E V A L U A C I O N D E L P E R S O N A L

D O C E N T E

CURSO: ANALISIS Y DISEÑO DE SISTEMAS DE PROCESAMIENTO DE DATOS 1992

FECHA: 20 de abril al 13 de mayo
lunes, miercoles y viernes
de 17 a 21 hrs.

	C O N F E R E N C I S T A	DOMINIO DEL TEMA	EFICIENCIA EN EL USO DE AYUDAS AUDIOVISUALES	MANTENIMIENTO DEL INTERES COMUNICACION CON LOS ASISTENTES AMENIDAD, FACILIDAD DE EXPRESION	PUNTUALIDAD	PROMEDIO
1.-	ING. ABANIA URIOSTEGUI MONDRAGON					
2.-	ING. MARIA TERESA SORIANO RAMIREZ					
3.-	ING. LAURA SANDOVAL MONTAÑO					
4.						
5.-						
	EVALUACION TOTAL					

ESCALA DE EVALUACION: 1 A 10

CURSO: ANALISIS Y DISEÑO DE SISTEMAS
PROCESAMIENTO DE DATOS 1992.

FECHA: 20 de abril al 13 de mayo
lunes, miercoles y viernes
de 17 a 21 hrs.

	TEMA	ORGANIZACION Y DESARROLLO DEL TEMA	GRADO DE PROFUNDIDAD LOGRADO EN EL TEMA	GRADO DE ACTUALIZACION LOGRADO EN EL TEMA	UTILIDAD PRACTICA DEL TEMA	PROMEDIO
1.-	ANALISIS DEL SISTEMA					
2.-	ANALISIS DE REQUERIMIENTOS					
3.	ESTIMACION DE COSTOS DEL SOFTWARE					
4.-	FUNDAMENTOS DE DISEÑO DE SISTEMAS					
5.-	DISEÑO ORIENTADO AL FLUJO DE DATOS					
6.-	DISEÑO ORIENTADO A LA ESTRUCTURA					
	DE DATOS					
7.	DISEÑO ORIENTADO AL OBJETO					
8.	IMPLEMENTACION DE SISTEMA					
9.	VERIFICACION VALIDACION DEL SIST.					
10.	MANTENIMIENTO					

EVALUACION TOTAL

ESCALA DE EVALUACION: 1 A 10

EVALUACION DEL CURSO

	C O N C E P T O	
1.-	APLICACION INMEDIATA DE LOS CONCEPTOS EXPUESTOS	
2.-	CLARIDAD CON QUE SE EXPUSIERON LOS TEMAS	
3.-	GRADO DE ACTUALIZACION LOGRADO EN EL CURSO	
4.-	CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
5.-	CONTINUIDAD EN LOS TEMAS DEL CURSO	
6.-	CALIDAD DE LAS NOTAS DEL CURSO	
7.-	GRADO DE MOTIVACION LOGRADO EN EL CURSO	
	EVALUACION TOTAL	

ESCALA DE EVALUACION: 1 A 10

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

MUY AGRADABLE ( ) AGRADABLE ( ) DESAGRADABLE ( )

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

PERIODICO EXCELSIOR ANUNCIO TITULADO DIVISION DE EDUCACION CONTINUA ( )

PERIODICO NOVEDADES ANUNCIO TITULADO DIVISION DE EDUCACION CONTINUA ( )

FOLLETO DEL CURSO ( )

CARTEL MENSUAL ( )

RADIO UNIVERSIDAD ( )

COMUNICACION CARTA, TELEFONO, VERBAL, ETC. ( )

REVISTAS TECNICAS ( )

FOLLETO ANUAL ( )

CARTELERA UNAM "LOS UNIVERSITARIOS HOY" ( )

GACETA UNAM ( )

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

AUTOMOVIL PARTICULAR ( )

METRO ( )

OTRO MEDIO ( )

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

_____

_____

_____

5.- ¿Recomendaría el curso a otras personas? ( ) SI ( ) NO

5.a.¿Qué periódico lee con mayor frecuencia?

_____

_____

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

_____

_____

_____

7.- La coordinación académica fué:

EXCELENTE ☐ BUENA ☐ REGULAR ☐ MALA ☐

8.- Si está interesado en tomar algún curso INTENSIVO ¿Cuál es el horario más conveniente para usted?

LUNES A VIERNES DE 9 a 13 H. Y DE 14 A 18 H. (CON COMIDAD) ☐

LUNES A VIERNES DE 17 a 21 H. ☐

LUNES A MIERCOLES Y VIERNES DE 18 A 21 H. ☐

MARTES Y JUEVES DE 18 A 21 H. ☐

VIERNES DE 17 A 21 H. SABADOS DE 9 A 14 H. ☐

VIERNES DE 17 A 21 H. SABADOS DE 9 A 13 H. DE 14 A 18 H. ☐

OTRO ☐

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

_____

_____

_____

10.- Otras sugerencias:

_____

_____

_____

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

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

ANALISIS Y DISEÑO DE SISTEMAS DE PROCESAMIENTO DE DATOS

NOTAS ELABORADAS EN EL CENTRO DE
CALCULO DE LA FACULTAD DE INGENIERIA POR:

ABANIA URIOSTEGUI MONDRAGON
MARIA TERESA SORIANO RAMIREZ
LAURA SANDOVAL MONTAÑO

TEMA 1. ANALISIS DEL SISTEMA

1.1 Antecedentes

Es común, al hablar de la comunicación, señalar el vertiginoso e impresionante progreso logrado en las pocas décadas de su existencia. El contexto en el que se ha desarrollado el software está fuertemente ligado a las cuatro décadas de evolución de los sistemas informáticos. Un mejor rendimiento del hardware, un tamaño más pequeño y un costo más bajo, han dado lugar a sistemas informáticos más sofisticados.

La figura 1 describe la evolución del software dentro del contexto de las áreas de aplicación de los sistemas basados en computadoras.Durante los primeros años el hardware (fig 1a) era de propósito general, por otra parte el software se diseñaba a medida para cada aplicación y tenía una distribución relativamente pequeña.

La mayoría del software se desarrollaba y utilizaba por la misma persona u organización, debido a este entorno personalizado del software, el diseño era un proceso implícito, ejecutado en la cabeza de alguien y, la documentación no existía normalmente.

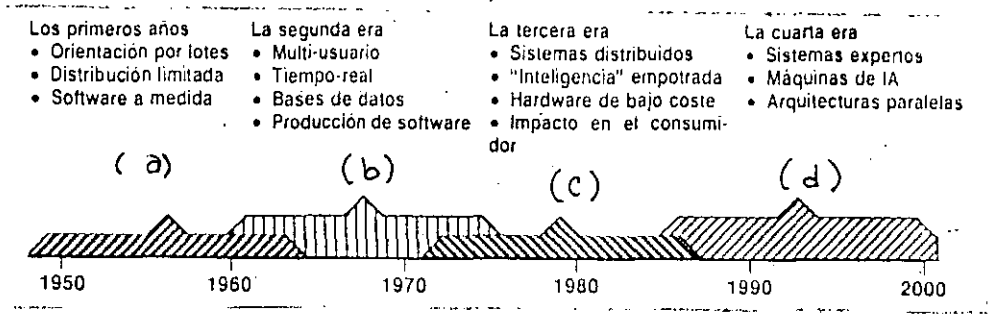


Fig 1.

La segunda era de la evolución de los sistemas de computadoras (fig 1b) se extiende desde la mitad de la década de los 60 hasta finales de los 70. La multiprogramación, los sistemas multiusuarios, introdujeron nuevos conceptos de interacción hombre-máquina.Las técnicas interactivas abrieron un nuevo mundo de aplicaciones, los sistemas podían reconocer, analizar y transformar datos de múltiples fuentes, controlando así los proceso y produciendo salidas en milisegundos en vez de minutos.

La tercera era (fig 1c) de la evolución de los sistemas de computadoras comenzó a mediados de los 70. El sistema distribuido (en este caso computadoras múltiples, cada una ejecutando funciones concurrentes y comunicándose con alguna otra) incremento notablemente la complejidad de los sistemas informáticos.

Redes de área local y global, comunicaciones digitales de alto ancho de banda,la llegada y amplio uso de los microprocesadores y computadoras personales, supusieron una fuerte presión sobre los desarrolladores de software.

La cuarta era (fig 1d) en software de computadoras ha comenzando. Las técnicas de la cuarta generación para el desarrollo de software están cambiando la forma en que algunos segmentos de la comunidad informática construye los programas de computadora. Los sistemas expertos y el software de inteligencia artificial se han trasladado finalmente del laboratorio a aplicaciones prácticas, en un amplio rango de problemas del mundo real.

Es así como el software se ha convertido en el elemento clave de la evolución de los sistemas y productos informáticos.

1.2 DEFINICION DE SISTEMAS Y CARACTERISTICAS.

La palabra sistema es posiblemente el término más sobreutilizado y del que más se ha abusado en el léxico técnico. Hablamos de sistemas políticos y educativos, sistemas bancarios y de ferrocarril. La palabra no dice poco.

Podemos definir un sistema como un conjunto u ordenación de elementos organizados para llevar a cabo algún método, procedimiento o control mediante el procesamiento de información.

Dependiendo a la forma en la que se analizan los sistemas se tienen dos tipos de sistemas :

- **Abierto** : En este caso se deja al libre albedrío de la persona encargada.

- **Cerrado** : Se delimitan fronteras a partir de las cuales se trabajará, de tal forma que aquello que se encuentre fuera de las fronteras no afectará al sistema.

Conforme al comportamiento que presentan los sistemas se tiene la clasificación siguiente :

- **Determinístico** : El comportamiento que presenta el sistema es siempre el mismo.

- **Equifinalidad** : En este caso sea cual sea el estado inicial del sistema siempre se llegará a un estado final único.

- **Homeostático** : Para este caso los sistemas que se tienen siempre se están retroalimentando.

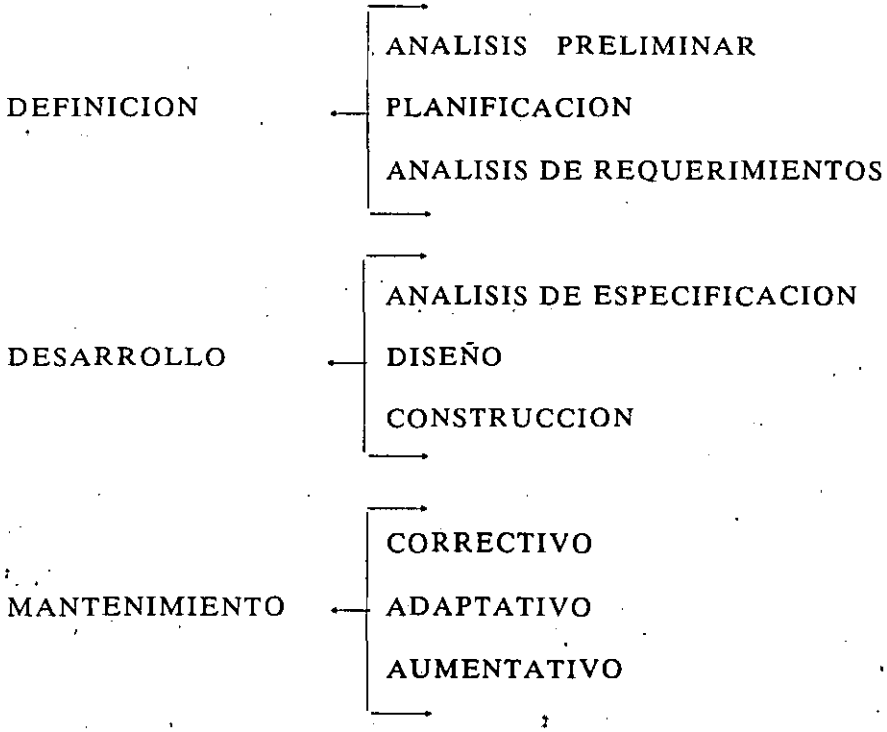
- **Teológico** : Este comportamiento se observa en todos los sistemas porque mantiene un proceso y persigue un objetivo.

1.3 CICLOS DE VIDA

Los procedimientos de la ingenieria del software son la cola que pega a los métodos y herramientas y facilita un desarrollo racional y oportuno del software de computadora. Los procedimientos definen la secuencia en la que se aplican los métodos, las entregas (documentos, informes, formas,etc.) que se requieren, los controles que ayudan a asegurar la calidad y coordinar los cambios y las guías que facilitan a los gestores del software establecer su desarrollo.

El ciclo de vida es aquel que requiere un enfoque sistemático, secuencial, del desarrollo del software que comienza en el nivel del sistema y progresa a través de sus diversas fases.

Cada ciclo de vida tiene las fases siguientes :



TEMA 2. ANALISIS DE REQUERIMIENTOS.

Subtema 2.1 Fundamentos del Análisis de requerimientos

Para realizar bien el desarrollo de software es esencial realizar una especificación completa de los requerimientos del mismo.

Es importante realizar una especificación completa de los requerimientos del software, la tarea de análisis de requerimientos es un proceso de descubrimiento y refinamiento, de ahí la importancia de una buena especificación.

El papel del desarrollador de software y del cliente es un papel activo, el cliente intenta reformular su concepto de la función y comportamiento de los programas en detalles concretos, el que desarrolla el software actúa como interrogador, consultor y el que resuelve los problemas.

El analista.

El analista ejecuta o coordina cada una de las tareas asociadas con el análisis de los requerimientos de software.

El analista debe exhibir los siguientes rasgos de carácter :

- Habilidad para comprender conceptos abstractos, reorganizarlos en divisiones lógicas y sintetizar "soluciones" basadas en cada división.

- Habilidad para entresacar hechos importantes de fuentes conflictivas o confusas.

- Habilidad para comprender entornos de usuario/cliente.

- Habilidad para aplicar elementos hardware y/o software a entornos de usuario/cliente.

- Habilidad par comunicarse bien en forma escrita y verbal.

Los individuos que se paran excesivamente en los detalles con frecuencia pierden de vista el objetivo global de los programas. Los requerimientos del software deben ser descubiertos de una manera "descendente"; las funciones importantes, interfaces e información deben comprenderse completamente antes de que se especifiquen los detalles de las capas sucesivas.

Los problemas subyacentes al análisis de requerimientos son atribuibles a muchas causas:

- Pobre comunicación : Esto hace difícil la adquisición de la información.

- Técnicas y herramientas inadecuadas : Esto produce especificaciones inadecuadas o imprecisas.

- Tendencia a acortar el análisis de requerimientos : Conduce a un análisis inestable.

- Un fallo en considerar alternativas antes de que se especifique el software.

Principios del análisis.

Cada método de análisis tiene una única notación y punto de vista. Sin embargo, todos los métodos de análisis están relacionados por un conjunto de principios fundamentales :

1. El dominio de la información, así como el dominio funcional de un problema debe ser representado y comprendido.

2. El problema debe subdividirse en forma que se descubran los detalles de una manera progresiva (o jerárquica).

3. Deben desarrollarse las representaciones lógicas y físicas del sistema.

Haciendo referencia un poco al dominio de la información. Todas las aplicaciones del software pueden colectivamente llamarse procesamiento de datos. Este término contiene la clave de lo que entendemos por requerimientos del software. El software se construye para procesar datos, pero transformar datos de una forma a otra; esto es, para aceptar entrada, manipularla de alguna forma y producir una salida.

Vayamos ahora a la construcción de prototipos del software

En algunos casos es posible aplicar los principios de análisis fundamental y derivar a una especificación en papel del software desde el cual pueda desarrollarse un diseño. En otras situaciones, se va a una recolección de los requerimientos, se aplican los principios de análisis y se construye un modelo de software, llamado prototipo, según las apreciaciones del cliente y del que lo desarrolla. Finalmente, hay circunstancias que requieren la construcción de un prototipo al comienzo del análisis, puesto que el modelo es el único medio mediante el cual los requerimientos pueden ser derivados efectivamente.

Perfil de las especificaciones de los Requerimientos del Software.

La especificación de requerimientos de software se produce en la culminación de la tarea de análisis. La función y comportamiento asignados al software como parte de la ingenieria de sistemas se refina estableciendo una descripción completa de la información, una descripción funcional detalla, una indicación de los requerimientos de rendimiento y las ligaduras de diseño, unos criterios de validación apropiados y otros datos pertinentes a los requerimientos.

Subtema 2.2 Métodos de Análisis de Requerimientos.

Las metodologías de análisis de requerimientos combinan procedimientos sistemáticos con una notación única para analizar los dominios de información y funcionalidad de un problema de software. En esencia, los métodos de análisis de requerimientos del software, facilitan al ingeniero de software aplicar principios de análisis fundamentales, dentro del contexto de un método bien definido.

Todos los métodos pueden ser evaluados en el contexto de las siguientes características comunes :

1) mecanismos para el análisis del dominio de la información

2) método de representación funcional

3) definición de interfaces

4) mecanismos para subdividir el problema

5) soporte de la abstracción

6) representación de las visiones físicas y lógicas

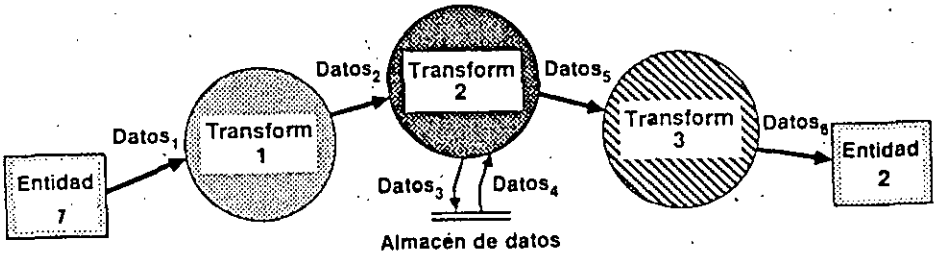
La mayoría de los métodos de análisis permiten al analista evaluar la representación física de un problema antes de derivar a la solución lógica.

Métodos de análisis orientados al flujo de datos.

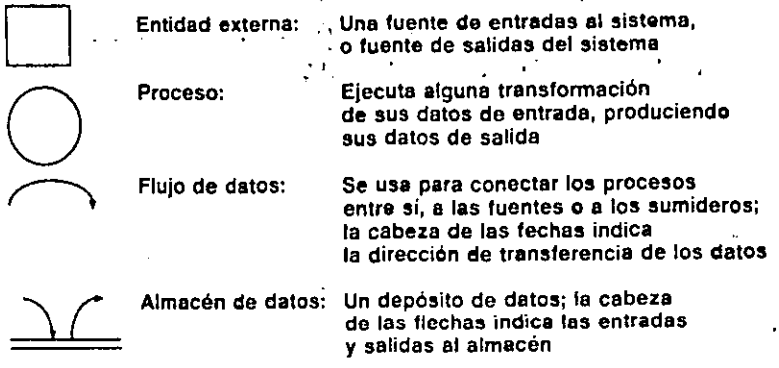
Flujo de Información.

(fig 5.1) Una técnica para representar el flujo de información a través del sistema basado en computadora lo podemos ver en esta figura. La función global del sistema se representa como una transformación sencilla de la información, representada en la figura cómo una burbuja. Una o más entradas, representadas como flechas con etiquetas, conducen la transformación para producir la información de salida.

Un diagrama de flujo de datos (DFD), es una técnica gráfica que describe el flujo de información y las transformaciones que se aplican a los datos, conforme se mueven de la entrada a la salida.



(fig 5.2) La forma básica de un DFD lo podemos ver en esta figura. El DFD puede usarse para representar un sistema o software a cualquier nivel de abstracción. Además pueden particionarse en niveles que representan flujo incremental de información y detalle funcional.



(fig 5.3) La simbología de un DFD la podemos ver en esta figura. Se utiliza un rectángulo para representar una entidad externa, esto es, un elemento del sistema (por ejemplo una persona). Un círculo representa un proceso o transformación que se aplica a los datos y que los cambia de alguna forma. Una flecha representa uno o más elementos de datos. Todas las flechas de un diagrama de flujo de datos deben estar etiquetadas. Una doble línea representa un almacenamiento de datos (información almacenada que es usada por el programa). Debido a la excepcional simplicidad de la simbología del DFD es una de las razones por las que las técnicas de análisis orientadas al flujo de datos son tan ampliamente usadas.

Métodos orientados a la estructura de datos.

Puesto que el dominio de la información para un problema de software comprende el flujo de datos, el contenido de datos y la estructura de datos. Los métodos de análisis orientados a la estructura de datos representan los requerimientos del software enfocándose hacia la estructura de datos en vez de al flujos de datos. Aunque cada método orientado a la estructura de datos tiene un enfoque y notación distinta, todos tienen algunas características en común como son:

Primero, todos asisten al analista en la identificación de los objetos de información clave (también llamados entidades) y operaciones (también llamadas acciones o procesos)

Segundo, todos suponen que la estructura de la información es jerárquica

Tercero, todos requieren que la estructura de datos se represente usando la secuencia, selección y repetición. (vistas anteriormente)

y cuarto, todos dan un conjunto de pasos para transformar una estructura de datos jerárquica en una estructura de programa.

Dentro de los métodos orientados a la estructura de datos, el desarrollo de sistemas estructurados de datos (DSED), también llamado metodología de Warnier-Orr, se basa en el trabajo sobre análisis del dominio de información, realizado por J. D. Warnier. Warnier desarrollo una notación para representar la jerarquía de la información usando las tres construcciones :

1) Secuencia

2) Selección

3) Repetición

El diagrama de Warnier facilita al analista representar jerarquías de información de una manera compacta. Se analiza el dominio de la información y representa la naturaleza jerárquica de la salida.

El método, comprende todos los atributos del dominio de información: flujo, contenido y estructura de datos. Para ilustrar la notación y dar una visión general del método de análisis.

Requerimientos de las bases de datos .

El término de base de datos actualmente se ha convertido en uno de los muchos tópicos del campo de las computadoras. Podemos definir una base de datos como una colección de información organizada de forma que facilita el acceso, análisis y creación de informes.

El análisis de requerimientos para una base de datos incorpora las mismas tareas que el análisis de requerimientos del software. Es necesario un contacto estrecho con el cliente; es esencial la identificación de las funciones e interfaces; se requiere la especificación del flujo, estructura y asociatividad de la información y debe desarrollarse un documento formal de los requerimientos.

TEMA 3. ESTIMACION DE COSTOS DEL SOFTWARE.

En los primeros días de la informática, el coste del software representaba un pequeño porcentaje del coste total del sistema informático, basado en computadora. Un error considerable en las estimaciones del coste del software tenía relativamente poco impacto. Hoy en día, el software es el elemento más caro en muchos sistemas informáticos. Un gran error en la estimación del costo puede marcar la diferencia entre beneficios y pérdidas. Sobrepasar el coste puede ser desastroso para el equipo de desarrollo. La estimación del proyecto de software puede transformarse de un oscuro arte en una serie de pasos sistemáticos que proporcionen estimaciones con un grado de riesgo aceptable.

Para realizar estimaciones seguras de coste y esfuerzo surge un número de opciones posibles:

1. Retrasar la estimación más adelante en el proyecto.

2. Utilizar "técnicas de descomposición" relativamente simples para generar las estimaciones del proyecto de software.

3. Desarrollar un modelo empírico para el coste y el esfuerzo del software.

4. Adquirir una o más herramientas automáticas de estimación.

Desafortunadamente, la primera opción, aunque atractiva, no es práctica: las estimaciones del coste deber ser proporcionadas "de antemano". Sin embargo, debemos reconocer que cuanto más tiempo esperamos, más cosas sabemos y, cuanto más sabemos, menor es la probabilidad de cometer serios errores en nuestras estimaciones.

Las tres opciones restantes son aproximaciones viables para la estimación del proyecto de software. Idealmente, las técnicas señaladas para cada opción deben ser aplicadas en conjunto, cada una usada para comprobar las otras. Las técnicas de descomposición utilizan una aproximación de "divide y vencerás" para la estimación del proyecto de software.

Los seres humanos han desarrollado una aproximación natural a la resolución de problemas: si el problema a resolver es demasiado complicado, tendemos a subdividirlo hasta encontrar problemas manejables. Entonces resolvemos cada uno individualmente y esperamos que las soluciones puedan ser combinadas para responder al problema original.

La estimación del proyecto de software es una forma de resolución de proyectos y, en la mayoría de los casos, el problema a resolver es demasiado complejo para considerarlo como una sola pieza. Por esta razón, descom-

ponemos el problema, re-caracterizándolo como un conjunto de pequeños problemas esperando que sean más manejables.

Estimaciones LDC y PF.

La medición es fundamental para cualquier disciplina de ingeniería del software no es una excepción.

Las métricas del software se refieren a un amplio rango de medidas para el software de computadoras. Dentro del contexto de la planificación del proyecto de software. Se considera al uso de líneas de código (LDC) como medidas clave. Los defensores de la medida LDC afirman que la LDC es un "artefacto" de todos los proyectos de desarrollo de software que puede ser fácilmente calculado, que muchos modelos de estimación del software existentes utilizan LDC o KLDC (miles de líneas de código) como clave de entrada y que ya existe un amplio conjunto de datos y de literatura basada en LDC.

Las métricas del software orientadas a la función son medidas indirectas del software y del proceso por el cual se desarrolla. Más que calcular las LDC, las métricas orientadas a la función se centran en la "funcionalidad" o "utilidad" del programa.

Los puntos de función son obtenidos utilizando una relación empírica basada en medidas contables del dominio de información del software y valoraciones subjetivas de la complejidad del software. La medida del punto de función fue diseñada originalmente para aplicarla en las aplicaciones de sistemas de información comerciales.

Elemento del dominio de información	Cuenta	Factor de ponderación			PF
		Simple	Med.	Complejo	
Número de entradas de usuario		3	4	6	
Número de salidas de usuario		4	5	7	
Número de peticiones de usuario		3	4	6	
Número de archivos		7	10	15	
Número de interfaces externas		5	7	10	

PF = cuenta por el factor de ponderación

(fig. 3.9). Los puntos de función son calculados rellenando la tabla que se muestra en la figura. Se determinan cinco características del dominio de la información y los cálculos aparecen indicados en la posición apropiada de la tabla. Los valores del dominio de la información están definidos de la siguiente manera:

Número de entradas de usuario: Se cuenta cada entrada de usuario que proporciona al software diferentes datos orientados a la aplicación. Las entradas deben ser distinguidas de las peticiones, que se contabilizan por separado.

Número de salidas de usuario: Se cuenta cada salida de usuario que proporciona al usuario información orientada a la aplicación. En este contexto, la salida se refiere a informes, pantalla, mensajes de error, etc. Los elementos de datos individuales dentro de un informe no se cuentan por separado.

Número de peticiones del usuario: Una petición está definida como una entrada interactiva que resulta en la generación de algún tipo de respuesta en forma de salida interactiva. Se cuenta cada petición distinta.

Número de archivos: Se cuenta cada archivo maestro lógico o sea, una agrupación lógica de datos que puede ser una parte de una gran base de datos o un archivo independiente.

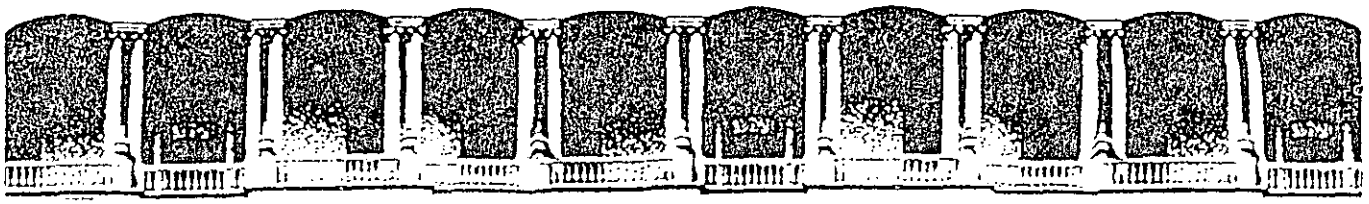
Número de interfaces externas: Se cuentan todas las interfaces legibles por la máquina por ejemplo, archivos de datos en cinta o disco, que son utilizados para transmitir información a otro sistema.

Una vez obtenidos los datos mencionados, se deberá asociar un valor de complejidad a cada cuenta. Las organizaciones que utilizan métodos de puntos de función desarrollan criterios para determinar si una entrada determinada es simple, medida o compleja.

Para calcular los puntos de función (PF) se utiliza la siguiente relación:

$$PF = cuenta.total \times [0.65 + 0.01 \times SUM(Fi)]$$

donde cuenta.total es la suma de todas las entradas obtenidas de la tabla. Los valores constantes de la ecuación anterior y los factores de ponderación aplicados a las cuentas de los dominios de información han sido determinados empíricamente.



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

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

ANALISIS Y DISEÑO DE SISTEMAS DE PROCESAMIENTO DE DATOS

ANALISIS Y DISEÑO DE SISTEMAS DE COMPUTO
TEMA I. ANALISIS DEL SISTEMA
LECTURA: FASES DEL CICLO DE VIDA.

El proceso de desarrollo del software contiene tres fases genéricas independientemente del paradigma de ingeniería elegido. Las tres fases, definición, desarrollo y mantenimiento, se encuentran en todos los desarrollos de software, independientemente del área de aplicación, tamaño del proyecto o complejidad.

La fase de definición se enfoca sobre el qué. Esto es, durante la definición, el que desarrolla el software intenta identificar qué información ha de ser procesada, qué función y rendimiento se desea, qué ligaduras de diseño existen y qué criterios de validación se necesitan para definir un sistema correcto. Por tanto, han de identificarse los requerimientos claves del sistema y del software. Aunque los métodos aplicados durante la fase de definición variarán dependiendo del paradigma de ingeniería del software aplicado, de alguna forma se producirán tres pasos específicos:

Análisis del sistema. El análisis del sistema define el papel de cada elemento de un sistema informático, asignando finalmente el papel que jugará el software.

Planificación del proyecto. Una vez que está asignado el ámbito del software, se asignan los recursos, se estiman los costes y se definen las tareas y planificación del trabajo.

Análisis de requerimientos. El ámbito definido para el software de la dirección, pero antes de comenzar a trabajar, es necesario disponer de una información detallada del dominio de la información y de la función del software.

La fase de desarrollo se enfoca sobre el cómo. Esto es, durante el desarrollo, el que desarrolla el software intenta descubrir cómo han de diseñarse las estructuras de datos y arquitectura del software, cómo han de implementarse los detalles procedimentales, cómo ha de trasladarse el diseño a un lenguaje de programación y cómo ha de realizarse la prueba. Los métodos aplicados durante la fase de desarrollo variarán dependiendo del paradigma de ingeniería del software aplicado. Sin embargo, de alguna forma se producirán tres pasos concretos:

Diseño del software. El diseño traslada los requerimientos del software a un conjunto de representaciones (algunas gráficas, otras tabulares o basadas en lenguajes) que describen la estructura de datos, arquitectura y procedimiento algorítmico.

Codificación. Las representaciones del diseño deben trasladarse a un lenguaje artificial, que da como resultados unas instrucciones ejecutables por la computadora. El paso de la codificación ejecuta esta traslación.

Prueba del software. Una vez que el software se ha implementado en una forma ejecutable por la máquina, debe ser probado para descubrir los defectos que puedan existir en la función, lógica e implementación.

La fase de mantenimiento se enfoca sobre el cambio que va asociado con una corrección de errores, adaptaciones requeridas por la evolución del entorno del software y modificaciones debidas a los cambios de los requerimientos del cliente para reforzar o aumentar el sistema. La fase de mantenimiento replica los pasos de las fases

de definición y desarrollo, pero en el contexto del software existente. Durante la fase de mantenimiento se encuentran tres tipos de cambios:

Corrección. Incluso con las mejores actividades para garantizar la calidad, es probable que el cliente descubra defectos en el software. El mantenimiento correctivo cambia el software para corregir los defectos.

Adaptación. Con el paso del tiempo es probable que cambie el entorno original (por ejemplo, CPU, sistema operativo, periféricos) para el cual se desarrolló el software. El mantenimiento adaptativo se traduce en modificación del software para acomodarlo a los cambios de su entorno externo.

Aumento. Conforme se utiliza el software, el cliente/usuario reconocerá funciones adicionales que podría ser beneficioso añadirlas.

Las fases y pasos relacionados descritos en esta visión genérica de la ingeniería del software, se complementan con varias actividades protectoras. Las revisiones se realizan durante cada paso para asegurar que se mantiene la calidad. La documentación se desarrolla y controla para asegurar que toda la información sobre el sistema y software estarán disponibles para un uso posterior. El control de los cambios se instituye de forma que los cambios puedan ser mejorados y registrados.

El método en cada caso puede variar de un paradigma a otro, pero el enfoque global que exige la definición, desarrollo y mantenimiento permanece invariable. Se puede realizar con disciplina y métodos bien definidos o de forma completamente desordenada. Pero habrá que realizarlos de alguna forma.

TEMA II. ANALISIS DE REQUERIMIENTOS
LECTURA: AREAS DEL ANALISIS DE REQUERIMIENTOS.
El análisis de requerimientos puede dividirse en cuatro áreas:

1) reconocimiento del problema.
2) evaluación y síntesis
3) especificación y
4) revisión

Inicialmente, el analista estudia la especificación del sistema (si existe) y el Plan del Proyecto. Es importante comprender el contexto del sistema y revisar el ámbito de los programas que se uso para generar las estimaciones de la planificación. Posteriormente, debe establecerse la comunicación necesaria para el análisis, de forma que se asegure el reconocimiento del problema. El analista debe establecer contacto con el equipo técnico y de gestión del usuario/cliente y con la empresa que vaya a desarrollar el software. El gestor del programa puede servir como coordinador para facilitar el establecimiento de los caminos de comunicación. El objetivo del analista es reconocer los elementos básicos del programa tal como lo percibe el usuario/cliente.
La evaluación del problema y la síntesis de la solución es la siguiente área principal de trabajo del análisis. El analista debe evaluar el flujo y estructura de la información, refinar en detalle todas las funciones del programa en detalle, establecer las características de la interface del sistema y descubrir las ligaduras del diseño. Cada una de las tareas sirven para describir el problema de forma que pueda sintetizarse un enfoque o solución global.
Por ejemplo, un importante suministrador de materiales de fontanería necesita un sistema de control de inventario. El analista encuentra que los problemas con el sistema manual actual incluyen:
1) imposibilidad de obtener rápidamente el estado de una componente.
2) dos o tres días de tiempo medio para actualizar un archivo de tarjetas y
3) múltiples reórdenes del mismo vendedor debido a que no hay forma de asociar vendedores como componentes, etc. Una vez que se han identificado los problemas, el analista determina qué información va a ser producida por el nuevo sistema y qué datos se le suministrarán al sistema. Por ejemplo, el cliente desea un informe diario que indique los elementos que se han tomado del inventario y cuántos elementos similares permanecen. el cliente indica que los empleados del inventario registrarán el número de identificación de cada pieza cuando dejen el área de inventario.
Hasta la evaluación de los problemas actuales y de la información deseada (entrada y salida), el analista comienza por sintetizar una o más soluciones. Un sistema basado en un terminal en línea resolverá varios problemas, pero parece que se va a necesitar un sistema de gestión de base de datos, pero ¿está justificada la necesidad de asociatividad el usuario/cliente? El proceso de evaluación y síntesis continúa hasta que el analista y el cliente creen que el software puede ser especificado adecuadamente para la fase de desarrollo.
En una etapa no es posible hacer una especificación detallada. El

cliente puede no estar seguro de precisar bien lo que quiere. El que desarrolla el software puede no estar seguro de que un enfoque concreto sea apropiado para realizar la función y comportamiento deseado. Por éstas razones puede presentarse un enfoque alternativo, llamado construcción de prototipos, para el análisis de requerimientos.(Se tratará la construcción de prototipos más adelante).

Las tareas asociadas con el análisis y especificación existen para dar una representación del programa que pueda ser revisada y aprobada por el cliente. En un mundo ideal el cliente desarrolla una Especificación de Requerimientos del Software completamente por sí mismo. Esto se presenta raramente en el mundo real. En el caso mejor, la especificación se desarrolla conjuntamente entre el cliente y el técnico.

Una vez que se hayan descrito las funciones básicas, comportamiento, interface e información, se especifican los criterios de validación para demostrar una comprensión de una correcta implementación de los programas. Estos criterios sirven como base para hacer la prueba durante el desarrollo de los programas. Para definir las características y atributos del software se escribe una especificación de requerimientos formal. Además, para los casos en los que se desarrolle un prototipo se realiza un Manual de Usuario Preliminar.

Puede parecer innecesario desarrollar un manual de usuario en una etapa tan temprana del proceso de desarrollo. Pero de hecho, este borrador de manual de usuario fuerza al analista a tomar el punto de vista del usuario del software. El manual permite al usuario/cliente revisar el software desde una perspectiva de ingeniería humana y frecuentemente produce el comentario: "La idea es correcta, pero ésta no es la forma en que pensé que se podría hacer esto". Es mejor descubrir tales comentarios lo más tempranamente posible en el proceso.

Los documentos del análisis de requerimientos (especificación y manual de usuario) sirven como base para una revisión conducida por el cliente y el técnico. La revisión de los requerimientos casi siempre produce modificaciones en la función, comportamiento, representación de la información, ligaduras o criterios de validación. Además, se realiza una nueva apreciación del Plan del Proyecto Software para determinar si las primeras estimaciones siguen siendo válidas después del conocimiento adicional obtenido durante el análisis.

LECTURA: ANALISIS ORIENTADO AL OBJETO.

El diseño orientado al objeto (DOO), como otras metodologías de diseño orientadas a la información, crea una representación del dominio del problema en el mundo real y lo transforma en un dominio de solución que es software. A diferencia de otros métodos , el DOO da como resultado un diseño que interconexiona los objetos de datos (elementos de datos) y las operaciones de procesamiento, de forma que modulariza la información y el procesamiento en vez de sólo el procesamiento.

La naturaleza única del diseño orientado al objeto está ligada a su habilidad para construir, basándose entres conceptos importantes de diseño del software: abstracción, ocultación de la información y modularidad. Todos los métodos de diseño buscan la creación de software que exhiba estas características fundamentales, pero sólo DOO da un mecanismo que facilita al diseñador adquirir los tres sin complejidad o compromiso.

Orígenes del diseño orientado al objeto.

Los objetivos y las operaciones no son un nuevo concepto de programación, pero sí lo es el diseño orientado al objeto. En los primeros días de la computación, los lenguajes ensambladores facilitaban a los programadores la utilización de las instrucciones máquina (operadores) para manipular los elementos de datos (operandos). El nivel de abstracción que se aplicaba al dominio de la solución era muy bajo.

Conforme aparecieron los lenguajes de programación de alto nivel (por ejemplo Fortran, algol, cobol, etc) los objetos y operaciones del espacio de problemas del mundo real podían ser modelados mediante datos y estructuras de control predefinidas, que estaban disponibles como partes del lenguaje de alto nivel. En general, el diseño de software se enfocaba sobre la representación del detalle procedimental usando el lenguaje de programación elegido. Los conceptos de diseño, tales como refinamientos sucesivos de una función, modularidad procedimental y, posteriormente, programación estructurada, fueron introducidos entonces.

Durante los años 1970, se intrudujeron conceptos tales como abstracción y ocultación de la información y, emergieron métodos de diseño conducidos por los datos, pero los que desarrollaban software aún se encontraban sobre el proceso y su representación. Al mismo tiempo, los lenguajes de alto nivel modernos (por ejemplo Pascal) introdujeron una variedad mucho más rica de tipos y estructuras de datos.

Aunque los lenguajes de alto nivel convencionales (lenguajes a partir de Fortran y algol) evolucionaron durante los años 1960 y 1970, los investigadores estaban trabajando mucho sobre una nueva clase de lenguaje de simulación de prototipos, tales como Simula y Smalltalk. En estos lenguajes, la abstracción de datos tenía una gran importacia y los problemas del mundo real se representaban mediante un conjunto de objetos de datos, a los cuales se les añadía el correspondiente conjunto de operaciones. El uso de estos lenguajes era radicalmente diferente del uso de los lenguajes más convencionales.

El método del diseño orientado al objeto ha emergido durante los últimos 17 años. Los primeros trabajos en diseño de software pusieron la base estableciendo la importancia de la abstracción,

ocultación de la información y de la modularidad en la calidad del software. En algunos aspectos, los métodos de diseño orientados a estructuras de datos tales como Desarrollo de Sistemas Estructurados de Datos (DSED) y Desarrollo de Sistema de Jackson (DSJ) , pueden verse como orientados al objeto.

Durante los años 1980 la rápida evolución de los lenguajes de programación Smalltalk y Ada causaron un creciente interés en DOO. En las primeras discusiones sobre el método para conseguir un diseño orientado al objeto. Abbott mostró "cómo el análisis de sentencias en lenguaje natural del problema y sus solución, pueden usarse como guía para desarrollar la parte visible de un paquete útil (un paquete útil que tenga los datos y procedimientos que operan sobre ellos) y el algoritmo particular para un problema dado". Booch extendió el trabajo de Abbott y ayudó a popularizar el concepto de diseño orientado al objeto. Hoy el DOO se está usando en aplicaciones de diseño de software que van desde animaciones de gráficos por computadora a las telecomunicaciones.

Conceptos del diseño orientado al objeto.

Como otros métodos de diseño, el DOO introduce un nuevo con junto de terminología, notación y procedimientos para la derivación de un diseño del software.

La función del software es relacionada cuando una estructura de datos (de distintos niveles de complejidad) es manipulada mediante uno o más procesos, de acuerdo con un procedimiento definido mediante un algoritmo estático y órdenes dinámicas. Para conseguir un diseño orientado al objeto, se debe establecer un mecanismo para:

1) la representación de la estructura de datos,
2) La especificación del proceso y
3) El procedimiento de llamada.

Un objeto es una componente del mundo real que se transforma en el dominio del software. En el contexto de un sistema basado en computadora, un objeto es normalmente un procedimiento o consumidor de información o un elemento de información. Por ejemplo, objetos típicos pueden ser: máquinas, órdenes, archivos, visualizaciones, conmutadores, señales, cadenas alfanuméricas o cualquier otra persona, lugar o cosa. Cuando un objeto se transforma en una realización software, consta de una estructura de datos privada y procesos, llamados operaciones o métodos, que pueden transformar legítimamente la estructura de datos. Las operaciones contienen el control y las construcciones procedimentales que pueden ser llamadas mediante un mensaje una petición al objeto para que ejecute una de sus operaciones. La realización software de un objeto se muestra en la figura (fig 9.1).

Refiriéndose a la figura, un objeto del mundo real (un diccionario) se transforma en una realización software para un sistema basado en computadora. La realización software de un diccionario exhibe una estructura de datos privada y las operaciones relativas a ella. La estructura de datos puede tomar la forma descrita mediante diagramas de Warnier-Orr de la figura (fig 9.2). Las entradas del diccionario están compuestas de palabras, una guía de pronunciación y una o más definiciones. Un dibujo o diagrama puede estar también contenido dentro de una entrada. El diccionario de objetos también

contiene un conjunto de operaciones que puede procesar los elementos de la estructura de datos. La parte privada de un objeto es la estructura de datos y el conjunto de operaciones para la estructura de datos.
Un objeto tiene también una parte compartida que es su interfaz. Los mensajes se mueven a través de la interfaz y especifican qué operaciones del objeto se desean, pero no cómo se va a realizar la operación. El objeto que recibe un mensaje determina cómo se implementa la operación solicitada.
Definiendo un objeto con una parte privada y dando mensajes para llamar al procesamiento adecuado, se consigue la ocultación de la información; esto es, los detalles de la implementación están ocultos a todos los elementos del programa exteriores al objeto. Los objetivos y sus operaciones dan una modularidad inherente; esto es, los elementos del software (datos y procesos) se agrupan junto con los mecanismos de interfaces bien definidos.
De esta forma se tiene que el análisis orientado al objeto puede describirse de la siguiente forma:

1. El software asignado (o el sistema entero) se describe usando una estrategia informal. La estrategia no es más que una descripción en lenguaje natural de la solución del problema que hay que resolver, mediante el software representado a un nivel consistente de detalle. La estrategia informal puede ser establecida en forma de párrafos sencillos, gramaticalmente correctos.
2. Los objetivos se determinan subrayando cada nombre o cláusula nominal e introduciéndolo en una tabla sencilla. Deben anotarse los sinónimos. Si se requiere que el objeto se implemente como una solución, entonces es parte del espacio de solución; en otros casos, si un objeto es necesario sólo para describir una solución, es parte del espacio del problema.
3. Los atributos de los objetos se identifican subrayando todos los adjetivos y luego asociándolos con sus objetos respectivos (nombres).
4. Las operaciones se determinan subrayando todos los verbos, frases verbales y predicados (una frase verbal indica un test condicional) y relacionando cada operación con el objeto apropiado.
5. Los atributos de las operaciones se identifican subrayando todos los adverbios y luego asociándolos con sus operaciones respectivas (verbos).

Para ilustrar el método de análisis orientado al objeto, considere al siguiente texto:

> El software SCCT recibirá la información de entrada de un lector de código de barras a intervalos de tiempo que estén de acuerdo con la velocidad de la cinta transportadora. Los datos del código de barras se decodificarán en un formato de identificación de caja. El software examinará una base de datos de unas 1000 entradas para determinar la posición del cubo apropiado para la caja que actualmente está en el lector (estación de clasificación). SE usará una lista FIFO para tomar nota de las posiciones de envío de cada caja conforme se mueve por la estación de clasificación.

El software SCCT recibirá también entrada de un tacómetro de pulsos que se usará para sincronizar la señal de control con el mecanismo de envío. Basándose en el número de pulsos que se generarán entre la estación de clasificación y el envío, el software producirá una señal de control a la estación de envío indicando la posición correspondiente de la caja...

Analizando un poco el texto anterior muchos de los nombres subrayados indican que residen en el espacio del problema y n o son necesarios para la implementación del programa. Otros nombres, tales como datos del código de barra, posición del cubo y señal de control, residen en el espacio de la solución e introducen en la tabla de objetos y operaciones. Siguiendo el procedimiento de análisis orientado al objeto, se subrayan a continuación todos lo adjetivos y se colocan en la tabla como se muestra en ella.
Cuando uno de los objetivos representa un elemento de información que debe ser procesado por el programa. Para determinar las funciones de procesamiento, debemos aislar todas las operaciones. Por lo que siguiendo el procedimiento de análisis:

El software SCCT recibirá la información de entrada de un lector de código de barras a intervalos de tiempo conforme a la velocidad de la cinta transportadora. Los datos del código de barras se decodificarán en un formato de identificación de caja. El software examinará una base de datos de unas 1000 entradas para determinar la posición apropiada del cubo para la caja que actualmente está en el lector (estación de clasificación). Se usará una lista FIFO para tomar nota de las posiciones de envío de cada caja conforme se mueve por la estación de clasificación.
El software SCCT recibirá también entrada de un tacómetro de pulsos que se usará para sincronizar la señal de control con el mecanismo de envío. Basándose en el número de pulsos que se generarán entre la estación de ordenación envío, el software producirá una señal de control a la estación de envío indicando la posición correspondiente de la caja...

TABLA
OBJETOS Y OPERACIONES

Objeto	Atributo	Operación correspondiente
Datos	Código de caja	Decodificador
Posición	Caja	Determinar
Señal	Control	Sincronizar, producir
Entrada	Tac. pulso	Recibir
Lista	FIFO	Tomar nota
Posiciones	Envío	Tomar nota
Intervalos	Tiempo	Conformar
Formato	Id. caja	Decodificar
Base de datos	1000 entradas	Examinar
Pulsos	Tac.	Generado

Algunas de las operaciones anotadas pertenecen al espacio de solución (por ejemplo recibir, buscar, producir) mientras que otras son parte del espacio del problema (por ejemplo sincroniza). Las operaciones del espacio de solución se añaden a la tabla de objetos y operaciones.

LECTURA: CONSTRUCCION DE PROTOTIPOS.

El análisis debe ser conducido independientemente del paradigma de ingeniería de software aplicado. Sin embargo, la forma que ese análisis tomará puede variar. En algunos casos es posible aplicar los principios de análisis fundamental y derivar a una especificación en papel del software desde el cual pueda desarrollarse un diseño. En otras situaciones, se va a una recolección de los requerimientos, se aplican los principios de análisis y se construye un modelo de software, llamado un prototipo, según las apreciaciones del cliente y del que lo desarrolla. Finalmente, hay circunstancias que requieren la construcción de un prototipo al comienzo del análisis, puesto que el modelo es el único medio mediante el que los requerimientos pueden ser derivados efectivamente.

El modelo entonces se transforma en una producción de software. En muchos casos (aunque no todos), la construcción de un prototipo, acoplado posiblemente con métodos de análisis sistemáticos, es una forma efectiva de ingeniería del software.

Todos los proyectos de ingeniería del software comienzan con una petición del cliente. La petición puede estar en la forma de una memoria que describe un problema, un informe que define un conjunto de objetivos comerciales o del producto, una Petición de Propuesta formal de una agencia o compañía exterior, o una Especificación del Sistema que ha asignado una función y comportamiento al software, como un elemento de un sistema basado en computadora. Suponiendo que existe una petición para un programa de una de las formas para construir un prototipo del software se aplican los siguientes pasos:

PASO 1. Evaluar la petición del software y determinar si el programa a desarrollar es un buen candidato para construir un prototipo. No todos los programas son adecuados para la construcción de prototipos. Varios factores de candidatura de construcción de prototipos pueden ser definidos: área de aplicación, complejidad, de la aplicación, características del cliente y características del proyecto.

Debido a que el cliente debe interaccionar con el prototipo en los últimos pasos, es esencial que:

1) El cliente participe en la evaluación y refinamiento del prototipo.
2) El cliente sea capaz de tomar decisiones de requerimientos de una forma oportuna.

Finalmente, la naturaleza del proyecto de desarrollo tendrá una fuerte influencia en la eficacia del prototipo.

PASO 2. Dado un proyecto candidato aceptable, el analista desarrolla una representación abreviada de los requerimientos. Antes de que pueda comenzar la construcción de un prototipo, el analista debe representar los dominios funcionales y de información del programa y desarrollar un método razonable de partición.

PASO 3. Después de que se haya revisado la representación de los requerimientos, se crea un conjunto de especificaciones de diseño abreviadas para el prototipo. El diseño debe ocurrir antes de que comience la construcción del prototipo. Sin embargo, el diseño de un prototipo se enfoca normalmente hacia la arquitectura a nivel superior y a los aspectos de diseño de datos, en vez de hacia el

diseño procedimental detallado.
PASO 4. El software del prototipo se crea, prueba y refina. Idealmente, los bloques de construcción de software preexistentes se utilizan para crear el prototipo de una forma rápida. Debido a que tales bloques construidos raramente existen. Alternativamente, pueden usarse herramientas de construcción de prototipos especializadas para asistir al analista/diseñador en la representación del diseño y traducirlo a una forma ejecutable.
Si la implementación de un prototipo que funcione es impracticable, el escenario de construcción de prototipos puede aún aplicarse. Para las aplicaciones interactivas con el hombre, es posible frecuentemente crear un prototipo en papel que describa la interacción hombre-máquina (preguntas, presentaciones, decisiones, etc.) usando una serie de hojas de historia. Cada hoja de historia contiene una representación de una imagen de la pantalla con texto, que describe la interacción entre la máquina y el usuario. El cliente revisa las horas de historia, obteniendo una perspectiva de usuario de la operación del software.
PASO 5. Una vez que el prototipo ha sido probado, se presenta al cliente, el cual "conduce la prueba" de la aplicación y sugiere modificaciones. Este paso es el núcleo del método de construcción de prototipo. Es aquí donde el cliente puede examinar una representación implementada de los requerimientos del programa, sugerir modificaciones que harán al programa cumplir mejor las necesidades reales.
PASO 6. Los pasos 4 y 5 se repiten iterativamente hasta que todos los requerimientos estén formalizados o hasta que el prototipo haya evolucionado hacia un sistema de producción. El paradigma de construcción del prototipo puede ser conducido con uno o dos objetivos en mente:

1) el propósito del prototipo es establecer un conjunto de requerimientos formales que pueden luego ser traducidos en la producción de programas mediante el uso de métodos y técnicas de ingeniería de programación, o
2) el propósito de la construcción del prototipo es suministrar un continuo que pueda conducir al desarrollo evolutivo de la producción del software. Ambos métodos tienen sus méritos y ambos crean problemas.

Métodos y herramientas para la construcción de prototipos.
Para que la construcción de prototipos de software sea efectivo, un prototipo debe desarrollar rápidamente, de forma que el cliente pueda comprobar los resultados y recomendar cambios. Para conseguir una construcción rápida de prototipo, existen tres clases genéricas de métodos y herramientas: Técnicas de la cuarta generación, componentes de software reusables, especificación formal y entornos de construcción de prototipo.
TECNICAS DE LA CUARTA GENERACION. Las técnicas de la cuarta generación (4GT) comprenden un amplio repertorio de lenguajes de informes y preguntas de base de datos, generadores de programas y aplicaciones y otros lenguajes no procedimentales de muy alto nivel. Debido a que las 4GT facilitan al ingeniero de programación general, código ejecutable rápidamente, son ideales para la construcción rápida de prototipos. Desafortunadamente, el dominio de aplicación de las 4GT está actualmente limitado a sistemas de

información comerciales.

COMPONENTES DE SOFTWARE REUSABLES. Otro método para la construcción rápida de prototipos es ensamblar, en vez de construir, el prototipo usando un conjunto de componentes software existente. Una componente software puede ser una estructura de datos (o base de datos) o una componente arquitectónica software (por ejemplo un programa o una componente procedimental (es decir, un módulo). En cada caso, la componente software debe ser diseñada en forma que facilite el ser reusada sin conocer los detalles de su funcionamiento interno. Debe observarse que un producto software existente puede ser usado como un prototipo para un "nuevo, mejorado" producto competitivo. De alguna forma, esto es una forma de reusabilidad para la construcción de prototipos de software.

ESPECIFICACION FORMAL Y ENTORNOS PARA LA CONSTRUCCION DE PROTOTIPOS. Se han desarrollado varios lenguajes de especificación formal para reemplazar las técnicas de especificación en lenguaje natural. Hoy, los desarrolladores de estos lenguajes formales están en el proceso de desarrollar entornos interactivos que:

1) facilite al analista crear interactivamente una especificación basada en el lenguaje de un sistema o software;
2) llame a herramientas automáticas que traduzcan las especificaciones basadas en lenguajes en código ejecutable y
3) faciliten al cliente utilizar el código ejecutable del prototipo para refinar los requerimientos formales.

La forma de especificar tiene mucho que ver con la calidad de la solución. Los ingenieros de software que se han esforzado en trabajar con especificaciones incompletas, inconsistentes o mal establecidas han experimentado la frustación y confusión que invariantemente se produce. Las consecuencias se padecen en la calidad, oportunidad y completitud del software resultante.

Las técnicas de análisis pueden conducir a una especificacion en papel que contenga las descripciones gráficas y el lenguaje natural de los requerimientos del software. La construcción de prototipos conduce a una especificación ejecutable, esto es, el prototipo sirve como una representación de los requerimientos. Los lenguajes de especificación formal conducen a representaciones formales de los requerimientos que pueden ser verificados o analizados.

LECTURA: BASES DE DATOS.

Un sistema de base de datos.

Se puede definir un sistema de base de datos como un sistema de mantenimiento de registros, basado en computadoras, es decir, un sistema que de manera general registra y mantiene información de importancia para la organización donde el sistema opera.

Un sistema de base de datos tiene cuatro componentes principales: datos, hardware, software y usuarios.

Datos

Se refieren a los valores registrados dentro de la base de datos. Es importante distinguir entre "datos" e información" ya que no siempre significan lo mismo. Un dato, simplemente es un valor (número, letras,...) mientras que la información es el significado de esos valores según el sentido que les dé el usuario. Los datos se dividen en una o más bases de datos, pero didácticamente es válido suponer que existe una sola base de datos.

Hardware

Se compone de los volúmenes de almacenamiento secundario (discos, tambores),...) donde se encuentra la base de datos junto con dispositivos asociados (unidades de control, canales,...).

Software

Es un sistema de administración de bases de datos o DBMS y existe entre el usuario y la base de datos física. Este DBMS maneja todas las solicitudes de acceso a la base de datos y protege a los usuarios contra los detalles a nivel hardware apoyando las operaciones del usuario.

Usuarios

Se pueden considerar diversos tipos de usuarios.

Existe el usuario programador de aplicaciones quien escribe programas de aplicación que utilicen bases de datos. Dichos programas recuperan información, crean información, suprimen o cambian información, y en general operan con los datos de todas las maneras posibles.

El usuario final es quien accesa la base de datos desde una terminal. Dicho acceso lo realiza a través de un lenguaje de consulta con el cual puede realizar todas las funciones de recuperación, creación, modificación y supresión de la información contenida en la base de datos.

El usuario administrador de Bases de Datos o DBA es quien organiza el sistema para un desempeño adecuado y le hace los ajustes necesarios al cambiar los requerimientos.

Objetivos de los sistemas de bases de datos

El objetivo primordial de un sistema de base de datos es proporcionar a la empresa un control centralizado de sus datos. Dicho control trae consigo diversas ventajas:

Control de la Redundancia.

Se evita que cada aplicación tenga sus propios archivos

(Redundancia=Información repetida). La Redundancia propicia un desperdicio del espacio de almacenamiento. En ocasiones, la redundancia no se elimina totalmente (por rabones técnicas o comerciales) sin embargo se controla y el sistema debe responsabilizarse de propagar las actualizaciones que se le hagan a los datos.

Evitar Inconsistencias.
Cuando en un sistema de bases de datos, existe redundancia, hay duplicidad de datos. Esta duplicidad puede originar que en determinado momento una actualización no se propague correctamente y se dé la inconsistencia de datos. En este momento, la información no será totalmente confiable, ya que, datos repetidos no son concordantes y la base de datos es inconsistente. Con lo anterior, se puede decir que la inconsistencia se puede evitar si se controla la redundancia.

Datos Compartidos.
Se pueden tener aplicaciones que compartan los datos de la base de datos, e incluso, aplicaciones que operen con los mismos datos almacenados. Con ellos, nuevas aplicaciones pueden realizarse sin tener que crear nuevos almacenamientos.

Seguridad.
Al tener el DBA completa jurisdicción de los datos se asegura que el único medio de accesar la base de datos sea a través de los canales establecidos, y que se definan controles de autorización que se aplican cada vez que se intenta el acceso a datos sensibles (aquellos datos cuyo valor es importante y que solo puede ser cambiado bajo condiciones muy particulares).

Integridad.
Se garantiza que los datos de la base de datos son exactos, esto es, no se presenta inconsistencia entre dos entradas que representan la misma cosa.

Independencia de los datos.
Para evitar que las aplicaciones dependan de la manera como los datos se organizan en el almacenamiento secundario y la manera como se accesan no sea dependiente de la aplicación. Cuando no existe independencia de datos es imposible cambiar la estructura de almacenamiento o la estrategia de acceso sin afectar la aplicación.

Modelos de Bases de Datos.
El modelo de datos es un grupo de herramientas conceptuales para describir datos, sus relaciones, su semántica y sus limitantes.
En general, los modelos de datos se pueden dividir en tres grupos:
Modelos lógicos basados en objetos
Su característica principal es que permiten estructuras flexibles y permiten especificar claramente las limitantes de los datos. Existen más de treinta modelos distintos de este tipo entre los cuales están el Binario, Semántico de Datos, Entidad-Relación, etc. El modelo Entidad-Relación es el más representativo de los modelos lógicos basados en objetos. Dicho modelo, se basa en un conjunto de

objetos básicos llamados entidades y de las relaciones entre estos objetos. Se manejan conceptos como Entidad y atributos.
Una Entidad es un objeto, persona, animal o cosa de interés para la comunidad de usuarios acerca de la cual es sistema de bases de datos debe mantener, correlacionar o desplegar información. Un atributo es una característica o cualidad de una entidad, debe ser de interés y estar dentro del alcance del sistema.
Lo que distingue a una entidad de otra es el conjunto de atributos que la describen. Por ejemplo los atributos nombre y dirección describen a un usuario especifico en una biblioteca.
Varias entidades pueden relacionarse entre sí mediante una relación o asociación, por ejemplo, la relación Libro-Autor asocia a un libro con cada uno de los autores que tiene (así pues, es posible tener un conjunto de entidades y un conjunto de relaciones, los cuales están constituidos por todas las entidades y todas las relaciones respectivamente).

La estructura lógica general de una base de datos puede expresarse gráficamente por medio de un diagrama Entidad-Relación.
Modelos Lógicos Basados en Registros
A diferencia de los modelos de datos basados en objetos, estos especifican tanto la estructura lógica general de la base de datos como una descripción en un nivel mas alto de la implantación. Estos modelos, no permiten especificar en forma clara las limitantes de los datos. Los modelos de este tipo con mayor aceptación son:
Modelo Relacional
Modelo en el que los datos se representan por medio de una serie de tablas, cada una de las cuales tiene varias columnas con nombres únicos (atributos). Cada renglón de las tablas es llamado Registro o tupla; entonces, un registro representa un elemento de la tabla con diversos atributos.
La cardinalidad de una tabla queda definida como el número de registros (renglones) que tiene; y el orden de la tabla queda definido como el número de columnas o atributos de la misma.
Por ejemplo en la tabla B:

Tabla A

nombre	calle	ciudad	número
Hugo	Central	México	113
Paco	Azafrán	Puebla	129
Luis	Sur	Guadalajara	317
Pedro	Norte	Monterrey	113
Pedro	Norte	Monterrey	317

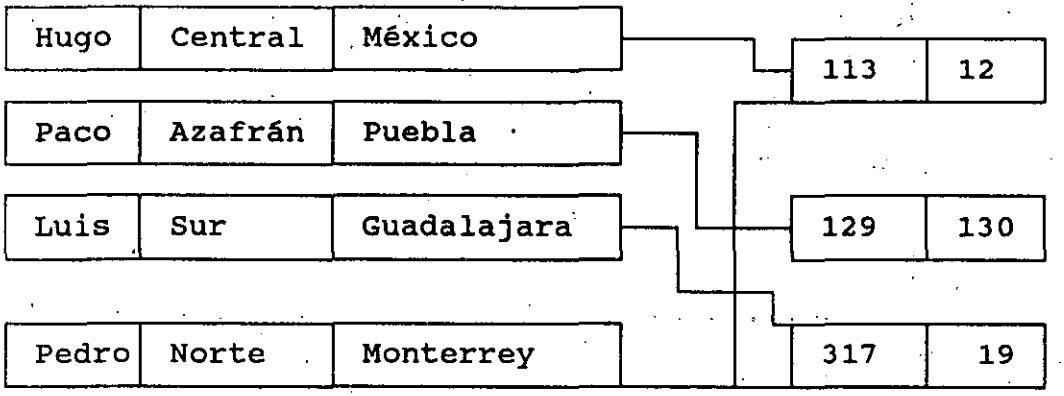
Tabla B

número	cuota
113	12
129	130
317	19

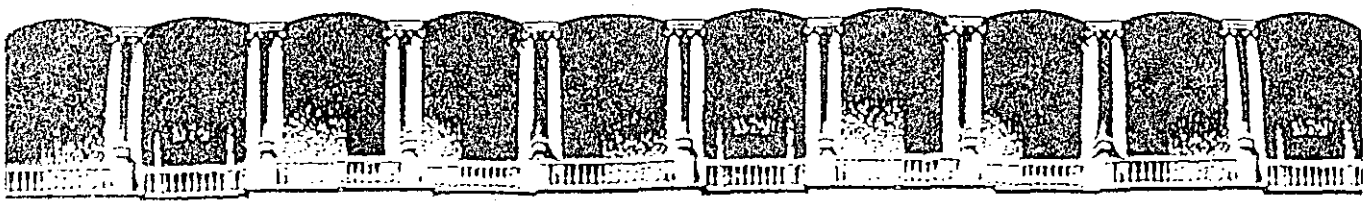
se tiene tres tuplas o registros y dos columnas o atributos, por lo tanto la tabla tiene orden 2 y cardinalidad 3.

Modelos de Red.
Los datos en el modelo de Red se representan por medio de conjuntos de registros (en el sentido de la palabra en Pascal o PL/1) y las relaciones entre los datos se representan con ligas, que pueden considerarse como apuntadores. Los registros de la base de datos se organizan en forma de conjuntos de gráficas arbitrarias. La

siguiente figura es un ejemplo de una base de datos de red.



Modelos Físicos de los Datos
Sirven para describir los datos en un nivel más bajo. A diferencia de los modelos lógicos de los datos, son muy pocos los modelos físicos utilizados. Los más conocidos son:
El modelo unificador.
La memoria de cuadros.



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

ANALISIS Y DISEÑO DE SISTEMAS DE PROCESAMIENTO DE DATOS.

TEMA 4

ABRI-MAYO, 1992.

Tema 4. Fundamentos del Diseño de Software

El diseño es el primer paso en la fase de desarrollo de cualquier sistema, se define como: "El proceso de aplicar distintas técnicas y principios con el propósito de definir un dispositivo, proceso o sistema con los suficientes detalles como para permitir su realización física".

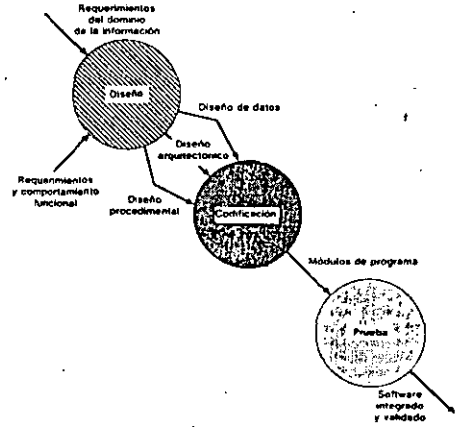
Hay que tener en cuenta que el objetivo del diseñador es producir un modelo de una entidad que será construida posteriormente. El proceso para el desarrollo de un modelo combina: intuición y criterios, es decir, se basa en la experiencia para construir entidades similares, principios que guían la forma en la que se desarrolla el modelo, criterios que facilitan discernir sobre la calidad y el proceso de representación del diseño final.

Subtema 4.1 La fase de desarrollo y el diseño de software

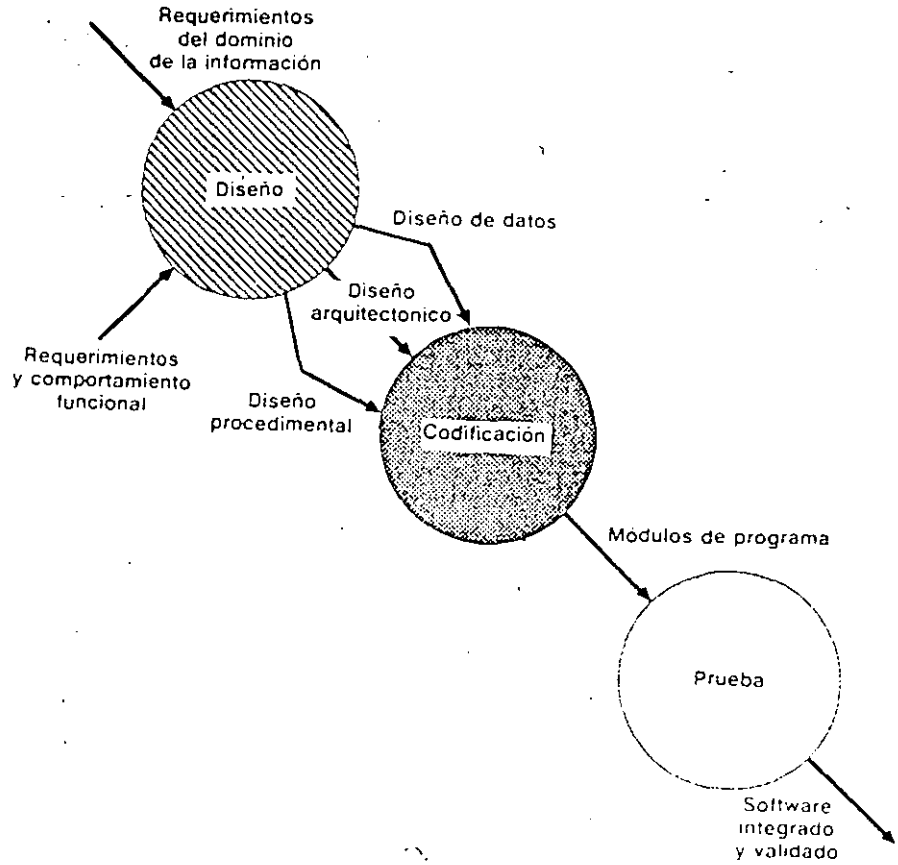
Una vez establecidos los requerimientos del software, la fase de desarrollo comprende tres pasos distintos:

1. Diseño
2. Generación de código
3. Prueba

Cada paso transforma la información de tal forma que finalmente se obtiene un software para computadora válido. En este tema explicaremos el primer paso "diseño" y posteriormente se desarrollaron los siguientes.



El flujo de información que alimenta la etapa de diseño se ve en la figura anterior En esta se aprecia las entradas las cuales son: los requerimientos del programa, los requerimientos funcionales y de comportamiento. Usando alguna de las distintas metodologias de diseño se realiza el diseño de datos, el diseño arquitectónico y el diseño procedimental, que posteriormente explicaremos.


Requerimientos
del dominio
de la información
Diseño
Diseño de datos
Diseño
arquitectonico
Requerimientos
y comportamiento
funcional
Diseño
procedimental
Codificación
Módulos de programa
Prueba
Software
integrado
y validado

Cabe hacer notar que el diseño es el lugar en donde se asienta la calidad del desarrollo del programa, nos da las representaciones del software que pueden establecerse para conseguir un producto con calidad.

Subtema 4.2El proceso de diseño

El diseño de programas es un proceso mediante el cual se traducen los requerimientos en una representación del software.

Desde el punto de vista de una gestión de proyecto, el diseño del software se realiza en dos pasos:

1.Diseño preliminar: que se refiere a la transformación de los requerimientos en datos y arquitectura del software

2. Diseño en detalle: que se enfoca hacia los refinamientos de la representación arquitectónica que conduce a una estructura de datos detallada y a representaciones algoritmicas del software.

Hablaremos ahora acerca del diseño y calidad de software que es esencial para el logro de un buen producto.

Diseño y calidad del software

Para evaluar la calidad de una representación de diseño presentamos los siguientes criterios:

Un diseño debe:

- exhibir una organización jerárquica
(que haga uso inteligente de control entre los elementos del software)

- ser modular
(esto es, el software debe estar particionado lógicamente en elementos que realicen funciones y subfunciones especificas)

- contener una representación distinta y separada de los datos y los procedimientos.

- conducir a módulos que muestren características funcionales independientes

- derivarse usando un método repetitivo
(que esté conducido por la información obtenida durante el análisis de requerimientos de software)

Subtema 4.3 Fundamentos del diseño

Se ha mencionado ya la importancia que representa la elaboración de un buen diseño, para que esto se lleve a cabo de manera eficaz, vamos a tocar algunos aspectos fundamentales.

El establecimiento de conceptos fundamentales en el diseño del software, se ha dado desde hace tres décadas, los cuales ayudan al ingeniero en software a responder a las siguientes preguntas:

- Que criterios pueden usarse para subdividir el software en componente individuales?

- Como se separan los detalles de una función o una estructura de datos de una representación conceptual del software?

- Existen criterios uniformes que definen la calidad técnica de un diseño de programas?

Trataremos algunos de estos conceptos, sobre todo aquellos que respondan a las preguntas antes mencionadas. Comenzaremos con:

Refinamiento:

Es una temprana estrategia de diseño descendente propuesta por Niklaus Wirth, "En cada paso del refinamiento, una o varias instrucciones del programa dado se descomponen en más instrucciones detalladas. Esta descomposición sucesiva o refinamiento de especificaciones termina cuando todas las instrucciones están expresadas en términos del computador usado o lenguaje de programación".

El refinamiento es realmente un proceso de elaboración. Se comienza con una declaración de la función (o descripción de la información) que se define en el nivel superior de abstracción, es decir, la declaración describe la función o información conceptualmente, pero no se da información sobre el funcionamiento interno de la función o la estructura interna de la información, el refinamiento hace que el diseñador amplié la declaración original, dando mas y mas detalles conforme se produzcan sucesivos refinamientos.

Arquitectura de software

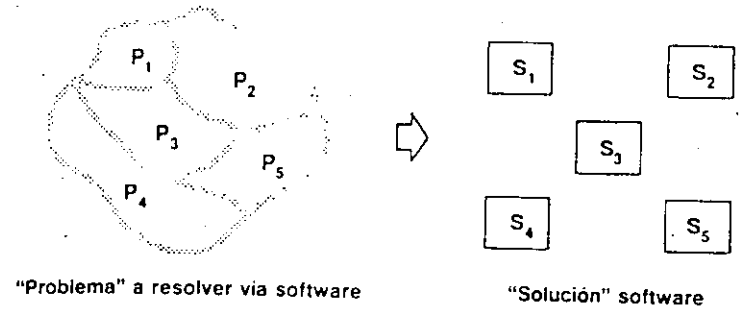
Otro concepto es la arquitectura del software y esto alude a dos características importantes en un programa:

1. la estructura jerárquica de los componentes procedimentales (módulos)

2. la estructura de datos

La arquitectura de software se deriva mediante un proceso de partición, que relaciona a los elementos de una solución del software, con partes de un problema real definido implícitamente durante el análisis de requerimientos. La evolución del software y la estructura

de datos comienza con una definición del problema. La solución ocurre cuando cada parte del problema se resuelve mediante uno o mas elementos de software, en la siguiente figura se muestra la representación de esta transición entre el análisis de requerimientos de software y el diseño.



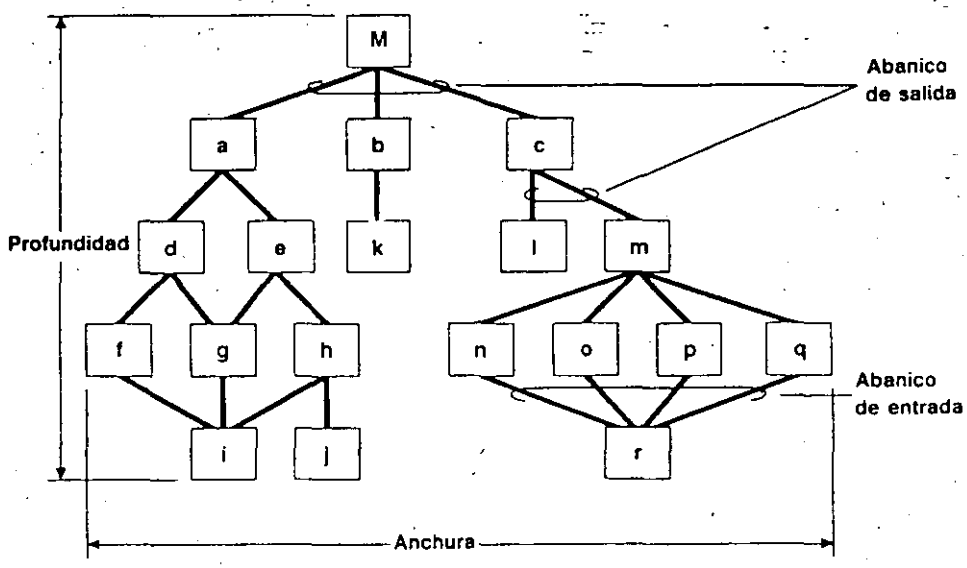
Estructura de un programa

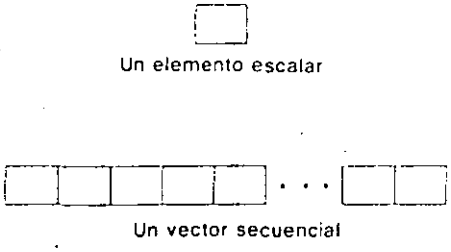
Un concepto mas es la estructura del programa que representa la organización jerárquica de las componentes del programa (módulos) e implica una jerarquía de control. Para representarla se utilizan notaciones diferentes, siendo la mas común el diagrama de árbol, llamado diagrama de estructura. En la figura siguiente se muestra un diagrama de árbol, con una profundidad y anchura que indican el numero de niveles de control y expansión global. Notese que el abanico de salida es una medida del numero de módulos que están directamente controlados por otros módulos, el abanico de entrada indica cuantos módulos controlan directamente a un modulo dado.

Estructura de datos

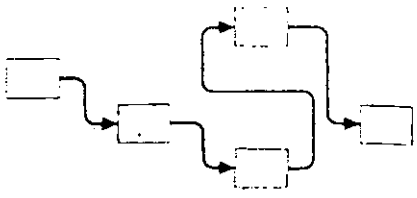
Siguiendo con los conceptos, uno mas es la representación de la relación lógica entre los elementos individuales de datos, conocidas como estructura de datos, la cual es muy importante y dicta la organización, los métodos de acceso, grado de asociatividad y alternativas de procesamiento para la información.

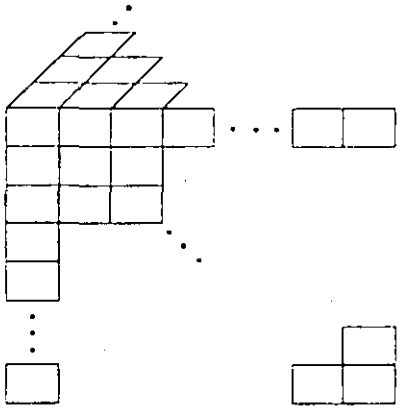
Algunas estructuras de datos clásicas, se muestran en la figura. El elemento escalar representa un elemento simple de información direccionado mediante un identificador. El vector es un conjunto continuo de elementos escalares que pueden ser de una, dos o más dimensiones. La lista enlazada esta organizada con elementos escalares, no continuos y pueden ser procesados como una lista. Cada nodo o elemento, contiene una organización propia y uno o más apuntadores que indican la dirección de memoria del siguiente nodo de la lista.

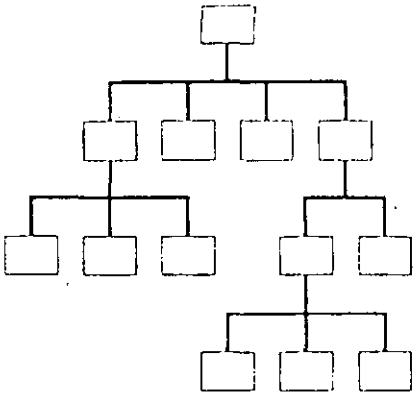



Un elemento escalar

Un vector secuencial


Una lista enlazada


Un espacio *n*-dimensional


Un árbol jerárquico

Procedimientos de software

El hablar de procedimiento de software, se enfoca sobre los detalles de procesamiento de cada módulo individualmente. El procedimiento debe dar una especificación precisa del procesamiento, incluyendo secuencia de sucesos, puntos de decisiones exactos, operaciones repetitivas e incluso organización/estructura de datos.

Existe una relación entre estructura y procedimiento, el procesamiento indicado por cada módulo debe incluir una referencia a todos los módulos subordinados del modulo que se describe.

Modularidad

Cuando se tiene un problema, para resolverlo, es más fácil dividirlo en partes manejables llamadas módulos, el modulo es un elemento con nombres y direcciones separadas, los cuales se integran para satisfacer los requerimientos del problema.

Es posible deducir que si subdividimos el software indefinidamente, el esfuerza que se requiere para desarrollar será significativamente pequeño. Sin embargo, hay otros factores que hacen que esta conclusión no sea del todo aceptada.

Abstracción

Al considerar una solución modular a cualquier problema, se pueden formular muchos niveles de abstracción, la cual facilita al diseñador representar un objeto de datos a diferentes niveles de detalle y especificarlo en el contexto de las operaciones que se le aplican. En el nivel superior de abstracción, se establece la solución en términos amplios usando el lenguaje del entorno del problema, en los niveles inferiores se toma una orientación más procedimental y se establece la solución en forma que pueda implementarse directamente.

Ocultamiento de la información

Al especificarse o diseñarse los módulos se debe hacer de tal manera que la información contenida dentro de un módulo sea inaccesible a otros módulos que no necesiten tal información.

Subtema 4.5 Diseño modular efectivo

En el subtema anterior tratamos acerca de la modularidad, la cual se ha convertido en un enfoque aceptado en todas las disciplinas de ingeniería. Algunas ventajas que proporciona el diseño modular es que reduce la complejidad, facilita los cambios y da como resultado una fácil implementación, posibilita además el desarrollo paralelo de diferentes partes de un sistema.

Tipos de módulos

Los módulos se dividen en categorías, las cuales son:

- módulo secuencial
(se llama y ejecuta sin interrupción aparente, son los más comunes y se categorizan como macros en tiempo de compilación y subprogramas convencionales, conocidas como funciones, procedimientos o subrutinas)

- módulo incremental
(se puede interrumpir antes de la terminación por software de aplicación y reestablecido en el punto de interrupción, se les llama comúnmente corrutinas, mantienen un apuntador de entrada que permite al módulo reestablecer el punto de interrupción)

- módulo paralelo
(se ejecuta simultáneamente con otro módulo en multiprocesamientos concurrentes, se encuentran en cálculos de alta velocidad que necesitan dos o mas procesadores trabajando en paralelo).

Independencia funcional

La comunicación entre módulos no debe ser excesiva y se deben desarrollar con una clara función de lo que va a hacer, para llegar a lo que se conoce como independencia funcional.

Siempre es preferible diseñar software de forma que cada módulo se enfoque a una subfunción específica de requerimientos y se tenga una interface sencilla.

Esta independencia se mide con dos criterios cualitativos:

- Cohesión

- Acoplamiento

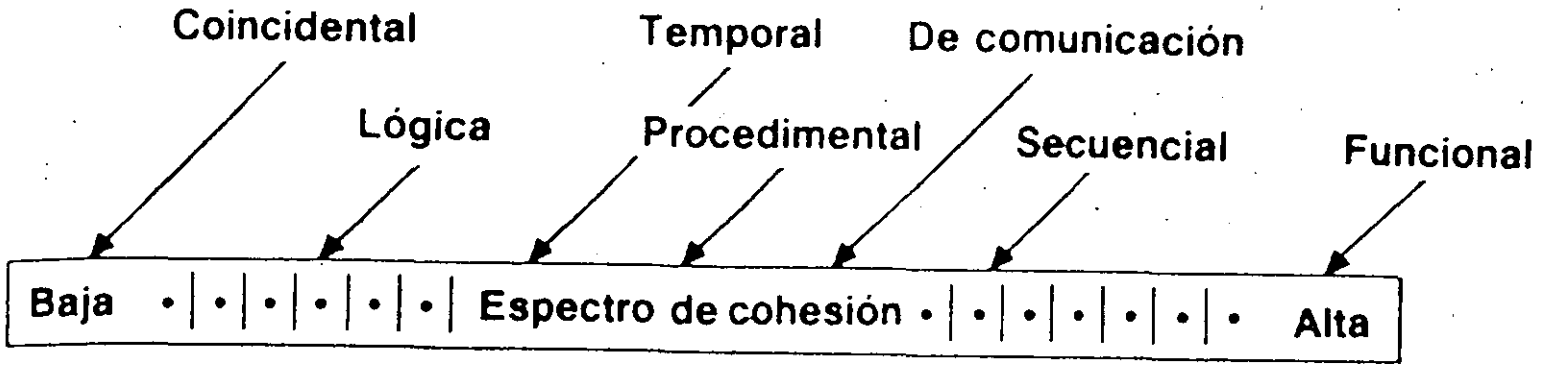
Cohesión:

Al hablar de un módulo coherente, se hace referencia a una tarea sencilla en un procedimiento de software y requiere poca interacción con procedimientos que ejecutan en otras partes de un programa, es decir, un módulo coherente debe hacer sólo una cosa.

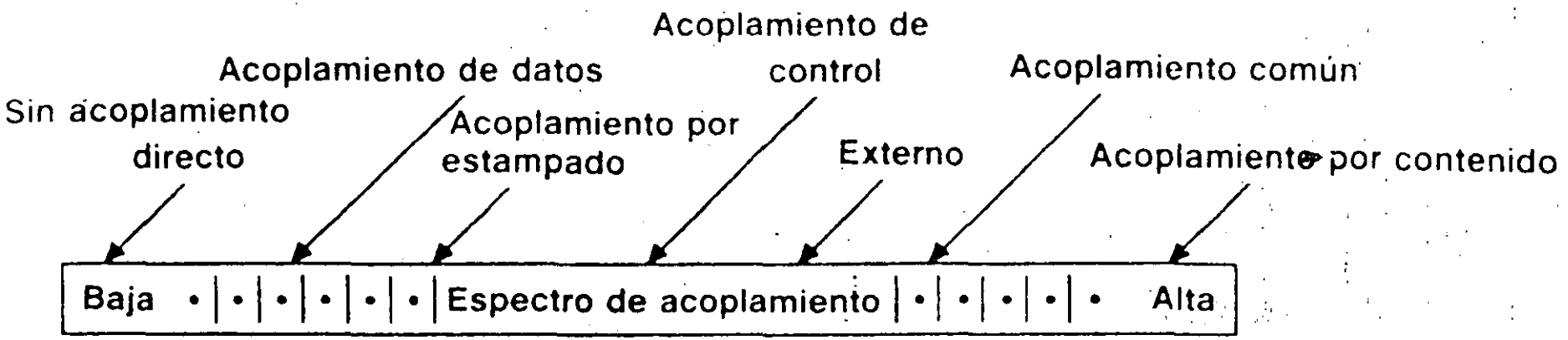
La cohesión puede ser representada como el "espectro" mostrado en la siguiente figura, en la cual se nota los diferentes tipos de cohesión que se tienen de acuerdo al nivel.

En la medida de la fortaleza funcional de un módulo, si se ejecutan un conjunto de tareas relacionadas con otras debilmente, se habla de **cohesión coincidental**, si se relacionan lógicamente es **cohesión lógica** y cuando se contienen tareas relacionadas por el hecho de que

Una medida de la fortaleza funcional relativa de un módulo



Una medida de la interdependencia entre módulos de software



se deben ejecutar dentro de un mismo tiempo, se exhibe **cohesión temporal**. Se puede apreciar también que la escala de la cohesión no es lineal, siempre se intenta conseguir un gran cohesión, aunque el punto medio del rango del espectro es aceptable.

Se habla de **cohesión procedimental,** cuando los elementos de procesamiento de un módulo están relacionados y deben ejecutarse en un orden especifico, y cuando los elementos de procesamiento se concentran sobre una área de un estructura de datos, se presenta **cohesión de comunicación.**

No es necesario determinar el nivel preciso de cohesión. Es importante buscar una cohesión alta y reconocer la cohesión baja de forma que el diseño del software pueda modificarse, de forma que contenga una mayor independencia funcional.

Acoplamiento

Otro criterio de medida de independencia, el cual es acoplamiento, que se puede definir como la medida de la interconexión entre módulos en una estructura de un programa, que depende de la complejidad de la interface entre módulos, del punto en el que se hace una entrada o referencia a un módulo y de los datos que pasan a través de la interface. Al igual que la cohesión se puede representar por un espectro, en el que se marcan los tipos de acoplamiento.

El **acoplamiento directo** se da cuando dos módulos están relacionados en forma directa, el **acoplamiento de datos** es cuando un modulo B es subordinado de un modulo A y esta accesible mediante una lista de argumentos convencionales, a través de los cuales pasa datos. Al pasar una porción de una estructura de datos mediante una interface de módulo que es una variación del acoplamiento de datos, se le conoce como **acoplamiento por estampado**. A niveles moderados, el acoplamiento se caracteriza por el paso de control entre módulos, en su forma sencilla, el control se pasa mediante un indicador sobre el que se toman las decisiones en un módulo subordinado o superior.

Se producen niveles altos de acoplamiento, cuando los módulos están ligados a un entorno externo al software. El **acoplamiento externo** es esencial, pero debe limitarse a un pequeño numero de módulos dentro de una estructura. Así mismo a niveles altos de acoplamiento se encuentra el **acoplamiento común,** que se da cuando varios módulos se hace referencia a un área de datos global. Y el mayor grado de acoplamiento se da en el **acoplamiento por contenido,** cuando un módulo hace uso de información de control o de datos mantenidos dentro de los limites de otro módulo.

Subtema 4.6 Diseño de datos

Una de las mas importantes de las tres actividades de diseño que se realizan durante la ingeniería de software, es el diseño de datos.

Esta actividad como ya mencionamos es la primera en el diseño de datos y se encarga principalmente de seleccionar las representaciones lógicas de los objetos de datos, conocidas

como estructuras de datos, las cuales se identifican durante las fases de definición y especificación de requerimientos.

Independientemente de las técnicas de diseño usadas, los datos bien diseñados nos pueden conducir a una mejor estructura del programa, modularidad y reducción de la complejidad de los procedimientos.

Al especificar los datos tenemos tener presentes los siguientes principios:

1. Los métodos de análisis sistemático, aplicados al software deben también aplicarse a los datos: deben desarrollarse y revisarse las representaciones de flujo y estructura de datos, considerar organizaciones alternativas y evaluar el impacto del diseño de datos sobre el diseño de software, es decir, si por ejemplo, la especificación de una estructura de lista doblemente ligada puede satisfacer agradablemente a los requerimientos de los datos, pero conducir a un diseño de software difícil de manejar.

2. Deben identificarse todas las estructuras de datos y operaciones que han de ejecutarse sobre cada una de ellas: Si por ejemplo, se considera una estructura de datos formado por un conjunto de diversos elementos de datos. La estructura será manipulada por varias funciones principales, para la evaluación de la operación ejecutada sobre esa estructura, se define un tipo abstracto de datos para usarlo en el diseño subsecuente del software, que puede llegar a simplificar el diseño de software.

3. Deben establecerse y usarse un diccionario de datos para definir el diseño de los datos y el software: En el diccionario de datos se representa explícitamente las relaciones entre los datos y las ligaduras de los elementos de una estructura de datos.

4. Las decisiones de diseño de datos a bajo nivel deben retrasarse hasta las ultimas etapas del proceso de diseño: Puede usarse un proceso de refinamiento sucesivo para el diseño de datos, es decir, definir una organización global de los datos durante el análisis de requerimientos, refinarse durante el diseño preliminar y especificarse en detalle durante el diseño detallado.

5. La representación de una estructura de datos debe ser conocida sólo por los módulos que hagan un uso directo de los datos contenidos dentro de la estructura: Este punto hace referencia a los conceptos antes vistos de ocultación de información y acoplamiento, los cuales proporcionan un aspecto importante en la calidad del diseño de software.

6. Debe desarrollarse una biblioteca de estructuras de datos útiles y de las operaciones que pueden aplicarse a ellas: Una biblioteca de estructuras de datos, pueden reducir el trabajo de especificación y diseño de los datos.

7. El diseño de software y el lenguaje de programación deben soportar la especificación y realización de tipos abstractos de datos: La implementación de una estructura de

datos sofisticada puede hacerse excesivamente difícil si no hay forma de realizar una especificación directa de la estructura.

Los principios descritos forman la base de un método de diseño de datos, que puede ser integrado en la fase de definición y desarrollo del proceso de ingeniería de software, dado que una clara definición de la información es esencial para desarrollar bien el software.

Subtema 4.7 Diseño Arquitectónico

El diseño principal del diseño arquitectonico es desarrollar una estructura de programa modular y representar las relaciones de control entre los módulos, además mezcla la estructura de programas y la estructura de datos, define las interfaces que facilitan el flujo de los datos a lo largo del programa.

Subtema 4.8 Diseño Procedimental

Una vez que se ha establecido la estructura del programa y de los datos, se realiza el diseño procedimental. En este se realiza la especificación procedimental que requiere definir los detalles algorítmicos que deben establecerse en un lenguaje natural, existe desafortunadamente un pequeño problema, en el diseño procedimental debe especificarse los detalles de los procedimientos sin ambigüedad y la falta de ambigüedad en un lenguaje natural no es corriente.

Programación Estructurada

En el desarrollo procedimental se requiere del uso de un lenguaje, como ya mencionamos, sus fundamentos se formaron a principios de los años 60, en el cual se propuso el uso de un conjunto de construcciones lógicas con las que podría formarse cualquier programa. Cada construcción tenía una estructura lógica predecible, se entraba por el principio y se salía por el final, y facilitaba al lector seguir más fácilmente el flujo procedimental. Las construcciones introducidas anteriormente en la representación de datos, son la secuencia, condición y repetición.

Secuencia: la secuencia, implementa los paso de procesamiento esenciales en la especificación de cualquier algoritmo.

Condición: la condición da la facilidad para seleccionar un procedimiento basado en alguna ocurrencia lógica.

Repetición: la repetición suministra el ciclo.

Estas tres construcciones son fundamentales en la programación estructurada, la cual es una técnica de diseño importante en el campo más amplio de lo que hemos aprendido de ingeniería del software.

Cualquier programa, independientemente del área de aplicación o complejidad técnica, puede diseñarse e implementarse usando sólo las tres construcciones estructuradas.

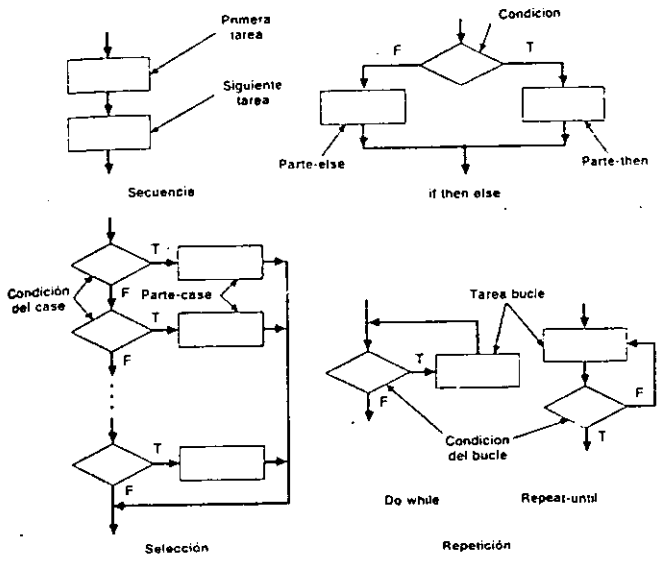
Herramientas gráficas de diseño

Alguien alguna vez dijo: un dibujo vale mas que mil palabras, pero es importante saber que dibujo y que miles de palabras"... No hay duda que las herramientas gráficas, tales como los diagramas de flujo o de cajas, dan una excelente forma gráfica de describir fácilmente los detalles procedimentales.

Diagrama de flujo:

El diagrama de flujo es la representación gráfica más usada para el diseño procedimental. Pero también es del que mas se ha abusado. Es un gráfico muy sencillo, una caja indica un paso de procesamiento, un rombo representa una condición lógica y las flechas muestran el flujo de control.

La siguiente figura muestra las tres construcciones de programación estructurada.



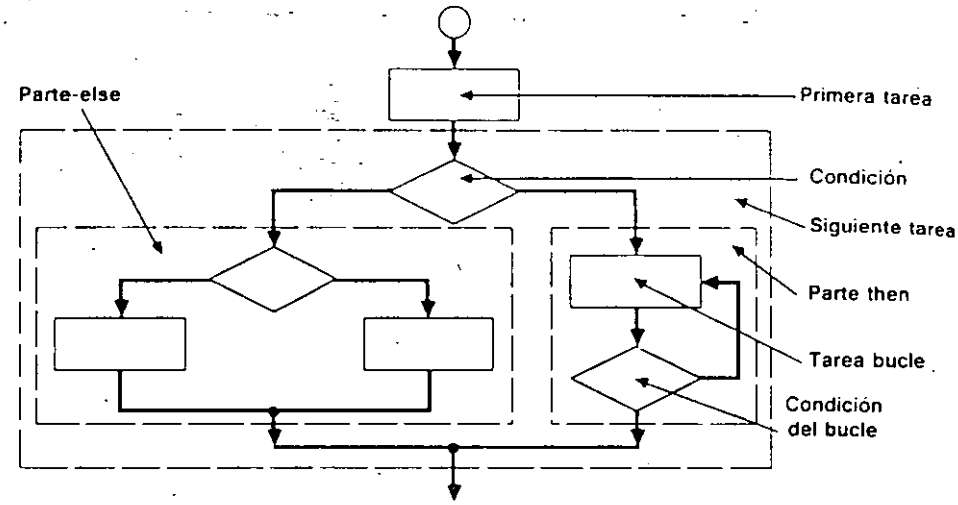
La secuencia esta representada por dos cajas de procesamiento conectadas por una línea de control.

La condición también llamada IF-THE-ELSE, se dibuja como un rombo de decisión, el cual, si es verdad, hace que se realice el procesamiento de la parte then y si es falso llama al procesamiento de la parte del ELSE.

La repetición se representa usando dos formas algo diferentes. La forma DO-WHILE prueba una condición y ejecuta repetidamente una tarea mientras la condición sea verdadera. La

forma REPEAT-UNTIL ejecuta primero el ciclo, luego checa la condición y repite la tarea hasta que falla la condición.

Estas construcciones estructuradas pueden estar anidadas unas en otras como se muestra en la siguiente figura.



Una herramienta mas en el diseño gráfico es el diagrama de cajas, que surgió del deseo procedimental que no permitiera la violación de las construcciones estructuradas, y tiene las siguientes características:

- Dominio funcional:
el ámbito de la repetición o un IF-THEN-ELSE, esta bien definido y claramente visible como una representación gráfica.

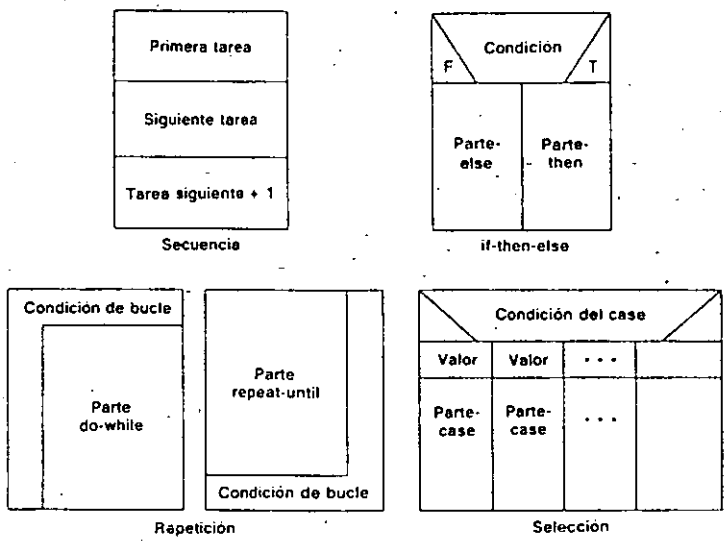
- Imposible transferencia arbitraria de control

- Determinación de los ámbitos de los datos locales y/o globales, estos pueden determinarse fácilmente.

- Representación fácil de la recursividad.

En la siguiente figura se muestra, la representación gráfica de construcciones estructuradas usando diagramas de cajas, en esta se ve que el elemento fundamental del diagrama es la caja, para representar la secuencia se conectan dos cajas seguidas. para representar el IF-THEN-ELSE, se pone una caja de la parte THEN y otra de la parte ELSE, a continuación de la caja de condición. La repetición se dibuja con una caja interior que encierra el proceso que ha de repetirse (parte del DO-WHILE o parte REPEAT-UNTIL).

Y la selección se representa usando la forma gráfica mostrada al final.



Las llamadas a los módulos subordinados puede representarse mediante una caja con el nombre del módulo encerrado dentro de un óvalo.

Lenguaje de diseño de programas

Para el diseño de programas se requiere de un lenguaje, llamado pseudocódigo, que es un lenguaje mezclado que utiliza el vocabulario de un lenguaje por ejemplo el inglés, y la sintaxis general de otro por ejemplo, un lenguaje de programación estructurada.

La diferencia entre el lenguaje de diseño de programas y un lenguaje de programación de alto nivel real se refiere al uso del texto narrativo insertado directamente dentro de las sentencias del lenguaje de diseño de programas.

Independientemente de su origen, un lenguaje de diseño debe tener las siguientes características:

- Sintaxis fija de palabras clave que den todas las construcciones estructuradas, declaraciones de datos y características de modularidad

- Sintaxis libre en lenguaje natural que describa las características del procesamiento

- Facilidad para declaración de datos que incluyan las estructuras de datos sencillas (escalares y arreglos) y complejas (listas ligadas o árboles).

- Definición de subprogramas y técnicas de llamada que soporten los distintos modos de descripción de interfaces.

La sintaxis básica del LDP debe incluir:

- definición de subprograma

- descripción de interfases

- declaración y tipificación de datos

- técnicas para estructurar en bloques

- instrucciones de condición

- instrucciones repetitivas

- instrucciones de entrada/salida.

El formato y semántica de un LDP es:

- Instrucción que permite representar las estructuras de datos locales y globales:

TYPE [nombre-variable] IS [calificador1] [calificador2]

donde:

[nombre-varible]	es una varible declarada dentro de un módulo o declarada para uso global o entre módulos.
[calificador1]	indica la estructura de datos especifica e incluye palabras reservadas como: SCALAR, ARRAY, LIST, STRING, STRUCTURE
[calificador2]	indica como se utilizan los nombres de variables en el contexto de un módulo o programa.

- Elementos procedimentales estructurados en bloques, es decir, pude definirse un pseudocódigo en bloques que se ejecute como una entidad simple.

Un bloque esta delimitado de la siguiente manera:

BEGIN [nombre-bloque]

[sentencias de pseudocódigo];

END

donde, [nombre-bloque] puede usarse para dar un nodo de referencia subsecuente a un bloque y [sentencias en pseudocódigo] son una combinación de otras instrucciones en LDP.

- Instrucción de condición en LDP, tiene la forma clásica if-then-else.

IF [descripción-condición]

THEN [bloque o sentencia de pseudocódigo];

ELSE [bloque o sentencia de pseudocódigo];

ENDIF

donde, [descripción-condición] indica la decisión lógica que debe hacerse para llamar a la parte de procesamiento THEN o ELSE.

- Instrucción de selección, es un conjunto de IF anidados, esta instrucción compara un parámetro especifico, la variable del case, con un conjunto de condiciones. Cuando se satisface una condición, se llama a un bloque o sentencia individual en pseudocódigo. Su representación es:

CASE OF [nombre-variable-case]

WHEN [condición1-case] SELECT [bloque o sentencia de pseudocódigo];

WHEN [condición2-case] SELECT [bloque o sentencia de pseudocódigo];

....

WHEN [ultima-condición-case] SELECT [bloque o sentencia de pseudocódigo];

DEFAULT [fallo o error del case. bloque o sentencia de pseudocódigo];

ENDCASE

- Instrucción de repetición en LDP, incluye ciclos pre-test y post-test así como un ciclo con índice.

DO WHILE descripción-condición

bloque o sentencia en pseudocódigo

ENDDO

REPEAT UNTIL descripción-condición

bloque o sentencia en pseudocódigo

ENDREP

DO FOR [índice = lista-índice, expresión o secuencia]

bloque o sentencia en pseudocódigo

ENDFOR

- Los subprogramas y sus correspondientes interfases se definen usando las siguientes instrucciones del LDP.

PROCEDURE [nombre-subprograma][atributos]

INTERFACE [lista-argumentos]

[bloques y/o sentencias en pseudocódigo];

donde, [atributos] de un subprograma describe sus características de referencia (internas o externas) y otros atributos dependientes de la implementación (lenguaje de programación) si los hubiera INTERFACE se utiliza para especificar una lista de argumentos del módulo que contiene los identificadores para la información de llegada y salida.

- Especificaciones de entrada/salida, dependen mucho de los lenguajes de diseño, las mas comunes son:

READ/WRITE TO [dispositivo] [lista E/S]

donde dispositivo indica el dispositivo físico de E/S y lista E/S contiene las variables que se transmiten.

Comparación de notaciones de diseño

Cualquier notación para el diseño procedimental, si se usa correctamente, puede ser de ayuda muy valiosa en el proceso de diseño, es por esto que cualquier comparación debe basarse en esta premisa, recíprocamente, incluso la mejor notación, si se aplica mal, añade poco a la compresión. Tomando en consideración lo anterior podemos aplicar comparaciones a las notaciones:

Modularidad:

Una notación de diseño debe soportar el desarrollo de software modular.

Simplicidad global:

Una notación de diseño debe ser relativamente sencilla de aprender, relativamente fácil de usar y generalmente fácil de leer.

Facilidad de edición:

La facilidad con la que una representación de diseño pueda ser editada, puede ayudar a facilitar cada uno de estos pasos de la ingeniería de software.

Legible por la máquina:

Una notación que pueda ser introducida directamente en un sistema de desarrollo basado en computadora, ofrece enormes beneficios potenciales.

Mantenimiento:

Es esta la fase mas costosa del ciclo de vida del software. El mantenimiento de la configuración de software casi siempre significa mantenimiento de la representación del diseño procedimental.

Exigencia de estructura:

La notación de diseño que refuerce el uso de únicamente construcciones estructuradas, promueve la práctica de un buen diseño.

Representación de datos:

La habilidad para representar los datos locales y globales es un elemento esencial en el diseño detallado. Una notación puede idealmente, representar directamente los datos.

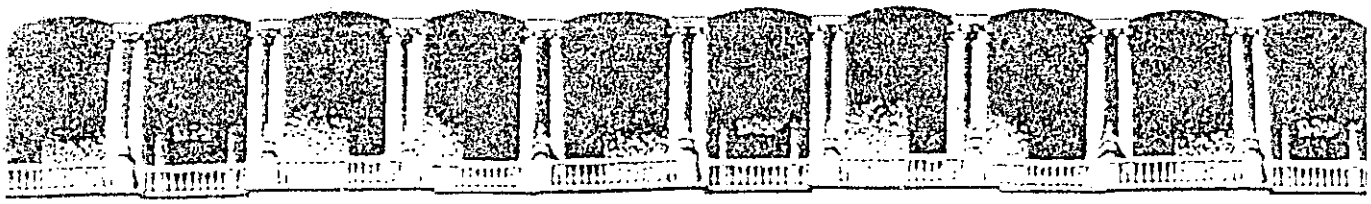
Verificación lógica:

La verificación automática de la lógica del diseño es un objetivo supremo durante la prueba de software. Una notación que la refuerce mejora grandemente la suficiencia de la prueba.

Disposición de la codificación:

Una notación que se convierta fácilmente a código fuente reduce el trabajo y los errores.

Parece que un lenguaje de diseño de programas ofrece la mejor combinación de características. Se puede insertar directamente en los listados fuente, mejorando de esta forma la documentación y haciendo menos difícil el mantenimiento del diseño.



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

ANALISIS Y DISEÑO DE SISTEMAS DE PROCESAMIENTO DE DATOS

T E M A 5

DISEÑO ORIENTADO AL FLUJO DE DATOS

ABRIL-MAYO, 1992.

Tema 5 DISEÑO ORIENTADO AL FLUJO DE DATOS.

El diseño se ha descrito como un proceso multipaso en el que las representaciones de la estructura de datos, estructura de programa y procedimiento, se sintetizan a partir de los requerimientos de la información. Es además, una actividad referida a la toma de decisiones importantes, frecuentemente de una naturaleza estructural. Comparte con la programación los aspectos referentes a la abstracción en la representación de la información y en las secuencias de procesamiento, pero el nivel de detalles es muy diferente en ambos casos. El diseño construye representaciones coherentes y bien planificadas de programas, concentrándose en las interrelaciones de las partes a un nivel más alto y en las operaciones lógicas implicadas en los niveles inferiores.

En este tema trataremos acerca del diseño orientado al flujo de datos (y en general el diseño de software), el cual tiene sus orígenes en los primeros conceptos de diseño que aventuraron la modularidad, diseño descendente y programación estructurada. Sin embargo, el enfoque de diseño orientado al flujo de datos amplió estas técnicas procedimentales, integrando explícitamente el flujo de la información en el proceso de diseño.

Subtema 5.1 Areas de aplicación.

El diseño orientado al flujo de datos puede aplicarse a un amplio rango de áreas de aplicación. Por lo cual es un factor importante de selección de un método de diseño, la amplitud de las áreas en la que se aplica.

Un método de diseño orientado al flujo de datos es particularmente útil cuando la información se procesa secuencialmente y no existe una estructura de datos jerárquica.

Las técnicas de diseño orientadas al flujo de datos se aplican también a aquellas que se relacionen con el procesamiento de datos y pueden aplicarse con efectividad, incluso cuando existen estructuras de datos jerárquicas.

Subtema 5.2 Consideraciones sobre el proceso de diseño.

El diseño orientado al flujo de datos permite una cómoda transición de las representaciones de la información a una descripción de diseño de la estructura del programa.

Esta transición se realiza en cinco pasos:

1) se establece el tipo del flujo de información;

2) se indican los límites del flujo;

3) el diseño de flujo de datos se convierte en la estructura del programa;

4) se define la jerarquía de control mediante factorización, y

5) se refina la estructura resultante usando medidas y heurísticas de diseño.

Subtema 5.3 Flujo de transformación y Flujo de Transacción.

Analizaremos ahora dos tipos de flujo de información. Primero veremos el flujo de transformación:

Recordando el modelo fundamental del sistema (diagrama de flujo de datos), la información debe entrar y salir en una forma del "mundo externo". Por ejemplo, los datos escritos sobre un teclado de una terminal, hablados sobre una línea de teléfono y dibujados sobre una presentación gráfica por computadora, son todos formas de información del mundo externo. Tales datos externos deben ser convertidos en una forma interna para el procesamiento.

La historia del tiempo del flujo de datos se muestra en la siguiente figura, en la que se aprecia, la información entra al sistema mediante caminos que transforman los datos externos en una forma interna y se identifica como flujo de llegada. En el núcleo del software ocurre una transición. Los datos de llegada se pasan a través del centro de transformación y comienzan a moverse a lo largo de caminos que conducen ahora hacia la "salida" del software, se llaman flujo de salida.

En lo que se refiere al flujo de transacción tenemos que :

El modelo fundamental del sistema implica un flujo de transformación; el flujo de información se caracteriza frecuentemente por un elemento de datos sencillo, llamado una transacción, que desencadena otro flujo de datos a lo largo de uno de los muchos caminos. Cuando un DFD toma la forma mostrada a continuación, se presenta un flujo de transacción.

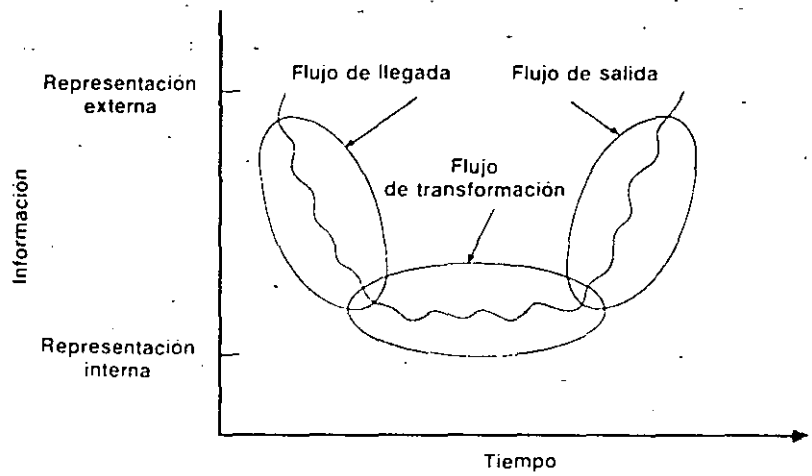
Este flujo de transacción se caracteriza por datos que se mueven a lo largo de un camino de llegada que convierte información del mundo externo en una transacción, esta transacción es evaluada, y basándose en este valor el flujo se inicia por uno de los muchos caminos de acción. El centro de flujo de información desde que salen muchos caminos de acción se llama un centro de transición. En un DFD para un gran sistema, pueden presentarse los dos tipos de flujos de transformación y de transacción.

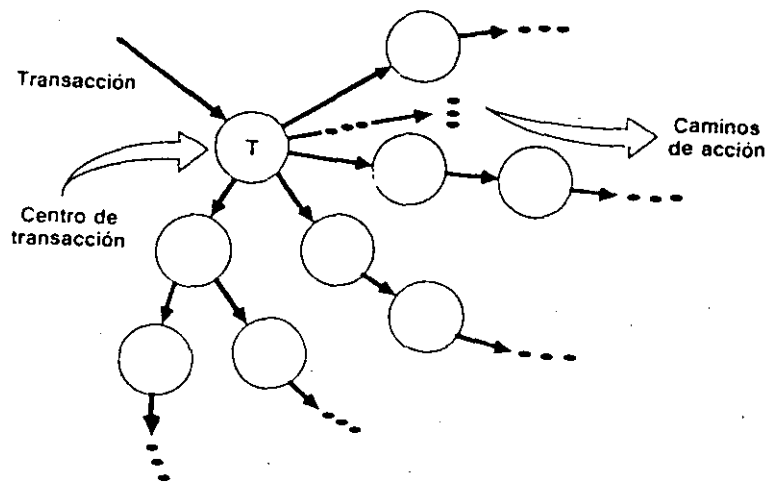
Para completar este punto vamos a definir que es un Análisis de Transformación.

El análisis de transformación es un conjunto de pasos de diseño que permiten a un DFD, con características de flujo de transformación, convertirse en una templeta predefinida para la estructura del programa.

Subtema 5.4 Pasos del diseño.

Los pasos de diseño comienzan con una revaluación del trabajo hecho durante el análisis de requerimientos y luego se va hacia el desarrollo de la estructura del programa. Analicemos paso a paso como se lleva a cabo este diseño:


Representación externa
Información
Representación interna
Flujo de llegada
Flujo de salida
Flujo de transformación
Tiempo


Transacción
T
Centro de transacción
Caminos de acción

Paso 1. Revisión del modelo fundamental del sistema. El modo fundamental del sistema abarca el nivel 01 DFD y el soporte de la información. En realidad, el paso de diseño comienza con una evaluación de la Especificación del Sistema y de la Especificación de los Requerimientos del Software. Ambos documentos describen el flujo y estructura de la información de la interfase del software.

Paso 2. Revisión y refinamiento de los diagramas de flujo de datos para el Software. Usando la información obtenida de una Especificación de Requerimientos del software, se deriva un diagrama de flujos de datos a nivel 03 para el implicado en una transformación ejecuta una función sencilla y diferenciada, el DFD contiene los suficientes detalles para un "primer corte" en el diseño de la estructura de programa, y procedemos sin más refinamientos.

Paso 3. Determinar si el DFD tiene características de transformación de transacción, el flujo de información de un sistema puede presentarse siempre como transformación. Sin embargo, cuando se encuentra una transacción, se recomienda una organización de diseño diferente.
En este paso, el diseñador selecciona una característica global del flujo basándose en la naturaleza prevalente del DFD. Además, se aíslan las regiones locales del flujo de transformación o transacción. Estos subflujos pueden usarse para refinar la estructura del programa derivada de la característica global descrita anteriormente.

Paso 4. Aislar el centro de transformación especificando los límites del flujo de llegada y salida.
Los límites del flujo de llegada y salida están abiertos a diferentes diseñadores pueden seleccionar puntos algo diferentes en el flujo como posiciones límites, pueden derivarse soluciones de diseño alternativas variando la colocación de los límites del flujo.

Paso 5. Realización del "primer nivel de factorización". La estructura del programa representa una distribución descendente del control. La factorización da como resultado una estructura de programa en la que los módulos de nivel superior toman las decisiones de ejecución y los módulos de nivel inferior ejecutan la mayoría del trabajo de entrada, computacional y de salida. Los módulos de nivel intermedio ejecutan algún control y realizan moderadas cantidades de trabajo.

Primer nivel de factorización.
El número de módulos del primer nivel debe limitarse al mínimo necesario, para que puedan realizarse las funciones de control y mantener al mismo tiempo unas buenas características de acoplamiento y cohesión.

Paso 6. Ejecución del "segundo nivel de factorización". El segundo nivel de factorización se realiza mediante la conversión de las transformaciones individuales de un DFD, en los módulos correspondientes de la estructura del programa. Comenzando en el límite del centro de transformación y yendo hacia fuera a lo largo de los

caminos de llegada y luego de salida, las transformaciones se convierten en niveles subordinados de la estructura del software.

Una estructura de programa de primer corte puede siempre refinarse aplicando los conceptos de independencia de módulos. Se puede aumentar o reducir el número de módulos para producir una estructura que pueda implementarse sin dificultad, probarse sin confusión y mantenerse sin problemas.

El objetivo de los siete pasos procedentes es desarrollar una representación global del software. Esto es, una vez que se define la estructura, podemos evaluar y refinar la arquitectura del software viéndola como un todo.

Subtema 5.5 ANALISIS DE TRANSACCION

En muchas aplicaciones software, un elemento de dato determina uno o más flujos de información que afectan a la función implicada por el elemento de dato determinante. El elemento de dato, llamado transacción.

Pasos del diseño

Los pasos del diseño para el análisis de transacciones son similares y en algunos casos idénticos a los pasos para el análisis de transformaciones. La principal diferencia se refiere a la conversión del DFD en estructura de programa.

Paso 1. Revisar el modelo fundamental del sistema.

Paso 2.Revisar y refinar los diagramas de flujo de datos par software.

Paso 3.Determinar si el DFD tiene características de transformación o de transacción. Deben establecerse límites de flujo para ambos tipos de flujo.

Paso 4.Identificar el centro de transacción y las características del flujo de cada camino de acción. La posición del centro de transacción puede discernirse inmediatamente a partir del DFD. El centro de transacción está ligado al origen de varios caminos de información que fluyen radialmente de el.

Paso 5.Transformar el DFD en una estructura software adecuada al procesamiento de transacciones. El flujo de transacción se convierte en una estructura de programa que contiene una bifurcación de entrada y una bifurcación de salida o expedición. La estructura para la bifurcación de entrada se desarrolla de la misma forma que el análisis de transformaciones.

Paso 6.Factorizar y refinar la estructura de transacciones y la estructura de cada camino de acción. Cada camino de acción del diagrama de flujo de datos tiene sus propias características de flujo de información, se puede encontrar un flujo de transformación de transacción.

La transformación crear y visualizar mensaje del DFD se convierte en un módulo de utilidad (es decir, un módulo llamado por dos o más módulos) que se utiliza por dos estructuras de flujo de acción.

Paso 7.Refinar la estructura del software de "primer corte" usando medidas y heurísticas de diseño. Criterios tales como independencia de módulos, práctica y mantenimiento, deben ser considerados cuidadosamente cuando se propagan modificaciones a la estructura.

Una vez que se ha desarrollado la estructura del programa usando el diseño orientado al flujo de datos, puede conseguirse una modularidad efectiva aplicando los conceptos introducidos en el tema anterior y manipulando la estructura resultante de acuerdo al siguiente conjunto de criterios.

I.Evaluar la estructura de programa preliminar para reducir el acoplamiento y mejorar la cohesión. Los módulos pueden expandirse o reducirse con la mirada puesta en la mejora de la independencia de los módulos.

II.Intentar minimizar las estructuras con un abanico de salida ancho; fomentar los abanicos de entrada conforme se incrementa la profundidad.

III.Mantener el efecto de un módulo dentro del ámbito de control de ese módulo.

IV.Evaluar las interfases de los módulos para reducir la complejidad y redundancia y mejorar la consistencia.

V. Definir módulos cuyas funciones sean predecibles, pero evitar módulos que sean demasiado restrictivos.

VI.Buscar los módulos de una única entrada y una única salida, evitando las conexiones patológicas.

VII.Empaquetar el software basándose en las restricciones del diseño y requerimientos de transportabilidad.

En forma global, podemos decir que las técnicas expuestas en este tema conducen a una descripción del diseño preliminar del software. Se definen módulos, se establecen las interfases y se desarrollan las estructuras de datos. Estas representaciones del diseño forman la base de todo el trabajo posterior de desarrollo.

TEMA 7. Diseño Orientado al Objeto

En este tema trataremos el diseño orientado al objeto, año con año se logran avances que conducen a la construcción de computadoras mas veloces, mas compactas y baratas; sin embargo, el aspecto de software no se ha desarrollado en forma similar, mientras los costos de hardware han disminuido continuamente, los de software han hecho lo contrario. No es una tarea fácil la construcción de software y en varias ocasiones los proyectos de programación sobregiran los presupuestos de tiempo y dinero.

El problema del mantenimiento de software

Como ya hemos mencionado en temas anteriores la construcción de software ha sido la principal preocupación de los expertos por lo que se han conseguido avances significativos para la construcción de programas en forma sistemática y a bajo costo. Algunos de los resultados de estos esfuerzos son las técnicas que con el diseño estructurado y el desarrollo descendente (top-down), han sido utilizadas durante mucho tiempo por los programadores. Pero hay aun tres grandes deficiencias:

- los productos que resultan de emplear estas técnicas son poco flexibles

- los programadores que las usan tienden a concentrarse en el diseño e implementación inicial del sistema, sin tomar en cuenta su vida posterior.

- no alientan al programador a aprovechar el trabajo de proyectos anteriores.

Una desventaja de utilizar una metodología que se concentra en el diseño inicial del sistema se hace evidente si se considera que la vida útil de un producto de software puede ser cinco o seis veces mas grande que el lapso en que se desarrolla. Los gastos que se hacen durante el último período (gastos de mantenimiento) representan el 70% del costo total del sistema. Alrededor del 42% del costo total se tienen que hacer por cambios en las especificaciones del usuario o en los formatos de los datos. Es por ello que la extensibilidad (la facilidad con que se modifica un sistema para que realice nuevas funciones) debe ser uno de los objetivos primarios de la etapa de diseño.

En la fase de mantenimiento es muy importante que cualquier método de diseño tenga como objetivo principal producir sistemas que faciliten su propio mantenimiento. Para ello se recomienda una serie de actividades y heurísticas que pueden servir como guía durante el desarrollo del sistema.

Reutilización de código

Otro de los factores que no se toma en cuenta en los métodos de diseño tradicionales es la reutilización de código. Cualquier programador que desarrolla un sistema nuevo debe escribir una buena cantidad de código que ha escrito anteriormente una y otra vez. Las rutinas de búsqueda, ordenamiento, manejo de menús y despliegue de ventanas, se repiten continua-

mente en proyectos diferentes. Los métodos de desarrollo de software deben alentar al programador a utilizar el código escrito previamente por el mismo o por otros programadores.

Se han empleado tres tipos de reutilización:

- de personal: un proyecto nuevo se le asignan a programadores que tienen experiencia en el desarrollo de proyectos semejantes.

- de diseño: consiste en emplear el diseño de un sistema para desarrollar otro sistema similar

- de código fuente: se da cuando un programador utiliza parte de un programa escrito con anterioridad para crear un nuevo programa.

Sin embargo, estas tres formas de reutilización se han empleado en forma muy limitada, por lo que es necesario buscar técnicas mas generales.

Modularidad

En temas anteriores mencionamos una de las herramientas de diseño más poderosa para facilitar el desarrollo y mantenimiento de sistemas de software, la modularidad. En este tema trataremos algunas de las características de esta herramienta.

Como recordaran, la modularidad es dividir un sistema complejo en unidades mas pequeñas y manejables, cada una de esas unidades se encarga de manejar un aspecto local de todo el sistema, interactuando con otros módulos para cumplir con el objetivo global.

El concepto de módulo es aún mas sofisticado. Entre las propiedades que tiene están: la decomponibilidad, componibilidad, comprensibilidad, continuidad y protección.

a) Decomponibilidad: Un método modular debe permitir descomponer un problema de diseño en problemas más pequeños que pueden resolverse en forma independiente.

b) Componibilidad: una vez que se cuenta con un conjunto de módulos que realizan una función especifica, se debe alentar al programador a usar esos módulos para construir nuevos programas.

c) Comprensibilidad: El lector de un programa o librería debe ser capaz de entender el funcionamiento de cada módulo sin necesidad de consultar el texto de otros módulos.

d) Continuidad: un cambio pequeño en las especificaciones de un programa debe causar cambios en un sólo módulo ó en un conjunto pequeño de ellos.

e) Protección: un error de ejecución en el funcionamiento de un modulo no debe expandirse hacia los demás módulos.

Para alentar estas propiedades se utilizan las siguientes estrategias:

Primero: Un modulo debe corresponder con una unidad sintáctica del lenguaje.

Segundo: Los módulos deber tener pocas interfaces (medio de comunicación entre módulos) y estas debe ser pequeñas. Un número pequeño de interfaces aumenta la independencia de los módulos, haciendo mas fácil el proceso de componer nuevos sistemas a partir de módulos prefabricados, ayuda además a evitar que los errores de un módulo se propaguen por todo el sistema.

y **Tercero:** Cada módulo debe ocultar su implementación y algoritmos internos al resto del sistema. No se debe permitir que un módulo modifique los elementos internos de otros módulos.

Diseño orientado a objetos

El diseño orientado a objetos (DOO), es un método que tiene como objetivo establecer técnicas de desarrollo de software para producir sistemas modulares. Un sistema orientado a objetos se puede entender fácilmente, facilitando así su desarrollo, depuración y mantenimiento.

El DOO, se basa en un medida sencillas; un sistema de computadora es un modelo que representa un subconjunto del mundo real; la estructura de ese sistema se simplifica en gran medida si cada una de las entidades u objetos, del problema que se esta modelando corresponde directamente con un objeto que se pueda manipular internamente en el sistema.

El proceso de representar entidades reales con elementos internos en un sistema recibe el nombre de abstracción de datos.

La abstracción de datos no se concentra en la representación interna de un objeto (una cadena, un arreglo, un entero, etc.)

sino en sus atributos y en las operaciones que se pueden realizar sobre ese objeto. De esta forma, un tipo de dato abstracto se puede describir concentrándose en las operaciones que manipulan a los objetos de ese tipo, sin caer en los detalles de representación y manipulación de datos.

La creación de tipos abstractos permite al diseñador adaptar el diseño a sus necesidades especificas, se debe ser capaz de definir tipos de datos que se ajusten al problema que se esta resolviendo y que puedan ser manipulados después por el lenguaje. Concentrándose así en los objetos que manipula su sistema y las relaciones entre estos objetos, haciendo a un lado la representación y manipulación de datos, para comprender mejor el problema.

Conceptos

Bien, ahora para entender un poco mas acerca del diseño orientado a objetos (DOO), trataremos algunos conceptos:

CLASE:

Es un tipo abstracto del diseño orientado a objetos, una clase describe un conjunto de objetos semejantes. Dicha descripción se hace en dos partes:

- los datos que especifican las propiedades de los objetos de esa clase, llamados atributos.

- y las funciones que manipulan esos datos, llamados métodos.

Un objeto puede recibir mensajes de otros objetos; entonces, debe escoger uno de sus métodos y dar una respuesta basándose en los datos que representan su estado.

ELEMENTOS PRIVADOS Y PUBLICOS

Cada objeto cuenta con elementos privados, que solo pueden ser usados por objetos de su misma clase. Los elementos públicos representan la interface de un objeto con su medio ambiente, únicamente esos elementos pueden modificar a los datos privados, los cuales representan el estado del objeto.

Se proponen dos estrategias para la reutilización de código: la composición y la herencia.

La COMPOSICION permite definir una nueva clase de objetos mediante la unión de un conjunto de clases ya existentes.

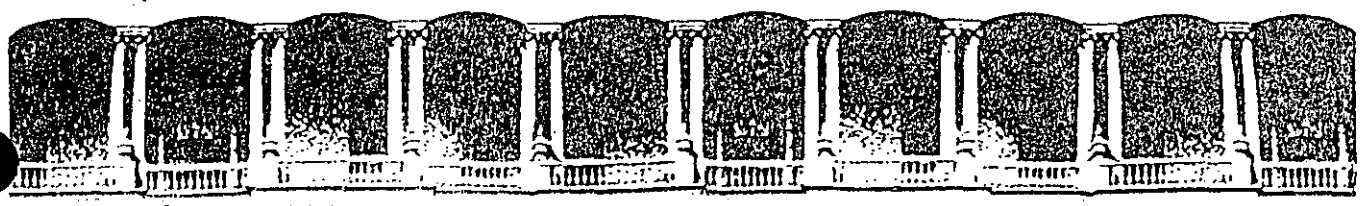
La HERENCIA permite derivar una nueva clase basándose en otra clase mas general. Una clase derivada adquiere todas las propiedades y métodos de la clase de la que se derivo, llamada clase base.

Además de facilitar la reutilización de código, la herencia es un medio ideal para crear sistemas con alta extensibilidad. Otra ventaja de esta técnica es que permite manipular objetos de clases diferentes como si fueran de la misma clase, con lo cual es posible definir interfaces uniformes para diferentes tipos de objetos.

Lenguajes orientados a objetos

Las técnicas en las que se basa el diseño orientado a objetos como son ocultamiento de información, abstracción de datos, manejo de clase, eran conocidas y utilizadas por los ingenieros de software desde hace muchos años, sin embargo hay pocos lenguajes que brindan todas las facilidades para escribir programas orientados a objetos, algunos de los mas populares

son Simula67, Smalltalk, Eiffel, Ada o Modula. Estos lenguajes orientados a objetos no habían tenido hasta hace poco una aceptación amplia entre la comunidad de programadores.



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

ANALISIS Y DISEÑO DE SISTEMAS DE PROCESAMIENTO DE DATOS

TEMA. 8

IMPLEMENTACION DEL SISTEMA

ABRIL-MAYO, 1992

TEMA 8. IMPLEMENTACION DEL SISTEMA

Introducción

Hasta el momento hemos aprendimos cómo se efectúa el análisis del sistema; es decir cómo elaborar un estudio de necesidades y requerimientos de nuestro sistema de información antes de desarrollarlo.

Posteriormente, conocimos las técnicas para diseñar, con base al análisis previo, nuestro sistema, orientándolo al medio en el que se desarrollará y aplicará.

Ahora sabremos cómo desarrollar nuestro sistema tomando en cuenta las especificaciones del diseño y el equipo de cómputo seleccionado.

En este tema conoceremos las formas de plasmar en un equipo de cómputo el diseño de un sistema. Para ello contamos con cuatro subtemas:

Introducción

Técnicas de codificación estructurada

Estándares de Implementación y

Documentación

Subtema 8.1 Introducción

Hasta ahora hemos hablado de cómo efectuar el análisis y el diseño de un sistema que van dirigidos hacia un objetivo final: la traducción de las especificaciones del diseño a una forma que puede ser "comprendida" por una computadora. Hemos llegado finalmente al paso de la codificación. El objetivo principal de la implementación del sistema es el escribir código fuente así como la documentación interna de modo que la concordancia del código con sus especificaciones sea fácil de verificar, así como facilitar la depuración, las pruebas y modificaciones al sistema. Para ello se debe elaborar un código claro utilizando 3 parámetros:

Técnicas de codificación estructurada

Estándares de implementación y

Documentación.

Cada uno de estos puntos lo tocaremos específicamente en los siguientes subtemas.

Subtema 8.2 Técnicas de codificación estructurada.

Las técnicas de codificación tienen como fin 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 programación estructurada es la aplicación de métodos básicos de descomposición de problemas para establecer una estructura jerárquica fácilmente manejable a través del proceso de refinamiento progresivo. Es decir, cualquier módulo, aún el más complicado, se detalla utilizando las reglas básicas de la programación estructurada que son:

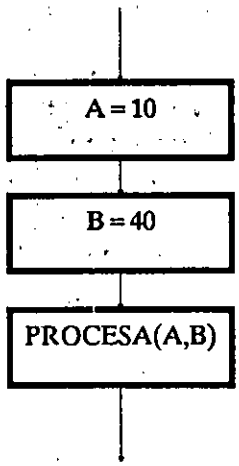
Secuencia

Decisión

Repetición

A continuación explicaré cada una de estas reglas.

La secuencia se define como una estructura de proceso que consiste en enunciar, una después de otra, una serie de instrucciones.



Gráficamente se representa como se muestra en la figura, en donde cada instrucción está encerrada en un cuadro y mantiene una jerarquía. Esto es, se van ejecutando las instrucciones una después de la otra.

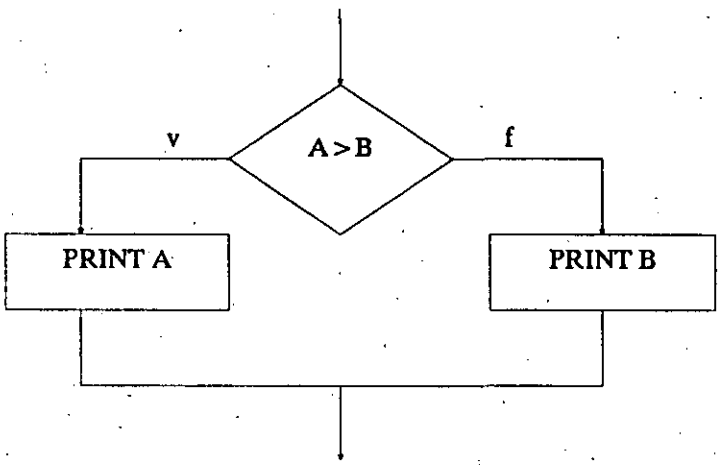
A = 10

B = 40

PROCESA(A,B)

Esta figura muestra un ejemplo en donde está parte de un programa que contiene instrucciones simples exclusivamente. Una proposición simple puede ser una asignación o un llamado a un procedimiento; una proposición compuesta es un if-then-else o un while-do. Todas estas proposiciones se pueden presentar como una secuencia.

La segunda regla básica, la decisión, es una estructura de proceso que permite especificar alternativas en la ejecución de instrucciones, dependiendo de una condición.



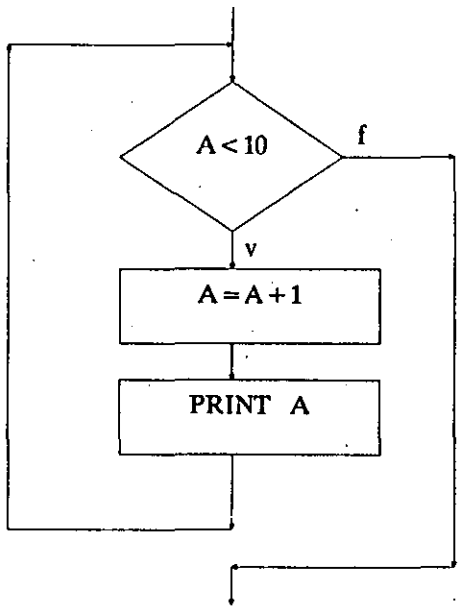
Gráficamente se representa como se ve en la figura, en donde la condición que determinará la alternativa a seleccionar está en un rombo y de éste se desprenden las alternativas a seguir.

Un ejemplo de codificación de una decisión se muestra a continuación. Las alternativas a realizar se presentan con cierta sangría para detectar con facilidad la dependencia de su ejecución con respecto a la condición. Comunmente se utiliza la proposición if-then-else para su construcción.

```
IF  A > B THEN
      PRINT A
ELSE
      PRINT B
ENDIF
```

Y la última regla básica, la repetición, es una estructura de proceso que permite la especificación de la ejecución iterativa de una serie de instrucciones.

En la figura se puede observar la representación gráfica de la repetición. La condición que determinará el número de iteraciones está encerrada en un rombo.



Al codificar una repetición, como se muestra en la figura, se deberán sangrar las instrucciones que se estarán iterando. Es común utilizar las proposiciones while-do o repeat-until para codificar la estructura de repetición.

```
WHILE  A < 10
      A = A + 1
      PRINT  A
END
```

Una vez que han quedado claras las tres reglas básicas de la codificación estructurada, podemos decir que la programación estructurada se basa en un teorema el cual establece lo siguiente:

Cualquier módulo de un programa con una entrada y una salida que tenga todas las proposiciones en algún camino de la entrada a la salida puede especificarse usando solamente secuenciación, decisión y repetición.

La propiedad de una entrada y una salida permite el anidamiento de construcciones dentro de otra en cualquier forma que se desee. Esto es, por ejemplo, si tenemos una proposición de decisión, una alternativa puede ser otra proposición de decisión que a su vez una de sus alternativas sea otra proposición de decisión. La codificación correspondiente a este ejemplo se muestra en la figura.

```
WHILE  A>B
       A=A-1
       WHILE  B>0
              B=A/B
              WHILE B>1
                     PRINT B
              END
       END
END
```

El proceso de la codificación, que es la comunicación entre los humanos y las computadoras mediante un lenguaje de programación, es una actividad humana y es por ello que las características del lenguaje afectan directamente a la calidad de la comunicación, tienen un impacto importante sobre el éxito de un proyecto de desarrollo de software y pueden influenciar la calidad del diseño (ya que prácticamente el diseño detallado se dirige hacia un lenguaje de programación específico).

La elección de un lenguaje de programación para un proyecto específico debe tener en cuenta sus características y ciertos criterios.

El problema asociado con la elección del lenguaje puede desaparecer si solo se dispone de un lenguaje o si el cliente demanda uno en particular.

Los criterios que se aplican durante la evaluación de los lenguajes disponibles están:

1) area de aplicación general

2) complejidad algorítmica y computacional

3) entorno en el que se ejecutará el software

4) consideraciones de rendimiento

5) complejidad de las estructuras de datos

6) conocimiento de la plantilla de desarrollo de software y

7) disponibilidad de un buen compilador o compilador cruzado.

Actualmente C es el lenguaje elegido comunmente en el desarrollo de software de sistemas.

Y volviendo con las técnicas de codificación estructurada podemos decir que en un nivel fundamental, todos los lenguajes modernos permiten al programador representar secuencias, decisiones y repeticiones -que son las construcciones lógicas de la programación estructurada. La mayoría de los lenguajes de programación modernos proporcionan un sintáxis para la especificación directa del if then else, del do while y del repeat until. Otros lenguajes, como Lisp y APL, requieren que el programador emule esas construcciones dentro de los límites de sintaxis del lenguaje.

Con frecuencia se menciona que el apegarse estríctamente a las técnicas de codificación estructurada existe la necesidad de emplear variables auxiliares, segmentos de código repetido y un excesivo llamado a subprogramas, dando como resultado una utilización ineficiente del espacio de memoria y del tiempo de ejecución. Es por ello que hablaré de la eficiencia de código.

La eficiencia del código fuente está directamente unida a la eficiencia de los algoritmos definidos durante el diseño detallado. Sin embargo, el estilo de codificación puede afectar a la velocidad de ejecución y a los requerimientos de memoria. Las siguientes directrices se deben seguir siempre en el proceso de traducción del diseño detallado al código:

- Simplificar las expresiones aritméticas y lógicas antes de convertirlas en código.

- Evaluar cuidadosamente las iteraciones anidadas para determinar si se pueden sacar fuera de ellos algunas sentencias o expresiones.

- Cuando sea posible, evitar el uso de arreglos multidimensionales.

- Cuando sea posible, evitar el uso de apuntadores y listas complejas.

- Usar rápidas operaciones aritméticas.

- Usar cuando sea posible aritmética entera y expresiones Booleanas o lógicas.

Por lo anterior, podemos decir que las técnicas de codificación estructurada no son en sí las que propician la ineficiencia sino los algoritmos y las estructuras de datos los que afectan la velocidad de ejecución y el espacio en memoria respectivamente.

Subtema 8.3 Estándares de Implementación.

A continuación veremos qué son y para qué nos sirven los estándares de implementación, dentro de la implementación de un sistema.

Los estándares de codificación son especificaciones para un estilo de codificación preferido. Dada una situación de elegir los caminos para lograr un efecto, se especifica un camino preferido. Los estándares de codificación a menudo son vistos por los programadores como mecanismos para restringir y devaluar las habilidades para resolver problemas creativos de los programadores. La creatividad siempre ocurre dentro de un marco de trabajo básico de estándares. Los artistas siguen los principios básicos de estructura y composición, los poetas se apegan al ritmo y a la métrica del lenguaje. Sin un marco de trabajo de estándares que guíen y canalicen una actividad, la creatividad se vuelve un caos sin sentido.

Así, es **deseable** que todos los programadores de un proyecto adopten un estilo de codificación similar, de modo que se produzca un código de calidad uniforme.

Algunos de los estándares que se pueden aplicar en todas las situaciones son:

- El uso de la instrucción GOTO debe evitarse en circunstancias normales. Esto es, utilizarse únicamente para lograr el formato de las construcciones de la codificación estructurada en los lenguajes primitivos. La proposición GOTO es valiosa para las salidas múltiples y prematuras de los ciclos, y para transferir el control al código manipulador de errores. Como pueden observar en su pantalla tenemos primeramente un código en el que se emplea la instrucción GOTO sin necesidad y posteriormente cómo debe ser.

- Introdúzcanse tipos de datos definidos por el usuario. Muchos lenguajes de programación proporcionan tipos de datos definidos por el usuario. El uso de distintos tipos de datos hace posible distinguir las entidades propias del problema. Por ejemplo si definimos un tipo de datos llamado color y declaramos los colores: amarillo, rojo, azul, verde y anaranjado; podemos saber que únicamente, de toda la gama de colores que existen, se van a utilizar esos.

- Examínense cuidadosamente las rutinas que tengan menos de 5 o más de 25 proposiciones. Un subprograma que tiene más de 25 proposiciones ejecutables probablemente contiene una o más subfunciones bien definidas que pueden agruparse como subprogramas distintos e invocarse cuando se necesiten. Se dice también que 30 proposiciones ejecutables es el límite superior de compresión para una sola lectura de un subprograma escrito en un lenguaje de programación. Un subprograma que tiene menos de cinco proposiciones normalmente es muy pequeño para proporcionar una función bien definida. Existen excepciones obvias a estos principios generales: por ejemplo rutinas para obtener el valor absoluto o el módulo de un número y que tiene menos de 5 líneas pueden ser muy útiles.

- Utilíncese sangrías, paréntesis, espacios y líneas en blanco, así como márgenes alrededor de los bloques de comentarios para aumentar la legibilidad. Si el formato del texto fuente lo hacen varios programadores, todos ellos deben acordar y apegarse a condiciones estándar.

- Proporciónense prólogos estándares de documentación para cada subprograma. Un prólogo de documentación contiene la información acerca de un subprograma o una unidad de compilación que no resulta obvia al leer el texto fuente del subprograma o unidad de compilación.

- Aíslense las dependencias de la máquina en unas cuantas rutinas. Si se necesita cambiar la naturaleza de la representación de datos, o si el programa se cambia a una máquina diferente, se localizan los detalles de la dependencia de la máquina y pueden modificarse en consecuencia.

- La profundidad de anidamiento de la construcciones de un programa debe ser de cinco o menos. Si el anidamiento se vuelve demasiado profundo, se vuelve difícil determinar las condiciones bajo las cuales se va a ejecutar la proposición más anidada.

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 de la buena programación, la experiencia nos indica que los estándares pueden servir como "guías de estilo".

Subtema 8.4 Documentación.

La programación por computadora incluye el código fuente de un sistema y todos los documentos de apoyo generados durante el análisis, diseño, implementación, pruebas y mantenimientodel sistema.

Las especificaciones de requisitos, documentos de diseño, planes de prueba, manuales de usuario y los reportes de mantenimiento son ejemplos de documentos de apoyo. Estos documentos son los productos que resultan del desarrollo y mantenimiento sistemático de la programación.

Un enfoque sistemático al desarrollo de la programación garantiza que los documentos de apoyo se desarrollen de una manera ordenada y que esos documentos se encuentren disponibles cuando se necesiten. Estos documentos deben desarrollarse como un producto natural paralelo al proceso de desarrollo. La calidad, cantidad, duración y utilidad de los documentos de apoyo son las principales medidas de la salud y la bondad de un proyecto de programación.

Los documentos sirven para explicar y/o aclarar lo que sucede en el flujo de control de un programa. Existen dos tipos de Documentación:

- Documentación interna

- Comentarios explicativos del cuerpo del módulo.

La documentación interna consiste en un prólogo estándar para cada unidad de programa, los aspectos autodocumentados del código fuente y los comentarios internos intercalados en la porción ejecutable del código.

La mejor documentación de un programa de computadora la constituye la claridad de su estructura; además la documentación más confiable de un programa es el código mismo. 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.

El formato característico de los prólogos de subprogramas es:

Nombre del autor:

Fecha de terminación:

Función realizada:

Rutinas que lo invocan:

Rutinas que invoca:

Autor/fecha/modificación:

Resumen

La implementación de un sistema tiene por objetivo obtener un código fuente claro, que facilite la depuración, prueba y modificación y que estas actividades consuman una gran porción de la mayor parte de los presupuestos en la programación.

En este tema se analizaron las técnicas de la codificación estructurada, el estilo de codificación, estándares y principios generales y la documentación interna y de apoyo del código fuente.

TEMA 9. Verificación y validación del Sistema

En el tema anterior conocimos las técnicas y consideraciones para implantar nuestro sistema de información en un equipo de cómputo elegido; sin embargo es muy importante efectuar verificaciones o pruebas en su implementación para obtener un sistema confiable y eficiente. El desarrollo de sistemas de software envuelve una serie de actividades de producción en las que las posibilidades de que aparezca la falla humana son enormes. Los errores pueden darse desde el primer momento del proceso en el que los objetivos se pueden especificar de forma errónea o imperfecta, así como los errores que aparecen en los posteriores pasos de diseño y desarrollo. Debido a la imposibilidad humana de trabajar y comunicarse de forma perfecta, el desarrollo de software ha de ir acompañado de una actividad que garantice la calidad.

Los objetivos de las actividades de verificación y validación son el valorar y mejorar la calidad de los productos de trabajo generado durante el desarrollo del diseño y la implementación.

La verificación se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica. La validación se refiere a un conjunto diferente de actividades que aseguran que el software construído se ajusta a los requerimientos del cliente. Podemos establecer como

Verificación al hacer la pregunta: ¿Estamos construyendo correctamente el producto? es decir, cómo estamos construyendo el producto, ¿de manera adecuada?. Mientras que para establecer a la validación podemos hacer la pregunta: ¿Estamos construyendo el producto correcto? es decir, el producto que se está elaborando es el que cubre todos los requerimientos.

La calidad de los productos de trabajo generados durante el análisis y el diseño se pueden estimar y mejorar utilizando procedimientos sistemáticos de control de calidad, mediante recorridos e inspecciones y por medio de verificaciones automatizadas para supervisar que sea consistente y que esté completo. La técnica para estimar y mejorar la calidad del código fuente incluye los procedimientos sistemáticos de la prueba de unidad, la prueba de integración, la prueba de validación y la prueba del sistema.

Subtema 9.1 Pruebas del Sistema.

Ahora detallaré cada una de las pruebas que se mencionaron anteriormente.

Iniciaremos con la Prueba de Unidad. La prueba de unidad centra el proceso de verificación en la menor unidad de diseño de software. Las pruebas de unidad comprenden el conjunto de pruebas efectuadas por un programador individual, antes de la integración de la unidad en un sistema más grande. Una unidad de programa suele ser lo suficientemente pequeña como para que el programador que la desarrolló pueda probarla con minuciosidad. Las pruebas que se dán como parte de la prueba de unidad son:

Prueba de interfaz

Prueba de las estructuras de datos

Pruebas de condiciones de límite

Pruebas de caminos independientes

Manejos de errores

Veamos ahora **Pruebas de integración.** Las pruebas de integración es una técnica sistemática para construir la estructura del sistema mientras que al mismo tiempo se llevan acabo pruebas para detectar errores asociados con la interacción entre sus módulos. El objetivo es tomar los módulos probados en unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.

Existen dos estrategias de integración:

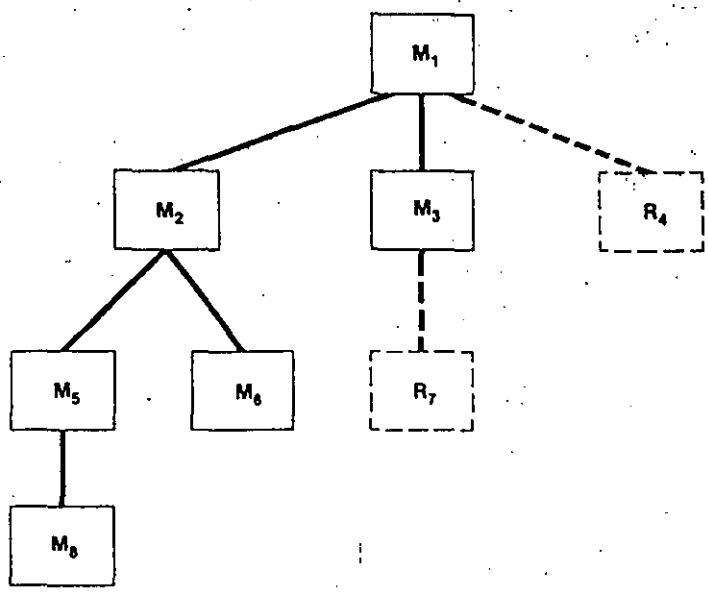
-Integración descendente

-Integración ascendente

Veremos cada una de ellas a continuación; comenzaremos primeramente con la Integración descendente.

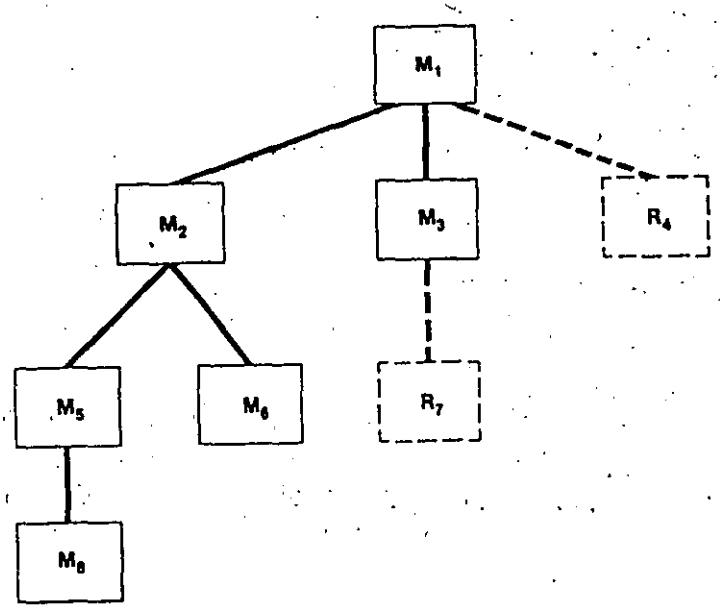
En la integración descendente se va haciendo una aproximación incremental a la construcción de la estructura de programas. Es decir, se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando con el módulo de control principal (llamado comúnmente programa principal). Los módulos subordinados al módulo de control principal se van incorporando en la estructura, ya sea de la forma primero-en-profundidad o de la forma primero-en-anchura

En la integración primero-en-profundidad se integran todos los módulos de un camino de control principal de la estructura. Para seleccionar el camino principal se puede hacer de una manera arbitraria y dependerá de las características específicas de la aplicación.



Veámoslo con un ejemplo. En la figura observarán la estructura de un sistema que está compuesta de ocho módulos, si hacemos el recorrido primero-en-profundidad, eligiendo el camino a mano izquierda, se integrarán primero los módulos M1,M2, y M5. A continuación será integrado M8 y si es necesario para un buen funcionamiento de M2, se integrará M6. Posteriormente se integrarán los caminos de control central y derecho.

Si la integración es de la forma primero-en-anchura se incorporan todos los módulos directamente subordinados a cada nivel, moviéndose por la estructura de forma horizontal. Según la estructura del sistema que tenemos de ejemplo, los módulos que se integran primeramente son M2, M3 y M4. Sigue después el siguiente nivel de control M5,M6 y M7. Y así sucesivamente.



Bien, ahora les hablaré de la segunda estrategia de integración: la Integración Ascendente.

La prueba de integración ascendente, como su nombre implica, empieza la construcción y la prueba con los módulos **atómicos** (o sea, módulos de los niveles más bajos de la estructura del programa). Dado que los módulos son integrados de abajo hacia arriba, el procesamiento requerido de los módulos subordinados siempre está disponible y se elimina la necesidad de resguardos.

Una estrategia de integración ascendente puede ser implementada mediante los siguientes pasos:

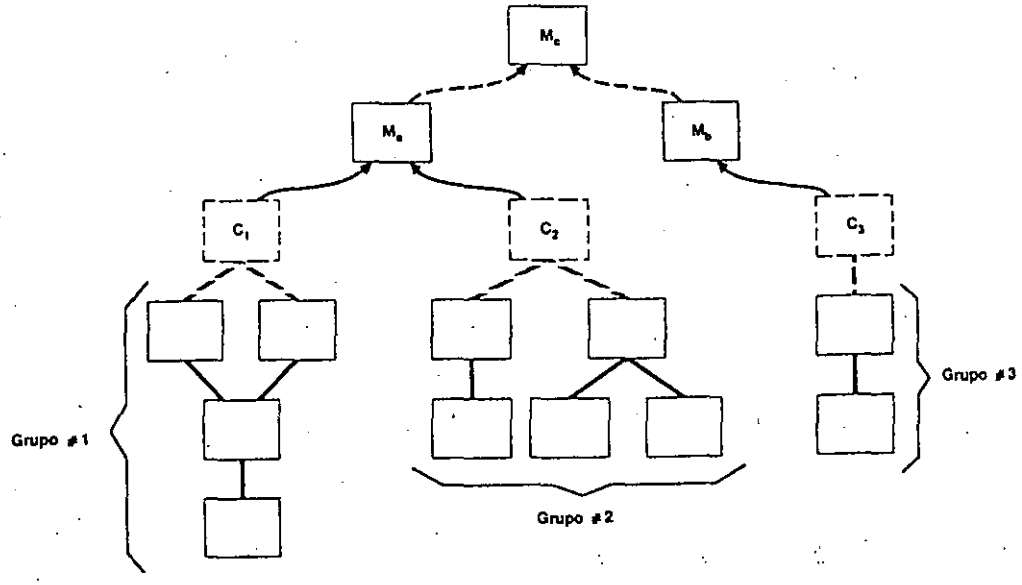
1o. Se combinan los módulos de bajo nivel en **grupos** (a veces denominados construcciones) que realicen una subfunción específica del software.

2o. Se escribe un conductor o sea, un programa de control de la prueba para coordinar la entrada y la salida de los casos de prueba.

3o. Se prueba el grupo

4o. Se eliminan los conductores y se combinan los grupos moviéndose hacia arriba por la estructura del programa.

Veámoslo a manera ilustrativa con el siguiente ejemplo: Siguiendo el esquema que se muestra en la figura, se combinan los módulos para formar los grupos 1, 2, y 3. Cada uno de los grupos se somete a prueba mediante un conductor que se muestra como un bloque punteado. Los módulos de los grupos 1 y 2 son subordinados de Ma. Se eliminan los conductores C1 y C2 y se conectan los grupos directamente a Ma. De forma similar se elimina el conductor C3 del grupo 3 antes de integrarlo con el módulo Mb. Finalmente, tanto Ma como Mb se integran en el Módulo Mc, y así sucesivamente.



A medida que la integración progresa hacia arriba, disminuye la necesidad de conductores de prueba separados. De hecho, si los niveles superiores del programa se integran de forma descendente, se puede reducir sustancialmente el número de conductores y se simplifica enormemente la integración de grupos.

Revisemos ahora la **Prueba de Validación.**

Tras la culminación de la prueba de integración, el software está completamente ensamblado como un paquete; se han encontrado y corregido los errores de interfaces, y debe comenzar una serie final de pruebas de software **la prueba de validación** y ésta se logra cuando el software funciona de acuerdo a las espectativas razonables del cliente. Para la prueba de validación tenemos que considerar 3 puntos.

Criterios para la prueba de validación.

Repaso de la configuración.

Pruebas alfa beta.

Por último pasemos a describir la Prueba del Sistema.

Una vez que el software es incorporado a otros elementos del sistema y se realiza una serie de pruebas de integración del sistema y de validación, se realizan las pruebas generales del Sistema.

Los pasos dados durante el diseño del software y durante la prueba pueden mejorar enormemente la probabilidad de éxito en la integración del software en el sistema total.

La prueba del sistema realmente está constituída por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en la computadora. Los tipos de prueba del sistema son:

Prueba de recuperación

Prueba de seguridad

Prueba de resistencia

Prueba de rendimiento

Bien, en este tema conocimos las pruebas que hay que aplicar ha nuestro sistema tanto en el momento de su desarrollo como cuando se ha integrado totalmente con la finalidad de obtener un sistema de alta calidad.

LECTURA: PRUEBAS DE UNIDAD.

En las pruebas de unidad se prueba la interfaz del módulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad del programa que está siendo probada. Se examinan las estructuras de datos locales para asegurar que los datos que se mantienen temporalmente conservan su integridad durante todos los pasos de ejecución del algoritmo. Se prueban las condiciones límite para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento. Se ejercitan todos los caminos independientes (caminos básicos) de la estructura de control con el fin de asegurar que todas las sentencias del módulo se ejecutan por lo menos una vez. Y finalmente, se prueban todos los caminos de manejo de errores.

Antes de iniciar cualquier otra prueba es preciso probar el flujo de datos de la interfaz del módulo. Si los datos no entran correctamente, todas las demás pruebas no tienen sentido. Myers, en su libro sobre prueba de software, propone una lista de comprobaciones para la prueba de interfaces:

1. ¿Es igual el número de parámetros de entrada al número de argumentos?
2. ¿Coinciden los atributos de los parámetros y los argumentos?
3. ¿Coinciden los sistemas de unidades de los parámetros y de los argumentos?
4. ¿Es igual el número de argumentos transmitidos a los módulos de llamada que el número de parámetros?
5. ¿Son iguales los atributos de los argumentos transmitidos a los módulos de llamada y los atributos de los parámetros?
6. ¿Son iguales los sistemas de unidades de los argumentos transmitidos a los módulos de llamada y de los parámetros?
7. ¿Son correctos el número, los atributos y el orden de los argumentos de las funciones incorporadas?
8. ¿Existen referencias a parámetros que no estén asociados con el punto de entrada actual?
9. ¿Entran sólo argumentos alterados?
10. ¿Son consistentes las definiciones de variables globales entre los módulos?
11. ¿Se pasan las restricciones como argumentos?

Cuando un módulo lleve a cabo E/S externa, se deben llevar a cabo pruebas de interfaz adicionales. De nuevo, de Myers:

1. ¿Son correctos los atributos de los archivos?
2. ¿Son correctas las sentencias de apertura?
3. ¿Cuadran las especificaciones de formato con las sentencias de E/S?
4. ¿Cuadra el tamaño del buffer con el tamaño del registro?
5. ¿Se abren los archivos antes de usarlos?
6. ¿Se tienen en cuenta las condiciones de fin de archivo?
7. ¿Se manejan los errores de E/S?
8. ¿Hay algún error textual en la información de salida?

Las estructuras de datos locales de cada módulo son una fuente potencial de errores. Se deben diseñar casos de prueba para descubrir errores de las siguientes categorías.

1. tipificación impropia o inconsistente
2. iniciación o valores implícitos erróneos

3. nombres de variables incorrectos (mal escritos o truncados)
4. tipos de datos inconsistentes
5. excepciones de desbordamiento por arriba o por abajo, o de direccionamiento

Además de las estructuras de datos locales, durante la prueba de unidad se debe comptobar (en la medida de lo posible) el impacto de los datos globales sobre el módulo (p.ej., el COMMON de FORTRAN).

Durante la prueba de unidad, la comprobación selectiva de los caminos de ejecución es una tarea escencial. Se deben diseñar casos de prueba para detectar errores debido a cálculos incorrectos, comparaciones incorrectas o flujo de control inapropiados. Las pruebas del camino básico y de bucles son técnicas muy efectivas para descubrir una gran cantidad de errores en los caminos.

Entre los errores más comunes en los cálculos estan: 1) precedencia incorrecta o mal interpretada; 2) operaciones de modo mixto; 3) inicializaciones incorrectas; 4) falta de precisión; 5) incorrecta representación simbólica de una expresión. Las comparaciones y el flujo de control están fuertemente emparejadas (p. ej., el flujo de control cambia frecuentemente tras una comparación). Los casos de prueba deben descubrir errores como: 1) comparaciones entre tipos de datos distintos; 2) operadores lógicos o precedencia incorrectos; 3) igualdad operada cuando los errores de precisión la hacen poco probable; 4) variables o comparadores incorrectos; 5) terminación de bucles inapropiada o inexistente; 6) fallo de salida cuando se encuentra una iteración divergente, y 7) variables de bucles modificadas de forma inapropiada.

Un buen diseño dicta que las condiciones de error sean previstas de antemano y que se dispongan unos caminos de manejo de errores que dirijan o terminen de una forma limpia el procesamiento cuando se dé un error.

Entre los errores potenciales que se deben comprobar cuando se evalúa la manipulación de errores están:

1. Descripción ininteligible del error
2. El error señalado no se corresponde con el error encontrado.
3. La condición de error hace que intervenga el sistema antes que el mecanismo de manejo de errores
4. El procesamiento de la condición excepcional es incorrecto.
5. La descripción del error no proporciona suficiente información para ayudar en la localización de la causa del error.

La prueba de límites es la última (y probablemente la más importante) tarea del paso de prueba de unidad. El software falla en sus condiciones límites. O sea, a menudo aparece un error cuando se procesa el elemento n-simo de un arreglo n-dimensional, cuando se hace la i-ésima repetición de un bucle de i pasos o cuando se encuentra los valores máximo o mínimo permitidos. Los casos de prueba que ejerciten las estructuras de datos, el flujo de control y los valores de los datos por debajo, en y por encima de los máximos y los mínimos son muy apropiados para descubrir estos errores.

LECTURA: PRUEBAS DE VALIDACION.

La validación del software se consigue mediante una serie de pruebas que demuestran la conformidad con los requerimientos. Un plan de prueba traza las pruebas que se han de llevar a cabo, y un procedimiento de prueba define los casos de prueba específicos que serán usados para demostrar la conformidad con los requerimientos. Tanto el plan como el procedimiento estarán diseñados para asegurar que se satisfacen todos los requerimientos funcionales, que se alcanzan todos los requerimientos de rendimiento, que la documentación es correcta e inteligible y que se alcanzan otros requerimientos (p. ej., portabilidad, compatibilidad, recuperación de errores, facilidad de mantenimiento).

Una vez que se procede con cada caso de prueba de validación, puede darse una de dos condiciones: 1) Las características de funcionamiento o de rendimiento están de acuerdo con las especificaciones y son aceptables, o 2) Se descubre una desviación de las especificaciones y se crea una lista de deficiencias. Las desviaciones o errores descubiertos en esta fase del proyecto raramente se pueden corregir antes de terminar el plan. A menudo es necesario negociar con el cliente un método para resolver las deficiencias.

Repaso de la configuración

Un elemento importante del proceso de validación es el repaso de la configuración. El repaso intenta asegurar que todos los elementos de la configuración del software se han desarrollado de forma adecuada, están catalogados y tienen el suficiente detalle para facilitar la fase de mantenimiento dentro del ciclo de vida del software.

Pruebas alfa y beta

Es virtualmente imposible que un encargado del desarrollo del software pueda prever cómo un cliente usará realmente un programa. Se pueden interpretar mal las instrucciones de uso, se pueden usar regularmente extrañas combinaciones de datos y una salida que pueden estar clara para el que realiza la prueba puede resultar ininteligible para un usuario normal.

Cuando se construye software a medida para un cliente, se lleva a cabo una serie de pruebas de aceptación para permitir que el cliente valide todos los requerimientos. Llevado a cabo por el usuario final en lugar del equipo de desarrollo, una prueba de aceptación puede ir desde un informal "paso de prueba" hasta la ejecución sistemática de una serie de pruebas bien planificadas. De hecho, la prueba de aceptación puede tener lugar a lo largo de semanas o meses, descubriendo así errores acumulados que pueden ir degradando el sistema.

Si el software se desarrolla como un producto a ser usado por muchos clientes, no es práctico realizar pruebas de aceptación formales para cada uno de ellos. La mayoría de los constructores de productos de software llevan a cabo un proceso denominado prueba alfa y beta para descrubrir errores que parezcan que sólo el usuario final puede descubrir.

La prueba alfa es conducida por un cliente en el lugar de

desarrollo. Se usa el software de forma natural, con el encargado del desarrollo "mirando por encima del hombro" del usuario y registrando errores y problemas de uso. Las pruebas alfa se llevan a cabo en un entorno controlado.

La prueba beta se lleva a cabo en uno o más lugares de clientes por los usuarios finales del software. A diferencia de la prueba alfa, el encargado del desarrollo normalmente no está presente. Así, la prueba beta es una aplicación "en vivo" del software en un entorno que no puede ser controlado por el equipo de desarrollo. El cliente registra todos los problemas (reales o imaginarios) que encuentra durante la prueba beta e informa a intervalos regulares al equipo de desarrollo. Como resultado de los problemas anotados durante la prueba beta, el equipo de desarrollo del software lleva a cabo modificaciones y así prepara una versión del producto de software para toda la base de clientes.

LECTURA: PRUEBA DEL SISTEMA.

Prueba de recuperación

Muchos sistemas basados en computadora deben recuperarse de los fallos y resumir el procesamiento en un tiempo previamente especificado. En algunos casos un sistema debe ser tolerante a fallos de procesamiento no deben hacer que cese el funcionamiento de todo el sistema. En otros casos, se debe corregir un fallo del sistema en un determinado período de tiempo a riesgo de que se produzca un serio daño económico.

La prueba de recuperación es una prueba del sistema que fuerza el fallo del software de muchas formas y verifica que la recuperación se lleva a cabo apropiadamente. Si la recuperación es automática (llevada a cabo por el propio sistema) hay que evaluar la corrección de la reinicialización, de los mecanismos de recuperación del estado del sistema, de la recuperación de datos y del arranque. Si la recuperación requiere la intervención humana, hay que evaluar los tiempos medios de reparación para determinar si están dentro de unos límites aceptables.

Prueba de seguridad

Cualquier sistema basado en computadora que maneje información sensible o lleve a cabo acciones que puedan impropiamente perjudicar (o beneficiar) a los individuos es objeto de penetraciones impropias o ilegales. La penetración incluye un amplio rango de actividades: "destripadores" que intentan penetrar sistemas como deporte, empleados disgustados que intentan penetrar por venganza e individuos deshonestos que intentan penetrar por obtener ganancias personales ilícitas.

La prueba de seguridad intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán, de hecho, de la penetración impropia. Citando a Beizer [BEI84]: "Por supuesto, la seguridad del sistema debe ser probada en su invulnerabilidad frente a un ataque frontal pero también debe ser probada en su invulnerabilidad frente a ataques por los flancos o por la retaguardia".

Durante la prueba de seguridad, el encargado de la prueba juega el papel de un individuo que desea penetrar en el sistema. ¡Todo vale! Debe intentar hacerse con las claves de acceso por cualquier medio externo del oficio; debe atacar al sistema con software a medida, diseñado para romper cualquier defensa que haya sido construida; debe bloquear el sistema, negando así el servicio a otras personas; debe producir a propósito errores del sistema, intentando penetrar durante la recuperación; o debe curiosear en los datos públicos, intentando encontrar la clave de acceso al sistema.

Con el suficiente tiempo y los suficientes recursos, una buena prueba de seguridad terminará por penetrar en el sistema. El papel del diseñador del sistema es hacer que el coste de penetración sea mayor que el valor de la información obtenida mediante la penetración.

Prueba de resistencia

Durante los anteriores pasos de prueba, la técnica de la caja

blanca y de la caja negra daban como resultado la evaluación del funcionamiento y del rendimiento normales del programa. Las pruebas de resistencia están diseñadas para enfrentar a los programas con situaciones anormales. En esencia, el sujeto que realiza la prueba de resistencia se pregunta: "¿A qué potencia puedo ponerlo a funcionar antes de que falle?

La prueba de resistencia ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales. Por ejemplo: 1) diseñar pruebas especiales que generan diez interrupciones por segundo, mientras que una o dos son las normales; 2) incrementar las frecuencias de datos de entrada en un orden de magnitud con el fin de comprobar cómo responden las funciones de entrada; 3) ejecutar casos de prueba que requieran el máximo de memoria o de otros recursos; 4) diseñar casos de prueba que puedan dar problemas con el esquema de gestión de memoria virtual; 5) diseñar casos de prueba que produzcan excesivas búsquedas de datos residentes en disco. Esencialmente, el encargado de la prueba intenta tirar abajo el programa.

Una variación de la prueba de resistencia es una técnica denominada prueba de sensibilidad. En algunas situaciones (la más común se da con algoritmos matemáticos) un rango muy pequeño de datos dentro de los límites de una entrada válida para un programa puede producir un procesamiento extremo e incluso erróneo o una profunda degradación del rendimiento. Esta situación es análoga a la existencia de una signularidad en una función matemática. La prueba de sensibilidad intenta descubrir combinaciones de datos dentro de una clase de entrada válida que pueda producir inestabilidad o procesamiento incorrecto.

Prueba de rendimiento

Para sistemas de tiempo real o sistemas empotrados, el software que proporciona las funciones requeridas pero que no se ajusta a los requerimientos de rendimiento es inaceptable. La prueba de rendimiento está diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado. La prueba de rendimiento se da durante todos los pasos del proceso de prueba. Incluso al nivel de unidad, se debe asegurar el rendimiento de los módulos individuales a medida que se llevan a cabo las pruebas de la caja blanca. Sin embargo, hasta que no están completamente integrados todos los elementos del sistema no se puede asegurar realmente el rendiminento del sistema.

Las pruebas de rendimiento a menudo van emparejadas con las pruebas de resistencia y a menudo requieren instrumentación tanto de software como de hardware. O sea, a menudo es necesario medir la utilización de recursos (p. ej., ciclos de procesador) de una forma exacta. La instrumentación externa puede monitorizar los intervalos de ejecución, los sucesos ocurridos (p. ej., interrupciones) y muestras de los estados de la máquina en un funcionamiento normal. Instrumentando un sistema, el encargado de la prueba puede descubrir situaciones que lleven a degradaciones y posibles fallos del sistema.

LECTURA: MANTENIMIENTO DEL SISTEMA.

Mantenimiento estructurado frente al no estructurado

Si sólo se dispone del código fuente como elemento de configuración, la actividad de mantenimiento comienza con una dolorosa evaluación del código, a menudo complicada por la pobre documentación interna. Las sutiles caractrísticas, tales cómo la estructura del programa, las estructuras de datos globales, las interfaces del sistema, el rendimiento y/o las limitaciones del diseño, son dificiles de descubrir y frecuentemente mal interpretadas. Es difícil asegurar cuáles son las ramificaciones de cambios llevados últimamente en el código. Es imposible llevar a cabo pruebas de regresión (repetir prueba anterior para asegurar que las modificaciones no han introducido fallos en el software previamente operativo), ya que no existe ningún registro de pruebas. Lo que hacemos es un mantenimiento no estructurado y pagamos el precio (en esfuerzo desperdiciado y en frustraciones personales) que se adjunta a todo software que no ha sido desarrollado mediante una metodología bien definida.

Si existe una completa configuración del software, la tarea de mantenimiento comienza con una evaluación de la documentación del sideño. Se determinan las importantes caractrísticas estructurales, de rendimiento y de interfaz del software. Se estudia el impacto de las correciones o modificaciones requeridas y se traza un plan de actuación. Se modifica el diseño (usando técnicas idénticas a las discutidas en capítulos anteriores) y se revisa. Se desarrolla nuevo código fuente, se realizan pruebas de regresión mediante la información contenida en la Especificación de Prueba y se vuelve a lanzar el software.

Esta secuencia de sucesos constituye el mantenimiento estructurado y aparece como resultado de una anterior aplicación de una metodología de ingeniería del software. Aunque la existencia de una configuración del software no garantiza un mantenimiento libre de problemas, se reduce la cantidad de esfuerzo requerido y se mejora la calidad general del cambio o de la corrección .

Costos de mantenimiento

El costo del mantenimiento de software ha crecido rápidamente durante los últimos veinte años. Los porcentajes de factura total del software que se gastan en el mantenimiento de software existente son: para los años 70's de 35-40%, para los años 80's de 40-60% y se estima que para los años 90's será de 70-80%. Aunque los promedios industriales son difíciles de asegurar y están abiertos a muchas interpretaciones, una organización de desarrollo de software típica gasta entre el 40 y el 70 por ciento de todo su dinero en el mantenimiento correctivo, adaptivo, perfectivo y preventivo.

Los costos en dólares del mantenimiento son los más obvios. Sin embargo, otros cotos, menos tangibles, pueden en último lugar ser la causa de muchas preocupaciones. Citando a Daniel McCracken [MCC80]:

> Los retrasos en las nuevas ampliaciones y en los grandes cambios medidos en años se van alargando. Como industria, ni siquiera podemos hacer frente -digamos sólo poner al día- a lo que nuestros usuarios quieren que hagamos.

McCracken alude a la organización en la barrera del mantenimiento. Un costo intangible del mantenimiento del software viene dado por una oportunidad de desarrollo que se propone o se pierde debido a que los recursos disponibles deben estar dedicados a las tareas de mantenimiento. Otros costos intangibles incluyen:

- insatisfacción del cliente cuando una petición de reparación o de modificación aparentemente legítima no se puede antender en un tiempo razonable
- disminución de la calidad global del software debido a los errores latentes que introducen los cambios en el software mantenido
- trastornos en otros esfuerzos de desarrollo al tener que "poner" a trabajar a la plantilla en tareas de mantenimiento.

El costo final del mantenimiento de software es una dramática reducción de la productividad (medida en LDC por persona-mes o en punto de función por persona mes) que se encuentra cuando se inicia el mantenimiento de viejos programas. Se ha informado de reducciones de la productividad de 40 a 1 [BOE79]. O sea, un esfuerzo de desarrollo que haya resultado en un costo de $25 por línea de código desarrollada, costaría $1.000 por cada línea de código que sea mantenida.

El esfuerzo gastado en el mantenimiento se puede dividir en actividades productivas (p.el., análisis y evaluación, modificación del diseño, codificación) y actividades "menos productivas" (p. eje., tratar de comprender lo que es el código; tratar de interpretar las estructuras de datos, las características de la interfaz, los límites de rendimiento). La siguiente expresión [BEL72] proporciona un modelo del esfuerzo de mantenimiento:

$$M = p + K \exp(c-f)$$

donde M = esfuerzo total gastado en el mantenimiento
p = esfuerzo productivo (descrito arriba)
k = una constante empírica
c = una medidad de la complejidoad que se puede atribuir a la falta de un buen diseño y de documentación
f = una medida del grado de familiaridad con el software.

Este modelo indica que el esfuerzo (y costo) puede aumentar exponencialmente si se usa una pobre aproximación de desarrollo de software (o sea, una falta de ingeniería del software) y si la persona o grupo que usan la aproximación no está disponible para llevar a cabo el mantenimiento.

Problemas

La mayoría de los problemas asociados con el mantenimiento de software se debe a las deficiencias de la forma en que el software ha sido definido y desarrollado. Aquí aparece el clásico sindrome del pague ahora o pague después" La falta de control y disciplina en las dos primeras fases del proceso de ingeniería del software

casi siempre se traduce en problemas para la última fase.

Entre los muchos problemas clásicos asociados con el mantenimiento de software se encuentran los siguientes:

- A menudo es excepcionalmente dificil comprender un programa "ajeno". A medida que existen menos elementos de configuración del software, mayor es la dificultad. Si sólo existe el código indocumentado, es de esperar que aparezcan serios problemas.
- Ese personaje "ajeno" a menudo no se encuentra alrededor para poder explicarse. La movilidad entre los profesionales del software es actualmente muy grande. No podemos esperar una explicación personal del software por el que lo ha desarrollado una vez que se requiere el mantenimiento.
- No existe una documentación apropiada o está mal preparada. Un primer paso es el reconocimiento de que el software debe ser documentado, pero para que la documentación sea de algún valor debe ser comprensible y consistente con el código fuente.

- La mayoría del software no ha sido diseñado previendo el cambio. A menos que el método de diseño prevea el cambio mediante conceptos tales como independencia funcional o clases de objetos, las modificaciones del software serán dificiles y propensas a errores.

- El mantenimiento no se ve como un trabajo atractivo. Muchas de las causas de esto vienen dadas por el alto nivel de frustración asociado con el trabajo de mantenimiento.

Todos los problemas que se acaban de describir se pueden, en parte, atribuir al gran número de programas actualmente existentes que ha sido desarrollados sin tener en cuenta la ingeniería del software. Tampoco se debe ver como una panacea el uso de una metodología disciplinada. Sin embargo, la ingeniería del software por lo menos proporciona soluciones parciales a cada problema asociado con el mantenimiento.

Como consecuencia de los problemas asociados con el mantenimiento de software, surgen varias cuestiones técnicas y de organización. ¿Es posible desarrollar software que esté bien diseñado y sea fácil de mantener? ¿Podemos mantener la integridad del software cuando tiene que ser modificado? ¿Existen enfoques técnicos y organizativos que se puedan aplicar con éxito al mantenimiento de software? En las secciones que vienen a continuación se discuten estos y otros puntos.

FUNCIONES PARA CONVERSION DE TIPOS DE DATOS

```
(angtos  <ángulo>  [<modo>  [<precisión>]] )
(ascii  <cadena> )
(atof  <cadena> )
(atoi  <cadena> )
(chr  <num> )
(fix  <num> )
(float  <num> )
(itoa  <entero> )
(rtos  <num>  [<modo>  [<precisión>]] )
(type  <elem> )
```

FUNCIONES PARA ARCHIVOS

```
(close  <apunt-arch> )
(findfile  <nombre-arch> )
(load  <nombre-arch>  [<onerror>] )
(open  <nombre-arch>  <modo> )
(prin1  <expr>  [<apunt-arch>] )
(princ  <expr>  [<apunt-arch>] )
(print  <expr>  [<apunt-arch>] )
(read-char  [<apunt-arch>] )
(read-line  [<apunt-arch>] )
(write-char  <num>  [<apunt-arch>] )
(write-line  <cadena>  [<apunt-arch>] )
```

FUNCIONES DE CONTROL DE FLUJO

```
(cond  (<expr-cond1>  <expresión1>  ...)  ... )
(if  <expr-cond>  <expr1>  [<expr2>] )
(progn  <expr>  ... )
(repeat  <number>  <expr>  ... )
(while  <expr-cond>  <expresión>  ... )
```

FUNCIONES PARA ENTRADA DE USUARIO

```
(getangle [<pt>] [<mensaje>] )
(getcorner <pt> [<mensaje>] )
(getdist [<pt>] [<mensaje>] )
(getint [<mensaje>] )
(getkword [<mensaje>] )
(getorient [<pt>] [<mensaje>] )
(getpoint [<pt>] [<mensaje>] )
(getreal [<mensaje>] )
(getstring [<cod-blancos>] [<mensaje>] )
```

FUNCIONES DE RELACION

```
(= <átomo> <átomo> ... )
(/= <átomo1> <átomo2> )
(< <átomo> <átomo> ... )
(<= <átomo> <átomo> ... )
(> <átomo> <átomo> ... )
(>= <átomo> <átomo> ... )
(eq <expr1> <expr2>)
(equal <expr1> <expr2> [<tol>])
```

FUNCIONES LOGICAS

```
(and <expresión> ... )
(not <expresión> )
(null <expresión> )
(or <expr> ... )
```

Para manejo de BITS:

```
(~ <num> )
(logand <num> <num> ... )
(logior <num> <num> ... )
(lsh <num1> <numbits> )
```

FUNCIONES PARA CALCULO DE DATOS

```
(angle <pt1> <pt2> )
(distance <pt1> <pt2> )
(inters <pt1> <pt2> <pt3> <pt4> [<prolong>] )
(osnap <pt> <cadena-modo> )
(polar <pt> <ángulo> <distancia> )
(trans <pt> <sis-origen> <sis-destino> [<desplaz>] )
```

FUNCIONES PREDEFINIDAS DE AUTOLISP

FUNCIONES ARITMETICAS

```
(+  <num>  <num>  ... )
(-  <num>  <num>  ... )
(*  <num>  <num>  ... )
(/  <num>  <num>  ... )
(1+  <num> )
(1-  <num> )
(abs  <num> )
(atan  <num1>  [<num2>] )
(cos  <ángulo> )
(exp  <num> )
(expt  <base>  <potencia> )
(gcd  <num1>  <num2> )
(log  <num> )
(max  <num>  <num>  ... )
(min  <num>  <num>  ... )
(minusp  <num> )
pi
(rem  <num1>  <num2>  ... )
(sin  <ángulo> )
(sqrt  <num> )
(zerop  <elem> )
```

FUNCIONES PARA EL MANEJO DE LISTAS

```
(append  <lista>  ... )
(assoc  <elem>  <lista> )
(car  <lista> )
(cdr  <lista> )
(caar,cadr,cddr,cadar, etc. )
(cons  <nuevo elemento>  <lista> )
(last  <lista> )
(length  <lista> )
(list  <expr>  ... )
(listp  <elem> )
(member  <expr>  <lista> )
(nth  <n>  <lista> )
(reverse  <lista> )
(subst  <nuevoelem>  <viejoelem>  <lista> )
```

FUNCIONES PARA EL MANEJO DE CADENAS

```
(strcase  <cadena>  [<cómo>] )
(strcat  <cadena1>  <cadena2>  ... )
(strlen  <cadena> )
(substr  <cadena>  <inicio>  [<longitud>] )
```

OTRAS FUNCIONES

```
(apply <función> <lista> )
(atom <elem> )
(boole <func> <entero1> <entero2> ... )
(boundp <átomo> )
(command <args> ... )
(defun <sim> <lista argumentos> <expr> ... )
(eval <expr>)
(foreach <nom> <lista> <expr> ... )
(getenv <nom-var-amb> )
(getvar <nom-var-acad> )
(graphscr)
(initget [<bits>] <cadena> )
(lambda <args> <expr> ... )
(mapcar <función> <lista1> ... <listan> )
(menucmd <cadena> )
(numberp <elem> )
(prompt <mensaje> )
(quote <expr> )
(read <cadena> )
(redraw [<ename> [<modo>]] ))
(set <sim> <expr> )
(setq <sim1> <expr1> [<sim2> <expr2>] ... )
(setvar <nom_var-acad> <valor> )
(terpri)
(textscr)
(trace <función> ... )
(untrace <función> ... )
(ver)
(vports)
(*error* <cadena> )
```