ESCUELA SUPERIOR POLITECNICA
DEL LITORAL
ESCUELA DE COMPUTACION

79

CONFIABILIDAD DE SOFTWARE

Leonor Isabel Amador Gonzalez

Oriana Alexandra Rumbea Thomas

Director de Proyecto:

Ing. Xavier Salinas Andrade

Guayaquil, 25 de Mayo de 1983

79

CONTENIDO

CONTENIDO

CONFIABILIDAD DE SOFTWARE

INTRODUCCION

La industria de la computación ha pasado por algunas fases. Durante la primera generación de computadores, los problemas en la programación fueron culpados a las severas restricciones impuestas por el hardware existente en ese entonces. Las aplicaciones eran largos procesos en lotes y programas científicos donde la eficiencia era la meta mas importante. Los programadores se preocupaban por aprovechar cada segundo para que el programa completara su ejecución, antes de que el hardware fallara. Desde entonces han habido tremendos avances tecnológicos en el campo del hardware.

La confiabilidad y velocidad del hardware, se ha incrementado en gran magnitud en la actualidad. La mayoría de las aplicaciones actualmente son en línea, controlando procesos críticos y se han convertido en la parte integral de las operaciones diarias de muchas empresas.

Aunque el hardware actualmente es mucho mas flexible, los problemas con la programación no han disminuído; al contrario, ellos son peores de lo que eran antes. En vista de que la velocidad y la confiabilidad del hardware se ha incrementado en gran magnitud, las restricciones físicas del hardware han sido reemplazadas por las restricciones invisibles de la capacidad de la mente humana. El problema mas significativo que encara el procesamiento de datos actualmente es el problema del software inconfiable.

La confiabilidad del software es definida como la probabilidad de que el software se ejecute por un período de tiempo particular sin un error, medida por el costo del usuario por cada error encontrado. La confiabilidad de software es así una función del impacto que los errores tienen sobre el sistema. La confiabilidad no es una propiedad inherente del programa, sino que está ampliamente relacionada con la manera en la cual un programa es usado.

Existen considerables implicaciones económicas relacionadas con la inconfiabilidad de software. Durante la implementación de un sistema de computación, más recursos son comsumidos en el software que en el hardware.

Los costos directos del software inconfiable, son una cantidad substancial de todos los costos del computador. Si además los costos indirectos, debido a los errores, son considerados, entonces el significado económico del software inconfiable es aún más pronunciado.

Hay ciertos factores que inciden en la confiabilidad y se manifiestan durante el ciclo de vida del software. El desarrollo del software incluye una serie de tareas que deben llevarse a cabo, que se explican a continuación:

La confiabilidad de un Sistema es esencialmente determinada durante la etapa de Diseño. Si el Diseño del sistema es pobre, esto es, sin una estructura buena y entendible, entonces en la mayoría de los casos no es posible proveer un producto altamente confiable.

El Diseño es una actividad muy extensa. Cubre las áreas de software comenzando con el establecimiento de requerimientos y objetivos y finalizando con el diseño de los módulos del programa. El desarrollo de un buen Diseño puede costar más tiempo y dinero, pero permite enormes ahorros durante la implementación y la vida productiva del sistema. Además puede mejorar la confiabilidad y reducir el costo del hardware y de los errores.

Existen ciertas metodologías que deben ser usadas durante la etapa de diseño, para obtener programas claros y entendibles, estas son: programación estructurada, diseño de arriba hacia abajo y refinamiento paso a paso.

La estructura de un sistema es fundamental para entenderlo y percibir su complejidad. La complejidad tiene una influencia decisiva sobre la probabilidad de que un error sea hecho durante la etapa de diseño. La confiabilidad puede así ser incrementada por una estructura clara y simple, limitada por la habilidad intelectual del cerebro humano. Si el sistema es difícil de entender, este ya contiene la semilla de la inconfiabilidad.

Una vez concluída la etapa de diseño, es necesario transformar los objetivos y requerimientos en sentencias entendibles por el computador mediante la codificación.

El estilo de programación tiene un efecto decisivo en la codificación del sistema. Existen muchas formas en las cuales un programa puede ser escrito. El estilo de programación individual es el que finalmente decide cuales alternativas son escogidas, si el programa es comprensible o no y la legibilidad y complejidad del mismo. La regla mas importante del estilo de programación es producir programas que sean simples, claros y manejables. El lenguaje de programación usado juega un papel importante en determinar el estilo individual. La elección del lenguaje se hará en base a sus atributos, que permitan lograr los objetivos propuestos durante la etapa de diseño.

Cuando se ha concluído la codificación, aún no se ha terminado la creación y producción de software, es necesario encontrar y corregir los errores presentes. Esta es la etapa de prueba y depuración. La prueba determina que un error existe, la depuración localiza la causa del error, pues siempre comienza con la evidencia de que el programa falla.

La prueba del software involucra un rango de actividades que son similares a la secuencia del desarrollo de sofware. Estas actividades incluyen: establecer objetivos de prueba, diseñar y escribir la prueba, ejecutarla y examinar los resultados de la misma. Existen algunas metodologías de prueba desarolladas como prueba de arriba hacia abajo, prueba de abajo hacia arriba, entre otras.

Una vez que la prueba muestra que existe algun error, el siguiente paso es determinar su localizacion precisa y corregirlo. Este es el proceso de la depuración y es quizás uno de los mas difíciles, pues en algunos casos, los síntomas se confunden con la verdadera causa del error. Aquí es necesario conocer los diferentes tipos de errores que existen y la forma en que mas comunmente se producen. Es por estos motivos que una gran parte del esfuerzo usado en el desarrollo y mantenimiento del software, es gastado en la prueba y depuración.

Cuando estas etapas son concluídas se ha completado la producción del programa. Si posteriormente surgen otras necesidades y se desean introducir cambios en los programas, se entra en la etapa de mantenimiento. Esta etapa esta dividida en dos categorías: actualización de software y reparación de software. Los programas por tanto deben ser portables y adaptables. Portables para poder ser transferidos a otro ambiente de ejecución y adaptables para que la estructura de sus algoritmos pueda ser cambiada.

DISEÑO

DISEÑO

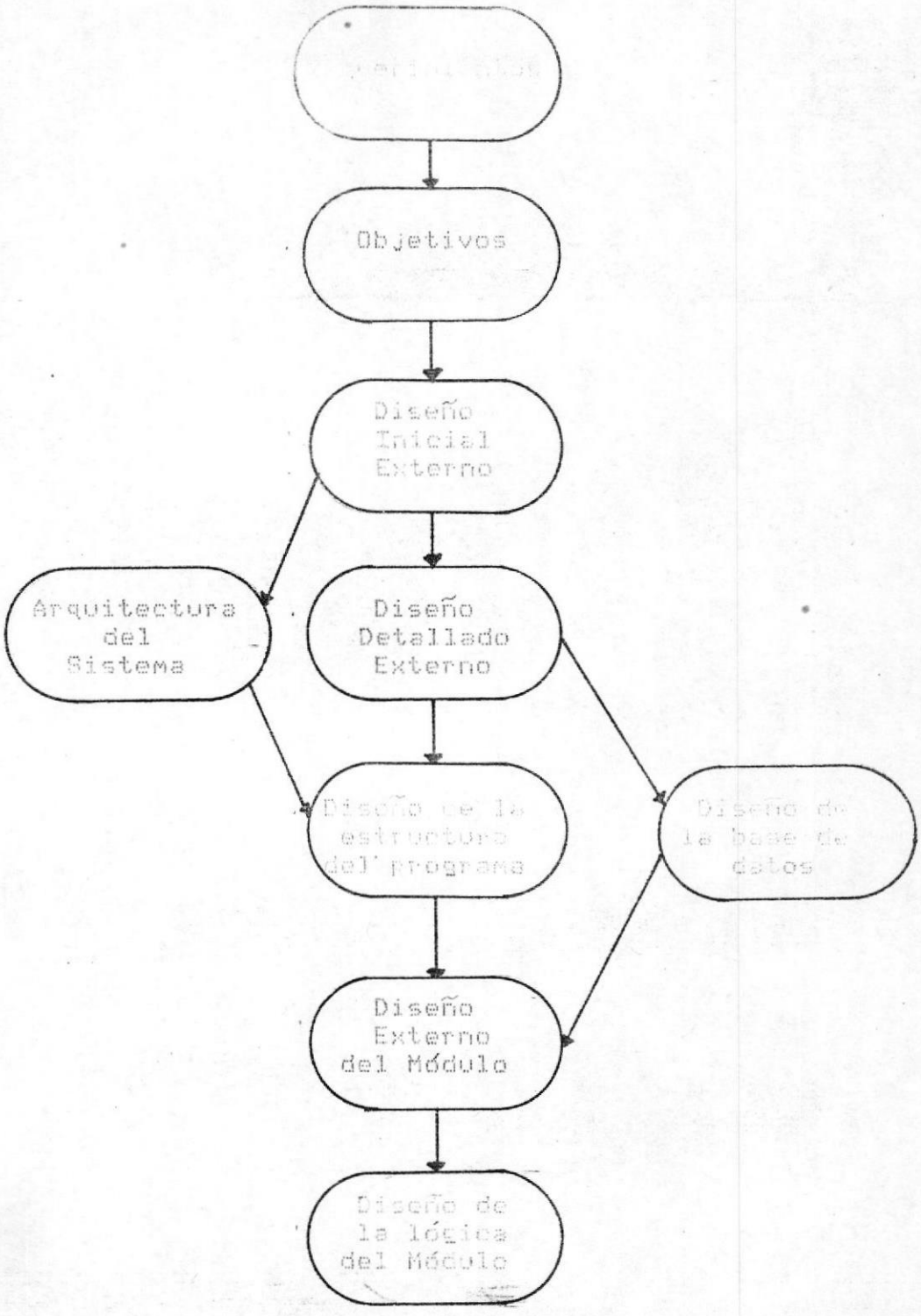
Muchas organizaciones restringen arbitrariamente el diseño a una descripción temprana de las actividades del software y usan palabras como implementación, desarrollo y programación para definir y describir actividades posteriores. Diferentes organizaciones agrupan diferentes procesos bajo estas denominaciones lo cual conduce a la confusión cuando tratan de comparar dos sistemas. El Diseño cubre las actividades de software comenzando con el establecimiento de requerimientos y objetivos y finalizando con la escritura de sentencias de programas para que así reconozcamos que hay diferentes estados de diseño.

Tal vez la razón mas importante para diseñar es que la creación de sistemas complejos involucra una gran cantidad de detalle y complejidad. Si la complejidad no es controlada entonces los resultados deseados rara vez seran conseguidos.

Otra razón es su impacto en la calidad del sistema. Se necesitan sistemas que sean confiables, centralizados al usuario, eficientes y portables, entre otras cualidades. Si las funciones del usuario y la modularidad del sistema, no son planeadas antes que la programación haya comenzado, entonces generalmente la confiabilidad sera inalcanzable, porque si no se considera lo que el usuario requiere del sistema, y se planea como se van a desarrollar las diferentes fases del sistema, no se cumplirán los objetivos del mismo.

ESTADOS DE DISEÑO DE UN PROYECTO

En la siguiente figura se muestran los estados del diseño para un sistema grande. Nótese que el modelo es una metodología independiente, esto quiere decir que todas las actividades mostradas en el gráfico deben ocurrir de tal forma que cualquier esfuerzo desarrollado sea independiente del lenguaje de programación usado, de la forma en que el usuario escribió los requerimientos inciales, del uso de programación estructurada o no, etc.



ESTADOS DE DISEÑO DE UN PROYECTO

El primer paso produce un conjunto de requerimientos, definidos como lo que el usuario espera del sistema. El siguiente paso esta relacionado con los objetivos impuestos, los cuales son las metas del sistema. El tercer paso es un diseño externo de alto nivel, el cual define la relación del usuario con el sistema, pero incluye muchos de los detalles tales como formatos exactos de entrada y salida. El diseño inicial externo conduce a dos procesos en paralelo, el proceso de diseño detallado externo que completa la interfase del usuario describiendola en más detalle, y el proceso de arquitectura del sistema, que descompone el sistema en un conjunto de programas y subsistemas y define las relaciones entre ellos. Estos dos pasos conducen al proceso de diseño de la estructura del programa, este proceso toma cada programa componente o subsistema y diseña sus módulos, sus interfases e interconexiones.

El siguiente proceso es un diseño externo del módulo que es la definición precisa de todas las relaciones de los módulos, y el último paso es el diseño de la lógica del módulo, este es el diseño de la lógica interna de cada módulo en el sistema incluyendo las sentencias del programa.

El restante proceso es el diseño de la base de datos, este es el proceso de definir cualquier estructura de datos externa al software, por ejemplo el diseño de los registros de los archivos o de los registros de la base de datos.

REQUERIMIENTOS, OBJETIVOS Y ESPECIFICACIONES

Los errores del software están presentes cuando el software no hace lo que el usuario razonablemente espera que haga, y se introducen en un sistema cuando los requerimientos y objetivos son establecidos. La causa de la mayoría de estos errores es la falta de entendimiento de las necesidades del usuario por parte de los programadores.

Mas errores se originan cuando los requerimientos y objetivos son traducidos a especificaciones externas, es decir cuando son expresados en función del tipo de entrada/salida, características del sistema, etc.

Desafortunadamente muy pocas personas hacen énfasis en esos procesos. Esto R.D. William [3] lo resume así "No hay nada, absolutamente nada, que pueda causar un resultado más devastador en una actividad de desarrollo que un incompleto y/o incorrecto conocimiento y especificaciones correspondientes del sistema y los requerimientos del software".

ESTABLECER REQUERIMIENTOS

El propósito de los requerimientos del software es establecer las necesidades del usuario para un particular sistema.

Los requerimientos para un sistema mediano o largo podrían ser desarrollados por un equipo de individuos. Un miembro del equipo debe ser un representante de la organización que requiere el proyecto, con suficiente autoridad para tomar decisiones. Pero sin embargo esta persona no es el usuario final del sistema, por esto el segundo participante en el equipo debe ser un individuo que vaya a usar el sistema cuando se encuentre desarrollado.

Otro representante podría ser una persona que eventualmente tenga un papel clave en el proceso de diseño interno. La meta aquí en términos de confiabilidad es asegurarnos de que los requerimientos del usuario son especificados tan exacta y precisamente como sea posible y asegurarse que el grupo de personas involucradas en el desarrollo del software pueda traducir estos requerimientos al diseño de un sistema con un mínimo número de errores.

El proceso de establecer requerimientos involucra el análisis de sistemas existentes, entrevistas con el usuario, estudios de factibilidad y beneficios estimados. Técnicas para estas actividades serán descritas posteriormente.

Un método manual y semiformal para describir requerimientos es la técnica HIPO (Hierarchy plus Input-Process-Output). Un diagrama HIPO es dibujado para cada función importante requerida. El diagrama destaca la entrada y salida de los datos para la función y los pasos básicos del proceso.

Otra técnica usada es la tabla visual de contenido, la cual es preparada mostrando la organización jerárquica de esas funciones. Con ella los diagramas son normalmente bosquejados por la organización de personas encargadas del desarrollo del software y presentados al ususario para su inspección. Su objetivo es proveer al usuario con una visión gráfica del sistema, permitiendole comprobar si realmente se ha hecho lo que el deseaba.

OBJETIVOS DEL SOFTWARE

Los objetivos son metas específicas para el sistema. Establecer objetivos de software es primeramente un proceso de establecer cambios, por tanto el propósito de los objetivos es establecer metas para el sistema y cambios entre esas metas donde sean necesarios.

El desarrollo de software necesita dos distintos conjuntos de objetivos:

Objetivos del Sistema, los cuales definen las metas del sistema desde el punto de vista del usuario, tales como: sumario de las actividades a desarrollarse, definición del usuario.

Objetivos del Proyecto, los cuales son metas que se fijan antes del desarrollo del proceso, tales como esquemas, costo, grado de prueba, etc.

Errores comunes cometidos en el proceso de fijar objetivos son:

1.- No haber escrito los objetivos.

2.- Tener un conjunto de objetivos, escritos rapidamente, donde los objetivos específicos son dejados sin plantear.

3.- Establecer objetivos para el producto de tal forma que unos estén en conflicto con otros. El resultado es que los programadores del sistema resolverán por sí solos los conflictos y cada programador lo hará de diferente forma, resultando una salida impredecible.

4.- Reconocer las necesidades de objetivos específicos, pero definiendo solamente objetivos del sistema faltando definir los objetivos para el proyecto.

Los objetivos del software pueden ser agrupados en 10 categorías mayores, la confiabilidad es una de ellas. A continuación se explicarán las relaciones entre la confiabilidad y las otras nueve categorías. Algunos autores nos dan evidencias al respecto basados en sus propias experiencias.

1.- Generalidad

Es la medida del número y alcance de las funciones del usuario, esto es , mientras mas funciones esté en capacidad de desarrollar el sistema, va a ser mucho mas general. La confiabilidad y generalidad están en conflicto, porque sistemas generalizados son normalmente largos y mas complejos. La solución a este conflicto no implica que el software debe ser extremadamente especializado, sin embargo este conflicto puede ser disminuído evitando cualquier generalización del sistema que sea poca o no beneficiosa al usuario.

Uno debe recordar que cada elemento del sistema que hace interfase con el usuario, aunque sea pequeño, tiene algún efecto negativo en la confiabilidad, esto quiere decir que mientras más interactivo sea el sistema, hay mayor posibilidad de error. Cada función del software debe ser medida en términos de su beneficio real al usuario vs. su efecto en la confiabilidad.

2.- Factores Humanos

Los factores humanos en un sistema son una medida de la facilidad de entender el sistema, de la facilidad de usarlo y de la frecuencia de errores del usuario. Muchos errores prolongados en un sistema operacional entran cuando nuevas condiciones ocurren súbitamente. Estas nuevas condiciones incrementan la complejidad del sistema y tienen

un efecto negativo en la confiabilidad. Buenos factores humanos minimizan la oportunidad de encontrar inesperadas acciones del usuario disminuyendo así las oportunidades de encontrar errores.

3.- Adaptabilidad

Es la medida de la facilidad de extender el sistema, tal como añadir una nueva función al mismo. Adaptabilidad y confiabilidad son compatibles, porque una de las muchas ventajas de las técnicas de diseño para confiabilidad,son sus efectos positivos en la adaptabilidad. Esto es que las técnicas de diseño no solamente permiten producir software confiable, sino que hacen que sea mas fácil de extender.

4.- Mantenimiento

Es una medida del costo y tiempo requerido para detectar y corregir los errores de software en un sistema operacional.

Mantenimiento y confiabilidad son compatibles porque el mantenimiento está relacionado muy cercanamente con la adaptabilidad, pues si es necesaria alguna modificación el sistema podrá ser extendido o modificado facilmente, sin el peligro de introducir nuevos errores durante este proceso.

5.- Seguridad

Es una medida de la probabilidad de que un usuario del sistema pueda accidentalmente o intencionalmente consultar o destruir los datos que son de su propiedad o de otro usuario, o que interfiera con la operación del sistema. Por tanto los sistemas mas seguros serán mas confiables. Así las medidas de seguridad involucran el aislamiento cuidadoso de los datos del usuario, el aislamiento de los programas entre un usuario y otro y el aislamiento de los programas del usuario con el sistema operativo.

6.- Documentación

Los objetivos de la documentación están en relación con la calidad y cantidad de información que el usuario nos da sobre el sistema. Los objetivos de documentación y factores humanos son similares en que se relacionan con la facilidad de entender y usar el sistema. La documentación es un proceso que no debe comenzar una vez que el programa finaliza, sino que debe comenzar con la definición del problema. No importa cuan bueno es el programa, este trabajará correctamente si las instrucciones adecuadas para su uso son incluídas. Por tanto la buena documentación tiene un efecto positivo en la confiabilidad. Se debe producir documentación de análisis del sistema, de implementación del sistema y de uso del sistema.

7.- Costo del Sistema

El costo de un sistema de software incluye el costo del desarrollo original del proyecto mas el costo de mantenimiento del mismo. Si existen muchos errores el software será inconfiable y su costo total aumentará considerablemente.

8.- Plan o Programa

Otro objetivo clave en cualquier proyecto es obtener el sistema para cierta fecha. Los beneficios de un sistema están cercanamente relacionados a su fecha de disponibilidad.

Los programadores tienen una tendencia a subestimar el tiempo necesario para la prueba. Este tiempo se relaciona con el número de errores de software remanentes en el sistema después de que su diseño es completado y este tiempo puede ser reducido minimizando el número de errores generados durante el proceso de diseño. Los objetivos de maximizar confiabilidad y minimizar el tiempo de desarrollo transcurrido son compatibles solamente si el tiempo transcurrido para lograr los objetivos no es tan extremo como para permitir una insuficiente cantidad de tiempo para un adecuado diseño.

9.- Eficiencia

Las relaciones entre eficiencia y confiabilidad son extremadamente complejas. Por ejemplo, las medidas de confiabilidad pueden añadir instrucciones adicionales que deben ser ejecutadas, reemplazando codificación obscura e intrincada. De esta forma se aumenta el tiempo de ejecución del programa y los requerimientos del mismo. Por otra parte un software inconfiable no puede decirse que es eficiente tan solo por su rapidez de ejecución.

En la primera fase de la computación cuando el Hardware de los computadores era inconfiable y las aplicaciones eran largos procesos en lote, lo mas importante era la eficiencia. Los programadores estaban interesados en exprimir cada posible segundo de proceso y de salida de su programa para así tener la oportunidad de completar la corrida del programa sin que el Hardware fallara.

En 1960 entramos a la segunda fase. Aquí la confiabilidad y velocidad de Hardware se ha incrementado en gran magnitud. Las aplicaciones se procesan en línea. Ineficiencias son tolerables en la mayoría de esas aplicaciones pero errores no. En esta fase cualquier desarrollador de software que sacrifique confiabilidad en favor de eficiencia, podría considerarse que está actuando irresponsablemente. Las ineficiencias de Software pueden ser arregladas mas tarde si es necesario, la inconfiabilidad es mucho mas difícil de rectificar. Los resultados de ineficiencia son usualmente predecibles (larga espera); los resultados de inconfiabilidad son impredecibles y a menudo desastrosos.

DISEÑO EXTERNO

Es el proceso de describir el comportamiento de un sistema y de como éste será percibido por un observador externo.

El propósito del Diseño Externo es la "ingeniería" de las interfases externas, usualmente humanas, del sistema sin considerar su estructura interna. El diseño externo es expresado en especificaciones externas las cuales tienen una audiencia que incluye: el usuario (para revisar y aprobar), los escritores de la documentación del usuario, todos los programadores involucrados con el proyecto y todo el personal involucrado con la prueba del proyecto.

Aunque no existen metodologías para diseño externo hay un principio a seguir, es la idea de la integridad conceptual, que es la armonía (o falta de armonía) entre las interfases externas del sistema. También es la medida de uniformidad de las interfases del usuario. El concepto implica que es mejor tener un conjunto de funciones coherentes que funciones independientes y sin coordinación.

En el diseño externo del sistema el diseñador está involucrado con tres areas que tienen relación con la confiabilidad de software: minimizar los errores del usuario, detectar los errores del usuario cuando estos ocurren y minimizar la complejidad.

Minimizar los errores del usuario, no disminuye el número de errores del software, pero sí incrementa la confiabilidad, disminuyendo la probabilidad de encontrar errores por requerimientos no definidos por el usuario. Presentamos algunas guías para minimizar errores del usuario en sistemas interactivos:

1.- El diseño de la información entrada por el usuario será tan breve como sea posible, pero no tanto que ésta pierda su significado. Esto se hace considerando la frecuencia con la cual el usuario promedio usará el sistema (frecuente u ocasionalmente).

2.- Tener una integridad conceptual entre los tipos de entradas y tambien entre las salidas. Por ejemplo, todos los reportes, pantallas de gráficos y mensajes, podrían tener el mismo formato, estilo y observaciones.

3.- Proveer una facilidad de "Ayuda", es decir un conjunto separado de funciones que provean al usuario con información por si acaso se confunde u olvida interfaces particulares del sistema.

Además de minimizar los errores del usuario, el sistema debe también tratar apropiadamente los errores del usuario cuando estos ocurran. Por otro lado, éstos errores sucederán independientemente de cuan buenas sean las interfases con el usuario.

Guías para Detectar los errores del usuario son:

1.- Diseñar el sistema para que reconozca cualquier dato de entrada. Si la información de entrada no es lo que el sistema ha definido como válida se informará al usuario. Este proceso se conoce con el nombre de validación de datos.

2.- Si el usuario entra una transacción compleja a través de muchos mensajes o secuencias de claves, hay que darle una oportunidad para verificar antes de que la acción se lleve a efecto.

3.- Donde la exactitud es vital, no importa la redundancia en los datos de entrada. Por ejemplo, en un sistema de Banco uno podría requerir el nombre del cliente junto con el número de la cuenta para formar la clave y así poder detectar algún error en la entrada del número de cuenta. Otra posibilidad es hacer el chequeo del número de cuenta, haciendo que el último dígito sea alguna combinación aritmética de los otros.

La confiabilidad se incrementa al Minimizar la complejidad del diseño externo para así minimizar la complejidad interna del sistema tal como se minimizan los errores del usuario.

Un diseño de salida que siempre aparece en el diseño externo de un sistema interactivo son las opciones para que el usuario entre la parte de datos que está incorrecta. Esto de aquí puede producir confusiones, pues en algunos casos no se puede determinar que parte es necesario cambiar para que esté correcta y esto en la mayoría de los casos puede llegar a confundir al usuario. Este tipo de opciones serán necesarias solamente si los datos de entrada son demasiado extensos, y si esto sucede es que estos datos han sido pobremente diseñados.

Otras áreas de complejidad surgen cuando los sistemas hacen asumpciones acerca de los datos de entrada y éstas asumpciones pueden producir errores de correcto deletreo u ortografía. Al diseñar las interfases del usuario para un sistema confiable, es necesario que éstas sean uniformes y simples de tal forma que el sistema espere cualquier dato como entrada e inmediatamente pueda detectar todos los errores para así poder corregirlos.

ESCRITURA DE LAS ESPECIFICACIONES EXTERNAS

Unas especificaciones externas bien hechas, permitirán comprenderlas para así poder utilizar una organización jerárquica de las tareas a seguir. El diseño externo detallado para cada función del sistema podría especificar los siguientes tipos de información:

1.- Descripción de la entrada.

2.- Descripción de la salida.

3.- Transformaciones del sistema.

4.- Características de confiabilidad.

5.- Eficiencia.

6.- Notas de programación.

Para cualquier función compleja del usuario, la tarea referente a salidas y transformaciones del sistema en la forma de causa y efecto, no es un trabajo fácil, y para esto pueden usarse las tablas de decisión. Las tablas de decisión son una técnica muy útil para confiabilidad de software, hacen una excelente especificación externa de los comandos y proveen una forma gráfica y rigurosa de mostrar las relaciones entre las entradas y las salidas.

En conclusión, las especificaciones externas describen cada posible entrada del sistema (válidas o inválidas) y describen la relación del sistema con cada una de ellas.

ARQUITECTURA DEL SISTEMA

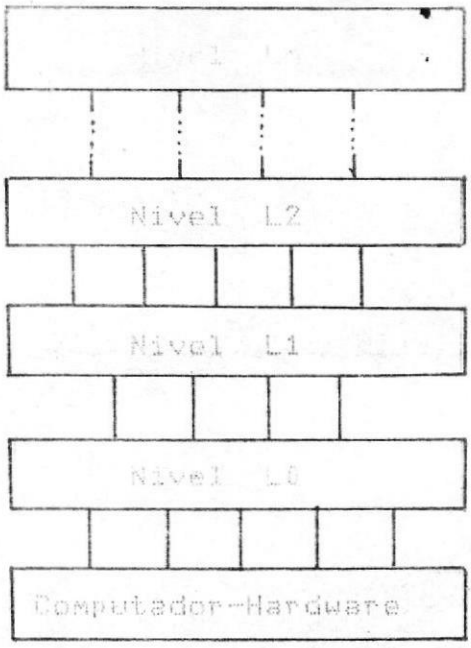
Arquitectura del sistema es el proceso de descomponer un software extenso en pequeñas piezas. Estas piezas tienen una variedad de nombres tales como programas, componentes, subsistemas o niveles de abstracción. Es un proceso necesario en el diseño de sistemas pero no en el diseño de programas. Si las especificaciones externas describen un sistema, el próximo paso es la arquitectura del sistema seguida por los otros pasos de diseño. Si las especificaciones de diseño describen un programa, la arquitectura del sistema es innecesaria, pues no tendrá subsistemas que lo compongan. Un sistema representa un conjunto de soluciones a un conjunto de problemas distintos pero relacionados y, está compuesto por varios programas.

NIVELES DE ABSTRACCIÓN

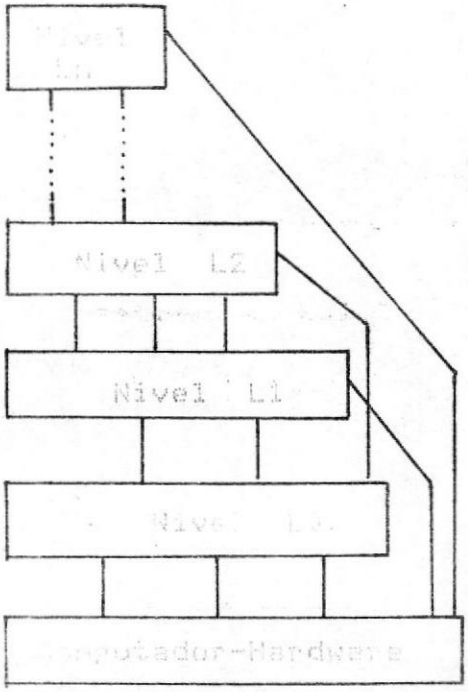
Esta idea fue originada por Dijkstra [1,2]. Un sistema es partido en distintas piezas jerárquicas llamadas niveles, hechos bajo ciertos criterios de diseño. Cada nivel es un grupo de modulos relacionados muy cercanamente. El propósito de los niveles es minimizar la complejidad del sistema. Cada nivel representa una abstracción de los recursos del sistema y de la representación de los datos.

Hay dos estructuras generales. En la primera figura el problema es visto como un desarrollo del "usuario-máquina", comenzando con el nivel mas bajo para la máquina, hardware o sistema operativo, luego siguiendo con los niveles superiores para lenguajes de alto nivel extendiendo sus capacidades. Cada nivel debe solamente referenciar (llamar) al nivel inmediatamente subordinado a él.

En la estructura de la segunda figura, los niveles superiores no son completas abstracciones de los niveles inferiores, pero permiten que cualquier nivel superior pueda referenciar a cualquiera de los inferiores. Por ejemplo, la segunda figura corresponde a los niveles de abstracción para diseñar un sistema operativo:



NIVELES DE ABSTRACCIÓN



El nivel 1 es el responsable del manejo de la memoria. El nivel 2 manipula todas las conecciones entre niveles altos y la consola del operador. El nivel 3 es responsable del área de almacenamiento temporal de entrada y salida (buffer). El nivel 4 corresponde a los programas o aplicaciones del usuario, etc.

Los niveles de abstracción minimizan el número de estados internos en el sistema, permitiendo la corrección de cada nivel y la verificación después de que ha sido diseñado, permitiendo una prueba del producto final.

Un efecto beneficioso de los niveles de abstracción es la portabilidad, es decir la facilidad de modificar una aplicación para operar en un diferente sistema de computación.

Por medio de la abstracción las modificaciones no solamente se limitan a partes aisladas del sistema, sino que también muestra precisamente cuales partes deben ser alteradas.

DOCUMENTACION

La fase final de la arquitectura del sistema es diseñar la documentación describiendo la descomposición del sistema de software en sus partes mayores como son: subsistemas, programas, procesos o niveles de abstracción. La documentación podría describir la función o funciones de cada componente, las interfases entre los componentes y la estructura del sistema. La estructura del sistema podría ser expresada de la siguiente forma:

1.- Jerarquía del flujo de control del sistema.

2.- Estructura del flujo de datos del sistema.

3.- Tareas jerárquicas y las relaciones entre ellas.

4.- La estructura de la memoria del sistema. Esto solamente se aplica si todos los componentes no residen en una simple localización de memoria.

5.- Una relación de la estructura de Software, con la configuración de Hardware.

DISEÑO DE LA ESTRUCTURA DEL PROGRAMA

El siguiente proceso en el diseño del software es el diseño de la estructura del programa, el cual incluye la definición de todos los módulos en el programa, la estructura jerárquica de los módulos y las relaciones entre ellos. Un módulo es una subrutina cerrada que puede ser llamada desde cualquier otro módulo en el programa, y puede ser compilado separadamente, además deben ser independientes entre sí.

Si lo que se está diseñando es un simple programa, este proceso incluye las especificaciones externas detalladas. Si es un sistema, el proceso incluye las especificaciones externas detalladas y la arquitectura del sistema, siendo este último el diseño estructural de todos los subsistemas o componentes en el sistema.

El método tradicional de manejo de complejidad es la idea de la modularización. Sin embargo en la práctica esta idea ha sido frecuentemente inefectiva. Existen tres razones para esto:

1.- Los módulos son diseñados para hacer muchas funciones relacionadas pero diferentes.

2.- Las funciones comunes no son identificadas en el diseño y como resultado se encuentran distribuídas entre muchos módulos diferentes.

3.- Los módulos interactúan sobre datos comunes en forma inesperada.

Una forma para hacer un programa menos complejo, es descomponiendolo en un conjunto de módulos pequeños y altamente independientes. Esta independencia se puede lograr maximizando las relaciones dentro de cada módulo y minimizando las relaciones entre módulos.

Dado que un programa eventualmente consistirá en un conjunto de sentencias, donde las sentencias tendrán algunas relaciones entre ellas, en términos de funciones realizadas

manipulación de datos, será necesario organizar estas en módulos separados, de tal forma que las sentencias dentro de un modulo tengan una relación cercana y que cualquier par de sentencias en dos módulos diferentes tengan una mínima relación. La meta es que cada módulo realize una función simple y única.

Vamos a indicar las ventajas y desventajas de la modularidad, para poder determinar sus efectos en la confiabilidad de software.

Argumentos a favor de la modularidad.-

Las opiniones a favor han sido dadas por programadores que han visto los resultados en sus propios programas. Un programa modular es:

1.- Fácil de escribir y depurar.

2.- Fácil de mantener y cambiar. Sus componentes funcionales pueden ser cambiados, reescritos, o reemplazados sin afectar otras partes del programa.

3.- Fácil de usar y controlar.

Argumentos contra la modularidad.-

Una vez mencionadas las ventajas de la modularidad, se podría preguntar: Por qué la mayoría de los programas no son escritos en forma modular?. Existen algunas razones:

1.- La mayoría de los programadores no entienden la modularidad.

2.- La modularidad requiere una gran cantidad de esfuerzo extra. Para escribir un programa modular, es necesario ser más meticulosos en la fase de diseño, así como también se requiere mas paciencia por parte del programador, porque deberá ir cambiando, modificando y documentando su módulo en la etapa de diseño antes de comenzar la codificación.

3.- Los programas modulares ocasionalmente requieren mas tiempo de CPU y mayor espacio de memoria. Este tiempo y espacio extra requerido no son mas que un 5-10% de lo normal. Tan solo en el caso de los minicomputadores con capacidad limitada, este podría ser un gran problema, en caso contrario, es un precio razonable a pagar por un programa que puede ser mantenido y cambiado facilmente.

Como podemos apreciar las ventajas de la modularidad, son muy significantes en función de la confiabilidad, y aunque parezca extraño las desventajas también lo son, pues si necesitamos tener mayor cuidado al diseñar un programa modular, podemos darnos cuenta de los errores que puedan producirse a tiempo, y de esta forma evitar que sucedan cuando el programa está en producción, pues esto implicaría mayor costo y mayor esfuerzo posterior en reparar los errores producidos.

DISEÑO EXTERNO DEL MODULO

El primer paso en el diseño de un módulo es definir sus características externas y toda la información necesaria. Las especificaciones externas de un modulo no deben contener información acerca de la representación lógica o interna de los datos del módulo; nunca deberán contener referencias de las llamadas a módulos donde este módulo es usado.

Las especificaciones externas de los modulos deberán contener los siguientes seis tipos de información:

NOMBRE DEL MODULO.- Este define el nombre usado para llamar al módulo. Para un módulo con múltiples puntos de entrada, es decir aquel que es llamado desde diferentes partes del programa, este nombre es el nombre de entrada al módulo.

FUNCION.- Esta es una definición de la función o funciones realizadas por el módulo.

LISTA DE PARAMETROS.- Define el número y orden de parámetros pasados al módulo.

ENTRADA.- Es una descripción precisa de todos los parámetros de entrada al módulo. Incluye una definición del formato, tamaño, atributos, unidades y dominios válidos de todos los datos de entrada.

SALIDA.- Esta es una descripción precisa de todos los datos de salida enviados por el módulo. Incluye una definición del formato, tamaño, atributos, unidades y rangos válidos de todas las salidas. Esta descripción podría relacionar las salidas con las entradas en causa y efecto, es decir cuales salidas son generadas a partir de cuales entradas.

EFECTOS EXTERNOS.- Esta es una descripción de cualquier acción tomada en forma externa al programa o sistema cuando este módulo es llamado. Ejemplos de efectos externos son: impresión de reporte, leer un comando, leer un archivo de transacción o escribir un mensaje de error.

Los efectos externos de un módulo incluyen cualquier efecto externo de los módulos subordinados a él. Por ejempo, si el módulo A llama al módulo B y B imprime un reporte, entonces este efecto externo debe aparecer en las especificaciones externas de ambos módulos A y B. Es importante separar las especificaciones externas de un módulo de otra documentación, tales como descripción de la lógica del módulo, porque el módulo lógico puede ser cambiado sin afectar las llamadas a los módulos, pero las especificaciones externas de los módulos normalmente conducen a cambios en las llamadas a los módulos.

El proceso final en esta larga secuencia de diseño de software, es el diseño y codificación de la lógica interna de cada modulo. El proceso de diseño de un módulo puede ser cuidadosamente planeado.

Los siguientes 11 pasos muestran la disciplina del diseño de un módulo.

SELECCION DEL LENGUAJE.- La selección del lenguaje usualmente es dictada por los requerimientos del proyecto. La selección del lenguaje debe ser hecha desde que el proyecto es planeado.

ESPECIFICACIONES DEL DISEÑO DEL MODULO EXTERNO.- Este es el proceso de definir las características externas de cada módulo.

VERIFICACION DE LAS ESPECIFICACIONES EXTERNAS DEL MODULO.- Las especificaciones para cada módulo, deberían ser verificadas, comparándolas con la información relacionada del diseño de la estructura del programa y asegurándose de que ésta es revisada por los programadores de todos los módulos que llaman a este módulo.

SELECCION DE ALGORITMOS Y ESTRUCTURAS DE DATOS.- Un paso vital en el proceso del diseño lógico, es la selección de un algoritmo y la correspondiente estructura de datos. Algunos algoritmos son inventados en el momento en que se diseña un módulo, otros algoritmos ya han sido inventados, y en otros casos se alteran uno o mas algoritmos que ya han sido inventados de acuerdo a las necesidades del diseñador. En vez de gastar nuestro tiempo reinventando algoritmos y estructuras de datos, es mejor buscar una solución conocida.

ESCRIBIR LA PRIMERA Y ULTIMA SENTENCIA.- El siguiente paso es escribir las sentencias Procedure y End para el módulo o sus equivalentes, dependiendo del lenguaje de programación.

DECLARAR TODOS LOS DATOS DE INTERFASE.- El siguiente paso es escribir las sentencias del programa que definen o declaran todas las variables en el módulo.

DECLARAR DATOS PERMANENTES.- Escribir las sentencias que definen o declaran todas las otras variables que van a ser usadas en el programa.

REFINAMIENTO DEL CODIGO.- Este paso involucra refinamientos sucesivos de la lógica del módulo, comenzando con una definición abstracta de la lógica y terminando con el código del módulo.

PULIR EL CODIGO.- El código del módulo es aquí pulido para dar claridad. Comentarios adicionales son añadidos para anticipar cualquier pregunta que un lector del código pueda tener.

VERIFICAR EL CODIGO.- El módulo es manualmente verificado para realizar algunas correcciones a errores que se presenten.

COMPILAR EL MODULO.- El último paso es la compilación del módulo. Este paso marca la transición entre el diseño y la prueba. El paso de compilación es actualmente el comienzo de la prueba del software.

EXPERIENCIA

La experiencia que un diseñador debe tener es de algunos tipos:

Experiencia diseñando, con situaciones de diseño semejantes y con las tecnologías que deben relacionar los procesos de diseño.

Como se puede notar, diseñar es un proceso complejo que no puede ser hecho en la forma de un conjunto de instrucciones explícitas. Se debe aprender cómo diseñar para hacerlo.

Muchas de las operaciones de diseño requiere una gran cantidad de intuición y toma de decisiones que no pueden ser expresadas por completo como decisiones racionales. Por ejemplo, saber cuántas alternativas considerar, cuál considerar primero, cuál está fuera de nuestro alcance, son decisiones que son bien hechas solamente si uno las ha hecho antes y ha aprendido con la experiencia. En vista de que tales decisiones aparecen repetidamente y en muchas formas diferentes, uno debe estar preparado para hacerlas lo mas rápido posible.

Además, el conocer diseños similares nos provee de un prototipo sobre el cuál basar los otros diseños. Este tipo de experiencia es más que un simple conocimiento, ya que un prototipo sirve más como guía para tomar decisiones que como una plantilla.

Otro tipo de experiencia que es esencial es con las Tecnologías de Diseño. Si un diseñador no tiene conocimiento personal de programación, le será difícil especificar las estructuras que pueden ser fácilmente programadas.

El conocimiento de los límites y posibilidades del Hardware son esenciales para el diseñador de Software. La experiencia en creación de nuevos sistemas de organización, puede ser una gran ayuda. La experiencia con manejo de procesos de diseño y de implementación proveen al diseñador de la visión necesaria, de algunos de los factores que pueden influenciar en los sucesos eventuales del diseño.

CONCLUSIONES

La confiabilidad de un sistema es esencialmente determinada durante la etapa de diseño. Si el diseño del sistema es pobre, esto es, sin una estructura razonable, entonces en la mayoría de los casos no es posible obtener un sistema altamente confiable y serán necesarias correcciones posteriores. Si por otra parte, se pone mayor esfuerzo en la eliminación de errores durante la fase del diseño, entonces se ahorra un tiempo considerable durante la etapa de prueba.

El diseño externo requiere especialistas que estén familiarizados con todas las fases del diseño de software y que entiendan los efectos de estas fases, como por ejemplo analistas de sistemas, graduados en Ciencias de la Computación, Ingenieros Industriales, entre otros. Los diseñadores deben acceder a las necesidades del usuario, evaluar costos y determinar esquemas realistas. Utilizando el personal adecuado y siguiendo los pasos aquí mencionados, existe una alta probabilidad de tener programas confiables y fáciles de mantener.

REFERENCIAS

1.- Dijkstra, E.W., "The Structure of THE Multiprogramming System", Comunications of the ACM, 1968.

2.- Dijkstra, E.W., Complexity Controlled by Hierarchical Ordering of Functions and Variability, P. Naur y B. Randell, Editores, Software Engeneering, Report on a Conference Sponsored by NATO Science Commettee, Bruselas, Belgica, NATO Scientific Affairs Division, 1968.

3.- Williams, R. D., "Managing the Developmente of Reliable Software", Proceedings of the 1975 International Conference on Reliable Software, New York, IEEE, 1975.

Metodologías de Diseño

PROGRAMACION ESTRUCTURADA

La programación estructurada es una filosofía de escribir programas de acuerdo a un conjunto de reglas rígidas, para disminuir los problemas de la prueba de programas, incrementar la productividad e incrementar la legibilidad del programa resultante [9].

El profesor Edsger W. Dijkstra de la Universidad de Eindhoven, Holanda, ha sido uno de los pioneros de la programación estructurada. En 1965 sugirió en el congreso de IFIP, realizado en New York, que el uso del GOTO debería ser eliminado de los lenguajes de programación y una de sus observaciones fue que "La calidad de un programador es inversamente proporcional al número de sentencias GOTO en sus programas". En 1968 Dijkstra envió una carta al editor de Comunicaciones de la ACM, titulada "GOTO Statement Considered Harmful" [2] ("Las Sentencias GOTO Consideradas Perjudiciales), en ella expresaba su convicción de que el GOTO debería ser abolido de los lenguajes de alto nivel, pues es demasiado primitivo y tiende a complicar los programas. También alentaba a buscar construcciones alternativas las cuales serían necesarias para satisfacer todas las necesidades de bifurcación en los programas. Dijo que había una cercana correspondencia entre el texto del programa y su flujo de ejecución, es decir, un programa debería leerse de arriba hacia abajo y ejecutarse en la misma forma y que el uso descuidado de las sentencias GOTO, interfieren con esta correspondencia.

Existe controversia sobre el uso del GOTO en los programas. Donald Knuth en su artículo "Structured Programming with GOTO Statement" (Programación Estructurada con Sentencias GOTO) explica la influencia que tiene el GOTO en la programación estructurada [4]. Knuth realizó estudios acerca de las ventajas y desventajas del uso del GOTO, utilizando como ejemplos programas que realizaban la misma función, unos con sentencias GOTO y otros sin ellas. Llegó a la conclusión de que existen casos en que es preferible el uso del GOTO cuando este elimina cálculos y lazos innecesarios, pero es aconsejable su abolición, cuando su uso excesivo resta legibilidad y complica el programa. Knuth concluyó que un programa puede ser bien estructurado y confiable conteniendo sentencias GOTO.

Un número de estudiantes y organizaciones de investigación de la Universidad de Carnegie-Mellon desarrollaron un lenguaje de implementación de sistemas llamado BLISS, que no tiene sentencias GOTO. De acuerdo al profesor William Wulf, la experiencia de tres anos con BLISS y su uso en el desarrollo de compiladores y sistemas operativos, demostró ser un práctico y útil lenguaje de programación. Los programadores familiarizados con lenguajes con la sentencia GOTO, que participaron en proyectos con BLISS y que habían estado presentes, tuvieron que atravezar por un doloroso periodo de adaptación. Una vez que pasaron este periodo, encontraron que la pérdida del GOTO no fue una desventaja, al contrario, la reacción invariante fue que la disciplina de la programación sin GOTO, estructura los programas y simplifica las tareas [9].

La abolición de los GOTO no debe tomarse como un dogma o como la base para producir programas estructurados. Eliminarlos depende en gran parte del tipo de lenguaje así como también de la aplicación, pues muchas veces es necesario para salir del ámbito de un DO, para evitar excesivos IF THEN ELSE anidados, para salir al final de un módulo, pero la condición para esto es que la bifurcación sea hacia un punto subsecuente en el programa, para preservar la legibilidad de arriba hacia abajo.

Otras guias de la programacion estructurada sugieren que la repetición del código en uno o mas lugares del módulo puede ser usada como una forma de eliminar las sentencias GOTO. En este caso la cura es peor que la enfermedad, pues la duplicación de la lógica incrementa la oportunidad de error cuando se deseen hacer cambios en el futuro, por tanto la confiabilidad de los programas disminuye grandemente [2].

La programación estructurada se debería caracterizar, no por la ausencia de los GOTO, pero sí por la presencia de estructuras [5].

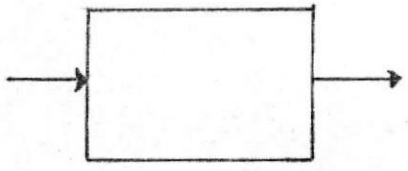
Giuseppe Jacopini en 1965 escribió un artículo, junto con Corrado Bohm, en el cual mostraba que cualquier programa dado en forma de diagrama de flujo, puede ser

transforma [illegible] produce el [illegible] base a sentencias del programa fuente, usando solamente tres estructuras básicas [illegible] condicionales y de iteración, además de [illegible] y variables auxiliares [11].

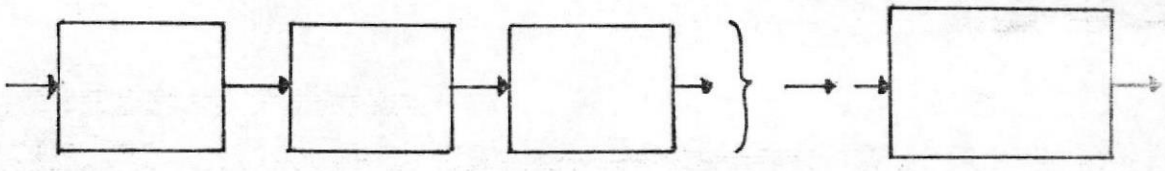
De acuerdo a Böhm y Jacopini, necesitamos tres construcciones básicas para estructurar un programa:

1.- Un bloque de proceso
2.- Un mecanismo de lazo generalizado.
3.- Un mecanismo de decisión binaria.

El bloque de proceso puede ser una simple sentencia, por ejemplo, una sentencia MOVE en COBOL, una sentencia computacional propia, es decir aquella que tiene un solo punto de entrada y un solo punto de salida, por ejemplo, una subrutina. Ademas se puede transformar una secuencia lineal de proceso en bloque, en un simple proceso en bloque.

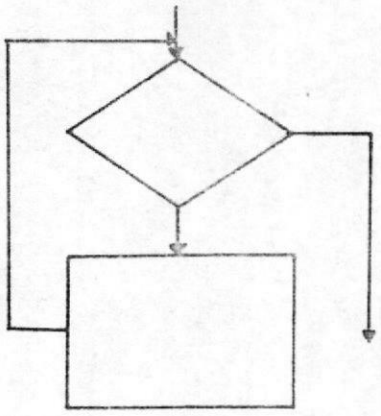


BLOQUE DE PROCESO

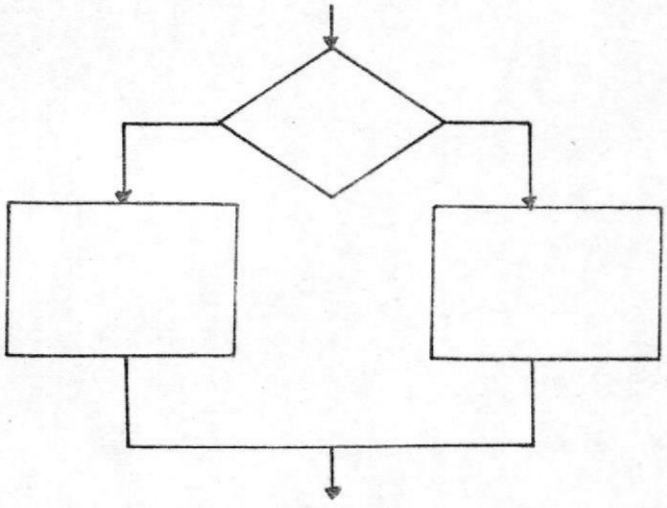


SECUENCIA LINEAL DE BLOQUES DE PROCESO

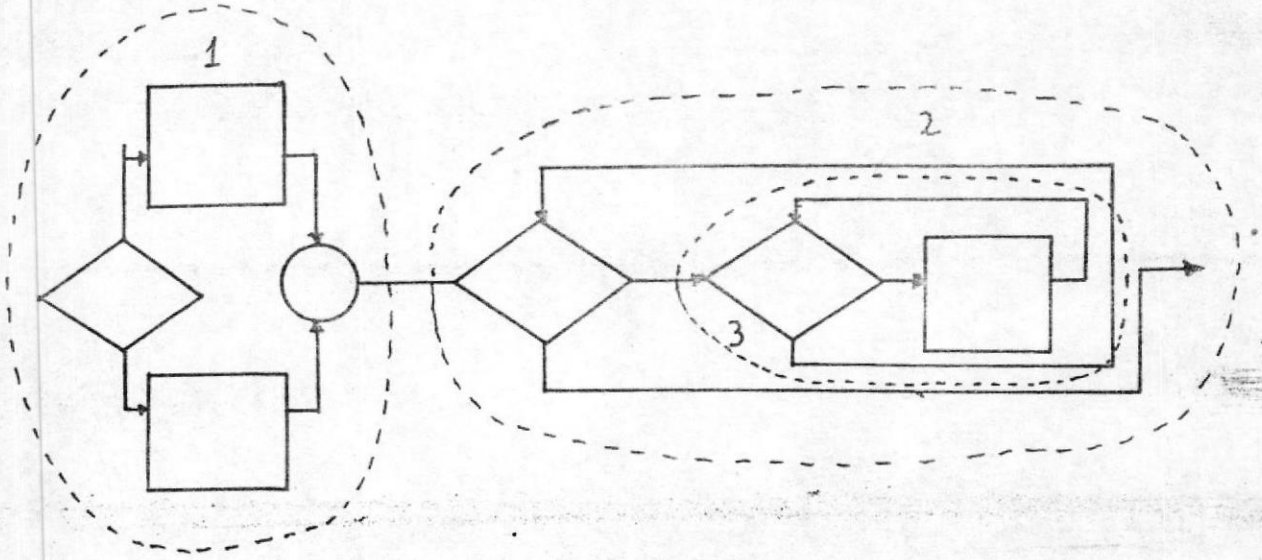
El mecanismo de lazo es la estructura de control DO WHILE

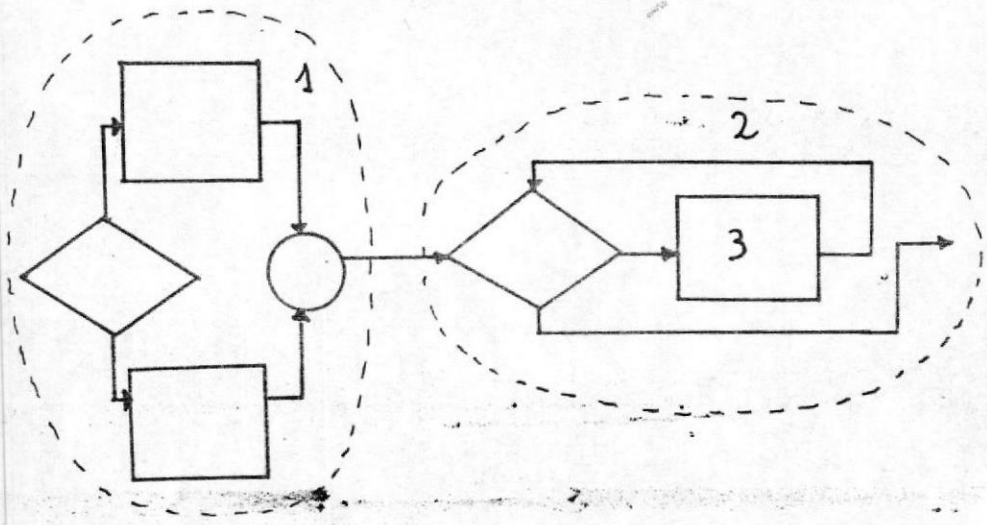


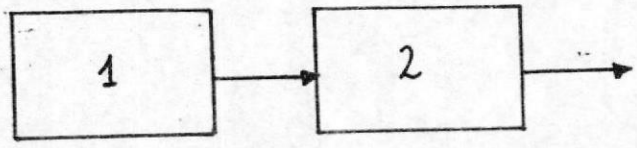
El mecanismo de decisión binaria es la estructura de control IF THEN ELSE.



Las construcciones WHILE DO e IF THEN ELSE, pueden ser concebidas como un proceso en bloque, debido a que solo tienen un punto de entrada y un punto de salida. Así podemos efectuar una transformación desde una operación de lazo o un mecanismo de decisión a un proceso en bloque. Estos bloques son recursivamente definidos, por ejemplo, un bloque secuencial podría ser un IF THEN ELSE y el otro podría ser un DO WHILE y el DO WHILE a su vez podría contener otro DO WHILE. A continuación se muestra esta situación graficamente.


1
2
3


1
2
3

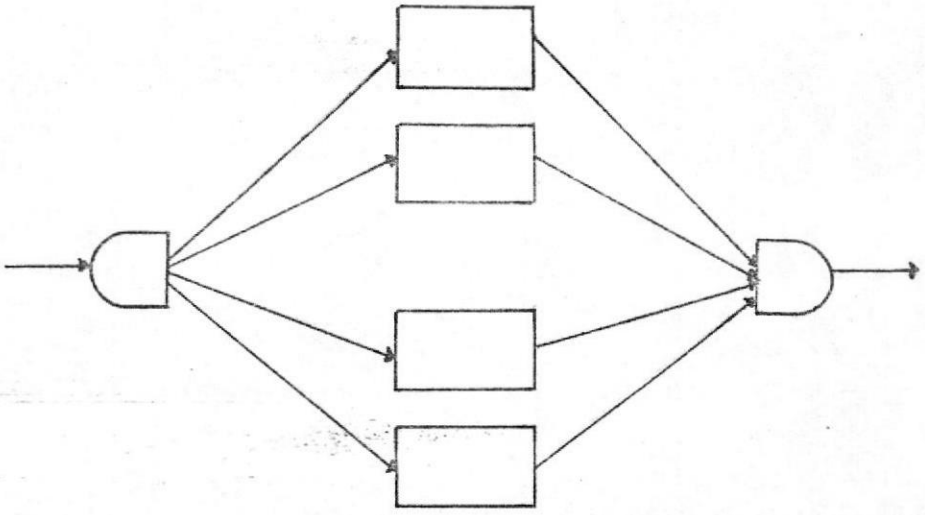

1
2

A pesar de que estos tres mecanismos son suficientes para producir un programa, existen otras construcciones que han sido añadidas posteriormente.

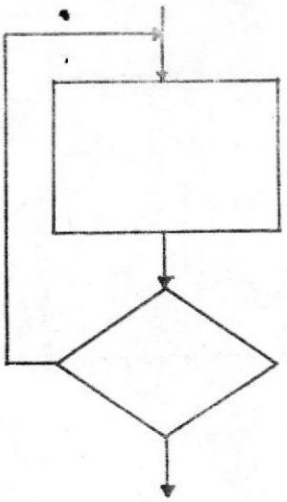
La sentencia CASE que es de la siguiente forma:

```
CASE  VALOR  OF
    BEGIN
    sentencia 1;
    sentencia 2;
    .
    .
    sentencia j;
    END;
```

Las sentencias dentro de la construcción CASE son procesos en bloque y son mutuamente excluyentes, es decir, si el contenido de VALOR es i, solo la sentencia i será ejecutada.



Existe otra construcción de lazo adicional, es la construcción REPEAT UNTIL, la cual ejecutará primero el proceso de bloque y entonces probará la condición para determinar si el lazo debe ser repetido nuevamente. Se diferencia del DO WHILE en que el proceso de bloque se efectúa por lo menos una vez y que la condición es probada al final.



Los puntos mostrados a continuación ilustran las reglas a seguir para obtener un programa estructurado.

1.- El código debe ser construído de la secuencia de los elementos básicos: sentencias secuenciales, sentencias de iteración como lazos WHILE DO y REPEAT UNTIL y sentencias condicionales IF THEN ELSE.

2.- El uso de las sentencias GOTO debe ser evitado mientras sea posible, en particular el tipo mas peligroso de GOTO, que es aquel que transfiere el control a sentencias previas del programa.

3.- El código debe ser escrito en un aceptable estilo (ver Estilo de Programación).

4.- El código debe ser indentado en el listado para poder seguir mejor la secuencia del programa (Ej: Las clausulas THEN y ELSE deben estar en la misma dirección) y para proveer mayor legibilidad.

5.- Debe haber un solo punto de entrada y un punto de salida en cada módulo.

6.- Las sentencias del código de un módulo deberían estar en solo en una página de listado.

La programación estructurada tiene una gran influencia positiva en la confiabilidad de los programas, pues hace que los programas sean mas entendibles y faciles de escribir, debido a las ventajas que ofrecen las estructuras de control; documentados por si mismos, compatibles y portables con máquinas que manejen lenguajes estructurados, es decir aquellos que establecen los mecanismos de estructuras básicas de control.

ESTRUCTURAS DE DATOS

Para completar un buen diseño de un programa estructurado, es necesario seleccionar las estructuras de datos apropiadas para el problema que vamos a resolver.

Las estructuras de datos son la forma de organizar los datos sobre los que operan los programas. Un programa de computador es la abstraccion de un problema del "mundo real" el cual necesita ser resuelto. Como parte de esta solucion, los datos del "mundo real", deben también ser abstraídos para que el programa pueda calcular algún valor [8]. El conjunto primitivo de datos incluyen enteros, reales, caracteres y booleanos. Estos tipos de datos estan disponibles para la mayoría de los lenguajes, con algunas excepciones, por ejemplo, el Fortran maneja como datos primitivos los enteros y los reales; el Snobol, los enteros, reales y caracteres. El Pascal, enteros, reales, caracteres y booleanos, etc.

Además de estos datos primitivos se necesitan tipos de datos adicionales, que son los llamados tipos agregados, los cuales agrupan conjuntos de datos con características determinadas. En términos de un lenguaje de programación esto se refiere al tipo de datos que una variable puede asumir.

Arreglos

Es una de las estructuras de datos mas simples. Es un conjunto de datos del mismo tipo, que tienen asociado los pares índice, valor; es decir para cada índice diferente hay un valor asociado.

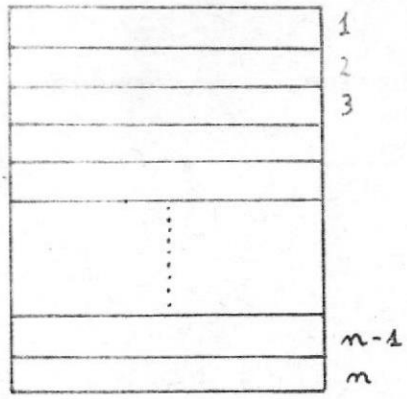


Gráfico de un arreglo de 1 dimension
de n elementos

Es una estructura de datos útil para almacenar información en forma tabular. Las operaciones asociadas con el arreglo son: el almacenamiento y la recuperación. La operación de almacenamiento ingresa un valor en una localización particular definida por el subíndice. La operación de recuperación retorna el valor almacenado en una particular localización del arreglo.

En cualquiera de las dos operaciones posibles en el arreglo, se tendrá un error si el subíndice sobrepasa las localizaciones de memoria asignadas al arreglo.

Un arreglo simple, de una dimensión, puede ser generalizado a un arreglo de n dimensiones. Cada dimensión tiene sus propios subíndices. Un arreglo de dos dimensiones se puede visualizar de la siguiente forma.

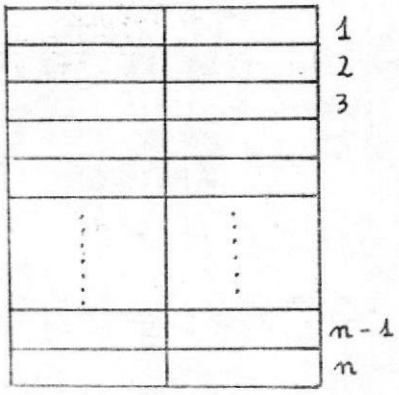
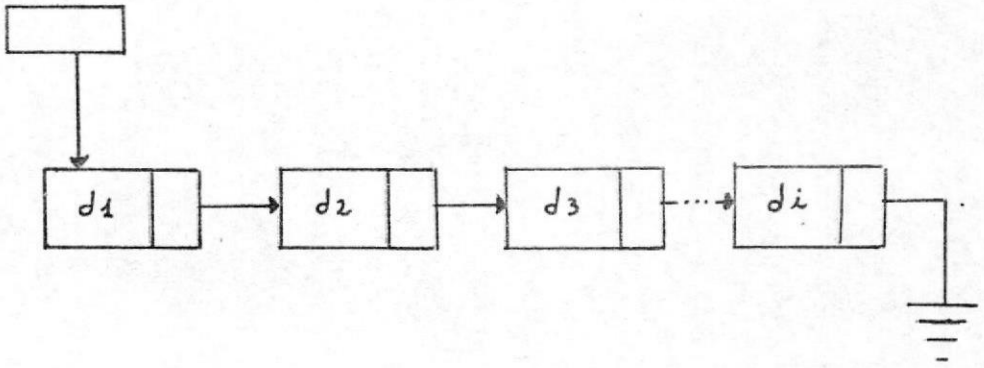


Gráfico de un arreglo de 2 dimensiones
con n elementos.

Listas Encadenadas

Es una lista ordenada de nodos los cuales son una colección de elementos de datos. Sus elementos se enlazan a través de apuntadores. El primer nodo es accesado a través de un puntero especial (llamado "cabeza"). El último nodo contiene un eslabón nulo (null - link-nil), que indica el final de la lista. La lista puede tener un número "infinito" de elementos, dependiendo de la capacidad de la memoria. Las inserciones y extracciones pueden ser hechas en cualquier punto de la lista. Un diagrama general de una lista encadenada es el siguiente:



La parte di de cada nodo corresponde a los campos de datos.

Colas

Una cola es una lista cuyos elementos son añadidos al final de la misma y accesados por el tope o inicio. Aquí el primer elemento que entra, es el primero en salir. Así una cola es como un "pila" donde la información es entrada y entonces procesada en el mismo orden que fue recibida. Se la conoce comúnmente con el nombre de lista FIFO (First-in, First-out) (Primero en entrar, Primero en salir).

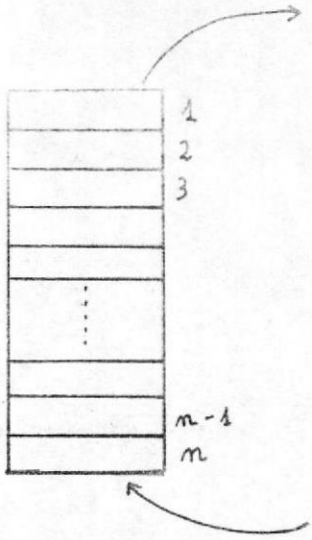


Gráfico de una cola de n elementos

Stacks

Es una lista cuyos elementos son añadidos y accesados en el orden inverso, es decir el primero que entra es el último en salir. Se la conoce también como lista LIFO (Last-in, First-out) (Ultimo en entrar, Primero en salir). El acceso y la inserción son permitidos solo por el tope de la lista.

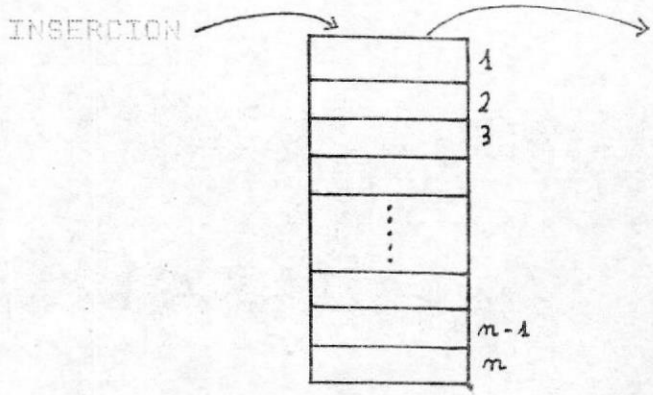


Gráfico de un Stack de n elementos

Así como se determina de que forma se va a estructurar el programa, se debe también determinar cuál será la forma mas conveniente en que los datos van a estar estructurados. Para producir un programa estructurado, además de aplicar las estructuras de control explicadas, son necesarias también ciertas reglas de disciplina

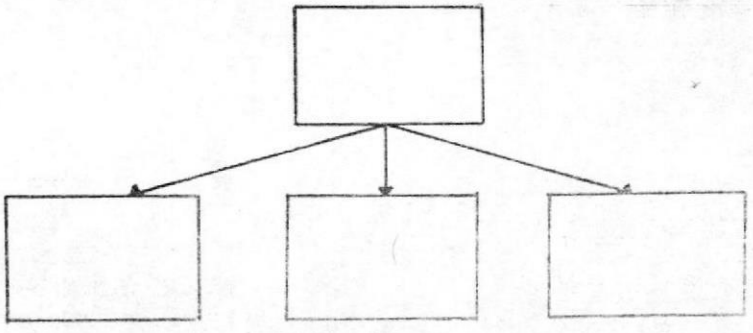
Es una metodología para conseguir un programa estructurado. Es conocida también como "Programación Sistemática" o "Diseño Jerárquico de Programas", el objetivo es identificar las funciones mayores que se llevarán a cabo y entonces proceder de allí a la identificación de funciones menores que se derivan de ellas.

Estas funciones se encuentran contenidas en módulos, siendo estos un conjunto de instrucciones que realizan alguna función lógica. Se debe tratar de expresar la implementación de un módulo, en una página de codificación.

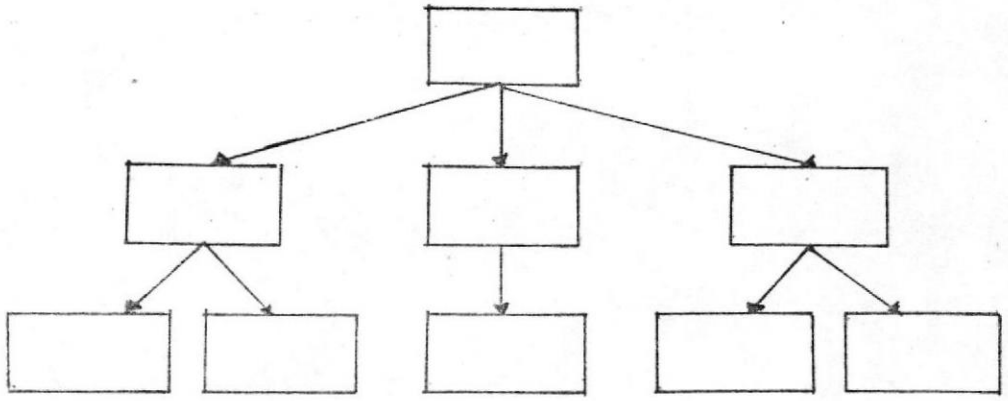
Para que la aplicación del diseño de arriba hacia abajo sea exitosa debe existir un riguroso esfuerzo por parte del programador para especificar la entrada, función y salida de cada módulo en el programa. Una vez que esté convencido de que una porción particular del problema puede ser contenida en un módulo, debe olvidarlo y no preocuparse de cómo implementarlo.

Es necesario prestar mucha atención al diseño de los datos, así como también al diseño de los procesos o algoritmos. En muchos casos los datos son interfases entre los módulos, y el diseño de los módulos no puede llegar muy lejos hasta que esas interfases hayan sido cuidadosamente especificadas.

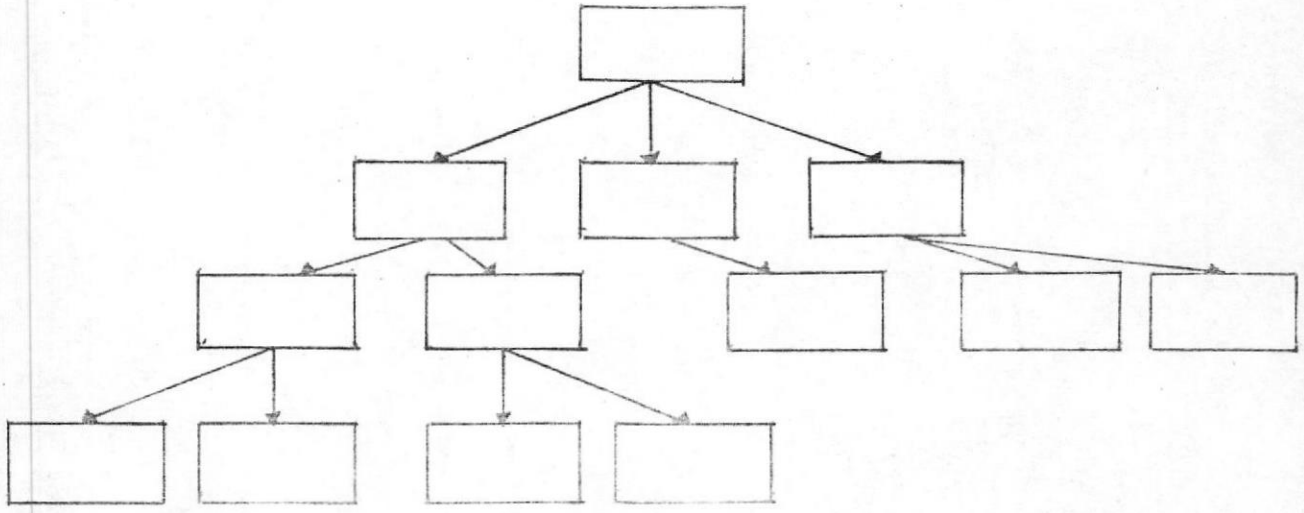
El concepto básico del diseño de arriba hacia abajo es simple, como mostramos en la figura a continuación. Se comienza con un programa principal y uno o dos niveles de módulos.



Con esta "esqueleto" vamos las interfases de los módulos a nivel de lógica. A medida que vamos desarrollando los módulos los vamos probando separadamente para corregir los errores a tiempo y evitar de esta manera, hacer al final del programa una corrección total lo cual es mas difícil o implica una pérdida de tiempo. Es decir, vamos haciendo un programa confiable desde sus inicios, porque la mayoria de los errores ya han sido detectados y corregidos antes de llegar al final del programa.



Cuando añadimos el último nivel de módulos al sistema, hemos finalizado ya el diseño e integrado los modulos entre sí.



Varios proyectos de ingeniería han demostrado los siguientes beneficios [9]. Basándonos en [illegible] incidencia positiva del diseno de arriba hacia abajo en la confiabilidad de software.

1.- La prueba de sistemas en el sentido clásico, es decir, en la forma usada actualmente, de probar el programa tan solo una vez que ha sido concluído, es virtualmente eliminada.

2.- Los diagramas de flujo, diagramas de bloque y otras técnicas, son un punto pobre de concentrar el diseno. En muchos casos, usar el código, es mas preciso y conveniente.

3.- Es mucho mas fácil evitar los problemas posteriores en los programas, si se comienza con un buen diseno.

PROCEDIMIENTO PASO A PASO

Es una metodología usada para diseñar la lógica del módulo, para obtener un programa estructurado. Se lo conoce también como "Proceso de Pensamiento" que fue originalmente concebido por Dijkstra [3] y posteriormente mejorado por Wirth [6,7].

El refinamiento paso a paso es un proceso simple. Involucra una expresión inicial de la lógica del módulo, como sentencias en un lenguaje hipotético de alto nivel, entonces un subsecuente refinamiento de varias sentencias en lenguaje de mas bajo nivel, hasta que finalmente uno alcanza el nivel de lenguaje de programación. A traves de este proceso la lógica es siempre expresada en las construcciones básicas de programación estructurada.

A medida que las tareas necesarias para resolver un problema se refinan, tambien los datos deben ser refinados, descompuestos o estructurados. Es, por lo tanto natural el refinar un programa y las especificaiones de los datos en paralelo. Cada paso del proceso de refinamiento implica alguna decisión de diseño. Es importante que esas decisiones sean hechas explicitamente, y que el programador esté consciente de los criterios fundamentales y de la existencia de soluciones alternativas.

Con el siguiente ejemplo indicaremos el uso de esta metodología para producir programas.

Problema: Ordene una lista de N (N>0) valores enteros en orden ascendente. Con este ejemplo mostraremos algunos algoritmos de diseno. Trataremos de segmentar el problema hasta encontrar las partes mas pequenas suficientes para resolverlo, entonces combinaremos las soluciones parciales para obtener una sola solución al problema.

Podemos dividir el problema en dos partes:

1.- Encontrar el valor mas pequeno entre el primero y N.
2.- Intercambiar el primero y el valor mas pequeno.

Se toma el primero elemento de la lista y se lo compara con los demás, hasta encontrar el menor valor. Este valor se lo intercambia con el primero. En caso de ser el primero el menor, permanecerá en el mismo lugar. Luego se continúa haciendo la búsqueda con los valores restantes (Primero+1,...,N), hasta tener toda la lista ordenada.

Una primera solución sería:

```
FOR por cada valor en el rango 1...N-1;
BEGIN
   Encuentre el valor mas pequeno entre el primero y N;
   Intercambie primero y el valor mas pequeno;
   Incrementar primero;
END;
Imprimir los valores clasificados;
```

Se puede refinar esta primera solución y obtener:

```
/* Obtener N y los valores de N */
Declarar el tipo de las variables;
BEGIN
   /* por cada valor en el rango de 1...N-1;
   FOR I:=1 to N-1 DO
   BEGIN
     /* Asignar el índice del menor valor de A[I]..A[N]
        a K, y cambie A[I] y A[K] */
   END;
/* Imprimir los valores clasificados */
END;
```

Es necesario validar que el valor de N sea positivo. Se puede chequear esta condición, insertando una sentencia IF antes del BEGIN del bloque.

```
IF N>0  /* Chequear valor */
   THEN BEGIN
      /*  Obtener los valores de N */
      /*  Clasificar la lista de valores */
        END;
```

Para encontrar el numero mas pequeño, entre el primero y N, se puede:

```
Asumir que el primer valor es el mas pequeno;
Si cualquier otro valor es mas pequeno, hacer ese el mas
pequeno.
```

Para encontrar el valor mas pequeño el programa se expande a:

```
FOR J:= I+1 TO N DO
   IF A[J] < X THEN
   BEGIN
      K:=J;X=A[J];
   END;
```

Y el intercambio con el primer valor es:

```
/* Intercambie el primero y el valor mas pequeño */
A[K]:=A[I];A[I]:=X;
```

Se puede ahora poner las piezas juntas e insertar las declaraciones apropiadas para obtener:

```
PROCEDURE SORT(N:INTEGER);
  VAR
    I,J,K,X:INTEGER;
    A:ARRAY[1..N] OF INTEGER;
  BEGIN
  IF N>0 THEN
    BEGIN
      FOR I:=1 TO N-1 DO
      BEGIN
        K:=I;X:=A[I];
        FOR J:=I+1 TO N DO
           IF A[J]<X THEN
           BEGIN
             K:=J;X:=A[J];
           END;
        A[K]:=A[I];
        A[I]:=X;
      END;
      /* Imprimir los valores clasificados */
    END;
  END;
```

Como se puede ver, mediante este ejemplo, la forma de obtener la versión final del problema ha sido mucho mas sencilla, pues se hicieron refinamientos graduales de los diferentes pasos en vez de codificar en forma desordenada sin seguir un buen diseño.

CONCLUSIONES

La ventaja de usar el refinamiento paso a paso, es que el código sirve como la documentación lógica del programa. Si se usaran diagramas de flujo para ir desarrollando el problema, esto sería incómodo, porque luego sería necesario transformar este diagrama en código fuente entendible por el computador. Este proceso es indeseable por algunas razones: Es redundante con el código y la redundancia en cualquier tipo de documentación debe ser evitada porque se incrementan los conflictos en los programas, por tanto disminuye la confiabilidad; además hay que tener cuidado de actualizar la documentación, lo cual es mas difícil si la documentación lógica esta físicamente separada del código.

Un programa que es diseñado de tal forma que chequee razonablemente los errores, será mas confiable. Esto puede ser conseguido usando las técnicas de Programación Estructurada, Diseño de Arriba hacia Abajo y Refinamiento Paso a Paso.

REFERENCIAS

1.- Bohm, Corrado, Giuseppe Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", Comunications of the ACM, Volumen 9, Numero 5, Mayo 1966.

2.- Dijkstra, E.W., "Goto Statement Considered Harmful", Comunications of the ACM, 1.968.

3.- Dijkstra E.W., "A Constructive Approach to the Problem of Program Conectness" BIT (3), 1968.

4.- Knuth, Donald E., "Structured Programming with GOTO Statement", Computing Survey, Volumen 6, Numero 4, Diciembre de 1974.

5.- Mills, Harlan, Mathematical foundations for Structured Programming, Report FSC72-6012, IBM Corp, Gaithersburg Maryland, 1972.

6.- Wirth, Niklaus, "Program Development by Stepwise Refinement", Comunications of the ACM. Volumen 14, Numero 4, Abril 1971.

7.- Wirth, Niklaus, "On the Composition of Well Structured Programs", Computing Surveys, Vol 6, Numero 4, Diciembre 1974.

8.- Wirth, Niklaus, Algorithms + Data Structures = Programs, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

9.- Yourdon, Edward, Techniques of Program Structure and Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

CODIFICACION

LENGUAJES DE PROGRAMACION

Una forma de obtener software confiable es desarrollando programas que tengan una alta probabilidad de ser correctos. Esta probabilidad puede ser incrementada de dos maneras: reduciendo la probabilidad de errores de programación, e incrementando la probabilidad de detectar cualquier error que exista. Un diseño apropiado del lenguaje puede contribuir para lograr ambas metas.

Los lenguajes de programación deberían guiar al programador no solo en el proceso de codificación, sino también en el proceso de producir programas correctos y del mantenimiento de los mismos.

Dos atributos de un lenguaje juegan un papel importante en la habilidad del programador para dominarlo: su tamaño y su naturaleza. Un lenguaje podría contener instrucciones que fomenten la mayor claridad posible en las sentencias, sin embargo la necesidad de más construcciones deben ser balanceadas con la necesidad de mantener el lenguaje simple. Los programadores deben preferir un lenguaje que dominen rapidamente, sin continuas referencias a manuales y sin la necesidad de crear programas de experimento para descubrir particularidades del compilador.

El lenguaje que se elija debe contener características que permitan que las intenciones que tenga el programador de violar las reglas del mismo, sean detectadas como errores. Ademas esta detección debe ser hecha lo más pronto posible (a tiempo de compilación), tanto para velocidad en el desarrollo del programa como para que se libre de la dependencia de las diferentes pruebas a tiempo de ejecución del mismo.

Para producir software con confiabilidad no se puede concluir cuando un programa está "terminado". Sistemas grandes nunca son estables, ellos continuamente necesitan crecer y ser corregidos. En este proceso se degenera la estructura del sistema y la familiaridad con el mismo. El rápido incremento del esfuerzo en el mantenimiento de sistemas puede ser reducido incrementando la comunicación entre los programadores que implementaron el sistema y los que lo deberán mantener. Es importante que el lenguaje de programación haga esta comunicación clara, conveniente y precisa.

Una característica de los lenguajes de programación que es común para muchos de ellos, pero aparece en diferentes formas, es el método de determinar el tipo de los operandos.

El poder del tipo de datos permite a un programador pensar en términos de su aplicación más que en las características de la máquina en la cual su aplicación correrá. Los tipos de datos pueden ser asociados con los operandos en tres formas: Estáticamente, aquí un tipo de dato es asociado con un identificador y durante su tiempo de vida solo se le pueden asignar valores del mismo tipo. Ej: El lenguaje PASCAL. Dinamicamente, aquí los tipos de datos son asociados con un operando dinamicamente durante la ejecución. Ej: El lenguaje SNOBOL. Indefinidamente, aquí el último tipo de un operando es el tipo del último valor asignado a él y son llamados lenguajes "sin tipo". Ej: El lenguaje BLISS.

Existen evidencias dadas por Gannon [2,3] que indican que el uso de operandos de tipo dinámico da como resultado errores que permanecen en los programas por mas tiempo que los errores que pudieran producirse con operandos de tipo estático.

EXPERIMENTO

Dos grupos formados por un total de 38 graduados y estudiantes de niveles altos en un curso de la Universidad de Mérida, programaron soluciones para un mismo problema dos veces. Un grupo de estudiantes programó soluciones, primero usando un lenguaje con declaraciones de tipo estático y luego con uno sin declaraciones de tipo. El segundo grupo también programó soluciones en los dos lenguajes pero en orden opuesto. Cada uno de los lenguajes contenían características comunes a los otros lenguajes implementados. Los problemas eran relativamente simples (48 a 297 líneas), y cuando un problema fue asignado se le dió a cada estudiante el manual del lenguaje que contenía ejemplos de programas similares.

Este estudio sirvió para realzar la confiabilidad de software en los lenguajes de tipo estático, ya que los errores cometidos por los usuarios con este lenguaje fueron menos frecuentes y severos que los errores cometidos por los usuarios que usaron el lenguaje sin declaración de tipo.

El lenguaje de programación usado tiene un considerable efecto en la confiabilidad de software. Mientras los lenguajes son de mas alto nivel producen menos errores, pues si un programador está usando un lenguaje de máquina, no usa la mayoría de su tiempo resolviendo el problema, sino estudiando los detalles de la máquina.

Los lenguajes de alto nivel eliminan algunos errores en el software, escondiendo las idiosincracias de las máquinas y permitiendo que cualquier función dada sea expresada en pocas sentencias. Los programas en lenguajes de alto nivel son mas entendibles, fáciles de cambiar, permiten mayor compatibilidad y portabilidad de programas y son mas fáciles de documentar.

La mayor ventaja de los lenguajes de alto nivel es la habilidad para expresar y manipular estructuras de datos complejas. La eficiencia es una consideración insignificante, podemos conocer que la eficiencia es lograda a través de la selección inteligente de algoritmos y estructuras de datos, no a través de microeficiencias logradas por el uso de un lenguaje de máquina. Además los compiladores generan código de máquina que es remarcadamente eficiente.

Por todo lo mencionado concluimos que debería eliminarse el uso de lenguajes de máquina y lenguaje ensamblador en la producción de programas, pues lo único que nos darían como resultado sería un programa inconfiable y difícil de mantener. Se deben reservar estos lenguajes para el diseño de compiladores y posiblemente también el diseño del hardware del sistema.

Además de la elección del lenguaje apropiado, es necesario también mantener un buen estilo de programación durante la codificación del sistema. Esto junto con el diseño, serán las bases principales para obtener un programa confiable.

ESTILO DE PROGRAMACION

Se denomina estilo de programación a la selección de hábitos o técnicas de programación que ayudan a producir programas que son:

1.- Correctos
2.- Eficientes
3.- Fáciles de Mantener y
4.- Legibles.

Las reglas de un buen estilo son el resultado del concenso entre programadores experimentados. Si cada programador usara un estilo muy individual al producir un programa, este podría ser incomprensible para otros. Una vez que los programadores se acostumbren a aplicar un buen estilo de programación, sus programas serán mas fáciles de entender, por ellos mismos, así como también por otros programadores.

La adherencia a las reglas de un buen estilo de programación, es un factor muy importante para obtener confiabilidad. Si desde la creación del software tenemos en cuenta un buen estilo de programación, obtendremos software mas confiable y mas fácil de mantener que otro que no siga estas reglas. Esta afirmación es hecha en base a experimentos realizados por programadores experimentados [7] sobre diferentes aspectos relacionados con el estilo de programación.

Para obtener mayor confiabilidad es necesario seguir las reglas indicadas a continuación, relacionadas con el estilo de programación, durante la etapa de creación del software:

APROVECHAR EL LENGUAJE

Una categoría importante en el estilo de programación es la forma en la cual el programador usa o no las características de los lenguajes de programación.

Aprender a usar las características del lenguaje: Ocasionalmente se puede ver que un programa en PL/I contiene un lazo DO para inicializar los elementos de un arreglo a cero. Esto es usualmente un signo de una falta de educación en el lenguaje de programación porque la simple sentencia A=0 podría ser suficiente. Lo mismo sería escribir un lazo para hacer una búsqueda en una tabla en COBOL, en vez de usar la sentencia SEARCH. Queremos indicar con esto que una buena educación en los lenguajes de programación es una buena inversión, pues con pequeños programas se eliminan fuentes potenciales de errores, con lo cual se reduce el costo del software, así como también se aumenta su confiabilidad.

Aprender a usar las bibliotecas y construir en ellas funciones: Muchos programadores están familiarizados con las funciones científicas que proveen los lenguajes como: seno, coseno, raiz cuadrada, pero existen otras no científicas que son también muy útiles. Por ejemplo el PL/I tiene cerca de 80 funciones como SUM y PROD que suman o multiplican, respectivamente, todos los elementos de un arreglo, o también la función TRANSLATE que reemplaza un caracter particular de una cadena de caracteres por el correspondiente caracter de otra cadena. El uso de tales funciones donde sean aplicables, proveen legibilidad a los programas y los hacen menos propensos a errores, aumentando por tanto el índice de confiabilidad de tales programas.

Evitar obscuridades del lenguaje: Ciertos aspectos como por ejemplo usar la variable de iteración de un lazo una vez que éste ha terminado podría ser una práctica peligrosa. Algunos lenguajes de programación hacen que el índice de un lazo sea igual a cero después de la finalización del mismo, otros hacen que el índice sea igual al último valor mas el incremento, etc, todo depende del lenguaje de programación que se esté usando. Por tanto, esta práctica, de usar el índice de un lazo después de que ha salido de su ámbito, debe ser evitada, pues puede producir errores en el programa.

No ignorar los mensajes de precaución: Existen ciertos compiladores que producen mensajes de precaución, que indican que el programador está cometiendo un error, y si éste no es de mayor severidad, completan la compilación. El código aquí debería ser cambiado para evitar así los errores que podrían producirse en la ejecución del programa.

MICROEFICIENCIAS

La peor violación al buen estilo son las muchas codificaciones para producir eficiencia. Por ejemplo: si cambiamos exponenciación por multiplicación para que sea más eficiente, si codificamos el programa en lenguaje de máquina, o si nos preocupamos de ver la forma mas rápida de hacer cero un registro, perdemos la meta principal del programa, su legibilidad, pues la codificación ya no indica tan claramente su objetivo. Además esta actitud está en cierta forma en conflicto con las eficiencias que el compilador está en capacidad de hacer. Las llamamos microeficiencias, porque se trata de hacer eficiente partes del código que no son relevantes, y en la mayoría de los casos no tienen incidencia en la eficiencia total del programa. Algunas reglas a seguir son las siguientes:

Ignorar todas las sugestiones de eficiencia hasta que el programa esté correcto: Una de las peores cosas que un programador puede hacer es tratar de que su programa sea eficiente, antes que funcione correctamente. Weinberg [9] refiere una anécdota acerca de un programa que fué inconfiable, sin esperanza de llegar a ser confiable, por su complejo diseño. Debido a esto un nuevo programador fué llamado a producir otra versión del programa que fuera confiable y en un plazo de dos semanas. Al momento de la demostración de la ejecución del programa, el nuevo programador remarcó que su programa tomaba 10 segundos por tarjeta de entrada. A esto el programador original dijo triunfante: "Pero mi programa toma solamente un segundo por tarjeta de entrada", a lo cual el nuevo programador respondió: "Pero su programa no funciona, y si el programa no tuviera que funcionar, yo podría escribir uno que me tome solamente un milisegundo por tarjeta de entrada".

Dejar que el compilador realice la optimización, muchos están capacitados para ello: Muchos compiladores usados en la producción de programas hacen una cantidad significante de optimización de código tales como reconocer que una expresión como A**2 puede ser evaluada como una multiplicación, remover expresiones constantes de los lazos DO, entre otras. De esta forma el código puede ser escrito de una manera mas simple, dejando que el compilador se preocupe de la optimización, y sin quitarle legibilidad al programa.

Nunca optimice a menos que esté forzado a hacerlo: Obviamente la eficiencia no es totalmente insignificante, pero a pesar de esto la optimización debería ser hecha solamente cuando la eficiencia es importante y conociendo precisamente que parte del programa necesita optimización.

DEFINICION DE LOS DATOS

Debido a la importancia de los datos son necesarias algunas reglas acerca de su definición y uso:

Declarar todas las variables explícitamente: Muchos lenguajes permiten a los programadores definir variables implícitamente, simplemente usandolas en sentencias ejecutables, pero los programadores que se consideren profesionales deberían definir o declarar todas las variables explícitamente al comienzo de cada módulo, indicando el nombre de la variable, su atributo o tipo, y lo que esta variable hace en el programa o módulo.

Declarar los atributos de cada variable: Ciertos lenguajes dan atributos a las variables si estos no son explícitamente dados por los programadores. Por ejemplo en FORTRAN si el nombre de una variable empieza con letras desde la I hasta la N, se asume que es una variable entera, además algunos compiladores permiten que se realizen alteraciones de los atributos asumidos como en el caso de lenguaje PL/I. Esta práctica resulta tremendamente peligrosa, Ejemplo: Un programador puede usar la variable entera ISUMA, cuando en realidad necesita una variable real. Como no especifica el tipo de variable, se olvida del atributo que esta tiene y en algunos casos puede pensar que la variable asumirá el tipo real de acuerdo al valor asignado a ella, pudiendo producir errores posteriores en el programa.

Nunca use una variable para mas de un propósito: El uso de la sentencia EQUIVALENCE en FORTRAN y REDEFINES en COBOL, las cuales dan alias a las variables, son una de las cosas mas confusas que pueden ser hechas en los programas. Su uso atenta contra la confiabilidad de los programas.

[illegible] para significados especiales. En la definición de parámetros de subrutinas es común [illegible] como "LINELEN representa el número de caracteres [illegible], o menos que [illegible] cero, [illegible] IXD está ocurriendo" [5]. El múltiple uso de las variables o parámetros [illegible] ambiguos y hace difícil [illegible] en los programas.

ESTRUCTURA DE LOS PROGRAMAS

Es necesario mantener cierta estructura en la escritura de los programas.

Codificar un ELSE por cada THEN: Esto debería hacerse aunque el programador no desee seguir una acción con la clausula ELSE, en cuyo caso se codificará un ELSE nulo. Esto facilita seguir el flujo de control del programa. Existen ciertas reglas referentes a que hay que evitar anidar un IF en una clausula ELSE, pero sin embargo es importante recordar que para ser consistentes, si un ELSE es codificado en un módulo, cada IF debe tener su propio ELSE.

Hacer decisiones exhaustivas: En la examinación de un parámetro de entrada del cual se esperen valores de 1, 2 y 3, es aconsejable que el programador haga decisiones por cada valor del parámetro, indicando la acción a seguir por cada uno, sin asumir, por ejemplo, que debe ser 3 y si no lo es, será 1 o 2. Ejemplo:

```
IF A=1 THEN 500
IF A=2 THEN 600
IF A=3 THEN 700
```

Es mucho mas claro usar esta definición que [illegible]

```
IF A<>3 THEN
          IF A=2 THEN 600
                 ELSE 500
       ELSE 700
```

Usar recursión si el lenguaje lo permite, solamente donde sea aplicable: Recursión es una forma alternativa de expresar la repetición en programas por medio de llamadas sucesivas al mismo módulo que la genera, sin embargo la recursión no debe ser usada donde una simple iteración podría ser suficiente. Por ejemplo para la función factorial X!=(X(X-1)!), podría ser usada una subrutina recursiva, sin embargo una manera mas simple de hacerlo es con una iteración DO.

La meta principal del programador debe ser escribir el código fuente para su audiencia principal, las personas, no para las máquinas. Esto requiere enfocar la atención en la claridad, simplicidad y comprensión del código, sacrificando,si no son importantes, otros criterios como la brevedad (tiempo de escritura y tipeado del programa), y eficiencia de ejecución.

Las siguientes guías tienen un efecto significante en la claridad de programas, para facilitar así el mantenimiento de los mismos:

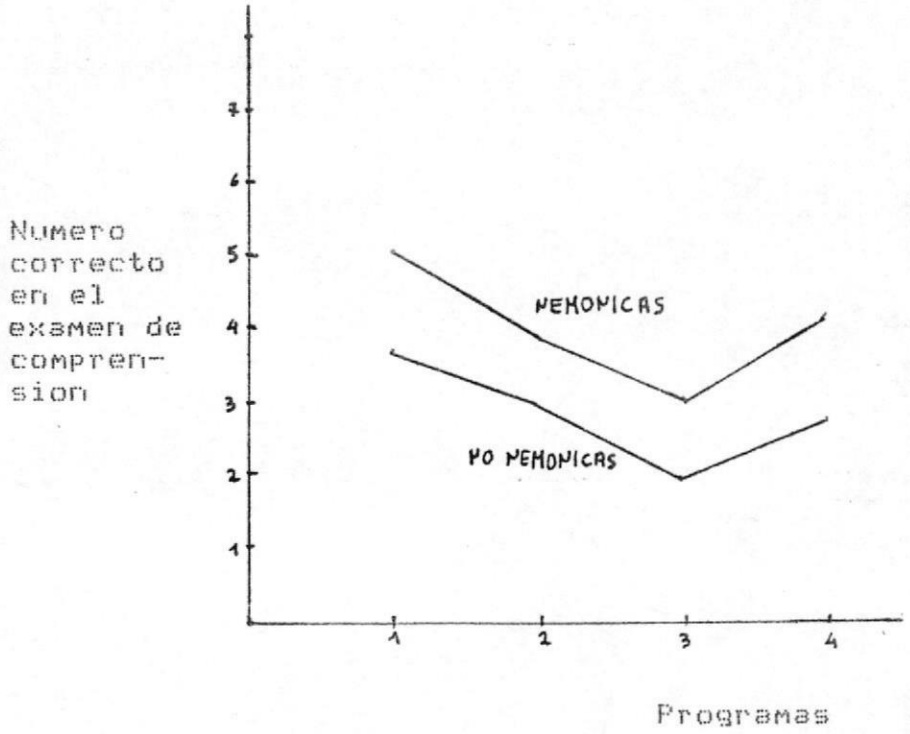
REGLAS DE ESTILO

NOMBRES DE VARIABLES

Los nombres de las variables deben ser seleccionadas para que simbolicen lo que ellas representan. Esta simple regla es una de las guías mas importantes en el estilo de programación. No hay nada peor que ver un programa con nombres de variables como: XX, XXX, EKK, AI, y una abundancia de comentarios (si es que existen) dentro del programa explicando el uso de ellas. Usar nombres de variables descriptivos ayudan a la legibilidad y minimiza el numero de comentarios necesarios. Tambien:

- Hay que evitar nombres de variables similares.
- Si se usan números en nombres de variables, hay que ponerlos solamente al final.
- Nunca use palabras claves del lenguaje como nombres de variables.
- Evitar variables temporales.
- Usar paréntesis para evitar ambiguedades en expresiones aritméticas.
- Escribir solamente una sentencia por cada línea del programa.

Don McKay [7] realizó un experimento con cuatro diferentes programas en FORTRAN que no contenían comentarios. Estos fueron escritos por categorías de dificultad por programadores experimentados y presentados a programadores novatos usando variables nemónicas (ISUMA, COEF, IDIVS) y no nemónicas (I1, I2, I3) con preguntas sobre comprensión de los mismos. Los grupos que analizaron los programas con variables nemónicas se comportaron significativamente mejor que los grupos no nemónicos para todos los 4 programas.



RESULTADOS DE COMPRENSION EN LOS EXPERIMENTOS CON NOMBRES DE VARIABLES

Este gráfico refleja los promedios de los puntajes obtenidos en las preguntas de comprensión para los cuatro programas. Por ejemplo en el programa 1 con variables nemónicas el puntaje promedio de todas las preguntas de comprensión realizada fue 5 mientras que el programa no nemónico tuvo un promedio tan solo de 3.80 en los resultados.

Cinco de los 10 experimentos realizados por Weissman [10] en programas PL/I y ALGOL-W estudiaron los nombres de las variables usadas. Estos experimentos se hicieron con tres tipos de nombres de variables: nombres nemónicos de

variables tendientes a ser largos (Ej: SPLIT-VALUE, MIDPOINT, STACK-LARGER-PIECE), nombres nemonicos cortos que fueron creados solamente borrando vocales (Ej: SPLT-VL, MDPNT, STCK-LRGR-PC), y nombres de variables sin significado que fueron largos terminos sin relacion (Ej: RAISE-FINAL, BEQUEATH, LEECH-ENTOMB-CLOSE). Los cinco experimentos produjeron un complejo y confuso set de resultados. Era muy dificil llegar a una acuerdo en la forma como incidian los nombres de las variables en el entendimiento de los programas, pero sin embargo Weissman concluyo que "Ambos nombres de variables usados, nemonicos y cortos nemonicos, ayudaban a la mejor comprension de los programas que los nombres de variables sin significado, y las variables con nombres nemonicos eran aun mejores que los nombres cortos nemonicos".

Significantes diferencias aparecieron en evaluaciones hechas por Weissman, sugiriendo que los programadores pensaron que los largos nombres nemonicos eran de mas ayuda de lo que realmente son. Es mas facil escudrinar programas con breves nombres nemonicos. W. J. Hansen [4] sugirio que que los nombres largos de variables son apropiados para variables globales las cuales son raramente usadas, mientras que las variables frecuentemente referenciadas (tales como indices de los lazos) pueden ser cortos.

COMENTARIOS

La mejor documentacion para la logica de un programa es una simple y limpia estructura de codigo usando indentacion y nombres de variables significativos, y siguiendo tambien otras guias del estilo de programacion mencionadas anteriormente. Teniendo el codigo estas propiedades, ademas de un numero suficiente de comentarios bien logrados el programa sera mucho mas facil de depurar.

Cuando una persona trata de leer un programa que no escribio el cual esta desprovisto de comentarios, esta forzado a pasar muchas horas siguiendo la logica del programa, o tendra simplemente que escribir un nuevo programa cuando algun cambio sea necesario. En este caso todo el tiempo ahorrado al evitar los comentarios originalmente, es usado muchas veces mas.

Un programa sin comentarios es probablemente el peor error que un programador puede cometer. Una buena regla a seguir es escribir comentarios mientras se escribe el programa por estar en este momento mas familiarizado con todos los detalles. Los comentarios insertados después son menos satisfactorios.

Hay que tratar de evitar comentarios excesivos que entorpecen la lectura del código y distraen la atención del mismo. Cuando uno ve un modulo con un gran numero de comentarios, inmediatamente se tienen dos pensamientos: o el programador escribió muchos comentarios porque su lógica era obscura y difícil, o el programador siguió la regla de que "al menos el 50% de todas las sentencias deben tener comentarios" [3], lo cual hace que muchos de ellos no sean significativos.

Los comentarios deben responder las preguntas del lector. Una técnica efectiva es inicialmente escribir el código sin comentarios y entonces leerlo, poniendo los comentarios donde creamos mas convenientes para el mejor entendimiento del código. Si sucede que el lector tiene una pregunta sobre un punto en particular, un comentario informativo para responder la pregunta deberá ser añadido.

Hay tres tipos de comentarios:

Comentarios de Prólogo:

Cada programa, subrutina o procedimiento debe tener un encabezamiento al comienzo que indique:

- Lo que hace el programa
- Como usarlo o llamarlo
- Una lista y explicación de variables y otras estructuras de datos usadas
- Instrucciones de entrada y salida
- Lista de subrutinas usadas
- El nombre de cualquier método cientifico usado, junto con una referencia donde obtener mas mas información sobre el mismo.
- Cantidad de memoria necesaria
- Tiempo de ejecución
- Requerimientos especiales de operación
- Limitaciones del programa
- Tamaño de los arreglos
- Fecha de escritura
- Nombre del autor.

El programador experimentado se regirá por estas reglas y agregará información que el considere también relevante en esta sección.

Comentarios de Directorio:

Si el programa es muy largo, se recomienda proveer un directorio o tabla de contenido en forma de comentario al inicio del programa. La tabla debe contener el nombre, localización y función de cada módulo.

Comentarios Explicativos:

Son insertados en el texto del programa para explicar cualquier parte del código que no sea obvio simplemente al leerlo. Deben aparecer antes de lazos importantes o sentencias condicionales indicando lo que hacen.

Los comentarios apropiadamente hechos proveen una indicación del flujo de datos y lógica del programa. Los comentarios deben explicar el propósito de un grupo de sentencias, no describir la operación de una sentencia. Ejemplo:

```
* Cheque    si COSTO es menor que cero *
IF COSTO    HEN GOTO 100
```

El ante no es un buen comentario ya que en este ejemplo el go puede decirnos que es lo que hace el programa, y unción de los comentarios es indicar por que lo hace

```
* Si       sto es negativo, imprimir totales por artículo.*
```

Weissma l realizó una serie de 10 experimentos con estudiantes iencias de Computación de la Universidad de Toronto usand rogramas de PL/I y Algol-W.

Sus experimentos consistian en comparar programas que tenian comentarios bien localizados y significativos, con programas cuyos comentarios eran muy breves, en programas de 50 hasta 150 lineas. Incluian ademas preguntas de llenar los blancos, simulaciones manuales y evaluaciones subjetivas (encierre un número del 0 al 9 indicando que tan bien se entiende el programa).

Los que tuvieron programas con comentarios extensos hicieron la simulación manual significativamente mas rápida, modestamente mejor en las preguntas de llenar los blancos, tuvieron significantes puntajes altos en la evaluación subjetiva pero cometieron mas errores en la simulación manual. Estos resultados, si bien favorecen el uso de comentarios, no proveen suficiente guía acerca de que tipo de comentarios son de mas ayuda y no define guías razonables para el volumen de comentarios. Los errores en la simulación manual se deben a que los programadores se guiaron más por los comentarios que por la codificación, en vez de usarlos como un complemento para el entendimiento del código.

Otro experimento fue realizado por Ken Yasukawa [6], construyendo dos versiones de un programa en FORTRAN de 23 líneas: uno con un simple comentario ("Este programa cuenta el número de items de entrada pares, impares y cero"), y el otro con nueve líneas adicionales de comentarios. Los 31 sujetos que recibieron la versión con el simple comentario tuvieron un puntaje promedio de 9.2, en las 15 preguntas de comprensión. Mientras que los 28 sujetos que recibieron la versión con muchos comentarios, tuvieron un resultado promedio de 10.0 siendo esta la base de calificación. Esta diferencia favoreciendo a los comentarios tuvo un nivel de significancia de 0.07. Mientras más bajo es el nivel de significancia, más alta es la seguridad de que los resultados obtenidos del experimento son confiables, por tanto este valor indica que el experimento tiene validez científica y sus resultados pueden ser tomados como válidos para sustentar la teoría de la conveniencia de los comentarios en los programas.

LINEAS EN BLANCO

El uso de líneas en blanco es una manera de mejorar el aspecto de un programa. Una simple línea en blanco puede ser usada para separar cada grupo similar de sentencias, y algunas líneas en blanco para separar cualquier sección mayor del programa.

El uso de líneas en blanco hace mucho mas fácil cualquier búsqueda de rutinas en un programa.

INDENTACION

Se refiere al uso de espacios en blanco, al comienzo de una línea, formateando el texto para indicar la estructura del programa. Se indentan sentencias para indicar que ellos juntos se pertenecen.

Esto no afecta la lógica del programa, pero ayuda grandemente a la legibilidad.

Algunos experimentos fueron realizados por Weissman [10] con indentación de programas en PL/I y ALGOL-W. El experimento tenía preguntas de llenar blancos y tareas de simulaciones manuales. La indentación no ofrecio ninguna mejora en estos aspectos, entonces se realizó el experimento incluyendo comentarios en los mismos programas, y esto reveló que los programas indentados fueron mas difíciles de simular manualmente y produjeron una pobre evaluación. Estos resultados sorpresivos fueron en contra de las recomendaciones contemporáneas en programación que sugieren el uso de comentarios e indentación.

Shneiderman y McKay [7] llevaron a cabo también experimentos con estudiantes de programación en dos grupos. Uno recibió un programa de búsqueda interactiva en un árbol binario en forma indentada y un programa de clasificación no indentado. El otro grupo recibió el primer programa en forma no indentada y el segundo en forma indentada. Cada programa fue escrito en Pascal y contenían una falla que debía ser encontrada.

Una vez efectuado el experimento, no fueron encontradas diferencias significativas. La consistencia de los resultados sugiere que la ventaja de la indentación puede no ser tan grande como se cree. Aunque la indentación puede no servir mucho para la búsqueda de errores en los programas, hay buenas razones para creer que puede ser efectiva en algunas circunstancias como: para organizar programas, anidar sentencias condicionales y de salto, así como también para producir un buen impacto visual del código.

CONCLUSIONES

Basandonos en los experimentos realizados por Weissman y Yasukawa sobre los comentarios en los programas, podemos decir que el buen estilo de programación tiene un efecto positivo en la confiabilidad de software. Un programador que sabe poner buenos comentarios y en el lugar que estos son necesarios, demuestra que sabe lo que está haciendo, pues los buenos comentarios surgen del conocimiento de los pasos seguidos durante la elaboración del programa. Estos programas por lo tanto seran mas confiables que los hechos por un programador que no tenga una idea clara del problema y no siga buenas reglas de estilo.

Los experimentos de Shneiderman y McKay realizados con nombres de variables, demuestran que si se utilizan nombres de variables significativos, los programas son mas confiables porque son mas fáciles de entender.

Los programadores profesionales (cualquier persona cuyo principal trabajo es producir programas para ser usados por otras personas [8]) gastan mas tiempo examinando programas existentes que escribiendo nuevos programas, por tanto necesitan tener a su disposición programas mas fáciles de entender.

Hemos visto que las reglas de estilo ayudan a la comprensión de programas. Esto es de vital importancia pues las actividades de depuración, mantenimiento y extensión de programas son largamente dependientes de la lectura y entendimiento de los mismos, y de como esto es representado en un lenguaje de programación.

Un buen consejo a seguir: el esfuerzo extra necesario para hacer un programa legible es mínimo en comparación con el costo de revisarlo, localizar un error o reescribir un programa ilegible y poco confiable. Cada tema de este texto es una evidencia de la incidencia favorable de estos hábitos de programación en la confiabilidad de sistemas.

REFERENCIAS

1.- Gannon J. D., "An Experimental Evaluation of Data Type Conventions", Comunications of the ACM, Volumen 10, Numero 8, Agosto 1977.

2.- Gannon J. D., J. J. Hornig, "Lenguage Design for Programming Reliability", IEEE Transactions of Software Enginiering, Volumen SE-1, Numero 2, Junio 1975.

3.- Gannon J. D., Data Types and Programming Reliability: Some Preliminary evidence, MRI Symp. on Comptr. Software Eng., Volumen 24, Polytechnic Press, Polytechnic Institute of N.Y., 1976.

4.- Hansen, E. J., Measurement of Program complexity by the pair (Cyclomatic number, operator count), ACM SIGPLAN Notices 13,3, Marzo 1978.

5.- Myers, Glendford J., Software Reliability: Principles & Practices, John Willey and Sons, New York, 1976.

6.- Shneiderman, Ben, Software Psychology, Human Factors in Computer and Information Systems, Winthrop Publishers, Inc., Cambridge, Massachusetts, 1980.

7.- Shneiderman, Ben, D. McKay, Experimental Investigations of Computer program, Debugging and Modifications, Proceedings of the 6th International Congress of the International Ergonomics Association, Julio 1976.

8.- Van Tassel, Denie, Program Style, Design, Efficiency, Debugging and Testing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

9.- Weinberg, G. M., The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1971.

10.- Weissman, L., Psychological Complexity of Computer Programs: An Experimental methodology, ACM SIGPLAN Notices, 9, 1974.

PRUEBA

PRUEBA

Es el proceso de ejecutar un programa con la intención de encontrar errores [1]. La meta de la prueba de un programa es asegurarse de que el programa resuelve el problema supuesto, y que se obtienen los resultados correctos bajo todas las condiciones, es decir cualquier conjunto de datos que sean aplicables al problema.

La persona que realiza la prueba debe tratar de que el programa falle. Es imposible garantizar la ausencia de errores en un programa, por tanto se debe tratar de buscarlos e indicar que estos existen. Si un programa trabaja correctamente bajo un gran conjunto de pruebas, esto no permite decir que el programa no tiene errores, pues muchas veces puede ser un error lo que el programa no haga.

Si la meta de la persona que realiza la prueba es demostrar, mediante un conjunto de pruebas, que el programa no tiene errores, su subconciente le hará seleccionar pruebas que tengan una alta probabilidad de ejecutarse correctamente, y bloqueará cualquier pensamiento de realizar pruebas con la que no se sienta seguro de los resultados. Experimentos sicológicos muestran que la mayoría de las personas que tienen esta meta, orientarán sus actividades para obtenerla. Subconcientemente no se permitirán trabajar contra esta meta, usando un conjunto de pruebas que traten de exponer los errores del programa. Ya que se reconoce que no existe ningún programa no trivial que esté perfectamente codificado y diseñado, y que siempre contendrá un cierto número de errores, el uso mas productivo de la prueba es encontrar tantos errores como sea posible [1]. Para hacer esto, y para evitar la barrera psicológica, la meta debe ser tratar de encontrar el mayor número de errores.

Nunca se debe asumir que el programa es correcto simplemente porque es aceptado por el computador, es completamente compilado, y sus resultados esperados son obtenidos. Lo que se debe hacer es obtener algunos resultados, no necesariamente los resultados correctos, haciendo pruebas con datos que sean absurdos al programa, pues muchos errores de lógica pueden aún existir.

La prueba de un programa debe ser hecha por partes, comenzando por la prueba de un módulo simple y terminando con la prueba final del sistema. Si la prueba no se realiza en una forma organizada, hay pocas probabilidades de desarrollar software confiable. Las formas tradicionales de pruebas normalmente son hechas en una de dos direcciones: Prueba de abajo hacia arriba (Bottom-Up) o la prueba de arriba hacia abajo (Top-Down).

PRUEBA DE ABAJO HACIA ARRIBA (BOTTOM-UP)

Esta es la forma tradicional de probar programas. Primeramente son probados los módulos llamados "unidades de prueba", que son los módulos terminales, o sea aquellos que no llaman a otro módulo. Una vez que estos módulos son probados, las llamadas a estos módulos, son confiables. El siguiente conjunto de módulos a ser probados, son los módulos que directamente llaman a estos módulos de prueba. Estos módulos de mas alto nivel, no son probados en forma aislada, son probados con los módulos de mas bajo nivel que fueron previamente probados. Este proceso se repite hasta llegar al programa principal. En este punto se completa la prueba de los módulos y la prueba del programa integrado.

La principal crítica a este modelo de prueba es que los errores serios en el diseño y en las relaciones que existen entre los módulos no aparecen hasta estar cerca del final del proyecto, entonces se hace necesario realizar mayores revisiones al código.

Con este método de prueba, obtener la confiabilidad del programa total es mas difícil, pues se puede llegar al final de la prueba y encontrar que existe un error que incide en los módulos inferiores, siendo necesario rehacer el programa y en esta práctica se pueden producir muchos nuevos errores, además del hecho de que se incrementará el costo del proyecto y probablemente se retrasará el avance del proyecto con respecto al tiempo estimado inicial.

PRUEBA DE ARRIBA HACIA ABAJO (TOP DOWN)

Se inicia la prueba con el módulo principal que en este método es la unidad de prueba en la estructura, del programa, una vez que este módulo es probado, los módulos directamente llamados por este módulo son enlazados uno a uno con el módulo tope (unidad de prueba) y esta combinación es probada. Este proceso se repite hasta que todos los módulos sean combinados y probados.

Una ventaja de este método es que permite detectar errores críticos [illegible] principal del programa es probado primero y continuamente son añadidos nuevos módulos y [illegible] se evita realizar cambios en los módulos de bajo nivel al encontrar errores en la lógica del programa principal produciendo una pérdida de trabajo.

Además los datos de prueba crecen con el programa, pues los datos de prueba son añadidos junto con los módulos que se van a probar. Este proceso evita tener datos de prueba separados para cada módulo. Con este método es mas fácil obtener programas confiables, pues los errores son detectados a tiempo para ser corregidos, sin que esto influya en el resto del programa.

Este proceso produce resultados rápidos, pues una parte del programa estará trabajando, y algunos resultados pueden ser generados aun cuando el programa no esté completo, estos resultados pueden ser mostrados al usuario, de esta manera el usuario no solo ve lo que está sucediendo si no que también puede darse cuenta de algo que requiera obtener del programa. Cualquier cambio u omisión puede ser rectificado a tiempo antes de que una extensiva codificación sea realizada.

PRINCIPIOS DE PRUEBA

Existen algunos principios que se deben seguir al probar los programas los cuales son fundamentales para la obtención de software confiable:

-UNA BUENA PRUEBA ES AQUELLA QUE TIENE UNA ALTA PROBABILIDAD DE DETECCION DE ERRORES QUE NO HAN SIDO DESCUBIERTOS, NO UNA PRUEBA QUE MUESTRE QUE UN PROGRAMA TRABAJA CORRECTAMENTE.

Este es el principio fundamental de la prueba, ya que es infructuoso tratar de demostrar que el programa no tiene errores. La prueba debe ser el proceso de tratar de descubrir errores previamente no detectados en el programa.

-UNO DE LOS PROBLEMAS MAS DIFICILES EN LA PRUEBA ES SABER CUANDO TERMINAR.

Realizar una prueba exhaustiva de cada valor de entrada es casi imposible, ya que hay tantas variaciones de datos de entrada y todas no pueden ser probadas. Lo recomendable es elegir basándose en el diseño de datos de entrada del programa un número finito de casos que maximicen la probabilidad de detección de errores.

- NO ES RECOMENDABLE PROBAR NUESTRO PROPIO PROGRAMA.

La prueba tiene que ser un proceso extremadamente destructivo, y existe una razon sicológica que evita que un programador sea destructivo con su propio programa [1]. Por este motivo, ademas de la persona que hizo el programa, deben existir por lo menos una o dos mas que realicen la prueba, para que de esta forma se puedan detectar errores que el programador pasó por alto.

-UNA PARTE NECESARIA DE TODA PRUEBA ES UNA DESCRIPCION DE LA SALIDA O RESULTADOS ESPERADOS.

Los resultados esperados deben ser conocidos de antemano, pues el error mas común es fallar en predecir los resultados esperados, antes de que la prueba sea ejecutada.

-DISENAR LAS PRUEBAS TANTO PARA CONDICIONES DE ENTRADA INVALIDAS COMO PARA CONDICIONES VALIDAS.

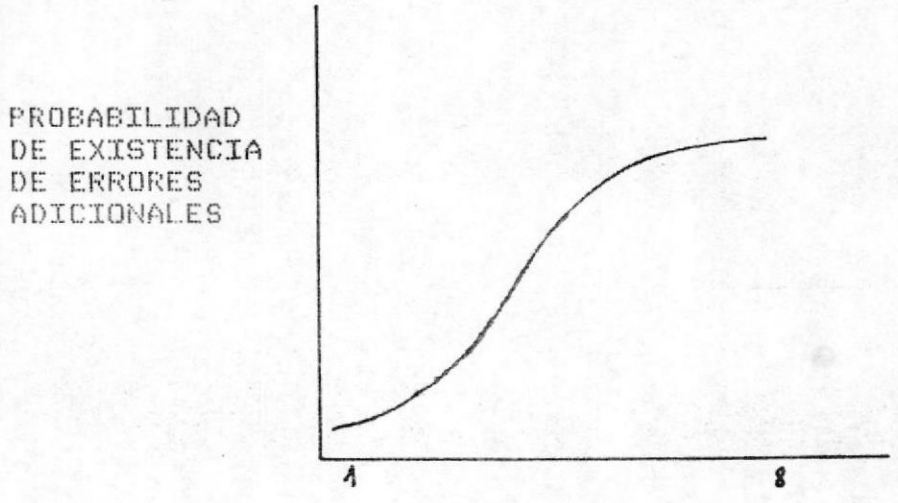
Muchos programadores orientan sus pruebas hacia condiciones de entrada esperadas, olvidándose de condiciones de entrada inesperadas. Las pruebas que representan condiciones de entrada inesperadas tienen frecuentemente un alto nivel de detección de errores.

-EXAMINAR LOS RESULTADOS DE UNA PRUEBA.

Probar un programa significa algo mas que ejecutar un número suficiente de pruebas, sino que también implica examinar sus resultados, pues esto facilita encontrar la causa de un error.

-EN LA MISMA MEDIDA QUE SE INCREMENTA EL NUMERO DE ERRORES DETECTADOS EN UNA SECCION DE UN PROGRAMA, SE INCREMENTA LA PROBABILIDAD DE EXISTENCIA DE MAS ERRORES NO DETECTADOS.

La relación entre los errores descubiertos y no descubiertos es mostrada en la siguiente figura [1]:



Dado dos módulos en los cuales la prueba ha detectado, 1 y 8 errores, respectivamente, la curva nos indica que el módulo con 8 errores conocidos, tiene mas alta probabilidad de tener errores aún no detectados que el primer módulo con 1 solo error. Si una parte particular de un sistema aparece con mayores errores durante la prueba, esfuerzos adicionales de prueba deberán ser enfocados en dicha parte. En otras palabras las pruebas en esta parte del sistema deberán ser de mayor rigor que en otras partes.

-ASIGNAR PROGRAMADORES CREATIVOS PARA QUE REALIZEN LA PRUEBA.

Es necesario tener un número de programadores altamente creativos, ya que la prueba, particularmente en el caso del diseño, es el área de software que demanda la mayor creatividad.

-ASEGURARSE QUE LA PRUEBA SEA UNA CLAVE OBJETIVA EN EL DISENO DEL SOFTWARE.

La dificultad de probar un programa está en relación al diseño y estructura del programa, por tanto debemos ser muy cuidadosos en esta etapa, así la prueba sera una tarea mas fácil.

-NUNCA SE DEBE ALTERAR EL PROGRAMA PARA HACER MAS FACIL LA PRUEBA.

Es frecuente alterar un programa para hacer la prueba mas fácil. Por ejemplo: un programador con un módulo que contenga un lazo WHILE DO, que tiene 100 iteraciones, podría alterar este lazo para iterarlo solo 10 veces y hacer la prueba mas rápida y fácil, pero corriendo el riesgo de que se presenten errores en las iteraciones finales del lazo.

-LA PRUEBA DEBE COMENZAR CON LOS OBJETIVOS DEL PROGRAMA

La prueba es una parte del ciclo de desarrollo de software. Las diferentes pruebas deben ser disenadas, implementadas, probadas y finalmente ejecutadas. Los objetivos del programa deben siempre estar presentes en las pruebas, y un porcentaje relativo de errores deben ser detectados en cada fase de la misma.

CONCLUSIONES

Un programador es juzgado principalmente por el número de errores que ocurren después de que el programa que escribió es entregado para uso general. En consecuencia, es mejor tener una reputación de ser un poco lento produciendo un buen y probado programa, que tener la reputación de ser rápido produciendo programas que contengan errores.

La prueba es una materia difícil de discutir, aunque tiene alguna relación con la confiabilidad de software, esta es limitada pues la confiabilidad de un programa es establecida por el grado de perfección en las etapas de diseño. La mejor manera de obtener software inconfiable, es dejando los errores iniciales producidos en el diseño; sin embargo la probabilidad de un diseño perfecto en un programa largo, es muy pequeña. El objetivo de la prueba es localizar el número de errores que permanecen en un programa.

Se puede evaluar la confiabilidad de un programa, al predecir los posibles errores y su frecuencia, es decir la probabilidad de que un sistema trabaje sin errores por un periodo determinado y el tiempo promedio que se requiere para reparar un error.

La prueba no es el método mas efectivo para asegurar la confiabilidad del software, pues esta puede ser asegurada mas fácil y productivamente, al comienzo del desarrollo del software, y el impacto que producen los errores de un programa, podrían ser reducidos mas efectivamente durante la etapa del diseño. Sin embargo, una prueba bien diseñada, llevada a cabo por programadores creativos y, preferiblemente, por otros programadores que no sean los mismos que codificaron originalmente el programa, permiten detectar errores remanentes en un programa bien diseñado, o en caso contrario, permiten encontrar la mayor cantidad de errores posibles, antes de que el programa entre a su fase de vida útil.

REFERENCIAS

1.- Myers, Glendford J., Software Reliability Principles & Practices, John Willey and Sons, New York, 1976.

DEPURACION

DEPURACION

Es el arte de localizar un error una vez que su existencia ha sido establecida. Un programa depurado es aquel para el cual no se han encontrado aún un conjunto de datos de prueba que puedan hacerlo fallar [2].

Muchos programadores confunden la depuración con la prueba de programas. Si es obvio que un programa no trabaja correctamente, entonces esta siendo depurado. Por lo tanto la depuración siempre empieza con una evidencia de que el programa falla. Si el programa parece estar trabajando correctamente, entonces está siendo probado. La prueba determina que un error existe; la depuración localiza la causa del error y lo elimina.

Remover errores de un programa no es una tarea fácil, por lo que es la mas evitada. Desafortunadamente, los humanos introducimos errores durante el proceso de desarrollo del programa [3]. Estos errores pueden ser:

1.- Sintácticos.- Sintaxis incorrecta en el uso del lenguaje de programación.

2.- Semánticos.- Errores en el diseño o composición del programa.

Los errores semánticos son mas problemáticos, pues el programa se ejecuta, pero no de acuerdo a las especificaciones. Los errores que se revelan en la salida pueden ser mas fáciles de encontrar que aquellos errores que ocurren irregularmente y no tienen impacto en la salida.

La depuración comienza después de que todos los mensajes de errores de sintaxis son eliminados.

Se usan datos de prueba simples para comenzar la depuración. Si estos datos de prueba no producen resultados correctos, entonces 5 situaciones son posibles [4]:

1.- El programa no completa la compilación pero no hay errores de sintaxis.

2.- El programa se compila, se ejecuta, pero no produce resultados.

3.- El programa se compila, se ejecuta, pero termina prematuramente.

4.- El programa se compila, se ejecuta, pero produce resultados incorrectos.

5.- El programa no termina de ejecutarse (tiene un lazo infinito).

1.- Compilación no completada.- Esta situación indica un error en algún lugar del programa. Los mensajes de errores del sistema aparecen en esta situación y pueden ser usados para localizar el error. Sin embargo se necesita un poco de experiencia para interpretar estos mensajes. Un buen método para ganar experiencia con los mensajes del sistema, es mantener un apunte de los mensajes anteriores y sus causas, pues es fácil olvidar que existió un problema si no se mantiene un registro escrito.

2.- Ejecución pero sin resultados.- El programa ejecuta pero no produce ningún resultado. Este tipo de error puede ser causado por un error lógico o un error del sistema. Un ejemplo de un error lógico podría ser un programa que comienza su ejecución y entonces bifurca al final del programa sin producir ninguna salida.

Los errores del sistema son causados por algún error que termina la operación del sistema sin permitir que la ejecución continúe. Normalmente no existe indicación del lugar donde se encuentra el error. Las causas de estos errores del sistema pueden ser una división para cero, bifurcaciones al área de datos tratando de ejecutarlos, subíndices de arreglos incorrectos, sobrepasar el rango numérico, etc.

3.- Terminación prematura.- El programa se compila, comienza su ejecución y produce alguna salida, pero entonces termina de ejecutarse prematuramente. Si produce alguna salida, entonces tenemos por lo menos alguna indicación del lugar donde se encuentra el error y podremos buscarlo a partir del punto donde la ejecución fue interrumpida. Si no se produce una salida, se complica mas el proceso de la depuración, pues no existe una idea del lugar donde se encuentra el error.

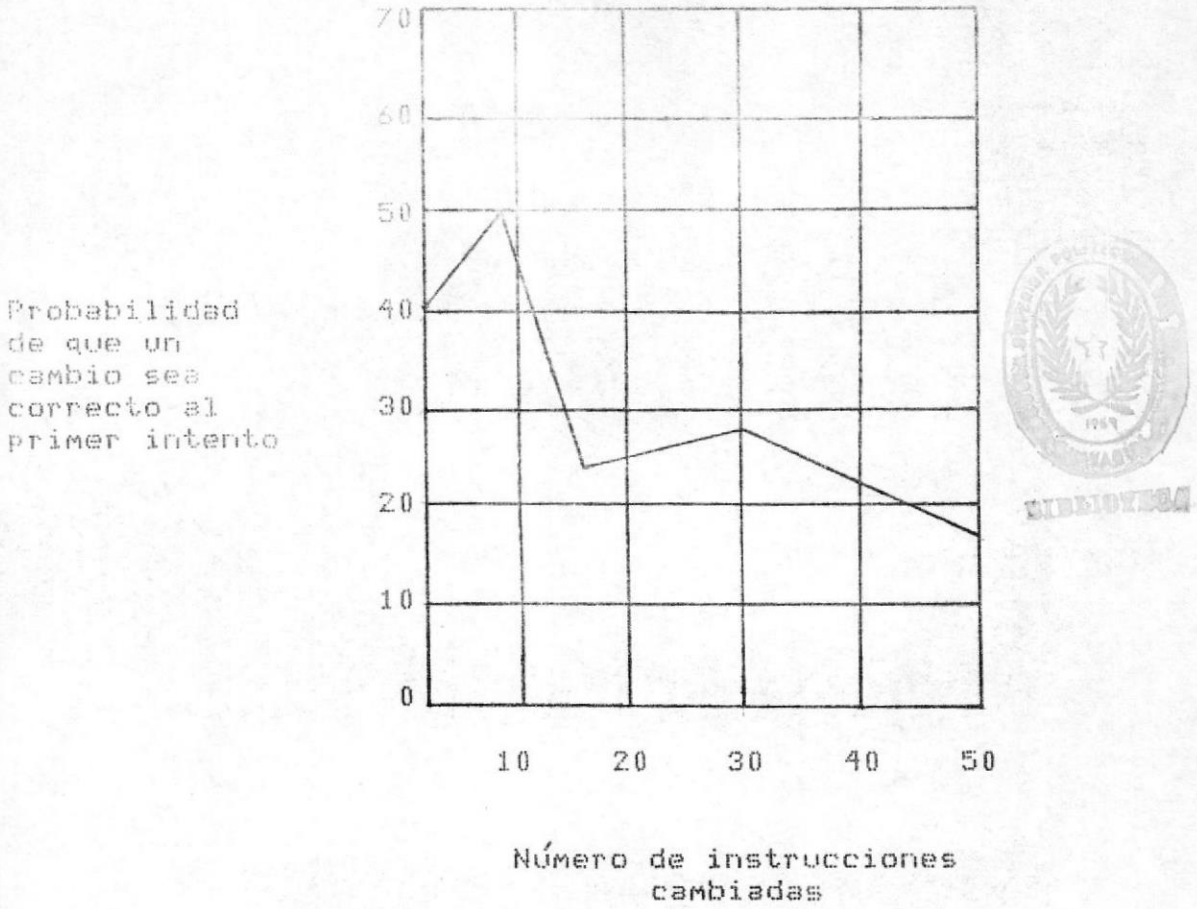
4.- Salida Incorrecta.- El programa se ejecuta pero produce una salida incorrecta. Los programadores de experiencia se consideran felices cuando alcanzan este estado, pues indica que el programa está basicamente bien y que la lógica es correcta.

5.- Un lazo infinito.- Este error normalmente no es difícil de encontrar. Si no se puede determinar inmediatamente cual es el lazo infinito, simplemente añada sentencias de impresión, con algún mensaje indicativo, antes y despues de cada lazo sobre el cual tenga sospechas. No ponga sentencias de impresión dentro de los lazos, porque producirán cientos de líneas impresas. Las sentencias de impresión proveeran salidas que indicarán cual es el lazo que comienza a ejecutarse pero que nunca termina.

CORRECCION DE ERRORES

Una vez que la causa de un error ha sido encontrada, hay muchas correcciones que podrian hacerse para solucionar el problema. Antes de que cualquier solución particular sea adoptada se debe determinar exactamente cual parte del sistema será afectada por la solución propuesta.

La experiencia demuestra que muy frecuentemente la corrección de un error introduce otros errores. En la siguiente figura se demuestra la probabilidad de que una corrección sea exitosa como una función del número de instrucciones que tienen que ser *cambiadas al hacer la corrección. Aunque solo una instruccion sea cambiada, la probabilidad de que la corrección sea correcta al primer intento es de apenas el 50% [1].



Si existen algunas alternativas para escoger, se debe seleccionar aquella que esté mas de acuerdo con la estructura existente en el programa.

Al mismo tiempo que un error es corregido, la documentación del sistema debe ser actualizada también.

Naturalmente despues de cada corrección, el sistema debe ser chequeado para determinar si el error actualmente ha sido eliminado y si nuevos errores han sido introducidos en el proceso. Esta advertencia es particularmente apropiada en el caso de que los cambios sean realizados a la lógica del programa. La modificación a la estructura lógica de un programa podria corregir un error pero frecuentemente introducirá nuevos errores.

La fase de prueba que debe seguir a cada corrección puede frecuentemente involucrar mucho mas esfuerzo que la corrección actual.

CONCLUSION

La principal forma de depurar un programa es por medio de información impresa, de esta manera los errores de los programas pueden ser localizados mas facilmente. Hay que determinar los lugares en los cuales se pueden poner las sentencias de depuración, pues de otra forma se perderá mucho tiempo de máquina y de personal para depurar un programa. Cuando la depuración es terminada, el programa definitivamente resuelve el problema.

La depuración tiene una amplia relación con la confiabilidad del software, pues esta es definida como la probabilidad de que el software se ejecutará por un particular periodo de tiempo sin errores, y la depuración encuentra estos errores y los corrige.

La magnitud de los errores difiere considerablemente, lo cual implica que la confiabilidad es una función directa de la severidad de los errores, de su frecuencia y del impacto que los errores tienen sobre el sistema.

REFERENCIAS

1.- Kopetz, K., Software Reliability, Springer-Verlag, New York, 1979.

2.- Myers, Glendford J., Software Reliability: Principles & Practices, John Willey and Sons, New York, 1976.

3.- Sheiderman, Ben, Software Psychology Human Factors in Computer and Information Systems, Winthrop Publishers, Inc., Cambridge, Massachusetts, 1980.

4.- Van Tassel, Dennie, Program Style, Design, Efficiency, Debugging and Testing, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

MANTENIMIENTO

MANTENIMIENTO

El desarrollo de un sistema se ha completado cuando el usuario lo ha puesto en funcionamiento, despues de las pruebas realizadas sobre el mismo. Cualquier trabajo que deba ser realizado en el sistema después que se encuentra en funcionamiento, se conoce como mantenimiento de software [3].

El mantenimiento es definido también como el proceso de modificar un software operacional mientras sus funciones primarias permanecen intactas [5]. Por ejemplo en un sistema de inventario las funciones primarias serian el ingreso, salida y actualización de artículos. Se pueden dividir dos categorias de mantenimiento: Actualización de software, que consiste en cambiar las especificaciones funcionales, por ejemplo cambiar el método para calcular el costo de un artículo y reparación de software, lo cual deja las especificaciones funcionales intactas, corrigiendo solo los errores que puedan aparecer.

La necesidad de mantenimiento de software aparece por las siguientes razones:

1.- La corrección de errores que no fueron detectados durante la Prueba.

2.- Cambio de los requerimientos del sistema: Los requerimientos del sistema pueden cambiar durante su ciclo de vida. En muchos casos el usuario no puede definir exactamente todas las tareas que el sistema debe ejecutar. Con la experiencia práctica del nuevo sistema, se llegan a determinar otras funciones que son necesarias.

3.- Cambios en la configuracion del hardware: Los avances de la tecnología en el campo de la computación, producen equipos que proveen una solución efectiva a aplicaciones particulares. Si se desea tomar ventaja de estos avances, entonces tanto el hardware como el software deben ser actualizados. Otra razón para cambiar el hardware, es que las aplicaciones pueden necesitar una mayor expansión en memoria, siendo el hardware actual insuficiente, entonces el software deberá ser adaptado al nuevo equipo.

4.- Perfeccionar el funcionamiento: Una vez que el sistema ha sido puesto en marcha, nos podemos dar cuenta de cuales son las relaciones entre sus componentes. Con cambios sistemáticos pueden ser eliminadas las relaciones innecesarias, perfeccionando el sistema.

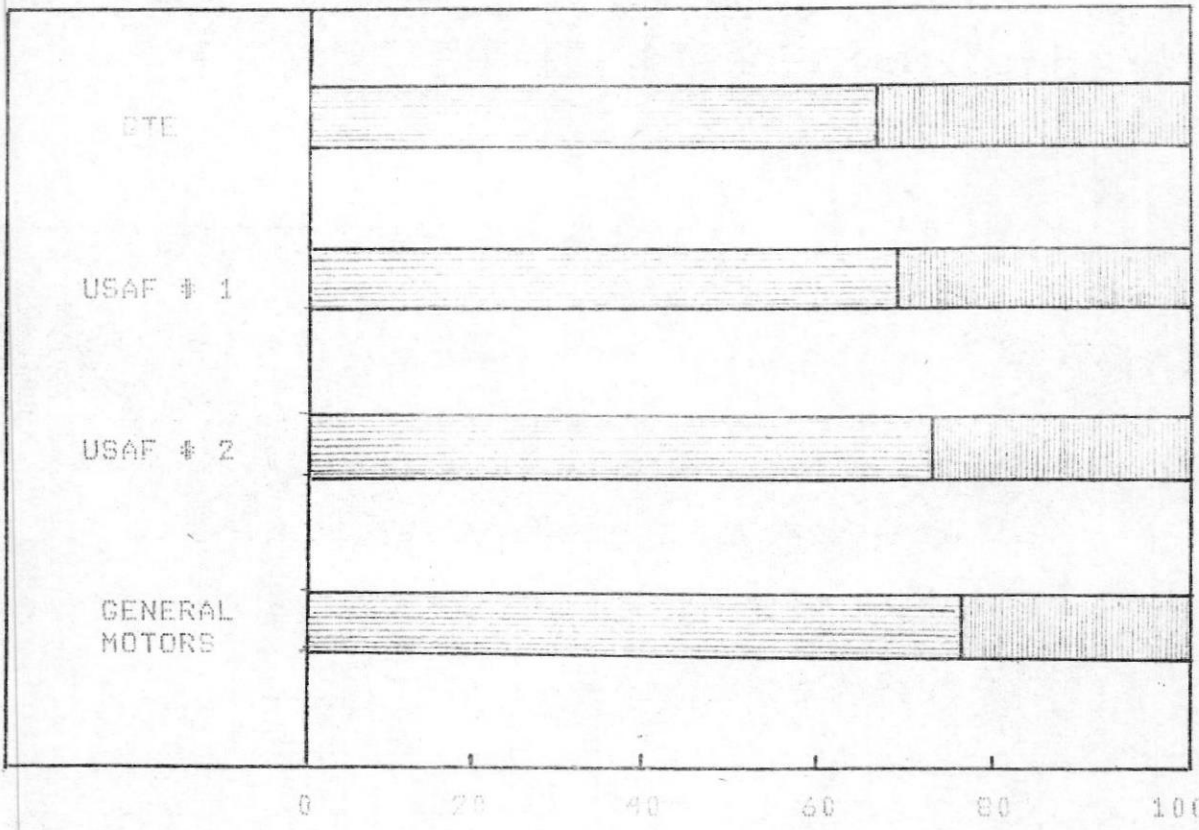
COSTO DE MANTENIMIENTO

Como se indica en la siguiente figura, probablemente cerca del 70% del costo total del software es gastado en el mantenimiento del mismo. Un articulo publicado por Elshoff [2] indica que la General Motors invirtió cerca del 75% del costo total de software en la etapa de mantenimiento y que esta situación es típica en las actividades de software.

Daly [1] indicó que el 65% de los costos para un software a tiempo real de la GTE (General Telephone and Electronics) durante 10 años de su ciclo de vida, fueron dedicados al mantenimiento. Asi también en dos sistemas del comando de la Fuerza Aérea de los Estados Unidos (USAF), la porción de mantenimiento de 10 años de su ciclo de vida, costó alrededor del 67% y 72% respectivamente, basandose en el costo total del software.

Debido a que el costo de mantenimiento a menudo excede del costo de programación original, los programas deben ser diseñados para facilitar la tarea de mantenimiento. Algunas instalaciones que realizan software han descubierto que una de las formas de reducir el costo es permitir que los programadores reescriban algunas rutinas completamente. Si estas rutinas problemas son identificadas y reescritas, el costo de mantenimiento asociado con ellas a menudo suele decrecer.

Además los sistemas no son optimizados en función de mantener la eficiencia, sino basándose en otros criterios, como por ejemplo cambiar el diseño de entrada, cambiar un proceso en particular. En algunos casos estos criterios comprometen la documentación y la estructura del sistema. En otros casos se quiere optimizar el lenguaje o el uso de hardware, dando como resultado un aumento en el costo de mantenimiento del software.



Porcentaje de costo de 10 años del ciclo de vida

ANALISIS DE COSTO DEL CICLO DE VIDA DEL SOFTWARE

Cada vez que el mantenimiento se lleva a cabo, por cualquier razón, es requerida la modificación de un sistema. Las consecuencias directas de esto son difíciles de prever. Se pueden agrupar los factores que influyen en el mantenimiento de software, en algunas categorías:

Entendimiento: Tiene dos aspectos principales: la disponibilidad del personal de software, que son quienes están familiarizados con el sistema y accesibles para el trabajo de mantenimiento; y la inteligibilidad del sistema por si mismo. Este último es el criterio mas importante por el cual la estructura del sistema es juzgada. Si la estructura del sistema es clara y entendible, el sistema será mas fácil de mantener y esto implica desde el punto de vista de la confiabilidad, que no se cometeran nuevos errores al llevar a cabo esta actividad.

Estandarización: En la mayoria de los casos no es el que desarrolló el sistema, sino otra persona, quien realiza el mantenimiento del mismo. En este caso puede ser decisivo para el mantenimiento que hayan sido usadas herramientas estandares durante el desarrollo del sistema, que se presente un buen enfoque sobre el mismo, es decir una idea clara y además la documentación correspondiente.

Documentación: La documentación para los propósitos de mantenimiento, será diferente a la documentación desarrollada durante el diseño del sistema. La documentación del mantenimiento debe ser hecha desde el punto de vista de cambios y modificaciones posibles. Es necesario ver el efecto que tendrá la modificación, de una cierta sección del sistema, en el resto del mismo. Es una buena práctica chequear el diseño original y documentar el cambio y sus resultados en la etapa del mantenimiento.

Prueba: Cada modificación hecha produce como resultado un nuevo sistema, del sistema original. Es necesario hacer una prueba completa del sistema modificado, esto será importante en el mantenimiento del nuevo sistema.

Expandibilidad: Un sistema es expandible, cuando es posible introducir un cambio deseable en él. Es aconsejable hacer provisiones de memoria durante el diseño del sistema para alteraciones eventuales.

CONCLUSIONES

En la práctica la razón de error de un sistema que ha experimentado numerosas modificaciones, inicialmente decrece, alcanza un mínimo y entonces se incrementa nuevamente [4]. Esta propiedad puede ser explicada por la influencia del mantenimiento en la confiabilidad. Inicialmente, despues que se ha puesto en marcha el sistema, errores previamente no detectados son corregidos. Posteriormente, los cambios introducidos durante el mantenimiento, perturban la estructura original del sistema y hay un incremento de la conexión entre los componentes del sistema. Esto equivale a un incremento en la complejidad y da como resultado una disminución de la confiabilidad.

Cualquier cambio propuesto debe ser cuidadosamente planeado y la estructura del sistema debe ser mantenida. Una estrategia planeada de mantenimiento, tiene un efecto muy positivo en la confiabilidad y en la vida esperada del sistema.

REFERENCIAS

1.- Daly, E. B., "Management of Software Development", IEEE Trans. Software Engineering, Julio 1979.

2.- Elshoff, J. L., "An Analysis of Some Commercial PL/I Programs", IEEE Trans. Software Engineering, Junio 1976.

[illegible] Yorl 1975.

4.- Odgin, J. [illegible] "Designing Reliable Software", Datamation, 10, 1972.

5.- Wegner, Peter, Editor, Research Directions in Software Technology, The MIT Press, Cambridge, Massachusetts, 1979.

COMPLEJIDAD

COMPLEJIDAD

Años atrás la mayor parte de los costos de una instalación de computación estaban concentrados en el equipo (Hardware). Actualmente dichos costos se estan desplazando hacia el area de programación (Software).

La medición de ciertas características del software que son necesarias para determinar los costos de Desarollo y Mantenimiento de Software son, por lo tanto, importantes desde el punto de vista económico.

Una de las características del software que es motivo de extensas investigaciones es la llamada complejidad.

La complejidad es la medida de los recursos que deben ser gastados por otros sistemas en su interacción con una pieza de software. Estos pueden ser máquinas, personas o también el medio ambiente externo. El enfoque de complejidad no solamente es sobre el software, sino también sobre las interacciones de este con otros sistemas [4].

Si el software interactúa con una máquina, la complejidad se medirá en base al tiempo de ejecución y espacio de memoria ocupado. Si el software interactúa con otro software, la medida se enfocara sobre el número de relaciones existentes entre ellos. Si la interacción es con personas, las medidas se basaran en el esfuerzo mental para comprender, mantener, cambiar y probar el software.

El medio ambiente actúa mas bien como un conjunto de restricciones, esto es, si un sistema debe ser transportado de un lugar a otro, hay ciertas limitaciones físicas y de tiempo que deben ser consideradas. Esto se refiere a la portabilidad de los sistemas, es decir que deben ser hechos de tal forma que puedan adaptarse de un medio ambiente a otro, sin que ellos implique aumento de la complejidad del sistema.

La complejidad del software puede ser enfocada desde varios puntos de vista:

COMPLEJIDAD PSICOLOGICA

Se refiere a las características que hacen que sea difícil de entender o producir software, por parte de seres humanos que interactúan con él.

COMPLEJIDAD COMPUTACIONAL O LOGICA

Se refiere basicamente a la característica operacional de algoritmos y procedimientos (siendo estos los elementos empleados para desarrollar software), las técnicas para expresarlos y los aspectos formales de su representación. También se refiere al uso de recursos del computador, esto es, los tiempos de ejecución de los algoritmos y los requerimientos de almacenamiento.

Se ha encontrado que la complejidad computacional es un buen antecedente para conocer el costo del programa y tiene también un potencial para predecir y explicar las relaciones con otras propiedades de los sistemas como esfuerzo, confiabilidad, flexibilidad y tiempo de ejecución [5].

Existen algunos métodos para medir complejidad, siendo uno de los mas conocidos el desarrollado por Thomas McCabe en 1976 [3], que se explica a continuación.

EL NUMERO CICLOMATICO V(G).-

Esta medida de la complejidad de un programa es desarrollada basandose en la teoría de gráficos. Se asume que la complejidad de un programa no depende de su tamaño (por ejemplo número de líneas), sino de su estructura de decisión y control.

El grado de complejidad se determina en base al número de caminos existentes en el gráfico del programa. Si el programa es completamente secuencial, el número de caminos será 1. Si el programa comprende bifurcaciones condicionales y/o incondicionales, el número de caminos se incrementará en el número en que estas ocurran, mas aun se dice que un programa con bifurcaciones hacia atrás, tiene potencialmente un número infinito de caminos [3]. En base a esto podemos decir que a medida que aumenta el número de caminos en el gráfico de un programa aumenta su complejidad lógica y también la psicológica, pues las contínuas bifurcaciones, dificultan la comprensión del programa.

Aunque es posible definir un conjunto de expresiones algebráicas para dar el número total de todos los posibles caminos a través de un programa (estructurado), esto es impracticable. Por tanto la medida de complejidad aquí desarrollada es definida en términos de los "caminos básicos" que al combinarse darán como resultado todos los posibles caminos del programa.

Podemos asociar un programa de computador (en realidad cada módulo de un programa) con un gráfico orientado que:

1.- Tiene un solo nodo de entrada y un solo nodo de salida.

2.- Cada nodo corresponde a un bloque de setencias sin transferencias de control.

3.- Cada rama corresponde a una transferencia de control.

Este gráfico es llamado gráfico de la estructura de control del programa.

Se asume que:

1.- Se puede llegar a cualquier nodo partiendo del nodo de entrada.

2.- Desde cualquier nodo se puede llegar al nodo de salida.

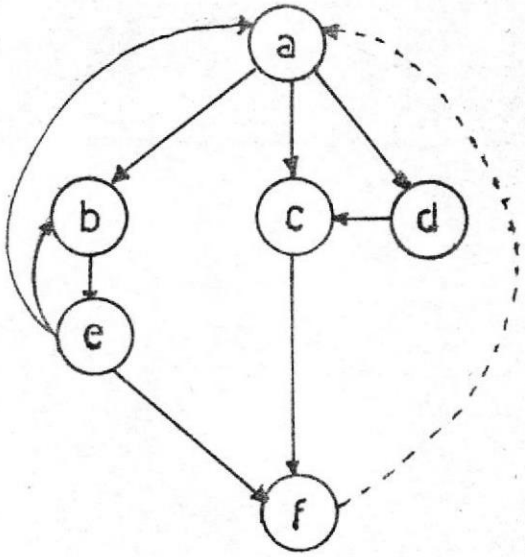
El número ciclomático V(G) de un gráfico de n nodos, e ramas y p componentes conectados es:

$$V(G) = e - n + 2p \qquad \text{Usualmente } p=1$$

En un gráfico completamente conectado G, el número ciclomático es igual el número de circuitos linealmente independientes.

Ejemplo:

G:



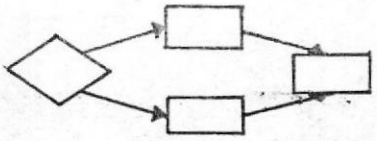
Este es un gráfico completamente conectado donde a es el nodo de entrada y f es el nodo de salida.

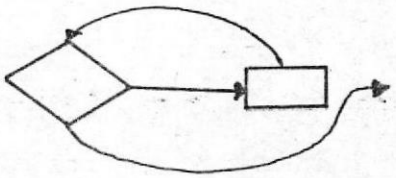
El máximo número de circuitos linealmente independientes es:

$$\begin{aligned} V(G) &= e - n + 2p \\ &= 9 - 6 + 2p \\ &= 5 \end{aligned}$$

Los gráficos de control de las construcciones usadas en programación estructurada, tienen las complejidades que se indican a continuación.

SECUENCIA  V(G) = 1 - 2 + 2 = 1

IF-THEN-ELSE  V(G) = 4 - 4 + 2 = 2

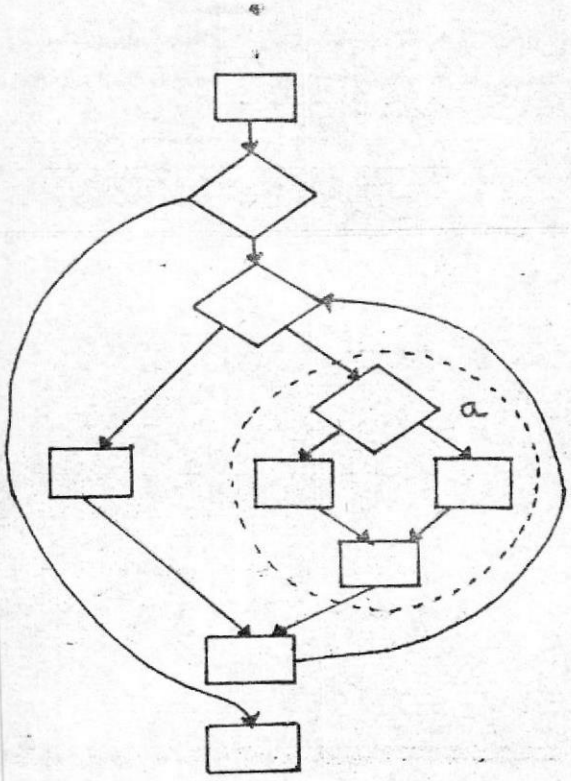
WHILE DO  V(G) = 3 - 3 + 2 = 2

REPEAT UNTIL  V(G) = 3 - 3 + 2 = 2

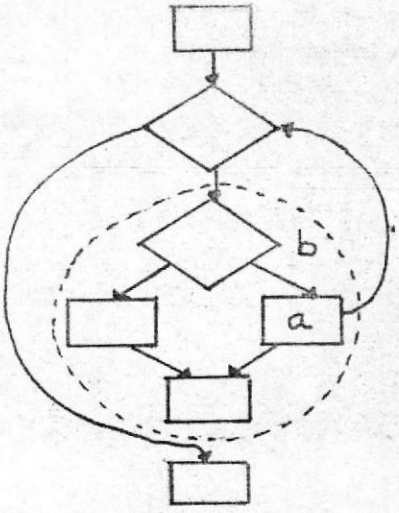
Un resultado importante de esta teoría es que un programa estructurado es reducible a un programa de complejidad 1.

La reducción es definida como el proceso de remover subgráficos (subrutinas o módulos) que tienen un solo nodo de entrada y un solo nodo de salida.

Ejemplo:

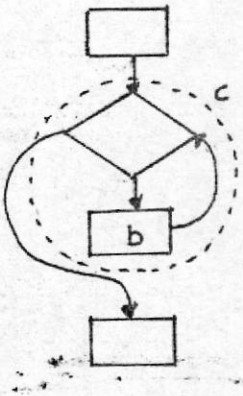


V(G) = 4



V(G) = 3

V(G) = 1



V(G) = 2

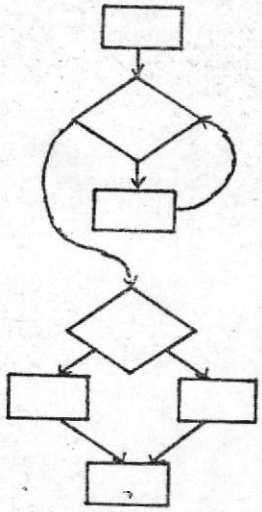
APLICACION DEL METODO DE McCABE PARA MEDIR LA COMPLEJIDAD

Un problema puede resolverse de varias formas y unas pueden ser más complejas que otras. Presentamos a continuación dos soluciones para un problema, cuya complejidad ha sido medida usando el número ciclomático.

Tenemos una tabla de diferentes valores A(1),...,A(M). Queremos saber si un valor x se encuentra en la tabla. Si al finalizar la búsqueda no lo encontramos, lo insertaremos como una entrada adicional en la tabla A. Además tenemos una tabla B donde B(I) será el número de veces que hemos buscado el valor A(I), entonces en caso que el valor x sea igual a un valor A(I), el B(I) respectivo se incrementará en 1 [2].

Primera solución:

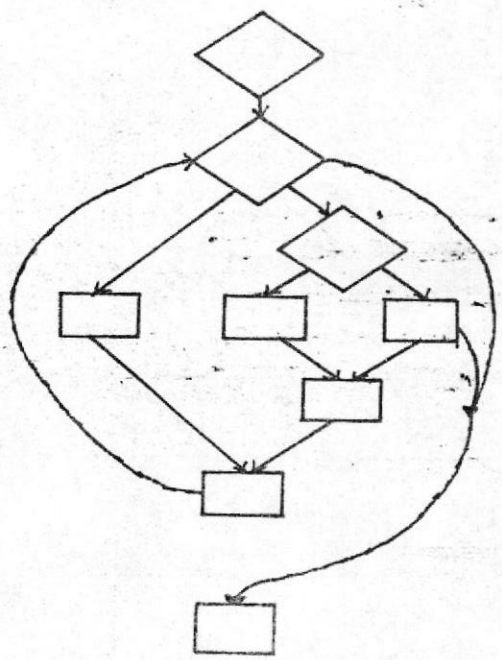
```
A(M+1):= X;
I:= 1;
WHILE A(I) <> X DO I:= I+1;
IF I > M
  THEN
    BEGIN
      M:= I;
      B(I):= 1
    END
  ELSE
    B(I):= B(I)+1
```



V(G) = 8 - 7 + 2 = 3

Segunda solución:

```
I:=M;
WHILE A(I) <> X DO
  BEGIN
    IF A(I) <> 0
      THEN I:= I-1
      ELSE
        IF I:= 0
          THEN I:= M;
          ELSE
            BEGIN
              A(I):= X
              B(I):= 0;
              GOTO FOUND;
            END
  END;
FOUND: B(I):= B(I)+1;
```



V(G) = 12 - 9 + 2 = 5

Hemos presentado dos soluciones alternativas al problema. En la primera usamos tan solo una sentencia de iteración WHILE y una condicional IF, mientras que en la segunda utilizamos una sentencia de iteración, 2 condicionales y una bifurcación incondicional (GOTO) fuera del lazo. Intuitivamente podemos apreciar, tan solo con ver el código del programa, que la segunda solución es más compleja que la primera. Además confirmamos esto al ver los gráficos de control de los programas, que nos muestran el número de caminos dentro del mismo, lo cual determina el grado de complejidad. Por último, usando la fórmula de McCabe, llegamos a obtener una comparación numérica, que establece la menor complejidad del primer programa.

El método de McCabe para medir la complejidad puede ser automatizado y de hecho lo ha sido. Una herramienta para investigar la complejidad de las estructuras de control, fue contruída en AFL, para un computador PDP-10, para analizar la estructura de programas FORTRAN. Esta herramienta llamada FLOW, tiene como entrada el código fuente de programas en FORTRAN. FLOW determina las distintas subrutinas de un programa y analiza la estructura de control de cada una. Para esto divide la subrutina en bloques, que son delimitados por sentencias que afectan el flujo de control: IF, GOTO referenciando a etiquetas, DO, etc.

Se uso FLOW para medir la complejidad de algunos programas FORTRAN y, luego de comparar numerosos gráficos de control y en base a los resultados obtenidos, se llegó a la conclusión de que un límite máximo razonable para la complejidad ciclomática de un programa es V(G)=10. Si la complejidad excede de 10, es aconsejable reconocer y modularizar subfunciones o rehacer el software. Se considera que V(G) en un rango entre 3 y 7 es una buena medida de complejidad.

La estrategia total sera: medir la complejidad de un programa, calculando el número ciclomático V(G), controlando el "tamaño" de los programas estableciendo un límite bajo para V(G) (en vez de controlar solo el tamaño físico del programa), y usando la complejidad ciclomática como una ayuda en la prueba de programas, pues el número de caminos probados puede ser comparado con la complejidad ciclomática, para descubrir de esta forma caminos adicionales que hubieren sido pasados por alto durante la etapa de prueba del software.

COMPLEJIDAD ESTRUCTURAL

Puede ser medida en términos de complejidad absoluta y complejidad relativa. La complejidad absoluta estructural es la medida del número de módulos, los cuales son definidos como bloques de programas que realizan una función específica, mientras que la complejidad relativa es el número de enlaces que existen entre los módulos.

No existen evidencias experimentales que soporten alguna medida de complejidad estructural, sin embargo Stevens [6] ha identificado y examinado los tipos de conecciones entre los módulos los cuales podrían ser considerados como medidas de este tipo de complejidad.

Las conecciones son referencias a etiquetas o direcciones definidas fuera de un módulo. La meta es minimizar las conecciones entre los módulos, evitando así caminos sobre los cuales errores pueden propagarse. La confiabilidad de los sistemas depende mucho de esto.

La medida de acoplamiento es la fuerza de asociación entre dos módulos. Por lo tanto, medir el acoplamiento es una forma de medir la complejidad. El acoplamiento depende de algunos factores: cuán complicada es la conección y si ésta se refiere al módulo mismo o a algo dentro de él.

Una medida cuantificable de la complejidad de sistemas es la del "medio ambiente común" la cual se refiere a la interfase con la misma área de almacenamiento, región de datos o dispositivos para dos o mas módulos. Un medio ambiente común de N elementos, compartido por M módulos, da como resultado N*M(M-1) caminos. Por ejemplo si un programa FORTRAN de tres módulos comparte un área común (COMMON) con 25 variables, tendrá 150 caminos, lo cual incrementa la complejidad del sistema [5].

La complejidad aumenta también mientras mas unidades sintácticas hayan en la sentencia de una conexión. Por ejemplo mientras mayor es el número de parámetros en una linea de llamadas de un módulo, mayor es la complejidad estructural del sistema. Estas características descritas por Stevens pueden ser usadas como un indicador de complejidad de software.

Existen tres conceptos que pueden ser adaptados para combatir la complejidad en el software:

Independencia.- Este concepto simplemente dice que para minimizar la complejidad se debe maximizar la independencia de cada componente del sistema.

Estructura Jerárquica.- Las jerarquías permiten estratificar el sistema en niveles de entendimiento. Cada nivel es un conjunto de las relaciones entre los niveles mas bajos. Las jerarquías permiten diseñar, describir y entender sistemas complejos.

Hacer conexiones aparentes.- Un problema básico en los sistemas es el gran número de efectos laterales desconocidos entre los componentes del mismo. Los efectos laterales son conexiones entre los módulos que no han sido previstas. Por ejemplo si en un programa un módulo A llama a un módulo B y este a su vez llama a un módulo C, algún cambio en el módulo A puede tener efectos en el módulo C que no son previstos durante el desarrollo del programa y se los llama conexiones aparentes.

Estos efectos hacen que los sistemas sean difíciles de entender. Para evitarlos es necesario comprender el sistema que se va a desarrollar así como también hacer un buen diseño del mismo.

CONCLUSIONES

La técnica normalmente usada para controlar la complejidad, consiste en dividir el proceso o estructura en pequeñas partes manejables y combinarlos para realizar una función específica.

Controlar la complejidad es una tarea importante en la programación de computadores. Se lo puede hacer especificando la función de cada módulo, en una manera independiente. Cada módulo debe producir una salida específica, de su correspondiente entrada. Una vez que la independencia es conseguida, es posible combinar los módulos del software en un gran sistema estructurado.

Un programa complejo tiene mayor probabilidad de contener errores que un programa simple y ya que la confiabilidad es la probabilidad de que el software se ejecute por un tiempo determinado sin error, este objetivo será difícil de alcanzar si se producen programas complejos.

En conclusión, programas complejos producen software inconfiable, pues contienen errores que en algunos casos son difíciles de detectar y corregir, toman mas tiempo de programación, y además dan como resultado un alto costo para el usuario.

REFERENCIAS

1.- Curtis, Bill, In Search of Software Complexity, Software Management Research Information System Programs, General Electric Company, Arlington, V.A. 22202.

2.- Knuth, Donald E., "Structured Programming with GOTO Statement", Computing Surveys, Volumen 6, Numero 4, Diciembre de 1974.

3.- McCabe, Thomas J., "A Complexity Measure", IEEE Transactions of Software Engineering, Volumen SE-2, Numero 4, Diciembre 1976.

4.- Myers, Glendford J., Software Reliability Principles & Practices, John Willey and Sons, New York, 1976.

5.- Shneiderman, Ben, Software Psychology. Human Factors in Computer and Information Systems, Winthrop Publishers Inc., Cambridge Massachusetts, 1980.

6.- Stevens, W. P., G. J. Myers, L. L. Constantine, "Structured Design", IBM System Journal, 1974.

7.- Yourdon, Edward, Techniques of Program Structure and Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

BIBLIOGRAFIA

BIBLIOGRAFIA

Basseli, Victor R., Quantitative Software Complexity Models: A Pannel Sumary, Workshop on Quantitative Software Models, Concord Hotel-Kiamesha Lake, N. Y., Octubre 9-11, 1979, IEEE Catalogo # TH0067-9.

Bowen, John B., Bughes Fullerton, "A Survey of Standards and Proposed Metrics for Software Quality Testing", Computer, Volumen 12, Numero 8, Agosto 1979.

Curtis, Bill, In Search of Software Complexity, Software Management Research Information Systems Programs, General Electric Company, Arlington, V. A. 22202.

Dijkstra E. W., "Goto Statement Considered Harmful", Comunications of the ACM, 1968.

Freeman, Peter, Anthony I. Wasserman, Editores, Tutorial on Software Design Techniques. Second Edition, IEEE Computer Society, New Jersey, 1977.

Knut, Donald E., "Structured Programming with GOTO Statements", Computing Survey, Volumen 6, Numero 4, Diciembre 1974.

Kopetz, K., Software Reliability, Springer-Verlag, New York, 1979.

McCabe, Thomas J., "A Complexity Measure", IEEE Transactions of Software Engineering, Volumen SE-2, Numero 4, Diciembre 1976.

Myers, Glendford J., Software Reliability: Principles & Practices, John Willey and Sons, New York, 1976.

Perlis, Alan J., Frederick G. Sayward, Mary Shaw, Editores, Draft Software Metrics Panel Final Report: Papers presented at the 30 June 1980, Meeting on Software Metrics, Washington D. C., Yale University, Department of Computer Science, Research Report, 182/80, Junio 1980.