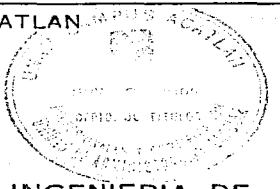
25
2ej.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ENEP ACATLAN



REUTILIZACION E INGENIERIA DE SOFTWARE ORIENTADA A OBJETOS

T E S I S

QUE PARA OBTENER EL TITULO DE

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A :

GOMEZ-EGUIARTE MARTINEZ ILLAN

ASESOR DE TESIS: DRA. HANNA OKTABA

MEXICO, D. F.. NOVIEMBRE, 1997

CAPÍTULO I : REUTILIZACIÓN.

CAPÍTULO II :MODELO ORIENTADO A OBJETOS Y REUTILIZACIÓN.

CAPÍTULO III : INGENIERÍA DE SOFTWARE Y REUTILIZACIÓN.

CAPÍTULO IV : IMPLEMENTANDO LA REUTILIZACIÓN EN C++.

INTRODUCCIÓN.

Durante la vida profesional el ingeniero de sistemas ha tenido, seguramente, que hacer frente a la tarea del análisis, diseño y construcción de productos de software. Debido a ello habrá percibido la necesidad, en mayor o en menor grado, de aprovechar o de obtener mayor ventaja del trabajo que alguna vez ha sido desarrollado. En la búsqueda de respuesta a la interrogante de ¿cómo lograr obtener mayor beneficio del esfuerzo y los recursos alguna vez asignados a un proyecto?, se ha incursionado en el campo de la reutilización que, aunque obviamente no es exclusivo de la ingeniería de software, ha sido abordado con creciente interés por ésta.

Partiendo de que existe una necesidad que demanda ser satisfecha, la anteriormente citada, el trabajo propone como solución la aceptación del paradigma de la ingeniería de software orientada a objetos.

Por otro lado se ha considerado que dentro del conjunto de lenguajes, algunos únicamente basados y otros totalmente orientados a objetos, que han surgido ante la aceptación de este paradigma existe un elemento en particular que por sus características intrínsecas permite llevar a la práctica de la programación los elementos del enfoque orientado a objetos y reutilización, dicho elemento es C++.

El trabajo pretende mostrar que: "*A través del paradigma orientado a objetos puede mejorarse la reutilización, y que los conceptos de éste paradigma pueden llevarse a la práctica de la programación apropiadamente con el lenguaje C++*".

En el capítulo primero se aborda el tema de la reutilización. En él se explica el concepto, se comentan los diversos enfoques que se le ha dado, se comentan las formas existentes de reutilización, sus tendencias y el estado actual que guarda.

En el capítulo segundo se inicia la presentación del modelo orientado a objetos, enumerando sus componentes, su evolución y sus características ; todo ello con el afán de mostrar cuales son los beneficios que ofrece, a la vez que se inducen como respuesta al problema de la reutilización y se incluye un experimento[1].

En el capítulo tercero se profundiza en la ingeniería de software orientada a objetos y se explican los aspectos de ella que involucran reutilización. Lo anterior se basa en su mayoría en las ideas de Ivar Jacobson, James Martin, Grady Booch y Edward Yourdon.

Por último el capítulo cuarto se concentra en mostrar que casi la totalidad de los conceptos manejados por el modelo orientado a objetos pueden aplicarse y aprovecharse para la reutilización haciendo uso del lenguaje de programación C++. Este punto de vista también esta reforzado principalmente por los trabajos del Dr. Miguel Katrib Mora.

[1] Lewis, John A. y Henry, Sally M. "On The Relationship Between the Object Oriented Paradigm and Software Reuse : An Empirical Investigation." Journal of Object Oriented Programming Vol. 5(4), Julio/Agosto 1992

CAPÍTULO I

REUTILIZACIÓN.

"La motivación en que se basa la creación de artefactos reutilizables, en lugar de transitorios, es intelectual y económica; pero además, se basa en el deseo del hombre por la inmortalidad, lo cual es una de las características que distinguen al hombre del animal."

Peter Wegner

INTRODUCCIÓN.

La necesidad de obtener mayores beneficios de los proyectos llevados a cabo nos impulsa hacia la búsqueda de mejores alternativas de reutilización de componentes. De la misma manera que en disciplinas tales como la electrónica o en industrias como la automovilística se emplea el ensamble y la reutilización, existe un movimiento tanto en Estados Unidos como en Japón (principalmente) mediante el cual se busca hacer parte a la reutilización de todas las fases de la ingeniería de software.

Así el contenido de este capítulo proporcionará algunas ideas sobre lo que es la reutilización en el campo de la ingeniería de software, sobre las diversas formas que ha adoptado al paso de los enfoques metodológicos, sobre las formas existentes, los obstáculos que enfrenta su desarrollo, los factores que la motivan y sus perspectivas.

I.1. PRELIMINARES SOBRE REUTILIZACIÓN.

I.1.1 TECNOLOGÍA DE SOFTWARE DE CAPITAL INTENSO.

En este trabajo, entenderemos por reutilización el aplicar una gran variedad de conocimientos acerca de un sistema a otro similar para reducir el esfuerzo de desarrollo y mantenimiento del nuevo sistema. Lo anterior incluye : el conocimiento del problema (conocimiento del medio), experiencia de desarrollo, decisiones de diseño, requerimientos, código, documentación, etc.

Reusabilidad : Es la capacidad con que cuentan los productos de software para ser reutilizados, en forma completa o en partes, para nuevas aplicaciones [Biggerstaff,1987].

La importancia de la reutilización es obvia al momento de explotar aspectos comunes y evitar reinventar soluciones a problemas que han sido superados con anterioridad, es importante mencionar que, este aspecto de la calidad influye sobre otros aspectos de la calidad de software.

Reutilizar quiere decir hacer uso de conceptos u objetos previamente adquiridos, es una medida de que tan fácilmente se puede llevar a cabo una tarea. Esto conlleva un proceso de comparar necesidades o situaciones actuales con necesidades o situaciones anteriores, en caso de encontrar semejanza por medio de la abstracción dicha reutilización se vuelve importante.

Existen dos niveles de reutilización :

1.- Reutilización de ideas o conocimientos.
2.- Reutilización de artefactos o componentes.

Considerando que en la práctica es difícil que un componente complejo cubra los requerimientos de manera cabal, aparece en escena de manera inevitable la adaptación y en el momento de buscar componentes para adaptarse, la manera en que éstos estén organizados cobra importancia capital.

Es por lo anterior que se considera importante el artículo de Prieto Díaz [Prieto,1985], ya que proporciona un medio ambiente que ayuda a localizar componentes y estima el esfuerzo de adaptación y conversión necesario para la reutilización.

Actualmente todos los desarrolladores de software hemos vivido el gran "cuello de botella" representado por el corregir, dar mantenimiento y desarrollar para cubrir necesidades particulares que enfrenta el desarrollo de software. Aunado a ésto, se espera que muchas compañías aumenten la inversión de capital en desarrollo de sistemas, por lo que, el panorama del desarrollo se volverá mucho más problemático.

El capital incluye los recursos humanos y sus elementos no materiales tales como : habilidades (destrezas y educación), terrenos, construcciones, equipos de todas las clases y todo aquello con lo que pueda contar una empresa.

El hecho de que haya una gran semejanza entre la tecnología industrial y la tecnología de software ha permitido que muchos términos de la primera se utilicen en la segunda, por ejemplo, ingeniería de software, herramientas de software y fábricas de software.

De la misma manera que la tecnología que impulso la revolución industrial se basó en la fuerza de trabajo en sus inicios y después en su madurez se volvió una tecnología de capital-intenso, así le sucede a la industria del desarrollo de software [Wegner,1984].

Entendemos por una tecnología de capital intenso aquella que requiere de herramientas costosas. La tecnología de desarrollo de software necesita cada vez más de herramientas poderosas cuyo costo es alto y requiere de mayor inversión.

Estas herramientas de que hablamos son reutilizables y su costo se va amortizando a medida que se utilizan. Dentro del campo del desarrollo de software contamos con herramientas altamente reutilizables como : compiladores y sistemas operativos que nos ayudan al desarrollo de programas y los programadores que también representan recursos reutilizables para el desarrollo de programas. Las actividades que contribuyen a incrementar las capacidades de los programadores, tales como la educación, incrementan el capital destinado al desarrollo de software ya que mejoran la reutilización del personal.

La actividad relacionada con el desarrollo de elementos de capital es llamada formación de capital y es dependiente de los modelos y conceptos de implementación, así como de las máquinas donde se desarrolla el software.

La reutilización es un principio de ingeniería cuya importancia deriva del evitar la duplicidad al tiempo que procura capturar las características comunes en clases de tareas similares. Esta situación tiene justificación intelectual y económica en el desarrollo de productos de software reutilizable ya que unifica y simplifica fenómenos similares y hace a los programadores más productivos. Como veremos en nuestro siguiente capítulo esta idea esta muy relacionada con los conceptos de clases, herencia y en general con el modelo orientado a objetos.

Anteriormente el costo del equipo era más importante dentro del costo de un sistema computarizado, por lo tanto, la atención se centraba en la eficiencia del equipo. De este modo, los sistemas operativos de tiempo compartido fueron creados de tal manera que un procesador poderoso fuera reutilizado por muchos usuarios. Con el decremento de los costos de hardware la atención se ha concentrado en la reutilización de software y en el uso productivo del personal.

Por otro lado se tiene la presión que impone el vertiginoso cambio tecnológico y el deseo de que los costos de inversión sean recuperados de manera pronta. Actualmente los costos en la mayoría de las áreas de sistemas están constituidos por el mejoramiento y mantenimiento.

La importancia con que cuentan las bibliotecas de programas reutilizables fue reconocida por Wilkes, Wheeler y Gill[1] a principios de la década de 1950; por este tiempo la idea de utilizar seudocódigo surgió al reconocer que el uso de subrutinas de indexación y de punto flotante simplificaban el proceso de programación, ya que un programa consumía la mayor parte del tiempo en las rutinas de indexación y de punto flotante ¿por que no simplificar la programación proporcionando un nuevo código de instrucciones más fácil de usar que el propio conjunto de instrucciones de la maquina?, cabe mencionarse que los autores no dieron gran importancia al hallazgo que habían logrado ya que en caso contrario no lo habrían sepultado en el apéndice "D" de su libro "The preparation of programs for an electronic digital computer"; más tarde vino la idea de que tales bibliotecas fueran tan accesibles como las piezas de ensamblaje de la industria automotriz [Biggerstaff-Perlis,1989].

El desarrollo en los métodos de abstracción de datos y del paradigma de la programación orientada a objetos en las decadas de 1970 y 1980 motivaron el deseo de desarrollar bibliotecas de componentes de software que fueran realmente viables.

I.1.2 DEFINICIÓN DEL PROBLEMA.

Debido al gran interés que se ha despertado desde los años ochenta por el tema de la reutilización se considera necesario hablar acerca de lo que significa y de los hechos que despertaron el interés por su estudio.

El factor principal que ha motivado su desarrollo es la inminente crisis en que ha caído la industria del desarrollo de productos de software, la cual comprende a la industria privada tanto como al gobierno, los síntomas de esta crisis han sido diversos, pero, entre los más comunes podemos contar los siguientes:

- Largos periodos de tiempo dedicados para decidir sobre varias categorías de productos de software.
- Gran cantidad de tiempo dedicada al mantenimiento del software.
- Problemática creciente dentro de los equipos de programación para productos grandes.
- Complejidad del código fuente.
- Lenguajes y practicas de programación tradicionales.
- Dificultad y falta de un método que permita trabajar con la complejidad inherente a los sistemas de software.

Estos factores han elevado los costos de mantenimiento, a la vez que han motivado a que los diseños de los programas definan globalmente o pasen de procedimiento en procedimiento las estructuras de datos importantes del programa, lo que ha acarreado serios problemas cuando

[1] Wilkes,Wheeler & Gill. "The Preparation of programs for an electronic Digital Computer." 1951

se modifica alguna de estas estructuras, ocasionando que se desheche mucho código del anteriormente escrito.

¿Como se ha llegado a esta crisis? La respuesta merece un estudio profundo que no haremos aquí, sin embargo, hablaremos de algunos de los elementos que acarrea consigo el desarrollo de un producto de software, estos temas pertenecen a todos los desarrollos y no son exclusivos de ninguno en particular.

- Complejidad del dominio del problema.
- Dificultad para manejar el proceso de desarrollo.
- La flexibilidad con que debe contar el producto desarrollado.
- La dificultad de caracterizar el comportamiento de un sistema discreto.

Acerca de los cuales trataremos a continuación.

COMPLEJIDAD DEL DOMINIO DEL PROBLEMA.

Generalmente los problemas que se trata de resolver dentro del desarrollo de productos de software deben su complejidad a una amplia gama de requerimientos en competencia, a veces contradictorios, entre los cuales tenemos los requerimientos de comportamiento o funcionales y otros requerimientos no funcionales tales como utilización, ejecución, costo, capacidad de sobrevivencia, confiabilidad, etc.

Es esta complejidad externa la que causa la arbitraria complejidad durante el desarrollo de los sistemas.

Usualmente esta complejidad surge de la falta de entendimiento entre los usuarios y los desarrolladores del sistema. Los primeros generalmente encuentran difícil expresar con precisión sus necesidades de manera que los desarrolladores puedan comprenderlo y en los peores casos los usuarios mismos apenas tienen ideas vagas de lo que quieren dentro del producto de software. Por otro lado, los desarrolladores no cuentan con un método que les permita capturar totalmente las peticiones y requerimientos de los usuarios.

No se puede culpar a ninguna de las partes por este problema, lo que sí se puede hacer es idear mecanismos bajo los cuales sea posible que los usuarios tanto como los desarrolladores conozcan más acerca del dominio de conocimiento del otro para que los enfoques acerca de la naturaleza del problema y las consideraciones que se hacen de la naturaleza de la solución no difieran unos de otras.

Otro de los factores que complica la situación es que los requerimientos de un producto de software cambian frecuentemente durante el proceso de desarrollo, debido a que el periodo de desarrollo en ocasiones es muy extenso y cambian las reglas del problema. Los gerentes y usuarios podrían establecer un congelamiento artificial de requerimientos en un momento dado, pero, los verdaderos requerimientos, y las necesidades del sistema continuarían evolucionando; [Fischer, 1989] señala: ***"Tenemos que aceptar los cambios de requerimientos como un hecho de la vida, y no condenarlo como un producto del pensamiento desordenado"***. A este respecto se sugiere el desarrollo de prototipos para permitir a los usuarios entender y articular mejor sus necesidades, al mismo tiempo que los

desarrolladores conocen más a fondo el dominio del problema y son capaces de hacer mejores preguntas para entender el comportamiento deseado por los usuarios.

DIFICULTAD PARA MANEJAR EL PROCESO DE DESARROLLO.

La tarea principal del equipo desarrollador de software es crear la ilusión de simplicidad dentro del producto, protegiendo al usuario de la complejidad externa.

El volumen de requerimientos de un sistema a veces nos obliga a escribir gran cantidad de código nuevo o a modificar software existente. Actualmente debido a la gran potencia que se ha ganado con los lenguajes de alto nivel (el tamaño de los productos se mide en cientos de miles o aún millones de líneas de código fuente), debido a que la demanda de trabajo es tal que se requiere del uso de crecientes equipos de desarrollo (idealmente debemos preferir un equipo tan pequeño como sea posible, ya que el tener más desarrolladores significa complejidad en la comunicación) y debido a la dificultad para coordinarse (particularmente si el equipo está disperso geográficamente) es que el manejo del proceso de desarrollo se ha vuelto cada vez más penoso. La clave para salir airoso de estos inconvenientes es : mantener, siempre, una unidad e integridad de diseño.

LA FLEXIBILIDAD CON QUE DEBE CONTAR EL PRODUCTO DESARROLLADO.

El trabajar con software ofrece la mayor flexibilidad, por lo tanto, para el desarrollador es posible expresar casi cualquier clase de abstracción. Esta flexibilidad debe convertirse en una propiedad que permita que el desarrollador cree los ladrillos sobre los cuales descansarán las abstracciones de más alto nivel. Debido a que no existen estándares para la construcción de tales ladrillos básicos, el desarrollo de software ha permanecido como una intensa labor de negocios.

LA DIFICULTAD DE CARACTERIZAR UN PROBLEMA DISCRETO.

Dentro de una aplicación contamos cientos o miles de variables y seguramente más de un flujo de control, esta colección de variables, sus valores reales, las direcciones actuales y las direcciones de pila dentro del sistema constituyen un estado de la aplicación. Más aún, debido a que ejecutamos nuestro producto en una computadora digital estamos tratando con un sistema de estados discreto, este tipo de sistemas aunque es cierto que tiene un número finito de estados, aunque, en un sistema grande este número es muy grande. A decir verdad, aunque tratemos de desarrollar sistemas de modo que el comportamiento de una parte no afecte al comportamiento en otra parte, el hecho que prevalece es que la transición entre estados discretos no puede ser modelada por funciones continuas. Además, el mapeo de estado a estado no siempre es determinístico, tal vez un evento externo pueda corromper el estado del sistema.

En los sistemas discretos todos los eventos externos pueden afectar alguna parte del estado interno del sistema, así que, dado que no tenemos las herramientas matemáticas ni la capacidad intelectual para modelar el comportamiento completo de grandes sistemas discretos lo único que nos queda es contentarnos con niveles aceptables de confiabilidad. Lo anterior no debe preocuparnos sobremanera ya que para consuelo nuestro el estudio de los sistemas ha arrojado luz sobre las características comunes que presentan todos los sistemas.

De [Biggerstaff-Perlis,1989], sabemos[2] que actualmente el desarrollo de los productos exige la satisfacción de problemas más complejos que antes, al mismo tiempo, demanda personal calificado para éste. Sin embargo, nuestros métodos y herramientas para desarrollar software no han desarrollado nuestra habilidad en el desarrollo de software, por ejemplo, se estima que en Estados Unidos la productividad en este ramo se ha incrementado solo en 3.8 % anual los últimos 20 años, mientras que la capacidad de procesamiento instalada se ha incrementado en 40 % anualmente. Se han hecho estudios que demuestran que generalmente el desarrollo de un sistema acarrea la reprogramación de cerca del 60% de software anteriormente programado, ante este hecho, naturalmente nos viene a la mente la idea de la reutilización.

I.1.3 EL PROCESO DE REUTILIZACIÓN.

Para llegar al empleo de un módulo reutilizable necesario para nuestro desarrollo y que satisfaga los requerimientos impuestos, es menester seguir el siguiente procedimiento.

1) Dada una especificación funcional, el desarrollador buscará una biblioteca de componentes, donde estén aquellos candidatos que satisfagan su especificación.

2) En caso de que un componente satisfaga todos y cada uno de los requerimientos, el problema esta terminado.

3) En el caso de que existan varios componentes que satisfagan todos los requerimientos, toca al desarrollador el decidir el más apropiado analizando otras características que diferencien a los componentes. Ahora en el caso de que encuentre un conjunto de elementos dentro del cual ninguno satisfaga todos los requerimientos (que es el caso más común) lo que sigue es la evaluación de cual es el componente más fácil de adaptar para que satisfaga todos los requerimientos.

I.1.4 LA INMENSA ESCALA DEL PROBLEMA DE REUTILIZACIÓN.

El porque aquí consideraremos la reutilización desde un punto de vista amplio se debe a que el no hacerlo así, ha constituido un impedimento para obtener grandes beneficios de ella.

La mayoría de los enfoques restringidos que se ha dado a la reutilización están relacionados con la reaplicación de componentes de código, lo cual es muy bueno como un primer paso. Al hablar de reutilización de código, debemos tener presente el alto grado de especificidad al que nos inducen los lenguajes de alto nivel, al mismo tiempo, debemos considerar que los componentes reutilizables tienen la característica de que mientras más grandes son se vuelven más específicos de modo que la probabilidad de ser reutilizados disminuye y viceversa al ser más pequeños y por lo tanto más generales el grado de reutilización se eleva.

Al respecto también debemos pensar que si construimos muchos componentes pequeños y generales al final nos encontraremos con que el esfuerzo necesario para componer a partir de ellos un sistema representa un gran trabajo y en consecuencia la construcción de esta superestructura podría representar un costo mayor que el de haber desarrollado el sistema sin intención de hacer reutilización.

[2] Los autores mencionan como fuente el trabajo de Fairley, R. Bollinger,T & Pfleeger, S.L., "Final Report: Incentives Reuse of Ada Components." Vols. 1-5, George Mason University 1989.

Como ya comentamos, el que un componente reutilizable crezca lleva consigo una especialización, es más, es ésta la que lo obliga a crecer, y por lo tanto, a volverse cada vez más específico y a que se reduzca la probabilidad de que sea reutilizado (debido a que será más difícil que se presenten problemas con tan alto grado de similitud). Otra desventaja de crear módulos reutilizables grandes es que en el caso de que sean usados por otros usuarios y estos tengan necesidad de hacerles adaptaciones, se presenta el problema de que al ser tan grandes es casi seguro que sean de complejidad considerable y que el entenderlos resulte tan costoso en tiempo como el desarrollo del componente.

Nosotros hablaremos de un concepto amplio de la reutilización en el cual incluiremos el conocimiento del medio ambiente, la experiencia de desarrollo, las decisiones de diseño, los requerimientos, el código, la documentación, etc. Acerca de este punto de vista podemos comentar que hasta hace algunos años no se había demostrado mucho interés por el tema, pero, a raíz de la fuerte competencia que se ha dado en el mercado de software la reutilización ha empezado a producir resultados satisfactorios.

Durante algún tiempo el tema de la reutilización estuvo totalmente ausente, sólo se presentaba en la mente de los desarrolladores cuya experiencia les permitía hacer uso de conocimientos y experiencias personales, más tarde, la reutilización cobro forma en los conjuntos de bibliotecas compartidas, en el caso de sistemas bajo el paradigma orientado a objetos es frecuente la reutilización de componentes (en este caso objetos) de un sistema en otro similar.

Durante el mantenimiento la reutilización está siempre presente ya que el ingeniero de mantenimiento está continuamente reutilizando la infraestructura asociada con el sistema.

I.1.5 CLASIFICACIÓN DE LA REUTILIZACIÓN DESDE EL PUNTO DE VISTA TECNOLÓGICO.

Desde el punto de vista tecnológico, la reutilización puede dividirse en dos grupos fundamentales dependiendo de los componentes a ser reutilizados. Éstos pueden ser, ya sea bloques construidos o bien patrones que generen módulos reutilizables los cuales son llamados respectivamente sistemas basados en composición y sistemas basados en generación. Fig 1.

Dentro de las tecnologías de composición los elementos a utilizar son independientes e idealmente no se cambian para poder reutilizarse, aunque, en la práctica los componentes pueden ser modificados para cumplir mejor los requerimientos computacionales del usuario. En el caso ideal, son elementos pasivos compuestos por un agente externo, ejemplos de ellos son las estructuras de código tales como : subrutinas, funciones, programas y objetos del estilo de Smalltalk.

Características	Enfoques de reutilización				
Componente reutilizado	Bloques de construcción		Patrones		
Naturaleza del componente	Atómicos e inmutablemente pasivos		Difusos y transformables		
Principio de reutilización	Composición		Generación		
Enfasis	Bibliotecas de componentes	Organización y principios de composición	Lenguajes basados en generadores	Generadores de aplicación	Sistemas de transformación
Sistemas típicos	-Bibliotecas de subrutinas	-Orientación a objetos -Archivos PIPE	-VHLLs -POLs	-CRT FMTRS -FILE MGMT	Transformadores de lenguajes

FIG. 1 ENFOQUES DE REUTILIZACIÓN: BLOQUES DE CONSTRUCCIÓN Y PATRONES.

El crear programas por entre la reutilización de bloques construidos es una forma de composición, unos cuantos principios de composición bien definidos han sido aplicados a los componentes, por ejemplo el mecanismo de redireccionamiento de archivos de UNIX es un buen ejemplo, ya que, con él se construyen programas más complejos a través de otros más simples al utilizar la salida de un programa como entrada de otro programa. Otro ejemplo es Smalltalk, en el cual los dos principios de composición son el paso de mensajes y la herencia, el primero es una generalización de las llamadas a funciones (un eslabón estático entre quien llama y quien es llamado), el último, permite la determinación dinámica del método a ser invocado.

Las tecnologías de generación no son tan fáciles de caracterizar como las de composición, ya que, los componentes a reutilizar no son concretos, ni son entidades autocontenidas. En el caso de la composición podemos señalar un bloque antes y después de su utilización, es decir es inmutable durante su utilización, sin embargo, en el grupo de generación los componentes reutilizables son patrones forjados dentro de un programa generador de patrones donde las estructuras resultantes frecuentemente presentan sólo algunas relaciones distantes con los patrones de los programas que los generan, además, cada ejemplo de un patrón puede ser muy independiente de otro generado con el mismo código. En este sentido la reutilización es menos una composición (de componentes) que una ejecución de generadores de componentes.

Los patrones reutilizables toman dos formas diferentes: patrones de código y patrones dentro de reglas de transformación, los primeros son usados en la generación de aplicaciones donde los patrones reutilizables de código existen dentro del generador mismo, por otra parte, los sistemas de transformación usan a los patrones reutilizables de código. En ambos casos los efectos de los componentes reutilizables individuales dentro del programa obtenido tienden a ser más difusos que los efectos de los bloques construidos.

No es fácil caracterizar el principio mediante el cual los patrones son reutilizados, excepto que es una reactivación de patrones. En este trabajo nos ocuparemos únicamente de los sistemas basados en composición y dejaremos a un lado los sistemas basados en generación.

I.2 ENFOQUES PARA LA REUTILIZACIÓN.

La capacidad que pueda tener un producto de software de que su código pueda ser reutilizado en otros proyectos es uno de los aspectos más importantes de la calidad del producto, si bien durante muchos años ha sido descuidado, hoy estamos conscientes de su gran importancia y por ello es que posteriormente hablaremos del enfoque orientado a objetos, por que pensamos que éste representa un enfoque poderoso para la reutilización.

I.2.1 CÓDIGO REUTILIZABLE.

La noción de código reutilizable fue tratada hace cerca de 30 años por D. Mellroy [Mellroy,1969] quien habla de lo que el llama una fábrica de componentes, la cual funcionaría de la siguiente manera : primero identificaría las técnicas para crear una familia de rutinas parametrizadas para algún propósito común, después, se preocuparía de la distribución de las rutinas, su catalogamiento y la mezcla de los productos de software ofrecidos.

Sobre estas ideas se despertó una gran respuesta por parte de la comunidad computacional cuyos comentarios se concentraron en tres aspectos fundamentalmente.

a) Es imposible crear un sistema parametrizado eficiente, confiable y conveniente para todos los sistemas y donde la naturaleza del paquete no se imponga sobre el usuario.
b) Nunca se podrá reducir todos los problemas de la dependencia de la máquina a representaciones tales como las representaciones numéricas fijas.
c) No se sabe como podría crearse un conjunto de catálogos de descriptores que permitiera a los usuarios buscar la rutina apropiada.

Para que el concepto de código reutilizable se vuelva realidad es necesario que los siguientes problemas se resuelvan.

a) Se encuentre un mecanismo para identificar componentes.-. El problema a resolver es ¿cómo determinar que componentes son utilizables o adaptables a muchos proyectos?, ya que el esfuerzo para identificar tales componentes sin restringirse hacia un dominio específico parece difícil de hallar.
b) Encontrar un método para especificar componentes.- Una vez que se decide lo que un componente funcional llevara a cabo, es necesario preguntarse de que manera podría construirse su descripción para que otros la entendieran.
c) Definir de que manera serán implementados los componentes.- ¿Se implementaran en algún lenguaje de alto nivel o por medio de algún lenguaje de diseño que permita mezclar construcciones de programación con lenguaje natural?.
d) Encontrar un método para catalogar a los componentes.- Debido a que podrían ser muchos los componentes creados será necesario crear un lenguaje metadescriptor para agrupar los componentes similares.

Es necesario anotar que si el trabajo necesario para acoplar componentes es muy pesado tal vez no se obtenga mucho éxito de la reutilización de código, además, como se verá

posteriormente la actividad dedicada a la programación en la mayoría de los casos es menor que la dedicada a las actividades de diseño, de donde se puede concluir que un enfoque parcial dirigido únicamente al código no sería de tanto valor. Por lo tanto, será necesario extender nuestro interés en la reutilización hasta el límite del diseño.

Por otro lado debemos considerar que la reutilización de código cobra mayor importancia cuando se piensa en los costos de mantenimiento de software. Para empezar diremos que existen ciertas consideraciones que debemos hacer para estar de acuerdo sobre lo que consideramos un componente de software reutilizable:

1.- Puede usarse en varias aplicaciones.
2.- Puede usarse en versiones sucesivas de un mismo programa.
3.- Son reutilizados cada vez que durante la ejecución del programa sean invocados.
4.- Son reutilizados en cualquier lugar que exista un programa que los contenga.

I.2.2 BIBLIOTECAS DE COMPONENTES REUTILIZABLES.

Al pensar en las consideraciones anteriores nos preguntamos cuáles son las características que presenta algún sistema reutilizable que funcione, actualmente, todos hemos podido observar que el mayor éxito dado en la reutilización de módulos sucedió con las bibliotecas matemáticas las cuales presentan las siguiente características a saber :

a) El dominio del problema es muy pequeño si consideramos la variedad de tipos de datos involucrados.
b) El dominio del problema es bien entendido, ya que cuenta con una estructura matemática evolucionada a través de cientos de años.
c) La tecnología que lo soporta es completamente estática, es decir, crece y evoluciona lentamente, de manera que las partes existentes de la tecnología permanecen inalterables o casi inalterables.

El que el dominio del problema sea muy pequeño en cuanto a tipos de datos involucrados permite que la reutilización de código sea más manejable, por lo mismo, la posibilidad de reutilización de los módulos es más probable con lo que el costo del proyecto es más rentable cuando se acude a la reutilización. Como el dominio del problema es estático las bibliotecas son estáticas.

Otra característica que beneficia mucho a este problema es que es tan conocido que la mayoría de las personas pueden aprender a utilizar los módulos sin más que una breve explicación.

De lo anterior se infiere que el peor ambiente para emplear la reutilización es aquel donde los fundamentos cambian rápidamente. Lo anterior no quiere decir que cuando el código no sea reutilizable, las ideas del diseño tampoco lo sean. El tratar de reutilizar un diseño presenta un problema importante referente a la forma en que se capturaría la información del diseño; la representación de código en esquemas es sencilla y actualmente existen muchos métodos de llevarla a cabo, pero, para los diseñadores no existe ninguna herramienta que funcione satisfactoriamente, por ejemplo, los lenguajes de diseño que existen y son procesables por la computadora son muy semejantes a los lenguajes de programación comunes y por lo tanto muy específicos, de todas formas las diversas investigaciones han concluido que el beneficio

potencial para la reutilización del diseño es muy alto y que es éste el camino que podría llevarnos hacia el incremento de la productividad y de la calidad.

I.2.3 REPETICIÓN DENTRO DE LA PROGRAMACIÓN.

Cualquier persona que observe un desarrollo de software queda impresionada por su naturaleza repetitiva, una y otra vez, los programadores construyen un número de patrones básicos tales como : ordenamientos, búsquedas, lecturas, escrituras, comparaciones etc., ahora, cualquier programador experimentado puede haber llegado a conocer ese sentimiento de que se están reprogramando muchas rutinas que podrían ser únicas.

Una manera de corroborar esta situación es preguntar a varios programadores, por ejemplo : ¿Cuántas veces durante los últimos seis meses, usted o sus compañeros han escrito un fragmento de código para hacer una búsqueda en una tabla?

Tal vez la respuesta sea una o mas, pero lo importante aquí, es que la mayoría lo han hecho en el nivel más bajo de abstracción en lugar de utilizar una rutina existente, la búsqueda en tablas es una de las áreas de la computación más ampliamente desarrolladas, existen libros excelentes donde se presentan los algoritmos fundamentales y pareceria que después de esto nadie debería necesitar escribir código acerco de ello, de la misma manera que los ingenieros no diseñan inversores estándar sino que los compran, pero el hecho es que se sigue creando rutinas de esta naturaleza.

I.2.4 LA REUTILIZACIÓN DE CÓDIGO FUENTE EN EL MEDIO AMBIENTE ACADÉMICO

.Mucho del actual desarrollo de la cultura UNIX se ha extendido en universidades y laboratorios gracias a la disponibilidad en línea que se tiene del código fuente, lo cual, permite a los usuarios estudiarlo, imitarlo y extender los sistemas. Es difícil que de esta forma se logre la meta deseada para productos que ya estén bajo un contexto industrial más tradicional, ya que, de esta manera no existiría ocultamiento de información, uno de los requerimientos esenciales para la reutilización a gran escala.

La reutilización de personal es una técnica de reutilización altamente utilizada en la industria, de esta manera se evita el que los desarrolladores de software no sepan como se hacen las cosas, al mismo tiempo que se aprovecha la experiencia de los ingenieros de software en otros proyectos.

A continuación presentaremos las formas más comunes en que se ha presentado la reutilización.

I.2.5 INTERFASES.

Los primeros componentes de software que examinaremos son las interfases, este componente de software proporciona a los usuarios características invariantes que determinan el comportamiento del componente.

Dentro de éstas tenemos a las interfases sintácticas, las cuales proporcionan características invariantes durante el momento de la compilación las cuales determinan de que manera

embonan los componentes dentro del programa. Por otro lado están las interfases semánticas que proporcionan características invariantes en el momento de la ejecución y que determinan lo que el componente hace.

Las primeras son suficientes únicamente para especificar como deben embonar los componentes, pero, de ninguna manera nos ayudan a discernir si los cálculos que se están llevando a cabo son correctos, recuerdese por ejemplo, que las primeras versiones de interfases tales como las funciones o los subprogramas de FORTRAN no hacían validaciones tan minuciosas como las que hacen los lenguajes de alto nivel actuales, lo que ocasionaba que hubiera errores en tiempo de ejecución muy difíciles de detectar (por ejemplo cuando se utilizaba el llamado GOTO computado en lugar del GOTO asignado y viceversa).

Algunos lenguajes de programación como Ada no cuentan con un manejo de interfases rígidas, y por lo tanto, han sido severamente criticados. En contraste a la débil interfaz sintáctica que manejan proporcionan características importantes en cuanto a flexibilidad tal como la abstracción de datos.

En general las interfases débiles proporcionan flexibilidad y eficiencia, mientras que las interfases rígidas proporcionan garantía de integridad.

Las interfases rígidas ayudan a elevar la productividad durante el proceso de desarrollo pero limitan mucho el poder de expresión. Su costo efectividad es mayor cuando se desarrollan programas de aplicaciones, donde los "cuellos de botella" se localizan durante las fases de desarrollo y mantenimiento.

I.2.6 RUTINAS.

El enfoque clásico de la reutilización ha sido construir bibliotecas de rutinas, entiéndase, procedimientos, funciones, subrutinas, subprogramas módulos, etc. Cada rutina definida en una biblioteca lleva a cabo una operación bien definida.

Una rutina en seudocódigo para llevar cabo una búsqueda secuencial sería semejante a la siguiente :

```
Buscar(x : Elemento, t : Tabla_Secuencial) : boolean is
        pos: Posición
begin
        INICIAR_BUSQUEDA;
        while not FIN_DE_BUSQUEDA and then not
ENCONTRADO(pos,x,t) do
                SIGUIENTE_ELEMENTO
        end;
        return not FIN_DE_BUSQUEDA
end;
```

Dentro de la computación científica este enfoque ha tenido un éxito completo ya que existen excelentes bibliotecas de rutinas usadas para resolver problemas de álgebra lineal, ecuaciones diferenciales y otros campos. Esta descomposición del software en rutinas es resultado del

análisis Top-Down de descomposición funcional, el cual proporciona un enfoque que funciona bien para problemas individuales que pueden ser identificados y sujetos a las siguientes limitaciones :

A) Cada problema debe admitir una especificación simple, en el sentido de que cada ejemplo del problema puede ser definido por un pequeño conjunto de parámetros.
B) Cada problema individual debe ser claramente distinto de otro : el enfoque no permite hacer un buen uso de los aspectos comunes significativos. Excepto haciendo una reutilización de análisis.
C) No se puede involucrar ninguna estructura compleja de datos : esta debería estar distribuida entre las rutinas que la usan, con lo cual, la autonomía conceptual de cada rutina o componente se perdería.

Las limitaciones del enfoque de rutinas se manifiestan claramente cuando consideramos el problema de la búsqueda en una tabla.

Ante esta dificultad tenemos la posibilidad de escribir varias rutinas:

A) Una para cada caso específico

B) Escribir una sola rutina que decida internamente que parte del código ejecutar

Sin embargo, es tan grande el número de casos que sería difícil llevar a cabo un buen empleo de la reutilización de código. En la figura 2 mostramos algunos de los puntos en que sería necesario emplear instrucciones específicas :

	Arreglo secuencial	Lista ligada	Archivo secuencial
INICIAR_BUSQUEDA	i:=1	l:=cabecera	rewind
SIGUIENTE_ELEMENTO	I:=i+1	l=l.next	readnext
FIN_DE_BUSQUEDA	i>tamaño	l=null	end_of_file

FIG. 2 OPERACIONES A LLEVARSE A CABO DURANTE UNA BUSQUEDA SECUENCIAL.

-En la primera solución (A), tendremos muchas subrutinas semejantes, de las cuales no se podrá utilizar las partes comunes de ellas para beneficiarnos y escribir menos código.

-En el segundo caso (B), tendremos una rutina que recibirá una infinidad de argumentos y que internamente tendrá una buena parte de código únicamente para decidir de que manera va a atacar el problema.

Y lo peor de todo es que una rutina no debe presentar muchos casos aislados sino solamente una operación con casos relacionados.

I.2.7 PAQUETES O MÓDULOS

Los lenguajes modulares tales como Modula-2 y Ada proporcionan un paso más adelante hacia la solución apropiada.

Estos lenguajes utilizan la noción de módulo (o paquete) como una estructura de programación de mayor nivel que la rutina. Tales módulos pueden contener más de una rutina, junto con declaraciones de tipos, constantes o variables, de esta forma un paquete puede estar dedicado al manejo completo de una estructura de datos y a sus operaciones relacionadas.

Los paquetes corrigen algunas de las más escandalosas deficiencias de las rutinas al permitir que un módulo contenga un grupo relacionado de operaciones en lugar de sólo una, la implementación de la estructura de datos también se describe dentro del paquete.

Para el ejemplo de la búsqueda, nuestro módulo no contendría únicamente la búsqueda, sino que además, contendría una implementación completa del concepto de tabla y sus operaciones asociadas.

El enfoque de los paquetes permite reunir a varios elementos (tipos, variables, rutinas) dentro de un mismo módulo.

La ventaja de los paquetes para los implementadores de módulos es que : se agrupan bajo el mismo módulo todas las entidades que relacionan una parte conceptual importante del sistema y se compilan juntas. Lo anterior facilita el mantenimiento y la evolución. A diferencia de las rutinas separadas donde siempre se corre el riesgo de olvidar actualizar alguna subrutina cuando se hace un cambio en la implementación. Además, para un programador cliente es más fácil encontrar y utilizar estas utilerías si todas están en el mismo lugar.

I.2.8 SOBRECARGA Y GENERALIDAD.

SOBRECARGA.

El concepto de sobrecarga no es nuevo, dentro del tema de las propiedades de los lenguajes de programación ya desde hace más de dos décadas que Algol-68 y Ada lo han manejado.

La sobrecarga puede definirse como la capacidad de dar más de un significado a un nombre que aparezca en un programa.

Los operadores son los candidatos típicos para la sobrecarga. Por ejemplo, si uno desea construir varias implementaciones de una tabla, cada una definida mediante una declaración de tipo, la sobrecarga permite llamarlas a todas de la misma forma, de tal manera que cuando uno desee buscar un elemento "x" en una tabla "t" lo único que tendrá que hacer es invocar al módulo encargado de hacer la búsqueda (buscar(x,t)) sin preocuparse por la manera en que fue implementada la búsqueda. Ésto funciona únicamente cuando se usa un con cheque estricto de tipos como Ada o Algol-68, en los que el compilador cuenta con información suficiente para decidir cuál es la versión apropiada de la función "buscar".

GENERALIDAD.

Este concepto se refiere a la capacidad de definir módulos parametrizados. Tal módulo llamado módulo genérico no es utilizable, es por así decirlo, un módulo patrón. Sus parámetros llamados parámetros genéricos reciben tipos específicos de datos.

Los módulos reales, llamados ejemplos del módulo patrón se obtienen al proporcionar tipos reales (parámetros genéricos reales) a cada parámetro genérico real.

La sobrecarga y la generalidad proporcionan muchas posibilidades para el implementador :

A) La sobrecarga es una ayuda para los programadores clientes : ya que hace posible escribir el mismo código, aunque, se utilicen diferentes implementaciones de una estructura de datos.

B) La generalidad es una ayuda para los programadores implementadores de los módulos, ya que les permite escribir el mismo módulo para describir todas las instancias de la misma implementación de una estructura de datos aplicada a diversos tipos de objetos.

C) La generalidad es una solución para que se permita que los tipos sean variados. La sobrecarga permite que varíen las estructuras de datos y los algoritmos, además, de que facilita el requerimiento de servicios sin necesidad de conocer su implementación.

Aunque el enfoque de paquetes resolvió algunos de los problemas que se tenían, podemos observar que no hemos logrado nada en cuanto a satisfacer nuestro deseo de capturar las semejanzas dentro de grupos de implementaciones de las mismas estructuras generales de datos. Hasta el momento no hemos podido describir una jerarquía compleja de representaciones con diferentes niveles de parametrización. Por otro lado no se ha logrado que el programador cliente llame a uno de estos módulos sin que desconozca totalmente los detalles de su implementación. Para lograr nuestro objetivo satisfactoriamente es necesario que utilicemos un enfoque más poderoso: el modelo orientado a objetos.

Podemos obtener de los párrafos anteriores cinco requisitos deseables de un módulo reutilizable :

1.- Permitir que los tipos de datos manejados sean variados.

2.- Permitir que las estructuras de datos y los algoritmos sean variados.

3.- No contengan sólo una operación sino varias operaciones relacionadas.

4.- Que sea posible solicitar una operación sin que sea necesario conocer su implementación.

5.- Manejo adecuado de las situaciones comunes.

I.2.9 REUTILIZACIÓN DE CÓDIGO VS. REUTILIZACIÓN DE DISEÑO.

REUTILIZACIÓN DE CÓDIGO.

El enfoque más obvio de la reutilización es la reutilización de código y la composición de bibliotecas reutilizables, pero, el enfocar el problema de esta manera trae como consecuencia que pronto se alcanza un punto a partir del cual la obtención de beneficios se hace muy difícil, algunos autores [Selby,1989] opinan que bajo este enfoque la proporción de un sistema que puede ser construida con reutilización no rebasa el 50% (Aunque en algunos lugares puede resultar de gran provecho).

REUTILIZACIÓN DEL DISEÑO.

La reutilización de diseños, más que la reutilización de implementaciones se ha llevado a cabo en la mayoría de las ocasiones con éxito en países como Japón donde, además, se han concentrado dentro del concepto de la fabrica de software aspectos como ingeniería de software y calidad total. La idea es que una compañía de desarrollo de software debe guardar los antecedentes y diseños de los sistemas que más frecuentemente ha tenido que desarrollar, de esta manera, va guardando sus experiencias en esa área. Estos documentos describen patrones de modelos más que módulos existentes.

La noción de diseños como productos de software independientes no puede ser llevada a cabo de manera aislada: ya que, si sólo el diseño es reutilizado existe un gran riesgo de reutilizar elementos incorrectos u obsoletos. Con lo anterior nos referimos a que es necesario adaptarlos a la dinámica evolutiva que presentan los sistemas actuales.

Aunque los enfoques de reutilización mencionados anteriormente son limitados sirven para mostrar aspectos importantes del problema de la reutilización :

a) La noción de código fuente reutilizable sirve como recordatorio de que el software está definido por el código, por lo tanto, una política satisfactoria para la reutilización es la de enfocarse en tratar de producir programas reutilizables. Por ejemplo, debería premiarse a los programadores que construyan componentes altamente reutilizables.

b) La reutilización de personal es necesaria pero no suficiente. Los buenos componentes de código son inusuales a menos que los programadores estén entrenados propiamente y hayan adquirido experiencia suficiente para reconocer una situación en la cual los componentes existentes puedan proporcionar ayuda, del mismo modo que sin esta experiencia es más difícil que tengan sensibilidad para preparar a sus componentes para que sean más generales.

c) La reutilización del diseño enfatiza la necesidad que tienen los componente reutilizables de contar con un nivel conceptual elevado de generalidad. Por ejemplo, las clases de la programación orientada a objetos pueden ser vistas como módulos de diseño, así como módulos de implementación.

I.3 UN ENFOQUE AMPLIO ACERCA DE LA REUTILIZACIÓN DE SOFTWARE.

Existen al menos tres diferentes escenarios donde surge la decisión de tomar soluciones acerca del software.

La primera es cuando una aplicación común de software es requerida, en cuyo caso la solución es la adquisición de algún paquete existente en el mercado. La segunda es cuando se necesita una combinación inusual de procesamiento de datos, aquí puede haber dos alternativas a saber : la adquisición y posterior adaptación de algún paquete existente o la contratación de alguna firma de desarrollo de sistemas. La tercera es aquella en que un sistema totalmente distinto constituye el resultado deseado en cuyo caso el desarrollo en casa es la salida recomendable.

Es este último caso el que representa la preocupación e interés de este trabajo.

A continuación se presenta una tabla [Frank, 1981]en la que pueden observarse los costos netos y los porcentajes de mejoramiento que pueden obtenerse durante las distintas fases de desarrollo de un sistema. En ella podemos fácilmente notar cuales son las fases en las que sería más provechoso hacer mejoramientos.

De manera semejante se presenta en la fig. 4 otra tabla, pero, esta vez no solamente se incluyen los costos relacionados con el desarrollo del producto. Por el contrario, se incluyen los costos que involucra el mantenimiento del sistema. Como puede observarse éstos son, la mayor parte de las veces, mucho más considerables que los costos de desarrollo e instalación. Lo anterior, da pauta a poner más cuidado en el desarrollo con miras a abatir los costos de mantenimiento.

Mejoramiento potencial de la productividad durante el ciclo de desarrollo del sistema.

	Costo actual	% de mejoramiento	Costo neto
Requerimientos del sistema	2	0	2
Requerimientos de hardware	8	25 %	6
Requerimientos de software	10	20 %	8
Diseño de software	12	40 %	7
Codificación	13	75 %	3
Prueba individual	24	50 %	12
Prueba de integración	13	30 %	9
Documentación	6	30 %	4
Prueba del sistema	12	25 %	9
	100	40 %	60

FIG. 3

Considerando al ciclo de desarrollo desde un punto de vista más amplio, es decir, incluyendo los costos del ciclo total de vida de sistema fig. 4, podemos observar que el costo del mantenimiento en particular es aproximadamente tres veces el costo del desarrollo del

sistema original. El 40% de mejoramiento obtenido anteriormente se refleja durante el tiempo de desarrollo. De donde puede observarse que el efecto de ese mejoramiento no es tan atractivo cuando se considera los costos del ciclo total de vida fig 4.

Mejoramiento potencial de la productividad durante el ciclo total de vida del sistema

	Costo actual	% de mejoramiento	Costo neto
Desarrollo del sistema	100	40 %	60
Instalación	15	20 %	12
Mantenimiento			
Correctivo	60	75 %	15
Adaptativo	60	30 %	42
Mejoramiento	180	40 %	108
	415	43%	237

FIG. 4

Anteriormente hemos hablado acerca de mejorar la productividad del software, pero, no hemos precisado a que nos referimos con este concepto.

1.3.1 PRODUCTIVIDAD DE SOFTWARE

Definamos a la productividad de software como la proporción existente entre salidas y entradas, donde las entradas incluyen trabajo, capital, material y energía; y las salidas, que en general son más difíciles de medir, se representan por unidades de costo del producto, tiempo de respuesta sobre consultas, etc. En ocasiones se ha considerado el número de líneas como un medio para medir la productividad.

Cuando analizamos cuales eran las características necesarias de un módulo reutilizable hablamos de algunas de ellas sobre las que ahora queremos profundizar, para lo cual responderemos a la pregunta siguiente :

¿Que características deseamos encontrar dentro de la estructura de los módulos que permitan componentes reutilizables apropiados?

1.3.2 REQUERIMIENTOS SOBRE LA ESTRUCTURA DE LOS MÓDULOS.

VARIACIÓN DE TIPOS.

En un módulo reutilizable es deseable que se pueda manejar elementos de diferentes tipos, esto es claro cuando pensamos, por ejemplo, en un módulo que va a realizar búsquedas u ordenamientos, en cuyo caso es previsible que deseemos buscar u organizar datos de diferentes tipos de tal manera que el módulo sea más general.

VARIACIÓN DENTRO DE LAS ESTRUCTURAS DE LOS DATOS Y LOS ALGORITMOS.

Por ejemplo, en un módulo diseñado para la búsqueda también desearíamos que se soportara diferentes tipos de estructuras sobre las que se va a llevar a cabo la búsqueda (árboles binarios, arreglos, archivos, etc.), de la misma manera que sería deseable que fuera posible emplear distintos algoritmos de búsqueda apropiados para el tipo de estructura que se estuviera utilizando (listas lineales, árboles, arreglos etc.) y para la forma en que estuvieran organizados los datos (orden, posorden, preorden).

RUTINAS RELACIONADAS.

Para hacer búsquedas u ordenamientos dentro de una estructura es necesario saber como fue creada la estructura, como añadir elementos, como borrar elementos, etc.

Por lo tanto es claro que una rutina no es suficiente por sí misma, ya que necesita de otras rutinas para que la auxilien en su función.

REPRESENTACIÓN INDEPENDIENTE.

Una estructura modular realmente flexible debe ser capaz de aceptar requerimientos de los clientes sin que estos conozcan la manera en que está implementada la rutina.

Por ejemplo, un cliente de una rutina de búsqueda podría solicitar que se llevará a cabo la búsqueda de un elemento en una estructura específica y la rutina debería ser capaz de llegar a su fin satisfactoriamente, eligiendo el método o algoritmo más eficaz u apropiado para el tipo de estructura donde va a realizar la búsqueda sin la intervención del cliente.

Este requerimiento en su forma más simple es una extensión natural del principio de ocultación de información. Pero en realidad no es sólo eso lo que esperamos obtener para que el módulo sea verdaderamente independiente, sino que queremos que esa independencia no únicamnete se mantenga durante el ciclo de desarrollo del producto, sino que sea lo suficientemente robusta para que resista llamadas de diferente índole en tiempo de ejecución, es decir que podríamos llamarla pasándole como parámetro el nombre de la estructura la primera vez, y luego cambiar la definición de esa estructura y queremos que por medio del enlace tardío la rutina detecte con que tipo de estructura esta tratando y reaccione de la manera adecuada. (Sobre este comportamiento llamado polimorfismo, hablaremos en el capítulo II, dedicado al modelo orientado a objetos).

Es preciso mencionar que aquí la discusión no se centra sobre la reutilización únicamente, sino que tiene que ver mucho con la extendibilidad, es decir, la capacidad con que la rutina podrá expandir sus servicios de manera que se llegue a la solución más adecuada. Tal vez para algunas personas pueda presentarse la dificultad de decidir en que parte se decidirá que método utilizar para llevar a cabo la búsqueda, en cuyo caso contamos con dos enfoques :

1.- Si es la rutina quien lleva a cabo esta decisión, entonces ella misma debe saber todo acerca de todas las estructuras que existan en el sistema, lo cual significa que cada que se añada una estructura habremos de modificar nuestra rutina para que la tome en cuenta.

2.- Si permitimos que sea el cliente quien tome la decisión, éste tendría que comunicar a la rutina el tipo de estructura de que se trata, lo que ocasionaría que si cambiamos la implementación de nuestra estructura tendríamos que modificar todas las llamadas a nuestro rutina para incluir ahí el nuevo cambio de información.

Presentaremos conceptos del modelo orientado a objetos tales como los de clase y herencia lo que facilitara nuestra implementación en el capitulo II.

ASPECTOS COMUNES DENTRO DE LOS SUBPROGRAMAS.

Este tema afecta a los módulos reutilizables en sí mismos, no a sus clientes, por lo que es de fundamental importancia ya que determina la posibilidad de escribir colecciones de módulos bien estructurados, sin excesiva repetición. Si existe mucha repetición entre módulos relacionados, su integridad conceptual es difícil de mantener.

Aquí explicaremos como el implementador puede obtener ventaja de los aspectos comunes de una posible implementación.

En el ejemplo de la rutina de búsqueda o en el de la rutina de ordenamiento, podemos observar que si elegimos solamente un método, entonces debido a que deseamos que este pueda tratar con diferentes estructuras, tendremos operaciones que conceptualmente serán las mismas (avanzar, comparar, detectar el fin de la estructura, etc.), pero que internamente deberán de ser implementadas de diferente manera.

Lo interesante aquí es encontrar una forma de capturar los aspectos comunes dentro de un grupo específico de implementaciones de búsqueda en tablas.

I.4 ESTRUCTURA DE LA REUTILIZACIÓN, EVALUACIÓN Y TENDENCIAS.

El tema de la reutilización ha sido visto como la clave para manejar el desarrollo de software, aunque ésto es cierto, es necesario comentar que la reutilización ha sido una disciplina que muchas veces no ha satisfecho cabalmente las promesas que ha hecho. Aquí explicaremos cuales son las razones que han despertado la creencia de que mejorará el desarrollo de software y la manera en que se debe dirigir para que proporcione los frutos prometidos.

I.4.1 DILEMAS ACERCA DE LA REUTILIZACIÓN.

Una de las experiencias adquiridas durante el desarrollo de sistemas ha hecho concebir la idea de que las tecnologías más generales, las que pueden ser aplicadas a una gran variedad de problemas o dominios de aplicación, tienden a ser menos provechosas que los sistemas que se abocan a uno o dos dominios de aplicación. De manera inversa las tecnologías más beneficiosas son aquellas cuyo dominio de aplicación es más restringido, como se puede apreciar en la fig. 5.

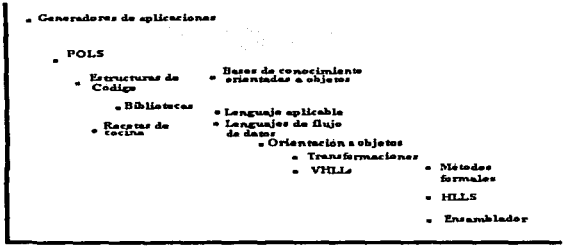


FIG. 5

Recuérdese que generalmente a medida que crece un componente reutilizable los beneficios de reutilizarlo también crecen, sin embargo, al mismo tiempo se vuelve más específico se reduce la posibilidad de reutilizarlo. Además, el costo de reutilizarlo cuando se requiere de alguna modificación se eleva mucho debido a la complejidad inherente a su tamaño.

El porque decimos que mientras más específicos se vuelven los sistemas se reduce la posibilidad de reutilizarlos involucra además de las razones antes mencionadas tales como que es más difícil encontrar un apareamiento exacto de comportamientos y que es menos probable que las mismas características que están contenidas dentro de un módulo se repitan de manera exacta, además, debemos considerar la gran cantidad de información que un módulo encierra en sí mismo, tal como el sistema operativo donde fue desarrollado, bibliotecas de tiempo de ejecución con que contaba durante su desarrollo, equipo o hardware disponible, modelo utilizado para el empaquetamiento de datos, características de sus interfases etc.

Ahora, cuando se considera la creación de una biblioteca reutilizable debe tomarse en cuenta que es el esfuerzo en el que más capital intelectual, real y tiempo se invierte antes de obtener el más mínimo provecho. Generalmente las compañías preincluyen esta gran inversión inicial de capital sin tomar en cuenta los beneficios que obtendrán a largo plazo, las estructuras organizacionales generalmente cuentan con equipos de personal que trabajan en proyectos específicos, los cuales tienen un presupuesto asignado para cumplir con ciertas expectativas las cuales no incluyen ningún esfuerzo adicional para capturar y generalizar las experiencias y resultados obtenidos para su posterior reutilización.

Muchos autores [Wegner,1984] coinciden en que el trabajar en un sistema viable de reutilización representa una inversión considerable en la cual los beneficios no son obtenidos a corto plazo, por consiguiente, muchas compañías no se deciden a enfrentar una inversión en reutilización. De acuerdo con lo anteriormente mencionado es que la reutilización es considerada una actividad de capital intenso.

Para que un sistema funcione correctamente es necesario que proporcione alguna solución a los cuatro problemas siguientes :

–Encontrar componentes.
–Entender a los componentes.
–Modificar los componentes y
–Organizar a los componentes.

El encontrar un componente va más allá del sólo encontrar aquellos que sean similares a simple vista, es necesario profundizar en la abstracción de cada componente para que puedan ser vistas las similitudes, de modo que los nuevos componentes únicamente sean desarrollados en parte a fin de reducir el esfuerzo y eliminar defectos.

Imaginemos que contamos con módulos altamente especializados y que a partir de ellos queremos construir un programa nuevo, entonces, podemos esperar que requiramos de todas las anteriores características para que el desarrollo del programa llegue a su feliz culminación. En casos como éste la importancia de encontrar la similitud entre las necesidades del nuevo sistema y las ventajas que proporcionan los componentes de nuestra biblioteca cobra gran importancia.

El asunto no es insoluble ya que existen formas de disminuir la especificidad y seguir manejando módulos relativamente grandes, de modo que la importancia del problema de hallar la similitud entre componentes reutilizables se vuelva una tarea menos importante. Por supuesto lo anterior puede lograrse haciendo que cada módulo capture únicamente un aspecto o principio de un algoritmo, de tal modo que el esquema de abstracción pueda llevarnos a un espacio de búsqueda relativamente pequeño.

El entendimiento de los componentes es esencial en cualquier caso, pero, más en el caso de que estos no satisfagan los requerimientos del desarrollador al 100% y en cuyo caso vayan a ser modificados. El usuario del módulo debe tener entendimiento acerca de lo que el módulo hace, que entradas necesita y que resultados proporciona; este conocimiento es algo de lo más difícil de adquirir.

Para poder organizar a los componentes es necesario que la representación usada para especificar componentes tenga un carácter dual, por un lado debe representar estructuras compuestas como entidades independientes con características computacionales bien definidas y por otro lado debe ser posible componer con ellas nuevos y más complejos componentes sin alterar sus características computacionales. A veces suele suceder que estos requerimientos son antagónicos.

Los cuatro procesos de encontrar, entender, modificar y ensamblar componentes han sido tratados típicamente de manera informal durante el diseño, un enfoque hacia la reutilización requiere que sean formalizados de modo que las estructuras de soporte automático puedan ser creadas.

1.4.2 PRINCIPIOS DE CLASIFICACIÓN.

Clasificar objetos consiste en agrupar en clases a objetos que presenten características en común, que no pertenezcan a ningún otro objeto que este fuera de la clase. Para poder

referirnos a los objetos de una clase es necesario contar con una herramienta para crear un orden sistemático basado en un vocabulario índice controlado (al que llamaremos cédulas de clasificación), a todo esto se le llama esquema de clasificación.

Existen dos relaciones importantes dentro de un esquema de clasificación una es la jerárquica y la otra es la sintáctica, la primera se basa en el principio de inclusión mientras que la segunda se utiliza para clasificar dos objetos de diferentes jerarquías.

Los esquemas de clasificación se llaman enumerativos cuando se propone un universo de conocimiento dividido en clases que incluyen a todos los objetos existentes y se llaman esquemas de clasificación en facetas cuando la construcción o sintetización de las características de un objeto se ve como una lista escrita en una cédula, de tal manera que cuando se intenta situar a un objeto dentro del esquema es necesario consultar su cédula para conocer sus características sintéticas o facetas.

El esquema enumerativo presenta el problema de navegar a través de un orden jerárquico para encontrar la clase apropiada, lo cual, puede ser frecuentemente una tarea difícil ya que muchas clases pueden parecer elecciones apropiadas. Fig. 6

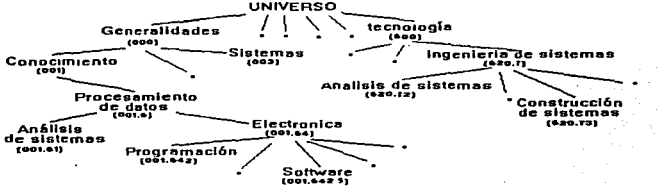


FIG. 6

Para compensar la ambigüedad anterior se proporcionan referencias cruzadas, lo que ocasiona que haya una gran cantidad de errores.

El esquema de clasificación por facetas va más directo al tema al presentar un esquema en el cual el objeto cuenta con varias facetas o por decirlo de otro modo puede ser visto a través de varios puntos de vista, gracias a lo anterior cuando queremos encontrar un objeto apropiado lo que hacemos es buscarlo asomándonos tomando como punto de partida la faceta con que debe contar que nos parezca la más importante, una vez encontrado un cierto conjunto de elementos módulos que satisfagan esta primera característica, buscaremos en el nuevo conjunto solución otra nueva característica reduciendo nuevamente el conjunto solución de nuestra búsqueda, continuaremos de esta manera hasta que lleguemos al elemento más adecuado.

Este método resulta adecuado para clasificar software debido a que las facetas que vamos eligiendo para llevar a cabo la búsqueda representan de alguna manera requerimientos en orden de importancia.

Otro beneficio adicional es la flexibilidad, por ejemplo, bajo el esquema de clasificación enumerativo agregar un nuevo objeto traería como resultado añadir engorrosas referencias cruzadas, mientras que en un esquema de facetas, sería semejante a añadir un lente diferente con el cual observar al objeto.

I.4.3 CLASIFICACIÓN DE MÓDULOS REUTILIZABLES.

Debemos pensar que una de las ideas que presentó [McIlroy,1969] acerca de los componentes de software era la de que existieran catálogos de componentes en los cuales se pudiera buscar por determinadas características deseadas por el usuario; respecto a este trabajo algunos autores[3] obtuvierón excelentes resultados.
Las compañías de desarrollo de software japonesas también han obtenido grandes avances en el desarrollo de software al incorporar prácticas de diferentes disciplinas como ingeniería de producción, manejo de recursos, control de calidad, ingeniería de software y psicología industrial.

Es evidente que para que las ideas de la reutilización tengan acogida debe de resultar más sencillo reutilizar que desarrollar. Por lo anterior, es conveniente identificar tres factores importantes de la reutilización :

1.-Acceso a código existente.
2.-Entendimiento de tal código.
3.-Adaptación.

Para poder accesar dicho código es necesario contar con un esquema dentro del cual se pueda clasificar al software. El entendimiento por otro lado depende de la experiencia del usuario y de algunas características del programa: tales como tamaño, complejidad, documentación, etc.; mientras que la adaptación depende de las necesidades del usuario así como de las características del `programa.

En esta sección trataremos el tema de la clasificación de módulos, debido a que es de gran importancia y merece atención.

En primer lugar, si por ejemplo, clasificamos los módulos en base a los requerimientos de software necesarios para utilizarlo, nuestra clasificación pecaría de ser muy pobre. En consecuencia debemos pensar que una colección debe contar con un método eficiente de búsqueda y recuperación, al mismo tiempo, debe permitir que el usuario decida, de entre un conjunto de módulos, cual utilizar en su proyecto. Sin este método por muy basta que fuera una biblioteca su explotación no sería la más satisfactoria.

Lo que se necesita en palabras de Rubén Prieto Díaz es :

"Un esquema de clasificación, incluido dentro de un sistema de recuperación que este soportado sobre un mecanismo de evaluación".

[3] Lanergan, R. G. y Poynton, B. A. "Reusable code : The aplication development technique of the future." Proceedings of the IBM SHARE GUIDE Software Symposium. Octubre 1979.

Las características principales a tomarse en cuenta durante la clasificación de una colección de módulos son :

a) **Expandibilidad**.- Anexar nuevas clases procurando el menor impacto sobre la colección.
b) **Adaptabilidad**.- Se refiere a que el esquema pueda ser fácilmente adaptado a un medio ambiente particular.
c) **Consistencia**.- Significa que componentes de distintas colecciones pero dentro de una misma clase comparten atributos en común.

1.4.4 CLASIFICACIÓN DE SOFTWARE.

Los esquemas de clasificación de software utilizados más comúnmente son enumerativos y están dirigidos a usuarios finales (aquellos a los cuales no les interesa la adaptación), por lo tanto, no están organizados por atributos relacionados con la reutilización, sino por áreas de aplicación, tipos de problemas a solucionar, hardware donde corren, etc. Por ejemplo el esquema de revisiones sobre computación de ACM, los esquemas por funcionalidad (GAMS, SHARE, SSP, SPSS, IMLS, etc.), esquemas por catálogos comerciales de software (ICP, IDS, IBM Software Catalog, Apple book, etc.).

El esquema de facetas propuesto por Rubén Prieto se basa en la consideración de que las colecciones de componentes reutilizables son muy grandes y se expanden rápidamente, además, de que existen grandes grupos de componentes similares.

Este esquema cuenta con un formato de descripción de componentes que se basa en un vocabulario de términos estándar con el cual el usuario puede buscar la mejor solución. En principio propone que los componentes de software pueden describirse por medio de tres atributos saber:

1.- La tarea o función que llevan a cabo.

2.- La forma en que la llevan a cabo y

3.- Los detalles de su implementación.

Estos descriptores caen naturalmente en facetas que pueden ordenarse por su importancia dentro de la reutilización, de manera que un descriptor puede verse como una tupla en donde cada término representa el valor de un atributo de una faceta en particular.

Debe ser claro que aunque dos módulos lleven a cabo la misma función, la forma en que la lleven a cabo sea radicalmente diferente dependiendo de los objetos sobre los que trabaje el programa, de modo que con estas tres características necesarias para describir un módulo podemos formar una tupla de la forma :

<acción,objeto,medio>

acción es la función que se lleva a cabo, el objeto es la estructura u elemento sobre el que se actúa y medio representa el lugar donde se lleva a cabo o sucede la acción.

MEDIO AMBIENTE DEL PROGRAMA.

Podemos decir que un programa existe en dos medios ambientes diferentes, el primero es el llamado medio ambiente interno del programa, que se refiere al lugar donde se ejecuta el programa, (por lo que se relaciona con la portabilidad del programa) y el segundo medio ambiente es el llamado medio ambiente externo que se refiere al lugar donde el programa se aplica. Lo que implica cambios de especificaciones y por lo tanto modificaciones de diseño.

Dentro del medio ambiente externo del programa se pueden seleccionar grupos de palabras y acomodarlos en tres grupos principales :

a) Términos que describen el tipo de sistema.

b) Términos que describen el área funcional.

c) Términos que describen el lugar de aplicación.

Al incorporar estos tres elementos a la 3-tupla obtenemos una 6-tupla donde emplearemos el orden de los atributos de acuerdo con la importancia que tienen para el usuario. Los cuales son en nuestro caso ingenieros de software diseñando o construyendo nuevos sistemas a partir de componentes.

<funcion,objetos,medio,tipo_de_sistema,area_funcional,medio_circundante>

La figura Fig. 7, muestra un ejemplo de algunos ejemplos de atributos para la cédula de clasificación bajo el esquema de facetas.

LISTA PARCIAL DE UNA CÉDULA DE CLASIFICACIÓN EN FACETAS

{ función }	*{ objetos }*	*{ medio }*	*{ tipo de sistema }*	*{ área funcional }*	*{ aplicación }*
sumar	argumentos	arreglo	ensamblador	cuentas-pagables	publicidad
anexar	arreglos	buffers	generación de código	cuentas-cobrables	fabricación de aviones
cerrar	retrocesos	tarjetas	optimizador de código	análisis-estructural	almacenamiento de accesorios
comparar	espacios en blanco	disco	compilador	auditoría	asociación
completar	buffers	archivo	manejador-DB	control de trabajos	reparación de autos
comprimir	caracteres	teclado	evaluador de expresiones	facturador	peluquería
crear	descriptores	línea	manejador de archivos	contaduría	estación de radio
decodificar	dígitos	lista	DB-Jerárquica	presupuesto	estación de cable
borrar	directorios	ratón	DB-Híbrida	planeación de la capacitación	vendedor de autos
dividir	expresiones	impresora	interprete	CAD	catálogo de ventas
evaluar	archivos	pantalla	analizador de léxico	costos contables	cementerio
intercambiar	funciones	sensor	editor de líneas	costos de control	circulación
expandir	instrucciones	pila	inversor de matrices	información de clientes	anuncios clasificados
dar formato a	enteros	tabla	igualador de patrones	análisis de DB	limpieza
entrada	líneas	cinta	predictive-parsing	diseño de DB	tienda de ropa
insertar	listas	árbol	DB-Relacional	manejo de DB	composición
unir	macros	.	recuperador	modelado	tienda de cómputo
medir	páginas	.	manejador	programación	.
modificar	.	.	.	.	.
mover	.	.	.	.	.

FIG. 7

En este punto el problema no ha quedado resuelto del todo, ya que el describir a un componente implica que al utilizar sinónimos se produzcan descriptores diferentes para un mismo componente.

<transfiere,cadena,variable>
<copia,apuntador,variable>

Para evitar tales ambigüedades es necesario utilizar un vocabulario controlado, en el cual se especifique para cada término un conjunto de sinónimos, para facilitar la relación entre estos en el esquema de búsqueda se relacionara a cada palabra con sus sinónimos a través de arcos con peso de manera que mientras las palabras tengan significados más cercanos este peso será mayor y viceversa; de modo que las palabras más comunes serán utilizadas como llaves.

Esta característica nos proporciona el poder de establecer lejanía o cercanía entre términos y componentes.

Al hacer la búsqueda, sino se encuentra un descriptor idéntico al que se proporciona para hacer la búsqueda se podrá encontrar aproximaciones cercanas.

Para el lector interesado en conocer un ejemplo real donde fue aplicado este esquema de clasificación de manera exitosa consulte [Prieto,1985]

I.5 FORMAS EXISTENTES DE REUTILIZACIÓN DE SOFTWARE.

Bibliotecas de Subrutinas.
Compiladores.
Simulación.
Sistemas parametrizados.

I.5.1 BIBLIOTECAS DE SUBRUTINAS.

En general existen dos tipos de bibliotecas de subrutinas : las que proporcionan los centros de desarrollo y las bibliotecas de subrutinas tales como los paquetes de programación lineal, las bibliotecas de estadística (BMD o SPSS), y bibliotecas de análisis numérico (IMSL). Ambas cuentan con un conjunto de características comunes las cuales mencionaremos a continuación.

- Las necesidades que debe cumplir la aplicación son claras.
- El propósito de cada módulo es usualmente definible de forma concisa en lenguaje cotidiano.
- Cada rutina es fija excepto por algunos parámetros.
- Las rutinas pueden ser insertadas dentro de un programa y utilizadas sin importar el lenguaje utilizado.

La forma más común del concepto de código reutilizable esta enfocado a la construcción de sistemas a partir de bibliotecas de subrutinas.

I.5.2 COMPILADORES.

Los compiladores son una herramienta básica para el desarrollo de aplicaciones, el lenguaje utilizado para escribir las especificaciones es la forma de Backus-Naur (BNF). Una vez que el formalismo de la forma BNF es comprendido se puede construir un generador del parser de manera que la especificación sintáctica produzca las tablas a ser utilizadas por el analizador sintáctico. La herramienta final es el compilador de compiladores, el cual permite la traducción del lenguaje fuente en lenguaje objeto.

I.5.3 SIMULACIÓN.

Esta aplicación ha sido por mucho tiempo reconocida como una de las más provechosas aplicaciones dentro de la computación, lo cual ha provocado que mucha gente defina los elementos esenciales comunes a toda simulación, los cuales incluyen un mecanismo para definir eventos y relaciones entre eventos dentro de un contexto de reloj simulado.

I.6 ESTADO ACTUAL DE LA REUTILIZACIÓN.

El estado actual de la reutilización dista ya mucho de aquellos tiempos en que la teoría era lo único que existía, ya desde los años 1980 se ha contado con proyectos de interés en reutilización.

Anteriormente mencionamos que la reutilización es considerada como una inversión de capital intenso, lo cual no motivo en un principio a las grandes compañías a tomar la reutilización como una alternativa viable, en cambio si provocó una cierta renuencia de parte de las compañías para decidirse a participar en proyectos de reutilización. Sin embargo los éxitos obtenidos en algunos proyectos han mitigado este desinterés y han alentado la reutilización. Aquí platicaremos de ciertos proyectos exitosos, así como de los mecanismos disponibles para soportar la reutilización.

Entre los mecanismos tenemos el depósito de software para Ada del departamento de defensa de los Estados Unidos (DOD Ada Software Repository) y al conocido como AdaNET, el primero se estableció en 1984 con la intención de promover el uso e intercambio de programas y herramientas Ada y para promover el desarrollo educativo en este lenguaje.

AdaNET es un servicio de información auspiciado por el gobierno de los EEUU, establecido en octubre de 1988 para facilitar la difusión de la Ingeniería de Software y de la tecnología Ada desarrollada por el gobierno para el sector privado. Este programa costeado por la División de Utilización de Tecnología de la NASA (National Aeronautics and Space Administration) y por la oficina del programa de unión para Ada del departamento de defensa :AJPO (Ada Joint Program Office); principalmente ofrece conexión a computadoras para la distribución de información acerca de software escrito en Ada, bibliografía, conferencias, seminarios, entrenamiento, educación, productos y estándares para desarrolladores en Ada.

Ambos mecanismos de soporte de reutilización proporcionan ayuda práctica a organizaciones interesadas en reutilización.

En cuanto a documentos existen muchos y muy variados que cubren el amplio espectro del tema de la reutilización, en particular J.W Hooper recomienda el libro "Software Engineering

with Ada" (escrito por Grady Booch) como una excelente referencia para el uso de Ada y para la creación de componentes de software reutilizables haciendo énfasis en el modelo orientado a objetos. En las referencias [Biggerstaff-Perlis.1989] y por [Hooper-Chester.1991] se podrá encontrar una basta referencia bibliográfica.

Otro programa de interes es el programa STARS (Software Technology for Adaptable Reliable Systems) de los EE.UU. El cual, hasta 1990 había proporcionado 33 componentes reutilizables en Ada sin cargo alguno, únicamente proporcionando un cartucho de cinta de un cuarto de pulgada, donde se incluia todo el código fuente, pruebas de software y documentación producida por el laboratorio de desarrollo de la marina de los EE.UU. NRL (National Research Laboratory) en cooperación con el programa STARS.

[Selby ,1989] estudio las actividades de reutilización de software llevadas a cabo en el centro de vuelo espacial Goddard; GFSC (Goddard Space Fligth Center); donde se han desarrollado sistemas medianos y grandes (de 30,000 a 112,000 líneas de código fuente en FORTRAN) que controlan dispositivos de vuelo sin tripulación.

Ahora hablemos un poco acerca de los beneficio que reportan diversos autores. Algunos reportan reutilización de código del 32 %, otros estiman que cerca del 60 % de sus diseños y códigos de aplicaciones eran redundantes y que al estandarizar las funciones en módulos reutilizables, experimentaron cerca de 50 % de beneficio en cuanto a productividad, a la vez que obtuvieron ahorros en el proceso de mantenimiento que les permitieron utilizar a gente de mantenimiento para desarrollo de nuevos sistemas.

[Biggerstaff-Perlis.1989] presentarón la documentación del trabajo de Lanergan-Grasso[4], así como la de Prywes-Lock[5] sobre la reutilización en los negocios. Prywes-Lock se basaron en el enfoque de generación de programas con los cuales obtuvieron el triple de beneficios en cuanto a productividad por programador.

[Traez,1990] piensa que el mayor beneficio de la reutilización se obtiene con el decremento de los costos de mantenimiento. Ya que reporto hasta 90 % de reducción en el costo en ese aspecto cuando se utiliza código reutilizable, patrones de código y generadores de aplicaciones.

Aunque muchas empresas están obteniendo beneficios de la inversión en reutilización, existen algunas empresas cuyos proyectos no presentan suficientes aspectos comunes para hacer recuperable la inversión (anteriormente ya comentamos un poco acerca de las características que presentan los proyectos factibles para aplicar los conceptos de la reutilización).

De entre los casos en que se ha decidido hacer una inversión en reutilización ha habido proyectos que han tenido un éxito considerable, los cuales han sido estudiados con el fin de obtener algunas características comunes, las cuales han sido consideradas como pautas o sugerencias para lograr resultados satisfactorios en los proyectos de reutilización.

[4] Lannergan R. G. & Grasso, C. A. "Software Engineering with Reusable Design and Code." IEEE Transactions on Software Engineering., SE 10(5), 1984

[5] Prywes, N. S. & Lock, E. D. "Use Of The Model Equational Language and Program Generator by Management Profesionals.", Software Reusability Applications and Experience., Vol. II. ACM Press , Adison Wesley Reading , Mass. 1989

I.6.1 FACTORES QUE FRENAN EL DESARROLLO DE LA REUTILIZACIÓN.

La más importante es la actual falta de un medio de representar las ideas del diseño de manera que se aliente la reutilización.

El segundo obstáculo es que no se tiene claro que estrategia representa el enfoque óptimo para la reutilización, (los directivos generalmente no toman decisiones hacia un camino hasta que no saben cual es la mejor ruta). Otro obstáculo es que los desarrolladores a veces se muestran renuentes a la reutilizacion ya que piensan que con la reutilización su trabajo dejará de ser creativo y que se volverá más tedioso (problema que puede manejarse con una administración adecuada y haciendo comprender a los desarrolladores que los nuevos problemas a resolver también requieren de creatividad).

Como no siempre se da el caso de la reutilización de código satisfactoriamente algunos investigadores han encontrado algunos obstáculos comunes como los siguientes.

I.6.2 OBSTÁCULOS NO TÉCNICOS.

Algunos obstáculos son económicos, por ejemplo, si una empresa de desarrollo de software vende a un cliente un producto de software ampliamente general y reutilizable, entonces, la empresa desarrolladora de software no podrá volver a obtener un contrato con ese cliente; porque ese cliente no necesitara ningún otro producto. Si la reutilización es exitosa a gran escala, entonces debe buscarse incentivos para recompensar a las compañías que desarrollan productos reutilizables y dentro de las compañías a los programadores cuyo trabajo sea ampliamente reutilizable.

Existen otros factores a considerar por ejemplo :

i) El producto más ampliamente reutilizable es totalmente inutilizable si nadie lo conoce, si toma mucho tiempo conseguirlo o si es muy caro (A mitad del capitulo platicamos un poco sobre la clasificación de módulos reutilizables con el fin de que la búsqueda y recuperación de estos sea eficiente).

ii) El éxito práctico de las técnicas de reutilización requiere de bases de datos o componentes de software que puedan ser encontrados fácilmente cuándo un usuario busque un componente apropiado para satisfacer sus necesidades.

iii) También seria necesario contar con mayor acceso a servicios de redes en los cuales se pueda adquirir productos a bajo costo y de la misma manera se proporcionen beneficios a aquellos desarrolladores cuyo producto tenga mayor aceptación.

Las dificultades psicológicas no deben ser menospreciadas, es de suponerse que los componentes de software que se reutilizen sean mejores en cuanto a calidad, facilidad de uso y costo. De esta manera una ventaja marginal no debe ser motivo para que un programador utilice componentes en vez de crear sus propias rutinas.

Se han llevado a cabo múltiples esfuerzos para tratar este tema, en EE.UU. el programa STARS (Software Technology for Adaptable, Reliable Systems), tiene como meta

proporcionar bibliotecas de componentes Ada, y en Japón se está trabajando sobre las llamadas "Fabricas de Software" instaladas en algunas compañías.

Desde el punto de vista de [Meyer,1988] las soluciones organizacionales para el problema de la reutilización de código son significativas sólo si parten de las bases apropiadas. Sin embargo, el principal obstáculo es técnico : sencillamente no tenemos la idea apropiada de módulo.

I.6.3 OBSTÁCULOS TÉCNICOS.

CAMBIO Y CONSISTENCIA.

Las dificultades de la reutilización de código saltan a la vista cuando uno hecha un vistazo a la naturaleza de la repetición dentro del desarrollo de software. Este análisis revela que aunque los programadores tiendan a hacer lo mismo una y otra vez, realmente es muy raro que lo hagan. Si ésto no sucediera la solución seria sencilla, pero, en la práctica no sucede así ya que muchos detalles cambian de manera que no se consigue éxito alguno al tratar de captura los aspectos comunes de las tareas llevadas a cabo por las rutinas en principio semejantes.

Por ejemplo : supongamos que se cuenta con una rutina que sirve para buscar un elemento dentro de una estructura (arreglo, lista, árbol, archivo etc.) aparentemente el proceso de buscarlos de manera secuencial es muy sencillo de resolver, pero, cuando se entra en los detalles de implantación de dicha rutina es necesario conocer muchos aspectos (por ejemplo que tipo de dato va a ser buscado de que manera se va a comprobar si el elemento actual es el elemento deseado, como se llevara a cabo el avance hacia el próximo elemento etc.), como podemos notar el cliente de este servicio necesita conocer un poco de la manera en que este será implantado para poder confiar en el. Lo anterior no quiere decir que la meta de reutilizar código este fuera del alcance de nuestras manos, existen ciertos enfoques simples que han sido aplicados con éxito.

I.6.4 FACTORES QUE MOTIVAN LA REUTILIZACIÓN.

Los factores que alientan reutilización son puramente económicos. Mientras los beneficios tecnológicos pueden proporcionar beneficios marginales, pronto llegan a un punto en el que es más difícil obtener beneficios. Por otro lado y aunque parezca incierto el principal aliciente en los Estados Unidos lo constituyó el éxito obtenido por los japoneses en este terreno.

I.6.5 PAUTAS ADMINISTRATIVAS PARA LA REUTILIZACIÓN.

Hay quienes opinan que los obstáculos más significativos para el éxito de la reutilización son puramente técnicos y hay quienes opinan que bastaría con resolver los problemas administrativos para obtener beneficios de la reutilización. Aquí compartimos el punto de vista de que el descuido o falta de atención en cualquiera de los dos aspectos representa una mengua en el éxito de la reutilización.
En esta sección presentaremos algunos aspectos administrativos relacionados con la reutilización que es preciso poner en claro o cuidar para obtener un mayor beneficio, tales como : el enfoque administrativo y organizacional, los relacionados con el comportamiento de la gente dentro de los proyectos, algunas consideraciones contractuales, legales y financieras.

I.6.6 ENFOQUES ADMINISTRATIVOS Y ORGANIZACIONALES.

Para obtener beneficios de un proyecto de reutilización existen varios factores o temas que necesitan ser resueltos desde antes. Existen factores económicos, organizacionales, legales, sociológicos, políticos, tradicionales, etc. que deben resolverse dentro de cada organización en particular, aquí se considerara un amplio grupo de sugerencias para eliminar o aliviar la renuencia y promover la reutilización.

I.6.7 ESTRUCTURA Y MANEJO ORGANIZACIONAL.

Los ejecutivos de nivel superior deben organizar los recursos necesarios de manera que exista un enfoque nuevo para el desarrollo y mantenimiento del software que incluya herramientas, personal bien entrenado y una biblioteca inicial adecuada para la reutilización. Esto significa hacer una inversión a largo plazo en la cual se deben fijar metas realistas y se debe aceptar los riesgos.

[Biggerstaff-Perlis,1989b] observaron que los proyectos exitosos que analizaron presentaban en común el compromiso administrativo de incluir una inversión y proporcionar seguimiento activo para el logro de los objetivos.

En su trabajo proporcionaron las siguientes recomendaciones administrativas :

[i] Proporcionar a los programadores de desarrollo los más completos y mejores recursos (estaciones de trabajo, herramientas y oficinas)
[ii] Establecer estándares organizacionales para diseño, reutilización y programación.
[iii] Aplicar el esfuerzo de los programadores y diseñadores al desarrollo de la organización
[iv] Obtener el apoyo de los administradores para que todo el trabajo sea hecho motivando un clima donde las buenas prácticas y la reutilización sean recompensadas, y las alternativas no.
[v] Al mismo tiempo, establecer organizaciones de desarrollo cuya misión sea explorar nuevas tecnologías que extiendan los estándares actuales de desarrollo, las herramientas, medio ambiente y tecnologías para el mejoramiento de los beneficios y para responder a movimientos similares de la competencia.

En la mayoría de los proyectos de reutilización que han tenido éxito los administradores de alto nivel han proporcionado apoyo a los proyectos.

El personal técnico debe saber que la alta administración está comprometida con la reutilización y que la falta de esfuerzo y negligencia al respecto no son aceptables, y que por otro lado, los esfuerzos y éxitos en la reutilización serán recompensados.

Muchas buenas ideas fracasan debido a que la administración proporciona poco o ningún apoyo al personal técnico, dejándole toda la carga y sin proporcionar incentivos. Es seguro [Hooper-Chester,1991] que la reutilización de software no puede tener éxito sin el apoyo administrativo. Ya que el personal técnico no podrá encaminar los esfuerzos necesarios para lograr el éxito a menos que los administrativos proporcionen recursos y metas realistas para el cumplimiento de las actividades.

Los directivos deben iniciar la planeación y toma de decisiones, incluyendo el alcance de la reutilización (que componentes organizacionales se incluirán, que fases del ciclo de vida del sistema se considerarán y que áreas de aplicación se incluirán). Deben decidir que tratamiento se le dará a la estructura y al comportamiento de la organización.

Probablemente sea necesario considerar los aspectos financieros y contractuales. Que proceso utilizará la organización para el desarrollo y mantenimiento de software.

Los resultados que se desea obtener después de haber superado los obstáculos son el incremento de la productividad y la calidad, el mejor incentivo para el personal ya sea técnico o administrativo será el de participar en un proyecto exitoso.

Otro aspecto importante a considerarse es el de que los educadores deben enseñar a sus alumnos como llevar a cabo la reutilización, incluyendo temas como dominio del análisis, generadores de aplicaciones y programación parametrizada.

Por otro lado [Curtis,1989] piensa que la demanda de programadores y analistas con conocimiento especializado no se reducirá por la disponibilidad de componentes reutilizables, ya que sólo estos serán capaces de distinguir entre varios componentes con funcionalidad similar. Además, la reutilización enfatiza el diseño, el control de interfases y el chequeo o prueba de sistemas, los cuales requieren de programadores capacitados. Más aún la reutilización no promueve la codificación y prueba de módulos lo que reduce la necesidad de programadores novatos. El ingeniero de software actualmente está evolucionando del programador al diseñador.

[Hooper-Chester,1991] basándose en el trabajo de Fairley-Bollinger-Pfleeger[8] proporciona algunas pautas administrativas a manera de recomendaciones sobre los puntos en los que debe poner cuidado la administración.

[i] Los directivos administrativos deben establecer metas de reutilización, crear una infraestructura organizacional para soportar la reutilización de software, establecer políticas y proporcionar los recursos necesarios.
[ii] Los administradores de nivel intermedio deben desarrollar procedimientos, distribuir recursos y establecer controles y medidores para llevar a cabo las metas.
[iii] Los administradores de primer nivel y el personal técnico deben llevar a cabo actividades de reutilización dentro de proyectos individuales.
[iv] Establecer una entidad organizacional cuyo carácter sea promover la reutilización a nivel corporativo.
[v] Establecer una relación cercana entre las actividades de reutilización y de mantenimiento.
[vi] Proporcionar diferentes tipos de entrenamiento para gerentes, desarrolladores y especialistas en reutilización.
[vii] Proporcionar un flujo financiero seguro para aquellos proyectos que practiquen la reutilización.
[viii] Proporcionar asignación de tareas al personal en las que se incluya la reutilización.
[ix] Darse un tiempo de dos o tres años después de iniciado el proyecto de reutilización antes de obtener beneficios económicos.
[x] Ser realista. No prometer mucho demasiado pronto.

[8] Fairley, R. Bollinger, T & Pfleeger, S.L., "Final Report: Incentives Reuse of Ada Components." Vols. 1-5. George Mason University 1989.

1.6.8 COMPORTAMIENTO ORGANIZACIONAL.

En ocasiones el personal también ha mostrado un comportamiento de desgana ante la reutilización en la mayoría de los casos [Hooper-Chester,1991] obtuvierón estos resultados después de analizar el trabajo de Fairley-Bollinger-Pfleeger[7]. La idea principal que sostiene el comportamiento de desgana es debido a que consideran más "divertido" construir que adaptar y mencionan los siguientes obstáculos para llegar a la reutilización de una manera satisfactoria : restricciones de tiempo, estilo y prisa durante el desarrollo. Además de que enfatizan el punto de vista de que para tener grandes logros se debe contar con apoyo administrativo desde las posiciones gerenciales superiores, es decir debe existir una cultura de reutilización.

Para motivar a los individuos a participar se sugiere proporcionar incentivos tales como premios, bonos, reconocimientos, etc. y principalmente mejorar la satisfacción psicológica del trabajo; por ejemplo promover la reutilización a niveles más altos que la simple codificación, definir cuidadosamente los roles del personal involucrado, determinar niveles de avance a través de programas de metas, propiciar el desarrollo profesional a través de la rotación de puestos, practicar el ocultamiento de información y el desarrollo orientado a objetos. En general la manera de mejorar el comportamiento organizacional puede resumirse en las siguientes sugerencias:

[i] Proporcionar recompensas e incentivos para la participación en la reutilización.
[ii] Buscar la manera de mejorar la satisfacción psicológica para motivar la participación.

1.6.9 CONSIDERACIONES LEGALES Y CONTRACTUALES.

Un problema que desalienta el llevar a la práctica la reutilización es el de los contratos o arreglos sobre la pertenencia de los derechos de autor. En ocasiones el alto costo de los productos desanima a los empresarios a adquirir productos reutilizables, se ha hablado de que el gobierno debería hacer atractivo a las compañías el crear software reutilizable por medio de incentivos para motivar la contribución o aplicación de la reutilización.(En Estados Unidos).

Las compañías desarrolladoras deben conservar la propiedad de cierto software como ventaja competitiva, pero, a la vez es satisfactorio el que compartan los beneficios de su desarrollo; de modo que deben cobrar regalías por el uso de software

Algunos autores[8] opinan que el gobierno de los Estados Unidos no debe de tratar de establecer bibliotecas de software reutilizable a menos que estas estuvieran constituidas por software desarrollado por el propio gobierno, además, piensan que estas bibliotecas tenderían hacia la obsolescencia debido a que el gobierno cuenta con menos recursos que la industria para pagar los gastos de esta inversión

[7] Fairley, R. Bollinger, T & Pfleeger, S.L., "Final Report: Incentives Reuse of Ada Components." Vols. 1-5, George Mason University 1989.
[8] Baker, B. & Deeds, A. "Industrial Policy and Software Reuse a Systems Approach." Proceeding of the reuse in Practice Workshop, Ed. J. Baldo and C. Braun Software Engeneering Institute, Pittsburg, Penn. 1989.

Una solución a este respecto es la de promover por medio del gobierno la reutilización de software proporcionando diversos terrenos de actividad a las empresas de modo que se evite la duplicidad de trabajo, lo cual podría lograrse tomando en cuenta las siguientes sugerencias :

[i] Motivar la reutilización dentro de la organización gubernamental.
[ii] Motivar la reutilización dentro de las compañías, en cuyos contratos se practique una política que permita decidir a la gerencia técnica de la compañía como manejar el proceso.
[iii] dentro de los contratos con múltiples compañías fomentar la compartición de software así como también la reutilización interna.

1.6.10 CONSIDERACIONES FINANCIERAS.

En general es más costoso crear software reutilizable que software común, lo anterior se debe a que es más difícil generalizar, clasificar, probar y documentar tales componentes reutilizables. Como hemos mencionado con anterioridad la reutilización es considerada una inversión de capital intenso, es decir, que requiere cada vez de mayor inversión y se espera que los beneficios; aunque satisfactorios; no sean inmediatos. Por lo anterior la industria de software corre el riesgo de sufrir una descapitalización, los factores principales que pudieran motivar este desarrollo indeseable y sobre los cuales es necesario poner especial atención son los siguientes :

⇒ El que los enfoques financieros de las compañías no logren adaptarse a la alta necesidad de capitalización que necesita el desarrollo de software

⇒ Muchas compañías no se dan cuenta de la importancia del software para su desarrollo futuro.

⇒ Se requiere mucho tiempo para que las compañías hagan uso de las nuevas tecnologías existentes.

Desde cierto punto de vista se piensa que aquellas compañías que sean capaces de reconocer la importancia creciente que tiene el software dentro de su posición competitiva, y sean capaces de enfrentarse a ella de manera agresiva tendrán mayor oportunidad de sobrevivir.

[Wegner ,1984] señala al respecto :

"La mayoría de la gente que trabajamos en el campo de la tecnología de software pensamos que los años 1980's son más excitantes que los años 1970's, y que los años 1990's serán más excitantes aún. El vivir en un periodo de rápidos cambios tecnológicos proporciona la oportunidad y la responsabilidad de dar forma al futuro. Esto requiere de ser más innovadores y correr mayores riesgos que en un periodo de estabilidad. El progreso sólo puede llevarse a cabo corriendo riesgos, tomando decisiones difíciles e invirtiendo en el futuro."

Para estudiar la posibilidad de éxito de un proyecto de reutilización es necesario contar con información de costos de proyectos anteriores, lamentablemente muchas organizaciones no tienen tales registros.

Existen diversos modelos de costos de software, en particular el presentado por Fairley-Bollinger-Pfleeger* es un modelo de costos enfocado a la reutilización. En el trabajo de [Hooper-Chester,1991] mismo se puede obtener una amplia referencia bibliográfica al respecto.

El punto de mayor dificultad dentro de un modelo de costos que involucre reutilización es medir el impacto económico al crear componentes reutilizables en un proyecto específico y evaluar el beneficio futuro que proporcionarán dichos componentes en proyectos futuros.

La inversión que se haga en reutilización no debe ser vista de manera mediocre, por ejemplo, pensando en la simple depuración de sistemas existentes; debe estar bien planeada de modo que en el futuro sea más fácil encontrar componentes que crearlos, lo cual nos llevara a lograr proyectos con menor esfuerzo y menor riesgo. Los costos y los beneficios de los nuevos proyectos pueden ser comparados con proyectos anteriores (el primero obviamente sin reutilización) a los cuales se les llama proyectos base.

Es natural pensar que para que los costos con reutilización sean razonables deben comportarse de la siguiente manera : el costo del desarrollo del componente reutilizable, así como el de todas las adaptaciones necesarias que se le practiquen para su reutilización debe ser menor que el costo de desarrollar el software necesario para cubrir todas las necesidades sin emplear reutilización.

Una forma de poder observar más claramente el comportamiento del costo del proyecto con reutilización es considerar a algunos participantes como fabricantes de software y a otros como consumidores de modo que sean más visibles las operaciones de intercambio que se llevan a cabo.

Actualmente la mayoría de los autores reconocen la importancia de la reutilización en todas las fases del ciclo de vida de desarrollo de software, haciendo notar que las actividades más tempranas del proceso de desarrollo pueden proporcionar mayores beneficios si se reutilizan. Como pautas a considerar sobre los aspectos financieros de la reutilización [Hooper-Chester,1991] mencionan las siguientes :

[i] Establecer un mecanismo para acumular datos financieros relativos a la producción y mantenimiento de software, incluyendo actividades de reutilización.
[ii] Proporcionar herramientas de modelos de costo, para la toma de decisiones acerca de la reutilización.
[iii] Considerar dentro de los modelos de costo proyectos múltiples.
[iv] Establecer mecanismos para compartir el costo de desarrollar componentes reutilizables a través de múltiples proyectos.

*Fairley, R. Bollinger, T & Pfleeger, S.L., "Final Report: Incentives Reuse of Ada Components." Vols. 1-5, George Mason University 1989.

I.6.11 PERSPECTIVAS DE LA REUTILIZACIÓN.

A continuación se enumeran algunas de las formas potenciales de mejorar la productividad de software.

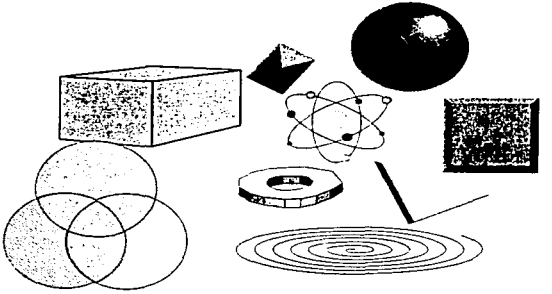
- Nuevos ambientes de computadora que ayuden al desarrollo de software.
- Nuevos elementos de hardware que creen diferentes modos de acceso a otras computadoras dentro de una red.
- El uso de enfoques administrativos enfocados hacia la productividad de software.
- Reutilización de código.
- Reutilización del diseño.
- Generadores de aplicaciones.
- Sistemas de Transformación y de especificación formal.

BIBLIOGRAFÍA PARA EL CAPÍTULO I

[Biggerstaff,1987]	Biggerstaff, T.J. & Ritcher, CH. "Reusability Framework, Assesment And Directions." IEEE Software, Vol 4, # 2 Marzo 1987.
[Prieto,1985]	Prieto Díaz, R. "Clasification of Reusable Modules." IEEE Software, Vol. 4, # 1 1985.
[Horowitz,1984]	Horowitz ,E. & Munson, J.B. "An Expansive View of Reusable Software." IEEE Transactions on Software Engineering, Vol. SE-10 # 5, septiembre 1984.
[Wegner,1984]	Wegner, P. "Capital-Intensive Software Technology." IEEE Software, Vol. 1 # 3, julio 1984.
[Biggerstaff-Perlis,1989]	Biggerstaff, T. J. & Perlis, A. J. "Software Reusability Vol. I: Concepts And Models." Reading Mass., Addison-Wesley; 1989.
[Meyer,1988]	Meyer, B. "Object Oriented Software Construction." Prentice Hall United Kingdom. 1988.
[Hooper-Chester, 1991]	Hooper, James W. & Chester, Rowena O. "Software Reuse : Guidelines and Methods." Plenum Publishing Corporation, U.S.A. 1991.
[Fischer,1989]	Fischer, Gerhard "Human Computer Interaction in Software: Lessons Learned, Challenges Ahead." IEEE Software, enero 1989
[Mellroy,1969]	Mellroy, M. D. "Mass Produced Software Components" Software Engineering Concept and Techniques." Brusels 39. Belgium : Petrochelli/Charter 1969.
[Curtis,1989]	Curtis, B. "Cognitive Issues in Reusing Software Artifacts." Software Reusability Applications and Experience, Vol II., Biggerstaff & Perlis, A.J. ACM. Press, Addison Wesley , Reading Mass. 1989.
[Selby,1989]	Selby, R. W. "Quantitative Studies of Software Reuse." Software Reusability Vol. II Application and Experience." Ed. T. J. Biggerstaff and A. J. Perlis 1989.
[Biggerstaf-Perlis,1989b]	Biggerstaf , T .J. & Perlis,A. J. "Software Reusability: Aplication and Experience." ACM Press, Addison Wesley, Vol. II Reading Mass. 1989.
[Tracz, 1990]	Tracz, W. "Where does reuse start.", ACM, software Engineering notes 15(2) 1990.

CAPÍTULO II

MODELO ORIENTADO A OBJETOS Y REUTILIZACIÓN.



"Durante el desarrollo del entendimiento de fenómenos complejos, la herramienta más poderosa con que cuenta el intelecto humano es la abstracción. La cual surge del reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real y la decisión de concentrarse en estas similitudes e ignorar momentáneamente las diferencias."

C. A. R. Hoare.

INTRODUCCIÓN.

La capacidad de clasificar a los elementos del mundo real en objetos con características propias se da a muy temprana edad durante la infancia (cerca del año de edad). Esta importante capacidad de reconocer objetos fue recientemente adoptado por la ingeniería de software para construir un nuevo paradigma, el cual ofrece una solución para el problema de reutilizar.

II.1 CONCEPTOS

II.1.1 OBJETO

El término objeto se ha vuelto muy común dentro de la comunidad computacional. Se ha utilizado para hacer referencia a un componente de software que tenga un estado escondido para el cliente y un conjunto de capacidades u operaciones mediante las cuales es posible cambiar el estado interno del objeto. La abstracción de procesos y la abstracción de procedimientos son ejemplos de orientación a objetos, aunque, el término es asociado comúnmente con el lenguaje Smalltalk desarrollado en Xerox PARC; el cual impone una gran abstracción a través de una jerarquía de herencias.

Primeramente un objeto representa un elemento de la realidad, más adelante, el concepto de objeto se extiende para incluir modelos abstractos de ciertos elementos reales e ideas o conceptos. De manera más formal definiremos un objeto como un elemento individual, identificable, ya sea real o abstracto, con un rol definido dentro del dominio de un problema.

[Yourdon,1990] nos da la siguiente definición :

"Un objeto es la encapsulación de una abstracción : una encapsulación de atributos y servicios exclusivos sobre aquellos atributos, una abstracción del dominio del problema que representa una o más ocurrencias de algo dentro del este dominio."

Una manera de decidir que elementos de nuestro medio son o no son objetos es preguntarnos si cuentan con tres características que tienen todos los objetos. Estas características son : estado, comportamiento e identidad. La estructura y el comportamiento de objetos similares se define en una clase común a dichos objetos de modo que estos objetos son instancias o ejemplares de clases, a lo largo de este trabajo consideraremos como equivalentes los conceptos de objeto e instancia.

Estado.

Hemos dicho que los objetos presentan un estado, pero, para que ésto se vea de forma más sencilla pensemos en un automóvil estándar, sabemos que para llevar a cabo un cambio de velocidades es necesario que primero pisemos el pedal del embrague, si no lo hacemos en este orden obtendremos una respuesta desagradable, de ahí obtenemos la idea de que el automóvil no siempre esta disponible para hacer cambios de velocidades, es decir, que a veces está en estado de rechazar esos cambios y a veces no. Todo lo anterior nos lleva a definir de un modo más formal lo que se entiende por estado de un objeto. El estado de un objeto involucra todas las propiedades (normalmente estáticas) del objeto, además de todos los valores (normalmente variables) de cada una de esas propiedades.

El hecho de que un objeto tenga un estado implica que existe un lugar ocupado por el objeto ya sea físico o en la memoria de una computadora.

Comportamiento.

La existencia aislada de los objetos no sería de ningún interés, lo interesante de los objetos es que actúan sobre ellos mismos y sobre otros objetos, por lo tanto, podemos decir que tienen comportamiento.

El comportamiento de un objeto se refiere a como un objeto actúa y reacciona en términos de sus cambios de estado y paso de mensajes. Una operación es una acción que un objeto ejecuta sobre otro objeto en espera de una reacción. Para nosotros los términos operación y mensaje serán considerados equivalentes, en los lenguajes orientados a objetos y basados en objetos se llama métodos a las operaciones que los clientes pueden llevar a cabo sobre los objetos.

Al hablar de operaciones nos parece pertinente mencionar los cinco tipos de operaciones sobre objetos que comúnmente están disponibles, estas son :

Modificación .- La que se refiere la modificación o alteración del estado del objeto.
Selección .- Que se refiere al acceso del estado de un objeto sin alterar su estado.
Iteración.- Se refiere al acceso de cada uno de los elementos del objeto
Construcción.- Crear un objeto o inicializar su estado.
Destrucción.- Liberar el estado de un objeto y/o destruirlo.

Los subprogramas libres a diferencia de los métodos también son operaciones, sólo que estos pueden construirse a partir de los métodos. Los subprogramas así como los métodos, constituyen el protocolo del objeto, es decir todo el comportamiento permitido para el objeto además de la visión interna y externa del objeto mismo.

Identidad. Identidad es la propiedad de un objeto que lo distingue de otros objetos.

En ocasiones resulta un tanto difícil el desligar los conceptos de identificador del objeto y el objeto mismo, cuando nosotros definimos que ciertas variables tienen el mismo tipo de los objetos de una clase aún no hemos creado esos objetos, lo ultimo se logrará en el momento en que llamemos al método constructor asociado al objeto que deseamos crear. Una vez que se ha creado al objeto, ¿qué pasa si a dos variables que designan objetos diferentes las hacemos que tomen el mismo valor?, sucederá que nos encontraremos con un problema de comportamiento estructural, es decir, una sola estructura en este caso el objeto podrá ser refererenciado mediante dos nombres diferentes con la consecuencia de que este objeto pudiera ser alterado sin que se dieran cuenta de ello los objetos que utilizan el otro nombre.

Los objetos dependiendo de sus acciones pueden jugar uno de tres papeles a saber :

Actor: Objeto que puede operar otros objetos, pero, nunca puede ser operado por otros objetos
Servidor: Objeto que nunca opera a otros objetos, sin embargo, él siempre es operado por otros objetos.
Agente: Objeto que tanto puede ser operado por otros objetos como puede operar a otros objetos.

Siempre que un objeto pasa un mensaje a otro objeto con el que tiene relación, ambos objetos deben estar sincronizados de alguna manera, de allí surge una clasificación de los objetos dependiendo del número de flujos de control de los que dependan.

II.1.2 CLASE

Existe un concepto que es útil para agrupar a los objetos que tienen comportamientos o protocolos iguales, este concepto es el concepto de clase, una clase es un conjunto de objetos que comparten una estructura y un comportamiento comunes.

La vista externa de la clase esta constituida por una interfaz que ayuda a remarcar el aspecto abstracto de la clase, ya que esconde los secretos del comportamiento y la estructura de la clase. Por otro lado llamamos vista interna a la implementación de una clase. La vista interna o implementación de una clase esta formada en parte por la implementación de la interfaz de la clase, dicha interfaz puede dividirse en tres partes, la pública, la protegida y la privada.

La primera es aquella parte de la interfaz que está a disposición de todos los clientes, la segunda es otra parte de la interfaz que tiene la característica de que no esta disponible para otras clases a menos que estas sean subclases de ella y por último la parte privada sólo puede ser accesada por la clase misma.

El conjunto de clases de Smalltalk tiene la estructura de un árbol cuya raíz es la clase objetos, la cual, contiene todas las propiedades que contienen los objetos, es decir, ellos heredan las propiedades de esta superclase. Diremos que existe una diferencia entre heredar atributos de una superclase e importarlos de un medio ambiente global, los atributos heredados pueden ser llamados por los objetos que en particular tengan acceso a ellos, pero los usuarios de tales objetos no pueden hacer uso de esos atributos. Lo anterior representa un avance en cuanto a algunos lenguajes (Ada por ejemplo) al impedir el acceso indiscriminado.

CLASIFICACIÓN.

La clasificación es el medio mediante el cual podemos ordenar el conocimiento. Al momento de diseñar, la tarea más difícil es la de elegir cuales serán los objetos y cuales serán las clases que formarán parte de nuestro sistema, para ayudarnos en la toma de tales decisiones debemos valernos del reconocimiento y de la invención.

El reconocimiento de los elementos a utilizar lo llevaremos a cabo al identificar las abstracciones y mecanismos que forman parte de nuestro problema y a través de la invención generalizaremos las abstracciones y los mecanismos que regularán el comportamiento de los objetos. A través de la clasificación podemos identificar la generalización, especialización y agregación a la vez que nos ayuda a tomar decisiones sobre la modularización. Todos estamos enterados de que a lo largo de la historia de la ciencia ha habido diversos métodos de clasificación para diversas ramas del conocimiento humano, desde la clasificación de las especies, de los elementos, etc. las cuales han utilizado diversos criterios para llevar a cabo la clasificación que han requerido de un gran esfuerzo intelectual, a través del tiempo y el razonamiento esas clasificaciones se han ido mejorando por medio de su revisión y reconsideración. Una vez creadas las clases podemos reconsiderarlas para derivar o factorizar las clases existentes, y obtener otras clases o unir a varias clases en una clase por medio de la composición.

Los enfoques que se han dado a través de la historia para el problema de la clasificación han sido en general tres los cuales mencionamos a continuación :

Categorización clásica.
Agrupamiento conceptual.
Teoría de prototipos.

El primero se basa en el siguiente enunciado:

"Todas las cosas que tiene una ó más características en común forman una categoría, tales propiedades son necesarias y suficientes para definir una categoría."

Este enfoque como podemos observar establece como criterio de clasificación la relación de propiedades para descubrir la similitud.

El agrupamiento conceptual, es un enfoque más moderno que el enfoque de categorización clásico y deriva del intento de explicar la representación del conocimiento. En este enfoque, las clases son generadas al crear descripciones de las clases y después tratando de ver que elementos caben en que categorías.

Los anteriores métodos de clasificación son suficientes la mayoría de las veces para enfrentar un problema complejo, sin embargo, existen situaciones en las que dichos enfoques son inadecuados. Para este tipo de problemas es apropiado utilizar el método llamado teoría de prototipos, el cual, surge del estudio en el campo de la psicología cognoscitiva que se fundamenta en la idea de que existen abstracciones que no tienen propiedades o conceptos bien delimitados ya que no existe ningún conjunto de propiedades que compartan ciertos objetos. Por consiguiente, lo que se hace es crear un prototipo que sí contenga ciertas características con las que cumplan algunos objetos, después, para incluir a otros objetos se debe ir modificando el prototipo a manera de que vaya englobando a los demás objetos.

Los enfoques de clasificación comentados arriba son la base del análisis y del diseño orientado a objetos, durante el análisis se busca modelar el mundo identificando las clases y los objetos que forman parte del vocabulario del dominio del problema y durante el diseño se inventan las abstracciones y mecanismos que proporcionan el comportamiento que el modelo requiere.

Generalmente las clases y objetos surgen de cosas tangibles como automóviles, casas, sensores de datos ,etc., de puestos o roles dentro del dominio del problema tales como empleado, profesor, regulador, etc., de eventos o sucesos en el dominio del problema tales como interrupción, cancelación, vencimiento, etc. o interacciones que se dan entre los objetos.

Desde el punto de vista del modelo de datos podemos tener como clases a gentes, lugares, cosas, organizaciones, conceptos y eventos.

[Yourdon,1991] sugieren otro conjunto de fuentes potenciales de objetos:

Estructuras.	Basada en las relaciones "clase de" y "parte de".
Otros sistemas.	Sistemas externos con los que la aplicación interactua.
Dispositivos	Dispositivos con los que la aplicación interactua
Eventos	Eventos históricos que la aplicación debe recordar.
Papeles o roles de los usuarios	Los roles que juegan los usuarios dentro del sistema.
Lugares	Lugares físicos tales como oficinas, y lugares importantes para la aplicación.
Unidades organizacionales	Grupos a los que los usuarios pertenecen.

¿Como descubrir las abstracciones claves para el modelado?

Este proceso involucra dos procesos el descubrir y el inventar. Durante el descubrir tendremos que buscar los elementos reconocidos por los expertos del área, después tendremos que inventar otras clases y objetos que aunque no estén dentro del problema nos servirán para regular el comportamiento de los primeros. Una vez que hayamos creado y obtenido algunos abstracciones como candidatos para nuestro sistema debemos hacernos preguntas como las siguientes : ¿Como se crean los objetos de esta clase?, ¿Pueden estos objetos ser copiados o destruidos?, ¿Que operaciones son válidas para este objeto?, ¿Cual es el significado de hacer tal o cual operación con el objeto o clase?. De esta manera podremos evaluar que tan claros están algunos conceptos y si es necesario reconsiderarlos o analizarlos más a fondo.

Por otro lado, el definir objetos como abstracciones del mundo real nos ayuda a tener un mayor entendimiento del problema y a modelar el contexto del problema, en este caso el contexto del sistema es un indicador de que tanto abarcaremos del dominio del problema con el sistema automatizado.

¿Como identificar los mecanismos de nuestro modelo?

Un mecanismo es una estructura en la cual los objetos trabajan juntos para proporcionar el comportamiento necesario para satisfacer los requerimientos de un problema. Mientras las abstracciones clave reflejan el vocabulario del dominio del problema, los mecanismos constituyen el alma del diseño.

En nuestro próximo capitulo nos dedicaremos a presentar una metodología sencilla de aplicar para hacer un análisis orientado a objetos (OOA) en el mismo capítulo describiremos también aspectos del diseño orientado a objetos (OOD) que deberán tomarse en cuenta al momento de construir un sistema. Dedicaremos el capítulo cuarto a tratar los aspectos comunes que existen en la práctica de la programación orientada a objetos al utilizar un determinado lenguaje.

RELACIONES ENTRE CLASES.

La mayoría de los lenguajes de programación basados en objetos y orientados a objetos cuentan con mecanismos para tratar a las relaciones entre clases. Las relaciones entre clases que explicaremos aquí son la herencia, las relaciones de uso, las relaciones de ejemplificación y las relaciones de metaclases.

Relación de herencia.

Esta relación es apropiada en el momento de decidir si un lenguaje es orientado a objetos en cuyo caso esta relación es soportada por el leguaje ó solo es basado en objetos en cuyo caso no contaria con esta relación.

La herencia es una relación entre clases donde una clase (clase paterna o superclase) comparte la estructura o comportamiento definido en una (herencia simple) o más (herencia múltiple) clases, con otra clase (clase hija o heredera). El concepto de herencia con anulación se refiere a que en algunos lenguajes es posible heredar algunas propiedades de la superclase y otras propiedades no, de modo que estas propiedades puedan ser anuladas o modificadas para proporcionar comportamientos diferentes.

Herencia significa que el comportamiento y los datos asociados con una clase hija son siempre una extensión de las propiedades asociadas con su clase (o clases) paterna(s), desde el punto de vista de que una clase tiene un comportamiento más especifico que el de una superclase, la subclase representa una especialización de la clase paterna ala vez que la clase paterna es una generalización de la subclase.

Regularmente una clase tiene dos tipos de clientes, los ejemplos o instancias y las subclases, es recomendable definir partes apropiadas de la interfaz para cada uno de ellos de ahí viene la idea de contar con tres partes en la implementación de la interfaz.

Relaciones de uso.

Existen dos formas distintas de que una clase haga uso de otra, la primera, se da cuando la interfaz de una clase hace uso de una clase (lo cual sucede cuando definimos que una clase maneja elementos de otra clase), para poder manipular los elementos de la clase servidora es necesario que la clase actor tenga acceso a la interfaz apropiada de la clase servidora sobre la que está actuando. Para lograr lo anterior, es necesario que la clase usada sea visible para todos los clientes que pretenden utilizarla. La segunda forma de utilización de un clase por otra es cuando la implementación de una clase hace uso de otra clase, en cuyo caso la clase servidora está escondida como parte de la implementación de la clase actora. Muchas veces se comete el error de confundir la relación de herencia con la de uso, una regla que nos puede ayudar a discernir que tipo de relación debemos utilizar es la de pensar que si una clase es más que la suma de sus partes entonces es mejor valernos de relaciones de uso que de relaciones de herencia.

II.1.3 HERENCIA.

Por herencia nos referimos a la propiedad que tiene una subclase o clase hija de tener acceso a los datos y al comportamiento (métodos) asociados con una clase paterna o supclase. La herencia tiene la propiedad de ser transitiva.

Cada clase se pueden organizar en una estructura de herencia jerárquica. Una subclase heredará atributos de una superclase. Una superclase abstracta es una clase que sólo se usa para crear subclases y para la cual no hay ejemplares directos.

El paradigma de la programación orientada a objetos (OOP) de Smalltalk se ha extendido a otros lenguajes, así, contamos con C++ y Object Pascal entre otros. La herencia puede utilizarse para enriquecer clases o para hacerlas más especializadas.

La estructura de herencia sobre los componentes de software refleja el crecimiento del conocimiento dentro de una base de datos, en vez de empezar a partir de nada en la creación de cada componente.

Actualmente existen muchos lenguajes que han sido llamados lenguajes orientados a objetos, para diferenciar entre éstos y los que no son en realidad orientados a objetos es necesario aclarar que características deben contener para ser orientados a objetos.

Los lenguajes orientados a objetos cuentan con las siguientes características a saber :

a) El elemento principal de los componentes de software está representado por los objetos. El estado interno de los objetos es inalterable entre operaciones sucesivas.

b) Las operaciones y estados de los objetos pueden ser heredados de las respectivas superclases, de modo que la nueva funcionalidad puede incrementarse por medio de la definición dentro de anteriores definiciones funcionales.

Las clases y objetos deben organizarse en bibliotecas que permitan la fácil localización y utilización de las clases paternas o abstractas.

El uso de bibliotecas promueve la reutilización del conocimiento durante la creación de nuevo conocimiento. Creemos que todo diseñador que pretenda crear una biblioteca de componentes debe tener en cuenta las siguientes cuestiones :

1.- Que clase de componentes podrá contener la biblioteca.
2.- Cual será la granularidad y el dominio de aplicación de la biblioteca.
3.- Que clase de clientes (programas o gente) harán uso de las bibliotecas.
4.- Como se cargarán, ligarán e invocarán los componentes.
5.- Como se crearán, insertarán y recuperarán dichos componentes.
6.- Como se expresarán las relaciones entre los componentes.
7.- Que clase de conocimiento es necesario que tengan los programadores para que puedan desarrollar programas con las bibliotecas de componentes.

Las bibliotecas de componentes pueden clasificarse ya sea por medio de las clases de componentes que contienen o por el tipo de clientes (analistas, programadores, personal administrativo) que hacen uso de ellas y cuyas expectativas son diferentes.
El diagrama de la Figura 8 [Martin,1993] nos ayuda a clasificar componentes de los diferentes lenguajes de programación, proporcionando una guía incompleta para la clasificación de lenguajes cuyo componente principal son las funciones y los subprogramas, pero proporcionando una clasificación estructurada simple para lenguajes con abstracción procedural y de datos, a la vez que arroja luz sobre las propiedades de los componentes.

La separación existente entre abstracción de proceso y de datos se basa en la idea de que la abstracción de datos es pasiva mientras que la abstracción procedural no lo es. Todas las operaciones dentro de la abstracción de datos pueden accesarse de manera pasiva en cualquier

momento, a diferencia de que en la abstracción procedural se controla cuando y donde aceptar una operación.

TAXONOMIA DE LOS COMPONENTES DE SOFTWARE (CON EJEMPLOS)

FIG. 8 TAXONOMIA DE LOS COMPONENTES DE SOFTWARE

La herencia significa que el comportamiento y los datos asociados con una subclase son siempre una extensión de lo que presenta una superclase. Una subclase comprende todas las propiedades de todas las clases antecesoras a ella y otras más. Por otro lado, recordemos que una subclase presenta un comportamiento más especializado que una clase paterna y es este enfoque bilateral de una clase hija como extensión y como expansión de una clase paterna lo que en muchas ocasiones provoca confusión, aunque, es también fuente de la potencia que proporciona esta técnica.

BENEFICIOS DE LA HERENCIA.

Reutilizacion de software

Este tema es el que planteamos aquí, y en el desarrollo del trabajo completo mostraremos como este mecanismo da amplias posibilidades de reutilización de código; aunque, de ninguna manera estamos afirmando que sea el único. Por el contrario diremos que existen muchos otros métodos mediante los cuales se puede mejorar la reutilización de código.

Al momento de heredar el comportamiento de una superclase ya no es necesario reescribir el código que proporciona ese comportamiento.

Otro beneficio es que el código cada vez se vuelve más confiable ya que al estar utilizando el código en más ocasiones se da oportunidad de que se depure más rápidamente (como todos sabemos no existe ningún programador que escriba código totalmente exento de errores, a menos que este no tenga la más mínima complejidad)

Compartición de código.

La compartición de código se presenta en diversas partes al utilizar programación orientada a objetos. Los usuarios o proyectos independientes pueden emplear las mismas clases, en otro nivel de compartición tenemos el ejemplo en el que el programador utiliza una clase existente para definir el comportamiento de una nueva clase.

Consistencia de la interfaz.

Cuando una o más clases heredan de una misma clase paterna, aseguramos el que todas tendrán el mismo comportamiento. De este modo es más sencillo garantizar que las interfases para objetos similares en realidad sean similares y además, se comporten de manera similar (en el supuesto de que no estemos hablando de herencia con anulación, ya que en esta forma de herencia es posible anular el comportamiento heredado de las superclases).

Componentes de software.

Una de los principales hechos que motivaron el desarrollo del modelo orientado a objetos fue precisamente el deseo de hacer posible la idea del componente de software, actualmente ya existen muchas bibliotecas que hacen realidad aquel propósito.

Modelado rápido de prototipos.

Cuando ya contamos con una biblioteca lo suficientemente completa para desarrollo a nuestra disposición, el tiempo empleado anteriormente en el desarrollo puede emplearse en conocer la parte del sistema que es nueva y desconocida.

De esta manera podemos llevar a cabo una forma de trabajo conocida como modelado rápido de prototipos o programación exploratoria, la cual ha sido sugerida por múltiples autores como un medio para conocer mas a fondo el dominio del problema y para ayudar a los usuarios a hacer más explícitos sus requerimientos.

Polimorfismo.

Anteriormente el software producido que se podía encontrar había sido desarrollado bajo el enfoque ***Top-Down***, es decir se escribían las rutinas de más bajo nivel, y a partir de ellas se construía otro conjunto de subrutinas de más alto nivel y sobre ellas otras cada vez más abstractas. No es difícil descubrir que a medida que las rutinas se van haciendo más abstractas la portabilidad de estas va decreciendo. Las rutinas de más bajo nivel eran más portables a otros proyectos mientras que las rutinas de más alto nivel ya estaban muy ligadas con aplicaciones específicas.

El llamado polimorfismo dentro de los lenguajes de programación permite a los programadores crear componentes reutilizables de alto nivel ajustables a diferentes aplicaciones. Dentro de los lenguajes de programación se llama objeto polimórfico a una variable o argumento a la cual le está permitido almacenar valores de diferente tipo durante la ejecución de un programa. De manera similar se llama función polimórfica a aquella función que permite que sus argumentos sean polimórficos. Aunque, es más sencillo escribir funciones

polimórficas en los lenguajes que cuentan con asignación dinámica de tipos, la mayoría de los lenguajes con chequeo estricto de tipos acepta el polimorfismo a través de la sobrecarga.

El poder del polimorfismo estriba en el poder de escribir algoritmos de alto nivel que puedan ser reutilizados una y otra vez con diferentes abstracciones de bajo nivel. En los lenguajes con asignación dinámica de tipos todos los objetos son potencialmente polimórficos, mientras que en los lenguajes con asignación estática de tipos el polimorfismo ocurre cuando la clase (estática) declarada de una variable intenta recibir a una clase (dinámica) del valor de la variable, en C++ ésto ocurre a través del manejo de apuntadores.

Ocultación de información.

Este aspecto ayuda a reducir la interconexión entre sistemas de software, ya que al utilizar una clase definida por otro programador sólo es necesario conocer su interfaz y de ninguna manera es necesario conocer las sutilezas que encapsula.

EL COSTO DE LA HERENCIA.

Es muy difícil que se nos presente un beneficio que no traiga consigo involucrado costo de algún tipo; bueno, pues tal es el caso de la programación orientada a objetos y no nos queda más que ser sinceros y tratar el tema sin mayor preámbulo.

Velocidad de ejecución.

Como es de esperarse una herramienta de propósito general en la mayoría de los casos no podrá competir en velocidad con una herramienta de propósito específico. Del mismo modo, los métodos desarrollados para soportar el que sean heredados deben ser más robustos y por lo tanto generalmente resultan más lentos.

Creemos que este momento es oportuno para mencionar que en la mayoría de las ocasiones los programadores tienen poca idea acerca de como se usa el tiempo de ejecución de los programas, lo recomendable a este respecto es desarrollar un sistema de trabajo, luego controlarlo para darse cuenta donde se está usando el tiempo de ejecución y mejorar tales secciones, en lugar de dedicar una cantidad excesiva de tiempo en preocuparse por la eficiencia en una fase temprana del proyecto. Al respecto recordemos el comentario de William Wulf que viene a propósito :

"Se cometen más pecados de computación en nombre de la eficiencia (sin lograrla necesariamente) que por cualquier otra razón, incluyendo la simple estupidez"

Tamaño del programa.

El uso de cualquier biblioteca de software conlleva una sanción sobre el tamaño del programa, aunque este tamaño puede representar un costo considerable, éste es cada vez menos importante debido a que el costo del almacenamiento cada vez se vuelve más barato.

Tiempo de procesamiento adicional del paso de mensajes.

Se ha comentado en muchas ocasiones que el paso de mensajes es más costoso que la simple invocación a un procedimiento. Sin embargo, con frecuencia una excesiva preocupación por el costo del paso de mensajes atiende el detalle y descuida el conjunto. Para algunos a menudo el mayor costo es marginal; tal vez una o dos instrucciones adicionales de lenguaje ensamblador y una sanción de tiempo de 10% aproximadamente (el tiempo de procesamiento adicional de paso de mensajes será más largo en lenguajes ligados dinámicamente que en lenguajes ligados estáticamente). De la misma manera que otros costos de la herencia éste debe ser evaluado junto con los muchos beneficios que proporciona el modelo orientado a objetos.

Complejidad de los programas.

Frecuentemente se ha pensado que la programación orientada a objetos es una solución al problema de la complejidad del software, no obstante, el abuso de la herencia constituye un elemento que proporciona mucha complejidad al software desarrollado ya que sin duda, el entender el flujo de control de un programa requiere de múltiples exploraciones a los niveles altos y bajos de la jerarquía de herencia, a lo cual se le ha denominado problema tipo yo-yo.

Herencia en Smalltalk.

A través del concepto de clases llegamos al concepto de herencia, la herencia como ya hemos dího dicho es fundamentalmente : un mecanismo que simplifica la definición de los componentes de software que son similares a aquellos previamente definidos.

El beneficio de la herencia dentro de un lenguaje orientado a objetos es que el diseñador/implementador puede utilizar las clases que ha creado alguna otra persona y únicamente añadirle comportamiento especializado que se adecue más a sus necesidades, de otro modo sucedería que el programador diría *"De acuerdo tu tienes algunas rutinas que trabajan de manera similar a lo que yo deseo, pero no funcionan de acuerdo a mis necesidades así que yo construiré las mías propias"*.

De esta manera observamos que la herencia es un mecanismo poderoso para incrementar la reusabilidad, ya que captura de manera explícita los aspectos comunes. El mecanismo de herencia consiste de cuatro aspectos importantes:

1.- Cada clase, comparte estructuras de datos definidas en sus superclases.
2.- Cada clase comparte los métodos definidos en sus superclases.
3.- Cada clase puede añadir elementos a la estructura de la clase.
4.- Cada clase puede añadir o extender (o anular en el caso de Smalltalk) métodos heredados.

Es importante tomar en cuenta que en los niveles superiores de la jerarquía de clases se encuentran los atributos y métodos comunes, mientras que a medida que se desciende de nivel éstos van siendo extendidos al añadir nuevas capacidades.

Después de haber observado un poco el comportamiento del mecanismo de la herencia dentro de Smalltalk hay algunos puntos importantes que conviene resaltar.

1.- Los atributos y los servicios exclusivos definidos sobre esos atributos son tratados como un todo.
2.- Las interfases entre objetos quedan definidas completamente mediante los métodos (servicios) que son visibles a otros objetos. El único camino mediante el cual un objeto puede conocer los datos escondidos dentro de otro objeto es mediante el envío de un mensaje apropiado al objeto, que corresponda a un servicio proporcionado por el objeto.
3.- La estructura de clasificación en la herencia proporciona diferentes maneras para modelar el espacio del problema y ganar poder mediante el uso expreso del uso común en los niveles superiores y la expansión de datos y procesamiento en los niveles inferiores.

II.2 ANTECEDENTES DEL MODELO ORIENTADO A OBJETOS

II.2.1 LOS SISTEMAS COMPLEJOS.

Cuando tratamos de entender un problema complejo tenemos de nuestro lado un enfoque particular que nos facilita su comprensión; este enfoque nace del estudio de las características comunes con que cuentan dichos sistemas y son:

1) Generalmente la complejidad muestra una estructura jerárquica, en la cual el sistema complejo esta compuesto de subsistemas interrelacionados, éstos a su vez exhiben la misma estructura hasta que se llega al nivel más bajo de los componentes elementales.

[Booch,1991] señala que el hecho de que los sistemas complejos manifiesten una estructura jerárquica es lo que nos permite entender o describir a los sistemas y a sus partes.

2) Dependiendo del interés del observador los sistemas serán descompuestos en los componentes primitivos que el considere necesario (abstracción).

[Booch,1991] llama a los sistemas jerárquicos descomponibles, debido a que pueden ser descompuestos en partes identificables, y los llama casi descomponibles cuando sus partes no son totalmente independientes.

3) Las relaciones dentro de los componentes son generalmente más fuertes (cohesión) que las relaciones entre los componentes (acoplamiento), este hecho tiene el efecto de envolver a los componentes junto con las relaciones más dinámicas, aislándolos de las relaciones menos dinámicas que involucran la interacción entre componentes.

4) Los sistemas jerárquicos están compuestos usualmente de unas pocas clases diferentes de subsistemas en varias combinaciones y arreglos.

5) Un sistema complejo que funciona es invariablemente la evolución de un sistema simple que trabajaba ... un sistema complejo diseñado desde el principio nunca funciona y no puede ser remendado, para que funcione, es necesario volver a empezar haciéndolo con un sistema simple que trabaje correctamente.

A medida que nos vamos adentrando en la abstracción, los objetos considerados una vez complejos se van convirtiendo en los objetos primitivos de sistemas más complejos.

El estudio de los sistemas complejos ha proporcionado formas de caracterizarlos y ha descubierto abstracciones y mecanismos comunes que facilitan mucho nuestro conocimiento acerca de ellos.

Podemos notar que dentro de la descomposición de un sistema complejo puede haber dos clases de jerarquías. Aquella en la que se descomponen los sistemas en sus partes y relaciones y la otra forma de descomponerlos es viendo a sus componentes de manera aislada y observando cuales de estos tienen comportamientos semejantes y pueden ser agrupados en clases.

Como instrumento para comprender el mundo real el hombre utiliza tres métodos de organización, los cuales penetran en su pensamiento:

1) Estableciendo la diferencia entre objetos particulares y sus atributos,
2) Estableciendo la distinción entre el todo y las partes que le componen y
3) La formación y la distinción entre clases diferentes de objetos.

La experiencia ha mostrado que es esencial analizar a los sistemas desde ambas perspectivas, es decir, desde su jerarquía de clases como desde su jerarquía de partes, las cuales llamaremos jerarquía de clases y jerarquía de objetos respectivamente.

Al combinar los conceptos de estructura de clase y de objetos junto con los cinco atributos de los sistemas complejos, tenemos que virtualmente todos los sistemas presentan esta forma canónica.

Por otro lado el revelar las estructuras de clases y de objetos del sistema nos ayuda a organizar la redundancia del sistema en estudio, sino lo hiciéramos así nos veríamos obligados a duplicar el conocimiento que tuviéramos de cada objeto. Los sistemas complejos más exitosamente diseñados son aquellos cuyo diseño encierra una buena estructura de clases y objetos que comprende los cinco atributos de los sistemas complejos.

II.2.2 *¿COMO SE HA ATACADO A LOS SISTEMAS COMPLEJOS PARA HALLAR SOLUCIONES ?*

El pilar principal de nuestra solución nace del antiguo dicho ***"Divide, et impera"****. Así, cuando diseñamos un sistema es menester dividirlo en partes cada vez más pequeñas para poderlas comprender, es decir, subsistemas menos complejos, después avanzaremos en el sentido inverso e iremos abstrayendo sus características internas para verlas como piezas de un sistema más grande.

Esta descomposición es muy común y muchos de nosotros la hemos llevado a cabo por medio de la descomposición algorítmica bajo el dogma del diseño estructurado. En este enfoque cada módulo representa un paso para el logro de un proceso más complejo. En contraste se sugiere un método de descomposición alterno, la descomposición orientada a objetos, la cual, en

* Divide y Venceras

primer lugar descompone al problema en las partes clave que tienen lugar en el dominio del problema; es decir; se toma como objetos de nuestra descomposición a los elementos actores o sobre los que se actúa en un momento determinado. De esta manera que nuestra descomposición ve al sistema como un conjunto de objetos con un comportamiento definido que interactuan por medio de mensajes para llevar a cabo una tarea específica.

Lo anterior no quiere decir que alguno de los métodos es el mejor, ya que por ejemplo, el enfoque de descomposición algoritmica arroja luz sobre el orden de los eventos, mientras que el enfoque de descomposición basado en objetos hace énfasis en los agentes que actúan o sobre los que actúan los métodos invocados por otros objetos.

Aunque no es posible estudiar un sistema desde los dos enfoques al mismo tiempo, se recomienda primero un análisis orientado a objetos y posteriormente basados en ese conocimiento hacer un análisis algoritmico para poner orden sobre los eventos.

El enfoque orientado a objetos como veremos más adelante proporciona algunas ventajas sobre el enfoque orientado a procedimientos; por ejemplo; la descomposición orientada a objetos permite la creación de sistemas más pequeños debido a que aprovecha los mecanismos comunes de cada parte, proporcionando economía de expresiones; los sistemas orientados a objetos son más flexibles al cambio y a la evolución ya que su diseño está basado en formas intermedias más estables, además, se disminuye el riesgo al construir sistemas de software complejos, ya que evoluciona de lo sencillo a lo complejo.

II.2.3 MÉTODOS DE ANÁLISIS.

En este momento platicaremos de manera general acerca de los cuatro enfoques más importantes para analizar los requerimientos, primero definamos a que nos referimos con el término análisis.

"Análisis es el estudio de un problema, antes de tomar alguna acción."

Analizar es el proceso de extraer las necesidades de un sistema (lo que el sistema debe hacer para satisfacer al cliente). Aquí hablaremos un poco acerca de los enfoques que se ha dado al análisis desde hace casi una década, estos enfoques cuentan cada uno con características especiales que los hacen importantes en ciertas ocasiones, por lo tanto, deben ser considerados como herramientas disponibles de gran ayuda en el momento indicado.

DESCOMPOSICIÓN FUNCIONAL.

La descomposición funcional puede ser caracterizada por que divide al problema en pasos y subpasos, una ecuación que representa este enfoque es la siguiente:

Descomposición funcional = Funciones + Sub-funciones + Interfases funcionales

la estrategia fundamental aquí, consiste en seleccionar anticipadamente los pasos y subpasos del procesamiento de un nuevo sistema basándonos en la experiencia previa. El punto clave de este enfoque es "¿qué procesamiento es necesario en el nuevo sistema?", después, el analista se enfocará en las interfases funcionales requeridas. Este enfoque que relaciona al mundo real

(dominio del problema) con funciones y subfunciones requiere que el analista descubra las sutilezas del sistema y documente las funciones y subfunciones que el sistema proporcionará.

Aunque este tipo de análisis no es malo, tiene el defecto de que hace difícil el que el analista y el usuario establezcan los requerimientos reales del sistema, aunque debe mencionarse que el análisis orientado a objetos se vale de él al descomponer un servicio muy grande y/o complicado. Este enfoque funcional proporciona una gran volatilidad debido al constante cambio de la capacidad funcional de los sistemas, como veremos más adelante, este tipo de características funcionales cambiantes deben quedar encapsuladas dentro de los métodos.

ENFOQUE HACIA EL FLUJO DE DATOS.

Otro método es el enfoque hacia el flujo de datos, más conocido como "análisis estructurado", al cual representaremos con la siguiente ecuación :

Análisis estructurado = Flujo de datos y de control + Transformación de datos y control+ Terminadores + Miniespecificaciones + Diccionario

Con ayuda de este método el analista relaciona el espacio del problema con diagramas de flujo y burbujas. Este mapeo requiere que el analista y el cliente sigan el flujo de datos del problema y lo representen por medio de diagramas de flujo. El AOO también se vale de este enfoque para identificar los servicios y los pasos dentro de esos servicios que proporcionaran los objetos.

El defecto de este enfoque es que proporciona una descomposición en pasos (abstracción procedural) y no en una descomposición que muestre su refinamiento gradual, a la vez que en ocasiones es muy difícil elegir la manera de agrupar las burbujas para representar el problema en niveles superiores de abstracción. Además, el tamaño del diccionario de datos pude crecer de manera inmisericorde.

Este enfoque aún tiene un fuerte contenido funcional, por lo mismo no es flexible y no resiste satisfactoriamente las embestidas del cambio de requerimientos. Por otro lado se ha observado que este enfoque no da gran importancia a las estructuras de datos utilizadas ni facilita el paso del análisis hacia el diseño.

MODELADO DE INFORMACIÓN.

La principal herramienta de modelado de información es el diagrama de entidad relación el cual ha evolucionado a partir de los modelos semánticos de datos, el modelar al mundo en datos ha ayudado a capturar el contenido del espacio del problema.

He aquí una ecuación para ayudar a identificar este modelo :

Modelado de información = Objetos + Atributos + Relaciones + Super tipos/Subtipos + Objetos asociativos

La vieja estrategia rezaba : Desarollar una lista identada de atributos. Poner los atributos de paquetes de objetos, añadir las relaciones, refinar con ayuda de supertipos/subtipos y objetos asociativos y después normalizar.

La nueva estrategia sigue siendo la misma excepto que el primer paso es encontrar objetos en el mundo real y describirlos por medio de sus atributos.

Este enfoque es un enfoque parcial ya que no atiende a aspectos de relevancia tales como :

1) **Servicios:** Requerimientos de procesamiento para cada objeto, encapsulados y tratados junto con los atributos como un todo.
2) **Herencia:** Representación explícita de atributos y servicios comunes.
3) **Mensajes:** Interfases bien definidas entre objetos.
4) **Estructura:** Estructura de clasificación y estructura de ensamblaje como métodos fundamentales de ordenación.

MODELO ORIENTADO A OBJETOS

El termino orientado a objetos es representado mediante la siguiente ecuación :

Orientado a Objetos = Objetos (Encapsulación de objetos y atributos exclusivos de algo en el dominio del problema) + Clasificación + Herencia + Comunicación con mensajes

El AOO esta construido sobre los mejores conceptos del modelado de información y los mejores conceptos del los lenguajes orientados a objetos. Del primero provienen los atributos, las relaciones, la estructura y la representación de un objeto del segundo provienen la encapsulación de atributos y servicios exclusivos y el tratamiento de estos como un todo, la estructura de clasificación y la expresión explícita de las características comunes por la herencia. El AOO se basa en la aplicación uniforme de :

a) Métodos de organización
b) Comunicación con mensajes
c) Clasificación del comportamiento

Las diferencia entre los diversos métodos de análisis están resumidas en el cuadro de la fig. 9.

	PRINCIPIOS					
	Abs. procedural	Abs. de datos	Encapsulación	Clasif. y herencia	Métodos de organización	Cat. de comporta-miento
MÉTODOS						
Descomposición funcional	X					
Información del flujo de datos	X			X	X	X
Modelado de información				X	X	X
Orientación a objetos	X	X	X	X	X	X

FIG. 9 MÉTODOS DE ANÁLISIS

Como ya dijimos, la estructura de clases agrupa a los objetos dentro de clases que muestran comportamientos similares, permite manejar de manera conveniente las características comunes de los objetos mientras que la estructura de objetos ilustra como los diferentes objetos actúan entre sí a través de patrones de interacción llamados mecanismos o métodos.

II.2.4 EVOLUCIÓN DE MODELO ORIENTADO A OBJETOS

SURGIMIENTO DEL MODELO ORIENTADO A OBJETOS

Es importante mencionar que las ideas básicas del modelo orientado a objetos no son nuevas, ni el modelo orientado a objetos es un modelo revolucionario que rompa con los elementos de los anteriores modelos de diseño, al contrario, los conceptos aquí manejados surgieron con el desarrollo de los lenguajes de programación, para responder a las crecientes expectativas que se tuvieron, conforme el ámbito de la ingeniería de software crecía junto con las necesidades, tamaño y complejidad de los sistemas. Es por esto que se le considera un enfoque evolutivo.

Los sistemas de software cada vez más grandes y complejos han creado la necesidad de mejorar las técnicas de la ingeniería de software, de la misma forma, han motivado el desarrollo de lenguajes más expresivos que han avanzado desde los lenguajes imperativos (dicen que hacer) hasta los lenguajes declarativos (describen las abstracciones del dominio del problema)

[Booch,1991] presenta una estructura de clasificación, basándose en el trabajo de Wegner[1], en la que ha clasificado algunos lenguajes de programación populares de acuerdo a las características que introdujeron en diferentes generaciones.

Los lenguajes de la primera generación fueron usados básicamente para aplicaciones científicas y de ingeniería. En consecuencia su vocabulario fue casi completamente matemático.

Estos lenguajes dieron un paso adelante hacia el espacio del problema ya que liberaron al programador de muchos de los apuros que traía consigo la programación en lenguaje ensamblador o lenguaje máquina; pero por otro lado, estos lenguajes se volvieron más dependientes de la máquina donde eran implantados. El ejemplo típico de los lenguajes de la primera generación es FORTRAN.

La dirección que tomaron los lenguajes de la segunda generación iba encaminada a la abstracción algorítmica, en ese momento las máquinas se iban a convirtiendo en más poderosas y económicas, por lo tanto, se pensó que más y más clases de problemas debían ser automatizados. Esta nueva generación de lenguajes se acerco un poco más al espacio del problema y se alejó más del problema de la dependencia de la máquina.

Con la introducción de los transistores y la posterior tecnología de los circuitos integrados los costos de los equipos se redujeron sobremanera mientras que inversamente su potencia aumentaba, de tal modo que, la mayoría de los problemas podía ser resuelto. Sin embargo, se creo una demanda de más clases de datos, de tal suerte que, lenguajes tales como Algol-60 y después Pascal evolucionaron para soportar el concepto de abstracción de datos y una vez más

[1] Wegner, P. "Research Directions in Software Technology". Cambridge, MA., The MIT Press 1980.

está generación de lenguajes de alto nivel se acerco más al dominio del problema al mismo tiempo que se aleja más de la dependencia de la máquina.

Durante la década de los años 1970 hubo un gran desarrollo de los lenguajes de programación pero fueron tantos que muchos de ellos desaparecieron dejando sus avances como herencia de los lenguajes que les sucedieron.

EL CAMINO DE LA REUTILIZACIÓN DE COMPONENTES DE SOFTWARE HACIA EL MODELO ORIENTADO A OBJETOS.

Aunque en el pasado esta meta fue muy buscada, no por lo mismo fue alcanzada satisfactoriamente. Como ya veremos, esto se debe a la estrecha interconexión que presentan los programas que fueron creados de manera convencional; con dichos métodos es difícil extraer código que pueda ser reutilizable en otro proyecto no relacionado, ya que cada parte del código puede tener interdependencias que sean el resultado de definiciones de datos o ser dependencias funcionales.

Recordemos que para poder hablar acerca de la reutilización de componentes es necesario distinguir entre los principales componentes de software que han surgido de distintos paradigmas y enfoques de la ingeniería de software.

En los años 1980 el desarrollo se enfocó en la abstracción de procesos y en la computación distribuida, los cuales, son considerados como nuevas perspectivas para reutilizar componentes de software. Por otro lado las bibliotecas de proyectos han sido una herramienta poderosa en el manejo de un proyecto ya que han sido de gran utilidad durante la fase de mantenimiento. Pero, el orden cronológico en que se han presentado estas diversas abstracciones puede servirnos para comprender la evolución y el destino que tiene la reutilización, al mismo tiempo que puede reflejarnos un poco del medio ambiente y de las necesidades que prevalecían en el momento en que se crearon estas diversas abstracciones.

TIPOLOGÍA DE LOS LENGUAJES DE PROGRAMACIÓN DE LA PRIMERA Y PRINCIPIOS DE LA SEGUNDA GENERACIÓN.

Los bloques de construcción de esta generación de lenguajes fueron los subprogramas los cuales accesaban a datos globales, lo que denota una endeble estructura física, ya que provocaban dependencia de los subprogramas sobre los datos, es decir, que cuando se llevaban a cabo modificaciones a los sistemas, éstos terminaban con una gran cantidad de acoplamientos cruzados y flujos de control alterados que hacían peligrar la confiabilidad del sistema completo a la vez que reducían la claridad de la solución.

TIPOLOGÍA DE LOS LENGUAJES DE PROGRAMACIÓN DE FINES DE LA SEGUNDA Y PRINCIPIOS DE LA TERCERA GENERACIÓN.

Hasta mediados de la década de los 1960 fue que se considero a los programas como parte importante entre el problema y la computadora, como [Shaw.1984] señala ***"La primera abstracción de software, ahora llamada abstracción procedural, surgió directamente de esta vista práctica del software ... Los subprogramas fueron inventados antes de 1950, pero no fueron apreciados como abstracciones en ese momento ... en lugar de eso fueron vistos***

como dispositivos guardadores de tareas ... muy pronto, los subprogramas fueron considerados como una forma de abstraer funciones de los programas".

Esto tuvo tres consecuencias de importancia : primera, los lenguajes fueron modificados para soportar paso de parámetros; segunda, se establecieron los fundamentos de la programación estructurada y tercera, surgieron los métodos de diseño estructurado, los cuales ofrecieron una guía para los diseñadores que trataban de construir grandes sistemas usando a los subprogramas como bloques de construcción básicos.

TIPOLOGÍA DE LOS LENGUAJES DE PROGRAMACIÓN DE FINES DE LA TERCERA GENERACIÓN.

Los lenguajes de principios de la tercera generación tuvieron avances significativos pero aún no resolvieron el problema de la programación en grande. Los lenguajes de fines de la tercera generación atacaron ya este problema de manera directa al considerar que los proyectos grandes requerían de equipos grandes de desarrollo, así que presentaron como solución el módulo compilado en forma separada, que en sus primeras concepciones no fue más allá de un contenedor de datos y subprogramas, pero, después incluyeron en los lenguajes chequeos entre las llamadas y encabezados de los subprogramas para detectar inconsistencias.

TIPOLOGÍA DE LOS SISTEMAS DE PROGRAMACIÓN BASADOS EN OBJETOS Y ORIENTADOS A OBJETOS.

La importancia que tiene la abstracción de datos para manejar la complejidad por medio del uso de procedimientos es correcta al usarse con la abstracción de las operaciones; pero, no es correcta cuando lo que se quiere abstraer es la descripción de objetos abstractos. En muchas aplicaciones la complejidad de los objetos dato a ser manejados contribuye substancialmente en la complejidad total del problema, lo cual, tiene como consecuencias: primero la evolución hacia el diseño dirigido a los datos; el cual proporciona un enfoque disciplinado para los problemas en que había que hacer abstracciones de datos en lenguajes orientados algoritmicamente y segundo impulsa el enfoque de las teorías hacia el concepto de tipos de datos.

La motivación de estas consecuencias desembocó en la creación de lenguajes tales como Simula primeramente y posteriormente Smalltalk, Object-Pascal, C++, CLOS (Common Lisp Object Sistem) y Ada en los cuales el bloque básico (representado por el módulo) no esta construido como un subprograma sino que representa una colección lógica de clases y objetos. Por lo tanto se ha definido a la programación orientada a objetos de la siguiente manera:

"La programación orientada a objetos es un método de implementación en el cual los programas están organizados como colecciones cooperativas de objetos, cada uno de los cuales representa un ejemplar de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unida por relaciones de herencia."

En este momento es necesario que distingamos a un lenguaje orientado a objetos de uno que no lo es bajo nuestro concepto están aquellos lenguajes y sólo aquellos que cumplen con las siguientes características :

a) Soportan objetos que son abstracciones de datos con una interfaz de operaciones nombradas y un estado local escondido.
b) Los objetos tienen un tipo asociado (que es su clase).
c) Los tipos (clases) pueden heredar atributos de los supertipos (superclases).

Hace falta aquí, definir la diferencia entre algunos conceptos, los cuales mencionaremos a continuación :

"El diseño orientado a objetos es un método de diseño que comprende el proceso de descomposición orientado a objetos y una notación para describir los modelos lógico y físico así como los modelos estático y dinámico del sistema bajo diseño."

"El análisis orientado a objetos es un método de análisis que examina los requerimientos desde una perspectiva de las clases y los objetos encontrados en el vocabulario del dominio del problema"

"Un estilo de programación es una forma de organizar los programas sobre la base de un modelo conceptual de programación y sobre un lenguaje apropiado para hacer claros los programas escritos en este estilo".

Existen cinco estilos de programación a saber :

i) Orientado a procedimientos (Algoritmos)
ii) Orientados a objetos (Clases y objetos)
iii) Orientados a lógica (Metas y cálculo de predicados)
iv) Orientados a reglas (Reglas si ... entonces)
v) Orientados a restricciones (Reglas invariantes)

II.3 EL MODELO ORIENTADO A OBJETOS.

El modelo orientado a objetos es frecuentemente considerado un nuevo paradigma, aclaremos lo que el historiador Thomas Kuhn define como paradigma en su libro "The structure of scientific revolutions" un paradigma es ***"Un conjunto de teorías, estándares y métodos que juntos representan una forma de organizar el conocimiento; es decir; una forma de ver el mundo"*** en este sentido el modelo orientado a objetos es un nuevo paradigma, pero, analicemos por que razones esta técnica debe considerarse como una herramienta importante.

Es justo que dentro de la ingeniería de software se procediera de la misma manera que se procede en la elaboración de otros objetos tales como edificios o automóviles, es decir, ensamblando cierta variedad de piezas o componentes prefabricados, en vez de fabricar cada componente partiendo de nada.

El mecanismo utilizado para resolver un problema por medio del modelo orientado a objetos consiste en encontrar agentes (objetos) apropiados, a los cuales se les pueda pasar un mensaje que contenga una petición para resolver el problema con ayuda de un método (algoritmo).

En este modelo la acción se inicia con la transmisión de un mensaje a un agente responsable de la acción, el cual contiene una petición; acompañada de todos los argumentos necesarios

para cumplirla. El objeto que recibe el mensaje acepta la responsabilidad de llevar a cabo la acción indicada, para la cual, se valdrá de algún método.

Es oportuno mencionar la ocultación de información en cuanto al paso de mensajes, ya que, el cliente que hace la petición no necesita saber de que manera será satisfecha ésta. Existe un par de diferencias importantes entre una llamada a un procedimiento y el envío de un mensaje a un objeto. En principio los dos reciben una petición, pero; en el paso de mensajes existe un receptor designado y la interpretación o selección del método que se utilice para satisfacer la petición depende del agente que recibe la petición. Generalmente el receptor específico para un mensaje dado no se conoce sino hasta el tiempo de ejecución, por lo que se habla de un enlace tardío o dinámico. Además, es muy importante notar que el comportamiento se estructura en términos de responsabilidades.

Todos los objetos son ejemplos de una clase, el método invocado por un objeto en respuesta de un mensaje es determinado por la clase del receptor.

La organización de los procedimientos se debe llevar a cabo bajo una jerarquía de procedimientos. Existe un principio que dice que el conocimiento de una categoría más general es aplicable también a las categorías más específicas llamado herencia.

La selección del método que se utiliza para satisfacer la petición contenida en un mensaje se llama enlace de métodos y se inicia en la clase del objeto receptor, si no se encuentra un método apropiado se lleva la búsqueda a la superclase inmediata, en caso de no encontrar un método apropiado se continua del mismo modo hasta que se encuentre el método o se agota la cadena de superclases. En el primer caso se ejecuta el método; en el último caso, se emite un mensaje de error.

Aunque el compilador no puede decidir que método invocará en tiempo de ejecución, en muchos lenguajes sí podrá determinar si habrá un método apropiado y enviará un mensaje de error en tiempo de compilación si no lo encuentra.

Otra característica de los sistemas de software desarrollados bajo métodos convencionales, que los colocan entre los sistemas más complejos desarrollados por la gente, es su alto grado de interacción, es decir, la dependencia de una parte del código de una sección con código de otra sección. Por ejemplo, considerese el uso de variables globales y apuntadores.

El problema de lidiar con la complejidad al que nos hemos enfrentado los programadores durante mucho tiempo, ha sido manejado a través de diversos mecanismos. El más importante ha sido la abstracción, esto es, la capacidad para encapsular y aislar la información de diseño y ejecución. En cierto sentido las técnicas orientadas a objetos son un producto evolutivo natural de una larga progresión histórica que va de las funciones y los procedimientos a los módulos, a los paquetes, a los tipos de datos abstractos y por último a los objetos.

Procedimientos y funciones : Fueron uno de los primeros mecanismos de abstracción que se usaron ampliamente en los lenguajes de programación, permitían concentrar en un mismo lugar las tareas que se ejecutaban en forma repetida o que se realizaban sólo con ligeras variaciones, para evitar la duplicidad de código. Además proporcionaron la primera posibilidad de ocultar información, ya que, un procedimiento podía ser desarrollado por un programador y este procedimiento podía ser invocado por otros programadores, los cuales no

necesitaban conocer las sutilezas de la implantación, únicamente, necesitaba conocer la interfaz necesaria para su utilización. Esta abstracción resolvió de manera aislada el problema, ya que, en los lenguajes en que se implemento aún dejo el problema del acceso irrestringido a los datos.

Módulos : Con el uso de procedimientos subsistía el problema de que había partes que debían ser públicas, por ejemplo las interfases de nuestros procedimientos, mientras que había otros datos internos a los procedimientos que deseábamos que no se compartieran, con la ayuda de los módulos podemos dejar como publicas a las interfases y podemos accesarlas desde cualquier otro modulo, mientras que la parte privada es accesible sólo dentro del módulo.

[Booch,1991] hace referencia al trabajo de Parnas[2], quien popularizó la noción de módulo, describió los siguientes dos principios para su uso apropiado:

1.- Uno debe proporcionar al usuario toda la información necesaria para ser correctamente el módulo y nada más.

2.- Uno debe proporcionar al implantador toda la información necesaria para completarlo y nada más.

Los módulos ofrecieron otra solución parcial al problema con una dosis mayor de abstracción y un manejo efectivo de la ocultación de la información, pero, todavía no permiten la creación de ejemplares, la cual es la capacidad para hacer copias múltiples de las áreas de datos.

Tipos de datos abstractos : Un tipo de dato abstracto es aquel que es creado por el programador y puede ser utilizado como un tipo nativo del lenguaje, es decir, cuenta con un dominio bien definido de los valores que puede aceptar y con un conjunto de operadores o funciones de mapeo que lo pueden utilizar como argumento. Aunque, está relacionado con el concepto de módulo, el concepto de dato abstracto es más teórico, para construir un tipo de dato abstracto debemos ser capaces de llevar a cabo cierto tipo de operaciones las cuales se enlistan a continuación :

a) Exportar una definición de tipo.
b) Proporcionar un numero de operaciones que puedan usarse para manipular los ejemplares del tipo.
c) Proteger los datos asociados con el tipo de tal manera que se pueda operar con ellos mediante las operaciones previstas.
d) Crear múltiples ejemplos del tipo.

Los paquetes encontrados en el lenguaje Ada son intentos directos de alcanzar los requerimientos teóricos de la abstracción de datos.

La programación orientada a objetos se basa en la idea de los tipos abstractos de datos pero añade además innovaciones importantes en lo que toca a la compartición de código y la reusabilidad.

[2] Parnas, D. L., Clemens, P. C. y Weiss, D. M. "Enhancing Reusability With Information Hiding." ITT Proceedings of the Workshop on Reusability in Programming. 1983.

El modelo orientado a objetos esta construido sobre un conjunto cuyos elementos son llamados colectivamente modelo de objetos, el cual encierra siete principios que a decir verdad no son nuevos, lo que si es cierto es que dentro del modelo de objetos estos elementos están unidos de manera cohesiva.

Debe quedar claro que el diseño orientado a objetos es, como ya lo dijimos, un nuevo paradigma y que en realidad es fundamentalmente diferente de los enfoques de diseño estructurales ya que en si constituye una nueva manera de ver el mundo y requiere de un modo distinto de pensar cuando se requiere descomponer un sistema. Como ya veremos posteriormente los productos de software construidos bajo este modelo están muy lejos de las arquitecturas conseguidas bajo el paradigma del diseño estructurado.

Estas diferencias de diseño y programas surgen al considerar la diferencia existente entre la programación estructurada y la programación orientada a objetos. Aquí hablaremos acerca de lo que es el diseño orientado a objetos y en que difiere de otros métodos de diseño, a través de los siete elementos que lo caracterizan, mostrados en la Fig. 10.

FIG. 10 ELEMENTOS DEL DISEÑO ORIENTADO A OBJETOS

De los principios del modelo orientado a objetos que mencionamos anteriormente, hablaremos a continuación.

II.3.1 ABSTRACCIÓN

Se refiere a tomar en cuenta las características principales que distinguen a un objeto de los demás objetos; enfocando la atención en los aspectos relevantes para un propósito dado; va muy relacionado con el principio de encapsulación al enfocarse en la vista externa del objeto sin entrar en los detalles de su implementación.

Dentro de este principio se encuentra el concepto de abstracción procedural el cual hace mención de que cualquier operación que lleve a cabo un efecto bien identificado puede ser tratada por los usuarios como una entidad simple sin importar que esta operación sea llevada a cabo mediante una secuencia de operaciones menores. Aunque, este tipo de abstracción no es la forma de abstracción principal dentro del análisis orientado a objetos juega un papel importante en el momento en que se está describiendo a objetos individuales.

Otro concepto importante dentro de la abstracción es el de abstracción de datos el cual define un tipo de datos en términos de las operaciones que aplica a objetos de un tipo con la restricción de que los valores de los objetos sólo pueden ser vistos y modificados mediante el uso de tales operaciones.

Existe además otro tipo de abstracciones que aquí mencionaremos ordenadas desde la más usual hasta la menos usual.

Abstracción de entidad es aquella en la que un objeto representa un modelo de una entidad dentro del dominio del problema.

Abstracción de acción es aquella en la cual un objeto proporciona un conjunto de operaciones que ejecutan la misma clase de función.

Abstracción de máquina virtual se llama a aquella que agrupa operaciones usadas por un nivel superior de control, o por operaciones que usan un nivel inferior de propiedades.

Abstracción coincidente se llama a la que agrupa un conjunto de operaciones que no tiene relación entre ellas.

Al reflexionar sobre estas definiciones podemos ver que éstas nos ayudan a comprender el comportamiento de un objeto, aquello que es también llamado el protocolo del objeto y que se refiere a las operaciones que los clientes (objetos que usan los recursos de otro objeto) pueden ejecutar sobre él, así como las operaciones que este puede ejecutar sobre los otros objetos.

ABSTRACCIÓN DE FUNCIONES.

De los diferentes métodos de abstracción que han sido creados por la mente humana como propuestas, cada una representa un elemento de construcción diferente y esta relacionado con un método de programación, primero trataremos la abstracción de funciones y después de la abstracción de procedimientos.

La abstracción de funciones puede especificarse mediante la relación entre sus entradas y salidas por medio de una función donde para cada x existe una y sólo una f(x) que depende únicamente del valor de la entrada x, f representa la transformación, la cual permanece oculta bajo el principio de ocultación de la información. Ver fig. 11

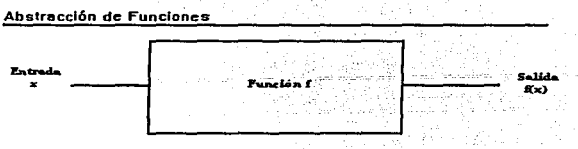


FIG. 11 MODELO DE ABSTRACCION DE FUNCIONES

ABSTRACCIÓN DE DATOS.

La abstracción de datos en contraste con la abstracción de funciones, además, de esconder la implementación, también esconde datos, de modo que esta abstracción funciona como un autómata finito, es decir, para cada entrada x, el valor y que es la salida no depende únicamente de la entrada x sino que $y = f(x,s)$ depende también del estado interno s del autómata, al mismo tiempo el nuevo estado $s'=g(x,s)$ se obtiene por medio de una función de transición g. Ver Fig. 12

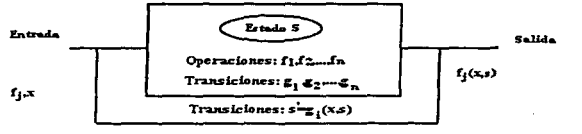


FIG. 12 MODELO DE ABSTRACCION DE DATOS

Al pensar en abstracción de datos es necesario hacer las siguientes consideraciones :

1. ¿Qué clase de recursos deben proporcionar las abstracciones de datos a los usuarios? ¿Debe proporcionar operaciones, tipos variables o algún conjunto de éstos?

2. ¿Qué reglas deben gobernar los permisos otorgados a los diferentes usuarios?

La abstracción de funciones enfatiza la reutilización de funciones de datos variantes, mientras la abstracción de datos enfatiza la reutilización de objetos de datos por medio de varias operaciones que se les pueden aplicar.

ABSTRACCIÓN DE PROCESOS.

Las funciones son operaciones abstractas, la abstracción de datos es abstracción de variables. Los procesos son computadoras abstractas.

Los procesos abstractos son similares a los datos abstractos en el sentido de que también tienen estados internos y que éstos estados cambian dependiendo de una transformación a otro estado. Se diferencían de los datos abstractos en que cuentan con un hilo de control independiente que determina el orden en que las operaciones serán llevadas a cabo. Existen dos clases de procesos, los procesos concurrentes y los procesos distribuidos.

a) Los procesos concurrentes pueden comunicarse entre si a través de estructuras de datos compartidos que son comunes a ambos (segmentos de memoria compartidos, semáforos, colas de mensajes, conductos, conductos nombrados, señales o receptáculos Berkeley), las cuales requieren de mecanismos para controlar el acceso concurrente.

b) Los procesos distribuidos no cuentan con datos compartidos y por lo tanto únicamente pueden comunicarse por medio del paso de mensajes, éstos no necesitan ningún medio de protección para los datos debido a que no existen accesos concurrentes.

El acceso concurrente a los datos compartidos puede controlarse por medio de monitores los cuales manejan una cola de acceso y controlan la ejecución secuencial de las operaciones.

Entre los puntos a considerar en el diseño de interfases para procedimientos abstractos están los siguientes :

1.- ¿Existen diferencias esenciales entre las necesidades de las interfases de datos y las de la abstracción de procesos?

2.- ¿Qué relación debe existir entre las interfases de los procesos y de los componentes para soportar la evolución del programa durante el desarrollo y la ejecución de los sistemas distribuidos?

Procesos distribuidos.

El modelo de procesos distribuidos es más simple que el de los procesos concurrentes con datos compartidos. Además, proporciona un paradigma para el desarrollo de componentes de software que es más poderoso que la abstracción de datos.

En el caso de adoptar este modelo es necesario decidir si se utilizará concurrencia interna dentro del proceso distribuido. Ya que la concurrencia interna hace que el computador comparta un área común de memoria entre múltiples procesos y en ocasiones mejora la eficiencia de ciertos tipos de procesos también es necesario considerar que se incrementa la complejidad del lenguaje de programación en raz+on a que se hace necesario un mecanismo de control para controlar de manera ordenada a los procesos concurrentes. De este modo, se necesitan dos mecanismos uno para manejar externamente los procesos distribuidos y otro para manejar la concurrencia interna.

Los procesos distribuidos con concurrencia interna son llamados procesos distribuidos concurrentes y son diferentes de los procesos distribuidos secuenciales.

Otra decisión importante de diseño para los procesos distribuidos es la manera en que se recuperará el sistema de las fallas.

Los mecanismos de recuperación programables proporcionan control al usuario sobre la recuperación, mientras que, los mecanismos transparentes liberan al usuario de esta responsabilidad dejándola al sistema

II.3.2 ENCAPSULACIÓN.

Encapsulación es el proceso de ocultar los detalles de un objeto que no contribuyan como características esenciales, la encapsulación cuando es llevada a cabo de manera inteligente localiza las decisiones de diseño que probablemente cambien a medida que el sistema evoluciona.

Este concepto es relativo, ya que lo que puede estar escondido en un nivel de abstracción puede ser visto como vista exterior en otro nivel de abstracción.

II.3.3 MODULARIDAD.

Frecuentemente es útil dividir un programa cuando se quiere reducir su grado de complejidad, pero, esta división es más provechosa cuando se hace para crear límites bien definidos y documentados dentro del programa, los cuales junto con sus interfases serán de gran ayuda para la comprensión del programa.

B. Liskov.

"La modularización consiste en dividir un programa en módulos que puedan ser compilados separadamente, pero que estén conectados con otros módulos."

D. Parnas.

"Las conexiones entre los módulos son las consideraciones que los módulos hacen unos de los otros"

M. Zelkowitz.

"Debido a que la solución puede ser desconocida cundo inicia el diseño, la descomposición en módulos más pequeños puede ser muy difícil"

Algunas anotaciones que es importante mencionar respecto de los módulos son por ejemplo : el que debido a que la meta principal de la descomposición en módulos es la de reducir el costo del software permitiendo que los módulos sean creados y revisados de manera independiente, éstos deben guardar una estructura simple para poderse entender completamente; además de que debe ser posible el cambiar la estructura de cualquier módulo sin que ésto afecte la estructura de ningún otro módulo. Otra recomendación importante es que los detalles del sistema que sean más inestables deben quedar encapsulados en módulos

separados, las únicas consideraciones que deben hacerse acerca de un módulo en particular son aquellas que se considere improbable que cambien.

Cada una de las estructuras de datos debe pertenecer y ser privada de un módulo en particular; aunque, debe ser accesible para uno o más módulos diferentes por medio de llamadas a programas del módulo dueño de la estructura de datos. En resumidas cuentas podemos decir que la modularidad es la propiedad de un sistema de estar descompuesto en un conjunto de módulos cohesivos, los cuales son independientes.

II.3.4 JERARQUÍA.

Después de haber hecho abstracciones, es necesario recordar que estas abstracciones generalmente pueden ordenarse de manera jerárquica, el entendimiento de tales jerarquías nos ayudara a simplificar los problemas que estemos abordando. [Booch,1991] define la jerarquía como una clasificación u ordenación de abstracciones. Dentro de un sistema complejo las jerarquías más importantes son la estructura de clases y la estructura de objetos.

El hablar de jerarquías o clasificación de abstracciones nos conduce a otro concepto de importancia fundamental: del que ya hemos hablado anteriormente; nos referimos al concepto de herencia el cual puede manifestarse de manera simple o compuesta.

La herencia simple es un elemento esencial de los sistemas orientados a objetos, éste define una relación entre clases, donde una clase comparte el comportamiento o la estructura de una o más clases superiores (llámese herencia simple o herencia múltiple respectivamente) típicamente una subclase o clase hija aumenta o refina la estructura o el comportamiento definido de una superclase o clase paterna. Generalmente se piensa en la herencia como una jerarquía de generalización/especialización. Las superclases son cada vez más generales mientras que las subclases son cada vez más específicas, a manera de explicación podríamos considerar la relación entre los principios de abstracción, encapsulación y herencia.

Por ejemplo la abstracción de datos intenta proporcionar una barrera opaca detrás de la cual los métodos y su estado permanecen escondidos, la herencia requiere de la apertura de una interfaz para accesar dichos métodos y estados sin que le estorbe dicha abstracción.

Para una clase cualquiera existen dos tipos de clientes los objetos que invocan operaciones aplicables a elementos de una clase y las subclases que heredan el comportamiento de dichas clases.

La jerarquía de clases determina un paradigma de capital intenso para la reutilización de datos definidos y de comportamientos adecuados para nuevos componentes.

II.3.5 CLASIFICACIÓN DE TIPOS.

El concepto de clasificación de tipos deriva de la teoría de los tipos abstractos de datos, un tipo es una caracterización precisa de las propiedades estructurales o de comportamiento que comparte una colección de entidades.

La clasificación de tipos constituye un refuerzo para la clase de un objeto, de tal modo que los objetos de diferentes tipos o clases no pueden ser intercambiados o por los menos sólo podrán intercambiarse de manera restringida.

Es necesario mencionar que dentro de los lenguajes de programación existen los que cuentan con un chequeo estricto de tipos, aquellos cuyo chequeo de tipos es débil e incluso los que no manejan tipos. En Object Pascal por ejemplo todos los campos y métodos declarados en la interfaz de la clase son públicos, por lo tanto, su implementación está visible, es decir, Object Pascal no permite encapsular completamente las abstracciones, al mismo tiempo, Object Pascal al ser un lenguaje con cheque estricto de tipos, permite que las declaraciones que invocan a los métodos sean verificadas en cuanto a lo que a consistencia de tipos de datos se refiere en el mismo momento que sucede la compilación, por el contrario, Smalltalk es un lenguaje que no maneja tipos. Las ventajas de manejar una u otra característica pueden resumirse de la siguiente manera :

1) Cuando no se usa un lenguaje con chequeo estricto de tipos, un programa puede fallar de manera misteriosa durante el tiempo de ejecución.

2) En la mayoría de los sistemas, la edición, compilación y depuración son tan tediosas que es indispensable la detección temprana de errores.

3) Las declaraciones de tipos ayudan a documentar los programas.

4) La mayoría de los compiladores pueden generar código objeto más eficiente si se declaran los tipos.

Cabe aclararse que el que un lenguaje ya sea con cheque estricto o débil de tipos no tiene ninguna relación con que tenga un enlace de tipos temprano o tardío, es decir, no tiene relación con que sea enlazado estáticamente o dinámicamente.

Chequeo estricto de tipos quiere decir : que existe una verificación de la consistencia de los tipos de datos en las interfases durante el tiempo de compilación. Por otro lado, establecimiento estático de tipos quiere decir que los tipos de todas las variables y expresiones son fijados al momento de la compilación, enlace dinámico quiere decir que los tipos de todas las variables y expresiones no son conocidos hasta el momento de la ejecución.

Debido a que los conceptos de chequeo estricto de tipos y enlace son ajenos podemos hablar de lenguajes con chequeo estricto de tipos que a la vez son estáticamente ligados (Ada), chequeo estricto de tipos y que soportan enlace dinámico (Object Pascal y C++), chequeo débil de tipos y que soportan enlace dinámico (Smalltalk), etc.

II.3.6 CONCURRENCIA.

Para algunas aplicaciones un sistema puede requerir del manejo de varios eventos simultáneos y posiblemente tales aplicaciones involucren tantas operaciones que excedan la capacidad del procesador. En tales casos es necesario considerar la utilización de procesadores multitarea o de un conjunto de computadores actuando en modo distribuido. Un proceso simple (flujo de control) es la raíz mediante la cual surgen múltiples acciones, dentro de un programa siempre

hay un flujo de control, pero en los sistemas distribuidos puede haber múltiples flujos de control.

[Booch,1991] menciona que algunos autores[3] opinan que las características que proporciona la programación orientada a objetos no son muy diferentes de las proporcionadas por otros lenguajes de programación más comunes, aún con la programación orientada a objetos los problemas tradicionales como los deadlocks, livelocks, starvation, exclusión mutua etc. permanecen ahí.

Lo cierto es que en sus niveles más altos de abstracción la programación orientada a objetos puede disminuir el problema de la concurrencia al esconderla dentro de las abstracciones reutilizables.

[Black,1986] sostiene que ***"El modelo de los objetos es apropiado para un sistema distribuido"***, ya que implícitamente define

1.- Las unidades de distribución y movimiento y
2.- Las unidades que las comunican.

La programación orientada a objetos se enfoca en la abstracción de datos, la encapsulación y la herencia, la concurrencia se enfoca en la abstracción de procesos y en la sincronización de los mismos. El concepto de objeto unifica estos dos puntos de vista : cada objeto (representación de algún ente del mundo real) puede representar un flujo de control (abstracción del proceso), tales objetos son llamados objetos activos.

Dentro de un sistema orientado a objetos podemos ver al mundo como un conjunto de objetos cooperativos, algunos de los cuales están activos y sirven como centros de actividad independiente. A partir de lo anterior se dice que la concurrencia es la propiedad que distingue a los objetos activos de los que no lo son.

Algunos lenguajes cuentan con mecanismos para expresar la concurrencia de procesos, en Ada ese mecanismo es llamado tarea, en Smalltalk se le llama proceso, en C++ se puede crear concurrencia cuando se trabaja sobre un sistema operativo multitarea (como UNIX) en cuyo caso se puede lograr la concurrencia con la llamada *fork()*, en Object Pascal y en CLOS no se tiene estos mecanismos ya que generalmente son utilizados para procesamiento secuencial.

II.3.7 PERSISTENCIA.

Existe un tiempo durante el cual un objeto puede existir, a este tiempo se le llama tiempo de persistencia del objeto, existen diferentes situaciones durante las cuales puede persistir un objeto. Entre ellas tenemos la persistencia de objetos en cálculos intermedios, la de objetos utilizados como variables locales, la de los objetos utilizados como variables locales, la de los objetos que persisten durante la ejecución del programa, la de aquellos objetos que existen en diversas versiones de los programas y la de los objetos que subsisten a los programas.

[3] Lim, J. & Jonshon, R. "Concurrent Objet-Oriented Programming in Act. 1." Object Oriented Concurrent Programming, Ed. Yonezama, Y.M. Tokoro. Cambrigde, MA., The MIT Press 1989.

Las tres primeras son del dominio común de los programas y las otras son del dominio de la tecnología de las bases de datos, el introducir el concepto de persistencia en el modelo orientado a objetos nos lleva al concepto de bases de datos orientadas a objetos.

II.4 PORQUE EL MODELO ORIENTADO A OBJETOS.

Mucha gente que ha trabajado con el paradigma orientado a objetos ha comprobado que proporciona algunas ventajas respecto de otros enfoques, pero otra cosa que también es cierta es que al cuestionar a estas personas acerca de temas tales como ¿por qué prefieren este enfoque?, ¿cuáles son las dificultades que presenta una empresa para cambiar del uso de técnicas convencionales a las técnicas de este nuevo paradigma?, las respuestas han sido muy diversas [Martin,1993], lo que nos lleva a pensar que en esos lugares no se han obtenido todas las ventajas potenciales del modelo orientado a objetos, las cuales es necesario entender y tratar de llevarlas a cabo de manera global y no únicamente parcial.

II.4.1 CARACTERISTICAS DE LAS TÉCNICAS ORIENTADAS A OBJETOS.

LAS TECNICAS ORIENTADAS A OBJETOS CUENTAN CON LAS SIGUIENTES CARACTERISTICAS.

1.- **Cambian la manera de pensar acerca de los sistemas.** La forma de pensar orientada a objetos es una manera natural de pensar en el hombre, desde su más tierna infancia el ser humano aprende a conocer los objetos que le rodean a través del comportamiento que estos presentan, podemos observar a los niños de brazos que tratan de conocer objetos nuevos sacudiéndolos o probándolos tal vez para poderlos situar dentro de las primeras categorías que han formado en su mente. También es notable el resultado de algunas pruebas en las cuales se enseño a un grupo de niños a pensar de acuerdo al modelo orientado a objetos y a otro grupo de experimentados programadores que utilizaban el enfoque procedural, al concluir esta enseñanza se les aplico una prueba de manejo y aplicación de los conceptos aprendidos obteniéndose como resultado que los niños que nunca habían conocido el enfoque procedural tenían mayor facilidad para manejar los conceptos y aplicar el modelo orientado a objetos.

2.- **Los sistemas pueden armarse con objetos existentes. Lo cual proporciona ventajas de reutilización y confiabilidad.**

3.- **La complejidad de los objetos es creciente.** Un objeto puede estar formado de otros objetos y estos a su vez de otras objetos lo que permite manejar una mayor complejidad.

4.- **Las técnicas orientadas a objetos tiene una relación natural con las herramientas CASE.** Ya existen algunas herramientas CASE que soportan el modelo orientado a objetos, pero existen otras que necesitan mejorarse.

5.- **Es más fácil crear sistemas que trabajen correctamente.** Esto se debe en parte a que las clases son diseñadas para ser reutilizables y en parte a que cada clase tiene sus métodos autocontenidos, lo que hace más sencilla la prueba y depuración de estos métodos y de la clase en general.

6.- **La existencia de la tecnología OO-CASE.** Con la ayuda de los generadores de código que producen código para los métodos descritos a través de declaraciones formales , proporciona un potencial de ingeniería de software altamente confiable.

II.4.2 BENEFICIOS DE LA TECNOLOGÍA ORIENTADA A OBJETOS.

Aunque en este trabajo no se trata a fondo el tema de los sistemas basados en generación (que incluye a los generadores de código), si incluiremos algunos de los beneficios obtenidos al utilizarlos, ya que estos se encuentran listados entre los beneficios más importantes de la utilización del modelo orientado a objetos.

REUTILIZACIÓN.

Las clases se diseñan de manera que puedan ser utilizadas en otros sistemas y para mayor aprovechamiento se deben diseñar de manera que sea sencillo adaptarlas a otras necesidades similares. Una de las metas del enfoque orientado a objetos es llevar cabo una reutilización masiva de componentes.

ESTABILIDAD.

Las clases que son diseñadas para ser reutilizables se van volviendo estables, de la misma manera que los microprocesadores se van volviendo estables.

EL DISEÑADOR PIENSA EN TÉRMINOS DE COMPORTAMIENTO Y NO DE DETALLES DE BAJO NIVEL O DE IMPLEMENTACIÓN.

Es muy común que durante una entrevista con un usuario el usuario este explicando los requerimientos del sistema y que el personal de desarrollo de sistemas ya este pensando en la manera de implementar el sistema. Aquí no se presenta ese problema debido a que la encapsulación esconde los detalles o sutilezas y permite la creación de clases complejas que son fáciles de usar. Las clases son como cajas negras el desarrollador las usa y no se percata de lo que contienen.

SE PUEDE CREAR OBJETOS DE GRAN COMPLEJIDAD.

De acuerdo a que los objetos pueden estar compuestos de otros objetos y estos últimos a su vez pueden estar compuestos de otros objetos, podemos crear objetos de alta complejidad.

Por ejemplo una manera de medir la complejidad de un programa es utilizando el Ciclomatic Complexity Metric de McCabe el cual ha sido utilizado para medir la complejidad de programas escritos en FORTRAN, COBOL, PL/1, Pascal y C. la complejidad ciclomática es una medida de la complejidad de la teoría de grafos que se basa en contar el número de rutas individuales contenidas en un programa. Por ejemplo el programa mostrado en al figura 13 tiene tres rutas lógicas posibles dependiendo de las condiciones de las declaraciones IF contenidas.

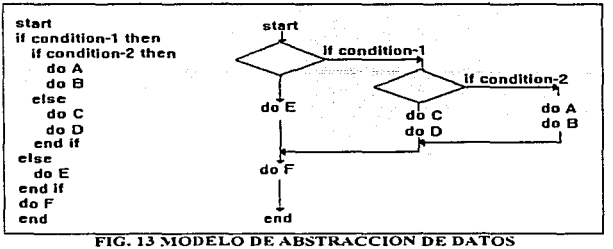


FIG. 13 MODELO DE ABSTRACCION DE DATOS

Este programa tiene una medida de complejidad ciclomática de tres, ya que existen tres posibles rutas a lo largo del programa. La programación orientada a objetos disminuye esta medida de esquemas que típicamente tienen una complejidad de entre 10 y 15 a una complejidad ciclomatica de tres.

La mayoría de los analizadores de programas que miden el número de McCabe revelan que muchos de ellos tienen medida de McCabe de orden 15, mientras que lo mas recomendable es que no excedan de 10.

Un estudio realizado durante el desarrollo del Sistema de Armamento Naval AEGIS, el cual se compuso de 276 módulos con aproximadamente la mitad de ellos excediendo la medida de McCabe recomendable, mostró que aquellos que se excedían tenían un promedio de 5.6 errores por cada 100 líneas de código fuente. En cambio aquellos que estaban debajo del valor 10 tenían un promedio de error de 4.6.[McCabe,1976], la compañía NCR tuvo un proyecto muy grande durante el cual se aplicaron las técnicas orientadas a objetos y obtuvo como resultado una medida de McCabe de 3, mientras que en proyectos anteriores que no incluían al modelo orientado a objetos habían obtenido una medida de McCabe de 10. La razón por la que este indicador de complejidad se ha visto disminuido de manera gradual es que dentro del modelo orientado a objetos cada método es relativamente simple y está autocontenido.

CONFIABILIDAD.

El software creado mediante la composición de clases relativamente estables y probadas es más confiable que el construido desde el principio, además los componentes primarios de estos productos pueden ser clases muy sencillas de construir y fáciles de probar.

VERIFICACIÓN DE PRECISIÓN.

El diseño orientado a objetos apoyado en técnicas formales para la creación de métodos es potencialmente una fuente de generación de código de alta confiabilidad.

INTEGRIDAD.

Las estructuras de datos sólo pueden ser accesadas a través de los métodos específicamente diseñados para ello que están dentro de la clase misma (encapsulación), lo que es de particular importancia cuando se piensa en el modelo cliente-servidor donde algunos usuarios no autorizados podrían intentar accesar al sistema.

FÁCIL PROGRAMACIÓN Y MANTENIMIENTO.

Los programas se construyen con piezas pequeñas, generalmente fáciles de crear, el programador crea un método a la vez y el cambio de estado del objeto después de la aplicación del método es sencillo de entender por si mismo. Por otro lado, el mantenimiento se simplifica ya que la persona encargada de dar mantenimiento al sistema no tiene la necesidad de lidiar con aspectos múltiples al mismo tiempo.

INVENTIVA.

Las personas que han ganado habilidad en el manejo de las herramientas OO-CASE, han descubierto que estas herramientas los motivan a inventar e implementar de manera rápida ideas que les surgen.

MODELADO MAS CERCANO A LA REALIDAD.

Los modelos del análisis orientado a objetos modelan a la empresa o a la aplicación de manera más cercana a la realidad que el análisis convencional. El análisis orientado a objetos se traduce directamente en diseño e implementación, mientras que con las técnicas convencionales el paradigma cambia a medida que se va cambiando del análisis al diseño y del diseño a la programación.

MEJOR COMUNICACIÓN ENTRE LOS USUARIOS Y LOS DESARROLLADORES DE SISTEMAS.

Autores experimentados en la aplicación del modelo orientado a objetos [Martin,1993] opinan que para la gente de negocios es mas sencillo entender el paradigma orientado a objetos,. ya que ellos piensan en términos de eventos, objetos y políticas de negocios que describen el comportamiento de objetos.

MAYOR NIVEL DE AUTOMATIZACIÓN DE BASES DE DATOS.

Las estructuras de datos en las bases de datos orientadas a objetos están ligadas a métodos que toman acciones automáticas. Una base de datos orientada a objetos tiene inteligencia construida en forma de métodos, mientras que una base de datos relacional no. Además las bases de datos orientadas a objetos han demostrado tener mejor desempeño que las bases de datos relacionales para aplicaciones que manejan estructuras de datos más complejas

Aunados a los anteriores beneficios están otros beneficios como interfases más atractivas para el usuario, manejo de imágenes, video y palabras, computación en paralelo entre otras.

II.5 EXPERIMENTO DE LA RELACIÓN ENTRE EL PARADIGMA ORIENTADO A OBJETOS Y LA REUTILIZACIÓN.

Muchas veces los defensores del modelo orientado a objetos han hecho afirmaciones a la ligera acerca de que el paradigma orientado a objetos es mejor en muchos aspectos al paradigma procedural. Aunque las experiencias y la evidencia anécdotales son útiles, dejan lugar a conclusiones engañosas.

En un estudio hecho por [Lewis-Henry,1992] se determinó que el paradigma orientado a objetos es cuantitativamente más benéfico que el enfoque procedural en términos de mantenimiento de software. Un punto de vista interesante recogido en este trabajo es el de que los participantes coincidieron en que las técnicas orientadas a objetos son más difíciles de aplicar.

Entre los beneficios implícitos que trae el paradigma orientado a objetos están los relacionados con la productividad del programador y la reutilización. Las características del enfoque orientado a objetos y las características necesarias para llevar a cabo la reutilización parecen complementarse unas con otras, algunas de estas características que parecen estar hechas a la medida de la reutilización son :

Poder y generalidad.
Encapsulación.
Jerarquía de clases.
Herencia.

En ese trabajo se trato de responder a una serie de preguntas tales como: ¿el enfoque orientado a objetos eleva la productividad de los programadores cuando tratan o no de reutilizar?, ¿la eleva cuando existe una motivación media hacia la reutilización ? o ¿se incrementa cuando existe una fuerte motivación hacia la reutilización? Con el propósito de aclarar estas dudas se diseño un experimento tomando en cuenta las anteriores preguntas. Se definió a la productividad como el inverso proporcional del esfuerzo necesario para producir un producto de software específico, es decir a menor esfuerzo necesario para el desarrollo de un producto mayor productividad..

Se planteo como hipótesis la siguiente :

"Que ambas la reutilización y el paradigma orientado a objetos son factores importantes en el esfuerzo de desarrollo de software."

II.5.1 DISEÑO DEL EXPERIMENTO.

En este experimento los autores consideraron que para determinar los factores que influían en el experimento era necesario que los participantes llevarán a cabo las siguientes tareas de interés : evaluación de productos potencialmente reutilizables, adaptación a situaciones nuevas e integración de los mismos en productos funcionalmente completos.

Se eligió como participantes a estudiantes de los últimos grados de ingeniería de software, los cuales desarrollaron dos sistemas que involucraban técnicas de programación variadas, incluyendo manejo de datos, procesamiento numérico, y gráficas.

Las especificaciones del sistema fueron proporcionadas de manera semejante a las proporcionadas por la mayoría de las empresas de desarrollo de software, a la vez que fueron divididas en dos áreas.

Durante este experimento no se investigaron factores que afectaran la reutilización tales como el impacto de las características de los componentes de código (confiabilidad, estructuración, etc.) o las técnicas utilizadas para encontrar componentes de código dentro de un conjunto posible de candidatos ya que estos temas ya habían sido tratados por otros autores.

Sin embargo, todos los proyectos concluidos fueron examinados para verificar que cumplieran un conjunto de requerimientos como documentación, calidad de código y exactitud.

El experimento consto de dos fases la primera fue preparatoria, en ella se diseñaron los componentes a ser implementados y los componentes potencialmente reutilizables.

Ya en la segunda fase del experimento (la más importante) los sistemas fueron desarrollados por un conjunto de participantes los cuales recibieron algunos componentes de código reutilizable.

En seguida podemos observar en la figura 14, un diagrama donde se muestran las fases del desarrollo del experimento, durante la primera fase un conjunto de desarrolladores apoyados en un conjunto de especificaciones crearon algunos componentes en dos lenguajes con distintos enfoques (procedural y orientado a objetos), en la segunda fase participo un grupo de sujetos al cual se le pidió que desarrollara un proyecto al mismo tiempo que se le pedía lograr reutilización con diversos grados de motivación.

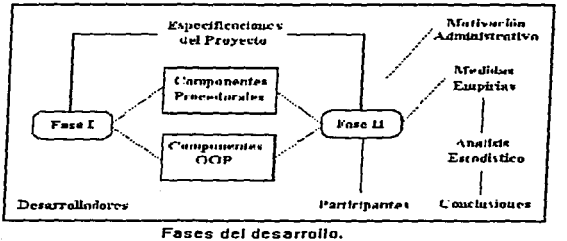


FIG. 14 DESARROLLO DEL EXPERIMENTO

II.5.2 FASE I DESARROLLO DE LOS COMPONENTES DE CÓDIGO REUTILIZABLE.

En esta fase se crearon dos conjuntos de componentes potencialmente reutilizables, uno de ellos se desarrollo en un lenguaje basado en procedimientos y el otro en un lenguaje orientado a objetos (Pascal y C++ respectivamente), los cuales se implementaron en Apple Macintosh.

Ambos conjuntos pasaron un conjunto de pruebas dirigidas a garantizar su equivalencia en cuanto a requerimientos funcionales y manejo de errores.

Al conocerse los requerimientos del sistema a implementarse, en la segunda fase se diseño cada componente de modo que contara con un nivel especifico de aplicación, los niveles de reuso fueron catalogados de la siguiente manera. :

A) Completamente reutilizable.
B) Reutilizable con una ligera adaptación (< 25%).
C) Reutilizable con una adaptación mayor (> 50%)
D) No reutilizable.

El sistema a crearse estaba diseñado para incluir el mismo número de elementos de cada nivel en su desarrollo.

II.5.3. FASE 2 IMPLEMENTACIÓN DEL PROYECTO.

Usando los dos conjuntos de componentes se asigno a veintiún participantes en seis grupos, una mitad desarrollo en Pascal y la otra desarrollo en C++, además dentro de cada lenguaje se hizo una división en tres grupos el primero de ellos sin reutilizar, el segundo con una motivación moderada hacia la reutilización y el tercero con un insistente requerimiento de reutilización (TABLA 1).

Como cada participante participaria en el desarrollo de dos aplicaiones entonces el numero de observaciones en total seria de cuarenta y dos, además la distribución de los participantes en los seis grupos fue aleatoria.

Influencia hacia la reutilización			
	No reutilizar	Motivación moderada	Motivación insistente
Lenguaje Orientado a Procedimientos	4 (8)	4 (8)	3 (6)
Lenguaje Orientado a objetos	3 (6)	4 (8)	3 (6)

TABLA-1 (Número de observaciones).

Los requerimientos funcionales del sistema estaban divididos en dos tareas relacionadas con manejo de empleados y manejo de negocios. La primera incluia el manejo de una base de datos de empleados, nómina y control de seguridad. La segundo estaba relacionada con el control de ventas en piso, pruebas de control de calidad, manejo de almacén e interacción con los clientes. A todos los sujetos se les proporciono descripciones por escrito de los requerimientos.

Las diferencias entre las tareas se controlaron de dos modos, primero las tareas elegidas fueron diseñadas de forma equitativa en cuanto a la dificultad de programación, cada tarea fué dividida en siete subtareas cada una de las cuales tenia una contraparte en la otra tarea

diseñada para requerir aproximadamente la misma cantidad de esfuerzo para desarrollarse. Segundo , una de las mitades implemento la tarea de manejo de empleados primero y después la de manejo de negocios, la otra mitad lo hizo en orden inverso.

II.5.4. ANÁLISIS DE DATOS.

Los datos recolectados durante el experimento midieron el esfuerzo necesario para implementar un sistema dado. Se considero a la productividad como relacionada de manera inversamente proporcional con el esfuerzo. La meta de este experimento era determinar cual de los dos grupos obtenía el mejor promedio de productividad significativamente mayor que el otro, las medidas de productividad empleadas fueron las siguientes :

a) **Corridas** :Número de corridas hechas durante el desarrollo y la prueba.
b) **ETE** :Número de errores en tiempo de ejecución ocurridos.
c) **Tiempo** :El tiempo necesario en minutos para corregir todos los ETE.
d) **Ediciones** :El número de ediciones llevadas a cabo para durante el desarrollo y la prueba.
e) **Sin** :El número de errores de sintaxis cometidos durante el desarrollo y pruebas.

Ya que cada sujeto implemento las mismas tareas, una comparación de los datos obtenidos proporcionó una visión del rendimiento de los sujetos en el desarrollo de una tarea. De éste modo un sujeto con una nota alta fue considerado menos productivo que uno que haya obtenido una menor puntuación.

Las múltiples medidas de productividad fueron utilizadas para obtener una imagen más completa del proceso de desarrollo, las tres primeras medidas fueron consideradas las variables de mayor interés. La información fue recabada en hojas de anotaciones llenadas por los participantes.

Los resultados obtenidos sirvieron para obtener las conclusiones siguientes aunque haría falta un análisis estadístico para percibir las diferencias. Se consideraron cuarenta y dos observaciones las cuales se distribuyeron como en la Tabla-1.

II.5.5 RESULTADOS DEL EXPERIMENTO.

En base a la información recabada en las siguientes tablas los experimentadores obtuvieron respuestas que favorecieron su hipótesis, recordemos que una puntuación baja en el esfuerzo necesario para llevar a cabo el desarrollo fue considerada como signo de mayor productividad.

La tabla siguiente nuestra los resultados de las mediciones cuando se reutiliza.

Reutilizando.			
	¿Significativo?	Procedural	Orientado a objetos
Corridas	Si	50.07	32.21
ESTE	Si	55.71	34.36
Tiempo	Si	301.86	208.86
Ediciones	No	189.00	208.64
Sin	No	137.14	164.71

TABLA-2

La siguiente tabla muestra la información recolectada cuando se motivo la reutilización de manera moderada.

Motivación moderada hacia la reutilización.			
	¿Significativo?	Procedural	Orientado a objetos
Corridas	Si	43.13	27.75
ETE	Si	49.50	32.00
Tiempo	No	264.65	196.13
Ediciones	No	192.13	189.50
Sin	No	143.25	146.75

TABLA-3

La siguiente tabla muestra los resultados obtenidos cuando hubo una fuerte motivación hacia la reutilización.

Fuerte motivación hacia la reutilización.			
	¿Significativo?	Procedural	Orientado a objetos
Corridas	Si	56.67	38.17
ETE	Si	64.00	37.50
Tiempo	Si	352.00	225.83
Ediciones	No	392.00	234.17
Sin	No	129.00	188.67

TABLA-4

El efecto de la reutilización en el esfuerzo necesario se reflejo de la forma siguiente :

Esfuerzo principal al reutilizar.			
	¿Significativo?	Procedural	Orientado a objetos
Corridas	Si	78.71	41.14
ETE	Si	83.79	45.04
Tiempo	Si	420.07	255.36
Ediciones	Si	405.71	198.82
Sin	Si	302.14	150.92

TABLA-5

En resumen las conclusiones obtenidas fueron las siguientes:

El paradigma orientado a objetos promueve mayor productividad que el paradigma procedural cuando los programadores reutilizan. cuando se les motiva ligeramente a la reutilización y cuando se les motiva con insistencia. Más aún, la reutilización promueve la productividad independientemente del paradigma que se use, el incremento de esta productividad motivada por la reutilización es todavía mayor cuando se utiliza el paradigma orientado a objetos.

La hipótesis era que el mejoramiento de la productividad con reutilización era mayor cuando se utiliza el paradigma orientado a objetos, la hipótesis fue cierta para las tres principales caracteristicas aunque para la tercera (tiempo de desarrollo) no fue muy significativa la diferencia.

En conclusión los resultados mostraron que el paradigma orientado a objetos demuestra una particular afinidad con el proceso de reutilización.

BIBLIOGRAFÍA PARA EL CAPÍTULO II

[Meyer,1988]	Meyer, Bertrand. "Object Oriented Software Construction." Prentice Hall; United Kingdom 1988 .
[Fischer,1989]	Fischer, Gerhard. "Human-Computer Interaction in Software: Lessons Learned, Challenges Ahead." IEEE Software, enero 1989.
[Martin,1993]	Martin, James. "Principles Of Object Oriented Analysis And Design." Englewood Cliffs, New Jersey, Prentice Hall International 1993.
[Yourdon,1990]	Yourdon, Edward & Coad Peter. "Object Oriented Analysis." Englewood Cliffs, New Jersey, Printece Hall 1990.
[Yourdon,1991]	Yourdon, Edward & Coad, P. "Object Oriented Design." Englewood Cliffs, New Jersey, Yourdon Press Computing Series 1991.
[Budd,1991]	Budd, T. "An Introducction to Object Oriented Programming" Reading Massachusetts, Ed. Addison Wesley 1991.
[Booch,1991]	[Booch, G. "Object Oriented Design with Applications" The Benjamin Commings Publishing Company Inc. 1991.
[Shaw,1984]	Shaw, M. "Abastraction Techniques in modern Programming Languages.", IEEE Software Vol. 1(4) 1984.
[Black,1986]	Black, A., Hutchinson, Jul, E. & Levy, H. "Object Structure in the Emerald System." SIGPLAN Notices, Vol. 23(3), Noviembre 1986.
[Lewis-Henry,1992]	Lewis, John A. & Henry, Sally M. "On The Relationship Between the Object Oriented Pradigm and Software Reuse : An Empirical Investigation." Journal of Object Oriented Programming Vol. 5(4), julio/agosto 1992.
[McCabe, 1976]	Mcabe, T. J. "A complexity Measure" IEEE Transactions on Software Engineering 2(4) 1976.

CAPÍTULO III

INGENIERÍA DE SOFTWARE Y REUTILIZACIÓN.

"Ningún hombre puede revelarnos cosa alguna que no se halle ya medio adormecida en el alba de nuestro conocimiento."

Gibrán Jalil Gibrán.

INTRODUCCIÓN

En este capítulo trataremos de la ingeniería de software y de como ésta puede ayudar a la reutilización, ya que en el capítulo II hemos establecido que existe una relación importante entre reutilización y modelo orientado a objetos es que nos inclinamos a presentar aquí a la ingeniería de software orientada a objetos (**OOSE** por sus siglas en inglés).

La ingeniería de software orientada a objetos se divide en dos partes fundamentales a saber, el análisis (**OOA**) y la construcción (**OOC**), de la primera parte se desprenden a su vez el modelo de requerimientos y el modelo del análisis, por su parte la construcción comprende el modelo del diseño propiamente dicho (**OOD**) y el modelo de la implementación.

A través de todo este proceso es que se descubren los objetos, se clasifican, se identifica el medio ambiente del sistema, se conocen las interfases, se depuran los objetos y se describen de manera cada vez más completa hasta llegar a la inclusión de las restricciones del medio ambiente y de la implantación. Por último concluir con las pruebas. Pero veamos paso por paso como sucede.

III.1 ANÁLISIS ORIENTADO A OBJETOS.

Discutiremos acerca del desarrollo de sistemas desde un punto de vista orientado a objetos, para ésto tendremos que obtener una especificación de requerimientos durante la que identificaremos a los actores y las tareas que conforman al sistema, después, la especificación de requerimientos será utilizada durante el análisis para definir los objetos que deberán tomar parte en el desarrollo del sistema. El producto obtenido durante esta etapa será de gran utilidad durante la fase de diseño.

III.1.1 INICIANDO EL ANÁLISIS ORIENTADO A OBJETOS.

Durante esta primera fase de análisis (**OOA**) es necesario empezar a buscar los objetos que nos ayudarán a ir formando nuestro modelo conceptual, el el capítulo II hicimos énfasis en la importancia que tiene la encapsulación dentro de este modelo, bueno pues ahora diremos que ésta forma la base para tratar a los atributos y a sus servicios correspondientes dentro del AOO como un todo intrínseco, además, de que ayuda a reducir el trabajo futuro de reestructuración al proporcionar una estructura mas estable.

El primer paso al inicio de un análisis es entender el problema, a medida que se van descubriendo objetos pertenecientes al dominio del problema se va ganando entendimiento el cual debe quedar registrado en un modelo del sistema y concluido durante fases siguientes, al mismo tiempo que creamos ese modelo vamos definiendo el alcance del sistema.

El propósito de todo análisis y por lo tanto también del análisis orientado a objetos es entender la aplicación, lo cual se obtiene con ayuda de los requerimientos funcionales del sistema.

El análisis orientado a objetos en general puede resumirse en las siguientes actividades.

1.- Encontrar objetos.
2.- Organizar estos objetos.

3.- Describir la interactúan.
4.- Definir las operaciones que llevan a cabo y
5.- Definir a los objetos internamente.

Estas diversas actividades se llevarán a cabo a medida que se desarrollen algunos modelos que surgen durante la fase de análisis. Durante la fase de análisis se desarrollan dos diferentes modelos el modelo de requerimientos y el modelo de análisis, la base de la que se parte es la especificación de requerimientos y las posteriores pláticas con los usuarios. El modelo de requerimientos debe sobre todo delimitar el sistema y definir cual deberá ser su comportamiento, para lo cual lo más recomendable es crear imágenes conceptuales del sistema usando objetos del dominio del problema y proporcionando descripciones de las interfases específicas del sistema. También se describirá al sistema como un conjunto de sucesos que serán llevados a cabo por un número de actores (los actores constituirán el medio ambiente del sistema) los sucesos serán las cosas que suceden durante la operación del mismo.

El modelo de análisis proporciona un esquema conceptual del sistema y contiene objetos de control, objetos entidad y objetos interfaz, el clasificar a los objetos dentro de las anteriores tres categorías nos da la oportunidad de verlos desde un punto de vista funcional que destaque sus responsabilidades. Cada uno de los diferentes tipos de objetos que se mencionaron anteriormente tiene un papel importante en el proceso de desarrollo de un sistema que cuente con la propiedad de ser robusto y extendible. Es oportuno mencionar que el modelo de análisis no tiene ninguna contraparte en otros paradigmas, es decir, es propio de la Ingeniería de Software Orientada a Objetos.

III.1.2 EL MODELO DE REQUERIMIENTOS.

DESARROLLO DE SISTEMAS BASADO EN REQUERIMIENTOS DEL USUARIO.

El modelo de requerimientos ayuda al desarrollador a delimitar el sistema y a definir la funcionalidad que el sistema debe ofrecer. Este modelo funciona como un contrato entre el grupo desarrollador y el grupo que requiere el desarrollo (consiste en la forma en que el desarrollador ve el sistema que quieren los solicitantes).

Este modelo constituye la parte central alrededor de la que se desarrollarán todos los demás modelos, se estructurará para crear el modelo de análisis, se conceptualizará para dar paso al modelo de diseño, se implementará en el modelo de implementación y se probará con el modelo de pruebas. El modelo de requerimientos servirá para verificar a los otros modelos y no sólo eso, sino que servirá como base para la construcción de estos y para el desarrollo de instrucciones operacionales y manuales.

Se le da tanta importancia a este modelo debido a que todo él está creado bajo el punto de vista del usuario por lo que será más fácil entenderse con el usuario utilizando este modelo como base.

El modelo de requerimientos consiste de tres partes:

- El modelo de casos de uso.
- El modelo de objetos del dominio del problema y
- El modelo de descripciones de interfases del usuario.

El modelo de casos de uso describe la funcionalidad con que el sistema debe contar a través del punto vista del usuario, además de describir que sucede dentro del sistema. Para lograrlo se vale de los conceptos de actores y tareas.

Actores y tareas, una forma de encontrar objetos.

Una de las principales actividades del análisis orientado a objetos es la de encontrar objetos, bueno pues en el momento de identificar actores y tareas vendrán a nosotros muchos elementos naturales pertenecientes al dominio del problema que será necesario manejar.

Los objetos aparecen como entidades naturales dentro del dominio del problema por eso es necesario que el analista se familiarice con los términos que se manejan en el, de ésta manera, se contará con algunos objetos candidatos a ser incluidos en el sistema. Naturalmente es un poco más difícil saber cuales objetos si serán utilizados en el sistema, pero, en general debemos de tratar de hallar a aquellos objetos que perduraran durante todo el ciclo de vida del sistema. Lo anterior es importante si queremos cumplir el objetivo de construir un sistema estable.

En primer lugar como en cualquiera de los métodos de análisis lo que el analista tiene que hacer es involucrarse en el problema, ahondar en el dominio del usuario por medio del desarrollo de una investigación, tratar de ir adquiriendo vocabulario por medio de la lectura y buscar la definición de los términos en diccionarios relacionados con el tema, a la vez es una buena practica pedir al usuario que nos proporcione alguna descripción escrita del sistema. También es necesario recolectar toda clase de diagramas que nos ayuden a comprenderlo, al mismo tiempo que se recomienda elaborar los propios diagramas que servirán para formarse una idea mas clara del funcionamiento del sistema y de sus requerimientos de operación.

Considere a otros sistemas y agentes externos con los que el sistema interactuará, a los dispositivos con los que el sistema se relacionara y analice y busque eventos que necesiten ser recordados por el sistema, considerar el papel que juegan los humanos dentro del sistema es otra fuente potencial de objetos, pregúntese también si existe algún lugar ya sea una oficina o sitio acerca del que éste deba estar enterado. Asimismo debe preguntarse de que manera se agrupan o que unidades organizacionales forma la gente.

Este modelo se vale de actores, para representar los diferentes roles que pueden jugar los usuarios, y de casos de uso (cada caso de uso es una descripción completa de los eventos que suceden en el sistema desde el punto de vista del usuario) para representar las cosas que los usuarios serán capaces de hacer con el sistema.

Lo primero que tenemos que hacer es identificar cuales serán los usuarios en el sistema para después poder definir los requerimientos de operación (o casos de uso). Un actor es un tipo de usuario o categoría, una sola persona puede comportarse en diferentes momentos como actores diferentes por lo que decimos que un actor define un papel que puede ser jugado por algún usuario. Los actores constituyen cualquier cosa que sea externa al sistema que se va a desarrollar o lo que es lo mismo cualquier cosa que tenga un intercambio de información con el sistema. La búsqueda de actores nos ayuda a establecer los límites del sistema. Así pues al identificar actores y tareas podremos definir cual es el límite entre los actores y los casos de uso en que en que serán capaces de participar los usuarios.

A manera de que quede más claro que es un actor y que es un usuario digamos que un actor es una clase, es decir, un tipo de objetos que cuenta con identidad, estado y comportamiento, mientras que un usuario puede jugar diferentes roles, es decir puede comportarse como diferentes actores y por ende valerse de distintos objetos.

Mencionamos que una fuente de objetos tentativos eran los diferentes actores que participan en la operación del sistema. Por ejemplo pensemos en que personas serán las beneficiadas con el desarrollo del sistema (quienes tendrán que tratar a diario con él) a los cuales llamaremos actores primarios. Otros actores identificables son aquellos que tendrán la responsabilidad de dar mantenimiento al sistema a estos los llamaremos actores secundarios. Esta manera de clasificar a los actores tiene el propósito de definir la estructura del sistema en términos de la funcionalidad necesaria para los actores participantes.

Encontrar actores que modelen gente es sencillo pero encontrar actores que modelen máquinas es un poco más complejo, aunque, a medida que se van especificando las tareas a llevarse a cabo se da uno cuenta que son necesarios.

CASOS DE USO.

Una vez terminada la definición externa del sistema podemos dar paso a la definición interna la cual se logra al definir las tareas funcionales que deberá llevar a cabo el sistema, estas tareas son los comportamientos a que dará respuesta un objeto cuando suceda un evento, es decir, que algún actor requerirá (por medio de un mensaje) de un uso específico del sistema, este evento ocasionará una respuesta (ejecución de ciertos métodos) por parte de otro objeto. Recordemos que, a diferencia de los procedimientos (utilizados en paradigmas funcionales de desarrollo), un objeto puede exhibir diferentes comportamientos dependiendo de quien le haya enviado el mensaje. Cada caso de uso específico del sistema constituye un curso completo de eventos iniciado por un actor y especifica la interacción que sucede entre el actor y el sistema, en otras palabras es una secuencia de operaciones (**transacciones**) relacionadas y llevadas a cabo para satisfacer la petición del actor. La colección completa de casos de uso constituye el conjunto de todas las formas de usar el sistema. Una manera de representar la relación entre actores y casos de uso es mediante diagramas de transición de estados, donde se debe mostrar cómo un estímulo del actor provoca un cambio de estado. También es recomendable al identificar a los actores y a los casos de uso del sistema representar a los actores como sujetos externos al sistema que interactúan con una caja que contiene diferentes elipses representando a los casos de uso. Ver Fig. 15

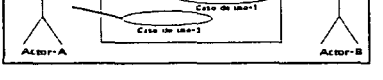


FIG. 15 REPRESENTACION DE UN CASO DE USO

El proceso de encontrar estos casos de uso es frecuentemente de naturaleza iterativa, ya que para concluir la búsqueda de manera satisfactoria es necesario hacer varios intentos antes de encontrar el más apropiado. Durante este proceso debemos encontrar respuestas para las siguientes preguntas : ¿Cuales son las principales tares de cada actor ?, ¿Es necesario que el actor lea/escriba/cambie información contenida en el sistema?, ¿Es necesario que el actor informe al sistema de los cambios que suceden fuera del sistema?, ¿Desea el actor que el sistema le informe de los cambios inesperados dentro del sistema?.

Una vez que se ha estabilizado el proceso anterior para cada tarea que se ha definido se describe el curso básico que seguirá (o comportamiento principal que exhibirá) una tarea, por otro lado se definirán como cursos alternos a aquellos comportamientos que sean resultado de errores (posiblemente causados por insatisfacción de condiciones necesarias). En el momento de estar haciendo la descripción completa de los casos de uso se está analizando al sistema muy de cerca y es posible que en ese momento se despejen algunas dudas que nacieron al momento de estudiar los requerimientos. Una recomendación oportuna en este momento es que cada tarea debe poder definirse como una tarea sencilla, es decir, que en lo posible debe de poder describirse sin tener que recurrir a otras tareas. El objetivo aquí es evitar en l lo posible que las tareas esten compuestas de otras tareas. Aunque también es apropiado que se utilice el concepto de extender o, como algunos otros autores lo llaman, especializar a una tarea o clase. Lo que se logra al organizar a las tareas de modo jerárquico permitiendo que las tareas más generales contengan especializaciones de la misma tarea pero con detalles más específicos.

Posiblemente no sea muy sencillo al principio el decidir cuando es apropiado extender el comportamiento de un objeto tarea, pero para nuestra fortuna [Jacobson,1992] nos proporciona algunas reglas sencillas para orientarnos :

- Cuando se desee modelar partes opcionales de los casos de uso o,
- Para modelar cursos o alternativas complejas que ocurran por si mismas,
- Para modelar cursos alternos que sólo serán seguidos en casos particulares,
- Para modelar que diferentes casos de uso puedan insertarse en un caso de uso especial con el fin de que compongan un caso de uso más general.

Cundo se inserta un caso de uso por medio de una extensión de comportamiento en un caso de uso específico este último debe conservar su mismo comportamiento y sólo en situaciones específicas debe reaccionar de la manera en que el caso de uso insertado le obliga, para después continuar con su operación normal como si nada hubiese sucedido.

Una vez que se ha encontrado algún candidato a objeto es necesario hacer un pequeño examen para decidir si se incluirá dentro del sistema o no. Primero piense si es necesario que el sistema tenga en cuenta algo acerca de este objeto, piense cuales serían los atributos de este objeto. ¿Tendrá que proporcionar el sistema algún servicio a este objeto ? o ¿necesitará mantener ocurrencias de este tipo de objetos?, ¿Es posible identificar una lista de atributos relacionados con el objeto?, ¿Existe algún conjunto común de servicios de procesamiento que pueda aplicarse a cada ocurrencia de un objeto?.

Al principio durante la investigación en busqueda de un conjunto inicial se considerará a los atributos y servicios comunes de un modo muy general, después a medida que se vaya

profundizando y analizando los servicios y atributos con mayor detalle se descubrirán detalles más finos que deberán incluirse.

Muchas veces nos encontramos con que existen en nuestro modelo muchos objetos que en realidad no son necesarios dentro del modelo, es posible que existan en el mundo real, pero que ya no tengan ninguna influencia sobre el sistema. Para eliminar este tipo de elementos sobrantes en nuestro análisis es necesario que sometamos cada objeto a reconsideración. Recordemos que para detectar objetos potenciales dentro del modelo consideramos aquellos objetos acerca de los cuales el sistema debía recordar algunos eventos, ahora reconsideremos si en realidad el sistema necesita estar enterado de esos datos, en caso de que no lo necesite preguntémonos si existe algún servicio que deba ser proporcionado para ese objeto, en caso de que no lo haya lo mejor es hacerlo a un lado no considerándolo en adelante. Después reconsidere a aquellos objetos de los cuales únicamente exista una ocurrencia, probablemente existan otros objetos que tengan atributos comunes a este objeto así que en un futuro estudie la posibilidad de crear una estructura de clasificación que encierre a los dos objetos o búsquese otros objetos que tal vez tengan los mismos atributos y necesidades de servicios e inclúyase al objeto dentro de esa misma clase.

Por último juzgue el modelo en lo relativo a los resultados derivados de atributos ya incluidos en él y decida si es mejor derivarlos cada vez que sea necesario o mantenerlos como atributos de algún objeto.

Incluso es recomendable tener alguna regla para nombrar a los objetos se sugiere utilizar nombres que aparezcan en el dominio del problema y llamarlos de la misma manera que los llaman los usuarios.

OBJETOS DEL DOMINIO DEL PROBLEMA.

Muchas veces al trabajar con el modelo de requerimientos se descubren ciertas vaguedades que son difíciles de eludir, en tales casos se recomienda iniciar el análisis aprovechando los objetos del dominio del problema, es decir, objetos que tienen una contraparte en el dominio de la aplicación y de los que el sistema a desarrollar debe tener conocimiento. Este modelo del dominio del problema puede ser de gran utilidad a la hora de definir los casos de uso específicos con que contará el sistema. Muchas veces resulta de gran valor el proporcionar una pluma al usuario y pedirle que haga una gráfica de como ve el sistema, de este modo y con la participación activa del desarrollador y del usuario se logra obtener un esquema bastante completo del sistema con un punto de vista orientado al usuario y con la ventaja de que se obtiene un lenguaje apropiado que será manejado durante el desarrollo.

Cuando se identifica a un objeto del dominio del problema este se va refinando cada vez más, las fases de refinamiento por las que pasa son, a grandes rasgos mostradas en la Fig. 16.

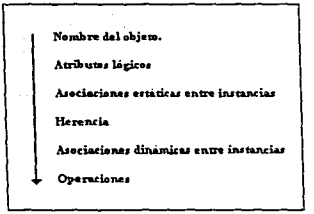


FIG. 16 REFINAMIENTO DE LOS OBJETOS DEL DOMINIO DEL PROBLEMA

Se recomienda que en este momento no se vaya más allá del tercer paso ya que no se desea que el modelo se vuelva demasiado rígido antes de que se logre la mejor estructura en cuanto a estabilidad y mantenibilidad.

Por asociaciones estáticas entre instancias se entiende asociaciones que son usadas cuando un objeto debe estáticamente saber de otro objeto. Asociación dinámica entre instancias se refiere a aquellas asociaciones donde un objeto puede enviar estímulos a otro, lo que implica una dependencia del protocolo del otro objeto.

DESCRIPCIÓN DE INTERFASES.

Cuando se esta pensando en la manera en que se comunicaran los usuarios con los casos de uso y en que estos serán útiles para diferentes usuarios siempre debe pensarse en estandarizar las interfases a fin de que los usuarios sepan como se utilizarán, incluso, antes de que se haya pensado en como construirlos. El principal objetivo de esta descripción de interfases es el de determinar a través de que mecanismos se comunicará el usuario con el sistema (pantallas, botones, listados, redes, etc.). Además se pretende, que sin llegar al diseño de los mecanismos internos para proporcionar las salidas deseadas el usuario y el desarrollador vayan afinando detalles y depurando malos entendidos. El desarrollador puede valerse de prototipos durante la descripción de interfases, lo que resulta de gran ayuda además de ser rápido.

Cuando se piensa en construir las interfases es necesario contar con la participación de los usuarios para que la interfaz refleje el punto de vista lógico del usuario acerca del sistema. En este punto no sólo es importante describir las interfases de usuarios, sino que ya podría pensarse en otras interfases del sistema tales como protocolos de comunicación.

III.1.3 REFINAMIENTO DEL MODELO DE REQUERIMIENTOS.

El modelo de requerimientos que hasta ahora hemos definido pudiera ser suficiente para especificar la funcionalidad del sistema, sin embargo, podemos elaborar este modelo de manera que no sólo mejore la reutilización sino para que la transición al modelo de análisis sea más sencilla.

CASOS DE USO ABSTRACTO Y CASOS DE USO CONCRETO (RELACIÓN DE USO Y RELACIÓN DE EXTENSIÓN).

Este refinamiento se logra abstrayendo los diferentes casos de uso a fin de identificar partes similares, estas partes que a su vez constituirán a los casos de uso reales serán llamados casos de uso abstractos, ya que sólo servirán para agrupar comportamientos comunes y no serán requerimientos explícitos por sí mismos. En cambio aquellos casos de uso que pueden componerse de estos distintos casos de uso abstractos y que sí son requerimientos de uso explícitamente requeridos por los usuarios serán llamados casos de uso concretos. Tal relación en que un caso de uso concreto se vale de un caso de uso abstracto para su propia definición es llamada relación de uso. De modo semejante, cuando un caso de uso como veíamos anteriormente sirve para extender el comportamiento de otro caso de uso decimos que el segundo usa (o esta en relación de uso con el primero) al primero. Como podemos observar estos casos de uso abstractos podran en un futuro convertirse en clases abstractas. Fig. 17

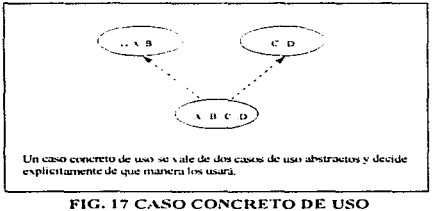


FIG. 17 CASO CONCRETO DE USO

La relación de extensión es utilizada cuando se quiere incluir dentro de una relación de uso a otra relación de uso que proporcionará a la primera un comportamiento definido en un caso muy específico.Ver Fig. 18

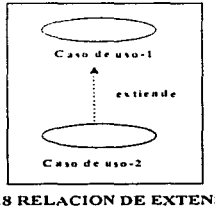


FIG. 18 RELACION DE EXTENSIÓN

Ya que ambas relaciones de uso y de extensión asocian clases, algunas veces puede dificultarse la decisión de cual de ellas utilizar, un criterio importante es observar que tan fuerte es la liga funcional que las une, si los cursos que se desprenden de ellas son independientes la extensión es más recomendable, de otra forma, si los cursos que se desprenden de ellas están ligados fuertemente la relación de uso es la más apropiada. Además recuérdese que existe mucha diferencia en la manera de identificar estas relaciones, las relaciones de uso se identifican a través de la extracción de secuencias comunes de diferentes casos de uso mientras que las relaciones de extensión se identifican a través de la detección de nuevos cursos de acción necesarios.

III.1.4 EL MODELO DEL ANÁLISIS.

Una vez concluido el modelo de requerimientos y de que éste sea firmado por los solicitantes es momento de dar paso al modelo del análisis, durante este paso describiremos al sistema utilizando tres tipos de objetos : interfaz, entidad y control. Con ayuda de ellos estructuraremos el modelo de requerimientos en el modelo de análisis, este último nos servirá como base para el modelo de diseño.

CLASIFICACIÓN DE OBJETOS.

El trabajo de desarrollar el modelo del análisis obliga a distribuir el comportamiento especificado en las descripciones de los casos de uso entre los objetos del modelo del análisis. Estos objetos en esta fase pueden ser definidos al describir su comportamiento, no precisamente en forma de operaciones pero si se debe hacer una descripción escrita de las responsabilidades o roles que juega cada objeto. A continuación describiremos los tres tipos de objetos antes mencionados. El conjunto total de tareas que se llevan a cabo en un caso de uso especifico y en todos los casos de uso es una combinación de varios objetos del análisis, por lo que diferentes casos de uso pueden tener objetos en común.

Objetos Interfaz.

Estos objetos sirven como medio de comunicación entre los actores, es decir, ayudan a traducir el comportamiento de los actores en eventos aceptados por el sistema y viceversa traducir los resultados o comportamiento del sistema en elementos presentados a los actores, en otras palabras los objetos interfaz pueden ser vistos como los medios para establecer la comunicación bidireccional entre los usuarios y el sistema.

Cada actor necesita al menos una interfaz para que sus acciones sean reconocidas como eventos del sistema; las pantallas de captura, menús, ventanas y todo lo que ponga en contacto al actor con el sistema es un objeto interfaz. Por lo tanto buscaremos a estos objetos interfaz analizando cuales son las interacciones de los usuarios con el sistema, por otro lado, si buscamos en la descripción de los casos de uso, también ahí encontraremos objetos interfaz. Una vez identificados los objetos interfaz deberemos proceder a describirlos por escrito dándoles un nombre, especificando que actor(es) se vale(n) de él, que partes contienen y para que sirven, con que otros objetos tienen relación y cual. Los objetos no son independientes del todo, por lo menos necesitan conocer a otros objetos, a este hecho se le llama relación de conocimiento, un objeto puede estar asociado con una o más instancias de una clase de objetos por lo que es necesario que ésto quede plasmado por medio de la asignación de un número

cardinal a cada asociación. Esta cardinalidad nos dice cuantas instancias de un objeto pueden asociarse con un ejemplar de otra clase. Fig. 19

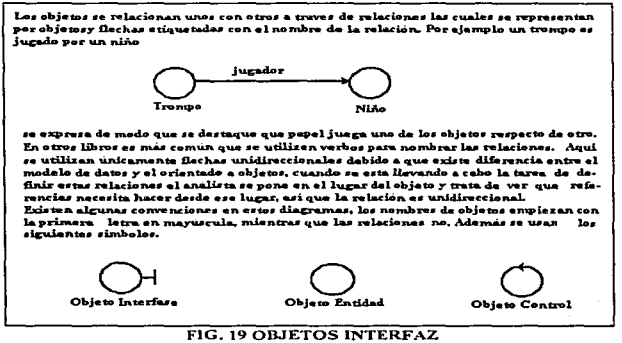


FIG. 19 OBJETOS INTERFAZ

Dentro de las relaciones de conocimiento se encuentra la relación *compuesto-de*, un objeto compuesto de otros es conocido como un agregado.

Objetos entidad.

Con ellos se modela toda la información que el sistema conservará por largo tiempo, en ellos encapsularemos los datos y el comportamiento de los entes que sean más trascendentales, este tipo de objetos se descubren fácilmente dentro del dominio del sistema y están relacionados con algún concepto del mundo real.

Es muy posible que se encuentren diferentes objetos que compartan características en común, por lo que debemos analizar la posibilidad de agruparlos en alguna clase común, del mismo modo existirán algunas otras características que no compartirán y que les darán identidad, debemos crear una estructura adecuada que soporte estas similitudes y estas diferencias de manera inteligente es decir agrupando las similitudes y extendiendo las diferencias.

Objetos de control.

Cuando se habla de sistemas más complejos, es posible que una vez que se han identificado los objetos interfaz y los objetos entidad aún quede por ahí algún comportamiento que no haya quedado representado de manera natural en alguno de estos tipos de objetos, tal comportamiento debe ser puesto en objetos de control, la razón por la que este comportamiento no ha quedado debidamente incorporado en alguno de los tipos de objeto

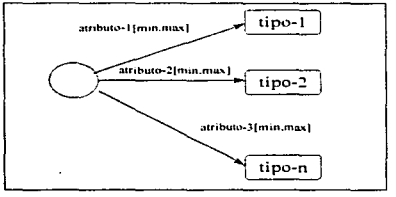
anteriores es que tal comportamiento en realidad no pertenece a la interfaz del sistema ni a la manera en que la información es manejada.

Los objetos de control trabajan como pegamento que une objetos sobrantes para formar tareas que no han sido modeladas, éstos generalmente son encontrados en las descripciones de tareas. Durante una primera revisión se asigna un objeto de control por cada uso abstracto, cada uno de los cuales normalmente involucra objetos interfaz y objetos entidad, aquel comportamiento que permanece sin incluirse en el modelo después de que los objetos interfaz y los objetos entidad han obtenido sus partes debe ser incluido en objetos de control.

Esta funcionalidad que se incluye en los objetos de control puede ser compleja, en cuyo caso debe dividirse en varios objetos de control, los cuales deben tener tareas limitadas que sean fáciles de describir y de entender. En el caso de que un objeto de control esté ligado con múltiples actores debemos entender que tal vez el comportamiento será diferente para los actores y que por lo tanto deberá ser separado en objetos de control diferentes. Lo anterior se sugiere así debido a que la mayoría de los cambios en los sistemas provienen de necesidades de los actores, cuando se habla de describir comportamiento es necesario enfocarse en que resultados se obtendrán, no de que manera se lograrán. Las descripciones formales deben evitarse cuando el modelo aún es inestable y deben irse incorporando a medida que el modelo se va volviendo más estable.

ATRIBUTOS Y OPERACIONES.

Ya que se identificaron los objetos que aparecerán en el modelo del análisis, con un poco más de esfuerzo se debe identificar que operaciones y que atributos les pertenecerán (recordemos que el único camino para manipular a los objetos es mediante la utilización de sus propios métodos a través de la encapsulación). Por lo que es necesario describir cierta información que se almacena en atributos, cada atributo es definido como un tipo específico (primitivo o compuesto). Un atributo es descrito como una asociación entre una entidad y un tipo, todos los objetos utilizan atributos para almacenar información, de modo que un objeto puede estar ligado a una variedad de atributos los cuales tienen un tipo específico, un atributo se define como un descriptor de la información almacenada, podemos utilizar diagramas como el de la Fig. 20, para especificar los atributos con que cuenta un objeto y cual es la cardinalidad de estos atributos.



Cada atributo de un objeto o entidad puede suceder un número mínimo de veces y un número máximo, además de que tiene asociado un tipo de dato específico.

FIG. 20 ATRIBUTO-OBJETO

A menudo se dificulta la identificación de entidades u objetos necesarios en el modelo, y por lo tanto se dificulta más la identificación de sus atributos y las operaciones que llevará a cabo, de ahí que la descripción detallada de los casos de uso sea de gran importancia.

También es común que el desarrollador no sepa en que objetos debe descansar la mayor cantidad de operaciones ya sea en objetos entidad, control o interfaz, la mayoría de los desarrolladores novatos delegan toda la responsabilidad de las operaciones en los objetos de control y utilizan a los objetos entidad como transportadores de datos, naturalmente la carga de las operaciones debe estar balanceada entre los diferentes tipos de objetos (aunque se puede hacer sistemas prescindiendo de los objetos de control), para lo que se sugiere que se trate de incluir en los tipos entidad e interfaz toda la operación del sistema y se incluya en los objetos de control sólo aquel comportamiento que no fue puesto desde el punto de vista del mantenimiento en los otros tipos de objetos de manera natural. Como recomendación se da la de trabajar de manera estructurada a medida que se vayan descubriendo nuevos objetos, esto debido a que en ocasiones se puede llegar a definir objetos en demasía lo que posteriormente acarreará problemas para depurar a este conjunto. Recuérdese que lo esencial aquí es buscar que objetos mantendrá el sistema durante mayor tiempo.

Aunque a veces es difícil decidir si una cierta pieza de información debe ser modelada como un atributo o como un objeto, se debe considerarse la forma en que esta información será utilizada, si la información va a ser manejada por separado debe de ser modelada como un objeto entidad, mientras que si la información va a ser manejada como una parte inseparable de otra pieza de información y que nunca será manejada de manera aislada debe ser modelada como un atributo de aquella de la que no puede separarse.

Posteriormente y debido a que la identificación de operaciones es una tarea de diseño comentaremos algunas técnicas empleadas por la ingeniería de software orientada a objetos (**OOSE**) para obtener los atributos y operaciones de cada objeto mediante el uso de diagramas de interacción.

Una lista típica de operaciones que debe ser incluida en un objeto es la siguiente :

- Almacenar y recuperar información.
- Cursos de acción que deben ser cambiado en caso de que cambie el objeto entidad.
- Creación y borrado de instancias del objeto.

Una operación en un objeto se da cuando se recibe una petición de información de otro objeto, este comportamiento se refleja en el modelo a través de asociaciones de comunicación, esta asociación se representa a través de un objeto que envía un estimulo a otro objeto que responderá de algún modo definido.

Hemos visto que el modelo del análisis no será reflejo del dominio del problema. Es importante entender ésto, la razón es simplemente proporcionar una estructura más estable, donde los cambios sean locales y por lo tanto más manejables, así que no modelamos la realidad como frecuentemente se dice del modelo orientado a objetos, sino que se modela la realidad de la manera que queremos verla y resaltamos lo que es importante para la aplicación.

FUNCIONALIDAD Y CONTROL.

El problema de dónde debe residir la funcionalidad del sistema tiene muchas respuestas que aquí no trataremos, aunque remitiremos al lector interesado a la referencia [Hartson-Hix,1989] y presentaremos las formas más comunes de manejar la funcionalidad:

Control dominante de cálculos.

Se refiere a que el control de la funcionalidad está dentro de los objetos entidad y dentro de los objetos control, los objetos interfaz no contiene mucha funcionalidad por lo que se puede ganar en eficiencia de ejecución pero no será fácil desarrollar prototipos.

Control dominante de diálogos.

En este caso se pone mucho control de funcionalidad en los objetos interfaz y muchos de ellos modelan la funcionalidad del sistema, por lo tanto no se tienen muchos objetos de control en el modelo. Esta estrategia ayuda al desarrollo de prototipos, aumentando la complejidad de las interfases.

Control distribuido.

Se distribuye el control en ambos lados permitiendo mayor flexibilidad, aunque, requiere más disciplina por parte de los programadores para mantener la independencia de los objetos.

Control balanceado.

Es cuando se aparta el control del diálogo y de los cálculos, el control global es vigilado por objetos de control, es decir, se encarga de controlar la secuencia de invocaciones que hacen los objetos interfaz a las funciones de cálculo.

AGRUPACIÓN DE OBJETOS EN SISTEMAS Y SUBSISTEMAS.

Cuando se ha terminado de identificar a los objetos que forman parte del modelo de análisis se cuenta con una amplia variedad de objetos que es necesario clasificar. Nosotros sabemos que para esconder la complejidad de un sistema éste puede ser dividido en subsistemas. Para hacer más sencillo su análisis; los sistemas están compuestos de subsistemas y estos a su vez también están compuestos de subsistemas menos complejos y hasta el último nivel de esta jerarquía están los objetos del análisis. Los subsistemas son formas de estructurar los sistemas para facilitar el futuro desarrollo y mantenimiento, su tarea es empacar los objetos de modo que la complejidad se reduzca. Algunos criterios que pueden utilizarse para subdividir a los sistemas son los siguientes.

i) Los diferentes grupos de desarrollo tienen competencia y recursos diferentes y es deseable distribuir el trabajo de desarrollo de acuerdo a los diferentes grupos contemplados.
ii) Dentro de un medio ambiente distribuido, puede desearse un subsistema para cada nodo lógico.

iii) Si un producto existente puede ser utilizado en este sistema, este debe ser considerado como un subsistema.

En todo problema existe una estructura que es la que proporciona la complejidad a un problema, a continuación platicaremos de que manera el análisis orientado a objetos proporciona maneras de identificar dicha estructura y de que manera ayuda a organizar los objetos al descubrir la interacción entre éstos, las operaciones en que se basará su comportamiento, sus atributos y sus servicios.

Existen diferentes formas de clasificar y organizar a los objetos y a sus clases una de ellas es la jerarquía de herencia la cual los organiza considerando que tan similares son unas clases con otras. Otra forma de clasificarlos es considerando que objetos se relacionan con otros o considerando que objetos son parte de otros.

Para obtener una imagen de como situar a los objetos en el sistema podemos describir diferentes escenarios a fin de mostrar la comunicación que se da entre ellos; las interfases de los objetos pueden elegirse a partir de estos escenarios, pero, algo que no debe olvidarse es la complejidad propia del problema. Ésta debe identificarse de manera completa para poder contar con un esquema que nos permita manejarla.

IDENTIFICANDO LA ESTRUCTURA INHERENTE AL PROBLEMA.

La estructura de clasificación proporciona una división del espacio del problema consistente en dividir los atributos y los servicios en grupos mutuamente exclusivos. Adicionalmente nos ayuda a identificar abstracciones de nivel superior, lo que es útil en el momento de poner atributos y servicios comunes a un nivel superior para después compartirlo con las abstracciones de nivel inferior.

Un objeto que se encuentra a nivel inferior puede añadir atributos y servicios con los que las estructuras de nivel superior no cuenten, logrando una mayor especialización en los niveles

más bajos. En consecuencia debemos de tratar de poner los atributos y los servicios comunes dentro de una estructura de clasificación que se valga de la herencia (ver capítulo II).

Además de la estructura de clasificación, existe otra relacionada con la manera en que se puede ensamblar a los objetos, con lo anterior nos referimos a que cada objeto individual puede interactuar con otros objetos para formar una estructura de interés. Si el sistema necesita manejar esta estructura tendremos que añadirla. Algunos autores recomiendan hacer esquemas del sistema en los que se pueda apreciar la estructura de ensamblaje así como diagramas que muestren la estructura de clasificación que representa a la relación *clase-miembro* (generalidad-especialización) considerando a cada objeto como una generalización y después como una especialización. Por otro lado la estructura de ensamblaje representa la agregación, reflejando el todo y sus partes componentes, considerando cada objeto como un todo y después como una parte.

Los sistemas del mundo real tienen un número muy grande de objetos y estructuras, ahora, pensemos que esto puede complicar el manejo del proyecto, recordemos que uno de los factores más importantes a considerar para que un método tenga éxito es su capacidad para facilitar la comunicación entre los participantes. En un artículo [Miller,1956] se establece que la mente humana sólo es capaz de atender un número de ideas entre cinco y nueve, bien, pues basados en ese conocimiento las técnicas del análisis se han visto afectadas, por lo que frecuentemente hemos oído hablar de presentar a lo más siete elementos en una pantalla o en un diagrama de flujo.

El resultado obtenido por el Dr. Miller ha sido interpretado de diferentes maneras una de ellas es que debemos controlar la visibilidad para lograr un modelo comprensible. En la práctica podemos llegar a tener sistemas tan complejos que el seguir esta regla nos lleve a tener que dividir nuestro modelo en tantas partes que el tratar de enlazarlas costituirá un verdadero problema.

Otra interpretación es guiar la atención del lector, a través de un diagrama grande, haciéndolo poner atención en un área que contenga siete componentes aproximadamente. Con esto se pretende tener pocas hojas de papel y pocos niveles de jerarquía, al mismo tiempo que la cantidad de información dada al lector esta regulada sin hacer uso de una pila de diagramas.

III. 2 CONSTRUCCIÓN ORIENTADA A OBJETOS.

En esta sección trataremos lo relacionado con el proceso de construcción, el cual incluye diseño e implementación y finaliza cuando la escritura de código esta completa. Naturalmente incluye las pruebas necesarias. En esta fase de suyo se obtiene dos modelos importantes : el modelo del diseño y el modelo de la implementación. Ambos van de la mano ya que el diseño adquiere realismo a través del segundo y éste último surge basado en el primero. Antes de explicar los dos modelos que comprende la construcción orientada a objetos incluiremos una breve introducción acerca del diseño.

III.2.1 LA TAREA DEL DISEÑO.

El papel del diseñador tiene visos de ciencia y de arte ya que puede involucrar mucha imaginación, experiencia y conocimiento. Una vez que el ingeniero de software ha montado el

diseño como artista, debe asumir el papel de ingeniero de software pero como científico y analizar su diseño rigurosamente para poder proporcionar una solución del problema que cumpla con los propósitos del diseño que Mostow [1] sugiere:

a) Satisfacer una especificación funcional.
b) Adaptarse a las limitaciones del medio ambiente en que operara.
c) Satisfacer implícita o explícitamente los requerimientos de operación y uso de recursos.
d) Satisfacer implícita o explícitamente los criterios de diseño relacionados con la forma que deberá tener el producto.
e) Satisfacer restricciones propias del proceso de diseño, tales como tiempo, costos, herramientas disponibles para hacer el diseño etc.

Según Yourdon el contexto del sistema muestra una restricción cuádruple representada por la siguiente ecuación :

Restricción cuádruple = ***Presupuestos***
+ ***Calendarización***
+ ***Capacidad***
+ ***Gente***

El diseñador debe balancear un conjunto de requerimientos para producir un modelo que permita razonar acerca de las estructuras, requerimientos y rutas a seguir durante la implantación.

RAZONES PARA LLEVAR A CABO EL DISEÑO DEL SISTEMA.

Algunas personas piensan que podría escribirse el código una vez que se ha terminado la fase de análisis (ya que para entonces se cuenta con la descripción de los objetos y como se relacionan), en cuyo caso no sabríamos para que sirve la fase de diseño, bueno pues existen tres razones que nos indican que debemos tener una fase de diseño, y son las siguientes :

La primera es que el modelo del análisis no es lo suficientemente formal, es decir, que es necesario refinar los objetos para poderlos codificar. Refinar las operaciones que ofrecerán, modelar los métodos a través de los cuales se comunicarán y definir que estímulos recibirán/enviarán son actividades que también forman parte del diseño y que no han sido llevadas a cabo durante la fase de analisis.

Segunda, el sistema actual debe ser utilizado como medio ambiente de la implantación, es decir que durante el análisis se asumió un medio ambiente ideal para el sistema por lo que ahora tendremos que hacer adaptaciones al medio ambiente para poder implantar al sistema. En esta fase deberemos poner en consideración los requerimientos de tiempo real, la concurrencia, las propiedades de los lenguajes, las propiedades del lenguaje manejador de base de datos y muchas más.

Y tercera, es necesario validar los resultados del análisis a través de la construcción, donde podremos ver que tan bien describen los modelos de analisis y de requerimientos al sistema. En caso de que durante este proceso descubramos puntos faltos de claridad en el modelo de

[1] Mostow, J. "Toward Better Models of the Design Process." A. I. Magazine Vol. 6(1) 1985.

análisis o en el modelo de requerimientos deberemos aclararlos regresando al modelo respectivo a fin de conservar la rastreabilidad, es por esto que se dice que el proceso completo de crear un sistema es iterativo.

LOS ELEMENTOS DE LOS MÉTODOS DE DISEÑO DE SOFTWARE.

Los métodos de diseño han evolucionado en respuesta a la creciente complejidad de los sistemas de software requeridos, en un principio no se construían programas demasiado grandes debido a la poca capacidad de las máquinas de ése entonces, a mediados de la década de los setentas el panorama cambio al crecer la capacidad de los equipos y disminuir su costo, así, cada vez resultaba más atractivo y económico el automatizar aplicaciones de complejidad creciente. Ya en este momento los lenguajes de programación de alto nivel entraron en escena como herramientas poderosas.

A lo largo de este periodo aparecieron muchos métodos de diseño, el más conocido fue el diseño estructurado Top-Down (o diseño compuesto), el cual fue influenciado por lenguajes de alto nivel tales como FORTRAN (Formula Translation) y COBOL (Common Business Oriented Language) los cuales tienen como unidad de descomposición fundamental al subprograma, por lo tanto la estructura de los programas escritos en estos lenguajes asemeja a la estructura de un árbol, en la cual los nodos más cercanos a la raíz representan procedimientos complejos; que están compuestos por subprocedimientos menos complejos que intervienen parcialmente para cumplir la tarea del nodo padre; del mismo modo todos los nodos representan procedimientos que a medida que se alejan más de la raíz comprenden subprocesos menos complejos. Es de esta manera como se ve reflejada la idea principal del método estructurado, dividir una tarea en subtareas o pasos cada vez más sencillos.

Ya para la década de los ochenta aquellas computadoras de grandes capacidades habían evolucionado y aunque el valor de la programación estructurada sigue en uso [Stein,1988] señala que la programación estructurada debe abandonarse cuando el tamaño de las aplicaciones excede las cien mil líneas de código, cabe señalarse que aunque han existido muchos métodos de diseño interesantes estos pueden agruparse en tres categorías definidas de la siguiente manera[2]:

- Diseño estructurado Top-Down. (Yourdon, Constantine, Wirth, Hoare y Dijkstra principalmente)
- Diseño dirigido a los datos. (Jackson, Warnier y Orr principalmente)
- Diseño orientado a objetos.(Booch y Yourdon)

El diseño dirigido a los datos tiene como base ver la estructura del sistema como un mapeo de entradas a salidas, mientras que el diseño orientado a objetos tiene como base la idea de ver la estructura del sistema como una colección de objetos que cooperan y que tienen sus propias responsabilidades, al tratar a cada objeto como un ejemplar de una clase dentro de una jerarquía de clases, el diseño orientado a objetos refleja directamente el modo de operación de

[2] Sommerville, I. "Software Engineering" 3.era. Edición, Workingham, England, Addison Wesley 1989

los lenguajes de alto nivel más recientes, tales como Smalltalk, Object Pascal, C++, CLOS (Common Lisp Object System) y Ada.

Entre estos diferentes métodos de diseño existen elementos en común, entre los que tenemos a los siguientes:

NOTACIÓN	LENGUAJE PARA EXPRESAR CADA MODELO.
Proceso	Directrices para la construcción ordenada del modelo.
Herramientas	Artefactos que eliminan el tedio de la construcción del modelo y refuerzan las regla de los modelos mismos, de tal manera que los errores e inconsistencias sean expuestos.

III.2.2 EL MODELO DEL DISEÑO.

El proceso de construcción como mencionamos en la introducción, arroja dos modelos, el modelo de diseño y el modelo de implementación, así que el proceso de construcción se divide en dos fases : diseño e implantación. El modelo del diseño comprende una mayor formalización y refinamiento del modelo de análisis donde las consideraciones del modelo de implementación han sido contempladas.

En general se recomienda seguir tres pasos para obtener el modelo del diseño.

El primero es identificar el medio ambiente de la implantación lo que incluye identificar e investigar las consecuencias que el medio ambiente de implantación tendrá sobre el diseño. Aquí deben tomarse todas las decisiones de implementación, tales como de que manera se incorporara el sistema manejador de bases de datos (DBMS, por sus siglas en inglés) en el sistema, qué librerías de componentes se utilizarán y cómo manejar la comunicación entre procesos, cómo se llevara a cabo el manejo de errores o cómo se llevara cabo la recolección de basura. Se recomienda que estas consideraciones se vayan tomando en cuenta desde la fase de análisis.

El segundo es incorporar estas conclusiones y desarrollar un primer enfoque para un modelo de diseño. Aquí usamos el modelo del análisis como base para traducirlo en el modelo de diseño que cumpla con el medio ambiente actual de la implementación.

Y por último describir como interactúan los objetos en cada caso específico. Aquí se formaliza el modelo de diseño para describir todos los estímulos enviados por los objetos y como una operación maneja a un objeto.

La actividad de implementación como su nombre lo dice es la fase en la que se implementan los objetos. Las especificaciones así como las operaciones y los atributos los obtendremos del modelo de diseño.

En el modelo de diseño se depura al modelo del análisis a la luz del medio ambiente de implantación. Aquí se definiran las interfases de los objetos y la semántica de las operaciones, al mismo tiempo que se definirá como manejar las características de los lenguajes de programación, de los DBMS, etc.

El modelo de diseño se compone de bloques (objetos del diseño), los cuales servirán para mostrar como se ha diseñado el sistema y después serán implementados en código fuente. Tales bloques sirven para abstraer el comportamiento real del sistema, luego servirán para hacer la implementación, algunas veces un bloque es implementado con una única clase, pero, es frecuente que se necesiten más de una clase para representar un bloque. Dependiendo del lenguaje que vayamos a utilizar para la implementación tendremos que hacer corresponder a un módulo con una clase o paquete (por ejemplo en los lenguajes orientados a objetos se relaciona a un módulo con una clase, pero en Ada se relaciona a un módulo con un paquete).

Naturalmente el primer paso para crear un modelo de diseño es convertir cada módulo encontrado en el modelo del análisis en un bloque del modelo de diseño, lo que nos asegura la correcta rastreabilidad de nuestro modelo: es decir que nuestro modelo de diseño surge del modelo de análisis. Durante la fase de análisis se busco lograr una estructura robusta en el modelo del análisis, de ahí que durante el diseño, a medida que se van incorporando nuevos objetos en el sistema estos últimos no afectan a los ya existentes, la propiedad de rastreabilidad es una de las más importantes durante el desarrollo de sistemas ya que todos sabemos que los sistemas van evolucionando con el paso del tiempo y por la influencia de muchos otros factores. Por lo anterior siempre que surjan cambios ya sea de requerimientos o como respuestas a problemas de operación siempre será necesario saber en que parte del código fuente es necesario hacer los cambios, por lo tanto es recomendable el propiciar una gran cohesión y evitar el acoplamiento entre clases para que los cambios no afecten muchas partes del sistema.

Dijimos que el primer paso para convertir del modelo del análisis al modelo del diseño es tomar el diagrama del modelo del análisis y convertir a todos los objetos en bloques (cambiar círculos por rectángulos), bueno, aunque este primer paso parezca demasiado sencillo es necesario aclarar que a pesar de que se haya concluido el modelo del diseño, el modelo del análisis siempre deberá de ser guardado como referencia para el futuro ya que muestra un punto de vista conceptual del sistema a desarrollar y está desarrollado en términos lógicos, cuando esté terminado el modelo del diseño mostrará como esta construido el sistema.

Una de las metas del diseño es conservar la estructura lógica del diseño, es decir, no violarla innecesariamente. En principio queremos que también cuente con una estructura robusta, por lo que la semántica de los bloques debe reflejar a la semántica de los objetos que existen en el sistema, por ejemplo, debemos cuidar que sin importar con que lenguaje de programación se trabaje, todas las relaciones (tales como extensión o herencia) entre los objetos sean implementadas de alguna manera.

A medida que vayamos avanzando en nuestro diseño veremos que algunos bloques tendrán que ser divididos para poder hacer más accesibles algunas de sus partes que así lo requieran. Al respecto [Jacobson,1992] de acuerdo con Lavendel[3] nos dice que es cualitativamente mejor evitar errores de construcción mediante la reducción de la complejidad en fases tempranas que buscar las fallas del sistema y luego corregirlas cuando el sistema ya ha sido concluido.

[3] Levendel, Y. "Reliability Analysis of Large Software Systems: Defect Data Modeling." IEEE Transaction On Software Engineering 16(2) 1990.

Para poder considerar las restricciones de implementación que afectarán al sistema es necesario primero identificarlas, lo que es recomendable hacerse lo más pronto posible. A continuación trataremos el tema del medio ambiente de la implementación.

III.2.3. EL MODELO DE LA IMPLEMENTACIÓN.

Uno de los cambios más comunes durante el desarrollo de un producto es el cambio del medio ambiente de implementación. Dedido a ello se recomienda manejar este tema de la misma manera que las otras partes del sistema, lo que implica que debe evitarse al máximo que los objetos sean dependientes de las restricciones del medio ambiente.

Para poder cambiar partes en el medio ambiente destino, estas partes deben estar encapsuladas en un nuevo bloque, así que será necesario incluir nuevos bloques que manejen de manera aislada las posibles diferencias de implementación, por ejemplo, si sabemos que nuestro sistema puede correr en medios ambientes diferentes, será mejor que todas las interfases que nuestro sistema necesite con estos medios ambientes sean creadas en bloques aparte. Ver. Fig. 21

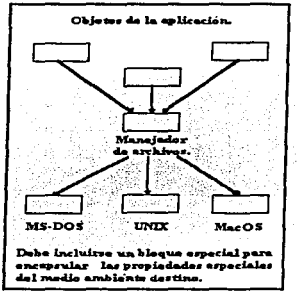


FIG. 21 MODELO DE IMPLEMENTACIÓN.

Otro factor a considerar es en que lenguaje de programación se llevará a cabo la implementación. Al pensar en un lenguaje de programación específico se manejan ciertos conceptos, por lo que será necesario hacer que esos conceptos sean traducidos en un lenguaje de programación particular. Como ejemplo de estos conceptos tenemos la herencia, herencia múltiple, polimorfismo, verificación de tipos, etc. También será necesario considerar las estrategias para el manejo de errores de ejecución. Algunos lenguajes proporcionan características que incluyen el manejo de errores, por ejemplo, Ada maneja excepciones e Informix tiene mecanismos automáticos de manejo de errores ; algunos lenguajes cuentan con métodos de recolección de basura ; otros no, etc. En fin todas estas características deben ser

tomadas en cuenta cuando se trata de incluir en el diseño las restricciones o las potencialidades que surgen del medio ambiente en que residirá nuestro producto.

Otro factor que afecta el diseño son los productos disponibles tales como sistemas manejadores de bases de datos (Data Base Management Systems), sistemas de manejo de interfases con el usuario (User Interfase Management System), utilerías de redes y muchos otras aplicaciones desarrolladas interna o externamente que serán utilizadas durante el desarrollo, así mismo, los productos o herramientas que se utilizan durante el desarrollo tales como compiladores, depuradores, preprocesadores, etc. constituyen otro factor que incluye ciertas implicaciones de diseño. La decisión de qué herramientas se utilizarán para la construcción del producto debe sustentarse en un estudio en el que se analice qué componentes contienen y qué funcionalidad ofrecen. De la misma manera será necesario familiarizarse con ellas para obtener un mayor aprovechamiento.

Además pueden existir muchos otros factores que afecten el desarrollo, pero ya dependerá de cada caso en particular, de la experiencia del equipo de desarrollo y de que decisiones se tomará. El hecho es que la posibilidad de cambio está siempre presente, y que sucederá, pero estos cambios deben estar justificados y documentados para mantener la rastreabilidad.

Se considera que es bueno introducir nuevos bloques durante el diseño cuando estos serán utilizados para manejar los aspectos cambiantes que incluye la implantación, pero durante el diseño no es recomendable incluir nuevos bloques para representar nuevos comportamientos o funcionalidad ya que estas características deben surgir durante la fase de análisis.

El borrado de bloques durante el diseño es un asunto más delicado, si se borran bloques con el afán de adaptar el sistema para su implementación está bien hecho pero si se borran por cualquier otra razón que involucre alterar la estructura lógica del modelo hay que tener cuidado..lo mismo debe decirse de la separación o unión de bloques ya que obviamente cualquier cambio afectará la robustez del sistema.

Los cambio más frecuentes suceden en las relaciones de asociación entre objetos, por ejemplo, se puede anexar una relación de extensión entre dos objetos, ésto normalmente se hace cuando se desea que el comportamiento de un objeto A sea incorporado en el comportamiento de un objeto B, de donde decimos que A extiende el comportamiento de B. Fig. 22

Las asociaciones de herencia deben manejarse con cuidado cuando se sabe que el lenguaje que se utilizará para la codificación no soporta esa asociación, la manera en que normalmente se implementa esta asociación es con una asociación de comunicación entre los bloques.

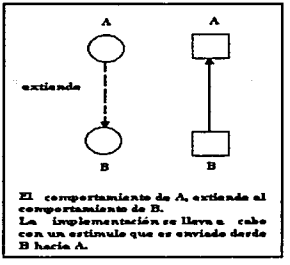


FIG. 22 RELACION DE ASOCIACION ENTRE OBJETOS

DIAGRAMAS DE INTERACCIÓN.

Una vez identificada la arquitectura del sistema describiremos como se comunicarán los bloques, lo cual se hace diseñando las casos de uso que serán usados para cada comportamiento requerido, para ello usaremos las definiciones de los usos que escribimos durante la fase de análisis. Estos casos de uso constituyen el pilar de la ingeniería de software orientada a objetos ya que controlan la fase de análisis y generan el modelo del análisis, se diseñan durante la construcción del sistema y definen los requerimientos externos de los bloques, serán examinados durante la fase de pruebas y serán la base para la preparación de manuales.

Por cada caso de uso concreto dibujaremos un diagrama de interacción, el cual describe como poner a disposición los servicios de un caso de uso por medio de objetos de comunicación y como los diferentes objetos emplearán a los casos de uso durante su mutua interacción. A medida que se va dibujando el diagrama de interacción también se van definiendo los estímulos (incluyendo parámetros) que un objeto manda a otro el propósito principal del diseño en esta parte es definir las interfases y protocolos de los bloques.

El modelo del análisis describe que comportamiento tiene cada objeto, toca a la fase de diseño refinarlo mostrando en el diagrama de interacción como se comportan los objetos en cada caso especifico, por lo que debemos definir con exactitud como se comunicaran los objetos.

El diagrama de interacción se construye de la manera siguiente para cada caso de uso especifico se busca que bloques participan en él, por cada bloque se dibuja una columna, el orden en que se dibujen las columnas no tiene ningún significado, únicamente se recomienda que se dibujen de la mejor manera atendiendo a la claridad del diagrama, en caso de que aparezcan múltiples instancias de un bloque estas pueden dibujarse en diferentes columnas o en una sola sin hacer a un lado la búsqueda de claridad.

En la mayoria de los diagramas de interacción se incluye una columna extra que representa el medio circundante, que se utiliza para describir lo que esta fuera del sistema. A esta columna se le llama ***"Limite del sistema"***.

La segunda columna desde la izquierda es utilizada como eje de tiempo y se interpreta que corre hacia abajo. La primera columna desde la izquierda es utilizada para la descripción de las operaciones o secuencias que se hará preferentemente en texto normal (para no depender de algún lenguaje en especifico).

Una vez que tenemos las operaciones en una columna y los bloques como encabezados de sus respectivas columnas lo que haremos será dibujar rectángulos en la intersección de todos los renglones (operaciones) con las columnas que representen a un determinado bloque, con el propósito de que el diagrama aparezca más claro podemos aislar cada operación con líneas punteadas.

Definición de estímulos.

Los diagramas de interacción son controlados mediante eventos, un evento da pie al inicio de una operación, por lo que constituye un estímulo para un objeto. Un estímulo se representa en los diagramas de interacción mediante una flecha horizontal que nace en la columna de donde parte el estímulo y termina en la columna del bloque que recibe el estímulo. Los estímulos que provienen del exterior nacen en la columna reservada para el límite del sistema.

Cuando se trabaja en el diseño se define que estímulos es capaz de recibir cada objeto, esta definición comprende el nombre y parámetros de cada estímulo, la definición de estímulos es una de las partes mas complicadas del diseño ya que lo más seguro es que trabajen diferentes personas en el proyecto y que algunas puedan diseñar los estímulos y otros necesiten valerse de los objetos por lo que será necesario que conozcan las interfases.

A continuación mencionaremos lo que el diseñador debe tener en mente en el momento de definir un estímulo.

La reutilización es más factible cuando se manejan menos parámetros, a la vez que simplifica el uso y aumenta la probabilidad de que dos estímulos sean similares, recordemos que mientras más especifico se vuelve un objeto más disminuye su probabilidad de reutilización. Cuando tengamos un estímulo que maneje muchos parámetros lo mejor será descomponerlo en estímulos más sencillos.

Los estímulos que involucren comportamientos similares deben tener nombres similares, Ya que de este modo es más fácil ver la similitud que existe entre ellos.

El nombre de los estímulos debe reflejar la distribución de responsabilidades entre los bloques, el lograr que los módulos sean independientes de otros módulos (evitar algunos tipos nocivos de acoplamiento) es una meta en cualquier diseño.

Los objetos pueden tener diferentes comportamientos dependientes de diferentes estímulos, los más importantes son llamados básicos y los otros son llamados alternos, mientras más cursos alternos describamos más robusto será el sistema.

Los estímulos pueden tener diferentes clasificaciones, cuando se implementan como llamadas normales a procesos se les llama mensajes y cuando comunican a dos procesos se les llama señales, se utilizan flechas diferentes para representar mensajes y para representar señales los primeros se representan con una flecha cerrada y las segundas con una flecha abierta. Fig. 23

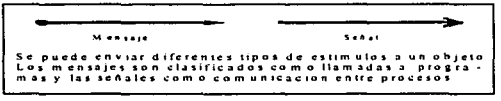


FIG. 23 MENSAJES Y SEÑALES.

Dentro de los diagramas de interacción es posible incluir las asociaciones de extensión, estas se incluyen por medio de ensayos. Un ensayo es una posición dentro de una descripción donde puede insertarse un comportamiento; las descripciones en las cuales sea necesario incorporar un nuevo comportamiento serán etiquetadas como ensayos en su descripción donde se redactará en que momento sucede o cuales son las condiciones para que sea incluido el comportamiento de otro objeto.

Es recomendable también homogeneizar los estímulos ésto significa que, para maximizar la reutilización y facilitar el uso de éstos debemos de procurar tener los menos posibles sin mermar la funcionalidad, al tiempo que debemos mantener la robustez y la capacidad de reutilización.

Los diagramas de interfaz muestran la manera en que se relacionan los objetos unos con otros, al mismo tiempo que nos permiten observar claramente cual es la estructura de nuestros casos de uso, en la siguiente Fig. 24 se muestra un ejemplo de un diagrama de bloques con estructura centralizada.

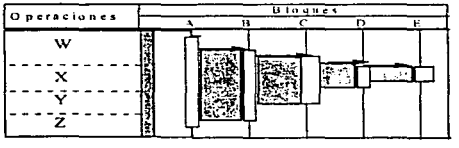


FIG. 24 ESTRUCTURA CENTRALIZADA

En la estructura centralizada podemos observar como la ejecución de las diferentes operaciones es controlada por un sólo bloque, el cual se encarga de decidir cuando entra en acción cada uno de los demás bloques.

A diferencia de la estructura centralizada, en una estructura descentralizada se observa que los bloques son menos dependientes de un bloque controlador y que pueden tomar acciones propias como enviar estímulos a otros bloques.

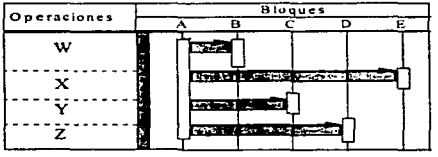


FIG. 25 ESTRUCTURA DESCENTRALIZADA.

Ahora surge la pregunta de cual de estas estructuras es la más apropiada, generalmente se piensa que la estructura descentralizada responde de mejor manera al paradigma orientado a objetos debido a que bajo ese módelo la responsabilidad debe distribuirse. En realidad a nosotros lo que debe importarnos más es lograr una estructurar donde los cambios que se sucedan sean fáciles de llevarse a cabo.

Si se desea manejar cambios tales como cambiar la manera en que se ejecuta una operación lo que debe encapsularse es a la manera misma en que se ejecuta, de modo que se obtendrá una estructura descentralizada, si en cambio, lo que se desea es soportar cambios en cuanto al orden en que se realizarán las operaciones entonces es el control de éstas el que cambiará y por lo tanto será de mayor utilidad el contar con una estructura centralizada, [Jacobson,1992] nos comenta algunas caracteristicas que nos ayudarán a elegir que tipo de estructura es más apropiada dependiendo de sus características.

Una estructura de control descentralizada es apropiada cuando las operaciones tienen una conexión fuerte(*)[4], cuando se ejecuten siempre en el mismo orden y cuando se desee encapsular el comportamiento. Por otro lado se sugiere una estructura centralizada cuando el orden de las operaciones puede cambiar y cuando es posible que se inserten nuevas operaciones.

[4](*) Se dice que existe una conexión fuerte entre objetos si estos están bajo la relación consiste de (un conjunto consiste de elementos), cuando componen una jerarquía de información (Constitución-código-fiscal), cuando representan una relación temporal fija (Emisión de una resolución-notificación de esta-pago-Pocedimiento administrativo de ejecución) o cuando forman una jerarquía conceptual (Ser vivo-animal-vertebrado-mamifero).

DISEÑO DE BLOQUES.

La interfaz del bloque.

Una vez diseñados todos los casos de uso para un bloque especifico es el momento apropiado para diseñar el bloque ya que extraeremos de los casos de uso la descripción de todos los requerimientos externos del bloque. Aunados a éstos y a los requerimientos que descubrimos durante la construcción del modelo del análisis tendremos lo necesario; la codificación de los bloques, o lo que es lo mismo el modelo de implantación de los bloques, debe postergarse hasta que las interfases de los bloques comiencen a estabilizarse.

En el momento de implementar los bloques se sugiere que primero se implementen los bloques ancestros y después los bloques descendientes.

Usaremos aquí los diagramas de interacción para analizar de que manera participan los bloques y para extraer las operaciones definidas para cada bloque.

Al ir a través de los diagramas de interacción obtendremos la interfaz de los bloques y la primera descripción de las operaciones (columna de la extrema izquierda), también nos servirá de apoyo el modelo del análisis en el cual encontramos otros requerimientos tales como los atributos y tipos que debe tener cada bloque, otros requerimientos los obtendremos de la especificación de los requerimientos estos últimos son las necesidades de memoria o de ejecución en tiempo real.

Como veremos en la práctica generalmente un bloque corresponde a una clase, aunque a veces por necesidades de implementación éstos tengan que ser implementados en varias clases.

Los bloques también son mecanismos de abstracción y encapsulamiento, por lo que es conveniente que los manejemos de esa manera. Lo lograremos expresando la interfaz del bloque de modo que todos los módulos objeto que se relacionen con su interfaz sean públicos y los métodos implementados internamente constituyan la parte privada.

Platicaremos a continuación un poco acerca de la manera en que se implementarán los objetos.

IMPLEMENTACIÓN DE OBJETOS.

Una vez que se ha decido que objetos se incluirán en el sistema, que se ha creado una jerarquía de clases apropiada, que se han definido las relaciones que tienen unos objetos con otros y se ha identificado que operaciones o que responsabilidades tendrá cada objeto ; lo que queda por hacer es implementarlo. Lo cual incluye decidir que estructuras de datos contendrá, decidir el número de instancias de que podrán ser creadas, etc. Como puede observarse al llegar a este punto es necesario que todas las fases anteriores hayan sido consideradas; la experiencia revela que un buen trabajo hecho durante la fase de análisis será de gran beneficio para la fase siguiente, además los objetos que surgen del dominio de la aplicación son generalmente más estables [Jacobson,1992].

Identificando los atributos.

El analista debe preguntarse que atributos son aplicables para cada instancia de un objeto o estructura de clasificación, para lo que es necesario remitirse al las descripciones del espacio del problema e interactuar con el usuario. Es posible que se dificulte la elección de atributos de un objeto especifico pero remitámonos al espacio del problema y consideremos que cosas son importantes para el sistema, tal vez un objeto del mundo real tenga muchos atributos pero no todos serán de interés para nuestro sistema, por otro lado es necesario que los atributos sean unidades de datos o que formen grupos naturales de elementos de datos.

En una fase temprana del análisis pudieron haberse identificado de manera casi casual algunos atributos de los objetos, estos pueden reconsiderarse en este momento bajo la luz de las restricciones del modelo de implementación.

La importancia de la identificación de atributos.

La identificación de atributos proporciona un medio conveniente de describir a una instancia simple de un objeto, además de sus conexiones con otros objetos, cada objeto y estructura de clasificación necesita de un identificador, así que por convención cada objeto y estructura de clasificación cuenta con un identificador implícito además de identificadores para las conexiones, la razón principal para proporcionar identificadores para estos elementos es facilitar la especificación de servicios.

Después de haber nombrado los atributos es necesario volver a revisar los objetos, siempre que se añaden atributos es necesario revisar los objetos, a continuación platicaremos un poco sobre los puntos que es necesario revisar para concretar un modelo apropiado.

Anteriormente platicamos sobre la posibilidad de que en algunos objetos podía presentarse que tuviéramos atributos que para ciertos elementos no fueran aplicables, si esto sucede es necesario considerar estructuras de clasificación adicionales.

Si un objeto o una instancia de una estructura de clasificación tiene sólo un atributo, entonces el modelo puede revisarse para que refleje un mayor nivel de abstracción, a manera de que tratemos de incluir ese atributo simple en un nivel superior.

Ahora el tercer paso del refinamiento sugiere revisar aquellos valores repetidos que pudieran presentarse en diferentes instancias, lo que conviene hacer aquí es ver si es conveniente crear un objeto adicional, pero sólo en el caso de que este nuevo objeto califique como tal de acuerdo con las reglas de estado, comportamiento e identidad, no es correcto añadir objetos con el único propósito de normalizar.

Descripción de atributos.

Una vez que hemos identificado a los objetos pensemos que papel juegan dentro del modelo y en consecuencia que datos relacionados con el objeto son de interés para el sistema, estos datos son los atributos del objeto y le son útiles para describirlo. Por medio de los atributos de un objeto podemos aclarar lo que representa el objeto dentro del modelo que estamos construyendo ya que estamos añadiendo mayor detalle a la abstracción.

Los atributos describen la información que esta escondida en el objeto (la cual únicamente podrá ser manipulada por los servicios de ese objeto), los atributos de un objeto y sus servicios exclusivos se manejan como un todo, de modo que si otra parte del sistema necesita información o necesita que se haga cierta manipulación de los datos, tendrá que mandar un mensaje apropiado que ponga en marcha un servicio del objeto. De manera que siempre ocultaremos información y emplearemos abstracción.

Es común cuando se hace un buen análisis orientado a objetos que los objetos permanezcan mientras que sus atributos cambien, un método sencillo de definir atributos es identificarlos, situarlos en el lugar adecuado dentro de la estructura de clasificación, identificar y definir las relaciones entre varias instancias, revisar los objetos y especificar las restricciones entre las relaciones *atributos-instancias*.

Las operaciones que debe llevar a cabo un objeto saltan a la vista cuando se consideran las interfases entre los objetos, también se pueden identificar directamente desde la aplicación al pensar en que puede hacerse con los elementos que se tiene, las operaciones básicas que habrá que considerar son las relacionadas con la creación, añadidura y destrucción de objetos, otras operaciones más complejas podrían ser operaciones de emisión de reportes etc.

Especificación de atributos.

Es necesario especificar claramente los atributos, dependiendo de que es útil para su comprensión y de cuales son los requerimientos del cliente.

Antes de imponer las restricciones es necesario ubicar a los atributos dentro de alguna de las siguientes categorías.

Descriptivo	Su valor es establecido y conservado mediante cambio, borrado, selección y anexión de instancias.
Definición.	Su valor es potencialmente aplicable a más de una instancia de un objeto o estructura de clasificación.
Siempre derivable	Su valor es derivable de otros datos en cualquier momento (de ahí que no requiera que su valor sea mantenido con el tiempo).
Ocasionalmente derivable	Su valor es ocasionalmente derivable de otros datos (de ahí que requiera que su valor sea mantenido con el tiempo).

Se tendrá que imponer algunas restricciones a los atributos (valores permitidos, rango límite, unidad de medida y/o precisión), especificar las restricciones de conexión de las instancias desde la perspectiva de una instancia y sus restricciones de mapeo hacia otras, también se debe incluir el mapeo de restricciones conveniente a cualquier conexión de ensamblaje en la que un objeto participe total o parcialmente.

Colocar los atributos en el lugar adecuado dentro de la estructura.

Es este el momento de valernos de la herencia dentro de la estructura de clasificación, si existe generalización/especialización lógicamente debemos poner los atributos comunes en los niveles superiores y poner la especialización en los niveles más bajos. Algunos lenguajes de programación aceptan la herencia múltiple y la anulación pero durante la fase de análisis es más sencillo evitar la herencia múltiple para lograr un modelo más sencillo.

IDENTIFICAR Y DEFINIR LAS RELACIONES ENTRE VARIOS OBJETOS.

Para definir relaciones entre instancias es necesario que primero dibujemos en un diagrama las líneas que reflejen las conexiones existentes entre las instancias de los objetos existentes dentro del espacio del problema, obviamente no tenemos que dibujar todas las relaciones existentes sino sólo aquellas acerca de las cuales el sistema deba estar enterado, dentro de un diagrama cada línea de conexión implica una conexión a través de un mensaje. Una línea de conexión representa un mapeo que asocia las instancias de un objeto o estructura con las instancias de otro.

Una vez que se han dibujado las conexiones entre instancias es menester pensar en la multiplicidad (cardinalidad) que maneja cada línea la cual puede ser 1:1 o 1:M la primera se representa poniendo una línea corta perpendicular cerca de cada objeto, la segunda a su vez se representa poniendo una línea corta perpendicular a la línea de relación del lado de la instancia que ocurre una vez y poniendo una línea triple cuando existen múltiples instancias. Después de haber definido la multiplicidad de las líneas relación es necesario definir la participación de cada una de las relaciones, esto se logrará al hacer la pregunta ¿Esta conexión es obligatoria u opcional? o lo que es lo mismo ¿Tiene sentido para una instancia de un objeto existir sin un instancia correspondiente del otro objeto?. En caso de que la relación sea obligatoria añada una barra simple al símbolo de multiplicidad y una "O" en caso de que sea opcional.

El último paso dentro de la identificación de relaciones es asegurarse de que no existan casos especiales, existen cuatro tipos de casos especiales, los cuales listamos a continuación:

1.- Conexiones a través de tres o más objetos o estructuras de clasificación.
2.- Conexiones muchos a muchos.
3.- Conexiones entre instancias del mismo objeto o estructura de clasificación.
4.- Múltiples conexiones entre dos objetos o estructuras de clasificación.

Las conexiones a través de tres o más objetos o estructuras de clasificación.

En el caso de que una o más conexiones sean opcionales puede ser necesario añadir conexiones adicionales, lo anterior quiere decir que si existen casos especiales en los que se pueda presentar que generalmente haya instancias de objetos relacionadas por una tercera instancia intermedia, para los cuales sea posible que dos instancias estén relacionadas sin que necesariamente haya una instancia intermedia que las una, lo que debe hacerse es dibujar esta relación como una relación alternativa entre las instancias involucradas.

Conexiones muchos a muchos.

Para cada una de las conexiones muchos a muchos que aparezcan en el modelo es necesario verificar si algunos atributos realmente describen la relación entre los objetos, esto con el fin de verificar que los atributos no pertenezcan a uno de los dos objetos, si no que realmente pertenezcan a la relación entre ellos cuando están conectados, si así sucede entonces habremos descubierto un nuevo objeto a incluir dentro del modelo. Este nuevo objeto tendrá que representarse con participación obligatoria.

Conexiones entre instancias del mismo objeto o estructura de clasificación.

Una instancia de un objeto puede tener conexiones potenciales con otras instancias del mismo objeto, verifiquemos si la relación tiene atributos que la describan, si no los tiene úsese a la conexión de manera aislada, en otro caso introduciremos un nuevo objeto para capturar los atributos de la conexión.

Múltiples conexiones entre dos objetos o estructuras de clasificación.

Cuando encontramos más de una conexión entre dos objetos, debemos volver atrás y considerar el significado de las dos conexiones. Capturemos en uno o más atributos las diferencias de estas relaciones y añadámoslos a los objetos apropiados a fin de dejar una sola relación entre los objetos.

DEFINIENDO SERVICIOS.

Todo sistema de procesamiento de datos tiene dos partes fundamentales los datos y el procesamiento. Durante el ciclo de desarrollo que hemos planteado, el analista enfoca su atención primero hacia la definición de objetos, estructuras, atributos y relaciones entre instancias de los objetos, adicionalmente a través de los casos de uso se han descubierto algunos de los servicios y mensajes que atañen al sistema, en la fase de construcción se debe definir de manera apropiada el protocolo de los objetos.

En esta sección describiremos brevemente un método apropiado para definir los servicios que forman parte del protocolo de un objeto.

[Yourdon, 1990] define a los servicios como procesamiento a llevarse a cabo sobre el receptor de un mensaje. El tema principal aquí es definir el comportamiento de los objetos y de las estructuras de control, para lo que se sugiere emplear las tres estrategias siguientes para identificar servicios :

Causa Inmediata	Repuesta dado un estado y un evento.
Historia	Comportamiento histórico del evento.
Función	Servicios básicos

Otro tema relacionado con la definición de servicios es la comunicación entre los objetos necesaria para activar los servicios.

Los servicios y los mensajes se describen mediante una especificación textual de los requerimientos de procesamiento, los servicios más que ser una descripción detallada de lo que sucede en la realidad indican que procesamiento será proporcionado por un objeto o estructura de clasificación, mucha de esta información surge de los casos de uso y de los diagramas de interacción, pero es en el momento de su definición donde en realidad cobran una existencia más acorde con las restricciones del medio ambiente.

La estrategia que sugerimos que se maneje aquí comprende cuatro pasos : el primero es identificar los servicios, lo que permitirá que anexemos a nuestro diagrama los nombres de los servicios que hayamos encontrado; el segundo paso es buscar servicios adicionales, con lo que podremos poner más nombres de servicios en nuestro diagrama; el tercer paso está

relacionado con la identificación de los mensajes a ser enviados de objeto a objeto y por último será necesario especificar los servicios, es decir, desarrollar los requerimientos de procesamiento. En las siguientes páginas platicaremos un poco mas acerca de los pasos anteriormente mencionados.

Identificar los servicios.

Recuérdese que anteriormente hablamos acerca de las cinco clases de operaciones que podían llevar a cabo los clientes sobre un objeto (modificar, seleccionar, iterar, construir y destruir), para algunos autores estas operaciones pueden clasificarse dentro de un grupo de servicios fundamentales. Los grupos fundamentales propuestos por [Yourdon,1990] son los siguientes : Ocurrir, Calcular y Monitorear.

Todos los modelos se valen de servicios de ocurrencia, algunos modelos del análisis orientado a objetos utilizan servicios de cálculo y algunos otros modelos del AOO usan servicios de monitoreo.

- *Servicio de ocurrencia.*

El servicio de ocurrencia establece y mantiene a una instancia de un objeto o estructura de clasificación, todo objeto o estructura de clasificación requiere de un servicio de ocurrencia ya que al estar contemplado dentro del modelo será necesario contar con él.

- *Servicio de cálculo.*

Un servicio de calculo es aquel, utilizado para responder a un mensaje de modo que el servicio mencionado ejecute un procedimiento, tal cálculo puede ser periódico o no, pero, es posible que el servicio de cálculo varie de acuerdo al cliente que requiere al servicio en cuyo caso debe incluirse dentro del servicio una estructura de clasificación que responda de manera adecuada a la petición del cliente.

- *Servicio de monitoreo.*

Un servicio de monitoreo es aquel que se encarga de verificar el estado de un elemento del sistema, este tipo de servicio es más usual en sistemas de tiempo real en los cuales aparecen modelados objetos del mundo real que requieren de verificaciones frecuentes para conocer su estado. Por otro lado tenemos que aqui también puede presentarse la necesidad de que el servicio tenga variaciones, como mencionamos anteriormente, será necesario incluir una estructura de clasificación en la que aparezcan en la parte superior los métodos comunes y en la parte inferior los métodos especializados.

Identificando los servicios (Estrategia secundaria)

- *Comportamiento histórico de un objeto.*

El comportamiento histórico de un objeto proporciona una visión del comportamiento de un objeto a través del tiempo. Aqui se sugiere un método para identificar servicios con base en el comportamiento histórico del problema.

Primero se sugiere definir la secuencia histórica de un objeto. Lo cual se puede lograr haciendo esquemas, primero sencillos, de la secuencia histórica de un objeto. Por ejemplo, uno donde pueda apreciarse su creación (construcción), su cambio o selección (modificar, seleccionar e iterar) y por ultimo su destrucción. Después es necesario que se analice cada uno de los servicios con el afán de detectar si existen variaciones dentro de tales servicios; posteriormente añadiremos a esta secuencia básica aquellos otros eventos a los que sea necesario que responda el objeto u estructura de clasificación.

- *Respuesta estado-evento.*

Una estrategia secundaria es la llamada respuesta estado-evento, la cual es utilizada para descubrir servicios adicionales para un objeto o estructura de clasificación y consiste en primer lugar en definir los estados importantes del sistema, después para cada estado es necesario listar los eventos externos y las responsabilidades requeridas, por último deberemos expandir los servicios (mensajes de conexión).

El primer paso consiste en responder a la pregunta ¿cuales son los principales estados del comportamiento para el sistema?. Una vez logrado lo anterior listaremos los eventos externos y las respuestas requeridas para cada estado. Lo anterior lo podemos representar en un diagrama estado-evento-respuesta (en sustitución de un STD diagrama de transición de estados), y por último será necesario extender los servicios para proporcionar el procesamiento adecuado en respuesta a cada evento.

¿Como identificar las conexiones a través de mensajes?

Una conexión a través de mensajes involucra respuesta a eventos y flujo de datos, una conexión a través de mensajes representa un mensaje enviado, así como una respuesta recibida.

Una conexión de mensajes mapea a una instancia de un objeto con otra instancia de otro objeto, el emisor envía un mensaje a un receptor para estimular la ejecución de un método. La necesidad de procesamiento es mencionada dentro de los servicios del emisor y es definida dentro de los servicios del receptor. Lo que se pretende aquí es manejar una disciplina en la que se cree una interfaz muy estrecha entre la fuerte encapsulación de los atributos y los servicios exclusivos sobre aquellos atributos utilizando la estructura de objetos y clasificación.

Es aconsejable empezar añadiendo mensajes de conexión entre los objetos y estructuras de clasificación ya conectadas mediante conexiones entre instancias. Examínese al mismo tiempo los objetos y estructuras de clasificación, junto con sus atributos encapsulados, buscando los servicios necesarios para conseguir los valores de los atributos, o para conseguir los procesamientos necesarios llevados a cabo en alguno de ellos. Es necesario documentar la conexión a través de mensajes en la especificación de los servicios del emisor; y documentar el servicio correspondiente llevado cabo en la especificación de los servicios del receptor.

Especificando los servicios

En las siguientes páginas presentaremos una manera de especificar los servicios utilizando un método comúnmente utilizado durante el desarrollo de sistemas.
Este método consta de seis pasos:

Enfocarse en el comportamiento requerido observable desde fuera.
Uso de una plantilla.
Anexar diagramas para simplificar la especificación de servicios.
Añadir cuadros de soporte.
Desarrollar narrativas de servicios.
Agrupar la documentación.

- *Enfocarse en el comportamiento requerido observable desde fuera.*

Para lograrlo es necesario preguntarse para cada servicio, si este puede ser observado externamente, considérese el escribir una prueba para cada requerimiento, lo que ayudara a depurar la declaración de los requerimientos y hará énfasis en los pasos del proceso.

Una sugerencia aquí es la de utilizar verbos sistemáticamente, a fin de proporcionar un énfasis especial a los requerimientos observables externos.

- *Uso de una plantilla.*

Recomendamos utilizar una plantilla para desarrollar una lista de requerimientos. un ejemplo de una lista tal es el siguiente:

Especificación <Nombre del objeto>

Atributo descriptivo<...>

Definición de datos<...>

Atributos siempre derivables< >

Atributos derivables ocasionalmente< ... >

Entradas externas al sistema < ... >

Salidas externas del sistema< ... >

Restricciones entre conexiones de instancias< >

Respuesta estado evento< >

Comportamiento histórico del objeto< >

Notas< >

intento/propósito< >

servicio< >

servicio< >

servicio< >Fin especificación

Donde :

Atributos descriptivos.	Son aquellos atributos cuyos valores son manipulados mediante servicios de ocurrencia (Añadir, cambiar, borrar, seleccionar). Listense también las restricciones de los atributos.
Atributos de definición de datos.	Son aquellos atributos cuyos valores son potencialmente aplicables a más de una instancia de un objeto o estructura de clasificación.
Atributos siempre derivables.	Son aquellos atributos para los cuales siempre es posible obtener su valor por lo que no necesitan ser almacenados. Listense también las restricciones de los atributos.
Atributos derivables ocasionalmente.	Estos atributos tienen valores que no pueden ser obtenidos en cualquier momento por lo que es necesario almacenar su valor. Listense también las restricciones de los atributos.
Entradas externas al sistema.	Son datos proveniente de un dispositivo o sistema externo. Listense también las restricciones de los datos.
Salidas externas del sistema.	Son los datos que proporciona el sistema a dispositivos de salida o sistemas externos.
Restricciones de conexiones entre instancias.	Son las restricciones de conexión de la instancia.
Respuesta estado evento.	Es la respuesta para un estado y un evento utilizada en la identificación de los servicios.
Comportamiento histórico de los objetos.	Es un patrón de comportamiento utilizado en la identificación de los servicios.
Notas.	Análisis y consideraciones.
Intento/propósito.	Es el propósito del requerimiento.
servicio.	La especificación de un servicio, lista de requerimientos.

A la lista anterior se le pueden añadir otros elementos en la lista dependiendo de las necesidades especificas de cada proyecto, tales como **rastreabilidad, criticalidad, presupuesto**.

- *Anexar diagramas para simplificar la especificación de servicios.*

Otra manera de simplificar las especificaciones es mediante la inclusión de diagramas para guiar al lector a través de las especificaciones. Podemos utilizar diagramas de bloques, diagramas de flujo de datos, diagramas de maquinas de estados finitos etcétera.

- *Cuadros de soporte.*

Las dependencias entre objetos quedan comprendidas dentro de un modelo que muestre las conexiones a través de mensajes, pero en el caso de que se este tratando con sistemas de tiempo real es conveniente incluir algunos cuadros que mejoren nuestra especificación, algunos ejemplos de estas son los cuadros siguientes:

Servicios y estados aplicables.

Este cuadro resumirá el comportamiento dependiendo del estado.

Flujos críticos de ejecución.

Identificara las secuencias de respuesta estado-evento que sean de mayor importancia para el sistema, este procedimiento incluye el identificar aquellos eventos para los cuales exista un flujo de control que deba ser conocido por el proceso para que sea de importancia para el sistema.

Análisis de tiempos y tamaños.

Hacer estimaciones para cada servicio junto con un flujo crítico de ejecución

Por último será necesario acomodar la documentación, el producto de toda la definición de servicios debe contener lo siguiente :

Diagramas del análisis orientado a objetos.
Capas de Sujeto, Objeto, Estructura, Atributo y Servicios.

Depósito del análisis orientado a objetos.
(Una entrada por cada servicio o estructura de clasificación)

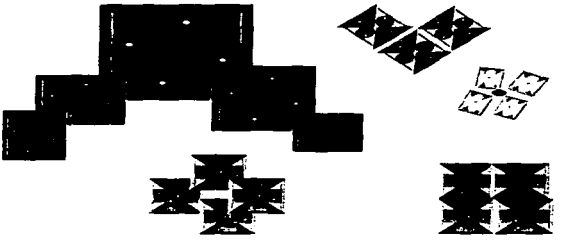
Cuadros de soporte (Si existe alguno)
Cuadro de estados y servicios aplicables, cuadro de flujos de ejecución críticos, cuadro de tiempos y tamaños.

BIBLIOGRAFÍA PARA EL CAPÍTULO III

[Parnas,1983]	Parnas, D. L., Clemens, P. C. y Weiss, D. M. "Enhancing Reusability With Information Hiding." ITT Proceedings of the Workshop on Reusability in Programming 1983.
[Stein,1988]	Stein, J. "Object Oriented Programming and Database Design." Dr. Dobb's, Journal of Software Tools for the Professional Programmer, No. 137, marzo 1988.
[Miller,1956]	Miller, G. "The magical number seven , plus or minus two: some limits on our capacity for processing information." The psychological review, Vol. 63(2) 1956.
[Yourdon,1990]	Yourdon, Edward & Coad, Peter. "Object Oriented Analysis." Englewood Cliffs, New Jersey, Printece Hall 1990.
[Yourdon,1991]	Yourdon, Edward & Coad, P. "Object Oriented Design." Englewood Cliffs, New Jersey, Yourdon Press Computing Series 1991.
[Booch,1987]	Booch, G. "Software Components with Ada." Benjamin Cummings Editions 1987.
[Jacobson,1992]	Jacobson, I. "Object-Oriented Software Engineering A Use Case Driven Approach." Addison-Wesley Publishing Company 1992.
[Martin,1993]	Martin, James. "Principles Of Object Oriented Analysis And Design." Englewood Cliffs, New Jersey, Prentice Hall International 1993.
[Mcabe,1989]	McCabe, Thomas & Butler, Charles. "Design Complexity Measurment and Testing." Communications of the ACM, 32(12) diciembre de 1989.
[Meyer,1988]	Meyer, B. "Object Oriented Software Construction." Prentice Hall; United Kingdom 1988.
[Fischer,1989]	Fischer, G. "Human-Computer Interaction in Software: Lessons Learned, Challenges Ahead." IEEE Software; enero 1989.
[Budd,1991]	Budd, T. "An Introducction to Object Oriented Programming." Reading Massachusetts, Ed. Addison Wesley 1991.
[Booch,1991]	Booch,G. "Object Oriented Design with Applications." The Benjamin Commings Publishing Company Inc. 1991.

CAPÍTULO IV

IMPLEMENTANDO LA REUTILIZACIÓN EN C++



"La programación orientada a objetos está de moda, pero es una buena moda de la que creemos, a diferencia de la mayoría de las modas, durará un buen tiempo. C++ es una buena ropa para vestir esta moda de los noventas."

Miguel Katrib Mora.

INTRODUCCIÓN.

En este capítulo cubriremos aquellos aspectos que provenientes del modelo orientado a objetos tienen una manera especifica de implementarse en el lenguaje C++. Debemos recordar que C++ incorpora algunos conceptos con las que no contaba el lenguaje C, es por ello que se considera a C++ como un superconjunto de C. Expondremos a lo largo del capítulo aquellas características que hacen de C++ un lenguaje apropiado para implementar la reutilización basándonos en el paradigma orientado a objetos. Empecemos presentando un vistazo al lenguaje.

IV.1 UN VISTAZO A C++.

C++ fue desarrollado por Bjarne Stroustrop en los laboratorios Bell de AT&T. Anteriormente muchos de los compiladores de C++ eran traductores de lenguajes que producían código en C (preprocesadores), actualmente la mayoría de compiladores sí convierten directamente de C++ a código objeto.

Existen ventajas y desventajas para ambos enfoques, el primero tiene la ventaja de que se puede utilizar algún compilador de C ya conocido e incorporar algunas funciones de código fuente de las bibliotecas ya conocidas. Por otro lado los verdaderos compiladores de C++ cuentan con la ventaja de generar código más eficiente, aunque representan la desventaja (para desarrolladores de compiladores) de que es más costoso depurar y actualizar programas con uso de compiladores que de traductores.

C++ es un lenguaje ideal para el desarrollo de sistemas por dos razones importantes. Primero porque conserva todas las características de C ; incluyendo esa sintaxis tan concisa, la riqueza del conjunto de operadores, etc. Y segundo por que C++ es capaz de implementar todas las características importantes de la programación orientada a objetos (excepto la recolección de basura). Específicamente, C++ proporciona tres herramientas importantes que son útiles durante el manejo de proyectos grandes, las cuales son la encapsulación, la descomposición jerárquica de clases y el enlace dinámico, éstas serán tratadas con mayor detalle a lo largo del capítulo.

Algunas ventajas que incluye C++ respecto de C son las que presentamos a continuación:

C++ incorpora el operador de resolución de ámbito (::), el cual se utiliza para especificar al compilador al momento de definir métodos cual es la clase a la que pertenece, además, es útil cuando dentro de un bloque particular existe una variable cuyo nombre es igual al de una variable global; como las variables locales tienen preferencia dentro de su ámbito sobre las variables globales, entonces cuando es necesario decir al compilador explícitamente que la variable deseada es la global se debe anteceder este operador al nombre de la variable.

La sobrecarga de funciones es otra de las ventajas que C++ ofrece, ésta se refiere a que podemos definir funciones con nombres iguales, las cuales pueden manejar distintos tipos de argumentos como parámetros formales. El compilador de C++ es capaz de saber a que función se hace referencia durante una llamada a través de la comprobación de los parámetros reales incluidos en la llamada a la función.

La más importante de las características que incluye C++ es la inclusión del especificador **"class"**, por medio del cual y junto con otras características hace posible la programación orientada a objetos. Una clase en C++ es similar a una estructura, sólo que además de contener datos contiene procedimientos que pueden declararse como públicos, privados o protegidos. En C cualquier estructura accesible a un programador permite la manipulación de sus componentes (acceso indiscriminado), en cambio en C++, valiéndose del uso de clases se controla el acceso a estas estructuras y componentes, es decir encapsula los componentes y los procedimientos.

Es oportuno mencionar que todos los lenguajes orientados a objetos tienen conceptos comunes tales como clases, paso de mensajes, métodos herencia, etc. y que éstos se asocian a diferentes términos dentro de cada lenguaje en particular.

En lo que resta del presente capítulo pondremos de manifiesto la manera en que el programador C++ puede implementar los diferentes conceptos del modelo orientado a objetos.

Daremos principio al desarrollo de este capítulo presentando una característica de la POO, por medio de la cual podemos hacer uso de datos abstractos.

IV.2 ABSTRACCIÓN Y ENCAPSULACIÓN.

El enfoque metodológico de programar haciendo uso de estructuras de datos abstractos es aquel en que la información se esconde a propósito dentro de una pequeña parte del programa, el programador diseña las estructuras de modo que éstas puedan ser vistas desde dos ángulos diferentes, desde fuera un usuario cliente verá sólo una colección de operaciones que definen el comportamiento de una abstracción; del otro lado el programador verá las variables que son usadas para mantener el estado interno de la misma. La primera vista es llamada interfaz la segunda es llamada implementación [Katrib,1992].

En esta parte debido a que estaremos hablando de como implementar la reutilización a través del modelo orientado a objetos en el lenguaje especifico C++, nos referiremos a un módulo como un programa dividido en funciones y estructuras de datos (clases y funciones miembro), ambos manejados como una unidad y almacenados en un archivo que puede compilarse de manera aislada.

Por otro lado nos referiremos al término complejidad de la superficie de un programa, al que tan fácil o difícil es comprender las relaciones entre los módulos que comprende el programa. La alta complejidad de la superficie de un programa no es una característica deseable en los sistemas y sugiere ocultar esta complejidad valiéndose del principio de ocultación de información y del manejo de datos abstractos, lo que se logra a través de la definición de clases por medio del manejo apropiado de las partes públicas y privadas.

El diseño de interfases adecuadas para las funciones es un tema muy importante que tiene que ver con el paso de parámetros entre funciones (comunicación entre funciones), éste se lleva a cabo de dos formas por medio del paso de argumentos o por medio de variable globales (**extern**), la mayoría de los expertos opina que la mejor manera de comunicar funciones es pasando argumentos y regresando resultados por medio de la sentencia **return**.

Actualmente, el manejo de interfases se lleva a cabo también a través de las secciones pública, protegida y privada en la definición de las clases.

Algunos lenguajes no cuentan con la característica de poder pasar argumentos por dirección, por fortuna C++ sí lo permite, esta forma de pasar parámetros estimula el buen uso de las funciones y de sus valores devueltos, aunque no por eso debe calificarse de trivial ya que como comentaremos más adelante existen ciertos aspectos sutiles cuando se consideran los mecanismos de herencia y polimorfismo.

IV.3 MODULARIDAD.

La mayoría de los proyectos de desarrollo tienen por su tamaño la necesidad de repartir el código en diferentes archivos ya que sería, si no imposible, sí muy difícil e impráctico el manejar todo un proyecto dentro de sólo archivo, por lo que es mejor optar por el manejo de proyectos divididos en módulos compilados, esta división en módulos compilados permite un manejo más apropiado de la información.

Ya hemos mencionado anteriormente los mecanismos con que cuenta el lenguaje C y por lo tanto C++ para comunicar módulos compilados, a propósito debemos mencionar que C y C++ son algunos de los lenguajes que mejor se prestan para trabajar bajo el esquema de módulos compilados ya que a través de directivas del procesador tales como **#define** o **#include** se pueden manejar partes dependientes del proyecto o incluir bibliotecas de la vasta cantidad que incluye el lenguaje o de creación personal.

Existen diferentes maneras de manejar la información de los programas contenida en los archivos de encabezados (**headers**). La primera consiste en guardar toda la información de los encabezados en sólo un archivo de encabezados (lo que es recomendable para proyectos muy pequeños y aislados, este manejo no permite la reutilización de manera satisfactoria), la segunda es asociar cada módulo compilado con un archivo de encabezados (este enfoque es ideal cuando se manejan muchos proyectos en los que hay correspondencia uno a uno entre los parámetros de los módulos y los módulos de funciones C) y la tercera es una mezcla de los dos primeros y es el más recomendable para proyectos grandes.

Por otro lado comentaremos que no existe un acuerdo explícito acerca de que tipo de información debe de estar incluida en los archivos de cabecera, pero en general deben contener alguna de la siguiente:

- Declaraciones de variables y funciones externas.
- Constantes o variables establecidas con una macro **#define**.
- Un número limitado de argumentos que tomarán las macro.

Además se recomienda incluir un archivo de encabezados por cada módulo compilado e incluir en este archivo todas las funciones prototipo de las funciones que están en el módulo.

Existen muchas maneras de dividir un proyecto en módulos, la mayoría de ellas se enfocan a las relaciones que tienen unas funciones con otras, sabemos por otro lado que el tamaño del programa ejecutable no debe ser demasiado extenso, así que será necesario poner cuidado en esta división modular. Por otro lado muchas veces la gente de sistemas se preocupa mucho por la optimización del código e incluso llegan a recurrir a lenguajes de bajo nivel, que

aunque ciertamente pueden tener mayor eficiencia de operación, su costo de desarrollo es mayor, no hablemos del costo de mantenimiento. De tal modo que consideramos que es necesario poner en la balanza por un lado la eficiencia del código y en la otra canasta el costo de desarrollo y de mantenimiento. En general los módulos deben dividirse apoyándose en el criterio de la reutilización.

Otra ventaja que proporciona el uso de archivos de encabezados es que con sólo consultar el archivo de encabezados se puede uno dar idea de como reutilizarlo a la vez que ahorra muchas horas dedicadas a la depuración relacionadas con las llamadas a funciones entre módulos. Por ultimo, los archivos de encabezados además de contener las declaraciones prototipo son útiles para guardar valores de ciertas constantes y variables globales.

IV.4 JERARQUÍA DE CLASES.

IV.4.1 CLASES EN C++.

Como ya lo dijimos anteriormente C++ es un superconjunto de C, lo que nos recuerda que en C utilizábamos archivos de encabezados, donde poníamos definiciones que queríamos pudieran ser incluidas en nuestros archivos fuente por medio de la directriz incluir (**#include**), bueno pues ahora llamaremos a estos archivos, archivos de interfases.

Estos archivos de interfases deben contener las descripciones de las clases que manejemos en el proyecto, la descripción de una clase empieza con la palabra clave **class**, durante la definición de la clase únicamente se permite incluir encabezados de procedimientos y valores de datos. La palabra clave **private** precede a aquellos elementos de la clase que pueden ser accesados sólo por los métodos de la clase misma, la palabra clave **protected** antecede a aquellos elementos de la clase que aparecerán como públicos para todas las clases derivadas de la clase que se está definiendo, mientras que la palabra clave **public** precederá a la verdadera interfaz de la clase, es decir a aquellos elementos que sean accesibles fuera de la clase.

La manera como se crean las clases en C++ debe contener cuatro elementos asociados: primero debe contar con un encabezado de clase, el cual sirve como un especificador de tipo para la clase, en segundo lugar debe tener una colección de miembros de datos definidos dentro del cuerpo de la clase, en tercer lugar una colección de funciones miembro definidas en el cuerpo de la clase y por último un conjunto de niveles de acceso a la clase. Los miembros de la clase pueden ser especificados como privados (**private:**), protegidos (**protected:**) o públicos (**public:**). Estos niveles controlan el acceso de otras clase a los miembros de la clase que se está definiendo.

```
class Clase1 // Encabezado de la clase.
{// Inicio del cuerpo de la clase.
public: // Funciones y estructuras de datos públicas.
protected:  // Funciones y estructuras de datos
protegidas.
private: // Funciones y estructuras de datos privados.
}; // Final del cuerpo de la clase.
```

El control de los niveles de acceso para estructuras de datos y funciones, está relacionado intimamente con la ocultación de información, básicamente ésta se implementa en la sección privada, por convención se debe presentar primero la sección pública y por último la sección privada, con el fin de que los desarrolladores que necesiten consultar la clase tengan esta información disponible de manera apropiada. Se considera una buena práctica el definir las estructuras de datos como privadas y manejarlas a través de funciones que sean públicas.

La sintaxis para definir una clase en C++ es la siguiente:

<classkey> <classname> [:baselist] {<memberlist>}

<classkey> puede ser cualquiera de las palabras clave **class, struct** o **union.**

<classname> puede ser cualquier nombre único dentro de su ámbito.

<baselist> es una lista de clases de las que la presente clase deriva.

<memberlist> declara los miembros de la clase y sus funciones miembro.

dentro de la clase,- los datos son llamados miembros datos y las funciones son llamadas funciones miembro.

IV.4.2 FUNCIONES MIEMBRO.

El archivo de implementación debe contemplar la definición de las funciones miembro, en él se escribe el cuerpo de cada método de la misma manera que se escribe una función convencional en C, con la única diferencia de que el nombre de la función debe ir precedido por el nombre de la clase a la que pertenece y por el operador de resolución de ámbito (::).

Para motivar el uso de los principios de diseño, abstracción y encapsulación C++ da la oportunidad de definir funciones en línea **(inline functions)**. Una función en línea se invoca de la misma manera que cualquier otra función, la diferencia es que el compilador puede decidir si expande la llamada de la función, colocando enseguida el código de ésta con el fin de aligerar la carga de trabajo llevada a cabo durante el envío y regreso de parámetros.

El uso de funciones en línea se antoja adecuado cuando se cuenta con la definición de funciones cuyo cuerpo es pequeño y que también se desea tenerlas encapsuladas, con ello se utilizan adecuadamente los principios de abstracción y encapsulación a la vez que se evitan los costos de la invocación de procedimientos.

Cualquier función que se declare como parte de una clase recibe el nombre de función miembro, éstas son invocadas mediante el envío de mensajes a objetos de la clase, el operador (.) se conoce como el operador "envía mensaje", las funciones miembro que pertenecen a la misma clase pueden enviarse mensajes unas a otras sin necesidad de utilizar al operador punto (.), a continuación propondremos una clase llamada **Racional:**

```
class Racional
//Números racionales con la operación suma.
{
public :
void inicializa(int numerador, int denominador);
void imprimir();
Racional suma(Racional r);

private :
int numerador, denominador;
int mcd(int i, int j);
void comunDenominador(Racional& r);
};
```

Normalmente las declaraciones de las clases se almacenan en archivos de encabezados (.h), mientras que las funciones miembro se almacenan en archivos de programas (.c), los cuales necesitan una directriz (#include) que haga mención al archivo de encabezados que contiene la declaración de la clase.

Un archivo de ejemplo que contenga una función miembro se presenta a continuación :

```
#include "Racional.h"
void Racional : :inicializa(int num, int den)
{
//Inicializa a los miembros numerador y denominador
this->numerador=numerador;
this->denominador=denominador;
}
```

La función inicializa es una función miembro de la clase **Racional**, como lo indica el operador de resolución (::) , this es un apuntador al objeto que recibe el mensaje.

La definición de la función miembro mcd que será utilizada para obtener el máximo común divisor de dos enteros se muestra a continuación :

```
int Racional : : mcd(int i, int j)
{
//Encuentra el máximo común divisor de dos enteros
int divisor;
i=abs(i);
j=abs(j);
for (divisor = ((i<j) ? i : j); divisor > 1 && (i % divisor
!= 0|| j % divisor != 0); divisor --);
return divisor;
}
```

```
void Racional : : comunDenominador(Racional& r)
//Dados dos racionales, los convierte de modo que
tengan un denominador común
{
int multiplicador1,multiplicador2;
int divisor=mcd(denominador,r.denominador);
multiplicador1=denominador/divisor;
multiplicador2=r.denominador/divisor;
numerador*=multiplicador2;
denominador*=multiplicador2;
r.numerador*=multiplicador1;
r.denominador*=multiplicador1;
}
```

La función **comunDenominador** es una función miembro de la clase **Racional**. En la siguiente línea, Racional& r,es una referencia variable de un objeto **r** :

```
void Racional : : comunDenominador(Racional& r)
```

Otros ejemplos de funciones miembro se muestran en la definición de la función miembro suma y de la función miembro imprimir :

```
Racional Racional : :suma(Racional r)
{
Racional temp=*this ;
Racional i ;

temp.comunDenominador(r) ;
i.inicializa(temp.numerador      +      r.numerador,
temp.denominador) ;
return i ;
}

void Racional : :imprimir()
{
cout << numerador << "/" << denominador ;
}
```

Cuando el argumento se pasa como una referencia constante, el valor de éste no puede alterarse. En este caso se utiliza a la variable **temp** para evitar efectos colaterales sobre el receptor del mensaje, la función miembro **suma** regresa el resultado a una nueva instancia de la clase **Racional**.

Notas acerca de la definición de clases.

- Las clases y las estructuras son equivalentes, excepto por que en una clase todos los miembros son privados por defecto, mientras que en una estructura son públicos por defecto.

- La palabra clave **this**, es un apuntador que hace referencia al receptor de un mensaje.
- Los elementos estructurales del receptor pueden referenciarse directamente mediante el uso delnombre del objeto (**objeto.elemento**).
- Las funciones miembro pueden accesar los datos privados de una instancia de la clase.

En otras palabras, cualquier función miembro recibida por una instancia puede accesar los datos públicos, privados o protegidos de esta instancia.

IV.4.3 CONSTRUCTORES Y DESTRUCTORES.

El lenguaje C++ a través de las características que maneja permite la creación de funciones miembro que sean capaces de crear (**constructores**) o destruir (**destructores**) instancias de clases. Estas funciones miembro que cuentan con la característica de tener el mismo nombre que la clase son llamados siempre que es necesario crear un objeto de la clase, ya sea por medio de declaraciones, paso de parámetros o retorno de funciones; los constructores únicamente pueden aparecer dentro de la declaración de la clase, aunque se permiten múltiples declaraciones de ellos y no se permite que regresen valores de ningún tipo.

En el siguiente párrafo se muestra un ejemplo de la clase **Racional** en la que se incluyen tres constructores diferentes.

```
Class Racional //Implementa números racionales con la
operación suma como única operación
definida.
{
public :
Racional() ;//Contructor vacío.
Racional(int n) ;//Constructor con un parámetro.
Racional(int n, int d) ;//Constructor con dos prámetros.
void print() ;
Racional suma(Racional r) ;

 private :
 int numerador ;
 int denominador ;
 int mcd(int i, int j) ;
 void comunDenominador(Racional& r) ;
 } ;
```

El constructor apropiado que será utilizado en un momento dado depende de la manera en que sea llamado, es decir, si la llamada a la función miembro no contiene párametros se usará al primero, si se invoca al constructor con un parámetro se utilizará al segundo y si se emplean dos parámetros durante su invocación se utilizará al tercero.

Por ejemplo el método suma de la clase **Racional** del siguiente ejemplo requerirá que se use la tercera definición del constructor para regresar valores de acuerdo a como se muestra a continuación :

```
Racional Racional : :suma(Racional r)
{
Racional temp=*this ;
temp.comunDenominador(r) ;
return    Racional(temp.numerador    +    r.numerador,
temp.denominador) ;
}
```

Para la creación de una función miembro que nos permita destruir objetos, C++ cuenta con la capacidad de crear funciones miembro llamadas destructores, éstas también tienen el mismo nombre que la clase a la que pertenecen sólo que su nombre debe ir antecedido por el carácter (~). Los destructores son llamados siempre que se destruye una instancia de la clase ya sea por que se abandona el ámbito donde fue creada la instancia, por que se libere la memoria de manera dinámica o se libere memoria temporal. A diferencia de los constructores sólo puede declararse un destructor para cada clase. La declaración del destructor únicamente puede hacerse dentro de la declaración de la clase, los destructores no pueden tener parámetros ni regresar valores. Tanto los constructores como los destructores son opcionales.

IV.4.4 HERENCIA.

Implementación del mecanismo de herencia en C++.

Un paso importante dentro de la programación orientada a objetos involucra adquirir conocimiento para hacer uso efectivo de las clases, organizándolas dentro de una estructura jerárquica basada en el concepto de herencia.

En capítulos anteriores hemos mencionado algunas de las ventajas de la herencia, tales como reutilización de software, compartición de código, consistencia de las interfases, componentes de software, desarrollo de prototipos, polimorfismo y ocultación de la información. De la misma manera hemos mencionado algunos de sus costos asociados, tales como velocidad de ejecución, tamaño de programas, sobrecarga en el paso de mensajes y complejidad de los programas.

Ahora trataremos de ilustrar al programador en C++ cuales son los usos más practicados de la herencia y la manera en que se llevan a cabo en el lenguaje.

El uso más socorrido de la herencia al igual que de la clasificación es el de la especialización y se aplica cuando se utiliza la siguiente regla "La clase *B* es una clase *A*", en cuyo caso *B* es una clase hija de la clase paterna *A*. si la relación que se presenta es "La clase *B* es una parte de la clase *A*", o "La clase *A* contiene elementos de la clase *B*", no sería apropiado crear una estructura de herencia.

Otro uso común de la herencia es para garantizar que las clases mantengan una cierta interfaz común, esto es, que ellas implementen los mismos métodos, en tal caso la clase paterna puede ser una combinación de operaciones implementadas y operaciones que son remitidas a sus clases hijas.

Existen otras razones para utilizar la herencia tales como la construcción y la genaralización [Budd,1991], otra es la extensión a través de la cual se pueden añadir propiedades totalmente

nuevas, la diferencia entre generalización y extensión es que la primera tiene que anular por lo menos un método de su clase paterna y reemplazarlo por otro mientras que en la segunda simplemente se añaden nuevos métodos a aquellos de la clase paterna. Otros autores [Katrib,1994] llaman a la generalización extensión por ampliación y la extensión la llaman extensión por especialización.

También puede emplearse para la creación de subclases y limitar el comportamiento en una clase hija, ésto sucede cuando el programador desea restringir el comportamiento de alguna clase existente, pero no puede modificarla, en tal caso valerse de la herencia con anulación es lo más indicado.

La ultima razón por la que querriamos hacer uso de la herencia es cuando deseamos combinar el comportamiento de dos o más clases paternas para construir una clase hija con las características de estas clases paternas, a este tipo de herencia se le llama herencia múltiple y aunque no es soportado por todos los lenguajes orientados a objetos sí lo es por C++.

Cuando consideramos que de alguna manera las clases son semejantes a los tipos de datos, entonces la jerarquía de clases representa una jerarquía de tipos, dado que una instancia de una subclase **B** es una instancia de una superclase **A**, debemos poder asignar el valor de una instancia de **B** a una variable de la clase **A**. Más aún, cualquier valor de una instancia de una clase debe poderse asignarse a una variable de la misma clase o de alguna clase paterna de la clase a la que pertenece. La situación inversa es conocida como el problema del contenedor.

- **Herencia y clases derivadas.**

Las clases derivadas se usan para crear una jerarquía de clases, la clase otorgante es llamada clase paterna, superclase o clase base; la clase que recibe o hereda las propiedades es llamada clase hija, subclase o clase derivada.

La forma en que se especifica la herencia es semejante a la siguiente:

```
class claseDerivada:[private|public] claseBase
{
// El resto de la declaración de la clase se pone aquí
};
```

Se puede usar herencia pública así como herencia privada precediendo el nombre de la clase base con la palabra clave **public** o **private** respectivamente, en caso de omitirse, el compilador C++ asumirá que se ha utilizado herencia privada.

utilizemos un ejemplo de herencia, dada la siguiente clase :

```
class Empleado
{
public:
Empleado();
Empleado(int num_emp, char nom_emp);
Empleado(const Empleado &e);
~Empleado();

float calcularPago();
void poner_tasa_de_calculo();
void imprimir();

protected:
float sueldo_bruto;

private:
int num_emp;
char *nombre;
float tasa_de_calculo;
};
```

Podemos construir una subclase como la siguiente :

```
class Empleado_jornalero : public Empleado
{
public :
Empleado_jornalero();
Empleado_jornalero(int num_emp, char *nom_emp);
Empleado_jornalero(int num_emp, char *nom_emp, float
tasa);
Empleado_jornalero(const Empleado_jornalero &e);

float calcular_pago();
void calculo_de_horas_trabajadas();
void calculo_de_pago_por_hora();
void imprimir();

private :
float horas_trabajadas;
float pago_por_hora;
};
```

Como podemos apreciar la clase **Empleado_jornalero** hereda públicamente de la clase **Empleado**, a la vez que extiende la definición de un empleado al añadir el número de horas que trabajó y la tasa que se aplica para el cálculo de su salario. También se añadieron las funciones miembro **calculo_de_horas_trabajadas** y **calculo_de_pago_por_hora**.

Empleado_jornalero anula la definición de **Empleado** para el cálculo del pago y proporciona su propio método para este cálculo.

- **Herencia múltiple.**

La diferencia básica entre la herencia normal y la herencia múltiple es que el diagrama de jeraquías de clases que se obtiene utilizando la herencia simple es un árbol, mientras que la herencia múltiple tendrá como representación gráfica un grafo acíclico dirigido.

Considérese la definición de la clase Empleado mostrada con anterioridad y considérese la definición de la clase Estudiante que a continuación se muestra :

```
class Estudiante
{
public :
Estudiante();
Estudiante(int      num_est(int      num_est,      char
*nombre_est);
Estudiante(const Estudiante &e);
~Estudiante();

void añadecurso(Curso &c);
void imprimir();

private :
char *nombre;
int num;
int num_cursos;
Course horario[MAXCURSOS];
};
```

Ahora podemos crear a la clase **EstudianteEmpleado** de la siguiente manera :

```
class EstudianteEmpleado : public Estudiante, public
Empleado
{
public :
EstudianteEmpleado();
EstudianteEmpleado(int num_ee, char *ee_nombre,
float ee_tasa);
EstudianteEmpleado(const EstudianteEmnpleado &ee);
~EstudianteEmpleado();

void imprimir();
};
```

La clase **EstudianteEmpleado** emplea herencia múltiple, cada clase de la lista de las clase base aparece en el encabezado de la declaración de la clase, cada una de estas clases base

puede ser calificada de pública o privada la clase **EstudianteEmpleado** heredará todos los métodos y datos de ambas clases **Estudiante** y **Empleado**.

El constructor nulo declarado en la clase **EstudianteEmpleado** llamará a cada uno de los constructores nulos de las clase **Estudiante** y **Empleado**, el segundo constructor de la clase llamará a cada uno de los constructores definidos en las clases respectivas para completar su definición. De manera semejante el método **imprimir** heredará al método **imprimir** de cada una de sus clases base y definirá al suyo propio.

Cabe mencionarse que los constructores para clases que heredan de múltiples clases base se invocan en el mismo orden en que aparecen las declaraciones en el encabezado de la clase declarada, por otro lado, los destructores se invocan en el orden inverso a la declaración de la clase.

En el caso anterior la clase **EstudianteEmpleado** heredó el nombre y un número de empleado, así como un número de estudiante de sus clase paternas **Estudiante** y **Empleado**, supongamos que en lugar de número de empleado y número de estudiante se hubiese utilizado el número asignado por la oficina de hacienda (rfc), de este modo podríamos haber hecho que la clase **Empleado** heredara este número de la clase **Rfc**, de la misma manera podríamos hacer que el número de estudiante fuese también heredado de la clase **Rfc**. Tendríamos el inconveniente entonces de estar manejando dos números y dos nombres que en realidad son los mismos, para evitar esto sería necesario definir a la clase **Rfc** como una clase virtual, el efecto que tiene definir a una clase como clase virtual es que cuando dos clases que heredan de ella se utilizan como clases base de otra clase derivada los miembros de la clase virtual sólo serán heredados una vez.

IV.4.5 VISIBILIDAD.

- **Visibilidad entre miembros.**

Los miembros de una clase están divididos en tres categorías de accesibilidad :

- **Los miembros de una clase están divididos en tres categorías de accesibilidad:**

- **Los miembros privados pueden accesarse sólo por miembro de la clase y por amigas de la clase.**

- **Los miembros protegidos se comportan como miembros públicos para las clases derivadas y se comportan como miembros privados para el resto del programa.**

- **Los miembros públicos son accesibles desde cualquier lugar del programa.**

- **Por medio de la herencia se puede cambiar la accesibilidad de los datos y métodos en las clases derivadas.**

- **Herencia y visibilidad.**

Cuando hablamos de visibilidad entre miembros de la clase derivada y de la clase base debemos tener en cuenta que la herencia privada cambia la visibilidad de los miembros de la clase base heredados en la clase derivada de la siguiente manera :

- Los miembros privados de la clase base conservan su característica de ser privados y no se les puede accesar mediante métodos de la clase derivada.

- Los miembros protegidos de la clase base se convierten en miembros privados en la clase derivada.

- Los miembros públicos de la clase base se convierten en miembros privados en la clase derivada.

- En el caso de la herencia pública los miembros públicos de la clase base permanecen como miembros públicos en la clase derivada.

- Un objeto perteneciente a una clase derivada es almacenado internamente como un objeto cuyos componentes heredados de la clase base aparecen primero, y son seguidos de los miembros incluidos a través de la clase derivada.

Respecto de la visibilidad entre objetos (objetos que pertenecen a una clase derivada) de la misma clase, o lo que sería la respuesta a la pregunta ¿Que partes puede accesar un objeto **X**, de otro objeto **Y**, cuando ambos pertenecen a la misma clase derivada?, podemos responder que un objeto siempre tiene acceso a sus partes públicas y protegidas sin importar que la herencia sea pública o privada, sin embargo, un objeto nunca puede accesar directamente la parte privada que heredó de la clase base. Para poder accesar o modificar cualquier parte que esté en su parte privada heredada, el objeto en cuestión debe valerse del uso de alguno de los métodos públicos o protegidos que heredó. En C++, los objetos de la misma clase tienen las mismas características de acceso a otros objetos que tienen para sí mismos.

Respecto de la visibilidad entre objetos de la clase base y de la clase derivada, o lo que es lo mismo, ¿Que partes de un objeto de la clase base son visibles por un objeto que pertenece a la clase derivada? diremos que el objeto de la clase derivada puede ver todas las partes públicas y protegidas del objeto de la clase base; sin importar que tipo de herencia se esté utilizando, respecto de la parte privada, ésta de ninguna manera será visible para el objeto de la clase derivada. En efecto la herencia privada niega la relacion "es-un" entre la clase base y la clase derivada.

Otra pregunta importante es ¿Que partes de un objeto pueden accesarse desde fuera de la jerarquía de herencia?, en este caso los objetos que no son instancias de alguna clase que pertenezca a la jerarquía de clases, sólo podrán accesar las partes públicas de la clase derivada.

Cuando se hereda públicamente la clase derivada puede ver todas las partes de la clase base excepto las partes declaradas como privadas, si se declara un objeto de la clase derivada en la función principal (**main**) este objeto no podrá accesar a los miembros protegidos ni a los miembros privados de la clase base de la que la clase derivada hereda, pero sí a los miembros

definidos en la parte pública. El objeto de la clase derivada definido en la función principal (**main**) tampoco podrá accesar a las parte privadas ni protegidas de la clase derivada, únicamente podrá accesar a las partes públicas.

Cuando se hereda de manera privada, una función miembro de la clase derivada puede ver las partes públicas y protegidas de la clase base pero no la parte privada de las clase base. Si se declara un objeto perteneciente a la clase derivada en la función principal este objeto no puede ver a ninguna de las partes de la clase base y sólo podrá ver a las parte públicas de la clase derivada.

- **Comentarios sobre funciones heredadas.**

Las funciones miembro heredadas de una clase base pública, pueden ser enviadas a instancias de la clase derivada.

Una instancia de una clase derivada puede pasarse como parámetro de una función que esté declarada en la parte publica de la clase base.

Una referencia de una clase derivada puede pasarse como parámetro de una función que esté declarada en la parte pública de la clase base.

Un apuntador a una clase derivada puede pasarse como un parámetro de una clase base definida como pública.

IV.4.6 PRINCIPIOS O PAUTAS PARA EL USO APROPIADO DE LA HERENCIA.

Para que el modelo orientado a objetos cumpla su promesa de incrementar la reutilización, la facilidad de comprensión y la facilidad de mantenimiento, es necesario que se trabaje con estructuras de herencia de alta calidad. No basta con construir numerosas clases que heredan características de superclases existentes, se deben crear estructuras de herencia o jerarquías de herencia que permitan el mejor uso de ésta. Desgraciadamente este mecanismo puede ser mal empleado y ahondar los problemas que trata de resolver.

Afortunadamente se han publicado ya muchos principios, pautas y guías [Firesmith,1995] para ayudar al uso apropiado de la herencia, los cuales presentaremos aquí, con el objeto de obtener los mayores beneficios de su uso.

P1:Nunca crear una nueva clase si se puede reutilizar otra ya existente. Este principio promueve la reutilización y minimiza la explosión de clases.

P2:Asignar a una sola persona como responsable de la creación de cada estructura de herencia importante para el sistema. Las buenas estructuras de herencia tienen en conjunto arquitecturas apropiadas que difícilmente surgen sin la supervisión de un buen arquitecto que desarrolle una arquitectura uniforme, consistente y que asegure elasticidad a largo plazo.

P3:Estar preparado para la evolución y cambio de las estructuras de herencia antes de que se estabilicen, pero trate de estabilizarlas tan pronto como sea posible. Ponga estructuras de herencia estables bajo la configuración del control. Las buenas estructuras de herencia

evolucionan a medida que hay mayor entendimiento de los problemas y a medida que se suscitan cambios en los requerimientos. La configuración del control protege a los usuarios de la arquitectura de modificaciones indocumentadas o arbitrarias.

P4:Optimice las estructuras de herencia mediante el traslado de características comunes (mensajes, operaciones, excepciones, atributos), que están contenidos en subclases hacia las superclases comunes. Ponga las características que sean independientes de la aplicación en la superclase y ponga las características específicas de cada aplicación en las subclases. Concéntrese en mover mensajes y operaciones visibles en lugar de excepciones y atributos

P5:Procure crear nuevas estructuras de herencia al identificar clases concretas y generalice sus características importantes al incluirlas en clases superiores.

P6:Extienda las estructuras de herencia de abajo hacia arriba al extender las superclases vía subclases especializadas.

P7:Las clases de dato abstractas sólo deben contener características abstractas comunes. Este principio motiva la precisión, mantenibilidad, capacidad de entendimiento y especialización de la herencia.

P8:Las superclases deben estandarizar a las subclases. Esto se refiere a que las superclases deben estandarizar los nombres y significados de las características comunes en sus clases hijas. Las subclases o deben cambiar el nombre o el significado de las características heredadas. Mediante la implementación del principio de ingeniería de uniformidad, esa pauta aumenta la facilidad de mantenimiento y de comprensión y de especialización la herencia. Además de que permite la substitución polimórfica de los objetos de las subclases por los objetos de las superclases.

P9:Una superclase no debe restringir el desarrollo de sus futuras subclases a menos que sea necesario asegurarse de que estas no violaran sus características de abstracción. En C++ por ejemplo las funciones miembro pueden declararse como virtuales para permitir el polimorfismo y en enlace dinámico. De manera similar los desarrolladores en C++ no deben usar miembros privados ni herencia privada sin una justificación importante. (Por ejemplo un miembro privado es usado solamente para aceptar miembros públicos protegidos en el nivel de abstracción actual). Este principio impulsa la capacidad de extensión de las clases y acepta la especialización por medio de la herencia.

P10:Si es apropiado use clases diferidas para controlar la futura clasificación de subclases, por ejemplo use funciones virtuales puras en C++ para forzar a las subclases a proporcionar una operación necesaria para todas las subclases de abstracción. Este principio ayuda a facilitar al diseñador el que las subclases futuras no violarán las abstracciones definidas por sus superclases.

P11:La especialización de la herencia implementa la relación "Una clase de", en la cual la subclase es una especialización de sus superclases. La interfase de la herencia es una herencia en la cual el protocolo de la subclase es un superconjunto de la unión de protocolos de sus superclases, con lo cual permite la substitución polimórfica de objetos de la subclase por objetos de sus superclase aún si la subclase no es una especialización de

sus superclases. Herencia de la implementación es herencia usada meramente con el propósito de compartir código, sin especialización ni consistencia de protocolos. Debe maximizarse el uso de la herencia de la especialización y evitar la herencia de la implementación. Este principio promueve la exactitud, la capacidad de extensión, la capacidad de manutención y la capacidad de comprensión, además de ayudar a la correcta substitución polimórfica.

P12:Use la herencia como especialización para capturar la relación "Una clase de". Cada subclase debe ser una especialización de sus superclases y cada superclase debe ser una generalización de sus subclases.

P13:Cada objeto de una subclase debe ser indirectamente un objeto de sus superclases (Capacidad de substitución).

P14:Use la herencia como especialización para permitir la substitución polimórfica de subclases por superclases.

P15:El protocolo de una subclase debe ser un superconjunto de la unión de los protocolos de sus padres. La subclase deber ser una extensión de sus superclases y debe por lo tanto ofrecer servicios adicionales, las subclases deben añadir mensajes pero no anularlos, pueden añadir parámetros pero no deben borrarlos.

P16:Los principios de una subclase deben ser extensiones consistentes de los principios de sus superclases. Las subclases nunca deben relajar las condiciones invariantes de las superclases, nunca deben restringir más las precondiciones de operación o relajar las post condiciones de las operaciones. Este principio promueve la especialización de la herencia y permite el polimorfismo.

P17:Los modelos de estados de las subclases deben ser extensiones consistentes de los modelos de estados de sus superclases. Las subclases no deben borrar estados heredados de sus superclase, aunque sí pueden añadir estados y subestados a aquellas heredado de sus superclases. Las subclases no deben borrar transiciones heredadas de sus superclases, tampoco deben transformar los estados heredados de sus padres en estados finales. Las subclases sí pueden añadir transiciones. Esta regla permite la especialización a través de la herencia y la substitución polimorfica.

P18:Ser cuidadoso cuando cambie el rango de tipos enumerativos en la operación de parámetros. Es decir, no se deben remover valores enumerativos que sean parámetros de entrada ni se deben añadir valores enumerativos como parámetros de salida. La falta de obediencia a esta regla podría dificultar el uso a los clientes.

P19:Las subclases deben ser diferentes de sus superclases. Las subclases que no añaden cambios o que no añaden diferentes características incrementan los costos de desarrollo inecesariamente.

P20:Las subclases no deben borrar características. Este principio no va de acuerdo con el concepto de herencia con anulación, lo que es explicable tomando en cuenta que existen diferentes puntos de vista acerca de como manejar la herencia.

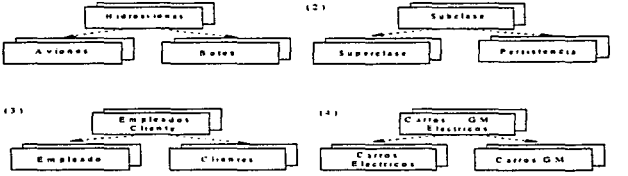
P21:Las clases abstractas probablemente no deben tener ancestros. La especialización a través de la herencia implica que las subclases son "un tipo" de sus superclases, por ello deben ser capaces de hacer todo lo que la superclases puede hacer, digamos crear instancias (objetos), borrarlas etc.

P22:Use la herencia cuando las subclases resultantes tengan diferentes características y comportamientos, de otra manera use instancias de clases poderosas [1], como atributos. La herencia debe ser utilizada para producir diferentes clases de cosas con diferentes propiedades o comportamientos. Use atributos para describir diferentes instancias de la misma clase de cosa. Por ejemplo, si los hombres y la mujeres en una aplicación tienen diferentes comportamientos o características, entonces construya las subclases hombres y mujeres de la clases personas. De otro modo sólo utilice un atributo para almacenar una instancia de la clase poderosa género.

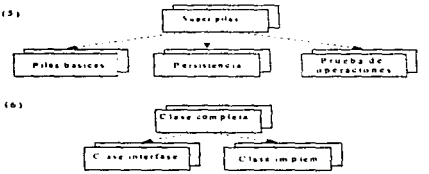
P23:La subclases que anulen más de 3 operaciones heredadas de una superclases no deben ser consideradas especializaciones de superclases .

P24: Use la herencia múltiple con cuidado y en los siguientes casos :

- Para herencia especializada
- Para añadir una capacidad común usando un mixin[2] simple.
- Para el desarrollo de subclases cuyo protocolo se divide en roles separados.
- Para subclases que sean combinaciones de superclases bidimensionales.
- Para componer clases grandes fuera de una clase primaria que proporcione las características fundamentales de las clases abstractas y de las clases mixin que proporcionan capacidades adicionales de abstracción y para separar aspectos de interfases de aspectos de implementación. (Especialmente con repositorios para reutilización)



[1] Una clase poderosa es una clase cuyas instancias son (o representan) subclases de otra clase.
[2] Un mixin es una clase abstracta diseñada para proporcionar capacidades comunes a a otras clases a través de la herencia múltiple.

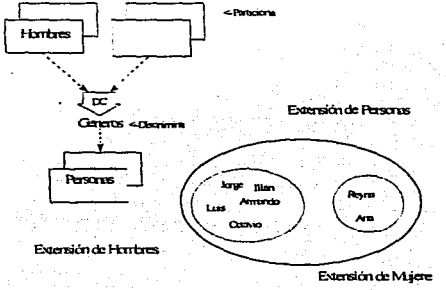


Algunos problemas que acarrea la herencia múltiple son : la perdida de entendimiento, el que los nombres de las operaciones heredadas se encuentre más de una vez en la lista de operaciones con que cuentan las clases y la duplicaciones de operaciones.

P25: una subclases nunca debe heredar de más de dos padres. Esta regla incrementa la reutilización.

P26:Un mixin debe combinarse a través de una generalización simple para producir una especialización con una o más capacidades adicionales. Este principio ayuda a entender mejor la jerarquía de clases y proporciona herencia especializada, mientras que se vale de la herencia de implementación para el mixin.

P27:Las extensiones de subclases en una dimensión deben particionar la extensión de la superclase. i.e. las extensiones de las subclases son disjuntas y componen a la clase paterna.



P28: Una subclases que ha sido definida vía herencia múltiple debe heredar cuando más de una clase en una dimensión.

P29:Evite estructuras de herencia demasiado profundas o demasiado superficiales.

P30:Desmonte y pruebe las clases dentro de las estructuras de herencia de otro modo no podrán ser reutilizadas.

P31:Use la herencia para heredar software de prueba , así como para software liberable. Desarrolle una estructura de herencia análogo de manejadores de prueba que incluya datos de prueba, operaciones de prueba dentro de cada clase de la estructura de herencia o desarrolle una estructura de herencia análoga de mixins que contengan datos y operaciones. Este principio facilita la prueba de unidades de una manera automática y minimiza el esfuerzo de pruebas de regresión.

P32:Conserve a las clases que estén dentro de una estructura de herencia con sus características de portabilidad e interoperabilidad.

P33:Use clases parametrizadas (genéricas) para simplificar la jerarquía de herencia cuando existan muchas variaciones simples. Este principio incrementa la facilidad de entendimiento y la capacidad de que las clases se extiendan mientras que disminuye el tiempo dedicado a la creación y mantenimiento al reducir el numero de clases requeridas.

P34:Las superclases no deben tener visibilidad de sus subclases.

P35:En las operaciones evite el uso de declaraciones CASE, SWITCH o IF anidados. Las declaraciones CASE muestran inapropiados usos de la herencia, especialmente cuando están basados en tipos de datos, las operaciones de las clases bien diseñadas rara vez necesitan de declaraciones CASE.

P36:Primero aprenda la estructura completa de la herencia, aprenda sobre el núcleo de las clases que modelan las abstracciones clave y que son utilizadas para construir clases, después aprenda en base a la necesidad que tenga de conocerlas y de la disponibilidad de tiempo. Las bibliotecas de clases contienen demasiadas clases como para aprenderlas en un periodo corto.

P37:Asegúrese de que todos los depuradores entiendan la herencia y el lenguaje de programación

IV.5. POLIMORFISMO.

Cuando se habla de un objeto polimorfo en un lenguaje de programación, se refiere a una entidad (variable o argumento de función) a la que se le permite almacenar valores de diferentes tipos durante la ejecución, así, se llaman funciones polimórficas a aquellas que aceptan argumentos polimorfos, tales funciones son fáciles de escribir en lenguajes con asignación dinámica de tipos (v.g. Smalltalk), sin embargo, el polimorfismo también sucede en lenguajes con asignación estática de tipos, el ejemplo más común del polimorfismo se da en la sobrecarga de operadores.

Una manera de ver el polimorfismo es en términos de abstracciones de alto y bajo nivel. Una abstracción de bajo nivel es una operación básica, es decir, alguna que este construida sobre sólo unos pocos mecanismos, mientras que una operación de alto nivel es un plan mas general que proporciona un enfoque general a seguirse pero no sus detalles.

En general los algoritmos se describen en un nivel alto de abstracción y se implementan en un lenguaje en particular sobre estructuras de bajo nivel. De alguna manera las abstracciones de bajo nivel son herramientas que pueden acarrearse de un proyecto a otro, mientras que en una abstracción de alto nivel en un lenguaje de programación debe referirse a tipos específicos de estructuras de datos. El potencial del polimorfismo es que permite escribir por única vez algoritmos de alto nivel que puedan usarse y reusarse repetidamente con diferentes abstracciones de bajo nivel.

En los lenguajes orientados a objetos, el polimorfismo ocurre como resultado natural de la relación "**como-un**" y de los mecanismos del paso de mensajes y la herencia, los cuales pueden combinarse de distintas maneras para producir diferentes técnicas de codificación, compartición de código y reutilización.

El polimorfismo puro ocurre cuando una función puede aplicarse a argumentos de tipo diferente es decir, existe una única función y un conjunto de interpretaciones diferentes.

Con la excepción de la sobrecarga de operadores, el polimorfismo en los lenguajes orientados a objetos es posible sólo debido a la existencia de objetos polimorfos (Que pueden almacenar valores diferentes).

Los operadores sobrecargados son aquellos que han sido redefinidos dentro de alguna clase, usando la palabra clave **operator** y seguidos por un símbolo de operador. Todos los operadores pueden cambiarse excepto los siguientes { .,*,::,?: }, y los símbolos del preprocesador **#** y **##**.

Con excepción del operador función de asignación {"=()"}, todos los operadores sobrecargados se heredan a las clases derivadas de la clase donde se define la sobrecarga.

En un lenguaje enlazado dinámicamente (Smalltalk,Objetive-C), todos los objetos son potencialmente polimorfos ya que pueden almacenar valores de diferentes tipos.

En los lenguajes con asignación estática de tipos tales como C++ y Object Pascal la situación es más compleja. El polimorfismo ocurre a través de la diferencia entre la clase (estática) declarada de una variable y la clase (dinámica) del valor contenido en la variable. Recordemos que en este caso una variable puede almacenar valores del mismo tipo de la clase declarada de la variable o de cualquier subclase de la clase declarada. Cuando se usa declaración estática de variables los objetos polimorfos se dan únicamente a través del uso de apuntadores y referencias (direcciones), cuando no se usan apuntadores: la clase dinámica de un valor esta limitada a ser la misma que la clase estática de la variable. Cuando se usan apuntadores y referencias el valor retiene su tipo dinámico.

El enlace de métodos en C++ puede llevarse a cabo de manera estática o dinámica. El enlace estático de métodos con mensajes se basa en la característica de la variable, el enlace dinámico se basa en el tipo de valor y no en el tipo de la declaración.

En C++ predominan dos objetivos de diseño, el primero es el manejo de espacio y el segundo es el de la eficiencia en tiempo, por lo tanto, C++ no acarrea costos adicionales por el uso de elementos que no están incluidos en el programa. Es decir, si se escribe un programa en C++ que no utilice ninguna de las características que hacen C++ diferente de C, el programa ejecutable no tendrá incluido ningún manejador de características específicas de C++. De lo que se desprende que la mayoría de las características de C++ son estáticas más que dinámicas.

C++ no contiene ningún método para determinar cual es la clase dinámica de un objeto, es decir, los valores no tienen auto conocimiento de su tipo asociado. Esta restricción haría que la solución al problema del contenedor fuera muy compleja, pero debido a la gran flexibilidad que proporciona para sobrecargar y anular métodos la solución se facilita. C++ proporciona enlace dinámico y estático entre métodos y mensajes, además de reglas complejas para determinar que forma se está usando. El enlace dinámico depende del uso de métodos virtuales y de que el receptor sea usado como un apuntador o como una referencia.

Aún cuando se use el enlace dinámico entre métodos y mensajes la legalidad de cualquier expresión de paso de mensajes es verificada basándose en el tipo estático del receptor.

Durante la búsqueda de un método a invocar en respuesta a un mensaje, generalmente se comienza buscando el método en la clase, con que se esta trabajando, posteriormente en la clase paterna y subsecuentemente se va escalando en la jerarquía de clases hasta encontrar un método apropiado o emitir un mensaje de error.

IV.5.1 POLIMORFISMO COMO SOBRECARGA DE FUNCIONES.

Podemos decir que un nombre de función está sobrecargado si existen dos o más cuerpos diferentes asociados con el mismo nombre, la sobrecarga es una parte de la anulación por lo tanto, puede existir sobrecarga sin anulación. Durante la sobrecarga el nombre de la función es el que es polimorfo.

Existen soluciones a problemas de polimorfismo que pueden resolverse únicamente mediante sobrecarga y otros que pueden valerse de la sobrecarga y la coerción o que involucren únicamente coerción.

La mayoría de los lenguajes de programación orientados a objetos permiten la sobrecarga de funciones al utilizar diferentes cuerpos y un mismo nombre. En C++ debido a su característica de asignación estática de tipos es necesario proporcionar todas las declaraciones para todos los cuerpos definidos, al revisarse los tipos de la lista de argumentos manejados, el compilador C++ decide a que función va a hacer mención con lo que la sobrecarga es muy semejante a la anulación.

Existe otro tipo de sobrecarga en la cual los procedimientos (o funciones o métodos), dentro de un mismo contexto pueden compartir el nombre, la diferencia se hace al verificar el número y tipo de argumentos que se reciben como parámetros; este último se llama sobrecarga paramétrica, mientras que el primero recibe el nombre de sobrecarga por rúbrica (**signature**). C++ permite que cualquier función método o procedimiento sea sobrecargado paramétricamente.

Es importante tomar en cuenta las siguientes consideraciones cuando se habla de sobrecarga : las funciones sobrecargadas deben ser distinguibles unas de otras por medio del número y tipo de argumentos que manejan, el valor que regresarán estas funciones no puede utilizarse como distintivo entre las misma y cuando el compilador necesita hacer un reconocimiento por rúbrica toma en cuenta el número, tipo y orden de los argumentos declarados.

El compilador de C++ al momento de llamar a una función miembro sobrecargada lleva a cabo las siguientes operaciones :

- Cada tipo de argumento (del parámetro real) utilizado en la llamada de la función sobrecargada se compara con el tipo de argumento (del parámetro formal) que le corresponde en orden en la definición de la función.
- El compilador escogerá a aquella función para la cual cada resolución de argumentos sea la misma o se acople de mejor manera.
- El algoritmo que utiliza el compilador para establecer la relación entre parámetros reales y parámetros formales distingue entre constantes y apuntadores que no son constantes.
- La comparación que se lleva a cabo comprende como primer paso la búsqueda de un apareamiento exacto entre parámetros.
- Después la búsqueda trata encontrar un apareamiento valiéndose de promociones.
- Posteriormente se trata de establecer un apareamiento por medio del uso de conversiones.
- Y por último se trata de establecer la relación entre parámetros tomando en cuenta las conversiones de datos que el mismo usuario haya definido.

La conversión de tipos (conversiones y promociones) suceden cuando se pasa argumentos a las funciones. La conversión involucra un cambio de la representación interna del valor en la máquina. En particular cuando dos tipos tiene diferentes tamaños, si el cambio es de un tipo más pequeño a un tipo de mayor tamaño se le llama promoción, si el cambio es entre valores que ocupan el mismo espacio o de un tipo de tamaño más grande a uno de menor tamaño se le llama conversión.

Es posible que se pierda información durante una conversión, pero no durante una promoción.

Otra manera de implementar el polimorfismo es mediante la declaración de funciones miembro constantes, es decir, puede existir una versión constante de una función miembro y una versión no constante, la función que se seleccione en un momento dado dependerá de si el objeto que recibe el mensaje está declarado como constante o no.

Las funciones miembro constantes (que actúan como mensajes constantes) sólo pueden ser enviadas a instancias definidas como constantes.

IV.5.2. FUNCIONES VIRTUALES Y MÉTODOS DIFERIDOS.

A través de la definición de funciones virtuales se puede llevar a cabo el polimorfismo en el lenguaje C++, lo primero que debe hacerse es declarar apuntadores a objetos de una clase base, lo que nos permitirá referenciar tanto a elementos de la clase base como a elementos de sus clases derivadas, después es necesario que definamos a la función miembro como una función virtual antecediendo la palabra clave virtual al tipo de la función. Con lo anterior habremos implementado el enlace dinámico de mensajes, la implementación del enlace dinámico en C++ es completamente eficiente, ya hemos mencionado que durante el enlace dinámico, la función miembro declarada como virtual será invocada dependiendo del tipo del receptor del mensaje.

Dentro de una clase abstracta se pueden declarar funciones virtuales, las que son comúnmente llamadas funciones virtuales puras, aquí debe tomarse en cuenta que cada subclase de la clase abstracta debe redefinir a las funciones virtuales puras, de la misma manera debe tomarse en cuenta que para destruir a un objeto que se maneja a través de apuntadores debemos crear destructores virtuales en la clase.

Un método diferido o método genérico (llamado en C++ método virtual puro), puede concebirse como una generalización de la anulación, en ambos casos el comportamiento descrito en una clase paterna es modificado por el definido en una clase hija, sin embargo, en un método diferido el comportamiento en la superclase es esencialmente nulo y toda la actividad es definida como parte de un método anulador.

Una ventaja de los métodos diferidos es que su uso permite a los programadores pensar en una actividad como una actividad asociada con una abstracción a un nivel más alto.

Un segundo uso más práctico para los métodos diferidos, en los lenguajes con asignación estática de tipos (como C++), es el siguiente : a un programador únicamente se le permite reenviar un mensaje a un objeto si el compilador puede determinar que el objeto esta ahí (es decir, el método correspondiente que se activara por medio del selector de mensajes).
En C++ un método diferido (llamado método virtual puro) debe declararse explícitamente usando la palabra clave "**virtual**", además no debe ponerse en la definición de la clase el cuerpo del método virtual puro, sino que debe asignársele el valor 0.

No es posible crear objetos pertenecientes a clases que contengan métodos virtuales si estos no han sido anulados. La definición del método virtual puro debe manejarse en una subclase inmediata.

IV.5.3 POLIMORFISMO COMO SOBRECARGA DE OPERADORES.

Naturalmente podemos definir dentro de nuestro grupo de funciones miembro de una clase específica a los métodos que nos permitan realizar operaciones con los objetos de la clase, pero tal vez nos gustaría que en lugar de utilizar nombres creados a nuestro libre pensar pudiésemos usar símbolos ya existentes en el lenguaje. En este caso lo que debemos hacer es sobrecargar operadores.

Sobrecargar un operador no es más que redefinir un operador que cuenta con una función auto construida, ésto se logra definiendo una función cuyo nombre consista de la palabra clave **operator**, seguida del símbolo del operador que se desea sobrecargar.

Por ejemplo, si tuviésemos una clase llamada **Cadena**, podríamos escribir la siguiente función para concatenar dos cadenas :

Cadena : :operator+(Cadena x) ;

con lo que podríamos concatenar dos cadenas usando la siguiente instrucción :

```
Cadena a,b,c ;
a=b+c ;
```

Cuando se sobrecargan operadores es necesario tener en cuenta que sólo pueden sobrecargarse aquellos operadores que están predefinidos, no puede alterarse la precedencia de un operador, no puede cambiarse la naturaleza ya sea binaria o unitaria del operador y que la definición de la sobrecarga del operador debe contener al menos un argumento del tipo de la clase especificada.

El número de operadores que se pueden cargar en C++ es aproximadamente de 39, entre los que se incluyen a partir de la versión 3.0 a los operadores (- - y + +).

El lenguaje C++ proporciona otra característica importante que debemos tener en cuenta, la definición de funciones y de clases amigas, las cuales tienen acceso a los miembros privados de la clase, las instancias de una clase en particular no pueden recibir como mensajes a funciones que hayan sido declaradas como amigas, por lo que no tienen acceso al operador **this**.

Las funciones amigas son necesarias cuando el operador izquierdo no es una instancia de la clase que se ha definido. En general no se recomienda el uso de funciones amigas si no hay necesidad de romper el encapsulamiento.

El aprovechamiento de las caraterísticas del lenguaje C++ en particular y en general de los lenguajes orientados a objetos, hace posible que el software desarrollado sea más robusto, lo que se logra gracias a características tales como el encapsulamiento y el manejo de excepciones. De manera semejante las características de poder derivar clases de otras ya existentes, de redefinir operadores y funciones, de la genericidad, de la herencia y del polimorfismo permite incrementar la reutilización y la extensibilidad.

Precisando, podemos decir que el uso de recursos tales como el manejo de elementos públicos y privados dentro de las clases, de constructores y de destructores nos garantiza la integridad de los objetos a la vez que aumenta la confiabilidad y facilita la reusabilidad.

IV.6 EJEMPLOS DE REDEFINICIÓN DE OPERADORES, HERENCIA Y POLIMORFISMO RELACIONES VIRTUALES, ENLACE DINÁMICO Y ENCAPSULAMIENTO.

IV.6.1 REDEFINCIÓN DE OPERADORES.

Si un lenguaje como C++ incluye la notación y la instrumentación tradicional para denotar las operaciones con los tipos simples predefinidos, entonces sería deseable que permita utilizar la misma notación, para los nuevos tipos que se definan. Para lograr esto C++ permite redefinir los operadores de C. Con este recurso podemos utilizar los mismos símbolos y la misma sintaxis de los operadores predefinidos de C pero con la semántica conforme a las nuevas clases que se definan.

El ejemplo que sigue presenta una clase que define la suma de números racionales. La intención es definir para los números racionales una operación similar a la que define C para el tipo aritmético predefinido suma (+).

```
Class RACIONAL
{ private :
int num,den ;
  int abs(int n){if (n<0) return -n ; else return n ;};
  void simplifica(void); // para simplificar el numero racional
  public:
RACIONAL (int n = 0,int d = 1) {num=n ; den=d ;} ;
RACIONAL operator = (RACIONAL &r) ;
RACIONAL operator += (RACIONAL &r) ;
RACIONAL operator + (RACIONAL &r) ;
int operator == (RACIONAL &r) ;
} ;
```

Esta clase va a permitir aplicar a operandos de tipo RACIONAL los operadores '=' ,'+=' ,'+' y '==' con la misma sintaxis que C aplica a los aritméticos. Si tenemos la declaración :

```
RACIONAL p,q ;
```

entonces podemos escribir

```
p=q  p+=q  p+q  p==q
```

lo que será interpretado como enviar los mensajes siguientes :

```
p.operator=(q) p.operator+=(q) p.operator+(q)p.operator==(q)
```

Notesé que en la descripción anterior se estan utilizando dos funciones privadas ***abs*** y ***simplifica*** para uso interno de las funciones miembros de la clase.

La función ***simplifica*** es privada. Esta es utilizada internamente para transformar al número racional y representarlo en forma simplificada cuando sea necesario. Su definición es :

```
void RACIONAL : : simplifica(void)
{if (den<0) {num =-num ; den =-den ;};
 for (int i= abs(num)< den ; abs(num) ; den ; (i>=2) &&
(den>1) ; i--)
 if ((num%i)==0 &&(den%i)==0{num/=i; den/=i ;} ;
}
```

Si en la notación de objetos esta asignación se lee como p.operator=(q) esto significa que la función operator = debe retornar el propio objeto actual después de haber cambiado los valores de sus componentes. Ésto se hará utilizando el identificador *this*. La definición del operador = queda entonces en la siguiente forma.

```
RACIONAL RACIONAL : : operator = (RACIONAL &r)
{ num = r.num ; den =r.den ;
  return (*this) ;
}
```

Las definiciones de las operaciones + y += hacen también uso de este recurso.

```
RACIONAL RACIONAL : : operator + (RACIONAL &r)
{
  return RACIONAL (num*r.den + r.num *den , den*r.den) ;
};

RACIONAL RACIONAL : : operator += (RACIONAL &r)
{ num=num*r.den + r.num *den ;
  den* = r.den ;
  return (*this) ;
};
```

IV.6.2 HERENCIA Y POLIMORFISMO.

Cuando en la definición de una clase ***B*** en la forma siguiente :

```
Class B
{ //...
  A x ;
  //...
} ;
```

Se usa una componente de otra clase ***A***, o tiene una función que utiliza a la clase ***A***, se dice que la clase ***B*** es *usuario* o *cliente* de la clase ***A***. el encapsulamiento no permite que ***B*** haga uso de las partes privadas de un objeto de la clase ***A***. Esto es correcto desde el punto de vista de esta relación de *clientelismo*, pues nos garantiza que ***B*** pueda usar la clase ***A*** pero respetando una funcionalidad ya existente, aumentando con ello la confiabilidad. Sin embargo, no es suficiente desde el punto de vista de la extensibilidad. Es decir, también es conveniente *adaptar* o *modificar*, a nuevos requerimientos, clases ya existentes, sin tener que violar el encapsulamiento, ésto se logra a través del concepto de herencia.

IV.6.3 EXTENSIÓN POR HERENCIA

En el siguiente ejemplo veremos una clase a la que llamaremos ***CIRCULO*** que representa a la figura geométrica círculo. Un círculo estará representado por dos componentes privadas *X* y *Y* que representan a las coordenadas del centro círculo, otra componente ***Radio*** que representa el radio del círculo, y otra componente ***Visible*** que indica si el círculo está siendo visualizado por pantalla o no. Un círculo tendrá las funcionalidades Muestra (que lo mostrara en la pantalla) y Oculta (que lo ocultará), así como la función traslada para trasladarlo de lugar en la pantalla. Así mismo tendrá las funciones Acceso Visible, Radio, X y Y que darán información sobre su estado.

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

class CIRCULO
{ private
   int pri_x,pri_y ;
   int pri_radio ;
   int pri_visible ;
  public
   CIRCULO (int InicX, int InicY, int InicRadio)
     {pri_x=InicX ; pri_y=InicY ; pri_radio=InicRadio ;}
   int X(void) const {return pri_x ;}
   int Y(void) const {return pri_y ;}
   int Radio(void) const {return pri_radio ;}
   int Visible(void) const {return pri_visible ;}
   void Muestra(void) ;
   void Oculta(void) ;
   void Traslada(int NuevoX,int Nuevo Y);
};

void CIRCULO : : Muestra(void)
{ pri_visible=1 ;
   Circle(pri_x,pri_y,pri_radio) ;
} ;

void CIRCULO : : Oculta(void)
{ if (pri_visible)
    {pri_visible=0;
     unsigned int Color_temp;
     Color_temp = getcolor();
     Setcolor(getbkcolor())};
     Circle(pri_x, pri_y,pri_radio); // pinta el circulo en el color del fondo
     Setcolor(Color_temp);          // reponer el color activo
    };
};
```

```
void CIRCULO : : Traslada (int NuevoX,int NuevoY)
{ if (pri_visible)
   { Oculta() ;              // Hacerlo invisible
     pri_x = NuevoX;  // Cambia las coordenadas a la nueva posición
     pri_y = NuevoY;
     Muestra();              // Muestra el punto en las nuevas coordenadas.
   }
  else
   { // estaba invisible, solamente se cambiarán las coordenadas
     pri_x = NuevoX ;
     pri_y = NuevoY ;
   }
}
```

Una aplicación puede ser :

```
CIRCULO Uncirculo(100,100,30) ;
//...
Uncirculo.Muestra ; // Muestra el círculo
//...
UnCirculo.Traslada(UnCirculo.X()+50,UnCirculo.Y()) ;   //Desplaza   el
círculo en el eje X.
```

Ahora deseamos ampliar las funcionalidades de un círculo y añadir dos nuevas funciones para aumentar y disminuir el círculo de tamaño, sin tener que hacer cambios en el texto original de la clase ***CIRCULO***, definiremos entonces una nueva clase ***CIRCULO_INFLABLE*** que hereda de ***CIRCULO*** todas sus propiedades y además añade las funciones *Inflar* y *Desinflar*.

```
Class CIRCULO_INFLABLE : public CIRCULO
 { public :
 CIRCULO_INFLABLE(int IncX, int IncY, int IncRadio):CIRCULO( IncX, IncY,IncRadio ){}
    void Inflar(int  Encuanto) ;
    void Desinflar(int  Encuanto) ;
 };

void CIRCULO_INFLABLE : :Inflar(int  EnCuanto)
 {if (Visible())  // Usa la función Visible, Hereda de CIRCULO a través del objeto en curso.
   {Oculta() ;   //Usa la función Oculta , heredada de CIRCULO
     pri_radio+=EnCuanto ;
     if (pri_radio<0) pri_radio=0 ; // pri_radio negativo es tomado como 0
Muestra() ; // Usa la función Muestra , heredada de CIRCULO
   };
  else
   {pri_radio +=Encuenato ;
    if (pri_radio<0) pri_radio=0 ; // pri_radio negativo es tomado como 0
   };
 };

void CIRCULO_INFLABLE : :Desinflar(int  EnCuanto)
{Inflar(-Encuanto) ;}
```

Que la nueva clase **CIRCULO_INFLABLE** hereda de **CIRCULO** se expresa en la linea

Class **CIRCULO_INFLABLE** : public **CIRCULO**

La especificación **public** indica que los clientes de la clase **CIRCULO_INFLABLE** tienen acceso a los mismos recursos públicos a los que tenían acceso los clientes de la clase **CIRCULO.**

Un cliente que tenga ahora

CIRCULO_INFLABLE UnCirculoInflable(100,100,50)

puede hacer

```
UnCirculoInflable.Muestra() ;
UnCirculoInflable.Oculta()
UnCirculoInflable.Traslada(150,200) ;
UnCirculoInflable.X() ;
UnCirculoInflable.Y() ;
UnCirculoInflable.Visible() ;
```

y también

```
UnCirculoInflable.Inflar(UnCirculoInflable.radio()) // Infla el círculo al doble
de radio.
UnCirculoInflable.Desinflar(20) ;
```

Un objeto de clase ***derivada o heredada*** seguirá teniendo todas las mismas componenetes de la clase base de la cual hereda. En nuestro ejemplo el objeto **UnCirculoInflable** tendrá las componentes ***pri_x***, ***pri_y*** y ***pri_radio*** que serán también privadas para sus clientes.

IV.6.4 FUNCIONES VIRTUALES Y ENLACE DINÁMICO.

Para poder aprovechar las componentes ***pri_visible***, ***pri_x***, y ***pri_y*** utilizadas dentro de **CIRCULO** tenemos, que definir a las mismas en una sección **protected** lo que significa que debemos de preveer esta extensión, para mantener el encapsulamiento, dentro de C++ lo resolvemos con el concepto de *funciones virtuales*. Una *función virtual* es una función precedida en su definición por la especificación virtual.

Así la clase **CIRCULO** se define de la siguiente forma :

```
Class CIRCULO
{ private
    int pri_x,pri_y ;
    int pri_visible ;
  protected
    int pri_radio ;
  public
    CIRCULO (int InicX, int InicY, int InicRadio)
      { pri_x=InicX ; pri_y=InicY ; pri_radio=InicRadio ;}
    int X(void) const {return pri_x ;}
    int Y(void) const {return pri_y ;}
    int Radio(void) const {return pri_radio ;}
    int Visible(void) const {return pri_visible ;}
    virtual void Muestra(void) ;
    void Oculta(void) ;
    void Traslada(int NuevoX,int Nuevo Y);
};
```

entonces la clase **CIRCULO-COLOREADO** no tiene que definir la función ***Traslada*** de nuevo y cuando ahora hacemos.

```
Setcolor(YELLOW) ;

CIRCULO-COLOREADO UnCirculoColoreado(100,100,30,BLUE) ;

UnCirculoColoreado.Muestra() ; // El círculo se dibuja en azul.

UnCirculoColoreado.Traslada(150,150) ; // El círculo  se traslada y se
vuelve a dibujar en azul.
```

El hecho de que la misma notación ***MUESTRAQ*** que ocurre dentro de traslada tenga distintas formas o interpretaciones es denominado polimorfismo.

Esta interpretación dinámica del llamado a una función virtual es aplicada cuando la función es llamada a través de una referencia o puntero al objeto.

Otro ejemplo de esto lo tenemos en :

```
CIRCULO *PtrUnCirculo ;
CIRCULO_INFLABLE
*PtrUnCirculo = new
  CIRCULO(100,100,50) ;
CIRCULO_COLOREADO
  *PtrUnCirculoColoreado=new
CIRCULO_COLOREADO(200,200,50,Yellow) ;
```

C++ permite asignaciones de la forma

```
if (condition) PtrUnCirculo = PtrUnCirculoinflable ;
else   PtrUnCirculo = PtrUnCirculoColoreado ;
```

Similar situación polimórfica ocurre cuando trabajamos con parámetros por referencia. Si por ejemplo tenemos.

```
Void F(CIRCULO & X)

{//...
 X.MUESTRA() ;
 // El círculo a mostrar depende del parámetro que se le pase.
 //...
} ;
CIRCULO                Uncirculo ;        CIRCULO_COLOREADO
UnCirculoColoreado ;
//...
F(UnCirculo) ;
// Se le pasa un círculo sin color. F mostrará el círculo con el color
implícito activo
//...
F(UnCirculoColoreado) ; // Se ele pasa un círculo con color
//F  mostrará el círculo con su color.
```

BIBLIOGRAFÍA PARA EL CAPÍTULO IV.

[Katrib.1994]	Katrib Mora, Miguel. "Programación Orientada a Objetos en C++." INFOSYS 1994.
[McCord.1992]	McCord, James W. "Developing Windows Applications with Borland C++ 3.1." Sams Publishing 1992.
[Clement.1996]	Clement Shammas, Namir. "Aprendiendo Visual C++ en 21 Días." Prentice Hall Hispanoamericana, 1996.
[Pappas-Murray.1993]	Pappas, Chris H. y Murray William H. "Manual de Borland C++." McGraw Hill Interamericana 1994.
[Firesmith.1995]	Firesmith, Donald "Inheritance Guidelines" Journal of Object Oriented Programming Vol. 8 (2) Mayo de 1995.
[Budd.1991]	Budd, T. "An Introduction to Object Oriented Programming." Reading Massachusetts, Ed. Addison Wesley 1991.

CONCLUSIONES.

A lo largo de este trabajo hemos mostrado que derivado de diversos factores tales como la mayor exigencia y complejidad en las necesidades de procesamiento e incluso de la evolución de la potencialidad de los equipos, la ingeniería de software ha tenido que cambiar en respuesta a la búsqueda de la satisfacción de esas necesidades complejas; ésto es parte del proceso de evolución (si se nos permite emplear este término de las ciencias naturales) de la ingeniería de software, es así, que a través de la experiencia que nos han dejado otros paradigmas hemos llegado a la conclusión de que estamos ya, desde los años ochenta en una coyuntura generacional en cuanto a lo que a ingeniería de software se refiere (muchas empresas de desarrollo o con departamentos de desarrollo aún no están conscientes del todo), razón por la cual se ha enfrentado el problema de aumentar la productividad y la confiabilidad de los sistemas.

Como respuesta fundamental al problema anterior, se ha considerado que la reutilización ofrece una solución apropiada, aunque dicho tema ha sido abordado bajo diferentes enfoques puede concluirse que la reutilización sí es una alternativa, y no sólo eso sino que debemos ya, enfrentar nuestros problemas de la ingeniería de software bajo un esquema dirigido a la reutilización. Por otro lado hemos visto que para que la reutilización sea satisfactoria no debemos perder de vista que la ingeniería de software es considerada como una tecnología de capital intenso.

Una vez que se decide enfrentar la necesidad de reutilización, hemos fijado nuestra atención en un enfoque particular : la Ingeniería de Software Orientada a Objetos. El paradigma orientado a objetos y en particular la Ingeniería de Software Orientada a Objetos, por sus características esenciales permiten llevar a cabo la reutilización, no sólo de código, sino del diseño y aún del análisis. Los depositos de objetos y las fábricas de software son ya realidades que es necesario aprovechar y enriquecer, al mismo tiempo alrededor de la reutilización hay muchos problemas y vertientes que es necesario resolver y explorar, por lo que no debemos perder de vista este tema, al contrario debemos estar al tanto de sus avances y de la experiencia que sobre él se vayan obteniendo, pero lo más importante es tomarlo de una buena vez.

Por último podemos concluir que el paradigma orientado a objetos puede implementarse de manera satisfactoria a través del uso del lenguaje C++, y que éste cuenta con muchas características intrínsecas que ayudan a llevar a cabo esta labor. Es nuestro deber, además, comentar que la cantidad de sistemas de software, programas de aplicación así como algunos de los llamados sistemas operativos de ambiente gráfico que están siendo creados actualmente han sido creados en C++, lo que nos da pauta a pensar en que esta herramienta se ha convertido en una de las más útiles en el campo de la ingeniería de software.

APÉNDICE.

BIBLIOTECAS REUTILIZABLES DE C++.

En este apéndice se mencionan únicamente algunas de las bibliotecas más importantes que incluye la versión 3.1 de C++. Se eligió en particular a la clase base ***ObjectWindows*** y se incluyeron todas las clases derivadas de ésta.

JERARQUÍA DE LA CLASE *ObjectWindows*.

C++ cuenta con una estructura de clases especial que le permite agrupar objetos de la siguiente manera :

En este diagrama de jerarquía el parentesco se representa de izquierda a derecha.

Por ejemplo : Object es el padre de TWindowsObject, TModule, y de TScroller
TWindowsObject es el padre de TDialog y de TWindow
TWindow es el padre de TBWindow, TControl, TMDIFrame, TEditWindow y de TMDIClient, etc.

Object (Object.h)
TWindowsObject (Windowobj.h)
Twindow (Window.h)
TBWindow (Bwindow.h)
Tcontrol (Control.h)
TScrollBar (Scrollba.h)
Tstatic (Static.h)
Tedit (Edit.h)
TBStatic (Bstatic.h)
TListBox (Listbox.h)
TComboBox (Combobox.h)
TGroupBox (Groupbox.h)
TBGroupBox (Bgrpbox.h)
Tbutton (Button.h)
TCheckBox (Checkbox.h)
TBCheckBox (Bcheckbox.h)
TRadioButton (Radiobutton.h)
TBRadioButton (Bradio.h)
TBButton (Bbutton.h)
TBDivider (Divider.h)
TBStaticBmp (Bstatbmp.h)
TMDIFrame (Mdi.h)
TEditWindow (Editwnd.h)
TFileWindow (Filewnd.h)
TMDIClient (Mdi.h)
Tdialog (Dialog.h)
TFileDialog (Filedial.h)
TInputDialog (inputdia.h)
TSearchDialog (Editwnd.h)
Tmodule (module.h)
Tapplication (Applicat.h)
Tscroller (Scroller.h)

Por otro lado TWindowsObject y TScroller son heredados de TStreamable.

RESUMEN DE LA CLASE *Objetc* (OBJECT.H)

Object es la raíz de la jerarquía de clases ObjectWindows. Como clase abstracta la clase Object define que deben hacer las clases derivadas de ella. Además, proporciona el mecanismo básico y la estructura útil para la verificación de tipos y la encapsulación

Como constructores define a los siguientes :

```
Object();
Object(RObject);
virtual~Object();
```

Y como funciones miembro cuenta con las siguientes :

```
Object::firstThat
Object::forEach
Object::hashValue
Object::isA
Object::isAssociation
Object::isEqual
Object::isSortable
Object::lastThat
Object::nameOf
Object::new
Object::~Object
Object::Object
Object::printOn
```

RESUMEN DE LA CLASE *TWindowsObject* (WINDOBJ.H).

TWindowsObject es una clase abstracta derivada de la clase Object que define el comportamiento fundamental compartido por todos los objetos interfaz ObjectWindows (ventanas, cajas de dialogo y controles).

TWindowsObject conserva una lista de ventanas hijas en su lista de hijos y proporciona el comportamiento que se aplica sobre los objetos de esa lista.

TWindowsObject adicionalmente define funciones miembro que proporcionan acceso a las ventanas de su lista de hijos.

TWindowsObject también juega un papel importante en el mapeo de mensajes que llegan contra la funciones miembro que se utilizan como respuestas.

Como constructores y destructores define a los siguientes :

```
TWindowsObject(PTWindowsObject AParent,PTModule AModule = NULL);
TWindowsObject(StreamableInit);
virtual ~TWindowsObject( );
```

Y como funciones miembro contiene a las siguientes :

```
TWindowsObject::ActivationResponse
TWindowsObject::AfterDispatchHandler
TWindowsObject::BeforeDispatchHandler
TWindowsObject::build
TWindowsObject::CanClose
TWindowsObject::ChildWithID
TWindowsObject::CloseWindow
TWindowsObject::CMExit
TWindowsObject::Create
TWindowsObject::CreateChildren
TWindowsObject::DefChildProc
TWindowsObject::DefCommandProc
TWindowsObject::DefNotificationProc
TWindowsObject::DefWndProc
TWindowsObject::Destroy
TWindowsObject::DisableAutoCreate
TWindowsObject::DisableTransfer
TWindowsObject::DispatchAMessage
TWindowsObject::DispatchScroll
TWindowsObject::DrawItem
TWindowsObject::EnableAutoCreate
TWindowsObject::EnableKBHandler
TWindowsObject::EnableTransfer
TWindowsObject::FirstThat
TwindowsObject::ForEach
TWindowsObject::GetApplication
TWindowsObject::GetChildPtr
TWindowsObject::GetChildren
TWindowsObject::GetClassName
TWindowsObject::GetClient
TWindowsObject::GetFirstChild
TWindowsObject::GetID
TWindowsObject::GetInstance
TWindowsObject::GetLastChild
TWindowsObject::GetModule
TWindowsObject::GetSiblingPtr
TWindowsObject::GetWindowClass
TWindowsObject::hashValue
TWindowsObject::isA
TWindowsObject::isEqual
TWindowsObject::isFlagSet
```

```
TWindowsObject::nameOf
TWindowsObject::Next
TWindowsObject::printOn
TWindowsObject::Previous
TWindowsObject::PutChildPtr
TWindowsObject::PutChildren
TWindowsObject::PubSiblingPtr
TWindowsObject::read
TWindowsObject::Register
TWindowsObject::RemoveClient
TWindowsObject::SetCaption
TWindowsObject::SetFlags
TWindowsObject::SetParent
TWindowsObject::SetTransferBuffer
TWindowsObject::SetupWindow
TWindowsObject::Show
TWindowsObject::ShutDownWindow
TWindowsObject::Transfer
TWindowsObject::TransferData
TWindowsObject::~TWindowsObject
TWindowsObject::TWindowsObject
TWindowsObject::WMActivate
TWindowsObject::WMClose
TWindowsObject::WMCommand
TWindowsObject::WMDestroy
TWindowsObject::WMDrawItem
TWindowsObject::WMHScroll
TWindowsObject::WMNCDestroy
TWindowsObject::WMQueryEndSession
TWindowsObject::WMVScroll
TWindowsObject::write
```

RESUMEN DE LA CLASE *TBWindow* (BWINDOW.H)

TBWindow se deriva de la clase TWindow. TBWindow es una ventana genérica cuyo fondo es gris claro.

TBWindow tiene como constructores los siguientes :

```
TBWindow(PTWindowsObject AParent, LPSTR ATitle,PTModule AModule = NULL);
TBWindow(StreamableInit);
```

Y como funciones miembro a las siguientes :

```
TBWindow::build
TBWindow::GetClassName
TBWindow::GetWindowClass
TBWindow::TBWindow
```

RESUMEN DE LA CLASE *TControl* (CONTROL.H)

TControl unifica clases de control derivadas, tales como TScrollBar y TButton.

Los objetos de control de las clases derivadas son usados para representar elementos de control de interfases en Windows.

Un objeto de control :

-Debe usarse para crear un control dentro de una clase base Twindow.
-Puede usarse para facilitar la comunicación entre la aplicación y los controles de un objeto TDialog.

Cuenta con los siguientes constructores :

```
TControl(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y, int W, int H,
PTModule ATModule = NULL);
TControl(PTWindowsObject AParent, int ResourceId, PTModule,ATModule = NULL);
TControl(StreamableInit);
```

Y con cuenta con las siguientes funciones miembro :

```
TControl::GetId
TControl::ODADrawEntire
TControl::ODAFocus
TControl::ODASelect
TControl::TControl
TControl::WMDrawItem
TControl::WMPaint
```

RESUMEN DE LA CLASE *TScrollBar* (SCROLLBA.H)

Los objetos TScrollBar representan barras de control horizontal y vertical. La mayoría de las funciones miembro de la clase se relacionan con el manejo de las barras de control en lo referente a su posición y rango.

Una característica especial de del tipo TScrollBar es el conjunto de funciones miembro que automáticamente ajustan las barras de desplazamiento en respuesta a los mensajes enviados por las barras de desplazamiento de Windows.

Nunca deben colocarse objetos TScrollBar en ventanas que cuenten, ya sea, con estilos de atributos WS_HSCROLL o WS_VSCROLL.

Esta clase cuenta con los siguientes constructores.

```
TScrollBar(PTWindowsObject AParent, int AnId, int X, int Y, int W, int H, BOOL IsHScrollBar,
PTModule AModule = NULL);
TScrollBar(PTWindowsObject AParent, int ResourceId, PTModule AModule = NULL);
TScrollBar(StreamableInit);
```

Y con las siguientes funciones miembro.

```
TScrollBar::build
TScrollBar::DeltaPos
TScrollBar::GetClassName
TScrollBar::GetPosition
TScrollBar::GetRange
TScrollBar::read
TScrollBar::SBBottom
TScrollBar::SBLineDown
TScrollBar::SBLineUp
TScrollBar::SBPageDown
TScrollBar::SBPageUp
TScrollBar::SBThumbPosition
TScrollBar::SBThumbTrack
TScrollBar::SBTop
TScrollBar::SetPosition
TScrollBar::SetRange
TScrollBar::SetupWindow
TScrollBar::Transfer
TScrollBar::TScrollBar
TScrollBar::write
```

RESUMEN DE LA CLASE *TStatic* (STATIC.H)

Un objeto Tstatic representa una interfaz de texto estático en Windows. Un objeto estático TStatic debe usarse para crear un control estático en un padre TWindow.

También puede usarse para facilitar las modificaciones al texto de un control estático en un objeto TDialog.

Como constructores cuenta con los siguientes :

```
TStatic(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y, int W, int H, WORD
ATextLen, PTModule AModule = NULL);
TStatic(PTWindowsObject AParent, int ResourceId, WORD ATextLen, PTModule AModule =
NULL);
TStatic(StreamableInit);
```

Y como funciones miembro con las siguientes :

```
TStatic::build
TStatic::Clear
TStatic::GetClassName
TStatic::GetText
TStatic::GetTextLen
TStatic::nameOf
TStatic::read
TStatic::SetText
```

TStatic::Transfer TStatic::TStatic TStatic::write

RESUMEN DE LA CLASE *TEdit* (EDIT.H)

Tedit es una clase de objetos interactivos que representan un control de edición. Un objeto TEdit debe usarse para crear un control en una ventana padre TWindow.

Un objeto TEdit puede usarse para facilitar la comunicación entre una aplicación y los controles de edición de la clase TDialog.

Existen dos tipos de controles de edición, aquellos que aceptan únicamente una línea y los que aceptan más de una linea de texto.

En esta clase se heredan dos funciones miembro de importancia de la clase TStatic : GetText y SetText.

Como constructores cuenta con los siguientes :

TEdit(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y, int W, int H, WORD ATextLen, BOOL Multiline);
TEdit(PTWindowsObjectAParent, int ResourceId, WORD ATextLen, PTModule ATModule = NULL);
TEdit(StreamableInit);

Y como funciones miembro con las siguientes :

TEdit::build TEdit::CanUndo TEdit::ClearModify TEdit::CMEditClear TEdit::CMEditCopy TEdit::CMEditCut TEdit::CMEditDelete TEdit::CMEditPaste TEdit::CMEditUndo TEdit::Copy TEdit::Cut TEdit::DeleteLine TEdit::DeleteSelection TEdit::DeleteSubText TEdit::ENErrSpace TEdit::GetClassName TEdit::GetLine TEdit::GetLineFromPos TEdit::GetLineIndex TEdit::GetLineLength TEdit::GetNumLines

```
TEdit::GetSelection
TEdit::GetSubText
TEdit::Insert
TEdit::IsModified
TEdit::Paste
TEdit::Scroll
TEdit::Search
TEdit::SetSelection
TEdit::SetupWindow
TEdit::TEdit
TEdit::Undo
```

RESUMEN DE LA CLASE *TBStatic* (BSTATIC.H)

TBStatic se deriva de la clase TStatic. Un objeto TBStatic es un objeto interfaz que representa una interfaz estática de texto en Windows.

Un objeto TBStatic object debe usarse para crear un control estático en un padre TWindow.

También puede usarse para facilitar las modificaciones de un control estático en un objeto TDialogs.

Esta clase presenta como constructores a los siguientes :

```
TBStatic(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y, int W, int H, WORD
ATextLen, PTModule AModule = NULL);
TBStatic(PTWindowsObject AParent, int ResourceId, WORD ATextLen, PTModule AModule =
NULL);
TBStatic(StreamableInit);
```

Y como funciones miembro a las siguientes :

```
TBStatic::build
TBStatic::GetClassName
TBStatic::TBStatic
```

RESUMEN DE LA CLASE *TListBox* (LISTBOX.H)

Un objeto TListBox es una interfaz que representa un elemento de una lista Windows.

Los objetos TListBox deben usarse para crear listas en un padre TWindow. Un objeto TListBox puede usarse para facilitar la comunicación entre una aplicación y los controles tipo list box de un objeto TDialog.

Las funciones mienmbro de TListBox también sirven como instancias de su clase derivada TComboBox.

Como constructores la clase cuenta con los siguientes :

TListBox(PTWindowsObject AParent, int AnId, int X, int Y, int W, int H, PTModule AModule = NULL);
TListBox(PTWindowsObject AParent, int ResourceId, PTModule AModule = NULL);
TListBox(StreamableInit);
Y como funciones miembro con las siguientes :

TListBox::AddString
TListBox::build
TListBox::ClearList
TListBox::DeleteString
TListBox::FindExactString
TListBox::FindString
TListBox::GetClassName
TListBox::GetCount
TListBox::GetMsgID
TListBox::GetSelCount
TListBox::GetSelIndex
TListBox::GetSelIndexes
TListBox::GetSelString
TListBox::GetSelStrings
TListBox::GetString
TListBox::GetStringLen
TListBox::InsertString
TListBox::SetSelIndex
TListBox::SetSelIndexes
TListBox::SetSelString
TListBox::SetSelStrings
TListBox::TListBox
TListBox::Transfer

RESUMEN DE LA CLASE *TComboBox* (COMBOBOX.H)

Los objetos TComboBox son interfases que representan elementos del tipo combo box en Windows. Un elemento TComboBox puede ser usado para crear un control combo box dentro de un padre TWindow. Un objeto TComboBox puede ser utilizado para facilitar la comunicación entre la aplicación y los controles combo box de un TDialog. Los objetos TComboBox heredan la mayoría de su comportamiento de la clase TListBox.

Como constructores se encuentran los siguientes :

TComboBox (PTWindowsObject AParent, int AnId, int X, int Y, int W, int H, DWORD AStyle, WORD ATextLen, PTModule
ATModule = NULL);
TComboBox(PTWindowsObject AParent, int ResourceId, WORD ATextLen, PTModule ATModule = NULL);
TComboBox(StreamableInit);

Y como funciones miembro a las siguientes :

```
TComboBox::build
TComboBox::Clear
TComboBox::GetClassName
TComboBox::GetEditSel
TComboBox::GetMsgID
TComboBox::GetText
TComboBox::GetTextLen
TComboBox::HideList
TComboBox::nameOf
TComboBox::read
TComboBox::SetEditSel
TComboBox::SetText
TComboBox::SetupWindow
TComboBox::ShowList
TComboBox::TComboBox
TComboBox::Transfer
TComboBox::write
```

RESUMEN DE LA CLASE *TGroupBox* (GROUPBOX.H)

Una instancia de la clase TGroupBox es una interfaz que representa un elemento group box en Windows.

Generalmente, TGroupBox no se usa en cajas de dialogo o ventanas de dialogo (objetos de la clase TDialog), sin embargo, se usan para agrupar elementos en una ventana.

Los objetos de esta clase visualmente agrupan cajas de selección (tales como check y radio buttons) u otros controles. También pueden manejar cambios de estado para sus elementos agrupados.

La clase TGroupBox cuenta con los siguientes constructores :

```
TGroupBox(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y, int W, int H,
PTModule AModule = NULL);
TGroupBox(PTWindowsObject AParent, int ResourceId, PTModule AModule = NULL);
TGroupBox(StreamableInit);
```

y con las funciones miembro siguientes :

```
TGroupBox::build
TGroupBox::GetClassName
TGroupBox::read
TGroupBox::SelectionChanged
TGroupBox::TGroupBox
TGroupBox::write
```

RESUMEN DE LA CLASE *TBGroupBox* (BGRPBOX.H)

La clase TBGroupBox se deriva de la clase TGroupBox. Una instancia de la clase representa una interfaz del tipo group box, sólo que esta aparece con un fondo gris. Los objetos de esta clase visualmente agrupan un grupo de cajas de selección (tales como check y radio buttons) u otros controles. También pueden manejar cambios de estado para sus elementos agrupados.

Cuenta con los constructores siguientes :

TBGroupBox(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y, int W, int H, PTModule AModule = NULL);
TBGroupBox(PTWindowsObject AParent, int ResourceId, PTModule AModule = NULL);
TBGroupBox(StreamableInit);

Y con las funciones miembro siguientes :

TBGroupBox::build
TBGroupBox::GetClassName
TBGroupBox::TBGroupBox

RESUMEN DE LA CLASE *TButton* (BUTTON.H)

Tbutton es una clase de objetos interfaz que representan botones dentro de objetos Windows. Úsese elementos de esta clase para crear botones de control dentro de padres pertenecientes a la clase Twindows. Puede usarse un objeto TBbutton para facilitar la comunucación entre una aplicación y los botones de control de un objeto TDialog.

Existen dos tipos de botones de botones a saber :

Los botones regulares que aparecen con borde delgado y los botones por defecto que aparecen con un borde más ancho y representan la acción por defecto en una ventana. Únicamente puede existir un botón de este tipo en una ventana.

Como constructores se cuenta con los siguientes :

TButton(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y, int W, int H, BOOL IsDefault,PTModule AModule =NULL);
TButton(PTWindowsObject AParent, int ResourceId, PTModule AModule = NULL);
TButton(StreamableInit);

Y como funciones miembro se cuenta a las siguientes :

TButton::BMSetStyle
TButton::build
TButton::GetClassName
TButton::SetupWindow
TButton::TButton
TButton::WMGetDlgCode

RESUMEN DE LA CLASE *TCheckBox* (CHECKBOX.H)

Los objetos TCheckBox actúan como interfases que representan cajas marcables dentro de una ventana padre.

Debe utilizarse un objeto de esta clase para crear un control del tipo check box en una ventana padre, de modo que se facilite la comunicación entre la aplicación y los objetos de control de la clase TDialog.

Los elementos de esta clase tienen dos posiblese estados : seleccionado y no seleccionado.

Las funciones miembro de la clase Tcheckbox están relacionadas con la operación de seleccionar o no alguna opción y por lo tanto pueden formar parte de un grupo funcional.

La clase cuenta con los siguientes constructores :

```
TCheckBox(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y, int W, int H,
PTGroupBox AGroup, PTModule
ATModule = NULL);
TCheckBox(PTWindowsObject AParent, int ResourceId, PTGroupBox AGroup, PTModule
ATModule = NULL);
TCheckBox(StreamableInit);
```

Y con las siguientes funciones miembro :

```
TCheckBox::BNClicked
TCheckBox::build
TCheckBox::Check
TCheckBox::GetCheck
TCheckBox::read
TCheckBox::SetCheck
TCheckBox::TCheckBox
TCheckBox::Toggle
TCheckBox::Transfer
TCheckBox::Uncheck
TCheckBox::write
```

RESUMEN DE LA CLASE *TBCheckBox* (BCHKBOX.H).

La clase TBCheckBox se deriva de la clase TCheckBox, la cual es un clase de objetos interfaz que representa interfases BWCC

Debe usarse esta clase para crear un check box del tipo BWCC en una ventana padre TWindow.

Puede usarse para facilitar la comunicación entre la aplicación y los controles de la clase TDialog. Las funciones miembro de la clase TBCheckBox están relacionadas con la operación de seleccionar o no alguna opción y por lo tanto pueden formar parte de un grupo funcional.

Los elementos dela clase TBCheckBox deben usarse para crear objetos de control check box en una ventana padre TWindow, los cuales pueden usarse par facilitar la comunicación entre la aplicación y los objetos de control de un objeto del tipo TDialog.

Los elementos de esta clase tienen dos posiblese estados : seleccionado y no seleccionado.

Como constructores la clase tiene los siguientes :

TBCheckBox(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y, int W, int H, PTGroupBox AGroup, PTModule ATModule = NULL);
TBCheckBox(PTWindowsObject AParent, int ResourceId, PTGroupBox AGroup, PTModule ATModule = NULL);
TBCheckBox(StreamableInit);

Y como funciones miembro define a las siguientes :

TBCheckBox::build TBCheckBox::GetClassName TBCheckBox::TBCheckBox

RESUMEN DE LA CLASE *TRadioButton* (RADIOBUT.H)

Un objeto de esta clase es un objeto interfaz que representa un radio button en un elemento Window. Debe usarse para crear un control del tipo radio button dentro de un padre Twindow.

Use estos objetos para crear controles radio button que faciliten la comunicación entre una aplicación y un control TDialog.

Estos objetos tiene dos estados posibles : seleccionados o no seleccionados. Además heredan el manejo de sus estados de las funciones miembro de la clase base TCheckBox.

Como constructores define a los siguientes :

TRadioButton(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y, int W, int H, PTGroupBox AGroup,
PTModule AModule = NULL);
TRadioButton(PTWindowsObject AParent, int ResourceId, PTGroupBox AGroup, PTModule AModule = NULL);
TRadioButton(StreamableInit);

Y como funciones miembro define a las siguientes :

TRadioButton::BNClicked TRadioButton::build TRadioButton::TRadioButton

RESUMEN DE LA CLASE *TBRadioButton* (BRADIO.H)

TBRadioButton se deriva de la clase TRadioButton. Un objeto de esta clase es un objeto interfaz que representa un radio button en un elemento Window. Debe usarse para crear un control del tipo radio button dentro de un padre Twindow.

Use estos objetos para crear controles radio button que faciliten la comunicación entre una aplicación y un control TDialog.

Estos objetos tiene dos estados posibles : seleccionados o no seleccionados. Además heredan el manejo de sus estados de las funciones miembro de su clase base TCheckBox.

La clase cuenta con los siguientes constructores :

```
TBRadioButton(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y, int W, int H,
PTGroupBox AGroup, PTModule AModule= NULL);
TBRadioButton(PTWindowsObject AParent, int ResourceId, PTGroupBox AGroup, PTModule
AModule = NULL);
TBRadioButton(StreamableInit);
```

Y con las siguientes funciones miembro :

```
TBRadioButton::build
TBRadioButton::GetClassName
TBRadioButton::TBRadioButton
```

RESUMEN DE LA CLASE *TBButton* (BBUTTON.H)

La clase TBButton se deriva de la clase TButton. Los objetos de la clase TBButton son interfases que representan mapas de bits de botones.

Deben usarse para crear controles botón en un padre TWindow, además, deben usarse para facilitar la comunicación entre la aplicación y los botones de control de un objeto TDialog.

Existen dos tipos de botones los regulares y los botones por defecto.

La clase cuenta con los constructores siguientes :

```
TBButton(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y, int W, int H, BOOL
IsDefault, PTModule AModule = NULL);
TBButton(PTWindowsObject AParent, int ResourceId, PTModule   AModule = NULL);
BButton(StreamableInit);
```

Y con la funciones miembro siguientes :

```
TBButton::build
TBButton::GetClassName
TBButton::TBButton
```

RESUMEN DE LA CLASE *TBDivider* (BDIVIDER.H)

TBDivider es una clase derivada de la clase TControl. Un objeto TBDivider es un objeto interfaz que representa un divisor estático. Los divisores aparecen en dos presentaciones : en bajo relieve o en alto relieve.

La clase cuenta con los constructores siguientes :

```
TBDivider(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y, int W, int H, BOOL
IsVertical, BOOL IsBump, PTModule AModule = NULL);
TBDivider(PTWindowsObject AParent, int ResourceId, PTModule AModule = NULL);
TBDivider(StreamableInit);
```

Y con las funciones miembro siguientes:

```
TBDivider::build
TBDivider::GetClassName
TBDivider::TBDivider
```

RESUMEN DE LA CLASE *TBStaticBmp* (BSTATBMP.H)

Los objetos de la clase TBStaticBmp son objetos interfaz que representan una interfaz estática de texto que se usa para desplegar imágenes. TBStatic se deriva de la clase TControl. Un objeto TBStaticBmp debe usarse para crear un control que presenta mapas de bits dentro de un objeto TWindow. También puede usarse para facilitar las modificaciones que se hagan a un mapa de bits en un objeto TDialogs.

La clase cuenta con los constructores siguientes:

```
TBStaticBmp(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y, int W, int H,
PTModule AModule = NULL);
TBStaticBmp(PTWindowsObject AParent, int ResourceId, PTModule AModule = NULL);
TBStaticBmp(StreamableInit);
```

Y con las funciones miembro siguientes:

```
TBStaticBmp::build
TBStaticBmp::GetClassName
TBStaticBmp::TBStaticBmp
```

RESUMEN DE LA CLASE *TMDIFrame* (MDI.H)

Las ventanas plantilla MDI (Multiple Document Interface), representadas por la clase TMDIFrame, permiten que sobre ellas se traslapen otras ventanas. De tal modo que son útiles como ventanas principales de una aplicación.

Estos objetos automáticamente manejan la creación e inicialización de un ventana cliente requerida por Windows. Los objetos TMDIFrames proporcionan también funciones miembro que manipulan ventanas hijas MDI, tales como TileChildren y CloseChildren.

La clase cuenta con los constructores y destructores siguientes :

```
TMDIFrame(LPSTR ATitle, int MenuId, PTModule AModule = NULL);
TMDIFrame(LPSTR ATitle, LPSTR MenuName, PTModule AModule = NULL);
TMDIFrame(HWND AnHWindow, HWND ClientWnd);
TMDIFrame (StreamableInit);
virtual ~TMDIFrame ();
```

Y con las funciones miembro siguientes :

```
TMDIFrame::ArrangeIcons
TMDIFrame::build
TMDIFrame::CascadeChildren
TMDIFrame::CloseChildren
TMDIFrame::CMArrangeIcons
TMDIFrame::CMCloseChildren
TMDIFrame::CMCreateChild
TMDIFrame::CMTileChildren
TMDIFrame::CreateChild
TMDIFrame::GetClassName
TMDIFrame::GetClient
TMDIFrame::GetWindowClass
TMDIFrame::InitChild
TMDIFrame::InitClientWindow
TMDIFrame::read
TMDIFrame::SetupWindow
TMDIFrame::TileChildren
TMDIFrame::~TMDIFrame
TMDIFrame::TMDIFrame
TMDIFrame::WMActivate
TMDIFrame::write
```

RESUMEN DE LA CLASE *TEditWindow* (EDITWND.H)

TEditWindow define a una ventana especializada que permite entrada de texto y edición.. Además incluye un control de edición de pantalla completa que maneja edición de texto.

La clase TFileWindow, es una clase especializada de TEditWindow,que pérmite la edición de archivos.

Como constructores la clase cuenta con los siguientes :

```
TEditWindow(PTWindowsObject AParent, LPSTR ATitle, PTModule AModule = NULL);
TEditWindow(StreamableInit);
```

Y con las funciones miembro siguientes:

```
TEditWindow::build
TEditWindow::CMEditFind
TEditWindow::CMEditFindNext
TEditWindow::CMEditReplace
TEditWindow::DoSearch
TEditWindow::readTEditWindow::TEditWindow
TEditWindow::WMSetFocus
TEditWindow::WMSizeTEditWindow::write
```

RESUMEN DE LA CLASE *TFileWindow* (FILEWND.H)

TFileWindowes una clase de edición de archivos basada en la clase TEditWindow. Las ventanas de archivos utilizan cajas de dialogo para abrir y salvar archivos. Sus funciones miembro manejan la caja de dialogo de los archivos y automáticamente responden a instrucciones típicas para archivos tales como las siguientes : Open, Read, Write, Save, and SaveAs.

La clase cuenta con los constructores y destructores siguientes :

```
TFileWindow(PTWindowsObject AParent, LPSTR ATitle, LPSTR AFileName, PTModule
AModule = NULL);
TFileWindow(StreamableInit);
virtual ~TFileWindow( );
```

Y con las funciones miembro siguientes:

```
TFileWindow::build
TFileWindow::CanClear
TFileWindow::CanClose
TFileWindow::CMFileNew
TFileWindow::CMFileOpen
TFileWindow::CMFileSave
TFileWindow::CMFileSaveAs
TFileWindow::NewFile
TFileWindow::Open
TFileWindow::Read
TFileWindow::ReplaceWith
TFileWindow::Save
TFileWindow::SaveAs
TFileWindow::SetFileName
TFileWindow::SetupWindow
TFileWindow::~TFileWindow
TFileWindow::TFileWindow
TFileWindow::Write
```

RESUMEN DE LA CLASE *TMDIClient* (MDI.H)

La venta cliente MDI (Multiple Document Interface) (representada por un objeto TMDIClient) maneja la ventana MDI hija de una ventana TMDIFrame.

Esta clase cuenta con los constructores y destructores siguientes :

```
TMDIClient(PTMDIFrame AParent, PTModule AModule = NULL);
TMDIClient(PTMDIFrame AParent, HWND AnHWindow,
PTModule AModule = NULL);
TMDIClient(StreamableInit);
virtual ~TMDIClient( );
```

Y con las funciones miembro siguientes:

```
TMDIClient::ArrangeIcons
TMDIClient::build
TMDIClient::CascadeChildren
TMDIClient::GetClassName
TMDIClient::read
TMDIClient::TileChildren
TMDIClient::~TMDIClient
TMDIClient::TMDIClient
TMDIClient::WMPaint
TMDIClient::WMMDIActivate
TMDIClient::write
```

RESUMEN DE LA CLASE *TDialog* (DIALOG.H)

Los objetos de la clase TDialog representan cajas de dialogo tanto del tipo modal como del tipo modeless. Un objeto TDialog tiene una definición de recursos que describe el lugar y la apariencia de sus controles.

Un objeto TDialog se asocia con una interfaz al llamar a su función constructora respectiva.

Esta clase cuenta con los constructores y destructores siguientes:

```
TDialog(PTWindowsObject AParent, LPSTR AName, PTModule AModule = NULL);
TDialog(PTWindowsObject AParent, int ResourceId, PTModule AModule = NULL);
TDialog(StreamableInit);
virtual~TDialog( );
```

Y con las funciones miembro siguientes:

```
TDialog::build
TDialog::Cancel
TDialog::CloseWindow
TDialog::Create
TDialog::Destroy
```

```
TDialog::Execute
TDialog::GetClassName
TDialog::GetItemHandle
TDialog::GetWindowClass
TDialog::isA
TDialog::nameOf
TDialog::Ok
TDialog::read
TDialog::SendDlgItemMsg
TDialog::SetCaption
TDialog::SetupWindow
TDialog::ShutDownWindow
TDialog::~TDialog
TDialog::TDialog
TDialog::WMClose
TDialog::WMInitDialog
TDialog::WMQueryEndSession
TDialog::write
```

RESUMEN DE LA CLASE *TFileDialog* (FILEDIAL.H)

La clase TFileDialog crea cajas de dialogo que permiten al usuario seleccionar el nombre de un archivo al momento de abrir o cerrar.

Dentro de FILEDIL.DLG se encuentran dos definiciones de recursos para una caja de dialogo : abrir y salvar.

La clase cuenta con los constructores siguientes:

TFileDialog(PTWindowsObject AParent, int ResourceId, LPSTR AFilePath, PTModule AModule = NULL);
TFileDialog(StreamableInit);

Y con las funciones miembro siguientes:

```
TFileDialog::build
TFileDialog::CanClose
TFileDialog::HandleDList
TFileDialog::HandleFList
TFileDialog::HandleFName
TFileDialog::SelectFileName
TFileDialog::SetupWindow
TFileDialog::TFileDialog
TFileDialog::UpdateFileName
TFileDialog::UpdateListBoxes
```

RESUMEN DE LA CLASE *TInputDialog* (INPUTDIA.H)

La clase TInputDialog proporciona una caja de dialogo para recuperar texto insertado por el usuario. Cuando esta caja de dialogo se construye, se le especifica su titulo y su texto de entrada por defecto.

La clase cuenta con los constructores siguientes:

TInputDialog(PTWindowsObject AParent; LPSTR ATitle, LPSTR APrompt, LPSTR ABuffer, WORD ABufferSize, PTModule AModule = NULL);
TInputDialog(StreamableInit);

Y con las funciones miembro siguientes:

TInputDialog::build
TInputDialog::SetupWindow
TInputDialog::TInputDialog
TInputDialog::TransferData
TInputDialog::write

RESUMEN DE LA CLASE *TSearchDialog* (EDITWND.H)

Recupera texto a través de las opciones Search/Replace proporcionadas por el usuario.

La estructura TSearchStruct pasada al constructor TSearchDialog transfiere datos en dos sentidos entrada y salida. STDWNDS.DLG contiene dos definiciones de recursos, una para buscar datos y otra para remplazar texto.

Para utlilizar un objeto TSearchDialog debe compilarse y anexarse a la aplicación Object Windows.

La clase cuenta con los constructores siguientes:

TSearchDialog(PTWindowsObject AParent, int ResourceID,, TSearchStruct _FAR &SearchStruct, PTModule AModule = NULL);

RESUMEN DE LA CLASE *TModule* (MODULE.H)

Las bibliotecas de enlace dinámico de ObjectWindows construyen una instancia de Tmodule, la cual actúa como un objeto que permanece en el módulo de la biblioteca DLL.

Las aplicaciones ObjectWindows construyen una inbstancia de Tapplication, derivada de Tmodule.

La clase TModule define el comportamiento compartido por ambas bibliotecas y módulos de la aplicación. Las funciones miembro de esta clase proporcionan capacidad para el manejo de memoria y el procesamiento de errores.

La clase cuenta con los siguientes constructores y destructores :

TModule(LPSTR AName, HANDLE LibhInstance, LPSTR LiblpCmdLine);
virtual ~TModule();

Y con las funciones miembro siguientes:

TModule::Error
TModule::ExecDialog
TModule::GetClientHandle
TModule::GetParentObject
TModule::hashValue
TModule::isA
TModule::isEqual
TModule::LowMemory
TModule::MakeWindow
TModule::nameOf
TModule::printOn
TModule::RestoreMemory
TModule::~TModule
TModule::TModule
TModule::ValidWindow

RESUMEN DE LA CLASE *TApplication* (APPLICAT.H)

La clase TApplication es una clase derivada de la clase TModule y actúa como un objeto de reemplazo para un modulo de una aplicación Windows.

TApplication y TModule proporcionan el comportamiento básico requerido para una aplicación Windows. Las funciones miembro de TAapplication llevan a cabo inicialización de instancias y procesamiento de mensajes.

Esta clase cuenta con los constructores y destructores siguientes:

TApplication(LPSTR AName, HANDLE AnInstance, HANDLE APrevInstance, LPSTR ACmdLine, int ACmdShow);
~TApplication();

Y con las funciones miembro siguientes:

TApplication::CanClose
TApplication::IdleAction
TApplication::InitApplication
TApplication::InitInstance
TApplication::InitMainWindow
TApplication::isA
TApplication::MessageLoop
TApplication::nameOf
TApplication::ProcessAccels

```
TApplication::ProcessAppMsg
TApplication::ProcessDlgMsg
TApplication::ProcessMDIAccels
TApplication::Run
TApplication::SetKBHandler
TApplication::~TApplication
TApplication::TApplication
```

RESUMEN DE LA CLASE *TScroller* (SCROLLER.H)

Tscroller es una clase cuyos objetos ayudan a desplazar elementos al momento de desplegarlos en una ventana.

Para aprovechar la funcionalidad que ofrece es necesario colocar al elemento de desplazamiento en el constructor del descendiente del objeto TWindow.

Esta clase cuenta con los constructores y destructores siguientes:

```
TScroller(PTWindow TheWindow, int TheXUnit, int TheYUnit, long TheXRange, long
TheYRange);
TScroller(StreamableInit);
~TScroller( );
```

Y con las funciones miembro siguientes:

```
TScroller::AutoScroll
TScroller::BeginView
TScroller::build
TScroller::EndView
TScroller::hashValue
TScroller::HScroll
TScroller::isA
TScroller::isEqual
TScroller::IsVisibleRect
TScroller::nameOf
TScroller::printOn
TScroller::read
TScroller::ScrollBy
TScroller::ScrollTo
TScroller::SetPageSize
TScroller::SetRange
TScroller::SetSBarRange
TScroller::SetUnits
TScroller::~TScroller
TScroller::TScroller
TScroller::VScroll
TScroller::write
TScroller::XRangeValue
TScroller::XScrollValue
```

TScroller::YRangeValue TScroller::YScrollValue

RESUMEN DE LA CLASE *TStreamable* (OBJSTRM.H)

Las clase que heredan de la clase Tstreamable se conocen como clases que pueden leerse o escribirse a partir de streams. Ya que TStreamable es una clase abstracta, no es posible crear instancias de esta clase, las clases que hereden de la clase deberán redefinir sus tres funciones virtuales puras.

La clase cuenta con las funciones miembro siguientes:

TStreamable::read TStreamable::streamableName TStreamable::write

Y con las clases amigas Ipstream y opstream.