Entre 100,000 y 1,000,000	Complejo
e) Más de 1,000,000	Casi imposible.

Tabla 3.2.1 Complejidad de un proyecto de Cómputo.

La relación entre la complejidad del proyecto y su administración puede resumirse en los siguientes términos:

Proyectos Triviales: Estos no necesitan ningún tipo de administración formal. Un programa de computadora que involucra unos cuantos cientos de líneas de código, generalmente lo puede realizar una sola persona en un período de pocos meses. Casi siempre el único factor a considerar, es el límite de tiempo, "¿Cuándo tiene que estar?". Aún cuando las técnicas de análisis estructurado, diseño estructurado y programación estructurada pueden ser de gran ayuda, el desarrollador puede, por fuerza bruta o por sentido común realizar el trabajo a su manera.

- Proyectos Simples: Se pueden decir muchas cosas similares para proyectos 'simples', que involucran programas de computadora o sistemas hasta de 10,000 líneas de código. Este tipo de proyectos es desarrollado, generalmente, por 4 a 5 analista programadores en 1 a 6 meses. Es suficientemente grande y de una duración tal, que se necesita cierto tipo de administración formal en el proyecto. Porque precisamente estos proyectos están dentro del campo de comprensión de un solo responsable, éste los puede organizar, administrar y controlar. Aunque las técnicas que se mencionan en este trabajo pueden ser muy útiles, se argumenta que son innecesarias. Sin embargo y precisamente porque para estos proyectos se siguen con frecuencia los procedimientos tradicionales de desarrollo, (codificación inmediatamente después de un ligero análisis y documentación al final, olvidando además toda documentación del programa), es en ellos donde se tienen los mayores fracasos. Cuántas veces todavía habrá que escuchar a un analista decir, después de oír unos requerimientos muy generales: 'Ahora mismo me pongo a trabajar (codificar) y en unas semanas se lo tengo', para finalmente fracasar.

Proyectos Difíciles) En proyectos que involucren escritura de 10,000 a 100,000 líneas de código, los costos van más allá de la habilidad de una sola persona, como para que ésta lo pueda administrar fácilmente. Este tipo de proyectos deberá realizarse con ayuda de una organización formal del tipo que se describe en la siguiente sección. Puede llegar a requerir de 40 hombres año de esfuerzo. Debe seguirse la metodología descrita: un método de análisis y diseño adecuados. Las técnicas de análisis, diseño y programación estructurada empiezan a cobrar una gran importancia en proyectos de esta magnitud. Con el uso correcto de estas técnicas existe una alta probabilidad de terminar el proyecto a tiempo y dentro de los límites de presupuesto. Un proyecto que involucra hasta 100,000 líneas de código es suficientemente complejo como para que el usuario no tenga una idea inicial clara de lo que debe realizar el sistema, y como su desarrollo puede durar de 2 a 3 años, la probabilidad de que el cliente detecte durante ese periodo nuevas necesidades que desee implementar es grande. Además, cambios tecnológicos pueden justificar otras modificaciones. Es indispensable poner mucho cuidado en la fase de requerimientos (2.7), estableciéndolos en interacción con el cliente y documentándolos claramente; si hubiere cambios deberá llevarse un cuidadoso registro de los

INSTITUTO DE INVESTIGACIONES ELÉCTRICAS

dinámica y sus efectos sobre el diseño y desarrollo de la información.

Problemas Escalares: Ante los proyectos "Complejos" (entre 100,000 y 1,000,000 líneas de código), aún el más experimentado director de proyectos tendrá que admitir que siente un cierto nerviosismo (para un proyecto de esta magnitud estamos hablando de aproximadamente 50 a 100 personas involucradas en la programación y con una duración de aproximadamente 3 a 5 años). Esto significa que además de los problemas de los proyectos difíciles pueden presentarse los siguientes:

Cualquier miembro del equipo puede abandonar el proyecto antes de su terminación

Falta de [illegible] niveles jerárquicos de administración del proyecto.

La primera dificultad [illegible] es más que en proyectos menores que la documentación sea una actividad que se realice paralela con el desarrollo del proyecto y no al final, con objeto de tener continuamente información completa sobre subproductos tanto en proceso como terminados. Una parte del programa que deja un [illegible] el cual si está bien documentado [illegible] del proyecto. Sin esta información en general el código no tiene valor.

El segundo problema obliga a establecer una estructura orgánica con responsabilidades bien definidas.

Seguramente el producto no estará dirigido a un solo usuario, sino a un grupo con requerimientos que no necesariamente coinciden. El grupo de análisis y definición de requerimientos deberá tomar en cuenta esta amplia gama de necesidades para llegar a requerimientos que satisfagan al mayor número de potenciales usuarios.

Siguiendo con los métodos convencionales de desarrollo de [illegible], se corre el riesgo de:

No concluir el proyecto.

[illegible]
[illegible] fuera de costos.

Que el usuario rechace el producto final por no ser adecuado a sus necesidades.

La metodología descrita en esta obra podrá garantizar el éxito de un proyecto de este tipo si la dirección del proyecto está respaldado por un conjunto de analistas y [illegible] altamente calificados y motivados.

5.2 Organización De Los Recursos Humanos

Para realizar productos de más de 10 000 líneas de código (ver el Capítulo [illegible] sección 5.7), la organización del grupo de desarrollo de programación deberá reflejar el enfoque [illegible] descrito en este [illegible].

5.2.1 Organización Informal

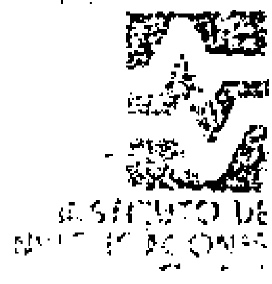
Para productos pequeños no se requiere de una organización formal [illegible] es indispensable que se [illegible] para el desarrollo [illegible] de desarrollo de [illegible] el Capítulo 7. En estos casos [illegible] personas conformarán el grupo de trabajo que consistiría de un Jefe, un Programador Líder, un bibliotecario y varios programadores. El Jefe es el responsable de definir los requerimientos, la arquitectura del programa y el desarrollo de módulos críticos, así como de la [illegible] de módulos no críticos asignados a otros programadores. El Programador Líder respalda en el desarrollo de módulos críticos al Jefe y es el sustituto del Jefe. [illegible] procedimientos de prueba (PLP y PRP) [illegible] responsabilidad de estos dos colaboradores [illegible] el Control de Calidad. Desde luego se está suponiendo que el análisis de requerimientos no requiere de

INSTITUTO DE INVESTIGACIONES ELECTRICAS

conocimientos especializados (control, modelado, sistemas de potencia, etc.) Si este fuera el caso, habría que recurrir a los servicios a tiempo parcial de un especialista. El bibliotecario mantiene las facilidades de soporte y actualiza la biblioteca de [illegible] de prueba, ayuda a la compilación de programas y su ejecución, al [illegible] de la documentación y mantiene un cuidadoso registro de todos los cambios que surgen durante el desarrollo del [illegible].

7.3.2 Organización Formal.-

Cuando el programa es mayor de 10,000 líneas de código la organización descrita anteriormente no resulta suficiente. Se hace necesario contar con una organización formal encabezada por un Jefe de proyectos de preferencia una persona que sea un ingeniero de sistemas con conocimientos tanto en el análisis de requerimientos como técnicos en el área de computación. Su función será principalmente de organización, de coordinación y de control. Deberá organizar el trabajo de acuerdo con la metodología expuesta, y controlar que se cumplan dentro del presupuesto las fechas de desarrollo de las actividades asignadas a cada grupo de [illegible].



3.3.2.1 Análisis De Sistema.

El grupo de Análisis [illegible] estará encabezado por un especialista en el campo analítico del programa que se está desarrollando. También se hará cargo de las actividades relacionadas fundamentalmente con la definición de requerimientos (3.2) y la elaboración de los planes de prueba (3.4.3). En particular las actividades de este grupo pueden ser:

- Definir con el usuario los requerimientos (DRP).
- Elaborar el reporte de análisis (RA).
- Producir un documento de requerimientos de programas de computadora (RPC).
- Organizar la revisión del RPC.
- Producir un plan de pruebas para la integración de la programación al sistema (PLP).
- Analizar resultados de las pruebas de integración del sistema.
- Editar el manual de usuario.

3.3.2.2 Desarrollo De Programación

El Grupo de Desarrollo de Programación, a cargo de un profesional de la computación, tiene a su cargo el diseño del programa (3.3) y empleando las técnicas de desarrollo expuesta (3.4), producirá código de módulos que integrará en subprogramas y programas, y finalmente el sistema. Empleando los planes de prueba [PLP] desarrollados por "Análisis de Sistemas" elaborará los procedimientos de prueba [PRP]. Una lista más detallada de las actividades del grupo de desarrollo de programas podría ser:

- Diseñar el programa de computadora [DPC]
- Diseñar interfaces y despliegues
- Codificar y documentar los módulos
- Estructurar los archivos
- Producir los procedimientos de prueba [PRP]
- Integrar módulos y programas

3.3.2.3 Control De Calid

La función de Control de Calidad desempeña un papel importante en la administración de un proyecto de programación. El Jefe de Control de Calidad realiza las funciones de un auditor al servicio del Jefe del proyecto verificando que las normas sean observadas y que la documentación se mantenga al día. A pesar de su función "policíaca" si el control se maneja con diplomacia y las normas son lógicas y bien definidas y se desarrollaron buscando el consenso de los integrantes del grupo, su función será bien recibida. Control de calidad deberá transmitir a los grupos de trabajo que su función evita [illegible] y [illegible] el éxito del proyecto. También [illegible] a esta función de [illegible] el empleo de herramientas automáticas que serán usadas por los programadores durante la etapa de desarrollo, garantizando que después sus productos pasen la auditoría de calidad sin problemas. La auditoría deberá llevarse a cabo durante el desarrollo del proyecto en forma continua y no solamente hasta el fin en la documentación.

Asociada a las actividades de Control de Calidad existe la función de administración de la configuración que se define como la disciplina que aplica directrices técnicas y administrativas para:

a) Identificar y documentar [illegible] características físicas y funcionales de los elementos que serán administrados.

b) Controlar los cambios en los productos de programación (RPS, DPS, COD, etc).

c) Registrar y reportar los procesos de cambio (FCC) y su implementación.

Los elementos controlados por la administración de la configuración son los productos y los sistemas de los cuales se desea supervisar contenido y formato.

[illegible] el desarrollo de un producto grande, por ejemplo, cambios en los [illegible] [illegible] los interfaces pueden [illegible] efecto [illegible] a través de toda la documentación y el código. [illegible] documentar con cuidado estos cambios y llevar un [illegible] de sus efectos sobre todo el desarrollo del producto.

[illegible] de las actividades de este grupo podría ser la siguiente:

Desarrollar normas de especificación de requerimientos

Desarrollar normas de [illegible]

Desarrollar normas de codificación

Desarrollar normas para nombres variables

Desarrollar normas para planes y procedimientos de prueba

Formar la biblioteca

Revisar el estilo de la documentación

Desarrollar y/o adquirir herramientas automáticas

Controlar modificaciones a los requerimientos de programación

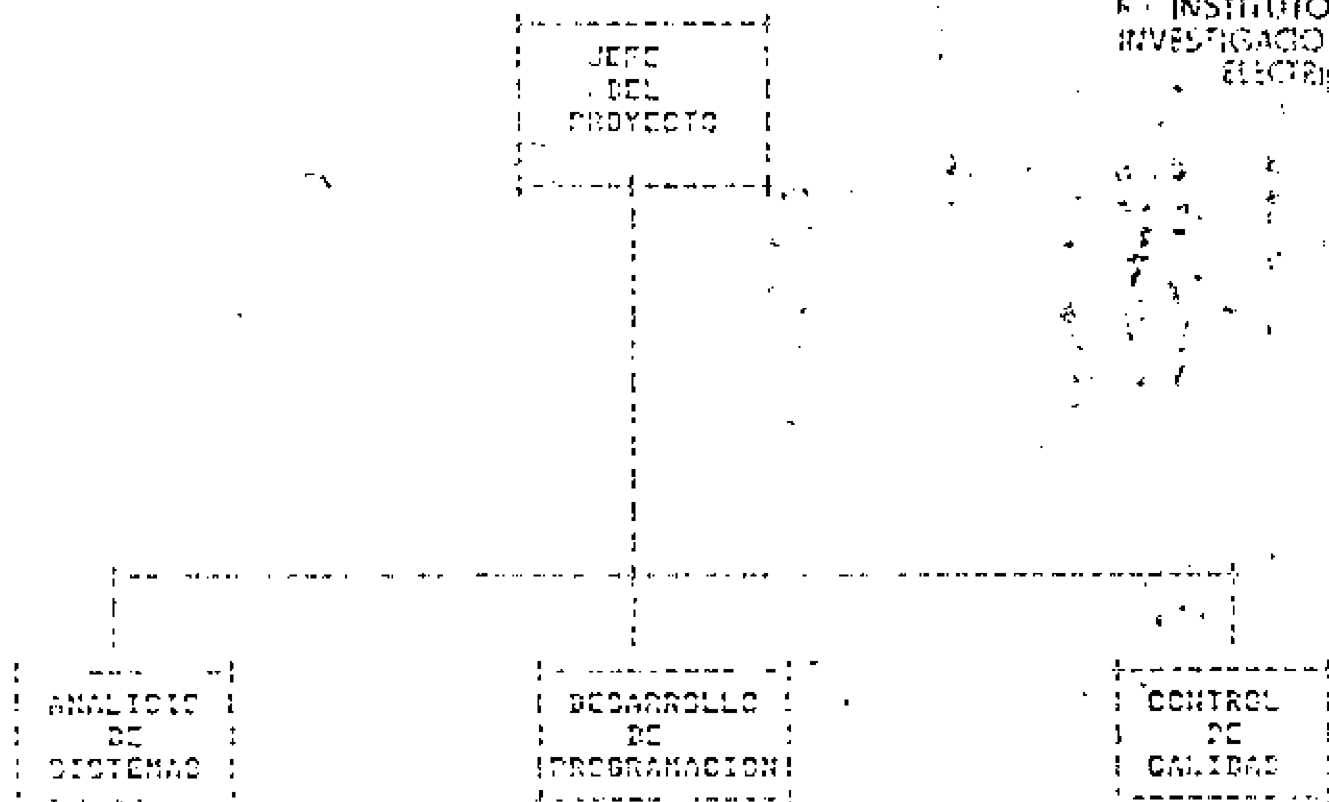
Realizar auditorías

La figura 5.3.2 muestra un posible organigrama de un grupo de desarrollo de programación integrado por 3 áreas específicas:

a) Análisis de sistemas

b) Desarrollo de programación

c) Control de calidad



[illegible]:

1) Documento de objetivos (DO)
2) Requerimientos de programas de computadora (RPC)
3) Planes de prueba (PLP)
4) Manual del usuario (MUS)

1) Arquitectura de programas de computadora (ARQ)
2) Diseño de programas de computadora (DPC)
3) Codificación (COD)
4) Integración
5) Procedimientos de (PRP)

1) Normas
2) Bibliotecas de soporte de programas
3) Auditoría control

FIG. [illegible] ORGANIZACION BASICA DE UN GRUPO DE DESARROLLO DE PROGRAMAS.

2.4 Estimación de Costos

Existe consenso [illegible] de que la estimación de costos es, en el mejor de los casos, un arte impreciso. Esta estimación depende de factores como el número de instrucciones, el lenguaje de programación y la complejidad del programa a elaborar. El primero tiene una métrica precisa, las potencialidades y limitaciones de los lenguajes de programación son conocidas, pero cómo se mide la complejidad? Agréguese a lo anterior el problema siguiente: cuando se estima el costo del equipo (Hardware) de un proyecto se conocen de antemano sus características; en el caso de la programación (Software) la situación es radicalmente diferente. Es necesario estimar un costo de desarrollo de un programa de computadora antes de conocer sus especificaciones precisas, es decir, los requerimientos de la programación [ERPC]. Además como ya se indicó, uno de los indicadores que sin lugar a dudas tiene mayor impacto sobre el costo es la complejidad del proyecto, factor para el cual no existe una métrica adecuada.

Existen, sin embargo, sistemas de estimación de costos desarrollados por compañías, como la RCA y TRW, basados no sólo en [illegible] estimaciones de costos de proyectos similares, sino también en la evaluación de la sensitividad del costo a diversos factores. Estos sistemas han dado

bastante buenos resultados aplicados a la evaluación de la [illegible]. Entre los principales factores tomados en cuenta para [illegible] en esta meta están:

1. El tamaño del programa: su número de líneas de código y el número de datos que maneja.

2. Los atributos del programa: su complejidad y el lenguaje de programación empleado.

3. La cantidad de "código original" que hay que diseñar y compilar; es decir, que no esté disponible de [illegible] anteriores.

4. Las limitaciones en el equipo (Hardware), y en la memoria del procesador.

5. Las limitaciones en el tiempo de ejecución del programa.

6. La confiabilidad y grado de definición de los requerimientos por parte del usuario.

Dentro de este rubro [illegible] el grado de conocimiento que tiene el usuario en la especificación de requerimientos de programas de computadora. Cuando se trata de un cliente que por primera vez define un programa se requerirá más tiempo en el proceso de definición de requerimientos ya que las probabilidades de que identifique o altere las especificaciones durante el desarrollo serán altas, afectando grandemente el costo del programa.

7. El medio ambiente en que se integrará la programación. Deberá analizarse con detenimiento si la programación va a ser integrada a un sistema del cual existe abundante información sobre factores como la base de datos que utiliza, acceso e interfases hombre máquina, o bien a un sistema que se desarrolla simultáneamente con la programación, sobre el cual existen muchas incógnitas que solamente se resolverán durante el desarrollo del proyecto de programación.

DIRECTORIO DE ASISTENTES AL CURSO: TECNICAS MODERNAS DE DESARROLLO Y ADMON. DE PROGRAMACION

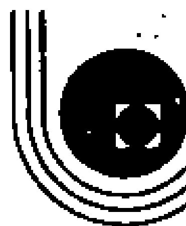
	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
1.	MEDARDO FEDERICO BARRERA GODINEZ Av. Cuernavaca No. 68 Col. Sta. María Nonoalco Mixcoac Del. Alvaro Obregón C.P. 01420 Tel. 563-47-20	BANPECO José María Marroqui No. 81 Col. Centro Del. Cuauhtémoc C.P. Tel. 521-43-80
2.	IGNACIO CARBAJAL CARBAJAL Fernando Ramírez No. 53 Col. Obrera Del. Cuauhtémoc C.P. Tel. 578-52-45	I.M.P. Av. Lazaro Cardenas 152 Tel. 56766-00 Ext. 2527
3.	ARMANDO CEJA SAUCEDO Bocanegra No. 21 INT. 30 Col. Guerrero Del. Cuauhtémoc C.P. Tel.	
4.	GUILLERMO DE LA FUENTE GUZMAN Calle 13 No. 101 Col. Olivar del Conde Del. Alvaro Obregón C.P. 01400 Tel.	BANCO DEL PEQUEÑO COMERCIO DEL D.F. José María Marroquí No. 81 Col. Centro Del. Cuauhtémoc C.P. 06050 Tel. 521-43-80 Ext. 190
5.	JESUS FUENTES FLORES Av. San Bernabe No. 206 Col. San Bernabe Del. Magdalena Contreras C.P. 10300 Tel. 595-60-69	BANCO DEL PEQUEÑO COMERDIO DEL José María Marroqui No. 81 Col. Centro Del. Cuauhtémoc C.P. 06050 Tel. 521-43-80 Ext. 190

DIRECTORIO DE ASISTENTES AL CURSO: TECNICAS MODERNAS DE DESARROLLO Y ADMON. DE PROGRAMACION

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
6.	JAVIER HERNANDEZ ROBERT Nacozari 37 Depto. 30-C Col. San Francisco Del. Mag. Contreras C.P. 10810 Tel.	I.M.P. Av. Eje Central Lazaro Cardenas No. 152 Tel. 567-66-00
7.-	VICTOR MANUEL LACANO ROMERO	BANCO DEL PEQUEÑO COMERCIO DEL DF José Ma. Marroquí No. 81 Col. Centro Del. Cuauhtémocs C.P. 06050 Tel. 521-43-80
8.-	MIGUEL ANGEL MORA ESPINOSA L. Cardenas No. 490-1301 Col. Tlatelolco Del. Cuauhtémoc C.P. 06900 Tel. 597-54-96	CENTRO DE ECODESARROLLO Altadema No. 8 Col. Nápoles Del. C.P. Tel. 523-18-02
9.-	JULIO MODESTO MUÑOZ GALVEZ Huasteca No. 263 Col. Industrial Del. Gustavo A. Madero C.P. Tel. 533-02-73	BENJAMIN MORA GONZALEZ, I.C. Londres No. 71 Col. Juárez Del. Cuauhtémoc C.P. 06600 Tel. 511-85-98
10.-	MARIO PALOMAR ALCIBAR Av. Cuauhtémoc No. 877-3 Col. Narvarte Del. Benito Juárez C.P. Tel. 543-78-85	D.E.P.F.I. UNAM Ciudad Universitaria Tel. 550-52-15 Ext. 4493

DIRECTORIO DE ASISTENTES AL CURSO: TECNICAS MODERNAS DE DESARROLLO Y ADMON. DE PROGRAMACION

NOMBRE Y DIRECCION		EMPRESA Y DIRECCION
11.-	CARLOS PEREGRINA RAMIREZ Calz. Ahuizotla 125 Int. 19 Col. Santiago Ahuizotla Del. Azcapotzalco C.P. 02750 Tel.	UNIVERSIDAD PEDAGOGICA NACIONAL Km. 0.5 Carr. Ajusco Col. Héroes de Padierna Del. Tlalpan C.P. Tel. 652-33-99
12.-	CLEMENTE JUAN PASIO RODRIGUEZ BARRON José Rodríguez González No. 9 Col. Constitución de 1917 Del. Iztapalapa C.P. 09260 Tel. 6910848	D.E.P.F.I. Ciudad Universitaria
-	GABRIEL SANCHEZ BOJORQUEZ Calle Mixcoac No. 21-B Col. Merced Gómez Del. Benito Juárez C.P. Tel.	SEGUROS AMERICA BANAMEX Av Revolucion 1510 Col. Guadalupe Inn Del. Alvaro Obregón C.P. Tel.
14.-	GABRIEL SANCHEZ GUERRERO Bretaña No. 124 Col. Portales Del. Benito Juárez C.P. Tel. 550-52-15 Ext. 4486	DIV. EST. POSG. FAC. ING. UNAM Ciudad Universitaria



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

TECNICAS MODERNAS DE DESARROLLO Y ADMINISTRACION DE PROGRAMACION

CONTENIDO DEL DOCUMENTO DE REQUERIMIENTOS DEL SISTEMA DE ADMINISTRACION DE CIRCULANTE Y TOMA DE DECISIONES

ING. GUILLERMO RODRÍGUEZ ORTÍZ

MARZO, 1983

Pala[illegible] de Minería Calle de Tacuba 5 primer [illegible] Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 A[illegible]o. Postal M-2[illegible]

CONTENIDO DEL DOCUMENTO DE REQUERIMIENTOS DEL SISTEMA DE ADMINISTRACION DE CIRCULANTE Y TOMA DE DECISIONES.

1.0 INTRODUCCION.
1.1 IDENTIFICACION.
1.2 PROPOSITO Y ALCANCE.
1.3 ORGANIZACION DEL DOCUMENTO.
2.0 REFERENCIAS.
3.0 DEFINICION CONCEPTUAL DEL SISTEMA.
3.1 ESQUEMATIZACION DEL SISTEMA, SU FRONTERA Y SU ENTORNO.
3.1.1 PROCESADOR.
3.1.2 IMPRESORA.
3.1.3 TECLADO.
3.1.4 CONSOLA.
3.1.5 EL USUARIO.
3.2 RESENA DEL SISTEMA.
3.3 GLOSARIO DE TERMINOS.
3.3.1 CALENDARIO DE GASTOS E INGRESOS.
3.3.2 CAJA.
3.3.3 CHEQUES.
3.3.4 TARJETA.
3.3.5 DEUDORES DIVERSOS.
3.3.6 ACREEDORES DIVERSOS.
3.3.7 INVERSIONES.
3.3.8 CUENTA EXTERNA.
3.3.9 FLUJO DE EFECTIVO.
3.3.10 TRANSACCION.
3.3.11 MOVIMIENTO DE CUENTA.
3.4 CONVENCIONES.
3.4.1 DESPLEGADOS.
3.4.2 MENSAJES.
4.0 DEFINICION DE REQUERIMIENTOS DEL SISTEMA.
4.1 REQUERIMIENTO 1.1:ACTIVACION DEL SISTEMA.
4.2 REQUERIMIENTO 1.2:SELECCION DE OPCIONES.
4.3 REQUERIMIENTO 1.3:TERMINACION DE SESION.
5.0 DEFINICION DE REQUERIMIENTOS DE LA FUNCION DE SIMULACION DEL FLUJO DE EFECTIVO.
5.1 REQUERIMIENTO 2.1:INICIALIZACION DEL CALENDARIO.
5.2 REQUERIMIENTO 2.2:ACTUALIZACION DEL CALENDARIO.
5.2.1 BAJA DE UN REGISTRO.
5.2.2 ACTUALIZACION DEL CONCEPTO.
5.2.3 ACTUALIZACION DEL MONTO.
5.2.4 ACTUALIZACION DE LA FECHA.
5.2.5 ACTUALIZACION DE CUENTA_FUENTE.
5.2.6 ACTUALIZACION DE LA CUENTA_DESTINO.
5.2.7 ACTUALIZACION DEL TIPO DE PAGO O ABONO.
5.2.8 ACTUALIZACION DEL NUMERO DE PAGOS O ABONOS.
5.2.9 ACTUALIZACION DE TAZA DE INTERES.
5.2.10 SELECCION DE OTRO REGISTRO.
5.3 REQUERIMIENTO 2.3:REGISTRO DE CALENDARIO.
5.4 REQUERIMIENTO 2.4:CONSULTA AL CALENDARIO.
5.5 REQUERIMIENTO 2.5:REPORTE DEL CALENDARIO.
5.6 REQUERIMIENTO 2.6:CALCULO DEL FLUJO DE EFECTIVO.
5.7 REQUERIMIENTO 2.7:CONSULTA AL FLUJO DE EFECTIVO.
5.8 REQUERIMIENTO 2.8:REPORTE DE FLUJO DE EFECTIVO.
6.0 DEFINICION DE REQUERIMIENTOS DE LA FUNCION DE MOVIMIENTO DE CUENTAS.
6.1 REQUERIMIENTO 3.1:INICIALIZACION DE CUENTAS.
6.2 REQUERIMIENTO 3.2:ACTUALIZACION DE TRANSACCIONES.
6.2.1 BAJA DE UNA TRANSACCION.
6.2.2 ACTUALIZACION DEL CONCEPTO.
6.2.3 ACTUALIZACION DEL MONTO.

6.2.4 ACTUALIZACION DE LA FECHA.
6.2.5 ACTUALIZACION DE LA CUENTA_FUENTE.
6.2.6 ACTUALIZACION DE LA CUENTA_DESTINO.
6.2.7 SELECCION DE TRANSACCION.
6.3 REQUERIMIENTO 3.3:REGISTRO DE TRANSACCIONES.
6.4 REQUERIMIENTO 3.4:CONSULTA DE MOVIMIENTOS.
6.5 REQUERIMIENTO 3.5:REPORTE DE MOVIMIENTOS.
6.6 REQUERIMIENTO 3.6:CALCULO DE SALDOS.
6.7 REQUERIMIENTO 3.7:RESTAURACION DE CONDICIONES INICIALES.
7.0 DEFINICION DE REQUERIMIENTOS DE LA FUNCION DE APOYO A DECISIONES.
7.1 REQUERIMIENTO 4.1:CALCULO DEL VALOR PRESENTE_PAGO UNICO.
7.2 REQUERIMIENTO 4.2:CALCULO DEL VALOR FUTURO_PAGO UNICO.
7.3 REQUERIMIENTO 4.3:FONDO ACUMULATIVO.
7.4 REQUERIMIENTO 4.4:RECUPERACION DE CAPITAL.
7.5 REQUERIMIENTO 4.5:CALCULO DEL VALOR FUTURO_SERIE CONSTANTE.
7.6 REQUERIMIENTO 4.6:CALCULO DEL VALOR PRESENTE_SERIE CONSTANTE.
8.0 ASPECTOS DE CONTROL DE CALIDAD.
8.1 CRITERIO DE ACEPTACION DEL SISTEMA.
8.2 MATRIZ DE REQUERIMIENTOS.
8.3 BITACORA DE CAMBIOS.

6.2.2.1 Factibilidad. -

Los objetivos No. 1: 'Desarrollar un sistema que apoye la toma de decisiones económicas personales' y los objetivos No. 2: 'Desarrollar un sistema que pueda ser ejecutado en un sistema de cómputo que incluya un procesador central, periféricos de memoria, un impresor, un teclado y una pantalla' llevaron al grupo de Análisis de Sistemas a conceptualizar el sistema esquematizado en la fig. 6.2.2.a, b, c y d.

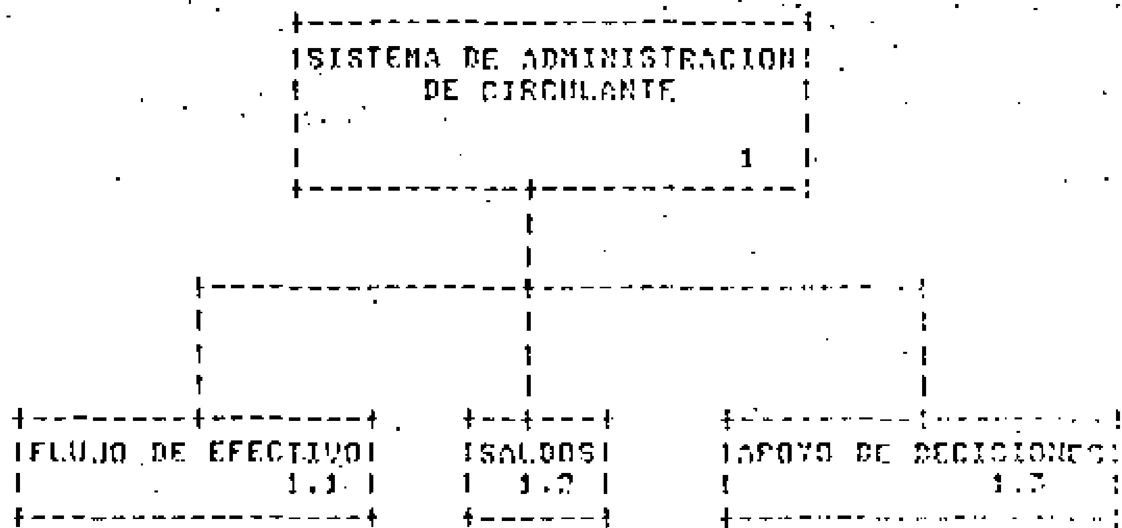


FIGURA 6.2.2.a. ESTRUCTURA DEL MODELO CONCEPTUAL DEL SISTEMA 'ADMINISTRACION DEL CIRCULANTE' PARA PROPOSITOS DE ESPECIFICACION.

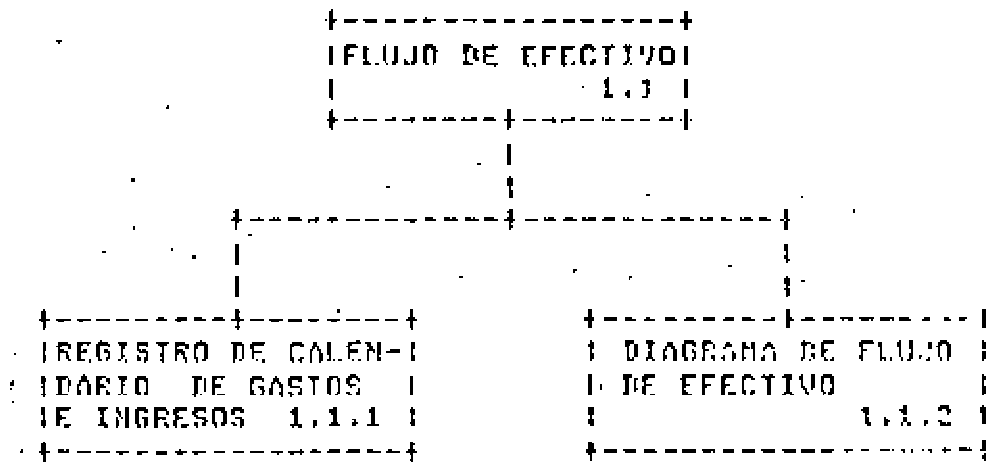


FIGURA 6.2.2.b. ESTRUCTURA DEL MODELO CONCEPTUAL DEL SISTEMA "ADMINISTRACION DEL CIRCULANTE" PARA PROPOSITOS DE ESPECIFICACION.

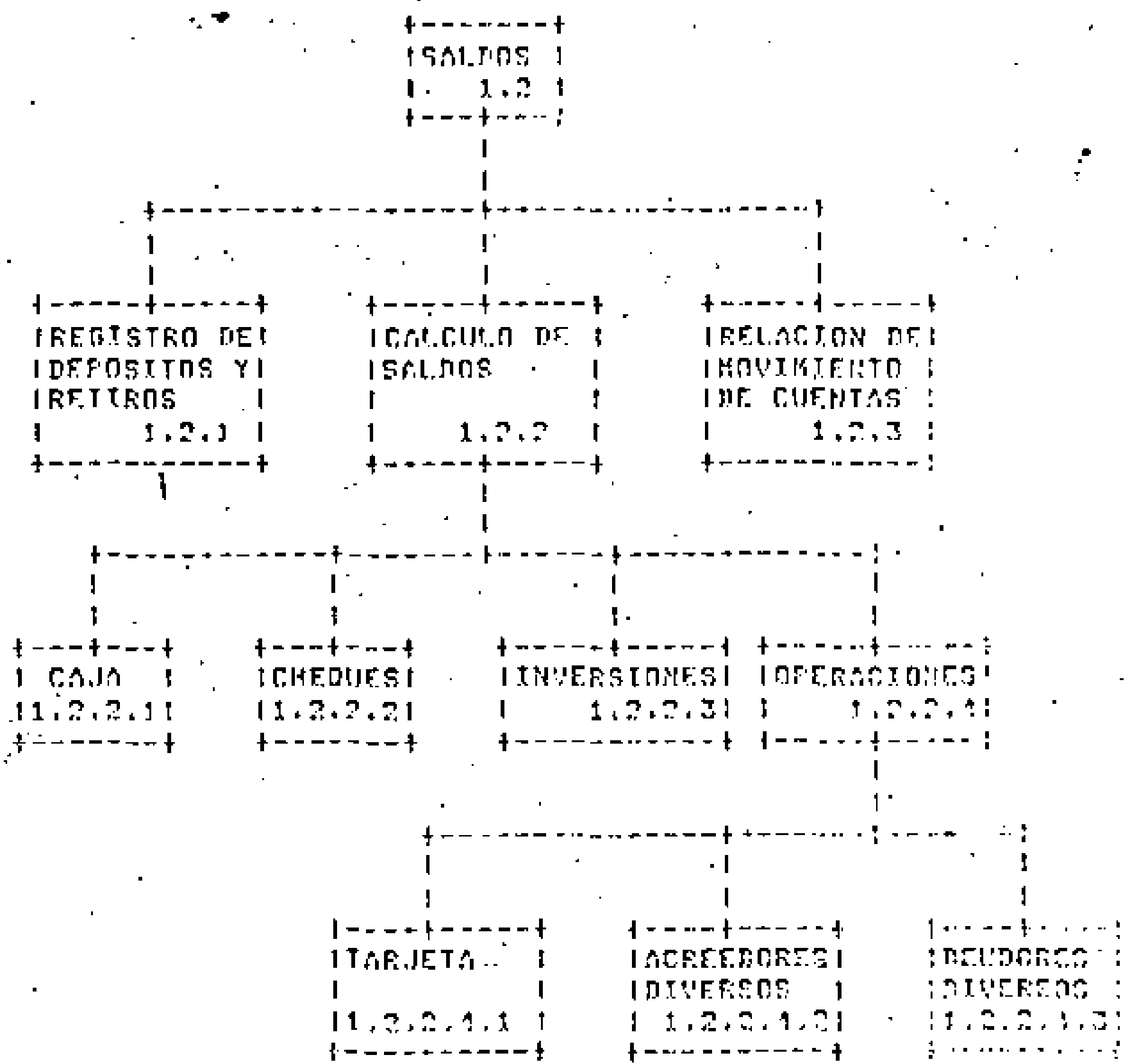


FIGURA 6.2.2.e. ESTRUCTURA DEL MODELO CONCEPTUAL DEL SISTEMA "ADMINISTRACION DEL CIRCULANTE" PARA PROPOSITOS DE ESPECIFICACION.

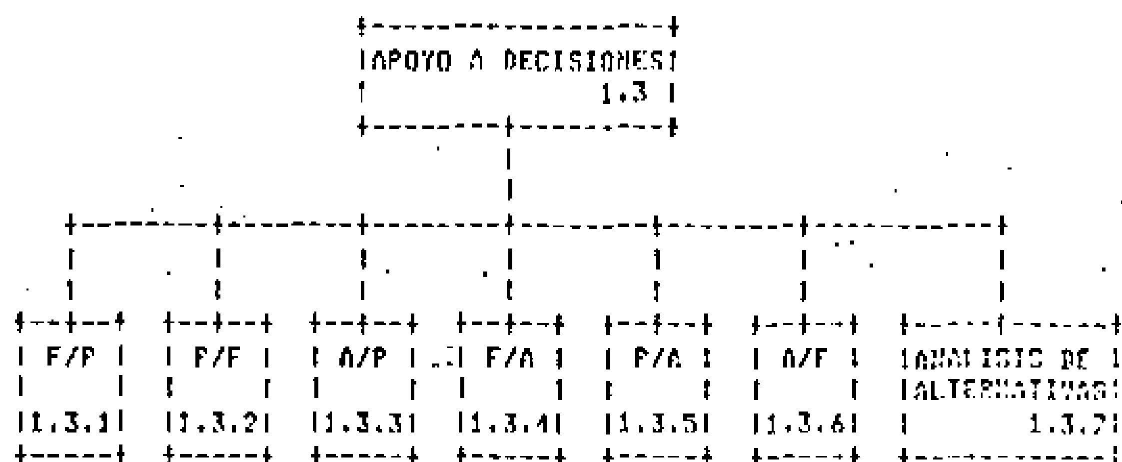


FIGURA 6.2.2.d. ESTRUCTURA DEL MODELO CONCEPTUAL DEL SISTEMA "ADMINISTRACION DEL CIRCULANTE" PARA PROPOSITOS DE ESPECIFICACION.

Como muestra la figura 6.2.2. este sistema tendrá las siguientes funciones que producen registros para el usuario:

1.1.1.1. Registro de calendario de gastos e ingresos.

1.1.2. Diagrama de Flujo de efectivo.

1.2.1. Registro de depósitos y retiros.

1.2.3. Relación de movimiento de cuenta.

Se controlarán tres cuentas:

1.2.2.1. Caja

1.2.2.2. Cheques

4.1 REQUERIMIENTO 1.1: ACTUACION DEL SISTEMA.

El sistema de administración de circulante y tema de decisiones financieras podrá ser activado bajo demanda del usuario a través del teclado.

El sistema será activado mediante el comando:

@ SACYTD

Una vez activado el sistema deberá aparecer en pantalla el desplegado D.1.1.1, presentará la lista de opción que el sistema ejerce al usuario.

4.2 REQUERIMIENTO 1.2 : SELECCION DE OPCIONES.

El usuario podrá seleccionar la opción deseada entre la lista de opciones del desplegado D.1.1.1, tecleando el número de la opción u opciones con la tecla de retorno.

Al seleccionarse una opción válida, el sistema deberá responder en la manera establecida por los requerimientos subsecuentes.

caracteres extraños ó números

Si la selección no es válida (diferentes a los números de las opciones entonces el sistema deberá responder haciendo aparecer el desplegado D.1.1.1 en pantalla.

5.1 REQUERIMIENTO 2.1 : INICIALIZACION DE CALENDARIO

El usuario podrá inicializar el archivo de calendario de gastos e ingresos, seleccionando la opción 1 en el desplegado D.1.1.1.

El sistema deberá borrar todos los registros del calendario que hasta ese momento existan almacenados.

Una vez inicializado el calendario, el sistema deberá mostrar en pantalla el desplegado D.1.1.1.

5.2 • REQUERIMIENTO 2.2 : ACTUALIZACION DEL CALENDARIO.

El usuario podrá dar de baja o modificar, uno a uno, todos los registros almacenados en el calendario de gastos e ingresos (vease glosario de términos). Esta función podrá ser realizada al seleccionar el usuario la opción 2 en el desplegado D.1.1.1. El sistema deberá responder solicitando con el mensaje M.2.2.1 (véase tabla T.2.2.1) el número del registro que se desea modificar.

El usuario podrá teclear un número entero, positivo, menor que 100. Si el número no corresponde a alguno de los registros almacenados, entonces el sistema deberá responder con el mensaje M.2.2.2 y deberá solicitar nuevamente el número de registro con el mensaje M.2.2.1.

Cuando el usuario teclee algún carácter o caracteres que no sean números enteros, el sistema deberá responder con el desplegado D.1.1.1.

Si el usuario teclea un número de registro almacenado en el calendario de gastos e ingresos, entonces deberá aparecer el desplegado D.2.2.1.

En el deplegado deberán aparecer los valores almacenados previamente por el usuario.

El usuario podrá dar de baja el registro o actualizar, uno a uno, cualesquiera de sus campos, tecleando la opción deseada.

Si el usuario nó teclea una opción válida, el sistema deberá responder con el mensaje M.2.2.3

5.2.1 BAJA DE UN REGISTRO

El usuario podrá dar de baja el registro que aparece en pantalla, seleccionando la opción 1. El sistema deberá borrar los campos del registro y renumerar los registros subsecuentes. Al terminar este proceso, el sistema deberá desplegar el desplegado D.2.2.1 con los campos correspondeintes al registro que al haberse renumerado posea el número de registro previamente establecido por el usuario.

Cuando cause baja el último registro del calendario, el sistema deberá desplegar el mensaje M.2.2.3 y en secuencia el mensaje M.2.2.1

5.2.2 ACTUALIZACION DE CONCEPTO.

El usuario podrá modificar el concepto tecleando la opción 2. El sistema deberá responder con el mensaje M.2.2.4.

El usuario podrá teclear hasta 20 caracteres, cuando el usuario teclee más de 20 caracteres, el sistema deberá truncar la serie y considerar únicamente los primeros 20.

El sistema deberá actualizar el registro y desplegar D.2.2.1 en pantalla con el contenido del registro actualizado.

5.2.3.- ACTUALIZACION DEL MONTO

El usuario podrá modificar el monto, tecleando la opción 3. El sistema deberá responder con el mensaje M.2.2.5.

El usuario podrá teclear un número real, positivo, menor o igual que: 999 999 999.99.

El sistema truncará toda fracción decimal a dos dígitos significativos a la derecha del punto.

Si el usuario no teclea en formación válida, el sistema deberá responder con el mensaje M.2.2.6.

Si el usuario no teclea el punto decimal, el sistema deberá suponer que la cantidad es exacta y que los dígitos a la derecha del punto son cero.

El sistema deberá actualizar el registro y a continuación presentar el desplegado D.2.2.1 en pantalla con el registro actualizado.

5.2.4 ACTUALIZACION DE FECHA

El usuario podrá actualizar la fecha, tecleando la opción 4. El sistema deberá responder con el mensaje M.2.2.7.

El usuario podrá teclear un número entero de hasta dos digitales, comprendido entre 1 y 31. Si el dato no es válido, el sistema deberá responder con los -- mensajes M.2.2.8 y M.2.2.7 en secuencia.

Si el dato es válido, el sistema deberá preguntar el mes con el mensaje M.2.2.9 El usuario podrá teclear un número entero de hasta dos dígitos, comprendido entre 1 y 12. Si el usuario no teclea un dato válido, el sistema deberá responder con los mensajes M.2.2.10 y M.2.2.9 en secuencia.

Si el dato es válido, el sistema deberá preguntar el año con el mensaje M.2.2.11

El usuario podrá teclear un número entero de dos dígitos comprendido entre 00 y 99. Si el usuario teclea un dato inválido, el sistema deberá responder con los mensajes M.2.2.12 y M.2.2.11 en secuencia.

Si el dato es válido, el sistema deberá actualizar el registro y presentar el desplegado D.2.2.1 con la información actualizada.

5.2.5 ACTUALIZACION DE LA CUENTA-FUENTE.

El usuario podrá modificar la cuenta-fuente tecleando la opción 5. El sistema deberá responder con el mensaje M.2.2.13. El usuario podrá teclear una clave de 3 caracteres entre las claves válidas: TAR, CAJ, INV, DEV, ACR, CHE, EXT.

Si el usuario no teclea una clave válida, el sistema deberá responder con los mensajes M.2.2.14 y M.2.2.13 en secuencia.

Si la clave es válida, el sistema deberá actualizar el registro y desplegar D.2.2.1.

5.2.6 ACTUALIZACION CUENTA-DESTINO

El usuario podrá modificar la cuenta-destino tecleando la opción 6. El sistema deberá responder con el mensaje M.2.2.15. El usuario podrá teclear una clave de 3 caracteres entre las claves válidas: TAR, CAJ, INV, DEV, ACR, CHE, EXT.

Si el usuario no teclea una clave válida, el sistema deberá responder con los mensajes M.2.2.14 y M.2.2.15 en secuencia.

Si la clave es válida, el sistema deberá actualizar el registro y desplegar D.2.2.1

5.2.7 ACTUALIZACION DE TIPO DE PAGO O ABONO.

El usuario podrá modificar el tipo de pago o abono, tecleando la opción 7. El sistema deberá desplegar el mensaje M.2.2.16.

El usuario podrá teclear M, S ó A. Si el usuario no teclea M, S ó A el sistema deberá desplegar M.2.2.16.

Si la clave es válida, el sistema deberá actualizar el registro y desplegar D.2.2.1

5.2.8 ACTUALIZACION DEL NUMERO DE PAGOS O ABONOS.

El usuario podrá modificar el número de períodos tecleando la opción 8. El sistema deberá desplegar el mensaje M.2.2.17.

El usuario podrá teclear un número entero entre 0 y 100. Si el usuario no te-- clea un dato válido, el sistema deberá responder con el desplegado M.2.2.17.

Si el dato es válido, el sistema deberá actualizar el registro y desplegar D.2.2.1

5.2.9 ACTUALIZACION DE LA TAZA DE INTERES.

El usuario podrá modificar la taza de interés tecleando la opción 9. El sistema deberá desplegar el mensaje M.2.2.18. El usuario podrá teclear un número - real comprendido entre 0.0 y 99.9. Si el usuario no teclea un dato válido, el sistema deberá desplegar M.2.2.19 y M.2.2.18 sucesivamente.

Si el dato es válido, el sistema deberá actualizar el registro y desplegar D.2.2.1

5.2.10 SELECCION DE OTRO REGISTRO

El usuario podrá dejar intacto un registro y seleccionar otro para realizar cambios, tecleando la opción 0. El sistema deberá dejar intacto el registro y preguntar el número de otro registro mediante el mensaje M.2.2.1.

El usuario podrá terminar su sesión de actualización de calendario, tecleando un carácter no numérico. El sistema deberá desplegar D.1.1.1.

SISTEMA DE ADMINISTRACION DE CIRCULANTE Y TOMA DE DECISIONES FINANCIERAS.

SIMULACION DE FLUJO DE EFECTIVO:

1). INICIALIZACION DE CALENDARIO
2). ACTUALIZACION DE CALENDARIO DE GASTOS E INGRESOS
3). REGISTRO DE CALENDARIO DE GASTOS E INGRESOS
4). CONSULTA DE CALENDARIO POR PANTALLA
5). REPORTE DE CALENDARIO
6). CALCULO DE FLUJO DE EFECTIVO
7). CONSULTA DE FLUJO DE EFECTIVO POR PANTALLA
8). REPORTE DE FLUJO DE EFECTIVO.

MOVIMIENTO DE CUENTAS:

9). INICIALIZACION DE CUENTAS
10). ACTUALIZACION DE MOVIMIENTOS
11). REGISTRO DE MOVIMIENTOS
12). CONSULTA DE MOVIMIENTOS POR PANTALLA
13). REPORTE DE MOVIMIENTOS Y SALDOS
14). CALCULO DE SALDOS
15). RESTAURACION DE CONDICIONES INICIALES

APOYO A DECISIONES:

16). FORMULAS DE INTERES COMPUESTO
17). TERMINACION DE SESION

TECLEE EL NUMERO DE LA OPCION DESEADA Y OPRIMA TECLA DE RETORNO: [X]

DESPLEGADO D.1.1.1 OPCIONES DEL SISTEMA.

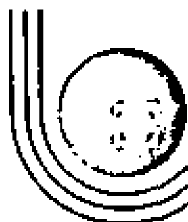
CLAVE DE MENSAJE	M E N S A J E
M.2.2.1	*** NUMERO DEL REGISTRO = [X]
M.2.2.2	*** EXISTEN X X X REGISTROS ALMACENADOS
M.2.2.3	*** OPCION INVALIDA. ESCOJA ENTRE 0 Y 9
	*** OPCION = [X]
M.2.2.4	*** CONCEPTO = [X]
M.2.2.5	*** MONTO = [X]
M.2.2.6	*** DATO NUMERICO INVALIDO
	*** MONTO = [X]
M.2.2.7.	*** DIA = [X]
M.2.2.8	*** FECHA INVALIDA. ESCOJA ENTRE 1 Y 31
M.2.2.9	*** MES = [X]
M.2.2.10	*** FECHA INVALIDA, ESCOJA ENTRE 1 Y 12
M.2.2.11	*** AÑO = [X]
M.2.2.12	*** FECHA INVALIDA, ESCOJA ENTRE 00 Y 99
M.2.2.13	*** CUENTA-FUENTE = [X]
M.2.2.14	*** CLAVE INVALIDA. ESCOJA ENTRE: TAR,
	*** CAJ, INV, DEV, ACR, CHE, EXT.
M.2.2.15	*** CUENTA-DESTINO = [X]
M.2.2.16	*** TIPO (M,S O A) = [X]
M.2.2.17	*** NUMERO (ENTRE 0 Y 100) = [X]
M.2.2.18	*** TAZA (%) = [X]
M.2.2.19	*** TAZA INVALIDA. ESCOJA ENTRE 0.0 Y 99.9

TABLA T.2.2.1 MENSAJES DEL REQUERIMIENTO 2.2

```
                                                                    TIPO
                                              FECHA                  DE
                                                DE                  PAGO   NUM
                                 MONTO     OPERACION CUENTA CUENTA    O    DE   TAZA
 REC       CONCEPTO                $        DD MM AA  FUENTE DESTINO ABONO PAGOS  %

 XXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXX XX XX XX    XXX     XXX       X     XXX   XXXX
***
***
*** SELECCIONE LA OPCION DESEADA:
***
***              1 DAR DE BAJA EL REGISTRO.
***              2 MODIFICAR CONCEPTO.
***              3 MODIFICAR MONTO ($).
***              4 MODIFICAR FECHA.
***              5 MODIFICAR CUENTA_FUENTE (EXT,TAR,CAJ,INV,DEU,ACR,CHE).
***              6 MODIFICAR CUENTA_DESTINO (EXT,TAR,CAJ,INV,DEU,ACR,CHE).
***              7 MODIFICAR TIPO DE PAGO (M,S,A).
***              8 MODIFICAR NUMERO DE PAGOS O ABONOS.
***              9 MODIFICAR TAZA DE INTERES (%).
***              0 NO ALTERAR EL REGISTRO.
***
*** OPCION=[X]
```

DESPLEGADO D.2.2.1 ACTUALIZACION DE REGISTRO.

REQUERIMIENTO	DESCRIPCION	RPC	ARO	DPC	FLP	PRP	BSP	MUS
1.1	ACT. DEL SISTEMA	4.1						
1.2	SEL. DE OPCIONES	4.2						
1.3	TERM. DE SESION	4.3						
2.1	INIC. DE CALENDARIO	5.1						
2.2	ACT. DE CALENDARIO	5.2						
2.3	REC. DE CALENDARIO	5.3						
2.4	CONS. DE CALENDARIO	5.4						
2.5	REP.DE CALENDARIO	5.5						
2.6	CAL. DE FLUJO	5.6						
2.7	CONS. DE FLUJO	5.7						
2.8	REP. DE FLUJO	5.8						
3.1	INIC. DE CUENTAS	6.1						
3.2	ACT. DE TRANS.	6.2						
3.3	REC. DE TRANS.	6.3						
3.4	CONS. DE MOV.	6.4						
3.5	REP. DE MOV.	6.5						
3.6	CALCULO DE SALDOS	6.6						
3.7	REST. DE COND. INIC.	6.7						
4.1	VALOR.F_PAGO UNICO	7.1						
4.2	VALOR F_PAGO UNICO	7.2						
4.3	FONDO ACUMULATIVO	7.3						
4.4	REC. DE CAPITAL	7.4						
4.5	VALOR F_SERIE CONST.	7.5						
4.6	VALOR F_SERIE CONST.	7.6						



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

TECNICAS MODERNAS DE DESARROLLO Y
ADMINISTRACION DE PROGRAMACION

FASE DE DESARROLLO - ETAPAS

- CODIFICACION
- INTEGRACION
- PRUEBAS DE ALTO NIVEL

MARZO , 1983

acio de Mineria Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

FASE DE DESARROLLO

ETAPAS

** C O D I F I C A C I O N

** I N T E G R A C I O N

** PRUEBAS DE ALTO NIVEL

CODIFICACION

ES LA TRADUCCIÓN LAS ESPECIFICACIONES DE -- PROCESO DE CADA MODULO, EN INSTRUCCIONES EJECUTABLES POR UN LENGUAJE DE PROGRAMACION.

INTEGRACION

RELACION FUNCIONAL DE LOS MODULOS DE UN PROGRA MA

PRUEBAS

EJECUCIÓN DE UN PROGRAMA CON LA INTENCIÓN DE DETECTAR ERRORES.

FASE DE DESARROLLO

OBJETIVOS

- CODIFICACION DE LOS MODULOS
- VERIFICACION DEL FUNCIONAMIENTO DE CADA MODULO
- INTEGRACION DE LOS MODULOS PARA FORMAR PROGRAMAS
- VERIFICACION DEL FUNCIONAMIENTO DE CADA PROGRAMA
- INTEGRACION DE LOS PROGRAMAS PARA FORMAR SISTEMAS

ENFOQUES DE ESTRATEGIA

* CONSERVADOR

  DISEÑAR TODO EL SISTEMA ANTES DE CODIFICAR

* RADICAL

  DISEÑAR UNO DE LOS PROGRAMAS DEL SISTEMA Y CODIFICAR Y PROBAR INMEDIATAMENTE.

FACTORES A CONSIDERAR

* CONOCIMIENTOS DEL USUARIO

  RADICAL - CONOCIMIENTO TECNICO ACEPTABLE

  CONSERVADOR - FALTA DE CONOCIMIENTO

* PRESIONES DE TIEMPO

  RADICAL - GRANDES PRESIONES DE TIEMPO

* CALENDARIOS PRECISOS

  CONSERVADOR - CALENDARIOS PRECISOS

ESTRATEGIAS DE CODIFICACION

** CODIFICACION DESCENDENTE
(TOP-DOWN DEVELOPMENT)

FORZAR LA DEFINICION PRECISAS DE LAS INTERFASES PRINCIPALES Y FORZAR QUE ESTAS INTERFASES SE - CODIFIQUEN Y SE PONGAN A TRABAJAR EN LA COMPUTADORA PARA ASEGURARSE QUE TRABAJEN.

** CODIFICACION ASCENDENTE
(BOTTOM-UP DEVELOPMENT)

ATACAR LOS MODULOS QUE TONIENEN OPERACIONES PRIMITIVAS PRIMERO PARA DE AHI CONSTRUIR LAS OPERACIONES MAS COMPLEJAS.

ESTRATEGIAS DE INTEGRACION

** INCREMENTAL

VALIDACION DE NUEVOS MODULSO (NO PROBADOS) AGREGANDOLOS A LOS MODULOS YA PROBADOS E INTEGRADOS.

** NO INCREMENTAL

VALIDACION DE PROGRAMAS A PARTIR DE PRUEBAS MODULARES INDEPENDIENTES.

LOS ERRORES AUMENTAN EL COSTO DE PROGRAMACION POR LO TANTO, HAY QUE DETECTARLOS OPORTUNAMENTE.

CUALQUIER ESTRATEGIA DE INTEGRACION VA A DETECTAR — ERRORES. PERO SERAN TODOS LOS ERRORES QUE SE VAN A PRESENTAR

?

TIPOS DE ERRORES

** CODIFICACION

- ERRORES EN LOS RESULTADOS

** ANALISIS

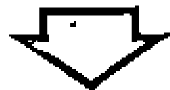
INSATISFACCION DE NECESIDADES DEL USUARIO

** ESPECIFICACION

INTERPRETACION EQUIVOCADA DE NECESIDADES

** DISEÑO

TIEMPO DE RESPUESTA EXCESIVO



PRUEBAS DE ALTO NIVEL

TIPOS DE PRUEBAS

** PRUEBAS FUNCIONALES

- ERRORES DE ANALISIS Y ESPECIFICACION
- TECNICAS DE ANALISIS ENTRADA/SALIDA

** PRUEBAS DE IMPLANTACION

- ERRORES EN EL EQUIPO DE COMPUTO
- INTERFASES, EQUIPO, SISTEMA OPERATIVO

** PRUEBAS DE SISTEMA

- MANUAL DE USUARIO

** PRUEBAS DE ACEPTACION

- COMPARAR PRODUCTO FINAL CON EL CONTRATO ORIGINAL

RESUMEN

1) LA ETAPA DE DESARROLLO NO DEBERA INICIARSE ANTES DE COMPLETAR LA REVISION DEL DFC. ESTO, SIN EMBARGO, NO SIGNIFICA QUE SE NECESITE TENER TODO EL SISTEMA DISEÑADO Y REVISADO.

2) LA EFICIENCIA DE UN SISTEMA ESTA INFLUIDA POR LA CALIDAD DEL PROGRAMADOR. SELECCIONE UN BUEN PROGRAMADOR Y EL SISTEMA PUEDE MEJORAR POR UN FACTOR DE 10.

3) VARIOS ESTUDIOS HAN DETERMINADO QUE LA REGLA 90-10 SE APLICA TAMBIEN A LOS PROGRAMADORES. ESTO ES 10% DEL CODIGO DE UN PROGRAMA CONSUME EL 90% DEL TIEMPO DE CPU (KNUTH, 1971). EN OTRAS PALABRAS SOLO UNA PEQUEÑA CANTIDAD DE CODIGO ES SIGNIFICATIVA, ASI QUE LA ESTRATEGIA DEBE SER CONSEGUIR QUE TRABAJE EL SISTEMA; LUEGO ENCUENTRE QUE PARTES SON INEFICIENTES Y LUEGO OPTIMICELAS.

4) ES MAS FACIL HACER QUE UN SISTEMA CORRECTO SEA EFICIENTE QUE LOGRAR QUE UN SISTEMA INCORRECTO SEA EFICIENTE. PRIMERO CONSIGA QUE EL SISTEMA TRABAJE; DESPUES PREOCUPESE POR LA EFICIENCIA.

GC20-1850-1

Improved Programming Technologies — An Overview

This document is intended to briefly describe to the reader eight techniques designed to improve the program development process: structured programming, top-down program development, chief programmer teams, development support libraries, HIPO (Hierarchy plus Input-Process-Output), structured design, structured walk-throughs, and application development inspections. These techniques are still evolving; initial use in a data processing activity should be subject to management review, to determine the form in which the techniques may best fit into each environment.

Second Edition (October 1978)

This edition is a minor revision of and does not obsolete the previous edition. Material on structured design and application development inspections and a bibliography have been added.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

Address comments concerning the contents of this publication to IBM Corporation, Technical Publications, Dept. 824, 1133 Westchester Avenue, White Plains, New York 10604.

© Copyright International Business Machines Corporation 1974, 1978

4

Contents

Introduction 1

Chapter 1: Structured Programming 2
Structured Programming Theory 3
Structured Programming Practices 3

Chapter 2: Top-Down Program Development 6

Chapter 3: Chief Programmer Teams 9
The Team Members 9
Why Change to Teams? 9
Chief Programmer Teams in Large Projects 10

Chapter 4: Development Support Libraries 11
Basic Elements and Method of Use 11
Additional Library Facilities 12

Chapter 5: Hierarchy plus Input-Process-Output (HIPO) 13

Chapter 6: Structured Design 18

Chapter 7: Structured Walk-Throughs and Inspections 19
The Need for Technical Reviews 19
Structured Walk-Throughs 20
Inspections 20
Key Differences Between Inspections and Structured Walk-Throughs 21

Appendix, Reading List, Supporting Materials, and Related Programs 23

Introduction

The last decade has been characterized by significant improvements in hardware speed and capacity, configuration flexibility, and programming system capability. There have also been many improvements in the capabilities of programming languages, but, in general, improvements in the techniques used in the program development process have lagged behind those in other areas. This period has also been characterized by the increasing complexity of application systems and by their importance to the organization. And, in the same period, application development, maintenance, and modification activities have constituted an increasing portion of the data processing budget. Data processing management, therefore, has been searching for ways to improve the program development process, with the objective of producing application systems that meet the needs of their users, are more error-free, require less maintenance, are easier to modify, and are developed on schedule with improved productivity.

This document describes eight evolving techniques that have been implemented in various ways in some program development efforts within IBM and that may be of assistance in achieving management objectives. These techniques, some of which have elements that have been advocated or used in the past, are structured programming, top-down program development, chief programmer teams, development support libraries, HIPO (Hierarchy plus Input-Process-Output), structured design, structured walk-throughs, and application development inspections. The first four have frequently been used together in IBM's Federal Systems Division. The others were developed separately and seem to logically complement structured programming, top-down programming, chief programmer teams, and development support libraries.

These techniques can be used individually or together. Since they are still evolving, their initial use in a data processing activity should be subject to management review, to determine the form in which they may best fit into its environment.

Chapter 1: Structured Programming

Traditionally, each programmer has applied a personal set of rules to the construction of the logic of a program. The programmer starts with this logic structure and, additional combinations of conditions to be met are encountered, they are added as afterthoughts rather than revisions of the logic of the program. The resultant control code might look like that shown in the left (Unstructured) column of Figure 1. This code contains a large number of GO TO statements and labels and its logic is not easy to follow. During subsequent unit and integration testing, disintegration of the programmer's original structure occurs as new constraints and conditions are imposed upon it - leading to more GO TO statements, more labels, and a final program whose original logic may be completely obscured. Reading, understanding, and testing such programs is difficult. The degree of confidence in their quality or correctness tends to be low. In addition, such programs tend to be difficult to maintain and modify.

UNSTRUCTURED	STRUCTURED
IF p GOTO label q	①IF p THEN
IF w GOTO label m	A function
L function	B function
GOTO label k	②IF q THEN
label m M function	③IF t THEN
GOTO label k	G function
label q IF q GOTO label t	④DOWHILE u
A function	H function
B function	④ENDDO
C function	I function
label r IF NOT r GOTO label s	③(ELSE)
D function	③ENDIF
GOTO label r	②ELSE
label s IF s GOTO label f	C function
E function	③DOWHILE r
label v IF NOT v GOTO label k	D function
J function	③ENDDO
label k K function	③IF s THEN
END function	F function
lable f F function	③ELSE
GOTO label v	E function
label t IF t GOTO label a	③ENDIF
A function	②ENDIF
B function	②IF v THEN
GOTO label w	J function
label a A function	②(ELSE)
B function	②ENDIF
G function	①ELSE
label u IF NOT u GOTO label w	②IF w THEN
H function	M function
GOTO label u	②ELSE
label w IF NOT t GOTO label y	L function
I function	②ENDIF
label y IF NOT v GOTO label k	①ENDIF
J function	K function
GOTO label k	END function

Figure 1. A comparison of structured and unstructured code

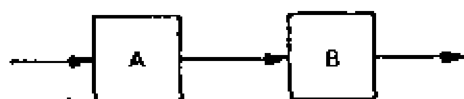
Research by computer scientists and mathematicians indicates that an alternative method of programming known as structured programming can help solve these problems. This technique involves coding programs using a limited number of control logic structures to form highly structured units of code that are more readable, and therefore more easily tested, maintained, and modified.

Structured Programming Theory

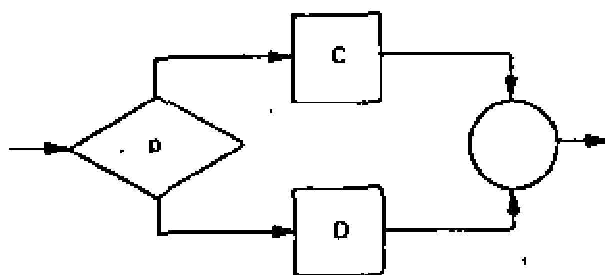
Structured programming is based on a mathematically proven structure theorem[1] that states that any program can be written using only the three control logic structures illustrated in Figure 2:

- Sequence of two or more operations (MOVE, ADD...)
- Conditional branch to one of two operations and return (the IF p THEN C ELSE D of Figure 2)
- Repetition of an operation while a condition is true (the DO E WHILE q of Figure 2)

Sequence of two operations



IFTHENELSE: Conditional branch to one of two operations and return



DOWHILE: Operation repeated while a condition is true

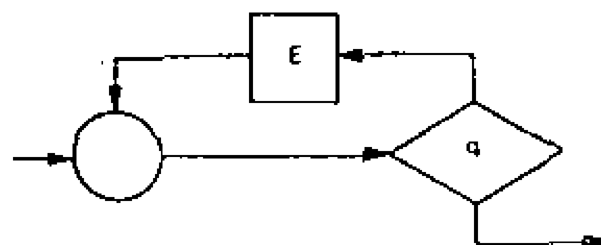


Figure 2. The three elemental logic structures of structured programming

Any program may be developed by the appropriate iteration and nesting of these three basic structures. Each of the three structures has only one entry and one exit. A program consisting solely of these structures is a proper program, a program with one entry and one exit. As illustrated in the structured code (right column) of Figure 1, it always proceeds from the beginning to the end without arbitrary branching. In PL/I, for instance, no GO TO statements are necessary. Proving the logical correctness of structured code is more feasible. The logic is easier to follow, permitting functions to be isolated, understood, and tested.

The use of the three control logic structures in structured programming is analogous to the hardware design practice of forming complex logic circuits from AND, OR, and NOT gates. This practice is based on a theorem in Boolean algebra that states that arbitrarily complex logic functions can be expressed in terms of basic AND, OR, and NOT operations. The use of three control logic structures in structured programming is similarly based on a solid theoretical foundation.

Extensions to the three basic logic structures are permitted as long as they retain the one-entry, one-exit property. An example of such an extension is the DOUNTIL structure (Figure 3), which provides for the execution of the function F until a condition is true.

DOUNTIL: Operation repeated until a condition is true

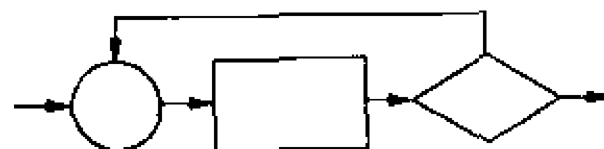


Figure 3. The DOUNTIL structure

Structured Programming Practices

Certain practices are followed to support the objective of producing readable, understandable structured programs – programs in which the writers can have a high degree of confidence.

Indenting within control structure blocks to reflect the logic of the program unit is one of these practices, as shown in the example of structured code in Figure 1. As illustrated by the number to the left of the statements, each logic structure nested within another is indented within it. All parts of a logic structure carry the same indentation level, and functions performed within a logic structure are indented within that structure. This practice highlights the

logic of the unit for the writer and the reader and thus contributes to the goal of more readable programs.

Limiting a unit of source code to a specified size - often one listed page, or fifty lines - permits the programmer to read and understand an entire logical expression or function without referring to multiple pages or relying on memory. Should the complete logical expression require more than 50 lines of source code, the programmer can segment the code through the use of such statements as %INCLUDE (in PL/I) or COPY (in COBOL) to specify the inclusion of another unit of code (see Figure 4).

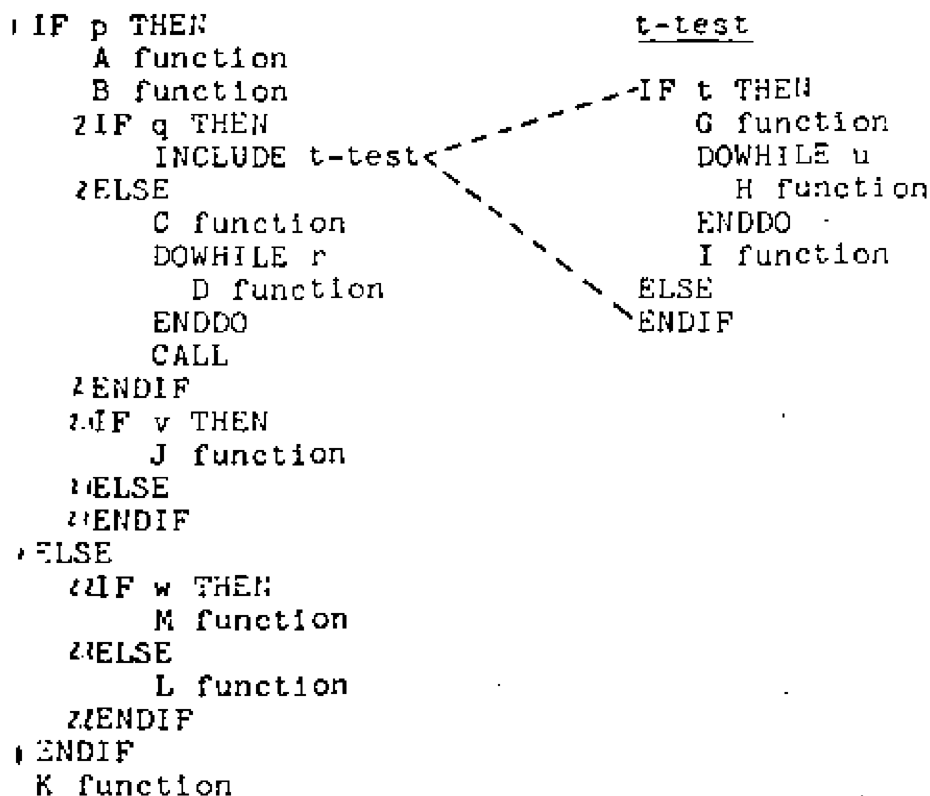


Figure 4. Segmentation

A graphic example of a program constructed of such units is shown in Figure 5. At the top are the job control and linkage editor statements that define the environment and major functions of the program. Subordinate to them is the hierarchy, or calling sequence, of the supporting units of code. Each unit specifies the invocation of the units immediately subordinate to it. Thus unit A would invoke units B and J, using COPY, CALL, PERFORM or %INCLUDE statements; unit B would invoke C and F; unit C would invoke D and E; etc. The next technique to be described, top-down program development, assumes such a hierarchical structure.

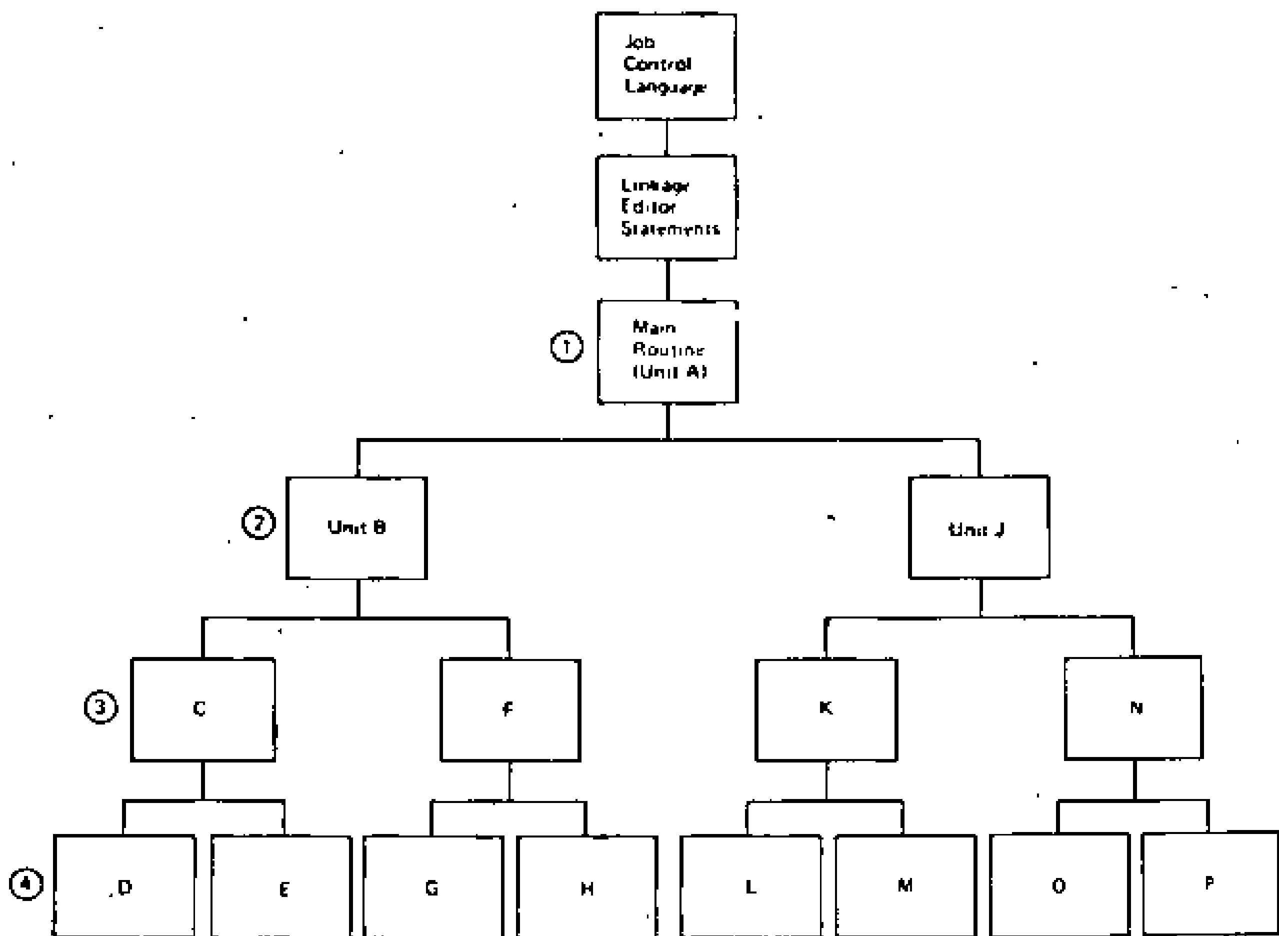


Figure 5. An hierarchical program structure

■ Bohm, C., and Jacopini, G., Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules *Communications of the ACM* 9, No. 3 (May 1966), 366-371.

Chapter 2: Top-Down Program Development

Traditional software development has often been approached as a bottom-up procedure where the lowest level units are coded first, unit-tested, and made ready for integration (see Figure 6). Data definitions and interfaces between units tend to be simultaneously defined by each of the programmers, including those working on the lowest levels of code, and are often inconsistent. During integration, definitions and interface problems are recognized. Integration is delayed while the data definitions and interfaces are correctly defined and the units are reworked and unit-tested to accommodate the changes. It is often difficult to isolate a problem during the traditional integration cycle because of the difficulty in identifying which of the many units combined during integration is the source of the problem. The resultant program, because of last minute redesign, coding, and testing, is often lacking in quality. Superfluous code in the form of driver programs is needed to perform the unit testing and lower levels of integration testing. Management control is often ineffective during much of the traditional development cycle because there may be no coherent, visible product until final integration.

Top-down program development is designed to reduce these problems by reordering the sequence in which units of code are written. A program unit is coded only after the unit that invokes it has been coded and tested. Therefore, top-down program development both assumes and is patterned after a program structure of hierarchical form as illustrated in Figure 5.

Figure 7 illustrates how the top-down approach is begun. Following program design, the job control language (JCL), link-edit statements (LEL), and mainline routine (first level unit) are written (Figure 7A).

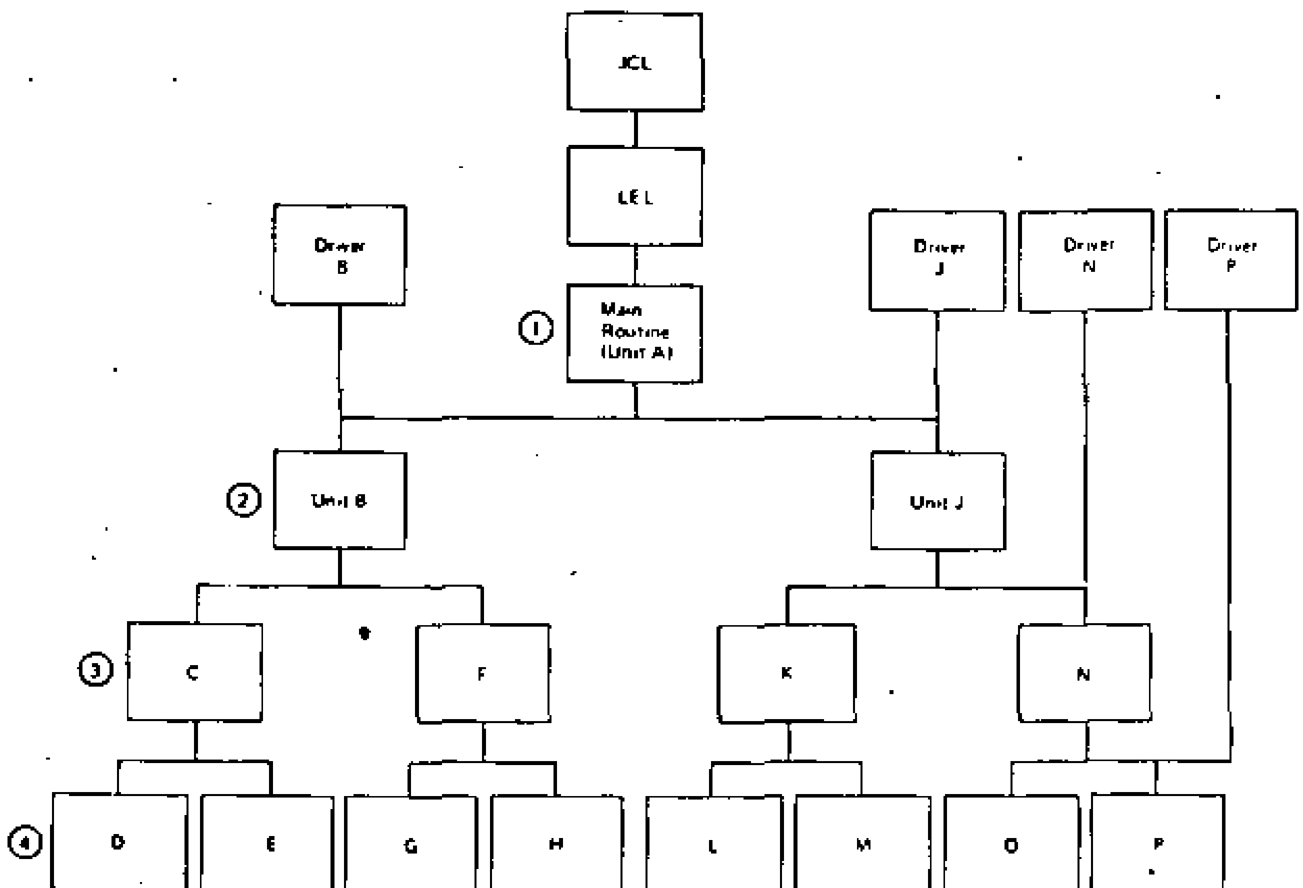


Figure 6 Traditional bottom-up development

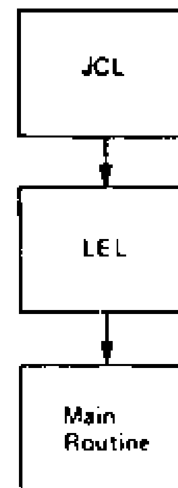
Top-down programming then proceeds by writing the second level units (Figure 7B). While this is taking place, the logic of the first level unit can be tested by substituting dummy units (program stubs) for the second level modules. The program stubs do not normally perform any meaningful computations, but often produce a message to indicate to the tester that they have in fact been executed, thus testing the logic of the next higher level unit. The lower level units are built and integrated in the same manner, substituting actual program units for the program stubs until the entire program has been integrated and tested. The program is continually being integrated – with the higher level units, often the most critical, being the most frequently tested.

Using this method to implement a program design reduces the problem of hypothetical interfaces. Each interface is defined in code. In programming terms, this means not only that units of code are written in calling sequence, but that data base definition statements are written and data records generated before the code requiring those records is written.

Top-down program development permits test data to be generated in an incremental manner. For instance, when the mainline routine is tested, only the test data needed to test the system up to that point need be in readiness. As each subsidiary unit is tested and integrated, the test data needed to test those functions can be added.

The single starting point of top-down program development does not imply that the implementation must proceed down the hierarchy in parallel. Some branches intentionally will be developed earlier than others. For example, branches directly affecting user operations might be developed early in the cycle to permit early user training.

A



B

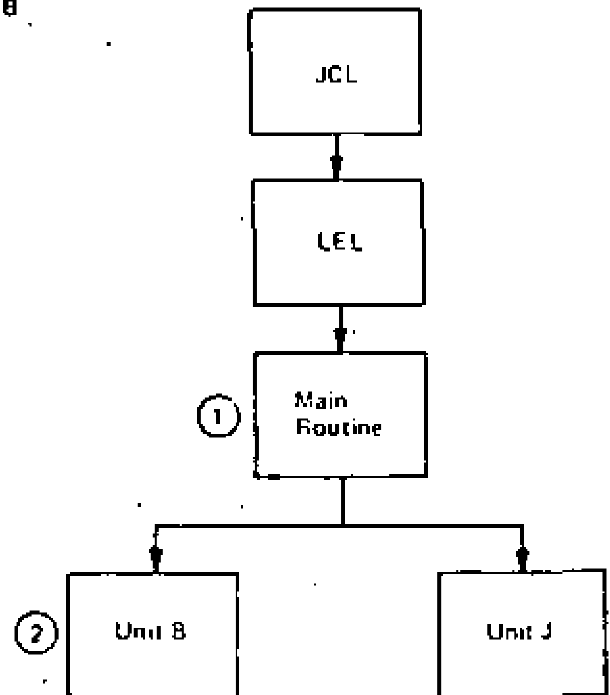
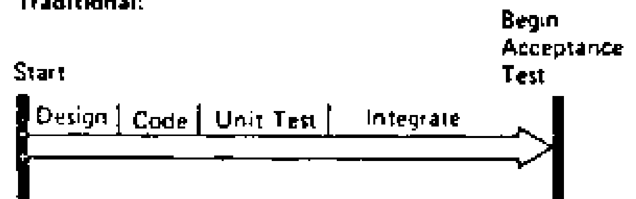


Figure 7. Beginning top-down program development

With top-down program development, the parts of a program and system are continually being integrated. A separate integration period does not exist (see Figure 8). Although it does appear to be theoretically possible that a project whose hierarchy diagram is narrow and long could experience an extended development cycle, it is expected that the cycle for other systems developed in a top-down manner should be no longer than for those developed in the traditional manner. In fact, projects using top-down development, structured programming, chief programmer teams, and development support libraries have described distinct productivity improvements along with improved program quality.



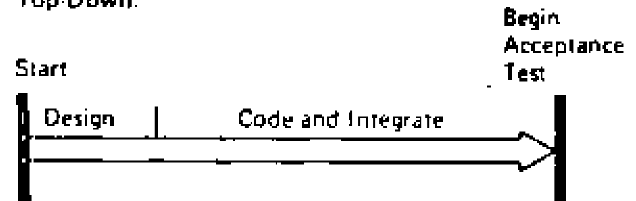


Figure 8 Effect on the development cycle of the traditional and top-down approaches

Chapter 3: Chief Programmer Teams

The increasing complexity of applications and major advances in hardware and software demand many advanced skills during the program development process. With increasing frequency, development managers find that applications and programs cannot be properly developed without a team effort. The chief programmer team is an organizational technique that complements the structured programming and top-down programming techniques, and is designed to coordinate the efforts of programming specialists while retaining the responsiveness and integrity of design expected of a skilled individual.

A chief programmer team is a small group of personnel, under the leadership of a senior level professional programmer called the chief programmer. It normally consists of three to five programmers, a librarian, and other specialists as appropriate. A chief programmer team represents an opportunity to improve both the manageability and the productivity of programming by moving the program development process from private art to public practice through an organizational technique that includes: restructuring the work of program development into specialized jobs that recognize the need for technical expertise in the leadership of the team effort and in the training and career development of its personnel; defining relationships among specialists; and using disciplines to help team members communicate effectively with one another and work effectively with a developing, always visible, project.

The Team Members

Chief Programmer

The chief programmer is responsible for program design of the system, is vested with complete technical responsibility for the project, writes the mainline routines, the critical code, and the operating system interfaces (job control language and linkage editor statements), defines modules to be coded by other team members, and is responsible for specifying the interfaces between modules and for the data definitions. The chief programmer reviews code written by other team members and oversees the testing and integration of all code, and also informs management of project status and arranges for additional team members, when necessary.

Since the chief programmer is the principal designer of the program, requisite duties begin early in the development cycle - while the program functional specifications are being formalized.

Backup Programmer

The backup programmer is a senior level programmer who works closely enough with the chief programmer on the tasks described above to be able to assume the chief's duties if necessary. The backup programmer may be called upon to explore alternative design approaches, perform test and integration planning, or execute other special tasks; and is an active participant in technical design, internal supervision, and external management functions.

Librarian

The librarian is a dedicated team member who has the administrative skills necessary to handle the machine and office procedures involved in the coding and testing effort, as described in Chapter 4, Development Support Libraries. The librarian is responsible for maintenance of project management statistics, and arranges for entry, compilation, and tests of programs as requested by team members. These responsibilities amount to a full-time job.

Other

Additional team members are scheduled into the team as required, as the development cycle progresses. They bring to the team such abilities as specialized application, hardware, or software knowledge, coding speed, or unique coding techniques.

Why Change to Teams?

The chief programmer team organization recognizes that program design is especially important in today's complex application environment and that it is best performed by a senior level professional programmer who also has responsibility for execution of that design.

Reintroducing senior people such as the chief and backup programmers into detailed program coding recognizes another set of circumstances in today's operating system environment. The job control language, data management access methods, utility facilities, and high level source languages are so powerful that there is both a need and an opportunity for using senior level personnel at this detailed, but critical, coding level. The need is to make the best possible use of an extensive set of facilities. The functions of the operating systems are extensive and they are called into play by language forms that require a good deal of study and experience to utilize in the most effective manner.

The very definition of responsibilities in a chief programmer team forces a high degree of public

practice. For example, the librarian is responsible for picking up all computer output, good or bad, and filing it in the notebooks of the development support library, where it becomes part of the public record. Identification of all program data and computer runs as public assets, not private property, is a key principle of chief programmer team operations.

Chief programmer teams can provide the opportunity for professional growth and technical excellence in programming. Since functions involved in maintaining program data are the responsibility of the librarian, more time and energy can be devoted to developing key technical skills and to building the programs. Moreover, the close association with senior level programming personnel who review all code, its testing and integration, provides good training for less experienced programmers and can help prepare them for leadership in future teams.

Chief Programmer Teams in Large Projects

Large projects may require for their execution a number of chief programmer teams, with each reporting to a higher level team, and the top level team reporting to the project manager. The responsibility of each chief programmer team is defined by the structure of the program. Beginning at the top level, each team designs and codes a functional capability down to a set of program stubs. These program stubs become the assignments of the teams at the next level. Each next level team continues the design and coding, possibly to a new set of program stubs, until all the coding is completed. Each chief programmer is directly responsible for members of his own team and for the chief programmers of subsidiary teams.

Over the life of a project, the upper level chief programmer teams go through definite phases of responsibility; that is, design, code, test, and certify. These phases are nested between levels, in that the program stubs produced by a design at one level trigger the next level process, and the certify phases include verification that the program stubs have been carried out satisfactorily by that next level. The top level team will complete its design, code, and test phases early and will spend the remainder of the project certifying the contributions of lower level teams to the system. Each succeeding level starts a little later and has less certifying to do, until the lowest level teams simply design, code, and test their own programs. Note that the team structure mirrors the program structure. In this way, the integrity of the program structure can be preserved during the detailed coding process.

Chapter 4: Development Support Libraries

The development support library (DSL) function supports the environment created by structured programming, top-down programming, and chief programmer team organization. It can also be used apart from these techniques. The technique consists of office and machine procedures used by a librarian to maintain units of structured code being tested and integrated. It is designed to promote efficiency and continuous product visibility during the program development cycle.

Basic Elements and Method of Use

A development support library function (outlined in Figure 9) consists of four elements: a machine-readable internal library, a human-readable external library, machine procedures, and office procedures.

The internal library contains all current project programming data, including program modules, linkage-editing statements, job control statements, and test information. The status of the internal library is reflected in the human-readable *external library* binders that contain current listings of all library members and archives consisting of recently superseded listings. The *machine procedures* consist of standardized JCL and utility control statements to perform such basic procedures as the following:

- Creating and updating libraries
- Retrieving modules for compilations and storing results
- Linkage editing jobs and initiating test runs
- Backing up and restoring libraries
- Producing library status listings

Office procedures are clerical rules used by librarians to perform the following duties:

- Accepting directions marked by programmers in the external library
- Using machine procedures
- Filing updated status listings in the external library
- Filing and replacing pages in the archives.

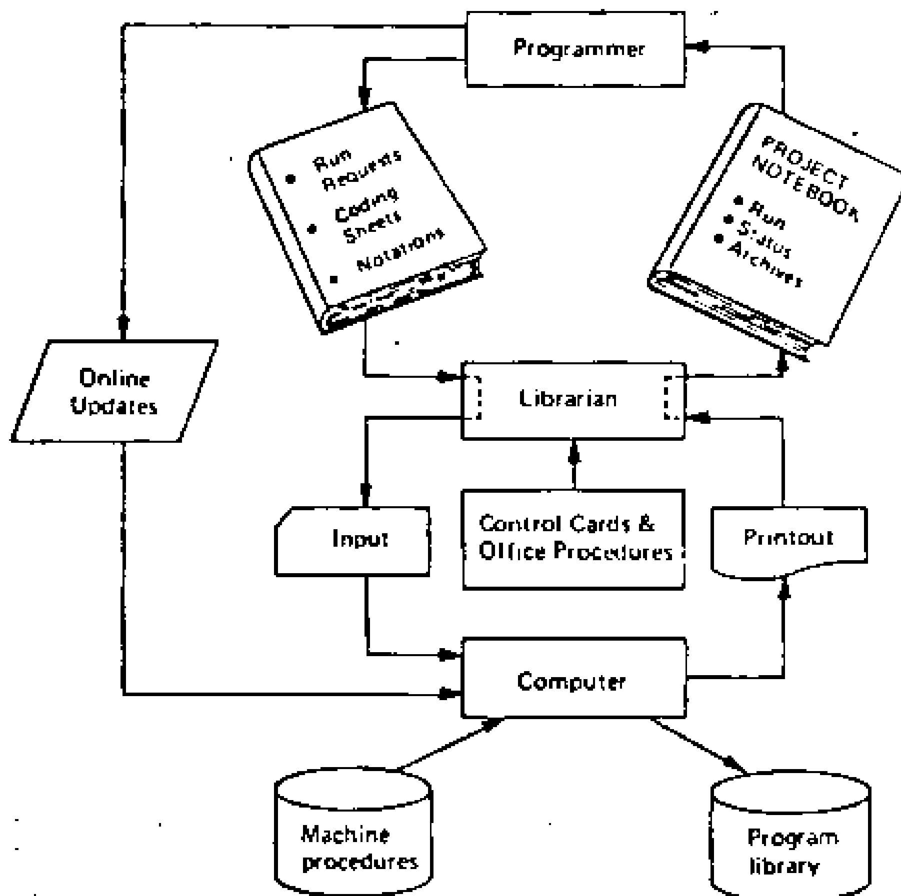


Figure 9. Flow of operations with a system development library

As shown in Figure 9, a programmer using a DSL prepares coding sheets and run requests, and submits them to the librarian, who arranges for the library create or update run. This generates the current version of the program in machine-readable and printed form. The librarian places the printed version into the program's external library binder. Later, the programmer receives these updated binders, which reflect the new status of the internal library. If interactive program development is used, the external library update may be generated at log-off time. Programmer-requested printouts plus copies of all code changes are sent to the librarian, who files them in the external library.

The programmers are freed from such tasks as handling decks and interacting directly with operations, and thus can make more effective use of their time. In addition, a development support library function contributes to manageability, productivity, and program quality by making possible a project whose developing components are visible and available to all, including management. It permits programmers to be certain of the data definition and interface requirements, as well as the operational details of other program units, by reading the actual code in the external library rather than by having to refer to a separate set of documents that may lag behind actual status.

Additional Library Facilities

Other facilities that users might consider for inclusion in their development support library function are:

- *Indentation listing.* Indents structured programming source statements to improve program unit readability.
- *Standards checking.* Checks source statements for adherence to installation standards.
- *Stub handling.* Provides the ability to support top-down programming by generating program stubs with a debug or trace capability and, if desired, routines to simulate time and/or storage to be used by the unit that will replace the stub.
- *Multiple project libraries.* Permits the existence of separate versions of a program. For instance, one set of libraries can contain program units under development while a separate set can contain an operable developing system with which tested program units will be integrated. Still other libraries may be used for operational systems.
- *Program hierarchy listing.* Shows the module calling sequence.
- *Management control listings.* Provides for the collection of statistics on program size, number of changes, compilations, tests, etc.

Chapter 5: Hierarchy plus Input-Process-Output (HIPO)

Application function documentation is often addressed towards the end of a project, and then described with prose or flowcharts, creating a twofold problem: (1) description of function is often incomplete or unclear because of the difficulty in extracting the function of a system from the computer operation performed by the programs, and (2) prose descriptions of function are often voluminous while remaining ambiguous and without a systematic means of relating them to the program modules performing the function. HIPO helps solve these problems by providing the designer with a documentation technique designed for documenting function from the beginning, before programming starts and while it is clear in the designers' minds. It is also designed to reduce the ambiguity and the amount of prose required to document function, and to provide a systematic means of identifying all the functions to be performed and the modules that perform them.

Although HIPO does not provide a systems design or analysis methodology, it does, by providing a design documentation technique, help solve the problems stated above.

In describing the functions to be performed, HIPO diagrams progress from a generalized functional description to greater levels of detail. The functions themselves are described in terms of the process that occurs, with its necessary inputs and resultant outputs. A HIPO package consists of a set of functionally oriented diagrams from generalized to more detailed descriptions of function. Specifically, a typical HIPO package consists of one or more overview diagrams, detail diagrams, and a visual table of contents.

The overview and detail diagrams describe function graphically, with each diagram consisting of three parts: (1) input - the inputs to the function (files, records, fields, control blocks, etc.), (2) process - the process steps that support the function being described, and (3) output - the outputs of the process (files, records, control blocks, etc.) (see Figure 10).

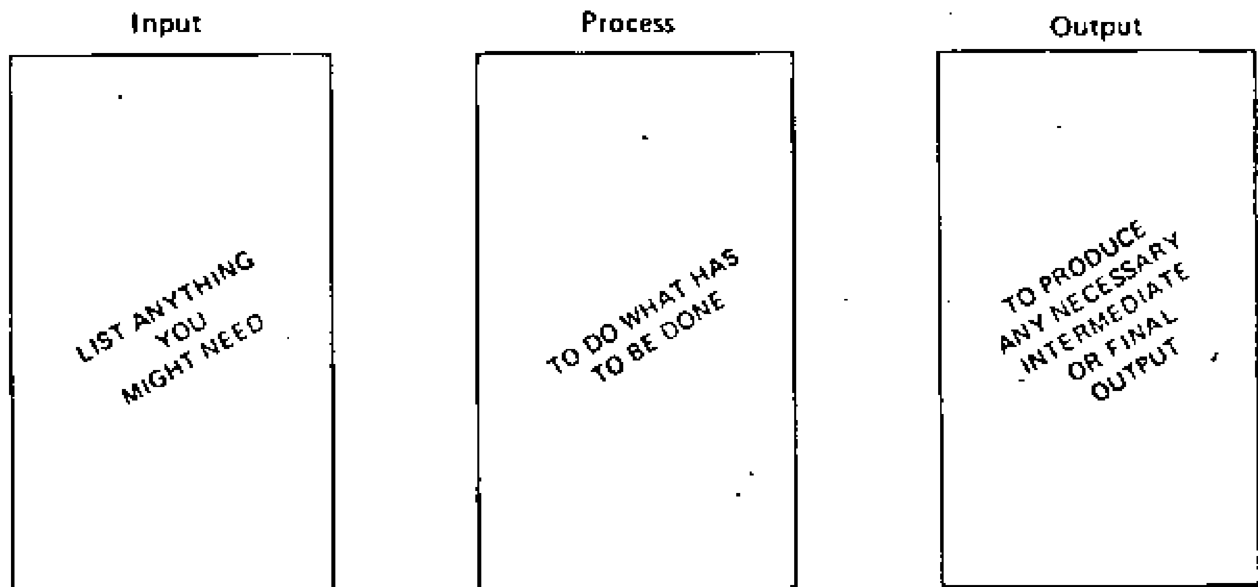


Figure 10. Input-Process-Output graphic relationships

An overview diagram describes, in general, one or more functions expanded by detail diagrams. Figures 11 and 12 illustrate an overview diagram and a detailed diagram, respectively.

In addition to the input, process, and output sections, each detail diagram includes an extended description section, keyed by numbers to the process section. In this section each numbered process may be described in more detail and can point to the program module or modules in which the process is implemented and to the module(s) calling the process. For an overview diagram, the extended description section can further describe each process and may also point to detail diagrams where the numbered processes are further expanded.

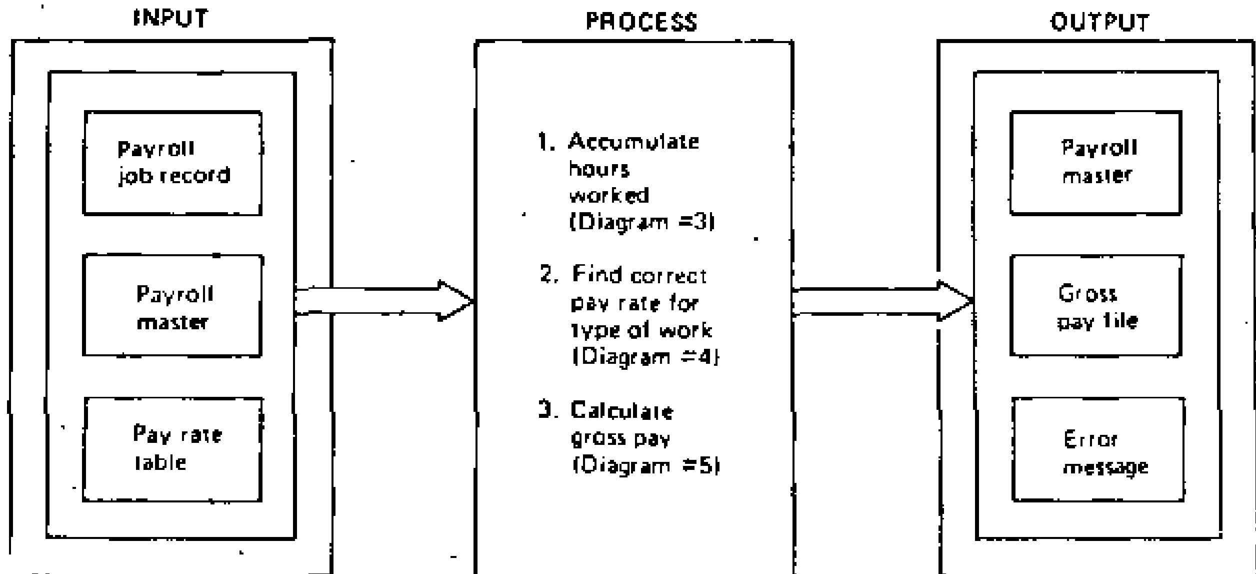
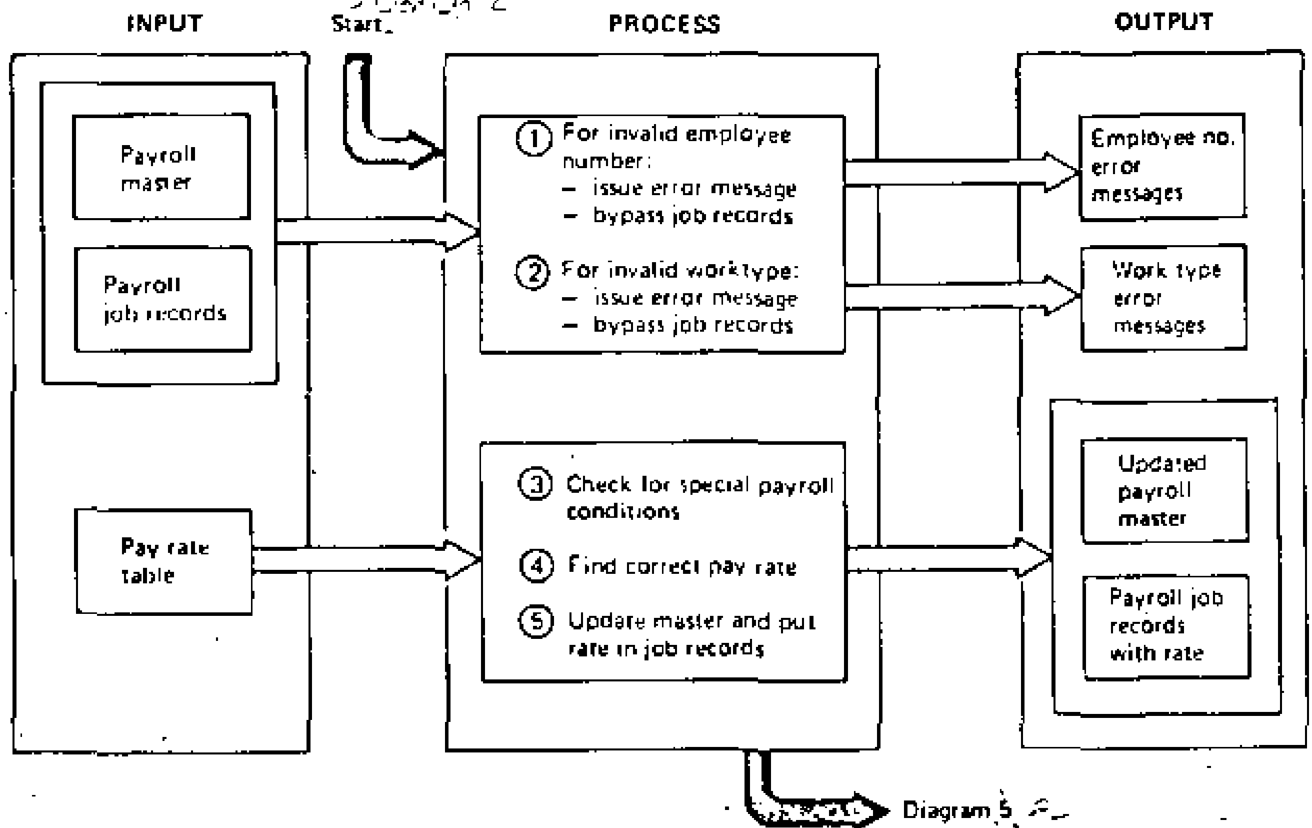


Figure 11. Overview diagram

Diagram 4. Determine Pay Rate



	Extended Description	Routine	Label
1.	The program checks for valid employee number. If valid, job records for that number are bypassed and an error message is printed.	IODNA	DETR
2.	A check is made for correct type of work. If invalid, bypass job records & print error message		
3.	Special conditions such as overtime, shift pay, vacation pay, or holiday pay are checked to help determine correct rate.		
4.	The master record, job records, & pay rate table are all referenced to determine correct pay rate.		
5	When all conditions are checked, payroll job records are rewritten with proper rate, payroll master updated.		

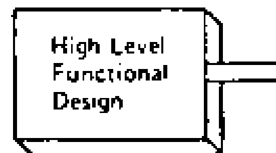
Figure 12. Detail diagram

The visual table of contents (see Figure 13) identifies all the overview and detail diagrams in the package, shows their hierarchical relationships, and permits the reader to quickly locate a particular level of information or a specific diagram.

As shown in Figure 14, HIPO diagrams can be used throughout the development cycle and after its completion. HIPO documentation evolves throughout the development cycle from an initial design package, to a detail design package, and finally to a maintenance package. The initial design package, prepared by a design group at the start of a project, describes the overall functional design of the project and is used as a design aid. The detail design package is prepared by a development group. Using the initial design package as a base, analysts and programmers design in detail, add more levels of HIPO diagrams, and use the resulting package for implementation. The maintenance package, frequently identical to the detail design package, serves as the final documentation for the system.

HIPO can help answer the requirements of the many types of people who rely on the documentation of a system. A development manager, for example, may want a system overview that is under-

- Initial Design Package



- Detail Design Package

- Maintenance Package

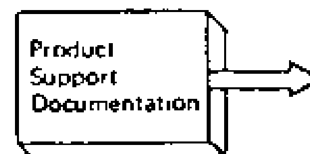


Figure 14 Types of HIPO packages

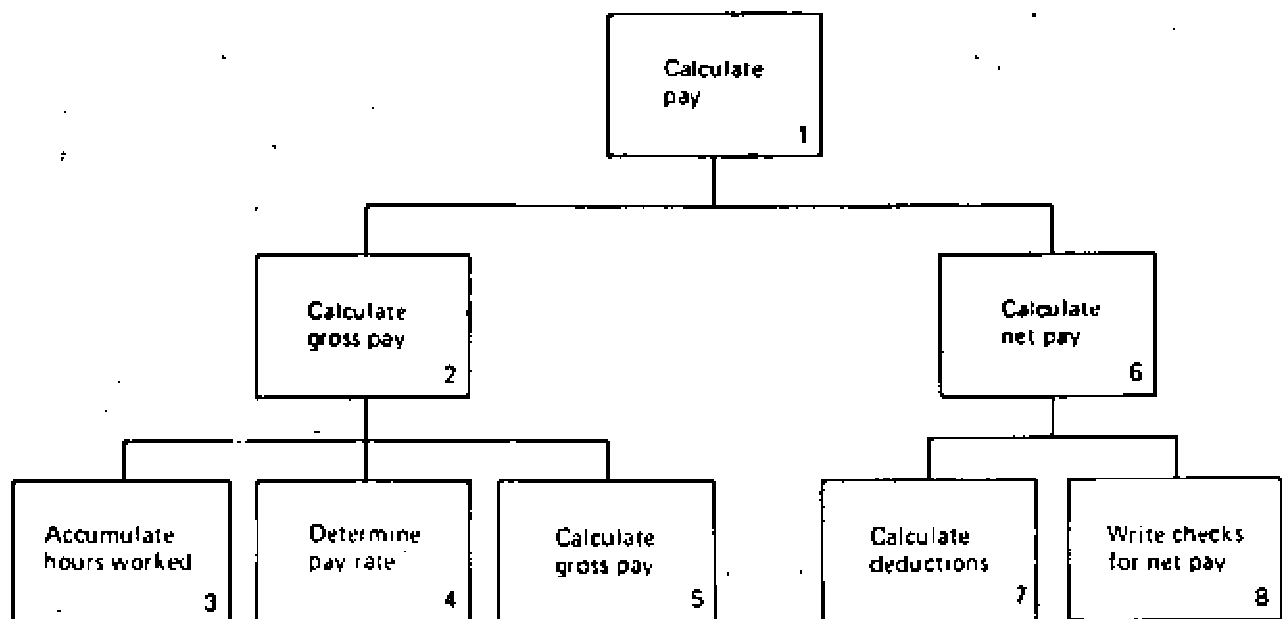


Figure 13. Visual table of contents

standable to a user. An application programmer can use the documentation to determine the detailed programming requirements. A maintenance programmer requires documentation that quickly identifies functions to which changes must be made, and the modules that execute them.

HIPO may be used apart from or in conjunction with the other techniques described in this text. Because of its hierarchical depiction of function, it very effectively supports the hierarchical structures assumed in top-down program development and structured programming.

Chapter 6: Structured Design

We have seen that HIPO is a documentation tool used in the hierarchical functional design process to describe application function. Structured design is a set of techniques for converting that application functional description to a functional, modular program structure. A key objective of structured design is to reduce the complexity of programs by dividing them into individually distinct modules that are independent of one another. Reduced program complexity can reduce the probability of logic errors and ease program modification and maintenance.

Two major structured design concepts are *module strength* (in relationships within a module) and *module coupling* (in relationships between modules), because individually distinct modules that are independent of each other can best be designed by maximizing or strengthening the relationships within a module and minimizing the strength of the relationships established by connections from one module to another.

Module strength. The relationship between the elements of a module can be categorized by the strength of the relationship. For example, a module whose elements are related only coincidentally, does not have the strength equal to one whose elements have a logical relationship to one another. (A module that performs all input and output operations or the editing operations for a program illustrates logical relationships.) Although any application may contain modules that have other strengths, the objective is to produce modules that have functional strength.

Module coupling. 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 (or global variables or modules without their own distinct set of variable names) can result in an enormous number of connections between the modules of a program. The complexity of a system is affected not only by the number of connections but by the degree to which each connection couples (associates) two modules, making them interdependent rather than independent. Coupling is the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules.

The structured design concepts of module strength and module coupling along with the graphic technique provided by HIPO can help application design develop in an orderly manner from a clear statement of the requirement to an intelligible, well constructed set of application functions. The Appendix lists sources of more detailed information on structured design.

Chapter 7: Structured Walk-Throughs and Inspections

Project reviews at the end of various application development phases have long been recognized as a vehicle for determining application development project status and identifying areas that need special attention. Such reviews are usually intended to accomplish the following:

1. Communicate project status to higher-level management and/or project personnel.
2. Establish that the project can be completed within the time and costs allotted, or adjust such schedules and costs.
3. Evaluate the technical accuracy of the project and arrange for the correction of technical errors uncovered as early as possible in the development cycle.

In practice, however, project reviews tend to omit this last function.

The Need for Technical Reviews

Technical reviews during which technical accuracy is evaluated and arrangements for necessary corrections made are needed at various points within the development cycle to help produce a product of higher quality that is easier to maintain while reducing the cost of finding and correcting errors, and to help management to better control the development process (see Figure 15).

Higher quality and improved maintainability are central objectives of data processing management. The number of completed programs in use has been increasing; many of them still contain errors and frequently are difficult to modify; as a result, the number of personnel devoted to program maintenance has also been steadily increasing. It has been estimated that 40-60 percent of every programmer/analyst staff dollar is devoted to the maintenance of operational programs. As the size and complexity of new programs continue to rise, this is likely to increase further.

Achieving quality and maintainability goals while reducing the cost of finding and correcting errors dictates a strategy of *early* error detection. The cost of finding and correcting errors increases substantially as the development process continues. That is, the cost of detection and correction of errors during testing is dramatically higher than during design, the expense is higher still after the program has been placed into productive use.

Technical evaluation of product quality at the end of development phases can help improve management control by providing more realistic evaluations of the developing product's quality, cost, and schedule status, and can make possible timelier corrective action. Two types of technical reviews now in use are structured walk-throughs and application development inspections. The fundamental differences between the two are discussed in this text. For a more detailed discussion of inspections, see *Inspections in Application Development – Introduction and Implementation Guidelines* (GC20-2000).

Type of Review	Beneficiaries	Function/Objectives
Project status review	Management	• Communicate project status • Check and adjust cost and schedules
Technical reviews	Management Developers	• Improve quality • Improve maintainability • Technical evaluation • Reduce costs • Improve management control

Figure 15. Technical reviews versus project status reviews

Structured Walk-Throughs

In a structured walk-through, a developer's work (program design, code, documentation, etc.) is reviewed by fellow project members invited by the developer. Structured walk-throughs, in various forms, are being used by many project development groups. The forms vary in their objectives and procedures, but most have the following characteristics:

1. They are arranged and scheduled by the developer of the work product being reviewed.
2. Management does not ordinarily attend the walk-through.
3. The developer selects the list of reviewers, but in most cases management examines the list to ensure that developers of related products are invited. The walk-through is usually attended by four to six reviewers. Participants can include:
   - Designers of the system, to ensure compatibility and continuity of design
   - Individuals responsible for documenting the function being reviewed
   - Testers responsible for functional and system testing
   - Developers of other parts of the system
   - Developers of other systems that interface with the one being reviewed
4. The reviewers are given the materials four to six days before the walk-through and are expected to review them and come to the session with a list of questions.
5. A typical walk-through is scheduled to last for a specified period, not longer than two hours. If the materials have not been completely reviewed at the end of that period, or if a significantly large list of issues have been created, another walk-through is scheduled for the next convenient time.
6. One person – usually the author whose work is being examined, or perhaps a project team leader – is appointed or elected to guide the session. That person compiles an action list consisting of all errors, discrepancies, exposures, and inconsistencies uncovered during the walk-through.
7. All issues are resolved after the session. The walk-through provides problem detection, not problem resolution.

Inspections

Inspections provide a more formal and rigorous method of performing technical reviews at the end of development phases. Application development inspections, as used in IBM development groups, can be characterized as follows:

1. Inspections appear as separate scheduled activities in the project development plan, and the project schedule contains a time allowance for rework of deficiencies identified in the inspection process.
2. Each development phase, at the end of which work products are to be inspected, is defined, as are the exit criteria for each.
3. At the end of an inspection, formal approval is required that deficiencies have been satisfactorily reworked and exit criteria have been satisfied, before work can proceed to the next development phase.
4. A specially trained moderator schedules and conducts the inspections and chooses the participants, who are usually selected on the basis of special skills or knowledge. The moderator is not the developer of the product being inspected nor a member of the development team; he does not usually devote full time to this role, but is a working analyst or programmer. A moderator's work may be inspected in turn by inspectors from other projects.
5. The inspection of a work product at the end of a development phase consists of six well-defined steps, as shown in Figure 16.
6. The inspections technique emphasizes the accumulation and analysis of data about the types of errors and their frequency of occurrence. As the data base of error patterns grows larger and the moderators gain experience with error detection, the moderators can better analyze the results of inspections, can better help designers and implementers avoid errors, and can help inspection teams learn to do a more thorough job of error detection. Checklists developed from the data base assist in this by assuring that all reasonable questions have been considered during an inspection. Also, by comparing current inspection results against the data base, both developers and management can become aware of situations in which corrective action

Step	Inspection Team Participants	Objectives
1. Planning	Moderator	Scheduling/distributing materials
2. Overview	Designer and other participants	Education
3. Preparation	Participants — individually	Self-study
4. Inspection meeting	Entire group	Finding errors
5. Rework	Moderator, author	Correcting errors (after their summarization and reporting)
6. Follow-up	Moderator, author	Assuring correct rework, improving development and inspections

Figure 16. The six steps of an inspection

must be taken to avoid schedule delays or to reduce the likelihood of creating excessively error-prone modules. Accumulation and analysis of error patterns can also highlight for management those development practices in need of revision or suggest some that could be initiated, thus leading to an improved development process.

In an installation that requires several moderators, management chooses one to serve as an inspection coordinator to oversee the implementation of inspections for the installation. The coordinator controls the establishment and modification of the inspection procedures. He normally serves as the first moderator; trains subsequent moderators; standardizes record keeping procedures, checklists, and exit and reinspection criteria; maintains the installation's inspections data base; and, by analyzing this information, provides such data as time estimates for the various inspection steps, expected error rates, and analyses of errors by type. The coordinator's analyses of the data also assist the moderators in maintaining a high level of effectiveness and efficiency during inspections.

7. An application development process that includes inspections can be compared to an industrial continuous flow process in that each provides the environment for better management control of the process, because management (1) defines the operations in the process, (2) defines the criteria for completion of each operation, and (3) measures the process by collecting data about the functioning of the process.

Key Differences Between Inspections and Structured Walk-Throughs

Key differences exist although there are variations of both techniques. Because of the variations, one or more of these differences may not be applicable in some installations.

The differences may be characterized as follows:

- Typically, walk-throughs are characterized by informality, with the developer (or chief programmer) requesting them. They may be performed on completed activities or during development of an activity. Although proper follow-up usually ensues, no formal approval is generally required before proceeding with further development work as is the case with inspections, although a second walk-through is usually performed when a "large" number of errors has been detected. In the inspections discipline, "exit criteria" for each development phase must be satisfied, errors corrected, and formal approval given before the work can proceed to the next phase.
- The moderator of the inspection is not part of the team that developed the work product. He is trained in the skills required for the moderator's role: planning the inspections, selecting participants, preparing for the inspection meeting, maintaining an efficient pace during the inspection meeting, assuring that all reasonable error possibilities have been considered, keeping interpersonal friction to a constructive level, recording and categorizing errors, following up on rework, and analyzing inspection results. He carries the experience he gains forward from project to

project, and becomes an expert in making the most effective possible use of the participants' time.

Procedural differences include the following:

1. The inspection procedure is divided into six distinct steps, each of which has its own stated objectives. In walk-throughs, these steps exist but are blended together, with several objectives being addressed simultaneously.
2. At a walk-through meeting, the developer of the work product usually conducts the meeting and "reads" the materials. In inspections, the moderator conducts the meeting and designates someone other than the developer to read the materials so that the developer's interpretation cannot inadvertently cover up an error.
3. Errors detected during the preparation period and discussed with the developer are usually not brought up at a walk-through meeting. During inspection meetings, *all* errors are noted and described to establish error patterns, improve the skills of all the participants, and flag any schedule slippage as early as possible.
4. Management does not usually participate in meetings in either form of review. In the case of structured walk-throughs, they do not attend because their presence may interfere with the free flow of discussion among the working group. In the case of inspections, managers are not discouraged from attending, but usually cannot add valuable input to an essentially technical discussion; they are therefore not needed. Management is, however, informed of the results of inspection meetings. In addition to traditional quantitative data, such as lines of code completed, management receives data regarding program quality (error rates) and time expected for rework and retesting.
5. Inspections emphasize uniformity and completeness through the use of checklists.
6. Errors are categorized during an inspection meeting and entered into a data base to improve the developers' and management's understanding of the types of errors that are occurring and where they most often occur. This information can be used to develop acceptable error rate standards to determine whether a module should be reinspected or rewritten. It can also assist management, testers, and those who are developing related modules to plan their work better and maintain more accurate schedules.

Briefly stated, the walk-through procedure relies heavily on technical team self-control and confines the visibility of shortcomings to within the development team itself. On the other hand, the inspection process allows for such visibility to extend beyond the development team and imposes technical controls from sources (the moderator and management) that are external to the team.

Note that the differences between structured walk-throughs and inspections tend to diminish as more of the formalities of inspections are included in walk-throughs. For instance, walk-throughs are now frequently scheduled into a project development plan.

Appendix: Reading List, Supporting Materials, and Related Programs

Development Support Libraries

OS Development Support Libraries (GC20-1663)

Hierarchy plus Input-Process-Output

HIPO – A Design Aid and Documentation Technique (GC20-1851)
HIPO Worksheet (GX20-1970)
HIPO Template (GX20-1971)
HIPODRAW (Installed User Program 5796-PFF). The HIPODRAW program generates HIPO and visual table of contents diagrams. It can process user input submitted in batch mode under DOS/VS or OS/VS or in interactive mode with TSO or VM/370 CMS. Related publications are:
Program Description/Operations Manual (SH20-1728)
User's Guide (SH20-1821)
Systems Guide (LY20-2201)

Inspections

M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, Volume 15, Number 2, 1976. A reprint of this article is available under order number G321-5033.
Inspections in Application Development – Introduction and Implementation Guidelines (GC20-2000).

Structured Design

W.P. Stevens, G.J. Myers, L.L. Constantine, "Structured Design", *IBM Systems Journal*, Volume 13, Number 2, 1974. A reprint of this article is available under order number G320-5323.
G.J. Myers, "Composite Design Facilities of Six Programming Languages", *IBM Systems Journal*, Volume 15, Number 3, 1976. A reprint of this article is available under order number G321-5034.

Structured Programming

J.G. Rogers, "Structured Programming for Virtual Storage Systems", *IBM Systems Journal*, Volume 14, Number 4, 1975. A reprint of this article is available under order number G321-5023.
An Introduction to Structured Programming in COBOL (GC20-1776)
An Introduction to Structured Programming in PL/I (GC20-1777)
An Introduction to Structured Programming in FORTRAN (GC20-1790)
Structured Programming, Independent Study Program course (SBOF-3640). The individual items in this course are: *Textbook* (SR20-7149), *Workbook* (SR20-7150), *Template* (SR20-7151).
Structured Programming Macros (Field Developed Program 5798-CLF). This FDP provides the full capabilities of the structure theorem to Assembler Language users. Related publications are: *Availability Notice* (GB21-1957) and *Program Description/Operations Manual* (SB21-1958).
TSO Display Support and Structured Programming Facility (Program Product 5740-XT8). The Structured Programming Facility (SPF) is a programming development tool for the TSO environment designed to increase productivity in developing and modifying programs. It supports VS2-TSO users who have a 24-line 3270 display terminal equipped with a full EBCDIC keyboard, including 12 program function keys. It can increase programmer productivity through:

- Display presentations that prompt the user and simplify command/data entry
- Time-saving use of program function keys for commonly performed operations
- Features that facilitate structured programming in a TSO environment

Related materials are: *Program Reference Manual* (SH20-1975), *Program Logic Manual* (LY20-2339), *General Information Manual* (GH20-1974), *Program Product Specifications* (GH20-4521).

General References (Contain information about more than one of the techniques)

P. Van Leer, "Top-Down Development Using a Program Design Language", *IBM Systems Journal*, Volume 14, Number 4, 1975. A reprint of this article is available under order number G321-5032.
J.F. Stay, "HIPO and Integrated Program Design", *IBM Systems Journal*, Volume 15, Number 2, 1976. A reprint of this article is available under order number G321-5031.
Improved Programming Technologies, In-Shop Training Course (SBOF-3655). The individual items in this course are: *Structured Programming Textbook* (SR29-7149), *Structured Programming Workbook* (SR29-7150), *Structured Programming Template* (SR29-7151), *Improved Programming Technologies Student Guide* (SR29-5100), *Improved Programming Technologies Instructor Teaching Guide* (SR29-5101), *Improved Programming Technologies Instructor Handbook* (SR29-5102).
Programming Productivity Techniques, System Science Institute Course W9889.

Impr Programming Technologies – An Over

Installation Management Manual

GC20-1850-1

READER'S COMMENT FORM

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name and mailing address:

____

____

____

What is your occupation? ______________________________

Number of latest Newsletter associated with this publication: ________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

GC20-1850-1

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

First Class
Permit 40
Armonk
New York

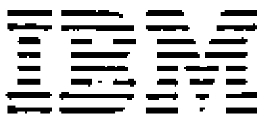
Business Reply Mail

No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department 824
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape | Please Do Not Staple | Fold and tape



International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
Hamilton Avenue, White Plains, N.Y., U.S.A. 1

Cut or Fold Along Line

Improved Programming Technologies — An Overview

Installation Management Manual

GC20-1850-1

READER'S COMMENT FORM

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name and mailing address:

What is your occupation? ______

Number of latest Newsletter associated with this publication: ______

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

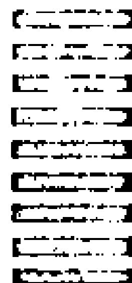
Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

GC20 1850-1

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

First Class
Permit 40
Armonk
New York



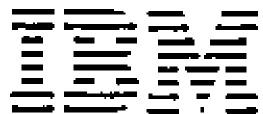
Business Reply Mail

No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department 824
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape | Please Do Not Staple | Fold and tape



International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

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

TECNICAS MODERNAS DE DESARROLLO Y
ADMINISTRACION DE PROGRAMACION

TECNICAS DE CODIFICACION

MARZO, 1983

Palacio de ía Calle de T ba 5 p er piso Γ g. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

- LA TRADUCCION A LENGUAJE ESTRUCTURADO SE CONVIERTE EN UNA TAREA TRIVIAL

FORTRAN 77

ESTRUCTURA

```
MIENTRAS ( BANDERA = 'S' )
     LLAMA ACTUALIZA PARÁMETROS
     LIMPIA PANTALLA
     ESCRIBIR ( PANTALLA )
        CAMBIOS ( S,N )
     LEER ( PANTALLA ) BANDERA
FIN ( MIENTRAS )
```

IMPLEMENTACION FO-77

```
WHILE ( BANDERA.EQ.'S'  )
    CALL ACTLZA
    REWIND 3
    WRITE ( 3,5000 ) CAMBIOS (S,N)

    READ (3,5100) BANDRA
END WHILE
```

LA TRADUCCION A UN LENGUAJE POPULAR PUEDE REPRESENTAR UN MAYOR PROBLEMA

COBOL

ESTRUCTURA

```
SI P ENTONCES
    SI Q  ENTONCES
      INSTRUCCION - 1

    SINO
      INSTRUCCION 2
    FIN (SI)
      INSTRUCCION - 3
    SINO
      INSTRUCCION - 4

FIN (S)
```

IMPLEMENTACION COBOL

```
IF P THEN
      PERFORM IF-INTERNO
      INSTRUCCION - 3
ELSE
      INSTRUCCION - 4
IF-INTERNO SECTION
      IF Q  THEN
      INSTRUCCION - 1
ELSE
      INSTRUCCION - 2
```

FORTRAN 66

EN EL CASO DE LENGUAJES POPULARES QUE NO CUENTAN CON LAS ESTRUCTURAS DE CONTROL ESTRUCTURADAS, LA ESTRUCTURA LOGICA DEL PROGRAMA DEBE SER CONSTRUIDA UTILIZANDO LA INSTRUCCION DE RAMIFICACION CONDICIONADA ('GO TO') LA CUAL, SI SE UTILIZA INDISCIPLINADAMENTE OBSCURECE LA LOGICA DEL PROBRAMA.

EN ESTE CASO DEBEMOS RESTRINGIRNOS A USAR TECNICAS DE CODIFICACION PARA SIMULAR LAS ESTRUCTURAS BASICAS EN FORMA ORDENADA.

* EMULACIONES
* PRECOMPILADORES

TECNICAS DE DESARROLLO

GUIA PARA LA CODIFICACION
INTEGRACION Y VERIFICACION DE LOS
MODULOS DE UN PROGRAMA

TECNICAS DE CODIFICACION

FACILITAN LA TRADUCCION DEL DISEÑO
DETALLADO A UN LENGUAJE DE
PROGRAMACION MANTENIENDO LA
ESTRUCTURA JERARQUICA DEFINIDA

LA TRADUCCION A UN LENGUAJE DEPENDE
DE LAS CARACTERISTICAS DEL PROPIO LENGUAJE

ESTRUCTURADOS

ALGOL, PASCAL, PL/ 1, FORTRAN 77

POPULARES

BASIC, COBOL, FORTRAN 65

E M U L A C I O N E S

FORTRAN 66

ESTRUCTURA (SI-ENTONCES-SINO)

Si P entonces	IF (.NOT.P) GO TO N1
Instrucción-1	Instrucción-1
	GO TO N2
Sino	N1 CONTINUE
Instrucción-2	Instrucción-2
Fin(si)	N2 CONTINUE

ESTRUCTURA (MIENTRAS)

Mientras (condición)	N1 IF (.NOT.condicion) GO TO N2
Instrucción-1	Instrucción-1
	GO TO N1
Fin(mientras)	N2 CONTINUE

ESTRUCTURA (EJECUTA)

Ejecuta	GO TO N2
	N1 IF (condicion) GO TO N3
	N2 CONTINUE
instrucción-1	Instrucción-1
	GO TO N1
Hasta (condición)	N3 CONTINUE
	END IF
	N2 CONTINUE

ESTRUCTURA (REPITE)

Repite inicial, final, incr.	INDICE= INICIAL
	N1 CONTINUE
Instrucción-1	Instrucción-1
	INDICE=INDICE+INCR
Fin(repite)	IF(INDICE.LE.FINAL)GO TO N1

PRECOMPILADORES

SISTEMA DE COMPUTADORA QUE EXTIENDE LA SINTAXIS DE UN LENGUAJE DE PROGRAMACION ESPECIFICO:

PERMITE AL PROGRAMADOR CODIFICAR SUS PROGRAMAS UTILIZANDO LAS ESTRUCTURAS BASICAS EN COMBINACION CON LAS INSTRUCCIONES PROPIAS DEL LENGUAJE :

VENTAJAS

1. PORTABILIDAD
2. MENORES VIOLACIONES A PROGRAMACION ESTRUCTURADA
3. MAYOR APROXIMACION ENTRE DISEÑO Y PROGRAMACION
4. MAYOR CONFIABILIDAD
5. MAYOR MANTENIBILIDAD

EJEMPLOS

- META COBOL
- WATFOR
- HIFTRAN
- BEST

ESTILO DEL PROGRAMADOR

- LA CLARIDAD DE UN PROGRAMA DEPENDE DEL ESTILO DEL PROGRAMADOR

- GUIA DE ESTILO

NORMAS DE CODIFICACION

TECNICAS DE INTEGRACION

IDENTIFICACION DE ERRORES DE COMPUTO QUE SE ORIGINAN EN LA ETAPA DE INTEGRACION.

ASOCIADAS AL ENFOQUE DE INTEGRACION INCREMENTAL -

- PRUEBAS DE ARRIBA HACIA ABAJO (PARAB)
  MODULOS CIEGOS
- PRUEBAS DE ABAJO HACIA ARRIBA (PABAR)
  MODULOS DIRECCIONADORES

CONCLUSION

DADA LA COMPLEJIDAD EN EL DISEÑO DE CASOS DE PRUEBA SE RECOMIENDA USAR LA TECNICA 'PABAR' PARA LA INTEGRACION INCREMENTAL DE PROGRAMAS.

NORMAS DE CODIFICACION

CONJUNTO DE REGLAS DE ESTILO QUE PERMITEN QUE LOS PROGRAMAS SEAN FACILES DE LEER, REVISAR, MODIFICAR Y POR LO TANTO MANTENER.

UNIFICAR EL ESTILO DE PROGRAMACION SIN LIMITAR LA CREATIVIDAD:

ESTILO ES LA SELECCION ADECUADA DE HABITOS DE PROGRAMACION.

CONSIDERACIONES

A. EL PROGRAMA DEBE SER CORRECTO
B. EL PROGRAMA DEBE SER FACTIBLE
C. EL PROGRAMA DEBE SER CLARO
D. EL PROGRAMA DEBE TENER BAJOS COSTOS DE DESARROLLO
E. EL PROGRAMA DEBE TENER BAJOS COSTOS DE PRUEBA
F. EL PROGRAMA DEBE TENER MINIMOS COSTOS DE MANTENIMIENTO.
G. EL PROGRAMA DEBE SER FACILMENTE MODIFICABLE
H. EL PROGRAMA DEBE TENER DISEÑO POCO COMPLICADO
I. EL PROGRAMA DEBE SER EFICIENTE

NORMAS DE CODIFICACION

- UNA ENTRADA, UNA SALIDA, UNA FUNCION
- CONVENCION DE NOMBRAMIENTO
- LONGITUD DE PROGRAMAS Y MODULOS
- BANDERAS DE COMPILACION CONDICIONAL
- DEFINICION EXPLICITA DE VARIABLES
- AUTO DOCUMENTACION
  + COMENTARIOS DE PROLOGO
  + COMENTARIOS EXPLICATIVOS
- INDENTACION
- ESPECIFICACIONES DE ERROR EN OPERACIONES E/ S
- NUMEROS MAGICOS
- INSTRUCCIONES PROHIBIDAS
  + GOTO
  + EQUIVALENCIA

4. NOMBRAMIENTO DE ARCHIVOS

Con el objeto de que los nombres de los archivos generados en la computadora HARRIS sean únicos, identifiquen al grupo o subsistema al que pertenecen y den una idea de su contenido, se ha convenido en usar la siguiente nomenclatura:

a) Existen 12 tipos de archivos primarios, los cuales son:

1.	Programa fuente	- S
2.	Listado de Compilación	- L
3.	Módulo Ejecutable	
4.	Lanzador	- J
5.	Programa Macro-Ensamblador	- M
6.	Módulo de biblioteca (personal)	- B
7.	Monitor Common	- C
8.	Archivo con un texto	- O
9.	Archivo de datos (DIRECTO)	- D
10.	Archivo de datos (SECUENCIAL)	- E
11.	Archivo de datos (INEXADO)	- I
12.	Archivo temporal	- T

a.1) Un archivo de programa fuente (S) es un área en disco, la cual contiene enunciados en FORTRAN-77.

a.2) Un requerimiento del ciclo de vida en el desarrollo de programas bajo el sistema M9000, es el mantener un área de programas ensamblados/compilados. Para soportar este requerimiento se han creado las áreas del tipo L.

a.3) La salida "vulcanizada" de un específico lanzador crea un módulo ejecutable. Los nombres de estos módulos son usados por el sistema M9000, por lo cual deben de ser únicos y contener 4 caracteres.

*lphl

a.4) Un archivo del tipo lanzador (J) contiene instrucciones que compilan y vulcanizan un archivo fuente. Estas instrucciones son codificadas en un lenguaje de bajo nivel.

a.5) Varios lanzadores se pueden agrupar por medio de un archivo del tipo macro (M).

a.6) Archivos de biblioteca (B) pueden ser creados para ser utilizados y anexados en el tiempo de ejecución.

a.7) Archivos que contienen los registros de un monitor común serán iniciados con la letra C.

a.8) Existen archivos que únicamente contienen párrafos con texto y los cuales no son ejecutables. Estos archivos, sin embargo, podrán ser editados por medio del paquete de utilería 'FORMAT' y deberán de empezar con la letra O.

a.9) Los archivos de datos pueden ser accesados por diferentes métodos. Aquellos que se accesan directamente (random) deberán de contener la letra D, los secuenciales (E) y finalmente los inexados la letra I.

a.10) Existen procesos que crean archivos para almacenar información temporal (T) que será destruida al término de la corrida.

*lphl

b) Resúmen

TIPO DE ARCHIVO:
(1 caracter)

O - FUENTE ORIGINAL BSP
S - FUENTE
L - LISTADO

EJECUTABLE
J - LANZADOR
M - MACRO-ENS
B - BIBLIOTECA
C - MONITOR COMM
X - TEXTO
D - DATOS DIRECTO
E - DATOS SECUENCIAL
I - DATOS INDEXADO
T - TEMPORAL
R - RELOCALIZABLE
U - RESULTADOS
A - PARAMETROS

C A L I F I C A D O R * | | 0 0 | | | | | |

SUBSISTEMA:
(2 caracteres)

DE - DESPACHO ECONOMICO
CI - CALCULO INTERC.
PC - PRONOSTICO DE CARGA
CH - COORDINACION HIDROT.
AD - ADMON. PROY.
BI - BSP
*1phl

NOMBRE DEL ARCHIVO:
(4 caracteres)

DESCRIPCION DEL ARCHIVO
EN FORMA RESUMIDA Y ASIGNADA POR EL DISEÑADOR.

c) Ejemplos:

1. Programa fuente para la inicialización del pronóstico de carga (parámetros meteorológicos)
ØØ43PCDP*S:PCIMET

2. Módulo ejecutable para la inicialización del pronóstico de carga:
PCIM

3. Programa temporal en biblioteca del auditor de código:
ØØ6ØBIAR*T:BIAUCO

4. Archivo de datos directo para coordinación hidro:
ØØ44CHDP*D:CHMIPG

5. Archivo de Monitor Common en el programa de Cálculo de Intercambios
ØØ5ØADAR*C:CIMCOM

6. Archivo de biblioteca personal
ØØ2ØCOCA*B:BIBSP

*lphl

5. NOMBRAMIENTO DE SUBRUTINAS

a) En la escritura de un Diseño de Programa de Computadora (DPC) se intenta pseudocodificar un conjunto de bloques, que sean fácilmente entendidos y por consiguiente mantenidos y corregidos. Con esta idea en mente, cada elemento de un diseño, ya sea una subrutina o una variable interna o externa deberá de ser plenamente identificable. Aun más, los nombres de las subrutinas deberán de ser representativos de su función, sin que resulten innecesariamente largos. La mayoría de los nombres de subrutinas deberán de contener de uno a seis caracteres, pues los nombres demasiados largos tienden a distraer la atención de la lectura, generan errores y finalmente, no son compatibles con otros compiladores.

b) Los nombres de subrutinas deberán estar formadas por dos letras iniciales de un verbo en modo imperativo, tal como se muestra en la lista mostrada en el ANEXO 2.

c) Los restantes cuatro caracteres seran asignados por el diseñador tratando de representar en forma mnemónica la subfunción que se realiza.

d) Ejemplos:

Subrutina para validar los datos de inicialización
VADAIN
Subrutina de Optimización Dantzig-Wolfe
OPDZWL
Subrutina de obtención de datos de usuario
OBDAUS

*lphl

INSTITUTO DE
INVESTIGACIONES
ELECTRICAS

Anexo 2

AbOrta	AO	ConVierte	CV	GRaba	GR	ReCibe	RC
AbRe	AR	CoPia	CP	GUarda	GU	ReGistra	RG
ACepta	AC	CRea	CR	IDentifica	ID	ReeMplaza	RM
AcoModa	AM	CrItica	CI	INcrementa	IN	RePorta	RP
Acumula	AU						
AcTualiza	AT	CrUza	CU	IniCializa	IC	ReAliza	RA
AGrega	AG	DEcrementa	DE	InSerta	IS	ReeStablece	RS
ALtera	AL	DeCodifica	DC	InTercala	IT	ReTiene	RT
ANaliza	AN	DeSplega	DS	LEe	LE	ResoLver	RL
ARchiva	AR	DeTecta	DT	LIbera	LI	ReVisa	RV
ArmA	AA	DeteRmina	DR	LiGa	LG	SAca	SA
ASigna	AS	DIstribuye	DI	LiMpia	LM	SaLta	SL
BOrra	BO	EDita	ED	LOcaliza	LO	SElecciona	SE
BUsca	BU	EJecuta	EJ	MArca	MA	SePara	SP
CAlcula	CA	EliMina	EM	MEzclar	ME	SUministra	SU
						Suma	SM
CaPta	CP	ENcadena	EN	MOdifica	MO	SuperVisa	SV
CarGa	CG	EnCuentra	EC	MUeve	MU	SupRime	SR
CIerra	CI	EnLaza	EL	NUmera	NU	SuSpende	SS
ClasiFica	CF	EScribe	ES	OBtención	OB	TErmina	TE
COdifica	CO	EsPecifica	EP	OPtimizar	OP	TOma	TO
ColoCa	CC	EstaBlece	EB	ORdenar	OR	TRae	TR
CoMienza	CM	EXamina	EX	POsiciona	PO	TrAduce	TA
ComplEta	CE	ExTrae	ET	PRocesa	PR	TransForma	TF
ComprueBa	CB	FIja	FI	ProDuce	PD	TransPorta	TP
CoNstruye	CN	FOrmatea	FO	PrueBa	PB	VAlida	VA
ConSulta	CS	FoRmar	FR	PUrga	PU	VErifica	VE
ControLa	CL	GEnera	GE	REchaza	RE		

*lphl

6. NOMBRAMIENTO DE VARIABLES EXTERNAS, INTERNAS Y FUNCIONES

Con el objeto de poder identificar en forma única y a la vez, el que sea representativo el nombre de las variables que se usen en la programación del departamento, se utilizará la siguiente convención:

a) FORTRAN77 permite el uso de identificadores de longitud variable, entre 1 y 63 caracteres. Se recomienda que los nombres de las variables tengan una longitud máxima de 6 caracteres, por compatibilidad con otros compiladores.

b) Todas las variables deben ser explícitamente declaradas (empleando los declaradores REAL, ENTERA, COMPLEJA, LOGICA, CARACTER). No dejar que el sistema asuma que las variables que empiezan con las letras A-H y O-Z sean -- del tipo REAL y las que comienzan con I-N sean del tipo ENTERO. Esta práctica, además de que puede alterar el significado del contenido de la variable, reduce en uno el número de caracteres disponibles para el identificador.

c) Se considerán dos tipos de variables, internas y externas. Las internas son aquellas que se usan y comparten por subrutinas de un mismo programa. Las externas, son aquellas que se transfieren de un programa a otro y que fueron definidas después de que el Documento de Interfaz Externo - (DIE) había sido terminado.

d) Los nombres de variables que cumplan con una función especial deberán de constar de un prefijo que indique dicha función:

*lphl

TIPO DE VARIABLE	INTERNA	EXTERNA
Parámetro de common	HC	XC
Parámetro de Monitor	HM	XM
Common		
Función	HF	XF

c) Ejemplos:

```
COMMON/CPCHI/HCPPIE, HCECS, HCPNIA
HFCNVG
COMMON/MICOM/HMBLOQ1(256), HMBLQ2/256)
```

f) Los nombres de las variables deben representar el contenido de las mismas. Las variables son creadas por el diseñador del programa y usualmente representan en forma mnemónica el significado de la misma. Sin embargo, cada diseñador asigna arbitrariamente este nombre mnemónico, lo cual crea muchas veces confusión.

Con el objeto de uniformizar esta asignación mnemónica a los nombres de las variables, se propone la siguiente metodología:

f.1) De el nombre completo de la variable sin abreviaturas
PRONOSTICO DE CARGA

f.2) Elimine las preposiciones y las vocales
PRNSTCCRG

f.3) Si existen consonantes contiguas repetidas, entonces se elimina una de ellas
PRNSTCRG

f.4) Con las conosonantes restantes tratar de formar el identificador, de longitud máxima de 6 caracteres y tomando aproximadamente un mismo número de caracteres por palabra.

PRNST CRG
PRNCRG

f.5) En caso de poder sustituir una consonante por otra, cuya fonética lleva implícita la vocal eliminada, deberá hacerse

SECUENCIA   SQNCIA
PRNCRG   PRNKRG

f.6) En caso de que no se completen los seis caracteres consonantes, se podrá emplear aquella vocal con mayor significado. Es muy difícil definir cual es la vocal con más significado, sin embargo, use su "sentido común".

CONTADOR CNTDR   CONTDR

f.7) En caso de que el nombre de la variable sea igual o menor que 6 caracteres, el mnemónico quedará igual al nombre original de la variable.
PASO

f.8) En caso de que el nombre de la variable sea demasiado largo, se tomará la primera inicial de cada palabra.

Potencia Programada de Intercambio Externo - PPIE
Factores de Sensitividad Restringidos para Intercambio de áreas con Sistemas Externos - FSRISE

*lphl

```
      PROGRAM TIPS
C
C** ********************************************************************
C** ************************ TIPS *************************************
  *********************************************************************
                                                                      *
C    P R O G R A M A S   D E   A P L I C A C I O N   A V A N Z A D A   *
C                                                                     *
C      I N S T I T U T O   D E   I N V E S T I G A C I O N E S        *
C                                                                     *
C                      E L E C T R I C A S                            *
C                                                                     *
C     D I V I S I O N   D E   S I S T E M A S   D E   P O T E N C I A *
C                                                                     *
C   D E P A R T A M E N T O   D E   A N A L I S I S   D E   R E D E S *
C                                                                     *
C**********************************************************************
C                                                                     *
C       PROGRAMA FUENTE                                               *
C          400EPO*S:TIPS                                              *
C                                                                     *
C       LANZADOR                                                      *
C          400EPO*J:TIPS                                              *
C                                                                     *
C       PROPOSITO                                                     *
C         LA COMPONENTE TIPS SE ENCARGARA DE LLAMAR A EJECUCION A     *
C         TODAS LAS DEMAS COMPONENTES DEL PROGRAMA                    *
C                                                                     *
C       REFERENCIAS                                                   *
C         D.P.C.  -                                                   *
C         R.P.C.  -                                                   *
C                                                                     *
C       LIMITACIONES                                                  *
C          NINGUNA                                                    *
C                                                                     *
C       ACLARACIONES                                                  *
C          NINGUNA                                                    *
C                                                                     *
C       MODULOS EMPLEADOS                                             *
C          LEPEMS - LECTURA DE PARAMETROS EMS                         *
C          LEIPRG - LECTURA DE INTERCAMBIOS PROGRAMADOS               *
C          VACGLB - VALIDA COMPONENTE GLOBAL                          *
C          LEDGAL - LECTURA DE DATOS GENERALES                        *
C          ORINFO - ORDENA INFORMACION                                *
C          CASIMA - CALCULO COMPONENTE GLOBAL POR SUBSISTEMA          *
C          CAACIA - CALCULO COMPONENTE GLOBAL POR AREA/COMPANIA       *
C                                                                     *
C       NOMBRE Y FECHA DE IMPLEMENTACION                              *
C          VICTOR BUERIA       10 DIC 82                              *
C                                                                     *
C       NOMBRE Y FECHA DE REVISION(ES)                                *
C          BENITO ZEHYLINSKI                                          *
C                                                                     *
C** ********************************************************************
C
C       -----------------------------------------
C       * INICIA COMPONENTE TIPS                 *
C       * TABULADOR DE INTERCAMBIOS PROGRAMADOS  *
C       -----------------------------------------
$ADD 400EPO*A:TIPS01
C
      INTEGER * 3 FLAT , CGLOK , COUCOM , LLAVES , LISAE , DTOAE
```

```
C          * INICIALIZACION DE BANDERA DE INFORMACION CORRECTA *
C          ----------------------------------------------------
           CGLOK = 0
C          ----------------------------
C          * LECTURA DE PARAMETROS EMS *
C          ----------------------------
           CALL LEPEMS ( CGLOK )
           CGLOK = 1
C          ---------------------------------------------------------------
C          * EJECUTAR EL MODULO DE LECTURA DE INTERCAMBIOS PROGRAMADOS *
C          ---------------------------------------------------------------
           CALL LETPRE ( CGLOK )
           IF ( CGLOK .EQ. 1 ) THEN
C              ----------------------------------------------------
C              * INFORMACION DE LA COMPONENTE GLOBAL CORRECTA      *
C              * INFORMACION DE LOS INTERCAMBIOS PROGRAMADOS OK.   *
C              ----------------------------------------------------
:SKE7 3
              WRITE(6,*) "CGLOK=",CGLOK
:ESKP
              CALL OPIMFO
C              ----------------------------------------------
C              * CALCULAR COMPONENTE GLOBAL POR SUBSISTEMA *
C              ----------------------------------------------
              CALL CASTMA
C              ----------------------------------------------
C              * CALCULAR SEGMENTOS NETOS POR AREA/COMPAÑIA *
C              ----------------------------------------------
              CALL CAAFIA
           END IF
        ELSE
C          ---------------------------------
C          * APERTURA DE ARCHIVOS INCORRECTA *
C          ---------------------------------
:SKE7 2
           WRITE(6,*) " ERROR EN APERTURA AE ",J
:ESKP
           CALL TIPMSG (17 , LTSAE(J) , 0 , 0D , 0D )
        END IF
C       --------------------------------
C       * CERRAR ARCHIVOS ESTRUCTURADOS *
C       --------------------------------
        ERR = .FALSE.
        J = J-1
        WHILE (.NOT.ERR.AND.J.GE.1)
           CODCOM = 0
:SKE7 2
           WRITE(6,*) "ANTES DE CERRAR ARCHIVOS ESTRUCTURADOS"
           WRITE(6,*) "J:",J,"LISAE:",LISAE(J),"LLAVE:",LLAVES(J)
           WRITE(6,*) "CODCOM:",CODCOM
:ESKP
           CALL CIFRAE(LTSAE(J),LLAVES(J),CODCOM)
:SKE7 2
           WRITE(6,*) "DESPUES DE CERRAR ARCHIVOS ESTRUCTURADOS"
           WRITE(6,*) "J:",J,"LISAE:",LISAE(J),"LLAVE:",LLAVES(J)
           WRITE(6,*) "CODCOM:",CODCOM
:ES 9
           IF (CODCOM.EQ.0) THEN
              J = J-1
           ELSE
              IF (CODCOM.EQ.3) THEN
:AS
          TOK     12
          DLU     SDELAY
```

```
      INTEGER * 6 CPUT
C
      LOGICAL      ERR
C
      SPECIAL COMMON LLAVES , MLTIR   , MISTR   , MLINM   , MISMA   ,
     >                 MPROMT , MTROMI , UTMAE   , LISAF
C
      COMMON / LLAVES / LLAVES ( 1 )
      COMMON / LISAF  / LISAF  ( 1 )
C
C     ------------------
C     * AYUDAS DE DEBUG *
C     ------------------
:SKEZ 5
      CALL STIME
      CALL RTIME
:ESKP
C     -----------------------------------------
C     * ENVIAR MENSAJE DE INICIO DEL PROGRAMA *
C     -----------------------------------------
      REWIND 3
      WRITE(3,*) "                              I N I C I A    T I P S "
:SKEZ 1
      WRITE(6,*) "I N I C I A   P R O G R A M A   P R I N C I P A L"
:ESKP
      CALL TIPMSG (1, 0, 0, 0D, 0D)
C     -------------------------------
C     * ABRIR ARCHIVOS ESTRUCTURADOS *
C     -------------------------------
      ERR = .FALSE.
      J = 1
      WHILE (.NOT.ERR.AND.J.LE.NAEUSZ)
         CODCOM = 0
:SKEZ 2
         WRITE(6,*) "ANTES DE ABRIR ARCHIVOS ESTRUCTURADOS"
         WRITE(6,*) "J:",J,"LISAF:",LISAF(J),"LLAVE:",LLAVES(J)
         WRITE(6,*) "CODCOM:",CODCOM
:ESKP
         CALL ABREAE (LISAF(J),LLAVES(J),CODCOM)
:SKEZ 2
         WRITE(6,*) "DESPUES DE ABRIR ARCHIVOS ESTRUCTURADOS"
         WRITE(6,*) "J:",J,"LISAF:",LISAF(J),"LLAVE:",LLAVES(J)
         WRITE(6,*) "CODCOM:",CODCOM
:ESKP
         IF (CODCOM.EQ.0) THEN
            J = J+1
         ELSE
            IF (CODCOM.EQ.3) THEN
:AS
                 TOK    12
                 BLU    SDELAY
:EA
            ELSE
               ERR = .TRUE.
            END IF
         END IF
      END WHILE
C
      IF (.NOT.ERR) THEN
:SKEZ 2
         WRITE (6,*) "NO HUBO ERROR EN APERTURA DE AE "
:ESKP
C        ----------------------------------
C        * APERTURA DE ARCHIVOS CORRECTA *
C        ----------------------------------
```

```
              FLSE
                 ERR = .TRUE.
:SKF7 2
                 WRITE (6,*) "ERROR EN CLAUSURA ",LLAVES(J)
   SP
              END IF
           END IF
        END WHILE
C       ----------------------------------------
C       * VERIFICAR EL CIERRE DE LOS ARCHIVOS *
C       ----------------------------------------
        IF (ERR) THEN
           CALL TIPMSG (18 , LISAE(J) , 0 , 0D , 0D )
        END IF
C       ------------------------------------------------
C       * ENVIAR MENSAJE DE TERMINACION DEL PROGRAMA *
C       ------------------------------------------------
        CALL TIPMSG ( 7 , 0 , 0 , 0D , 0D )
        REWIND 3
        WRITE(3,*) "      T E R M I N A C I O N   N O R M A L      T I P S"
C        -------------------
C        * AYUDAS DE DEBUG *
C        -------------------
:SKF7 1
        WRITE(6,*) "T E R M I N A   P R O G R A M A   P R I N C I P A L"
:ESKP
:SKF7 5
        WRITE(6,*) "          P R O G R A M A   T I P S"
        WRITE(6,*)
        CALL ETIME2(CPUT)
        CALL ATIME2(ELAT)
        WRITE(6,5000) CPUT,ELAT
:E
        WRITE (3,*) "ELAT=",ELAT
5000    FORMAT(5X,"TIEMPO DE CPU =",I15,"MSEGS.",/,5X,
     >           "TIEMPO TRANSCURRIDO =",I8," SEGS.")
C        ---------------------
C        * TERMINA COMPONENTE *
C        ---------------------
         CALL EXIT
         END
```

```
0010 REM "01IIZZO" "CARGA DE POLIZAS"
0012 IF M1=1 THEN GOTO 0560
0013 IF O0=2 THEN GOTO 0540
0014 IF O0=1 THEN GOTO 0660
0021 BEGIN
     LET X$=FID(0); OPEN (1)"0"; READ (1,IND=NUM(X$(2)))O$,V$,Q$,*,Q1$,Q2$,*,*,
0023 *,*,*,Q3$,Q4$,Q5$; CLOSE (1)
0030 GOSUB 0260
0035 DIM U$(74,"="),C$(17),U1$(5," "),U2$(8,"0"),Y$(40,10E9),D$(45),G6$(48),U0$
0070  5," ")
0073 PRINT 'CS'
0075 INPUT (0,ERR=0075,SIZ=1)@(10,12),"TECLEE # DE MES DE TRABAJO O CR ",C$(1,1
0075 ),(""=0000,LEN=1,1)
0080 LET J$="123456789ABC"; IF POS(C$(1,1)=J$)=0 THEN GOTO 0075
0090 OPEN (1)Q1$; OPEN (2)Q2$; OPEN (6)Q2$
0180 IF O$<>" 2" THEN GOTO 0240
0190 OPEN (3)Q3$; OPEN (4)Q4$; OPEN (5)Q5$
0240 PRINT 'CS','SB',@(0,0),V$,@(55,0),"ENTRADA A POLIZAS",@(0,2),"FECHA",@(0
0240 ,3),"POLIZA"
0250 PRINT @(0,4),U$,@(0,5),"DSC AFECTACION REFER  SUBREF",@(36),"CONCEPTO",@(6
0250 3),"TM",@(67),"IMPORTE",@(0,6),U$,@(0,20),U$
0255 LET N$="#"###,###.00-"
0256 GOTO 0540
0260 INPUT 'CS',@(10,10),"CARGA DE POLIZAS",@(10,12),"DESEA BORRAR LOS ARCHIVOS
0260  ANTERIORES ? (SI/CR) ",Y0$,@(10,12),'CL'
0270 IF Y0$<>"SI" THEN RETURN
0280 PRINT @(10,12),'PB',"RECUERDE ANTES DE BORRAR LOS ARCHIVOS",@(20,13),"DEB
0280 EN SER RESPALDADOS"; WAIT 2
0285 INPUT (0,ERR=0285)@(10,15),"DESEA REALIZAR RESPALDO DE ARCHIVOS ? (NO/CR)
0285 ",Y0$,("NO"=0295,""=0287)
     RUN "240..."
0290 RETURN
0295 PRINT @(0,12),'ES'
0300 OPEN (2)Q1$; LET U$=FID(2); CLOSE (2)
0310 OPEN (6)Q2$; LET U1$=FID(6); CLOSE (6)
0320 ERASE Q1$
0330 ERASE Q2$
0340 FILE U$
0350 FILE U1$
0400 PRINT @(0,14),'CL'; RETURN
0540 LET L=0,O=0
0550 INPUT (0,ERR=0550,SIZ=6)'CF','SF',@(6,2),"DDMMAA",@(6,2),K$,("FIN"=0690,""
0550 =0570,LEN=6,6)
0560 LET D$(1,6)=K$
0570 IF D$(1,6)="      " AND O... THEN GOTO 0550
0575 PRINT @(6,2),D$(1,6)
0580 LET M1=0
0590 PRINT @(7,3),C$(1,1)
0600 LET O9=O9+1
0610 LET P$=STR(O9)
0620 LET P$=U1$(1,5-LEN(P$))+P$,C$(2,5)=P$
0630 PRINT @(7,3),C$(2,5)
0640 LET ...
0650 RUN "01...1"
0660 INPUT @(7,3),O9
     LET O0=0
     GOTO 0...
0750 RUN "......0"
```

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

TECNICAS MODERNAS DE DESARROLLO
Y
ADMINISTRACION DE PROGRAMACION

PROBLEMAS EN LA PROGRAMACION

MARZO, 1983

SOFTWARE DIGEST

[illegible] LITTLE RIVER TURNPIKE, ANNANDALE, VA. 22[illegible] PHONE (703) 354 [illegible] TWX 710 [illegible] TELEX [illegible]

Vol. 12 No. 16 April 24, 1980 Page 1

SOFTWARE PREPARATION COSTS WILL INCREASE FROM THE PRESENT $40 PER LINE TO $65 per line by 1984, according to predictions by the Department of Defense. Thus, software preparation will represent eight per cent of the total U.S. defense budget – expected to rise from $6.6 billion in 1979 to $10.5 billion in 1984. ▫

PROBLEMAS EN LA PROGRAMACION

** SOBRE COSTOS

** RETRASO EN LA ENTREGA

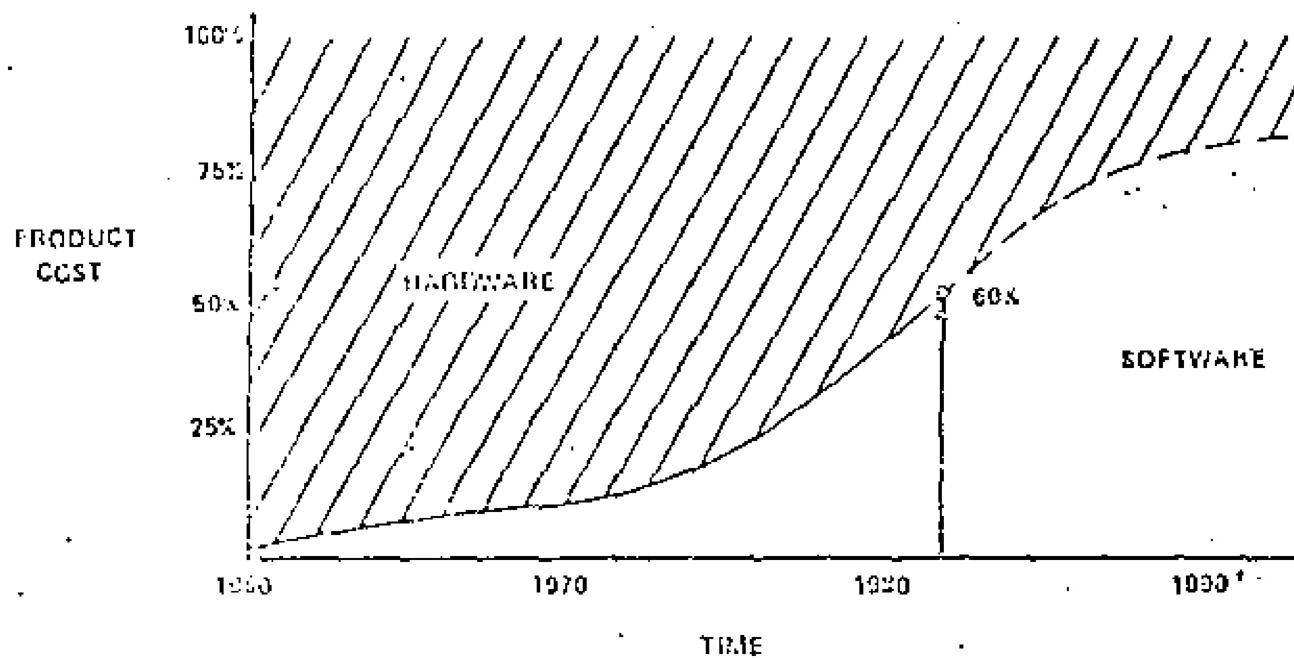
** DIFICULTAD DE MANTENIMIENTO

** BAJA CONFIABILIDAD

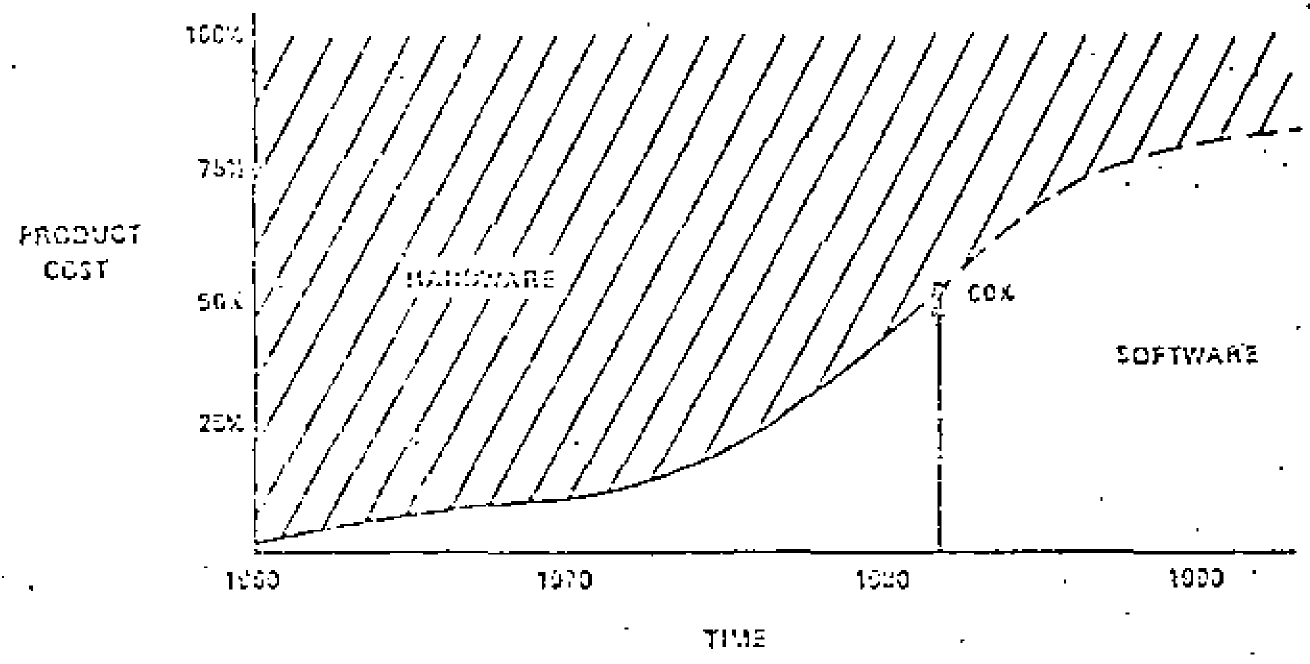
** INCONFORMIDAD DEL USUARIO

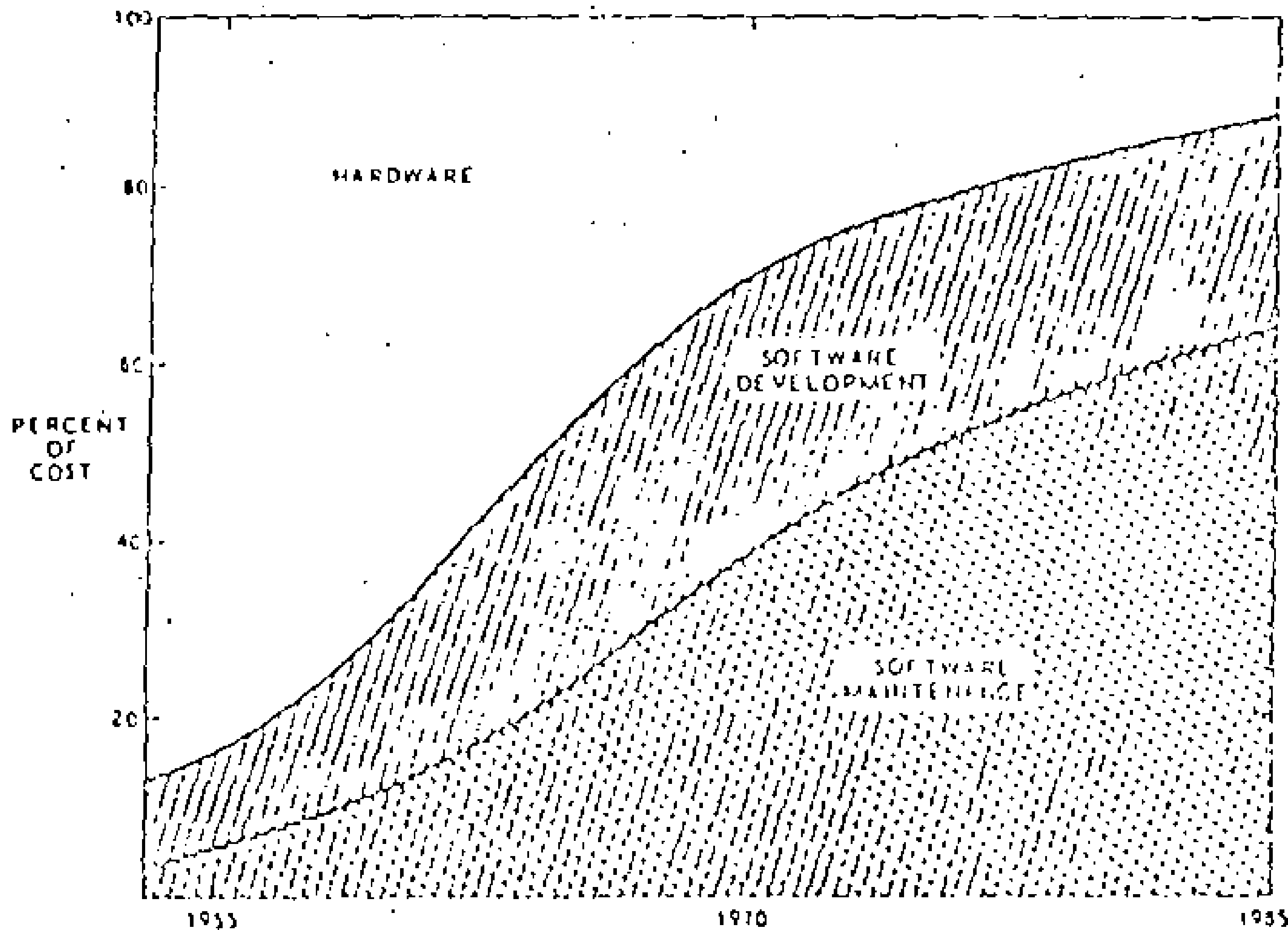
** INTEGRACION

NEED — GROWING SOFTWARE COST

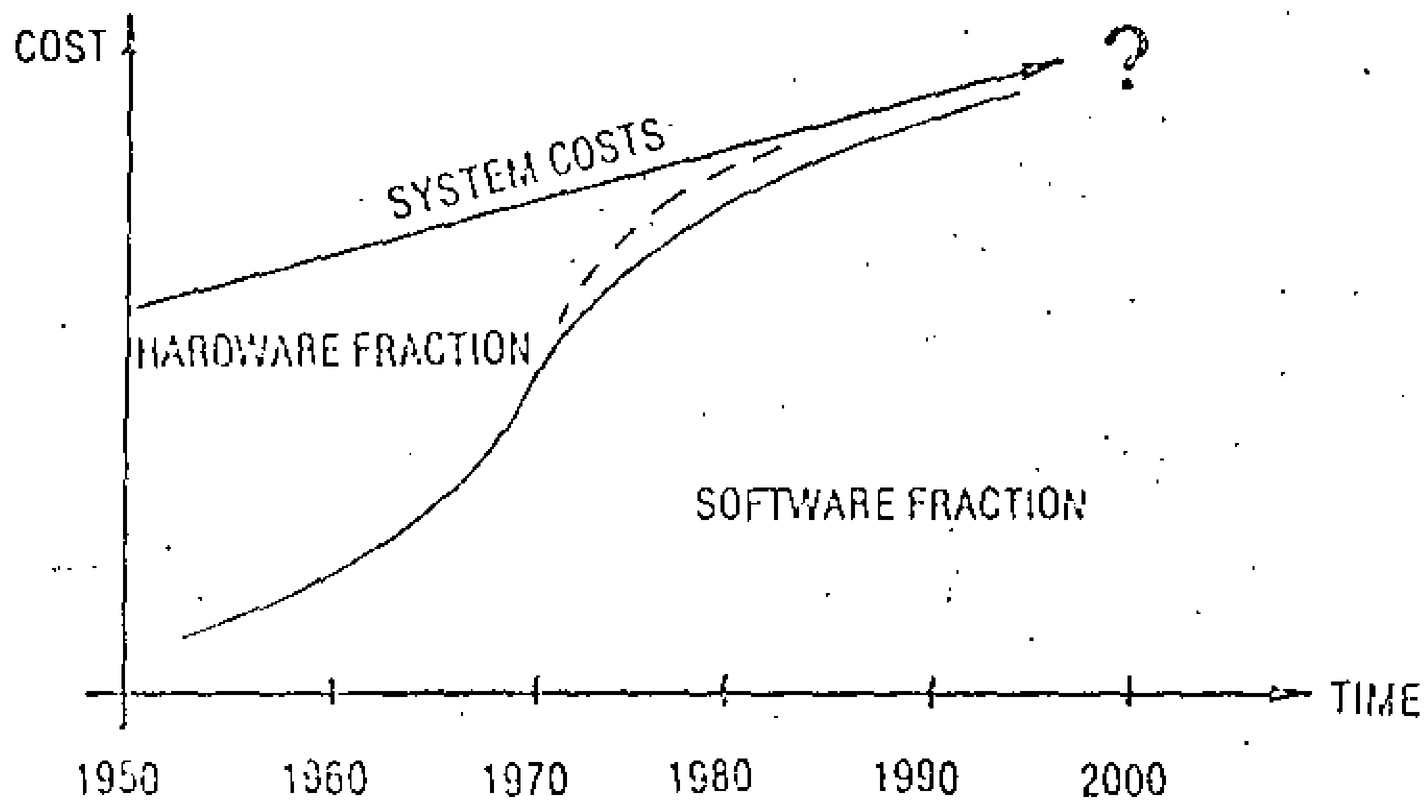


NEED — GROWING SOFTWARE COST

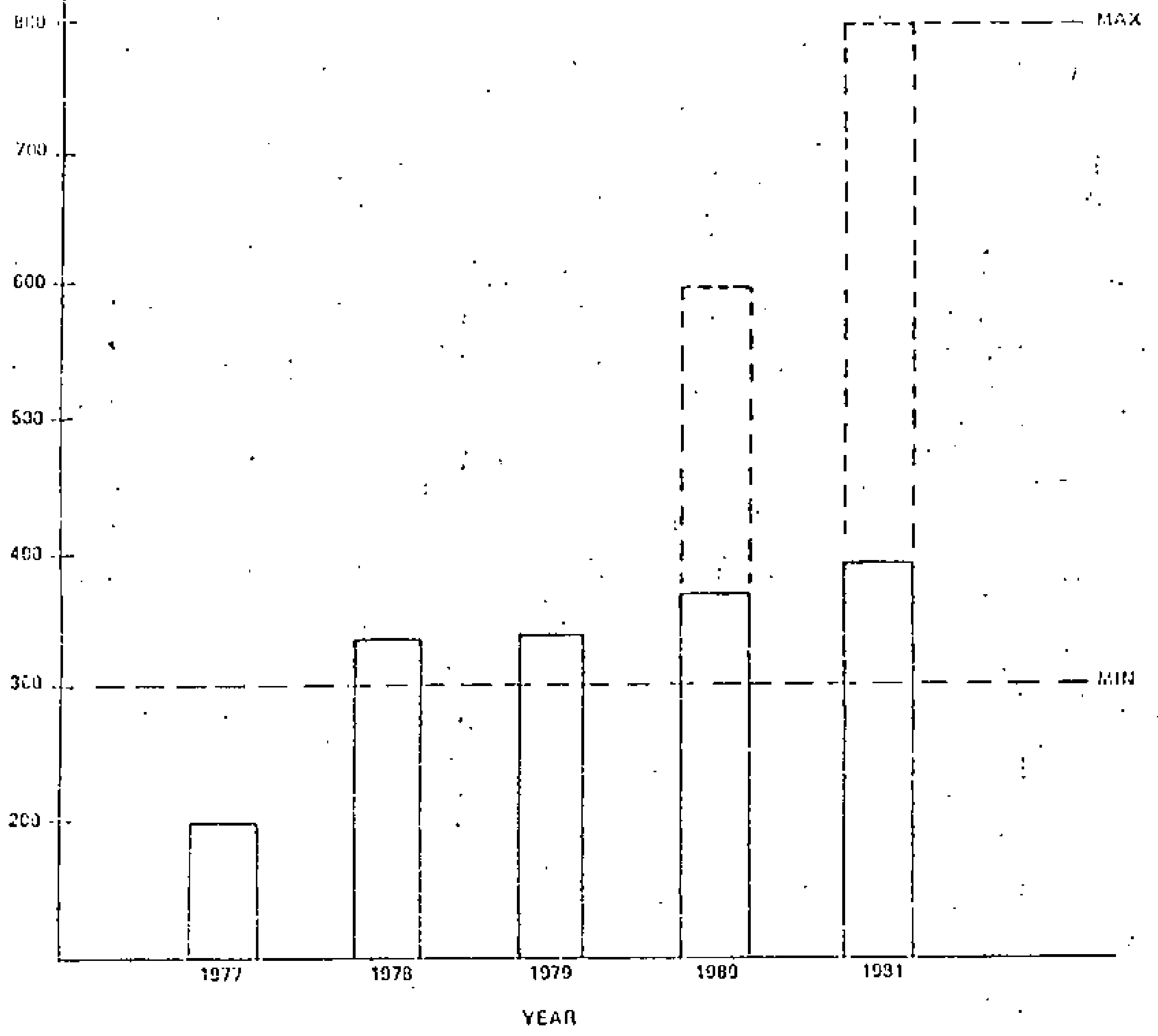




Hardware-software cost trends.



Trends in hardware and software system costs.



800
700
600
500
400
300
200
MAX
MIN
1977
1978
1979
1980
1981
YEAR

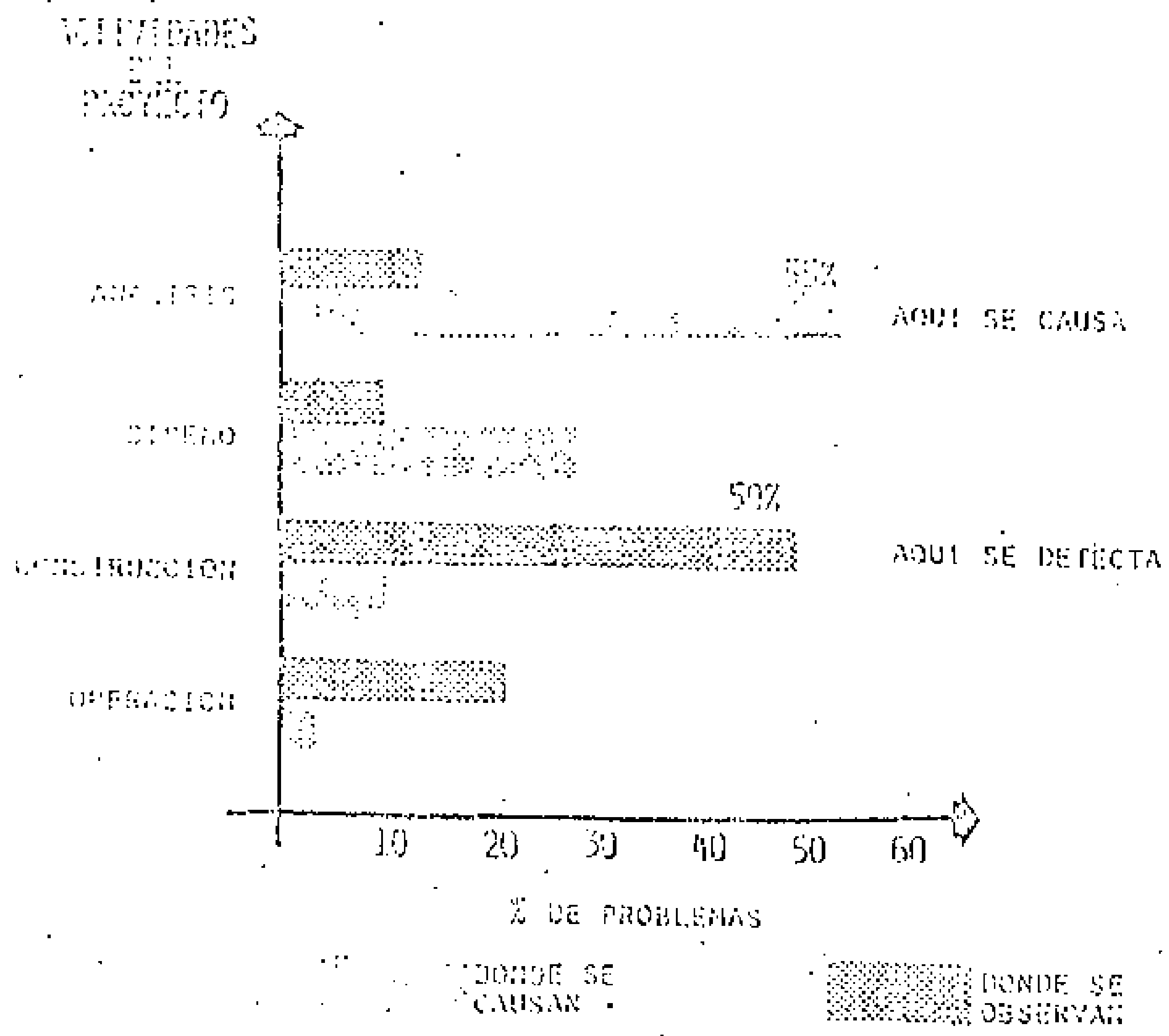


FIG. 1.2.1 ORIGEN DEL ERROR Y LUGAR DE SU DETECCION

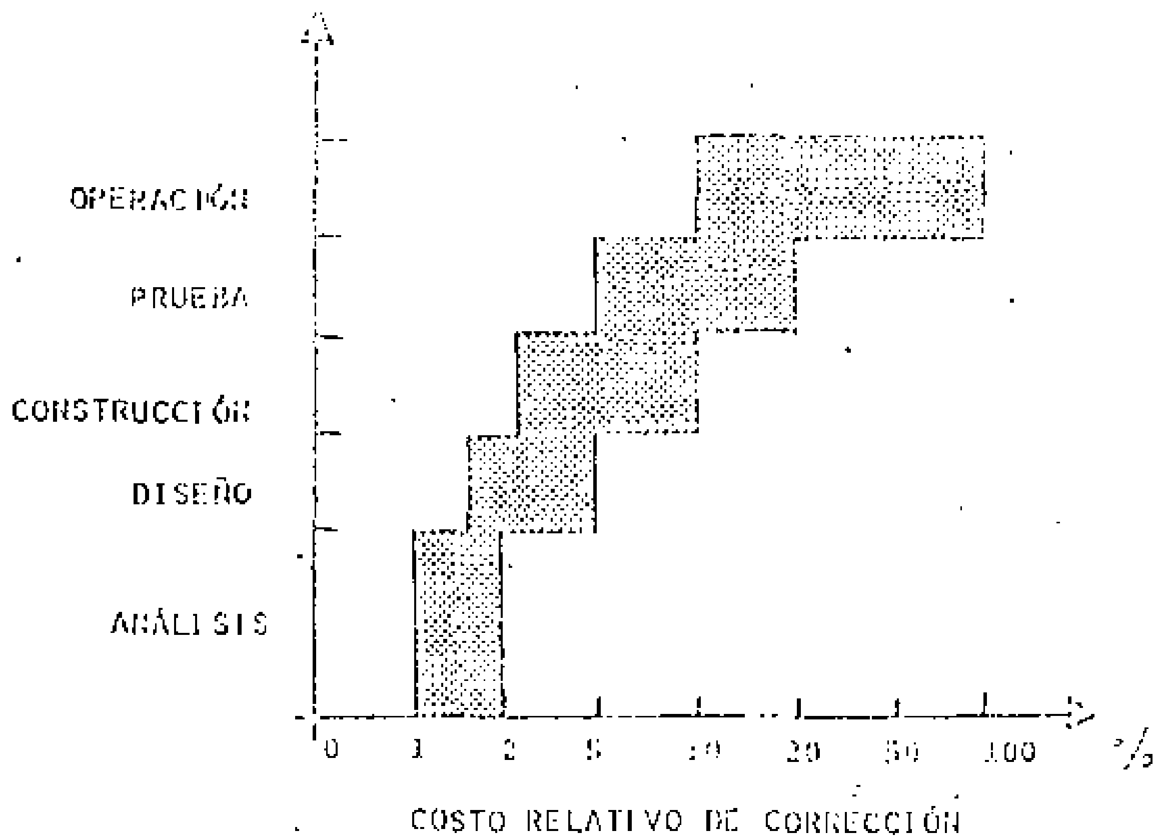
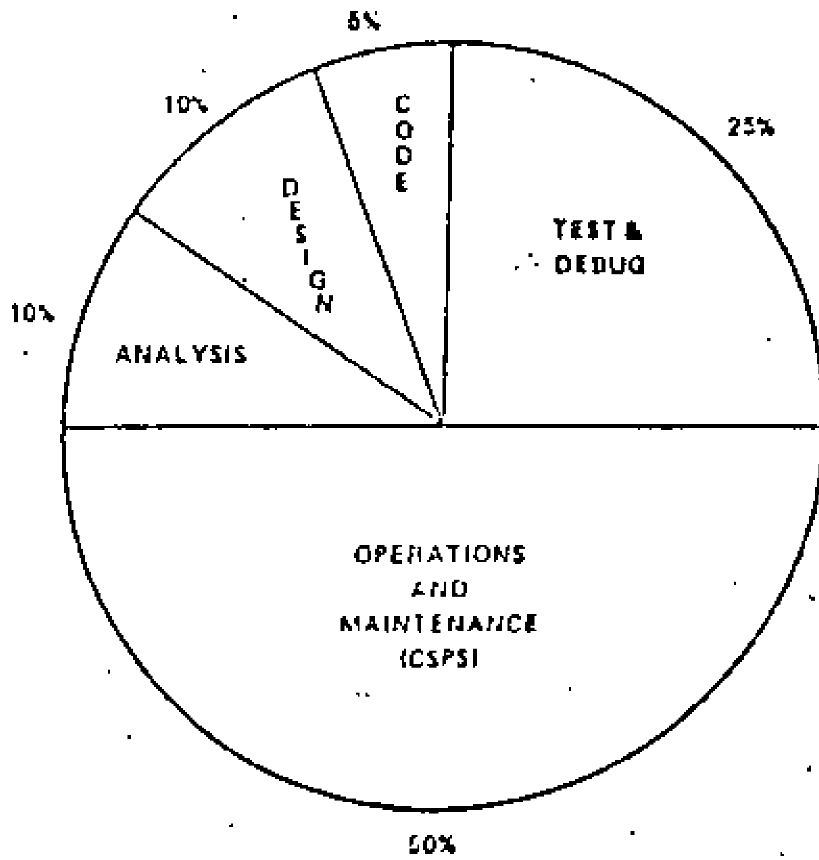
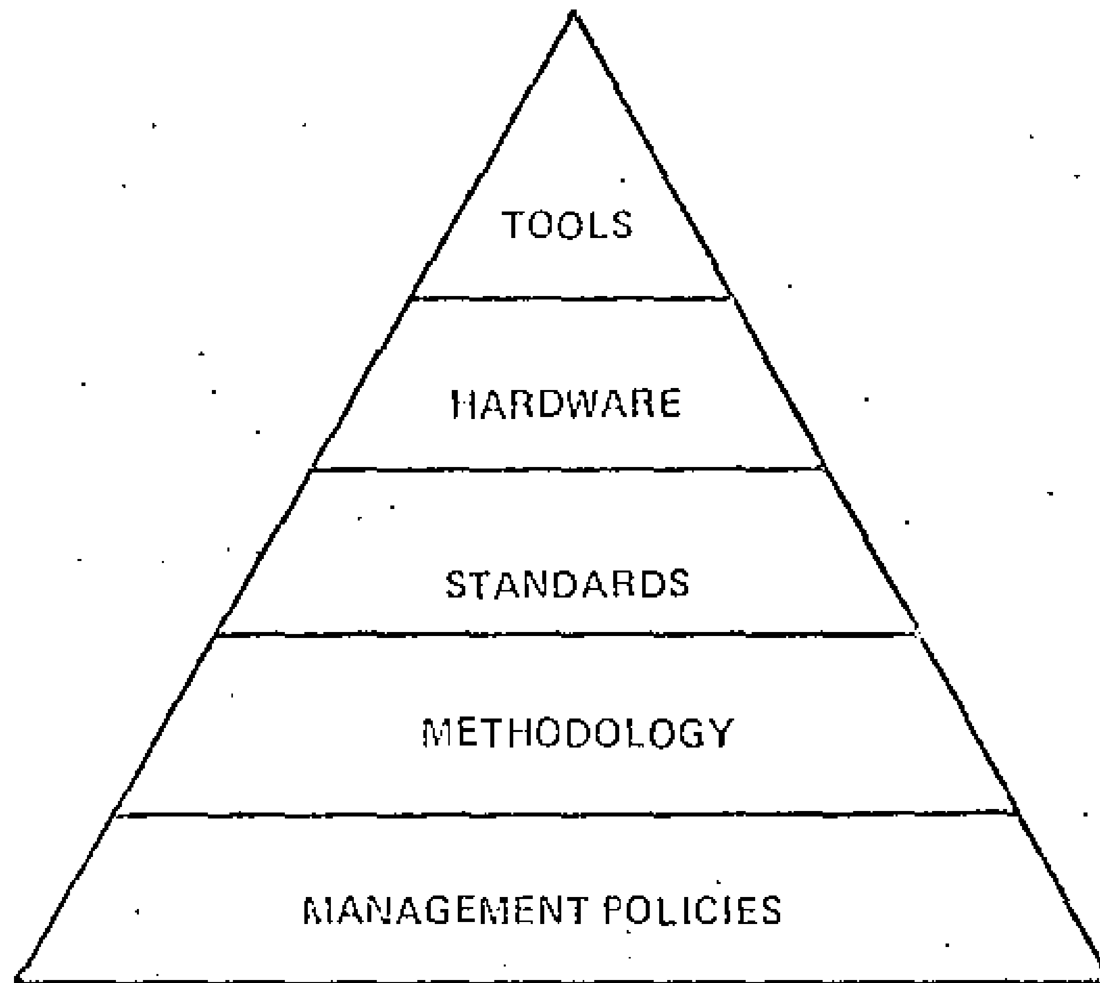


FIG. 1.2.2 COSTO DE CORRECCIÓN DE ERRORES

TYPICAL COST DISTRIBUTION





ELEMENTS WHICH CAN BE COMBINED TO REDUCE SOFTWARE COSTS

OBJETIVOS

** EL PROGRAMA DEBE DE TRABAJAR

** BAJOS COSTOS DE DESARROLLO

** BAJOS COSTOS DE PRUEBA

** COSTOS MINIMOS DE MANTENIMIENTO

** FACILMENTE MODIFICABLE

** DE DISEÑO POCO COMPLICADO

** EFICIENTE

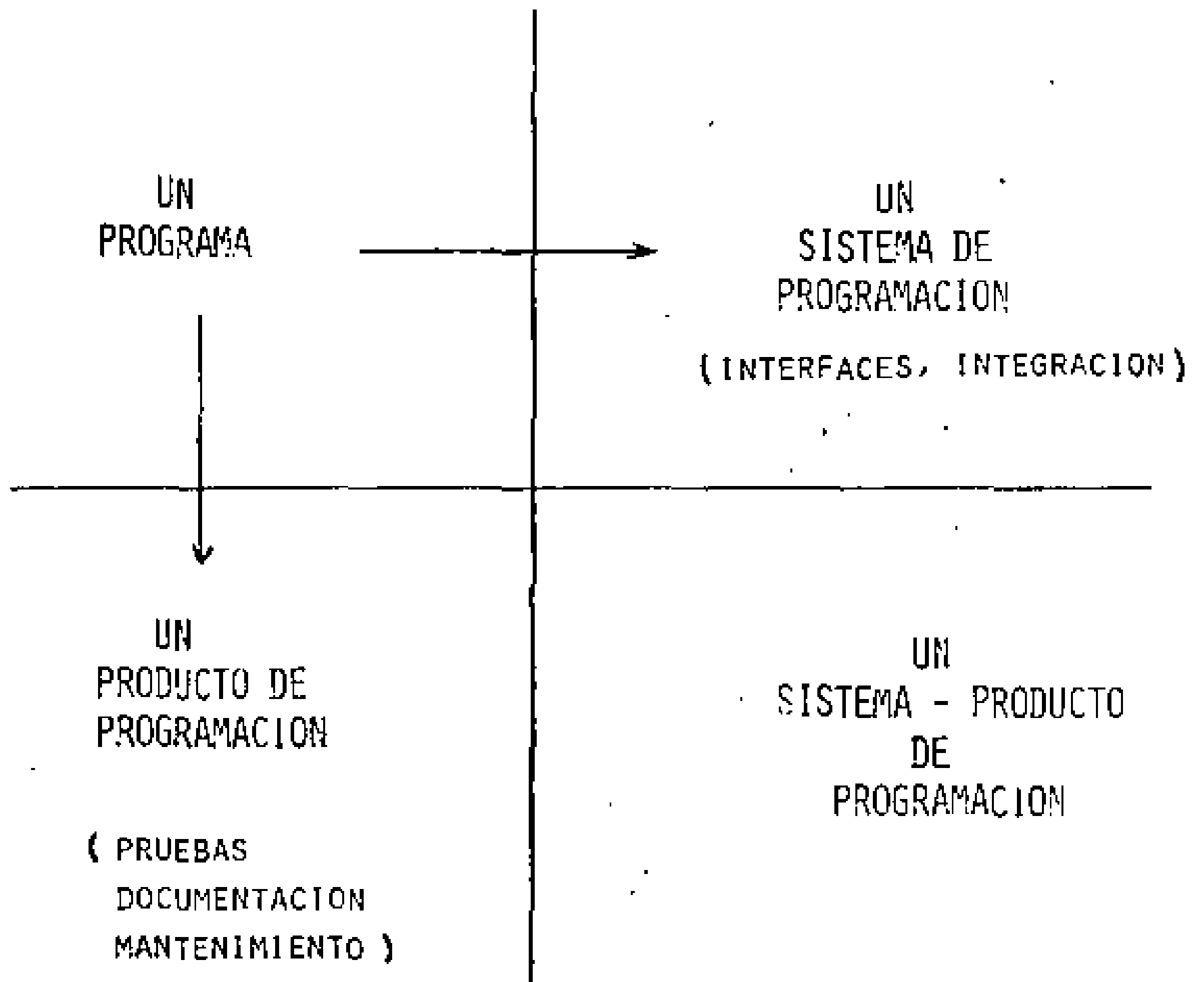
POLITICAS DE ADMINISTRACION

A D M I N I S T R A C I O N

** ORGANIZAR EL EMPLEO DE LOS RECURSOS PARA ALCANZAR UN OBJETIVO

** PLANEAR, ORGANIZAR Y CONTROLAR

PROYECTO DE PROGRAMACION



MODELO DEL ENFOQUE SISTEMICO EN TERMINOS

DE

E N T R O P I A

** TRANSFORMAR PROGRAMAS EN PRODUCTOS

** TRANSFORMAR PROGRAMAS EN SISTEMAS

** TRANSFORMAR PROGRAMAS EN SISTEMAS - PRODUCTO

C A U S A

E N T R O P I A

$$E1 = W + S$$

E1 = ENERGIA DE ENTRADA

W = PRODUCTOS GENERADOS

S = ENTROPIA

EXISTEN 2 CLASES DE ENTROPIA

S 1 = NO CONTROLABLES

EJEMPLO:

- VACACIONES
- ENFERMEDAD
- MORAL
- LLAMADAS TELEFONICAS
- ETC:

S 2 = CONTROLABLES

** FALTA DE COOPERACION
** INCOHERENCIA
** CONFUSION
** FALTA DE DIRECCION

COMPLEJIDAD DE LOS PROYECTOS

COSTO DE MANTENIMIENTO 40 - 70% DEL COSTO TOTAL DEL PROYECTO

MANTENIBILIDAD

- COMPLEJIDAD DE LOS PROGRAMAS
- COMPRENSIBILIDAD
- MODIFICABILIDAD
- FACILIDAD PARA PROBARSE

COMPLEJIDAD DE UN PROGRAMA

- TAMAÑO DEL PROGRAMA
- ESTRUCTURA DE DATOS
- FLUJO DE DATOS
- FLUJO DE CONTROL

COMPLEJIDAD DE UN PROYECTO EN FUNCION DEL NUMERO DE LINEAS DE CODIGO

LINEAS DE CODIGO	COMPLEJIDAD
A) HASTA 1000	TRIVIAL
B) ENTRE 1,000 Y 10,000	SIMPLE
C) ENTRE 10,000 Y 100,000	DIFICIL
D) ENTRE 100,000 Y 1,000,000	COMPLEJO
E) MAS DE 1,000,000	CASI IMPOSIBLE

SIGUIENDO CON LOS METODOS TRADICIONALES DE DESARROLLO DE PROGRAMACION, SE CORRE EL RIESGO DE :

- NO CONCLUIR EL PROYECTO
- TERMINAR EL PROYECTO MUCHO DESPUES DEL TIEMPO ESTIPULADO Y POR LO TANTO FUERA DEL PRESUPUESTO
- QUE EL USUARIO RECHACE EL PRODUCTO FINAL POR NO CUBRIR SUS NECESIDADES.

PLANEACION

ANALISIS DE PRODUCTIVIDAD

1. ESTABLECIMIENTO DE OBJETIVOS

2. SUBDIVISION DEL PROYECTO EN SUBPROYECTO

   ARQUITECTURA

3. ESTABLECIMIENTO DE LOS NIVELES CUANTITATIVOS DEL PRODUCTO O SUBPRODUCTO

   - LINEAS DEL CODIGO
   - PAGINAS DE DOCUMENTACION

4. ESTABLECIMIENTO DE LA PRODUCTIVIDAD DEL PERSONAL

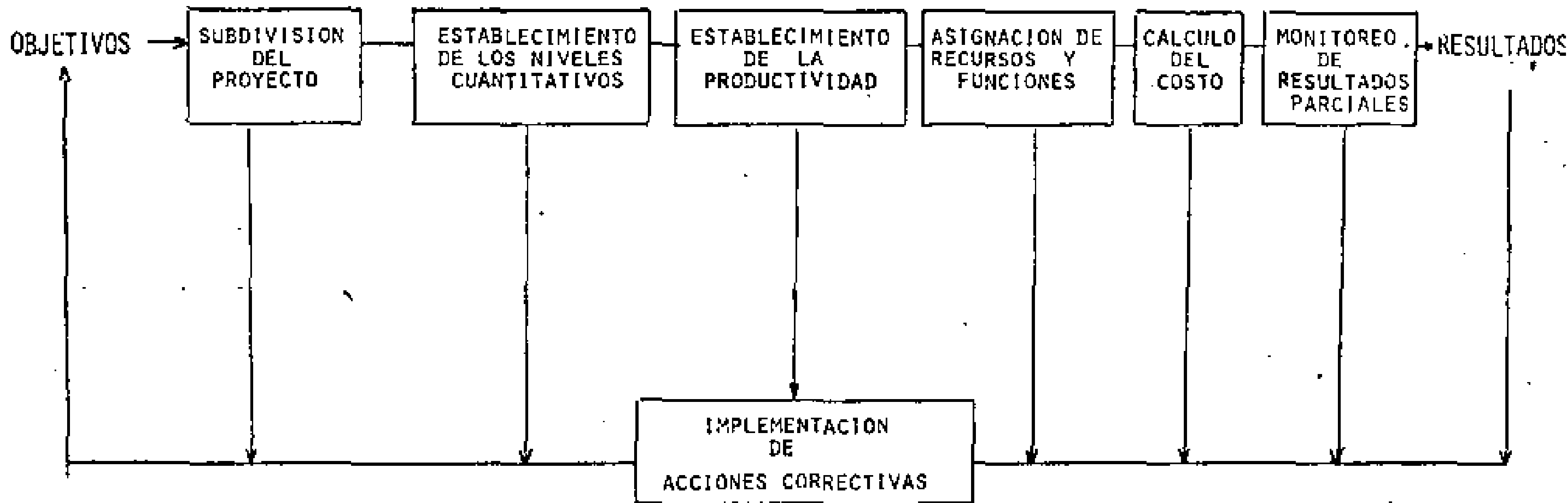
   - LINEAS DE CODIGO POR MES-HOMBRE
   - PAGINAS DE DOCUMENTACION POR MES-HOMBRE

5. ASIGNACION DE RECURSOS Y FUNCIONES

6. CALCULO DEL COSTO

7. MONITOREO DE RESULTADOS PARCIALES

8. IMPLEMENTACION DE ACCIONES CORRECTIVAS


OBJETIVOS
SUBDIVISION DEL PROYECTO
ESTABLECIMIENTO DE LOS NIVELES CUANTITATIVOS
ESTABLECIMIENTO DE LA PRODUCTIVIDAD
ASIGNACION DE RECURSOS Y FUNCIONES
CALCULO DEL COSTO
MONITOREO DE RESULTADOS PARCIALES
RESULTADOS
IMPLEMENTACION DE ACCIONES CORRECTIVAS

O R G A N I Z A C I O N

ORGANIZACION INFORMAL

- PROYECTOS DE 10,000 LINEAS
- 4 A 8 PERSONAS
- JEFE DE PROYECTO
- PROGRAMADOR LIDER
- BIBLIOTECARIO
- PROGRAMADORES

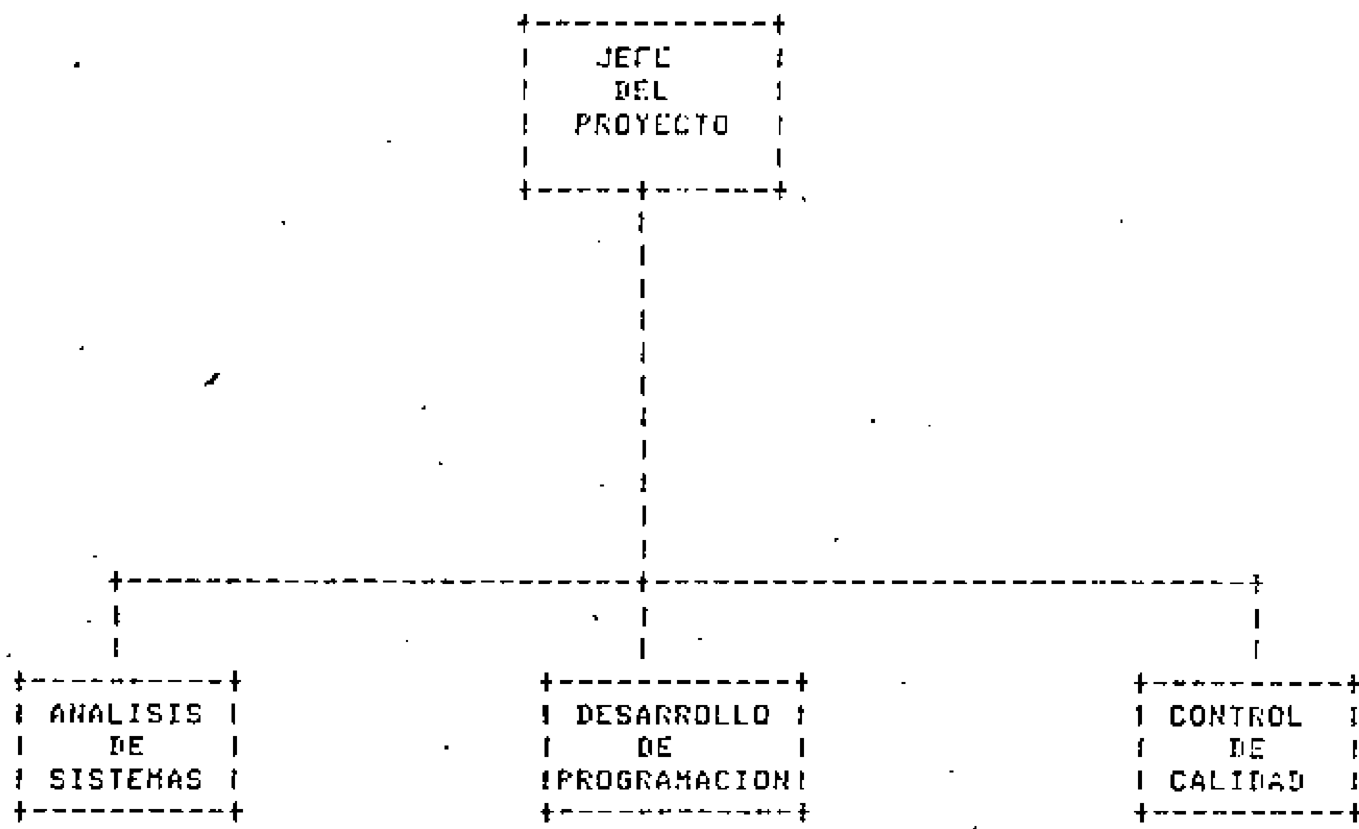
ORGANIZACION FORMAL

** CONSERVADORA

- ANALISIS DE SISTEMAS
- DESARROLLO DE PROGRAMACIÓN
- CONTROL DE CALIDAD

** RADICAL

- PROGRAMA DE CÁLCULO AUTOMÁTICO
- DESPACHO ECONÓMICO RESTRINGIDO
- PRONOSTICO DE CARGA
- COORDINACION HIDROTÉRMICA



Actividades:

1) Documento de objetivos (DO)
2) Requerimientos de programas de computadora (RPC)
3) Planes de prueba (PLP)
4) Manual del usuario (MUS)

1) Arquitectura de programas de computadora (ARQ)
2) Diseño de programa de computadora (DPC)
3) Codificación (COD)
4) Integración
5) Procedimientos de (PRP)

1) Normas
2) Biblioteca de soporte de programas
3) Auditoría control

Fig. 5.3.2. ORGANIZACION BASICA DE UN GRUPO DE DESARROLLO DE PROGRAMAS.

ESTIMACION DE COSTOS

- COMUNMENTE UN ARTE IMPRECISO

- EXTRAPOLACIONES DE PROYECTOS SIMILARES

- MALAS EXPERIENCIAS

ALGUNOS FACTORES A CONSIDERAR

* TAMAÑO DEL PROGRAMA
* ATRIBUTOS DEL PROGRAMA
* CANTIDAD DE CODIGO ORIGINAL
* LIMITACIONES DE EQUIPO
* LIMITACIONES EN EL TIEMPO DE RESPUESTA
* LA DEFINICION COMPLETA DE LOS REQUERIMIENTOS
* MEDIO AMBIENTE

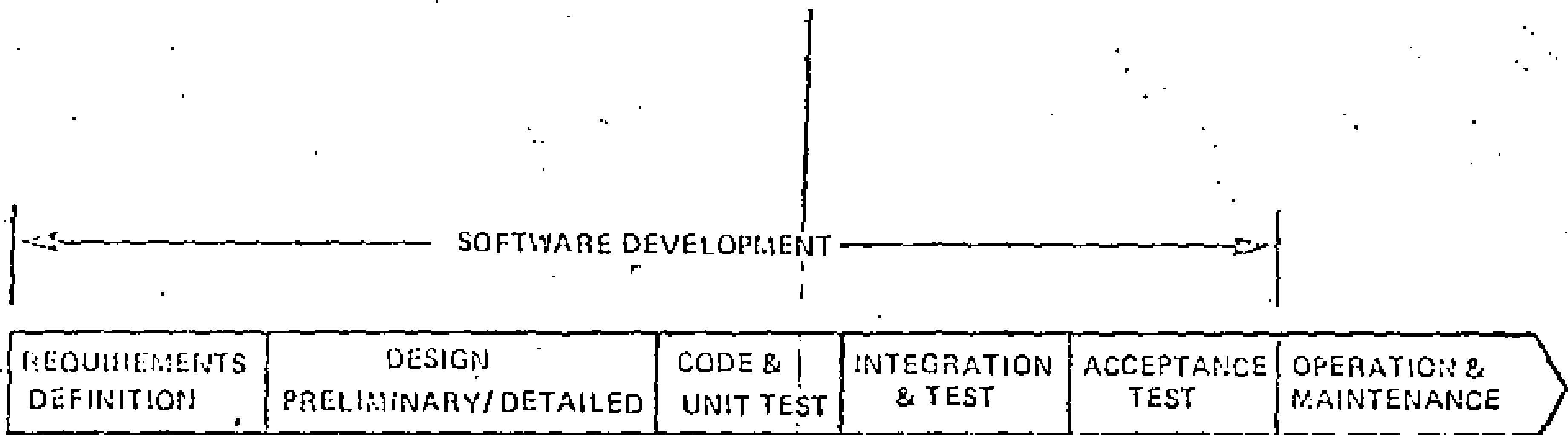
REVISIONES Y AUDITORIAS

COMO SE SEÑALO CON ANTERIORIDAD EL NUMERO DE ERRORES QUE SE COMETEN EN LA FASE DE ANALISIS ES ELEVADO Y SU DETECCION EN FASES POSTERIORES DEL PROYECTO ES MUY COSTOSO.

RESULTA MUY CONVENIENTE DURANTE LA VIDA DEL PROYECTO CALENDARIZAR UNA SERIE DE REVISIONES, A FIN DE SEÑALAR ERRORES EN LAS ETAPAS JOVENES DEL PROYECTO .

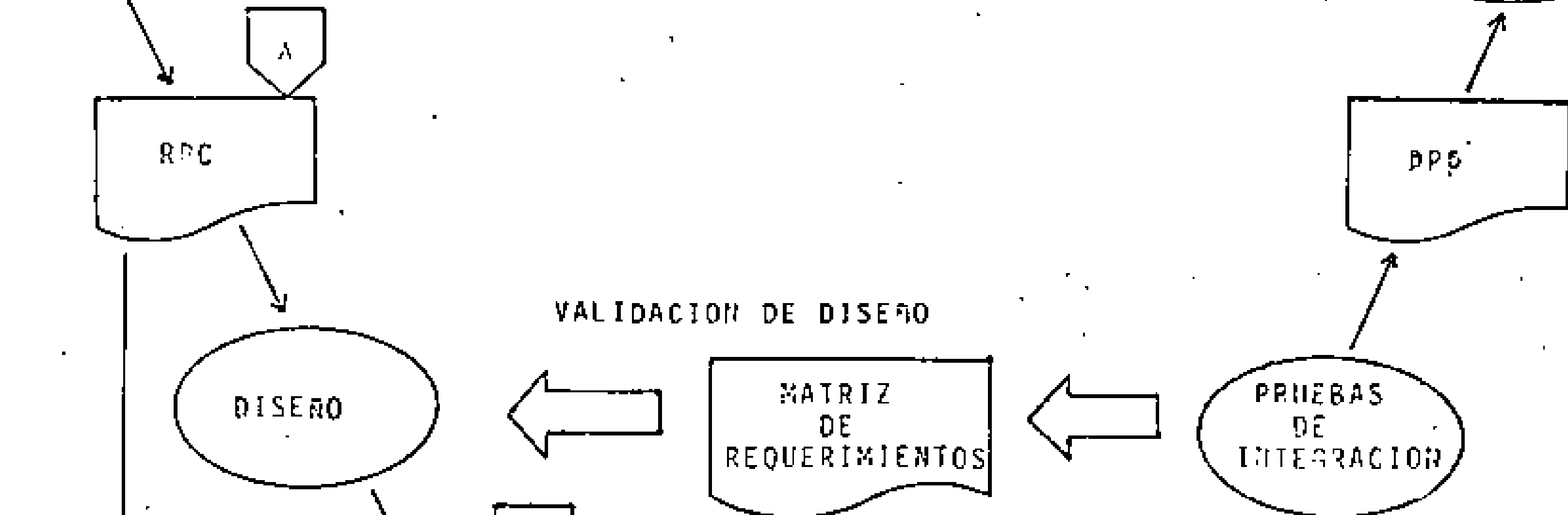
ESTAS REVISIONES PUEDEN SER DE DISTINTOS GRADOS, DESDE PLATICAS ESTRUCTURADAS HASTA HERRAMIENTAS AUTOMATICAS, POR EJEMPLO .

M E T O D O L O G I A

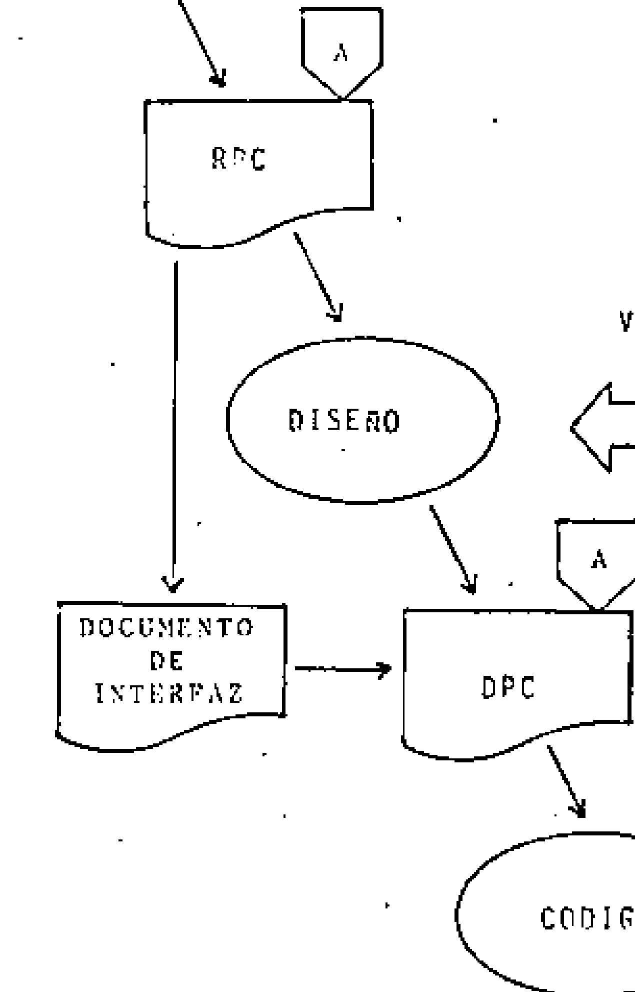


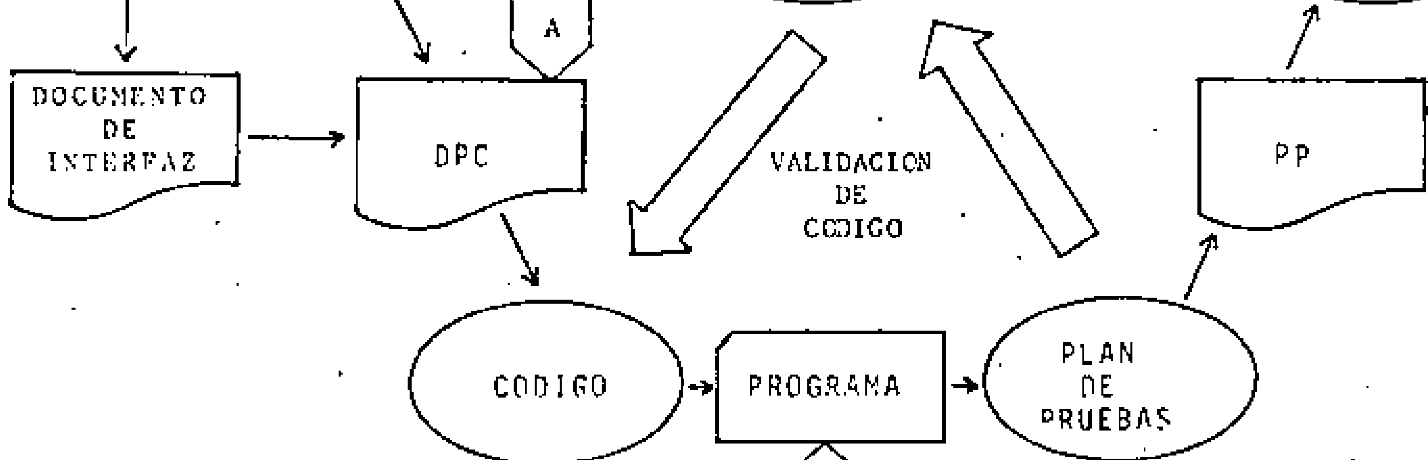
SOFTWARE LIFE CYCLE

VALIDACION DE REQUERIMIENTOS
NIVEL
DE
ABSTRAC-
CION
ANALISIS
MATRIZ
DE
REQUERIMIENTOS
PRUEBAS
DE
ACEPTACION

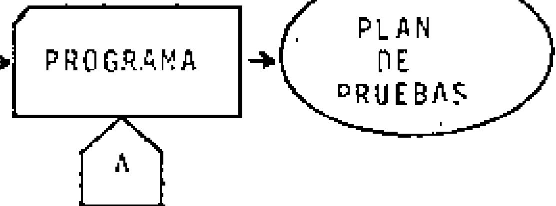

A
RPC
VALIDACION DE DISEÑO
DISEÑO
MATRIZ
DE
REQUERIMIENTOS
PRUEBAS
DE
INTEGRACION

PPP




DOCUMENTO
DE
INTERFAZ
A
DPC
VALIDACION
DE
CODIGO
CODIGO
PROGRAMA
PLAN
DE
PRUEBAS

PP


A

TIEMPO

T E C N I C A S

D E

P R O D U C T I V I D A D

TECNICAS DE PRODUCTIVIDAD PARA EL DESARROLLO DE PROGRAMACION

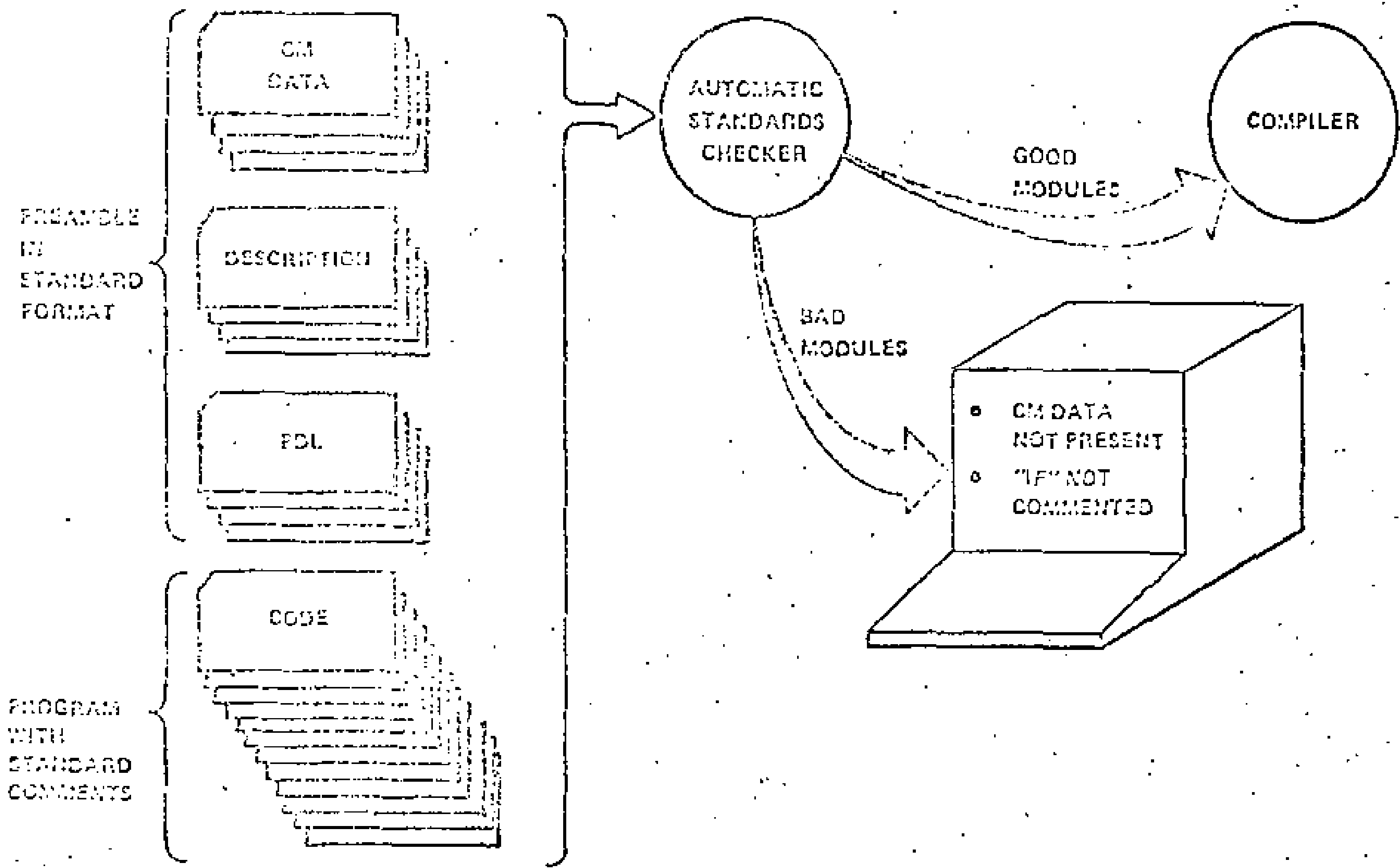
- ANALISIS ESTRUCTURADO
- DISEÑO ESTRUCTURADO
- PROGRAMACION ESTRUCTURADA
- EQUIPOS DE TRABAJO
- REVISION ESTRUCTURADA
- RUTA CRITICA
- HERRAMIENTAS AUTOMATICAS

HERRAMIENTAS

AUTOMATICAS

PARA LLEVAR A CABO LA FUNCION DE CONTROL DE CALIDAD HACEN FALTA HERRAMIENTAS AUTOMATICAS QUE NO SOLAMENTE FACILITEN LA LABOR DESCRITA, SINO QUE ASEGUREN UNA UNIFORMIDAD ENTRE LOS DISTINTOS PROGRAMAS DESARROLLADOS. LAS HERRAMIENTAS AQUI DESCRITAS SE REFIEREN A:

1. AUDITOR DE CODIGO: el cual valida el código escrito por los programadores.
2. PROGRAMA DE CONTROL DE BIBLIOTECA: el cual automatiza la introducción y los cambios efectuados en cada uno de los programas desarrollados.



CM DATA
DESCRIPTION
PDL
CODE
PREAMBLE IN STANDARD FORMAT
PROGRAM WITH STANDARD COMMENTS
AUTOMATIC STANDARDS CHECKER
COMPILER
GOOD MODULES
BAD MODULES
CM DATA NOT PRESENT
"IF" NOT COMMENTED

HERRAMIENTAS AUTOMATICAS

PROBLEMAS

- Incremento en tamaño y complejidad de los sistemas
- Intangibilidad de los programas
- Costos del 50% en validación y pruebas
- Ambigüedad inherente en las especificaciones a cumplir
- Estilo no uniforme

BENEFICIOS

- Mayor volúmen de pruebas comparado con pruebas manuales
- Mayor vigor en el Proceso de Validación
- Mayor aceptación por los participantes
- Reducción del costo y tiempo
- Estilo único

34

EJEMPLOS DE HERRAMIENTAS AUTOMATICAS

BSPG - BIBLIOTECA DE SOPORTE DE PROGRAMAS

AUCO - AUDITOR ESTÁTICO DE CÓDIGO

B S P G

BIBLIOTECA DE SOPORTE DE PROGRAMAS

I. I. E.

36

B S P G

OBJETIVOS

- Catalogación de Módulos reusables
- Administración y Control de Proyecto

B S P G

FUNCIONES

1. INTRODUCCIÓN DE LOS PRODUCTOS DESARROLLADOS POR EL I.I.E. PARA EL PROYECTO DE CONTROL EN TIEMPO REAL DEL CENACE.

   - PROGRAMAS FUENTE
   - LANZADORES

   FORMA DE TRANSMISIÓN DE INFORMACIÓN (FTI)

2. ACTUALIZACIÓN Y REGISTRO DE CAMBIOS DE LOS PRODUCTOS DESARROLLADOS POR EL IIE.

   - PROGRAMAS FUENTE
   - LANZADORES

   FORMA DE ORDEN DE CAMBIO (FOC)

37

INSTITUTO DE INVESTIGACIONES ELECTRICAS

FORMA DE TRANSMISION DE INFORMACION

Fecha de emisión AAMMDD	Fecha de atención AAMMDD
Solicitado por:	Atendida por:
Autorizada por IIE:	Autorizada por CFE:

Referencia Externa

Breve descripción del archivo (1,72 cols.)

Breve descripción del archivo cont. (1,72 cols.)

ASIGNADO POR BSPG

FTI No.	Subsistema:	Tipo de archivo:
	Pronóstico de Carga (PC) Coordinación Hidrotérmica (CH) Despacho Económico (DE) Tabulador de Intercambios (TI) Cálculo de Intercambios (CI)	Programa Fuente (1) Lanzadores (2)

Nombre archivo original	Nombre archivo última versión
B S P G *	B S P G *

FORMA DE ORDEN DE CAMBIO

Fecha de emisión A A M M D D	Fecha de atención A A M M D D
Solicitado por:	Atendida por:
Autorizada por IIE:	Autorizada por CFE:

Referencia Externa

Razón del Cambio:

Mejoras en producto	(MP)
Nuevas especificaciones	(NE)
Error en producto	(EP)

ASIGNADO POR BSPG

FOC No.	Subsistema:	Tipo de archivo:
	Pronóstico de Carga (PC)	Programa Fuente (1)
	Coordinación Hidrotérmica (CH)	Lanzadores (2)
	Despacho Económico (DE)	
	Tabulador de Intercambios (TI)	
	Cálculo de Intercambios (CI)	

Nombre archivo original	Nombre archivo última versión
B S P G X	B S P C W

lphl

B S P G

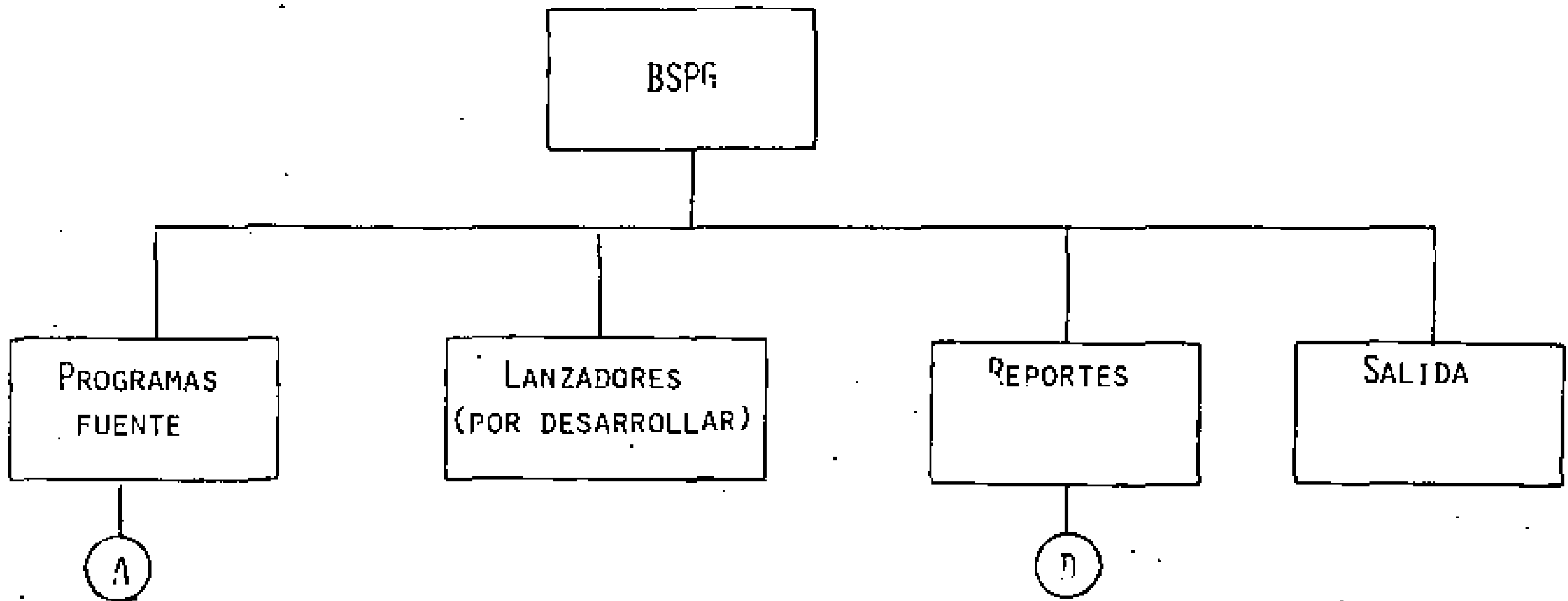
FUNCIONES

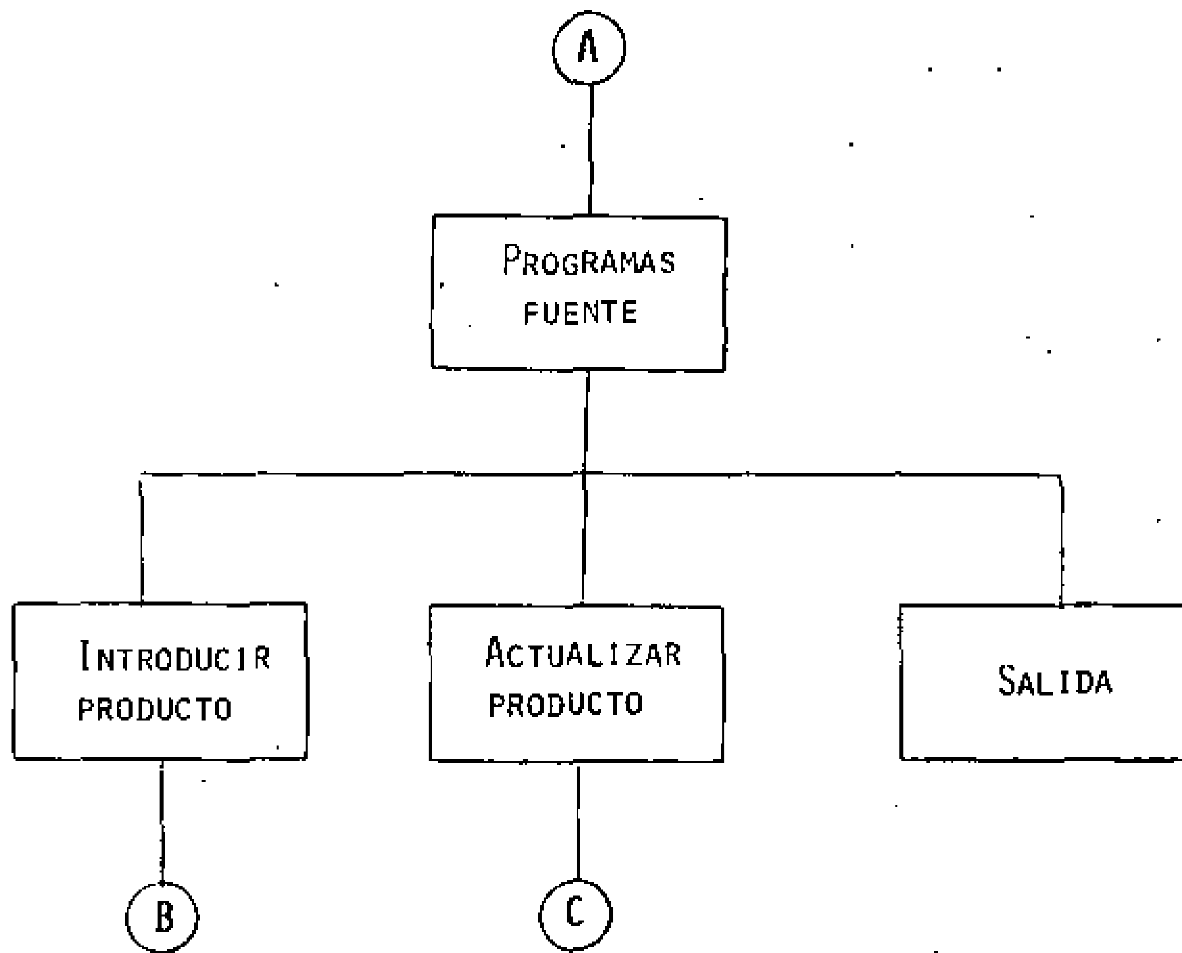
3. REPORTAR EL ESTADO DE LOS PRODUCTOS DESARROLLADOS POR EL I.I.E.

   - PRODUCTOS EXISTENTES EN BIBLIOTECA (FTI)
   - RESUMEN DE LOS CAMBIOS REALIZADOS A CADA PRODUCTO ORIGINAL (FOC o FOCS).

B S P G

ARBOL DE OPCIONES




A
PROGRAMAS FUENTE
INTRODUCIR PRODUCTO
ACTUALIZAR PRODUCTO
SALIDA
B
C

B

INTRODUCIR PRODUCTOS

CAPTURA NOMBRE ARCHIVO A INTRODUCIR

CAPTURA NOMBRE ARCHIVO EN BSPG

CAPTURA INFO. FTI

AUDITACIÓN DE CÓDIGO

COMPILACIÓN

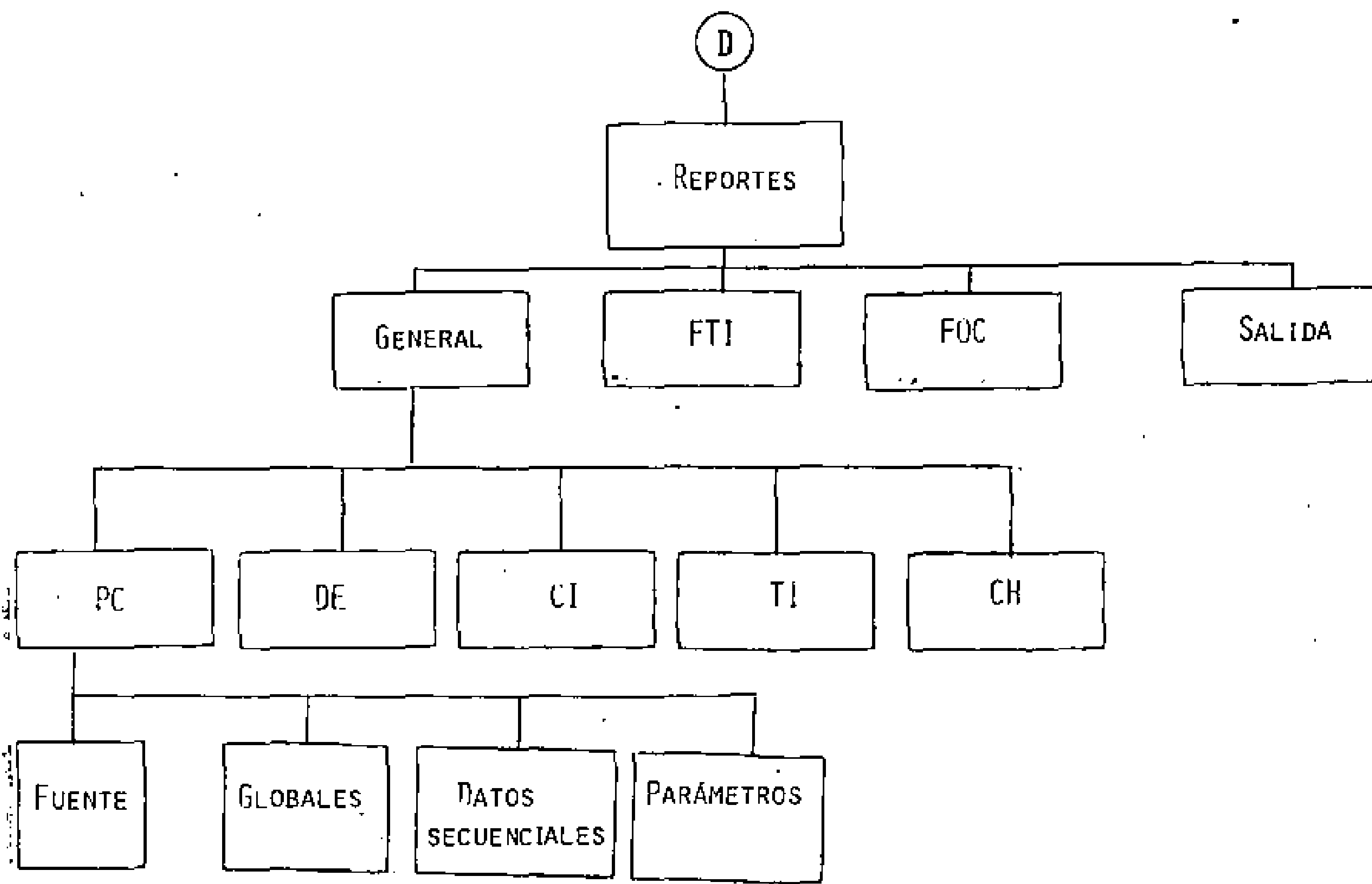
REGISTRAR PRODUCTO EN BSPG

43

C

ACTUALIZAR PRODUCTO

- CAPTURA NOMBRE ARCHIVO A ACTUALIZAR
- CAPTURAR INFO FOC
- CAPTURA CAMBIOS Y ACTUALIZA ARCHIVOS
- AUDITACIÓN DE CÓDIGO
- COMPILACIÓN
- REGISTRA PRODUCTO EN BSPG


D
Reportes
General
FTI
FOC
Salida
PC
DE
CI
TI
CH
Fuente
Globales
Datos secuenciales
Parámetros

FORMA DE TRANSMISION DE INFORMACION

FECHA DE EMISION : 830104

FECHA DE ATENCION : 830105

SOLICITADO POR : VICTOR SEVILLA

ATENDIDO POR : RAFAEL SIERRA

AUTORIZADA POR IIE : MAURICIO MIER

AUTORIZADA POR CFE : ING. CALDERON

REFERENCIA EXTERNA :
MINUTA 123, 12 DICIEMBRE 1982.

DESCRIPCION DEL ARCHIVO :
COMPONENTE LEIPPG
REALIZA LA LECTURA DE LOS INTERCAMBIOS PROGRAMADOS DE ARCHIVOS ESTRUCT.

ASIGNADO POR BSPG

FTI : 1

SUBSISTEMA : IT

TIPO DE ARCHIVO : 1

PRONOSTICO DE CARGA (PC)
COORDINACION HIDROTERMICA (CH)
DESPACHO ECONOMICO (DE)
TABULADOR DE INTERCAMBIOS (IT)
CALCULO DE INTERCAMBIOS (CI)

PROGRAMA FUENTE (1)
LANZADOR (2)

NOMBRE ARCHIVO ORIGINAL	NOMBRE ARCHIVO ULTIMA VERSION
BSPG*S.IIPS03	BSPG*S:IIPS03

```
*---------------------------------------------------------------------------*
*                                                                           *
*            F O R M A   D E   O R D E N   D E   C A M B I O                *
*                                                                           *
*---------------------------------------------------------------------------*
*                                     *                                     *
*   FECHA DE EMISION :                *   FECHA DE ATENCION :               *
*                 830104              *                        830105       *
*                                     *                                     *
*   SOLICITADO POR :                  *   ATENDIDO POR :                    *
*          ADRIAN INDA                *             RAFAEL SIERRA           *
*                                     *                                     *
*   AUTORIZADA POR TIE :              *   AUTORIZADA POR CFE :              *
*           MAURICIO MIER             *             ING. CALDERON           *
*                                                                           *
*   REFERENCIA EXTERNA :                                                    *
*   MINUTA 24.  01 NOVIEMBRE 1982                                           *
*                                                                           *
*   RAZON DEL CAMBIO :                                                      *
*                                              MP                           *
*                                                                           *
*                     MEJORAS EN PRODUCTO         (MP)                      *
*                     NUEVAS ESPECIFICACIONES (NE)                          *
*                     ERROR EN PRODUCTO           (EP)                      *
*                                                                           *
*---------------------------------------------------------------------------*
*                                                                           *
*                     A S I G N A D O   P O R   B S P G                     *
*                                                                           *
*---------------------------------------------------------------------------*
*                                                                           *
*   EOC             SUBSISTEMA :                         TIPO DE ARCHIVO :  *
*      125                                    TT                        1   *
*                                                                           *
*                   PRONOSTICO DE CARGA       (PC)       PROGRAMA FUENTE (1)*
*                   COORDINACION HIDROTERMICA (CH)       LANZADOR        (2)*
*                   DESPACHO ECONOMICO        (DE)                          *
*                   TABULADOR DE INTERCAMBIOS (TT)                          *
*                   CALCULO DE INTERCAMBIOS   (CT)                          *
*                                                                           *
*                                                                           *
*        NOMBRE ARCHIVO ORIGINAL        NOMBRE ARCHIVO ULTIMA VERSION       *
*                                                                           *
*         BSPG*S.TTPS03                 BSPG*S.TTPS03                       *
*              --------                      --------                       *
*---------------------------------------------------------------------------*
```

FECHA : [illegible]3-01-20 HOJA 1

SUBSISTEMA : TABULADOR DE INTERACCIONES

TIPO DE AREA : FUENTE

NOMBRE ARCHIVO ULTIMA VERSION	NOMBRE ARCHIVO ORIGINAL	COMIENZO DE FIL	COMIENZO DE FOC
[illegible]	[illegible]	0001	0185
			0300

47

E S T A N D A R E S

(CONTROL DE CALIDAD)

CONTROL DE CALIDAD

EN LA

PROGRAMACION

EL OBJETIVO DE CONTROL DE CALIDAD ES GARANTIZAR QUE TODOS LOS PRODUCTOS TERMINADOS CUMPLAN O EXCEDAN LOS REQUERIMIENTOS DE COMPORTAMIENTO Y ACEPTACION QUE HAN SIDO MUTUAMENTE ACORDADOS ENTRE EL USUARIO Y QUIEN DESARROLLA LA PROGRAMACION.

EL CONTROL DE CALIDAD EN EL DESARROLLO DE LA PROGRAMACION ES UN PROCESO COMPLEJO DADA LA INTANGIBILIDAD DE LOS PROGRAMAS DE COMPUTADORA, LA DIFICULTAD DE ESTABLECER LOS REQUERIMIENTOS ADECUADOS Y LA AMBIGUEDAD INHERENTE EN LAS ESPECIFICACIONES A CUMPLIR.

THE WORD STANDARD MAY MEAN

1. A MANDATORY REQUIREMENT

2. A RECOMMENDED PRACTICE OR CONVENTION

3. A GUIDELINE

EL ESPECTRO DE LAS FILOSOFIAS

EN ESTANDARIZACION

Tipo de reglas	NINGUNA	GUIAS DE ESTILO	GUIAS Y ESTANDRES	ESTANDARES RIGIDOS
Método de reforzam.	NINGUNO	PLATICAS ESTRUCT.	REVISIONES Y HTAS. AUTOM.	HTAS. AUTOM. Y REVISIONES
Grado de reforzam.	NINGUNO	FLEXIBLE	FIRME	RIGIDO
Analogía	ANARQUIA	DEMOCRACIA	SOBERANIA	DICTADURA

ESTANDARES Y NORMAS

** NORMAS DE ANALISIS

** NORMAS DE DISEÑO DE PROGRAMAS

** NORMAS DE CODIFICACION

** NORMAS PARA NOMBRAMIENTO DE ARCHIVOS, VARIABLES, ETC.

** NORMAS PARA DESARROLLAR EL PSEUDOCODIGO (LDP)

** NORMAS PARA PLANES Y PROCEDIMIENTOS DE PRUEBAS

MATRIZ DE REQUERIMIENTOS

SUBSISTEMA: DESPACHO ECONOMICO RESTRINGIDO

REQUERIMIENTO No.	REFERENCIA A.T.	FASE No.	REFERENCIA R.P.C.	REFERENCIA D.P.C.	REFERENCIA CODIGO	COMENTARIOS
A. Salidas del programa:						
1. Punto base de operación para cada generador despachable.	C.3.18	1	4.3.1.1.4.2.3.4			
2. Factor de participación para cada generador despachable, subida y bajada.	C.3.18	1	4.3.3.1.3.5			
3. Factor de sensibilidad de intercambio de cada área, para el cambio en demanda total del sistema subida y bajada.	C.3.18	1	4.3.3.1.3.5			

RESUMEN

* SUMINISTRE RECURSOS ADECUADOS
* HAGA QUE LAS ESTRUCTURAS DE LOS RECURSOS CASEN CON LAS TAREAS
* USE TECNICAS DE PRODUCTIVIDAD
* ESTABLEZCA UN CONTROL DE CAMBIOS RIGUROSO
* MANTENGA COMUNICACION COHERENTE LOS PARTICIPANTES
* MANTENGA VISIVILIDAD Y CONTROL EFECTIVO

TECNICA DE PRODUCTIVIDAD	- ANALISIS ESTRUCT. - EQUIPOS DE TRABAJO - RUTA CRITICA - REVISION ESTRUCT.	- DISEÑO ESTRUCT. - PROGRAMACION ESTRUCTURADA - REVISION EST.	- PROGRAMACION ESTRUCTURADA - BIBLIOTECA DE SOPORTE - CHECADOR DE CODIGO	REVISION EST.
ESTANDAR	GUIA	GUIA	NORMA	GUIA
METODOLOGIA — FASE	ANALISIS DE REQUERIMIENTOS	DISEÑO	DESARROLLO	OPERACION
METODOLOGIA — ?A	*PLANTEAMIENTO DE OBJETIVOS *ANALISIS *ESPECIFICACION DE REQURIMIENTOS	*ARQUITECTURA *DIAGRAMA ESTRUCTURAL *DETALLE DE MODULOS	* CODIFICACION * INTEGRACION * PRUEBAS	* OPERACION * MANTENIMIENTO
POLITICA ADMINISTRATIVA	ANALISIS DE PRODUCTIVIDAD: -SUBDIVISION DEL PROYECTO - ESTABLECIMIENTO DE LOS NIVELES CUANTITATIVOS Y DE PRODUCTIVIDAD. -ASIGNACION DE RECURSOS -CALCULO DEL COSTO -MONITOREO DE RESULTADOS PARCIALES -ACCIONES CORRECTIVAS	- MEDIDAS CORRECTIVAS - AUDITORIA	- MEDIDAS CORRECTIVAS - AUDITORIA	- MEDIDAS CORRECTIVAS - AUDITORIA
DOCUMENTACION	- CONTRATO DE TRABAJO - ANEXO TECNICO - MATRIZ DE REQUERIM. - DOP - RA - RPC	- ARQ - DPC	- PROGRAMAS - SISTEMA - PLP - PRP	- MUS

R E F E R E N C I A S

Boehm, B.W. (1973): "Software and Its Impact: A Quantitative Study", Datamation, Mayo 1973.

Boehm, B.W. (1976): "Software Engineering", IEEE Transactions on Computers, Dic. de 1976.

Buxton, J.M., P. Naur y B.Randell, eds (1976): Software Engineering: Concepts and Techniques, Petrocelli/Charter, 1976.

Brooks, F. (1975): The Mythical Man-month, Addison-Wesley, 1975.

Couger, J.D. y R.W. Knapp (Eds.) (1974): System Analysis Techniques, Wiley, 1974

Davis, C.G. y C.R. Vicks (1976): "The Software Development -- System",

IEEE Transactions on Software Engineering, Ene. de 1977

Dijkstra, E.W. (1968): " GO TO Statement Considered Harmful", Comm. of the ACM. 11, mar. de 1968

Dijkstra, E.W. (1972): "Notes on Structured Programming", en O-J Dahl et al, Structured Programming, Academic Press, 1972.

Dijkstra,E.W. (1976): A Discipline of Programming, Prentice-Hall, 1976.

Gane. C. y T. Sarson (1977): Structured Systems Analysis, -- Improved System Technologies, 1977.

Gilb, T y G.M. Weinberg (1977): Humanized Input, Whintrop,1977.

IBM (1975): HIPO - A Design Aid and Documentation Technique, Orden GC20-1851, 1975.

Jackson, M.A.(1975): Principles of Program Design, Academic Press, 1975.

Jones, M.N. (1976): "HIPO for Developing Specifications", Datamation, Marzo 1976.

Katzan, H. (1976) : System Design and Documentation, Van Nostrand Reinhold, 1976.

Kernighan, B.W. y P.J. Plauger (1974): The Elements of Programming Style, McGraw-Hill, 1978, 2a. edición

Kernighan, B.W. y P.J. Plauger (1976): Software Tools, Addison-Wesley, 1976.

Knuth, D.E. (1974): "Structured Programming with GO TO Statements", Computing Surverys, Vol.6, No. 4.

Metzger, P. (1975): Programming Project Management, Prentice-Hall, 1975.

Mills, H.D. (1976): "Software Development", IEEE Transactions on Software Engineering, Dic. 1976.

Myers, G.J. (1975): Reliable Software Through Composite Design, Petrocelli/Charler, 1975.

Myers, G.J. (1976): Software Reliability, Wiley, 1976.

Myers, G.J. (1978): Composite/Structured Design, Van Nostrand - Reinhold, 1978.

Orr, K.T. (1977): Structured Systems Development, Yourdon -- Press, 1977.

Ross, D.T. (1976): "Structured Analysis (SA): A Language for Comunicating Ideas", IEEE Transactions on Software Engineering, Ene. de 1977.

Ross, D.T. y K.E. Schoman Jr. (1976): Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, Ene. de 1977.

Sackman, Erickson y Grant (1978): "Exploratory Experimental - Studies Comparing Online and Offline Programming Performance", Comm. of the ACM. Enero 1978.

Stay , J.F. (1976): "HIPO and Integrated Program Design", IBM Systems Journal, Vol. 15, No. 2, 1976.

Tausworthe, R.C. (1976): Standarized Development of Computer - Software, Jet Propulsion Laboratory, 1976.

Teichroew, D. y E.A. Hershey, III (1976): "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of information Processing Systems", IEEE Transactions on Software -- Engineering, Ene. 1977.

Warnier, J.D. (1974): Logical Construction of Programs, Van - Nostrand Reinhold, 1976, 3a. edición.

Warnier, J.D. (1975): L'Organisation des Donnees d'un Systeme, Les Editions d'Organisation, París, Francia, 1975.

Warnier, J.D. (1975): La Transformation des Programmes, Les - Editions d'Organisation , París, 1975.

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

Weinberg, G.M. et alii: High-level COBOL Programming, Winthrop, 1977.

Weinwurn, G. ed (1970): On the Management of Computer Programming, Auerbach, 1970.

Wirth, N. (1973): Systematic Programming, Prentice-Hall, 1975

Wirth, N. (1974): "On the Composition of Well-structured - Programs", Computing Surverys, Vol. 6, No. 4.

Wirth, N. (1976): Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

Yourdon, E. (1975): Techniques of Program Structure and Design, Prentice-Hall, 1975.

Yourdon, E. y L. Constantine (1975): Structured Design. Yourdon Press, 1978, 2a. edición.

Yourdon, E. (1976): How to Manage Structured Programming, Yourdon Press, 1976.

Yourdon, E. (1977): Structured Walkthroughs, Yourdon Press, 1977

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

TECNICAS MODERNAS DE DESARROLLO Y ADMINISTRACION DE PROGRAMACION

- TIPOS DE PRRUEBAS DE PRODUCTOS DE COMPUTO
- CONSIDERACIONES ·ENERALES SOBRE PRUEBAS
- ›ETODOS DE DISEÑO DE CASOS DE PRUEBA
- PLANES Y PROCEDIMIENTOS DE PRUEBA
- EJEMPLO DE DISEÑO DE CASOS DE PRUEBA

ING. ROLANDO NIEVA

MARZO 1983

Clasificación de errores

- Error de análisis.
- Error de especificación.
- Error de diseño.
- Error de código.

Error de análisis

- Responsable de:
  - objetivos irrelevantes. No satisfacen necesidades.
  - Métodos incapaces de resolver el problema.

Error de especificación

- Responsable de especificación incompleta, imprecisa, incorrecta
- Repercute en diseño y código
- Originado en etapas de análisis, diseño de sistema y detallado.

Error de diseño

- Diseño infactible
- Diseño insatisfactorio

Control de calidad

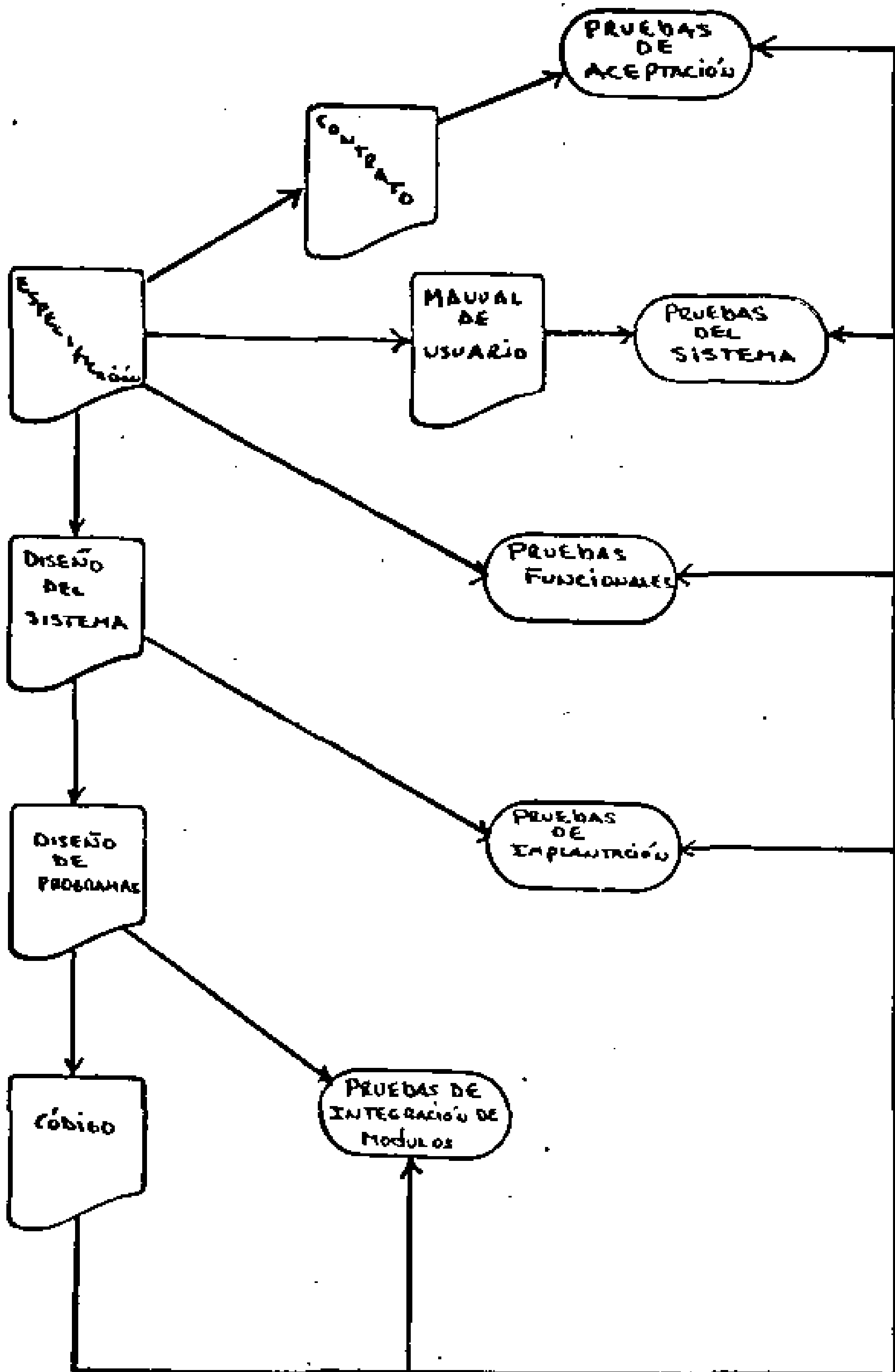
- Documentar ideas, especificación precisa
- Validación de productos durante y al final de cada etapa.
- Pruebas sobre el código.

Tipos de pruebas

- Pruebas modulares (integración, construcción).
- Pruebas de implantación.
- Pruebas funcionales.
- Pruebas de sistema.
- Pruebas de aceptación.

Pruebas Modulares

- Identifican desapego entre código y el diseño detallado.
- En general, responsabilidad del programador.
- Técnica de prueba se basa en el diseño detallado del programa.
- Se considera la lógica interna del programa.



Tipos De Pruebas

Pruebas de implantación

- Identifican desapego entre código de programas y diseño del sistema.
- Técnica de prueba se fundamenta en el diseño del sistema.
  - Arquitectura
  - Base de datos
  - Interfaces
- Responsabilidad del grupo de integración.

Pruebas funcionales

- Identifican desapego entre código y especificación funcional.
  - Errores en diseño debido a mala interpretación de requerimientos.
  - Algoritmos o métodos de solución incorrectos (análisis).
  - Diseño insatisfactorio.
- En general responsabilidad del analista.
- Técnica de prueba se fundamenta en la especificación. No se considera la lógica interna de programas.
- Se consideran los resultados de la etapa de análisis.
  - Simulaciones.
  - Prototipos.

Pruebas de sistema

- Identifica desapego entre código, el manual de usuario y objetivos.
- Fuente básica de información es el manual de usuario.
- Beneficio: revisión exhaustiva del manual de usuario.
- No se considera la lógica interna de los programas.
- En general es responsabilidad de analistas

Prueba de aceptación

- Compara el producto final con los términos del contrato.
- Responsabilidad del cliente.
- Naturaleza similar a las pruebas funcionales y de sistema
- No se considera la lógica de programas
- Se acostumbra subcontratar a otra institución.

CONSIDERACIONES GENERALES

* Objetivo de probar:
  - Encontrar errores: Actitud crítica.
* Práctica efectiva:
  - Suponer errores e idear casos de prueba para encontrarlos.
* Probar:
  - Proceso difícil y costoso.
  - Requiere de ingenio y creatividad.
  - A pesar de su importancia es un arte. Fundamentado en la experiencia.
  - Exhaustivamente;- imposible.
* Un sistema puede ser útil aunque no satisfaga el 100% de los requerimientos.
  - Compromiso práctico entre costo y confiabilidad.
* Prueba mejor un ente extraño al desarrollo del objeto de prueba.
  - No hay polarización.
  - No hay lazos sentimentales.

DEFINICIONES

* **CASO DE PRUEBA.**

CONJUNTO DE DATOS DE ENTRADA Y SALIDA, DONDE LOS DATOS DE SALIDA SON LOS QUE EL PROGRAMA, AL ESTAR LIBRE DE ERROR, PRODUCIRÁ A PARTIR DE LA ENTRADA.

- BONDAD DE UN CASO: POTENCIAL DE ENCONTRAR ERRORES.
- CASO EXITOSO: AQUEL QUE DETECTA AL MENOS UN ERROR HASTA ENTONCES DESAPERCIBIDO.

* **PRUEBA.**

CONJUNTO DE CASOS DE PRUEBA DISEÑADOS PARA ENCONTRAR ERRORES ASOCIADOS A UNO O VARIOS REQUERIMIENTOS EN COMÚN.

* **PLAN DE PRUEBAS**

DOCUMENTO QUE DESCRIBE LAS PRUEBAS PROPUESTAS.

Procedimientos de prueba

Documento que describe la logística requerida para realizar las pruebas.

- Equipo
- Software
- Base de datos
- Personal
- Guía de acción
- Permite reproducir pruebas.

Métodos de diseño de casos de prueba

- Análisis de entrada-salida (caja negra).
- Cobertura de lógica.

* Propósito:

Diseñar un número reducido de casos con alta probabilidad de encontrar errores.

MÉTODO DE ANÁLISIS DE ENTRADA-SALIDA

- PERMITE DISEÑAR CASOS DE PRUEBA A PARTIR DE LA ESPECIFICACIÓN DE REQUERIMIENTOS.
- NO SE REQUIERE DE LA LÓGICA INTERNA DE PROGRAMAS.
- PERMITE PLANEACIÓN DE PRUEBAS EN PARALELO CON DISEÑO Y DESARROLLO.
- BENEFICIO SECUNDARIO: DETECCIÓN DE DEFICIENCIAS EN ESPECIFICACIÓN.
- PRÁCTICA BASADA EN 3 PRINCIPIOS FUNDAMENTALES:

1.- UN PROGRAMA SIN ERRORES ARROJA EL RESULTADO CORRECTO PARA TODAS Y CADA UNA DE LAS ENTRADAS POSIBLES.

2.- COSTO Y TIEMPO AUMENTA CON EL NÚMERO DE CASOS DE PRUEBA.

3.- ES FACTIBLE AGRUPAR LAS ENTRADAS POSIBLES EN UN NÚMERO FINITO Y REDUCIDO DE CLASES (SUBCONJUNTOS DE ENTRADAS CON CARACTERÍSTICAS SIMILARES).

- COBERTURA DE ENTRADAS.
- ELEMENTOS "TÍPICOS" DE CADA CLASE.

Definiciones

Entrada:

Conjunto de datos que un programa procesa para producir un conjunto de datos de salida. La entrada está constituida por componentes.

Ejemplo. Entrada compuesta por:

a) bandera con dos posibles estados

b) matriz cuadrada NxN de números reales.

c) secuencia de m caracteres alfanuméricos entre w secuencias posibles.

Entrada arbitraria

Elemento del conjunto de todas las posibles entradas.

Universo de entradas

Conjunto de todas las posibles entradas.

ETAPAS DE ANÁLISIS DE ENTRADA-SALIDA

- ANÁLISIS CONCEPTUAL.
- CARACTERIZACIÓN DE LA SALIDA.
- CARACTERIZACIÓN DE LA ENTRADA.
- SELECCIÓN DE CASOS DE PRUEBA.

ANÁLISIS CONCEPTUAL

- RESPUESTA A:
  - PROBLEMA?
  - CONCEPTO CLAVE QUE FUNDAMENTA LA SOLUCIÓN?
  - CONDICIONES QUE SON SUSCEPTIBLES A ERRORES DE DISEÑO Y CÓDIGO?
- DIFERENCIA ENTRE CASOS AL AZAR Y CASOS CON ALTA PROBABILIDAD DE SER EXITOSOS.

{ · FACTIBILIDAD
  · DEPENDENCIA FUNCIONAL DE ENTRADAS Y SALIDAS

CARACTERIZACIÓN DE ENTRADAS Y SALIDAS

- AGRUPAMIENTO DE ENTRADAS Y/O SALIDAS EN CLASES.
- PROCEDIMIENTO HEURÍSTICO.
- DISCRIMINACIÓN ENTRE ENTRADAS VÁLIDAS E INVÁLIDAS.
- DISCRIMINACIÓN ENTRE SALIDAS VÁLIDAS E INVÁLIDAS.
- REQUIERE DE ANÁLISIS MINUCIOSO DE LA ESPECIFICACIÓN.
- DOS CRITERIOS:

SELECCIÓN DE CASOS DE PRUEBA.

- SELECCIONAR CASOS TÍPICOS QUE CUBRAN CADA CLASE DE ENTRADA.
  - ENTRADAS VÁLIDAS E INVÁLIDAS.
  - SALIDAS VALIDAS.
- SELECCIONAR CASOS TÍPICOS QUE CUBRAN LAS FRONTERAS ENTRE CLACES.
- CASO TÍPICO POR CADA CONDICIÓN DE DATOS INVÁLIDOS.
- SUPONER ERRORES DEBIDOS A:
  - IMPRECISIÓN DE ESPECIFICACIÓN
  - OMISIONES
- ELIMINAR REDUNDANCIA EN EL CONJUNTO DE CASOS PROPUESTO.

5.1 ESPECIFICACIÓN DEL PROGRAMA EJEMPLO.

El programa lee 3 valores enteros.

Los 3 valores son interpretados como las longitudes de los lados de un triángulo.

El programa imprime un mensaje indicando a que tipo de triángulo: isósceles, escaleno o equilátero, corresponden los datos de entrada.

Cuando los datos no corresponden a triángulo alguno el programa imprime un mensaje indicando que los datos son inválidos.

ANÁLISIS

a) Los lados de un triángulo son números positivos.

b) La suma de dos lados cualesquiera de un triángulo debe ser mayor que la longitud del lado restante.

5.3 CARACTERIZACION DE LA SALIDA.

La salida del programa es un mensaje (un solo componente).

El conjunto de mensajes válidos es |ESCALENO, ISOSCELES, EQUILATERO, DATOS INVALIDOS|.

El conjunto de salidas inválidas no es finito. Está compuesto por todo mensaje diferente a los válidos.

El universo de salidas (el conjunto de todas las posibles salidas) puede ser particionado en las 5 clases de la tabla 1.

CLAVE CLASE	CONDICION SOBRE SALIDA	VALIDA(V)/INVALIDA(I)
S(1)	ESCALENO	V
S(2)	ISOSCELES	V
S(3)	EQUILATERO	V
S(4)	DATOS INVALIDOS	V
S(5)	*	I

*NOTA: NO EXISTE MENSAJE DE SALIDA O NO ES UNO DE LOS 4 MENSAJES PERMITIDOS.

TABLA 5.1 CLASES DEL UNIVERSO DE SALIDA PARA EL EJEMPLO

5.4 CARACTERIZACIÓN DE LA ENTRADA.

La entrada al programa es un conjunto de 3 datos que para propósitos de este ejemplo serán representados por a, b y c.

Las condiciones que establecen la validez o invalidez de los datos de entrada se listan en la tabla 5.2

ESPECIFICACION	CONDICION VALIDA	CONDICION INVALIDA
NUMERO DE DATOS DE ENTRADA	3 ENTEROS POSITIVOS	UNO O MAS DATOS NO SON ENTEROS POSITIVOS
LONGITUDES DE LADOS DE UN TRIANGULO CUALQUIERA	LA SUMA DE CADA COMBINACION DE DOS LADOS ES MAYOR QUE EL LADO RESTANTE	LA SUMA DE AL MENOS UNA COMBINACION DE 2 LADOS ES MENOR O IGUAL AL RESTANTE

TABLA 5.2 CONDICIONES SOBRE DATOS DE ENTRADA PARA EL EJEMPLO 1.

El universo de entrada puede ser particionado en 4 clases, agrupando en cada clase de entrada, todas las posibles entradas que causen una misma salida válida.

En notación de conjuntos, si P representa la 'Transformación' que el programa realiza sobre sus entradas y P[e(i)] la salida asociada a una entrada cualquiera e(i). Las clases de entrada [E(i)] son definidas en la forma siguiente:

E(i) es el conjunto de entradas e(i), tales que P[e(i)] es elemento de la clase S(i).

La clase E(4) que agrupa a todas las entradas inválidas puede ser particionada a su vez en subclases que agrupan a entradas que exhiben solamente una condición inválida. Cada subclase agrupa todas las entradas que exhiben una misma condición inválida. Las subclases propuestas son:

E(4.1): El conjunto de entradas tales que entre sus componentes al menos uno no es entero positivo.

E(4.2): El conjunto de entradas tales que sus componentes son enteros positivos y la suma de al menos una combinación de 2 lados es menor o igual al restante.

5.5 RESEÑA DE LA PRUEBA.

Los casos de prueba propuestos a continuación, se obtuvieron al seleccionar entradas típicas de cada clase o subclase de entrada. Cuando fue posible se seleccionaron entradas en las fronteras de las clases.

En total son 16 los casos de prueba propuestos.

1 Caso válido de triángulo escaleno (clase E(1)).

3 Casos válidos de triángulo isósceles (clase E(2)); que incluye las permutaciones de los dos lados iguales y el desigual

1 Caso válido de triángulo equilátero (clase E(3)).

5 Casos cuyas entradas son tales que entre sus componente al menos uno no es entero positivo (clase E(4.1));

a) Todos los datos son enteros, uno es negativo.

b) Todos los datos son enteros, uno es cero.

c) Uno de los datos no es entero.

d) Uno de los datos no es número.

e) Uno de los datos no existe.

6 Casos cuyas entradas son compuestas por números enteros positivos tales que la suma de al menos una combinación de dos lados es menor o igual al lado restante (clase E(4.2));

a) a+b<c
b) a+b=c
c) a+c<b
d) a+c=b
e) b+c<a
f) b+c=a

El método de verificación consistirá en inspeccionar el listado impreso por el programa.

Se declarará la existencia de error cuando para un caso de prueba el resultado impreso por el programa no coincida con la salida esperada

CASOS DE PRUEBA

NUMERO DE CASO	CLASE	ENTRADA A	ENTRADA B	ENTRADA C	SALIDA
1	E(1)	3	5	4	ESCALENO
2	E(2)	4	5	5	ISOSCELES
3	E(2)	5	4	5	ISOSCELES
4	E(2)	5	5	4	ISOSCELES
5	E(3)	5	5	5	EQUILATERO
6	E(4.1)	5	4	-5	DATOS INVALIDOS
7	E(4.1)	5	0	5	.
8	E(4.1)	4	3.1	5	.
9	E(4.1)	Z	4	5	.
10	E(4.1)	4	-	5	.
11	E(4.2)	5	5	10	.
12	E(4.2)	5	5	11	.
13	E(4.2)	5	11	5	.
14	E(4.2)	5	10	5	.
15	E(4.2)	11	5	5	.
16	E(4.2)	10	5	5	.

TABLA 5.3 CASOS DE PRUEBA PARA EL PROGRAMA DEL EJEMPLO 1.

OBSERVACIONES

LA ESPECIFICACIÓN ES DEFICIENTE. DEBE INCLUIR:

- FORMA DE ALIMENTAR DATOS.
- TEXTO DE LOS MENSAJES DE SALIDA.
- CONDICIONES SOBRE LOS DATOS DE ENTRADA QUE DAN ORIGEN A LOS DISTINTOS MENSAJES DE SALIDA.
- CARACTERIZACIÓN DE LA ENTRADA: QUE DATOS PUEDEN PRECENTARSE.

DIRECTORIO DE ASISTENTES AL CURSO: TECNICAS MODERNAS DE DESARROLLO Y ADMON. DE PROGRAMACION

NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
1. MEDARDO FEDERICO BARRERA GODINEZ Av. Cuernavaca No. 68 Col. Sta. María Nonoalco Mixcoac Del. Alvaro Obregón C.P. 01420 Tel. 563-47-20	BANPECO José María Marroquí No. 81 Col. Centro Del. Cuauhtémoc C.P. Tel. 521-43-80
2. IGNACIO CARBAJAL CARBAJAL Fernando Ramírez No. 53 Col. Obrera Del. Cuauhtémoc C.P. Tel. 578-52-45	I.M.P. Av. Lazaro Cardenas 152 Tel. 56766-00 Ext. 2527
3. ARMANDO CEJA SAUCEDO Bocanegra No. 21 INT. 30 Col. Guerrero Del. Cuauhtémoc C.P. Tel.	
4. GUILLERMO DE LA FUENTE GUZMAN Calle 13 No. 101 Col. Olivar del Conde Del. Alvaro Obregón C.P. 01400 Tel.	BANCO DEL PEQUEÑO COMERCIO DEL D.F. José María Marroquí No. 81 Col. Centro Del. Cuauhtémoc C.P. 06050 Tel. 521-43-80 Ext. 190
5. JESUS FUENTES FLORES Av. San Bernabe No. 206 Col. San Bernabe Del. Magdalena Contreras C.P. 10300 Tel. 595-60-69	BANCO DEL PEQUEÑO COMERDIO DEL José María Marroquí No. 81 Col. Centro Del. Cuauhtémoc C.P. 06050 Tel. 521-43-80 Ext. 190

DIRECTORIO DE ASISTENTES AL CURSO: TECNICAS MODERNAS DE DESARROLLO Y ADMON. DE PROGRAMACION

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
6.	JAVIER HERNANDEZ ROBERT Nacozarí 37 Depto. 30-C Col. San Francisco Del. Mag. Contreras C.P. 10810 Tel.	I.M.P. Av. Eje Central Lazaro Cardenas No. 152 Tel. 567-66-00
7.-	VICTOR MANUEL LACANO ROMERO	BANCO DEL PEQUEÑO COMERCIO DEL DF José Ma. Marroquí No. 81 Col. Centro Del. Cuauhtémocs C.P. 06050 Tel. 521-43-80
8.-	MIGUEL ANGEL MORA ESPINOSA L. Cardenas No. 490-1301 Col. Tlateloico Del. Cuauhtémoc C.P. 06900 Tel. 597-54-96	CENTRO DE ECODESARROLLO Altadema No. 8 Col. Nápoles Del. C.P. Tel. 523-18-02
9.-	JULIO MODESTO MUÑOZ GALVEZ Huasteca No. 263 Col. Industrial Del. Gustavo A. Madero C.P. Tel. 533-02-73	BENJAMIN MORA GONZALEZ, I.C. Londres No. 71 Col. Juárez Del. Cuauhtémoc C.P. 06600 Tel. 511-85-98
10.-	MARIO PALOMAR ALCIBAR Av. Cuauhtémoc No. 877-3 Col. Narvarte Del. Benito Juárez C.P. Tel. 543-78-85	D.E.P.F.I. UNAM Ciudad Universitaria Tel. 550-52-15 Ext. 4493

DIRECTORIO DE ASISTENTES AL CURSO: TECNICAS MODERNAS DE DESARROLLO Y ADMON. DE PROGRAMACION

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
11.-	CARLOS PEREGRINA RAMIREZ Calz. Ahuizotla 125 Int. 19 Col. Santiago Ahuizotla Del. Azcapotzalco C.P. 02750 Tel.	UNIVERSIDAD PEDAGOGICA NACIONAL Km. 0.5 Carr. Ajusco Col. Héroes de Padierna Del. Tlalpan C.P. Tel. 652-33-99
12.-	CLEMENTE JUAN PASIO RODRIGUEZ BARRON José Rodríguez González No. 9 Col. Constitución de 1917 Del. Iztapalapa C.P. 09260 Tel. 6910848	D.E.P.F.I. Ciudad Universitaria
-	GABRIEL SANCHEZ BOJORQUEZ Calle Mixcoac No. 21-B Col. Merced Gómez Del. Benito Juárez C.P. Tel.	SEGUROS AMERICA BANAMEX Av Revolucion 1510 Col. Guadalupe Inn Del. Alvaro Obregón C.P. Tel.
14.-	GABRIEL SANCHEZ GUERRERO Bretaña No. 124 Col. Portales Del. Benito Juárez C.P. Tel. 550-52-15 Ext. 4486	DIV. EST. POSG. FAC. ING. UNAM Ciudad Universitaria