UNIVERSIDAD LASALLISTA BENAVENTE

ESCUELA DE INGENIERÍA EN COMPUTACIÓN
CON ESTUDIOS INCORPORADOS A LA
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
CLAVE 8793-16

"PROPUESTA METODOLÓGICA PARA EL DESARROLLO DE SOFTWARE ORIENTADO A OBJETOS"

TESIS

PARA OBTENER EL TITULO DE:

INGENIERA EN COMPUTACIÓN

PRESENTA:

LAURA KARINA GARCÍA RODRÍGUEZ

ASESOR:

ING. LAURA LISBETH RAMÍREZ TREJO

CELAYA, GUANAJUATO, FEBRERO DE 2003

Propuesta metodológica para el desarrollo de software orientado a objetos

A mis padres
Por haber comenzado con mi educación e impulsarme a seguir adelante...

A mis suegros
Por su comprensión y apoyo constante...

A mi esposo Ricardo
Por haber estado conmigo siempre y confiar en mí...

A mi hijo Ricardo
Por ser mi más hermoso impulso y mi mayor orgullo...

...Gracias

"Propuesta metodológica para el desarrollo de software orientado a objetos"

Introducción

Capítulo 1

Conceptos y principios básicos de la ingeniería de software orientada a objetos (OO)

Capítulo 2
Análisis orientado a objetos (AOO)

E

Capítulo 3
Diseño orientado a objetos (DOO)

Capítulo 4
Pruebas orientadas a objetos (PrOO)

Capítulo 5
Métricas orientadas a objetos

Capítulo 6

Propuesta metodológica orientada a objetos

Introducción

Una de las preocupaciones actuales más urgentes de la industria de la computación es la de crear software de manera rápida y a más bajo costo. Para hacer un buen uso de las computadoras. La alta calidad es esencial en el desarrollo del software, ya que una calidad pobre es un desperdicio de dinero.

Las necesidades de crear un mejor software se aplica tanto en el interior de la propia industria del software como dentro de la empresas de todo tipo que crean sus propias aplicaciones de computación. Las organizaciones relacionadas con la tecnología de la información necesitan crear y modificar aplicaciones en forma mucho más rápida. Si el desarrollo de aplicaciones tarda de dos a tres años, las empresas no pueden crear más aplicaciones ni reaccionar ante la competencia de manera rápida.

Es probable que los medios para dar el salto mencionado ya estén a la vista. En lugar de una técnica, se requiere la combinación de muchas herramientas y técnicas. Ellas se resumen en las técnicas orientadas a objetos.

Las técnicas orientadas a objetos modifican el punto de vista de los analistas de sistemas de información acerca del mundo. En vez de pensar en los procedimientos y su descomposición, piensan en objetos y en su comportamiento.

El análisis y diseño OO, son vías poderosas para pensar en sistemas complejos ya que el análisis y el diseño *convencionales* se hacen cada vez más difíciles al crecer los sistemas y volverse más complejos, hasta que, en última instancia, limitan la complejidad que podemos controlar.

El punto de vista orientado a objeto no modela en términos de lenguajes de programación sino que modela la forma en que las personas comprenden la realidad. La comprensión y el conocimiento de las personas es el componente esencial en el diseño de cualquier sistema. Este punto de vista nos permite analizar cualquier área de la realidad humana, no sólo la del procesamiento de

datos.

El software de computadora es una de las tecnologías que tienen un impacto en la sociedad humana desde hace más de cuarenta años. Se trata de un mecanismo que automatiza negocios, industrias y gobiernos, un medio de transmisión que captura programas, documentos y datos para que pueda ser utilizada por otras personas, que permite examinar el conocimiento en conjunto de alguna corporación.

El enfoque orientado a objetos se propuso a finales de los años 60 y que actualmente se ha convertido en un modelo de elección para muchos productores de software, ya que lleva un desarrollo de software más rápido y programas de mejor calidad, utilizando un lenguaje sencillo y de fácil entendimiento tanto para el desarrollador como para el cliente. Adicionalmente son más fáciles de adaptar y escalar.

Por tal motivo, los desarrolladores de software que implementen la técnica orientada a objetos que se propone en el último capítulo, se convencerán de que es el mejor enfoque que se puede emplear en la realización de sistemas.

En esta tesis que tiene por nombre **"Propuesta metodológica de ingeniería de software orientada a objetos"**, se muestra lo que se debe llevar a cabo en el desarrollo del software orientado a objetos desde su inicio hasta la terminación del mismo y consta de seis capítulos de los cuales se hace una mención de su contenido para una orientación al lector que decida conocer lo que aporta esta tesis en su conocimiento.

En el **primer capítulo** se describe las características y conceptos básicos que se utilizan en el desarrollo de software orientado a objetos para darle una visión completa y general al lector para su mayor entendimiento y comprensión.

Después de conocer los conceptos básicos o generales continuamos con el **segundo capítulo**, en donde se ilustra o se da a conocer la información de los objetos que serán relevantes en el desarrollo del sistema, como se relacionan

entre sí y como se comportan dentro del sistema, de manera que podamos crear un diseño eficaz.

En el **tercer capítulo**, transforma el análisis orientado a objetos hacia la construcción del diseño del sistema para convertirlo en un prototipo que se utilice o desarrolle en la construcción del software orientado a objetos.

En el **cuarto capítulo**, después de haber diseñado y desarrollado el software prototipo, se debe verificar que cumpla con los requerimientos solicitados, en caso contrario se deberá modificar antes de ser entregado el proyecto final.

Quinto capítulo, se revisa que el proyecto final cumpla la calidad suficiente para asegurar un éxito en el proyecto y que se hayan cumplido todos los requerimientos del cliente que solicite este servicio.

Por último en el **sexto capítulo** se realiza una propuesta de los pasos o técnicas que se deben emplear en la realización de software y de las ventajas que podemos obtener los desarrolladores de software con la tecnología de objetos en el desarrollo de sistemas cumpliendo con cada una de sus fases.

Debo aclarar que la estructura o el orden que lleva esta tesis es debido a que se establece una comparación entre el software estructurado o convencional al de el software orientado a objetos. Ya que los ciclos de vida entre ambos difieren mucho, pero para el programador puede resultarle más familiar el orden que lleva a cabo esta tesis, he aquí la razón de su estructura.

Capítulo 1

Conceptos y principios básicos de la ingeniería de software orientado a objetos (OO)

1.1 Antecedentes

Nos encontramos en un mundo lleno de objetos que existen en la naturaleza, en los negocios y en los productos que usamos. Estos pueden ser clasificados, descritos, organizados, combinados, manipulados y creados. Debido a esto se propone una visión orientada a objetos para la creación de software hecha por computadora, de manera que podamos entenderlo y desarrollarlo mucho mejor.

Un enfoque orientado a objetos se propuso a finales de los años 60, sin embargo han necesitado casi veinte años para llegar a ser ampliamente usadas. Durante la mitad de los años 90, la ingeniería de software orientada a objetos se convirtió en el **paradigma** (modelo, ejemplo o molde) de elección para muchos productores de software, profesionales de la ingeniería y un gran número de sistemas de información. A medida que pasa el tiempo las tecnologías de objetos hay sustituido a los enfoques clásicos de desarrollo de software, ya que las ***tecnologías de objeto***[1] llevan un número de beneficios a niveles de dirección y técnicos.

¿Usted se preguntará el porqué?, es muy simple, las tecnologías de objetos llevan a reutilizar y la reutilización (de componentes de software) lleva a un desarrollo de software más rápido y programas de mejor calidad. El software orientado a objetos es más fácil de mantener debido a que su estructura es compuesta (consta de varios procedimientos o subsistemas). Esto permite tener menor frustración en el ingeniero de software y al cliente cuando se deben hacer cambios, adicionalmente, son más fáciles de adaptar y escalar (por ejemplo, pueden crearse grandes sistemas ensamblando subsistemas reutilizables).

1.2 El paradigma orientado a objetos

Durante años el término orientado a objetos **(OO)** se usó como un enfoque de lenguajes orientados a objetos (Ada 95, C++, Eiffel, Smalltalk).

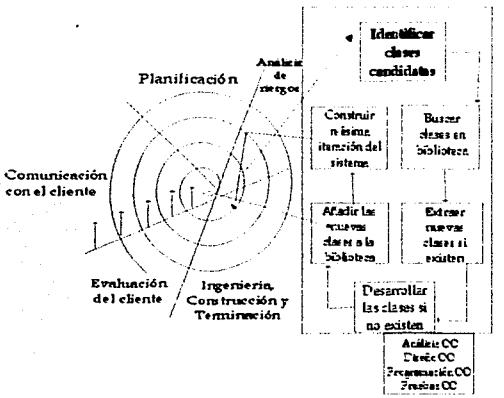
[1] ***tecnologías de objetos*** se usa para encerrar todos los aspectos de una visión orientada a objetos e incluye el análisis, diseño y métodos de prueba, lenguajes de programación, herramientas, bases de datos, y aplicaciones que fueron creadas usando el enfoque orientado a objetos.

Hoy día, el **paradigma OO** encierra una visión completa de ingeniería de software. Esta tecnología debe tener un impacto en todo el proceso, los ingenieros de software y sus directores deben considerar los siguientes elementos:

- Análisis de requisitos orientados a objetos (AROO)
- Diseño orientado a objetos (DOO)
- Análisis de dominio orientado a objetos (ADOO)
- Sistemas de gestión de bases de datos orientado a objetos (SGBDOO)
- Ingeniería de software orientada a objetos asistida por computadora (ISOOAC)
- Ya que si se desarrolla un simple uso de programación orientada a objetos no brindará los mejores resultados.

Los sistemas OO tienden a evolucionar con el tiempo. Por esto el modelo evolutivo de proceso que fomenta el ensamblaje (reutilización) de componentes es el mejor paradigma para ingeniería de software OO.

Figura 1.1 Modelo de proceso OO



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

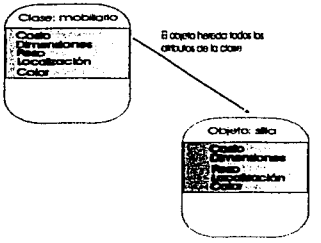
El proceso OO (Figura 1.1) se mueve a través de un espiral evolutiva que comienza con comunicación con el usuario. Es aquí donde se define el problema y se identifican las clases básicas del problema. La planificación y el análisis de riesgos establecen una base para el plan de proyecto OO. El trabajo técnico sigue el camino iterativo mostrado en la caja sombreada. La ingeniería de software OO hace hincapié en la reutilización. Por lo tanto, las clases se buscan en una biblioteca (de clases OO existentes) antes de construirse. Cuando una clase no se encuentra en la biblioteca, el desarrollador de software aplica **análisis orientado a objetos (AOO), diseño orientado a objetos(DOO), programación orientada a objetos (POO) y pruebas orientas a objetos (PrOO)** para crear la clase y los objetos derivados de la clase. La nueva clase se pone en la biblioteca de tal manera que pueda ser reutilizada en el futuro.

A medida que el análisis OO y los modelos de diseño evolucionan, se vuelve aparente la necesidad de clases adicionales; por ello, el paradigma arriba descrito trabaja mejor para la OO.

1.3 Conceptos de orientación a objetos

Para entender la visión orientada a objetos, consideremos una silla. La silla es un miembro (*instancia*) de una *clase* mucho más grande que llamaremos mobiliario. Un conjunto de atributos genéricos puede asociarse con cada objeto, en la clase mobiliario. Por ejemplo, todo mueble tiene un costo, dimensiones, peso, localización y color entre otros. Éstos son aplicables a cualquier elemento sobre el que se hable. Como silla es un miembro de la clase mobiliario, silla *hereda todos* los atributos definidos para la clase.

Figura 1.2 Ilustración de la herencia de clase a objeto



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

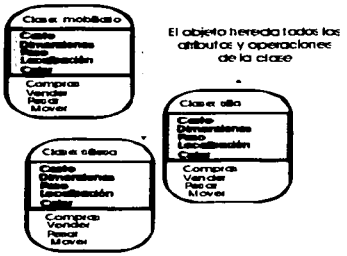
Una vez definida la clase (figura 1.2), los atributos pueden reutilizarse para crear nuevas instancias de la clase. Por ejemplo, supongamos que tenemos que definir un nuevo objeto llamado sillesa (un cruce entre una silla y una mesa) que es un miembro de la clase mobiliario. La sillesa hereda todos los atributos de mobiliario.

Todo objeto en la clase mobiliario puede manipularse de varias maneras. Puede comprarse y venderse, modificarse físicamente (por ejemplo, se puede eliminar una pata o pintar el objeto) o moverse de un lugar a otro. Cada una de estas *operaciones* («servicios» o «métodos») modificará uno o más atributos del objeto. Por ejemplo, si el atributo *localización* es un dato compuesto como:

localización = edificio + piso + habitación

entonces una operación denominada *mover* modificaría uno o más de los elementos dato *(edificio, piso* o *habitación)* que conforman el atributo *localización.* Para hacer esto, *mover* debe tener «conocimiento» sobre estos elementos. La operación *mover* puede usarse para una silla o una mesa, debido a que ambas son instancias de la clase mobiliario. Todas las operaciones válidas (*comprar, vender, pesar)* de la clase mobiliario están «conectadas» a la definición del objeto (figura 1.3) y son heredadas por todas las instancias de ésta.

Figura 1.3 Herencia de operaciones de clase a objeto



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

El objeto silla (y todos los objetos en general) encapsula[2] datos (los valores de los atributos que definen la silla), operaciones (las acciones que se aplican para cambiar los atributos de la silla), otros objetos (pueden definirse objetos compuestos), constantes (fijar valores) y otra información relacionada.

Después de este ejemplo podrá ser más fácil entender un concepto formal.

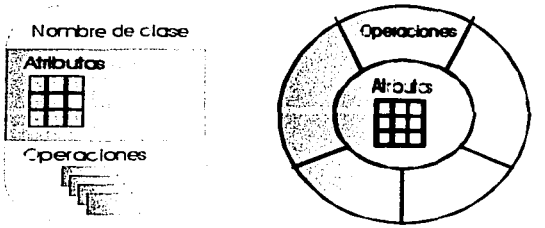
1.3.1 Clases y objetos

Una *clase* contiene un conjunto de atributos que la describen y un conjunto de operaciones que definen su comportamiento (figura 1.4).

Figura 1.4 Representación alternativa de una clase orientada a objetos

[2] El *encapsulamiento* significa que toda esta información se encuentra empaquetada bajo un nombre y puede reutilizarse como una especificación o componente del programa.

Figura 1.4 Representación alternativa de una clase orientada a objetos



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

Los atributos se encuentran encerradas por una muralla (llamada *operaciones métodos o servicios*) capaces de manipular los datos. La única forma de alcanzar los atributos (y operar sobre ellos es a través de alguno de los métodos que forman la muralla. Por tanto, la clase encapsula datos <dentro de la muralla> y el proceso que manipula los datos (métodos que componen la muralla). **Esto posibilita el ocultamiento** de información y reduce el impacto asociados a cambios.

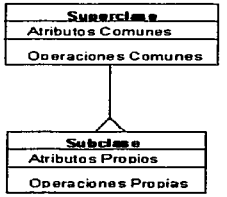
Como estos métodos tienden a manipular un número limitado de atributos y como la comunicación ocurre sólo a través de estos métodos, la clase tiende a un acoplamiento con otros elementos del sistema. Todas estas características conducen a un software de alta calidad.

Por definición, **todos los objetos** que existen dentro de una clase **heredan sus atributos** y las operaciones disponibles para la manipulación de los atributos.

1.3.1.1 superclase, subclase y jerarquía de clases

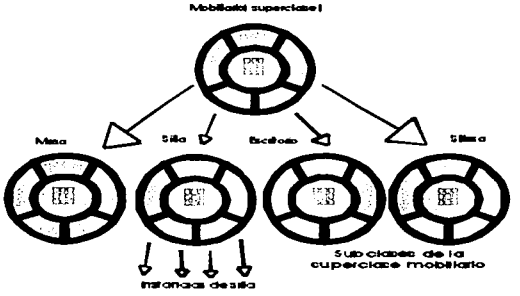
Una *superclase* es una colección de clases y una *subclase* es una instancia de una clase (figura 1.5).

Figura 1.5 Representación de una superclase y una subclase



Fuente: www.google.com

Figura 1.6 Ejemplo de una jerarquía de clases



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

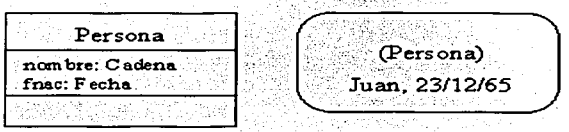
Estas definiciones implican la existencia de una ***jerarquía de clases*** en la cual los atributos y operaciones de la superclase son heredados por subclases que pueden añadir, cada una de ellas, atributos privados y métodos (figura 1.6).

1.3.2 Atributos

Los atributos están asociados a clases y objetos, y que ellos describen la clase o el objeto de alguna manera.

Las entidades de la vida real están a menudo descritas con palabras que indican características estables. La mayoría de los objetos físicos tienen características tales como forma, peso, color, y tipo de material. Las personas tienen características incluyendo fecha de nacimiento, padres, nombres y color de ojos (figura 1.7).

Figura 1.7 Representación de una entidad en la vida real



Fuente: www.google.com

Una característica puede verse como una relación entre una clase y cierto dominio.[3] Por ejemplo, una clase coche tiene un atributo color. El dominio de valores de color es: blanco, negro, rojo, azul, etc.

1.3.3 Operaciones, métodos y servicios

Un objeto encapsula datos (representados como una colección de atributos) y los algoritmos[4] que procesan estos datos son llamados operaciones, métodos o servicios. La implementación específica de una operación determinada en una clase determinada se denomina *método*.

[3] Un *dominio* es un conjunto de valores específicos.
[4] *Algoritmo* conjunto de pasos ordenados para resolver un problema, tal como una fórmula matemática o las instrucciones de un programa.

Cada una de las operaciones encapsuladas por un objeto proporciona una representación de uno de los comportamientos del objeto. Por ejemplo, la operación *DeterminarColor* para el objeto automóvil extraerá el color almacenado en el atributo color. La existencia de esta operación es que la clase automóvil ha sido diseñada para recibir un estímulo (mensaje) que requiere el color de una instancia particular de una clase. Cada vez que el objeto inicia un estímulo, éste inicia un cierto comportamiento, que puede ser tan simple como determinar el color del coche o mucho más complejos.

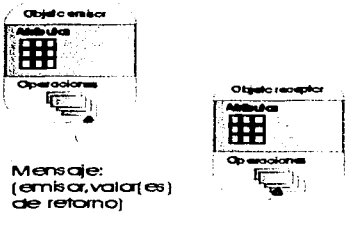
1.3.4 Mensajes

Los mensajes son el medio a través del cual los objetos interactúan. Un mensaje estimula la ocurrencia de cierto comportamiento en el objeto receptor. El comportamiento se realiza cuando se ejecuta una operación. La interacción entre objetos (se ilustra esquemáticamente en la Figura 1.8). Una operación dentro de un objeto emisor genera un mensaje de la forma:

mensaje: [destino, operación , parámetros]

Donde *destino* define el objeto receptor el cual es estimulado por el mensaje, *operación* se refiere al método que recibe el mensaje y *parámetros* proporciona información requerida para el éxito de la operación.

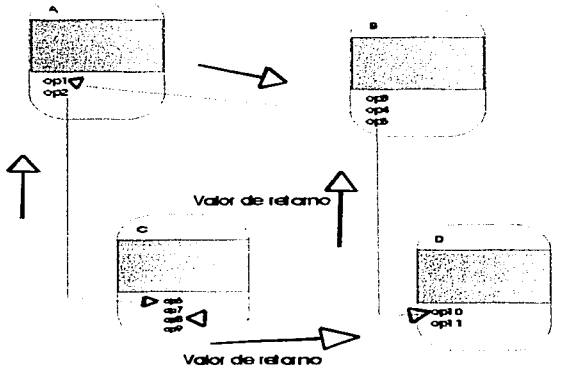
Figura 1.8 Paso de mensajes entre objetos



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

Como un ejemplo de paso de mensajes dentro de un sistema OO, considere los objetos que se muestran (figura 1.9) .

Figura 1.9 Paso de mensajes



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

Los cuatro objetos *A,B,C,D* se comunican unos con otros a través del paso de mensajes. Si el objeto *B* requiere el proceso asociado con la operación *op10* del objeto *D*, el primero enviaría a *D* un mensaje que contendría:

mensaje: [D, op10 , <datos>]

Como parte de la ejecución de *op10*, el objeto *D* puede enviar un mensaje al objeto *C* de la forma:

mensaje: [C, op08, <datos>]

C encuentra *op08*, al ejecutar, y entonces envía un valor de retorno apropiado a *D*. La operación *op10* completa su ejecución y envía un valor de retorno a *B*.

El paso de mensajes mantiene comunicados un sistema OO.

1.3.5 Encapsulamiento, herencia y polimorfismo

Existen tres conceptos importantes que diferencian el enfoque OO de la ingeniería de software convencional.

El encapsulamiento, como se mencionó anteriormente, empaqueta datos y las operaciones que manejan estos datos en un objeto simple con denominación y puede reutilizarse como una especificación o componente de programa.

Esto proporciona un gran número de beneficios:

- ✓ Los detalles de implementación interna de datos y procedimientos están ocultos al mundo exterior (ocultamiento de información).
- ✓ Las estructuras de datos y las operaciones se encuentran dentro de la clase. Esto facilita la reutilización de componentes.

- ✓ Las interfaces entre objetos encapsulados están simplificadas. Un objeto que envía un mensaje no se tiene que preocupar por los detalles de estructuras de datos internas en el objeto receptor. Por tanto, se simplifica la interacción y el acoplamiento del sistema tiende a reducirse.

La herencia es una de las diferencias clave entre sistemas convencionales y sistemas OO. El concepto de herencia se refiere a la compartición de atributos y operaciones basada en una relación jerárquica entre varias clases. Una clase puede definirse de forma general y luego refinarse en sucesivas subclases. Cada clase hereda todas las propiedades (atributos y operaciones) de su superclase y añade sus propiedades particulares.

La posibilidad de agrupar las propiedades comunes de una serie de clases en una superclase y heredar estas propiedades en cada una de las subclases es lo que permite reducir la repetición de código en el paradigma OO y es una de sus principales ventajas.

Por ejemplo, una subclase *Y* hereda todos los atributos y operaciones asociadas con sus superclase *X*. Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e implementados para *X* están inmediatamente disponibles para *Y* (no incluye trabajo extra). La reutilización se realiza directamente.

Cualquier cambio en los datos u operaciones contenido dentro de una superclase se hereda inmediatamente por todas las subclases que se derivan de la superclase. Debido a esto, la jerarquía de clases se convierte en un mecanismo a través del cual los cambios (a altos niveles) pueden propagarse inmediatamente a través de todo el sistema.

Al crear una nueva clase debe tomarse en cuenta varias opciones:

- ➢ La clase puede diseñarse y construirse de la nada. No se usa la herencia.
- ➢ La jerarquía de clases puede ser rastreada para determinar si una clase ascendiente[5] contiene la mayoría de los atributos y operaciones requeridas.

[5] Los términos *descendientes* y *ascendientes* son usados a veces para reemplazar a *subclase* y *superclase*, respectivamente.

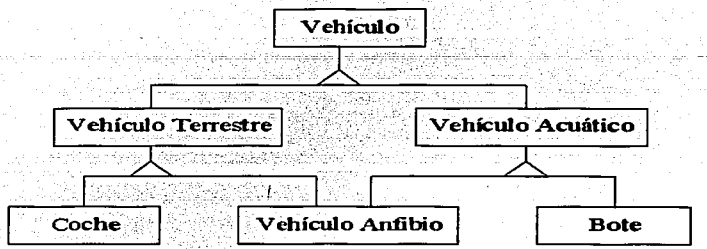
- La nueva clase hereda de su clase ascendiente, y puede añadirse a los elementos requeridos.
- La jerarquía de clases puede reestructurarse de tal manera que los atributos y operaciones requeridos puedan heredarse por la nueva clase.
- Las características de una clase existente pueden sobrescribirse y se pueden implementar versiones privadas de atributos u operaciones para la nueva clase.

Es importante notar que la reestructuración puede ser difícil; por ello se usa a veces la anulación. La anulación ocurre cuando los atributos y operaciones se heredan de manera normal, pero después son notificados según las necesidades específicas de la nueva clase "la herencia no es transitiva".

La ***Herencia Múltiple*** hereda algunos atributos y operaciones de una clase y otros de otra clase, permite a una clase tener más de una superclase, y así heredar las características de todos sus padres. Esto complica las jerarquías de herencia, que dejan de ser árboles para convertirse en grafos. La gran ventaja de la herencia múltiple es el incremento en las posibilidades de reutilización. El inconveniente es la pérdida de simplicidad conceptual y de implementación.

La herencia múltiple presenta dos problemas: conflictos y herencia repetida. Cuando una clase hereda simultáneamente sus propiedades de varias clases, pueden existir en estas propiedades representadas por el mismo identificador con significados diferentes; a esto se le denomina *conflicto*. Los conflictos que provoca la herencia múltiple se agravan en presencia de situaciones en las que una clase es heredada por otra por medio de caminos distintos; es lo que se denomina ***herencia repetida*** (figura 1.10*)*.

Figura 1.10 Representación de una herencia repetida



Fuente: www.terra.com

Existen muchas propuestas para resolver este tipo de problemas. Las ventajas e inconvenientes de las posibles alternativas de la herencia múltiple se han estudiado con cierta profundidad, llegando a la conclusión de que cada alternativa tiene razones a favor y en contra, lo que impide tomar decisiones de diseño que satisfagan todo tipo de exigencias. En la mayoría de los casos, la interpretación que se da a la herencia múltiple depende más de su adecuación a la implementación que a otros motivos.

El ***polimorfismo*** permite que la cantidad de operaciones diferentes posean el mismo nombre, reduciendo la cantidad de líneas de código necesarias para implementar un sistema y facilita los cambios en caso que se produzcan.

Por ejemplo: Si se necesitará dibujar cuatro tipos de gráficos: gráficos de línea, gráficos de curva, gráficos de rectángulo, gráficos de puntos. Idealmente, una vez que se han recogido los datos necesarios para un tipo particular de gráfico, el gráficos debe dibujarse él mismo. En una aplicación normal se haría esto: desarrollar módulos de dibujo para cada tipo de gráfico.

Case of tipo_grafico:

If tipo_grafico=grafico_linea **then** DibujarLinea(datos);

If tipo_grafico=grafico_curva **then** DibujarLinea(datos);

If tipo_grafico=grafico_rectangulo **then** DibujarLinea(datos);

If tipo_grafico=grafico_puntos **then** DibujarLinea(datos);

Para resolver esto en un sistema OO, cada uno de los gráficos mencionados arriba se convierte en una subclase de una clase general llamada *grafico*. La subclase define una operación llamada *dibujar*. Un objeto puede enviar un mensaje dibujar a cualquiera de los objetos instanciados de cualquiera de las subclases. El objeto que recibe el mensaje invocará su propia operación *dibujar* para crear el gráfico apropiado reduciéndose a: *tipo_grafico dibujar*.

Cuando se desea añadir un nuevo tipo de gráfico al sistema, se crea una subclase con su propia operación *dibujar*. Pero no se requieren cambios dentro de un objeto que quiere dibujar un gráfico pues el mensaje *tipo_gráfico dibujar* permanece sin cambiar.

1.4 Identificación de los elementos de un modelo de objetos

Los elementos de un modelo de objetos son: ***clases y objetos,* atributos, operaciones y mensajes**. A continuación se presenta una serie de criterios informales que nos ayudaran en la identificación de los elementos de un modelo de objetos.

1.4.1 Identificación de objetos

Los objetos se determinan subrayando cada nombre y cláusula nominal e introduciéndola en una tabla simple. Los sinónimos deben destacarse. Si se requiere que el objeto implemente una solución, entonces éste formará parte del espacio de solución; pero si se necesita solamente para describir una solución esta forma parte del espacio del problema (los objetos se manifiestan de alguna de las formas mostradas en la figura 1.11).

Los objetos pueden ser:

a) ***Entidades externas*** (dispositivos, personas otros sistemas) que producen o consumen información a usar por un sistema.

b) ***Cosas*** (informes, presentaciones, cartas, señales) que son parte del dominio de información del problema.

c) ***Ocurrencias o eventos*** (por ejemplo, transferencia de propiedad o la terminación de una serie de movimientos) que ocurren dentro del contexto de operación del sistema.

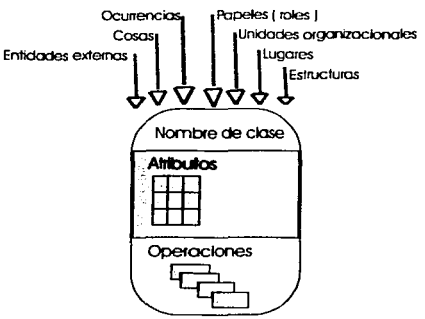
d) ***Papeles o roles*** (por ejemplo, director e ingeniero vendedor) desempeñados por sistemas que interactúan en el sistema.

e) ***Unidades organizacionales*** (división, grupo, equipo) que son relevantes en una aplicación.

f) ***Lugares*** (por ejemplo, planta de producción o muelle de carga) que establece el contexto del problema y la función general del sistema.

g) ***Estructuras*** (por ejemplo, sensores, vehículos de cuatro ruedas o computadoras) que definen una clase de objetos o clase relacionadas de objetos.

Figura 1.11 Representación de alguna de las formas en que se manifiestan los objetos



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

1.4.2 Especificación de atributos

Los atributos describen un objeto que ha sido seleccionado para ser incluido en el modelo de análisis. En esencia, son los atributos que definen al objeto, que clarifican lo que se representa con el objeto en el contexto del espacio del problema. Para desarrollar un conjunto significativo de atributos para un objeto, el analista debe estudiar la narrativa de proceso (o descripción del ámbito del alcance) para el problema y seleccionar aquellos elementos que pertenecen al objeto.

1.4.3 Definición de operaciones

Las operaciones definen el comportamiento de un objeto y cambian, de alguna manera, los atributos de dicho objeto. Concretamente, una operación cambia valores de uno o más atributos contenidos en el objeto.

Tipos de operaciones:

- Operaciones que manipulan de alguna manera, datos (añadiendo, eliminando, formateando, seleccionándolo).
- Operaciones que realizan algún cálculo.
- Operaciones que monitorizan un objeto frente a la ocurrencia de un suceso de control.

1.4.4 Fin de la definición del objeto

La historia de la vida de un objeto puede definirse reconociendo que dicho objeto puede ser creado, manipulado, modificado o leído de manera diferente, y posiblemente borrado. Después de haber definido los atributos y operaciones para todos los objetos especificados, se habrán creado los inicios del modelo **análisis orientado a objetos (AOO)**.

1.5 Gestión de proyectos de software orientado a objetos

La gestión de proyectos puede dividirse en las siguientes actividades:

1. Establecimiento de un marco de proceso común para el proyecto.
2. Uso del marco y de métricas para desarrollar estimaciones de esfuerzo y tiempo.
3. Especificación de productos de trabajo y avances que permitirán la medición del progreso.
4. Definición de puntos de comprobación para asegurar la calidad y el control.
5. Gestión de cambios al progresar el proyecto.
6. Seguimiento monitorización y control del progreso.

En el desarrollo de un proyecto de software OO se aplican estas seis actividades. Cabe destacar que cada uno tiene un matiz diferente y debe ser enfocado usando el modelo propio. A continuación se describen de una manera más detallada.

1.5.1 El marco de proceso común para OO

Un **marco de proceso común (MPC)** define un enfoque organizado para el desarrollo y mantenimiento de software. El MPC siempre es adaptable de tal manera que siempre cumpla con las necesidades individuales del equipo del proyecto. Esta es su característica más importante.

Un MPC no es un modelo lineal secuencial,[6] ya que por su naturaleza, la ingeniería de software orientada a objetos debe aplicar un modelo que complete el desarrollo iterativo (evoluciona a través de un número de ciclos).

Debido a esto se sugiere un modelo recursivo/paralelo que funciona de la siguiente manera:

[6] Los *modelos lineales secuenciales*, más conocidos como modelos de ciclo de vida o de cascada, suponen que los requisitos están definidos de principio a fin del proyecto y que las actividades de ingeniería progresan de una manera lineal secuencial.

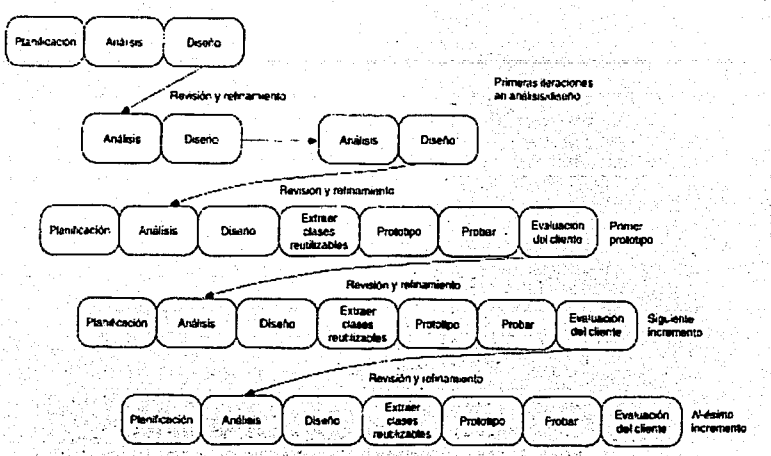
- Realizar los análisis suficientes para aislar las clases de problemas y las conexiones más importantes.
- Realizar un pequeño diseño para determinar si las clases y conexiones pueden ser implementadas de manera práctica.
- Extraer objetos reutilizables para construir un prototipo previo.
- Conducir algunas pruebas para descubrir errores en el prototipo.
- Obtener realimentación del cliente sobre el prototipo.
- Modificar el modelo de análisis basándose en el prototipo.
- Refinar el diseño prototipo.
- Construir objetos especiales (no disponible en la biblioteca).
- Ensamblar un nuevo prototipo usando objetos de la biblioteca y los objetos que se crearon nuevos.
- Realizar pruebas para descubrir errores sobre el prototipo.
- Obtener realimentación del cliente sobre el prototipo.

La diferencia entre el modelo recursivo/paralelo y modelo de proceso OO es el reconocimiento de que:

1) Modelo de análisis y diseño para sistemas OO no puede reutilizarse a un nivel uniforme.
2) El análisis y diseño pueden aplicarse a componentes independientes del sistema de manera concurrente.

Cada iteración del proceso recursivo/paralelo requiere planificación, ingeniería (análisis, diseño, extracción de clases, prototipo y pruebas) y actividades de evaluación (figura 1.12).

Figura 1.12 Secuencia típica de un proceso para un proyecto OO



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

Durante la planificación las actividades asociadas con cada una de las componentes independientes del programa son incluidas en la misma (con cada iteración se ajusta la agenda para acomodar los cambios asociados con la iteración precedente). Durante las primeras etapas del proceso de ingeniería, el análisis y el diseño ocurren iterativamente. Se producen versiones incrementales del software a las cuales se revisan y evalúan por el cliente produciendo una realimentación que afecta a la siguiente actividad de planificación y el subsiguiente incremento.

1.5.2 Estimación en proyectos orientados a objetos

Las técnicas de estimación en proyectos de software convencionales requieren estimados de líneas de código (LDC) o puntos de función (PF) como controlador principal de estimación. Las

estimaciones LDC tienen poco sentido en proyectos OO debido a que el objetivo es la reutilización. Las estimaciones a partir de PF es más efectiva ya que se puede determinar a partir del planteamiento del problema. Puede aportar valores para estimaciones pero no provee ajustes de planificación y esfuerzo a realizar, los cuales se requieren cuando iteramos a través del paradigma recursivo/paralelo.

1.5.3 Un enfoque OO para estimaciones y planificaciones

La estimación debe derivarse usando diferentes técnicas, respecto al esfuerzo y la duración usadas en el desarrollo de software.

La estimación de costos para software convencional debe sustituirse por un enfoque diseñado específicamente para software OO. Lorenz y Kidd sugieren el siguiente enfoque:

a. Desarrollo de estimaciones usando la descomposición de esfuerzos, análisis de PF, y cualquier otro método aplicable a aplicaciones convencionales.
b. Reconocer en número de códigos que pueden incrementar con el progreso del proyecto.
c. Clasificar el tipo de interfaz (Interfaz no gráfica "No IGU", interfaz de usuario basada en texto, interfaz gráfica de usuario "IGU", interfaz grafica de usuario compleja) para la aplicación y desarrollar un multiplicador (tiempo para las clases de soporte).
d. Multiplicar la cantidad total de clases por el número de unidades de trabajo por clase (15 y 20 días por persona por clase).

Capítulo 2

Análisis orientado a objetos (AOO)

2.1 Análisis orientado a objetos

El análisis se refiere a entender de manera detallada el problema que tiene que resolver para poder implementar un sistema o software. Ya sea que emplee modelos o funciones que permitan tener comunicación entre el hombre y la máquina.

El método de análisis tiene un conjunto de principios y normas a seguir para el desarrollo de un sistema de software:

1. Debe entenderse el problema a solucionar.
2. Deben definirse las funciones que debe realizar el software a desarrollar.
3. Debe representarse el comportamiento que tendrá el software a eventos externos.
4. Deben dividirse los modelos que representen información o sus funciones de manera jerárquica.

El **Análisis Orientado a Objetos (AOO)** se basa en conceptos sencillos, conocidos desde la infancia y que aplicamos continuamente: objetos y atributos, el todo y las partes, clases y miembros.

Ventajas del análisis orientado a objetos:

- ***Dominio del problema***. Representa el sistema en términos del mundo real, en vez de términos informáticos.
- ***Comunicación***. El concepto OO es más simple y está menos relacionado con la informática que el concepto de flujo de datos. Esto permite una mejor comunicación entre el analista y el experto en el dominio del problema (es decir, el cliente).
- ***Consistencia***. Reduce la distancia entre el modelo de procesos y el de datos.
- ***Expresión de características comunes***. Evita la duplicación de características comunes a varios elementos.

- ***Resistencia al cambio***. Al ser más próximo al sistema real es más estable frente a cambios en los requisitos.
- ***Reutilización.*** Favorece la reutilización, tanto interna como externa.

2.1.1 Enfoques convencionales y enfoques OO

El análisis estructurado toma un visión diferente. Los datos se consideran separadamente de los procesos que los transforman. El comportamiento del sistema, aunque importante, tiende a jugar un papel secundario en el análisis estructurado.

El AOO ofrece un enfoque nuevo para el análisis de requisitos de sistemas software. En lugar de considerar el software desde una perspectiva clásica de entrada/proceso/salida, como los métodos clásicos, **se basa en modelar el sistema mediante los objetos que forman parte de él y las relaciones estáticas (herencia y composición) o dinámicas (uso) entre estos objetos**. Este enfoque pretende conseguir modelos que se ajusten mejor al problema real, a partir del conocimiento del llamado ***dominio del problema***. Desde este punto de vista, el AOO modela los sistemas desde un punto más próximo a su implementación en un ordenador (entrada/proceso/salida).

Este intento de conocer el dominio del problema ha sido siempre importante. Por ejemplo, no tiene sentido empezar a escribir los requisitos funcionales de un sistema de control de tráfico aéreo, y menos aún diseñarlo o programarlo sin estudiar primero qué es el tráfico aéreo o qué se espera de un sistema de control de este tipo. La ***ventaja*** del AOO es que se basa en la utilización de objetos como abstracciones del mundo real. Esto nos permite centrarnos en los aspectos significativos del dominio del problema (en las caracteristicas de los objetos y las relaciones que se establecen entre ellos) y este conocimiento se convierte en la parte fundamental del análisis del sistema software, que será luego utilizado en el diseño y la implementación.

En el AOO, los objetos encapsulan tanto atributos como procedimientos (operaciones que se realizan sobre los objetos), e incorpora además conceptos como el polimorfismo o la herencia que facilitan la reutilización de código.

El uso de AOO puede **facilitar mucho la creación de prototipos**, y las técnicas de desarrollo evolutivo de software. Los objetos son inherentemente[7] reutilizables, y se puede crear un catálogo de objetos que podemos usar en sucesivas aplicaciones. De esta forma, podemos obtener rápidamente un prototipo del sistema, que pueda ser evaluado por el cliente, a partir de objetos analizados, diseñados e implementados en aplicaciones anteriores. Y lo que es más importante, dada la facilidad de reutilización de estos objetos, el prototipo puede ir evolucionando hacia convertirse en el sistema final, según vamos refinando los objetos de acuerdo a un proceso de especificación incremental.

Características del enfoque orientado a objetos:

Identidad

- Los elementos del dominio del problema se organizan en entidades discretas y distinguibles llamadas *objetos.*
- Los objetos encapsulan *atributos* (datos) y *operaciones.*
- Cada objeto tiene *identidad propia*, aunque los valores de sus atributos coincidan con los de otro.

Clasificación

- Los objetos con propiedades comunes se agrupan en *clases.*
- Las clases son *abstracciones* de características comunes a una serie de objetos.
- Las clases son *arbitrarias*: dependen del dominio del problema.
- Cada uno de los objetos agrupados en una clase se llama *instancia.*
- Las instancias de una clase comparten atributos, operaciones y comportamiento pero tienen valores distintos en los atributos.

Polimorfismo

- Una misma operación puede realizarse de formas distintas en clases distintas. La *semántica* es común pero la *implementación* varía en cada clase.
- La implementación de una operación en una clase se denomina *método.*

[7] *Inherente* significa que esta unido por naturaleza

Herencia

- Es una *relación jerárquica entre clases* con características comunes.
- La *clase padre* define características comunes a todas las hijas.
- Cada *clase hija* hereda las características del padre y las amplia o refina.

2.1.2 El panorama AOO

El paradigma "orientado a objetos" ha ido madurando como un enfoque de desarrollo de software alternativo a la programación estructurada o modular. Se empezaron a crear diseños de aplicaciones de todo tipo usando una forma de pensar orientada a los objetos, y a implementar estos diseños utilizando lenguajes orientados a objetos. Sin embargo, el análisis de requisitos se quedó atrás. No se desarrollaron técnicas de análisis específicamente orientadas a objetos.

Esta situación ha ido cambiando poco a poco, a medida que se desarrollaban técnicas de análisis específicas para desarrollar software orientado a objetos, e incluso como complemento de otros métodos de análisis. Ejemplos de estas nuevas técnicas son los métodos de Coad/Yourdon, Jacobson, Booch y Rumbaugh (**OMT** Object Modelling Techniques).[8]

2.1.2.1 Técnicas de análisis para el desarrollo de software orientado a objetos

1) Técnica del método de Booch

El método de Booch, abarca un «micro proceso de desarrollo» y un «macro proceso de desarrollo». El nivel micro define un conjunto de tareas de análisis que se reaplican en cada etapa en el macro proceso. Por esto se mantienen un enfoque evolutivo. El método Booch está soportado por una gran variedad de herramientas automatizadas.

[8] Veáse en S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

A continuación se especifica el microproceso de desarrollo de Booch:

- Identificar clases y objetos.
  - Proponer objetos candidatos.
  - Conducir el análisis de comportamiento.
  - Identificar escenarios relevantes.
  - Definir atributos y operaciones para cada clase.
- Identificar la semántica de clases y objetos.
  - Seleccionar y analizar escenarios.
  - Asignar responsabilidades para alcanzar el comportamiento deseado.
  - Dividir las responsabilidades para equilibrar el comportamiento.
  - Seleccionar un objeto y enumerar sus papeles y responsabilidades.
  - Definir operaciones para satisfacer las reponsabilidades.
  - Buscar colaboraciones entre objetos.
- Identificar relaciones entre clases y objetos.
  - Definir las dependencias que existen entre objetos.
  - Describir el papel (rol) de cada objeto participante.
  - Validar los escenarios por revisión completa.
- Realizar una serie de refinamientos.
  - Producir diagramas apropiados para el trabajo realizado en los puntos anteriores.
  - Definir jerarquías de clases apropiadas.
  - Crear agrupamientos basados en clases comunes.
- Implementar clases y objetos (complementar el modelo de análisis).

2) El método de Coad y Yourdon

El método de Coad y Yourdon se considera, con frecuencia, como uno de los métodos del A00 más sencillos de aprender. La notación del modelado es relativamente simple.

A continuación sigue una descripción resumida del proceso de A00 de Coad y Yourdon:

- Identificar objetos usando el criterio de «qué buscar».
- Definir una estructura de generalización-especificación (sección 2.3.3).
- Definir una estructura de todo-parte (sección 2.3.3).
- Identificar temas (representaciones de componentes de subsistemas).
- Definir atributos.
- Definir servicios.

3) Técnica del método de Jacobson

También llamado ISOO (ingeniería del software orientada a objetos), el método de Jacobson es una versión simplificada de Objectory, un método patentado, también desarrollado por Jacobson. Este método se diferencia de los otros por la importancia que da al *caso de uso*, una descripción o escenario que describe cómo el usuario interactúa con el producto o sistema.

A continuación sigue el proceso de A00 de Jacobson:

- Identificar los usuarios del sistema y sus responsabilidades globales.
- Construir un modelo de requisitos.
  - Definir los actores y sus responsabilidades.
  - Identificar los casos de uso para cada actor.
  - Preparar una visión inicial de los objetos del sistema y sus relaciones.
  - Revisar el modelo usando los casos de uso como escenarios para determinar su validez.
- Construir un modelo de análisis.
  - Identificar objetos de interfaz usando información del tipo actor-interacción.
  - Crear vistas estructurales de los objetos de interfaz.
  - Representar el comportamiento del objeto.

Aislar subsistemas y modelos para cada uno.

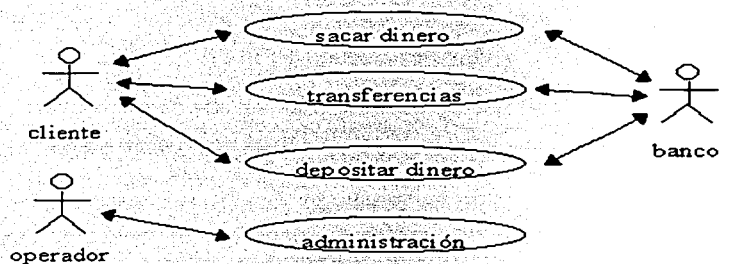
Revisar el modelo usando casos de uso con escenarios para determinar su validez.

4) Casos de uso de Jacobson

Una forma de describir los requisitos iniciales del usuario, durante la fase de conceptualización, es construir casos de uso del sistema, descritos inicialmente por Jacobson en 1987 y actualmente incorporados a la mayor parte de las metodologías de AOO.

Un **caso de uso** está formado por una serie de interacciones entre el sistema y un *actor* (una entidad externa, ejerciendo un rol determinado), que muestran una determinada forma de utilizar el sistema (Figura 2.1). Cada interacción comienza con un evento inicial que el actor envía al sistema y continua con una serie de eventos entre el actor, el sistema y posiblemente otros actores involucrados.

Figura 2.1 Ejemplo de un caso de uso que se desarrolla en un banco cualquiera



Fuente: www.google.com

Un caso de uso puede ser descrito en lenguaje natural, mediante trazos de eventos o mediante diagramas de interacción de objetos.

5) Técnica de modelado de objetos (OMT) de Rumbaugh

La esencia del desarrollo de software OO es la identificación y organización de conceptos del dominio del problema, más que en su implementación final usando un determinado lenguaje.

La Técnica de Modelado de Objetos (OMT, Rumbaugh, 1991) es un procedimiento que se basa en aplicar el enfoque orientado a objetos a todo el proceso de desarrollo de un sistema software, desde el análisis hasta la implementación. Los métodos de análisis y diseño que propone son independientes del lenguaje de programación que se emplee para la implementación. Incluso esta implementación no tiene que basarse necesariamente en un lenguaje OO.

Al igual que los métodos estructurados, OMT utiliza tres tipos de modelos para describir un sistema:

- ***Modelo de objetos***. Describe la estructura estática de los objetos de un sistema y sus relaciones. El modelo de objetos (veáse figura 1.6 del capitulo 1) contiene diagramas de objetos, clases, jerarquías, y relaciones .

  Elementos del modelo de objetos:

  Instancias. Cada uno de los objetos individuales.

  Clases. Abstracción de objetos con propiedades comunes.

  Atributos. Datos que caracterizan las instancias de una clase.

  Operaciones. Funciones que pueden realizar las instancias.

  Relaciones. Se establecen entre clases.

  Asociación. Relación de uso en general.

  Multiplicidad. Número de instancias que intervienen en la relación.

  Atributos. Algunos atributos pueden depender de la asociación.

  Calificación. Limita la multiplicidad de las asociaciones.

  Roles. Indican los papeles de las clases en las relaciones.

Restricciones y ordenación. Relaciones funcionales entre entidades de un modelo de objetos de manera ordenada

Composición. Relaciones todo/parte.

Generalización. Relaciones padre/hijo.

Redefinición. Modificación de las propiedades heredadas.

Enlaces. Instancias de una relación. Relaciona instancias.

- ***Modelo dinámico.*** El modelo dinámico describe las características de un sistema que cambia a lo largo del tiempo. Se utiliza para especificar los aspectos de control de un sistema. Por ejemplo tenemos un sistema ya terminado y tenemos que desarrollar otro con características similares, entonces tomo lo que me sirve del otro programa y lo implemento en este y conforme se haya avanzado lo puedo estar modificando de acuerdo a las necesidades del cliente.
- ***Modelo funcional.*** Describe las transformaciones de datos del sistema, quiere decir los cambios que se realizan en el sistema conforme a las necesidades del cliente o a la función que tenga que realizar. Muy similar al ejemplo anterior pero de manera más detallada.

Los tres modelos son vistas ortogonales (independientes) del mismo sistema, aunque existen relaciones entre ellos. Cada modelo contiene referencias a elementos de los otros dos. Por ejemplo, las operaciones que se asocian a los objetos del modelo de objetos figuran también, de forma más detallada en el modelo funcional. El más importante de los tres es el modelo de objetos, porque es necesario describir qué cambia antes que decir cuándo o cómo cambia.

OMT es una metodología OO de desarrollo de software basada en una notación gráfica para representar conceptos OO. La metodología consiste en construir un modelo del dominio de aplicación e ir añadiendo detalles a este modelo durante la fase de diseño.

OMT consta de las siguientes fases o etapas:

Fases:

- *Conceptualización.* Consiste en la primera aproximación al problema que se debe resolver. Se realiza una lista inicial de requisitos y se describen los casos de uso.
- *Análisis.* El analista construye un modelo del dominio del problema, mostrando sus propiedades más importantes. Los elementos del modelo deben ser conceptos del dominio de aplicación y no conceptos informáticos tales como estructuras de datos. Un buen modelo debe poder ser entendido y criticado por expertos en el dominio del problema que no tengan conocimientos informáticos.
- *Diseño del sistema.* El diseñador del sistema toma decisiones de alto nivel sobre la arquitectura del mismo. Durante esta fase el sistema se organiza en subsistemas basándose tanto en la estructura del análisis como en la arquitectura propuesta.
- *Diseño de objetos.* El diseñador de objetos construye un modelo de diseño basándose en el modelo de análisis, pero incorporando detalles de implementación. El diseño de objetos se centra en las estructuras de datos y algoritmos[9] que son necesarios para implementar cada clase. OMT describe la forma en que el diseño puede ser implementado en distintos lenguajes (orientados y no orientados a objetos, bases de datos, etc.).
- *Implementación.* Las clases de objetos y relaciones desarrolladas durante el análisis de objetos se traducen finalmente a una implementación concreta. Durante la fase de implementación es importante tener en cuenta los principios de la ingeniería del software de forma que la correspondencia con el diseño sea directa y el sistema implementado sea flexible y extensible. No tiene sentido que utilicemos AOO y el diseño orientado a objetos **DOO** (detallado en el capítulo 3) de forma que potenciemos la reutilización de código y la correspondencia entre el dominio del problema y el sistema informático, si luego perdemos todas estas ventajas con una implementación de mala calidad.

Algunas clases que aparecen en el sistema final no son parte del análisis sino que se introducen durante el diseño o la implementación. Este es el caso de estructuras como árboles o

[9] *Algoritmo*, conjunto de pasos ordenados para resolver un problema, tal como una fórmula matemática o las instrucciones de un programa.

listas enlazadas, que no suelen estar presentes en el dominio de aplicación. Estas clases se añaden para permitir utilizar determinados algoritmos.

Los conceptos del paradigma OO pueden aplicarse durante todo el ciclo de desarrollo del software, desde el análisis a la implementación sin cambios de notación, sólo añadiendo progresivamente detalles al modelo inicial.

A continuación el proceso de Rumbaugh resumido:

- Desarrollar una declaración del ámbito del problema.
- Desarrollar un modelo de objetos.
  - Identificar clases relevantes al problema.
  - Definir atributos y asociaciones.
  - Definir enlaces de objetos.
  - Organizar las clases de objetos usando la herencia.
- Desarrollar un modelo dinámico.
  - Preparar escenarios.
  - Definir eventos y desarrollar una traza de eventos para cada escenario.
  - Construir un diagrama de flujo de eventos.
  - Desarrollar un diagrama de estados.
  - Revisar el comportamiento para comprobar consistencia y complejidad.
- Desarrollar un modelo funcional para el sistema.
  - Identificar entradas y salidas.
  - Usar diagramas de flujo de datos para representar transformaciones del flujo.
  - Desarrollar EP (Especificación del proceso) para cada función.
  - Especificar criterios de restricciones y optimización.

6) El método de Wirfs-Brock

El método de Wirfs-Brock no hace una distinción clara entre las tareas de análisis y diseño. En su lugar, se propone un proceso continuo que comienza con la valoración de una especificación del cliente y termina con el diseño.

A continuación el análisis de Wirfs-Brock.

- Evaluar la especificación del cliente.
- Usar un análisis gramatical para extraer clases candidatas de la especificación.
- Agrupar las clases en un intento de determinar superclases.
- Definir responsabilidades para cada clase.
- Asignar responsabilidades a cada clase.
- Identificar relaciones entre clases.
- Definir colaboraciones entre clases basándose en sus responsabilidades.
- Construir representaciones jerárquicas de clases para mostrar relaciones de herencia.
- Construir un grafo de colaboraciones para el sistema.

El análisis en sistemas orientados a objetos puede ocurrir a muchos niveles diferentes de abstracción. Al nivel de negocios o empresas, las técnicas asociadas con el AOO pueden acoplarse en un esfuerzo por definir clases, objetos, relaciones y comportamientos que modelen el negocio por completo. En el nivel de un área específica de negocios (o una categoría de productos o sistemas). Al nivel de las aplicaciones, el modelo de objetos se centra en los requisitos específicos del cliente pues estos afectan la aplicación que se va a construir.

El AOO a su nivel de abstracción más alto es el nivel de empresa, al nivel más bajo el AOO cae dentro del alcance general de la ingeniería del software orientado a objetos. Esta actividad, llamada ***análisis del dominio***, tiene lugar cuando una organización desea crear una biblioteca de clases reutilizables o reusables (componentes) ampliamente aplicables a una categoría completa de aplicaciones.

El **proceso de análisis del dominio** consta de:

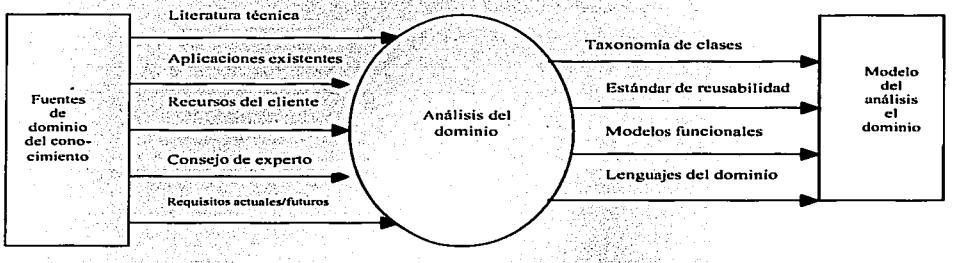
El ***análisis del dominio del software*** es la identificación, análisis y especificación de requisitos comunes de un dominio de aplicación específico, normalmente para su reusabilidad en múltiples proyectos dentro del mismo dominio de aplicación.

El ***análisis orientado a objetos del dominio*** es la identificación, análisis y especificación de capacidades comunes y reusables dentro de un dominio de aplicación específico, en términos de objetos, clases, submontajes y marcos de trabajos comunes.

El **objetivo** del análisis del dominio es claro: encontrar o crear aquellas clases ampliamente aplicadas, de tal manera que sean reusables. Por ejemplo, el papel de un análisis del dominio es similar al de un maestro tornero dentro de un entorno de fabricación fuerte. El trabajo del maestro tornero es diseñar y construir herramientas que pueden usarse por varias personas que trabajan en aplicaciones similares, pero no necesariamente idénticas.

La Figura 2.2 ilustra las entradas y salidas clave para el proceso de análisis de dominio. El proceso de análisis de dominio puede caracterizarse por una serie de actividades que comienzan con la identificación del dominio a investigar y finaliza con una especificación de los objetos y clases que caracterizan este dominio:

Figura 2.2 Entrada y salida para análisis de dominio



Fuente: : S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

Definir el dominio a investigar. El analista debe primero aislar el área de negocio, tipo de sistema, o categoría del producto de interés. A continuación, se deben extraer los <<elementos>> OO y no OO. Los elementos OO incluyen especificaciones, diseños y código para clases de aplicaciones OO ya existentes. Los elementos no OO abarcan políticas, procedimientos, planes, estándares y guías.

Clasificar los elementos extraídos del dominio. Los elementos se organizan en categorías y se establecen las características generales que definen la categoría. Se establecen jerarquías de clasificación en caso de ser apropiado.

Recolectar una muestra representativa de aplicaciones en el dominio. Asegurar que la aplicación en cuestión tiene elementos que caen dentro de las categorías ya definidas.

Analizar cada aplicación dentro de la muestra. Las siguientes etapas ocurren durante el proceso de análisis de dominio:

- Identificar objetos candidatos reusables.
- Indicar las razones que hacen que el objeto haya sido identificado como reusable.
- Definir adaptaciones al objeto que también puede ser reusable.
- Estimar el porcentaje de aplicaciones en el dominio que pueden reutilizar el objeto.
- Identificar los objetos por nombre y usar técnicas de gestión de configuración para controlarlos.

El analista debe estimar qué porcentaje de una aplicación típica pudiera construirse usando los objetos reusables.

El análisis del dominio es la primera actividad técnica en una amplia disciplina que algunos llaman *ingeniería del dominio.* El objetivo es ser capaz de crear software dentro del dominio con un alto porcentaje de componentes reusables. Argumentos a favor de un esfuerzo dedicado a la ingeniería del dominio son: bajo costo, mayor calidad y menor tiempo de comercialización.

2.2 Componentes genéricos del modelo de análisis OO[10]

El análisis se ocupa de proyectar un modelo preciso, conciso, comprensible y correcto del mundo real. **El propósito de análisis orientado a objetos es modelar el mundo real de forma tal que sea comprensible.** Para ello se debe examinar los requisitos, analizar las implicaciones que se deriven de ellos y reafirmar de manera rigurosa.

Un conjunto de componentes de representación genéricos que aparecen en todos los modelos de análisis OO. Los *componentes estáticos* son estructurales por naturaleza, e indican características que se mantienen durante toda la vida operacional de una aplicación. Los *componentes dinámicos* se centran en el control, son sensibles al tiempo y al tratamiento de eventos. Ellos definen cómo interactúa un objeto con otros a lo largo del tiempo.

Los siguientes componentes pueden identificarse:

Vista estática de clases semánticas. Se imponen los requisitos y se extraen (y representan) clases como parte del modelo de análisis. Estas clases persisten a través de todo el período de vida de la aplicación y se derivan en base a la semántica de los requisitos del cliente.

Vista estática de los atributos. Toda clase debe describirse explícitamente. Los atributos asociados con la clase aportan una descripción de la clase, así como una indicación inicial de las operaciones relevantes a esta clase.

Vista estática de las relaciones. Los objetos están «conectados» unos a otros de varias formas. El modelo de análisis debe representar las relaciones de manera tal que puedan identificarse las operaciones (que afecten estas conexiones) y que pueda desarrollarse un buen diseño de intercambio de mensajes.

Vista estática de los comportamientos. Las relaciones indicadas anteriormente definen un conjunto de comportamientos que se adaptan al escenario utilizado (casos de uso) del sistema. Estos comportamientos se implementan a través de la definición de una secuencia de operaciones que los ejecutan.

---

[10] Véase en COAD, P. y E. Yourdon, ***Object-Oriented Analysis***, 2ª Edición , ed. Prentice-Hall, México, 1991

Vista dinámica de la comunicación. Los objetos deben comunicarse unos con otros y hacerlo basándose en una serie de mensajes que provoquen transiciones de un estado a otro del sistema.

Vista dinámica del control y manejo del tiempo. Debe describirse la naturaleza y tiempo de duración de los eventos que provocan transiciones de estados.

2.3 El proceso de AOO[11]

El proceso de A00 no comienza con una preocupación por los objetos. Más bien comienza con una comprensión de la manera en la que se usará el sistema: por las personas, si el sistema es de interacción con el hombre, por otras máquinas, si el sistema está envuelto en un control de procesos o por otros programas, si el sistema coordina y controla otras aplicaciones. Una vez que se ha definido el escenario, comienza el modelado del software.

2.3.1. Casos de uso o de utilización *(use cases)*

La recopilación de requisitos es siempre el primer paso en cualquier actividad de análisis del software. La recopilación de requisitos puede tomar la forma de un rápido encuentro en el cual el cliente y el desarrollador acuerdan definir los requisitos básicos del sistema y del software. Basado en estos requisitos, el ingeniero del software (analista) puede crear un conjunto de escenarios de manera tal que cada uno identifique una parte *(hilo)* del uso que se le dará al sistema a construir. Los escenarios, a menudo llamados *casos de uso*, aportan una descripción acerca de cómo el sistema será usado.

Para crear un caso de uso, el analista debe primero identificar los diferentes tipos de personas (o dispositivos) que usan el sistema o producto. Estos *actores* actualmente representan papeles ejecutados por personas (o dispositivos) cuando el sistema está en operación. Definido de

[11] Veáse en COAD, P. y E. Yourdon, ***Object-Oriented Analysis***, 2ª Edición , ed. Prentice-Hall, México, 1991.

una manera más formal un actor es cualquier cosa que se comunique con el sistema o producto y que sea externo a él.

Es importante observar que un actor y un usuario *no* son la misma cosa. Un usuario típico puede desempeñar un cierto número de papeles (roles) cuando usa el sistema, mientras que el actor representa una clase de entidades externas (a menudo, pero no siempre, las personas) que sólo desempeñan un único papel. Como ejemplo, consideremos un operador de máquina (un usuario) que interactúa con la computadora de control de una cadena de fabricación que contiene un cierto número de robots y máquinas de control numérico. Después de un cuidadoso estudio de los requisitos, el software para la computadora de control requiere cuatro modos (papeles) diferentes de interacción: modo de programación, modo de prueba, modo de monitorización y modo de tratamiento de errores. Por esto, pueden definirse cuatro actores: el programador, el verificador, el controlador y el reparador de errores. En algunos casos, el operador de la computadora puede realizar estas cuatro tareas. En otros, diferentes personas pueden representar los papeles de cada actor.

Como en otros aspectos del modelo de análisis OO, no se identifican todos los actores durante la primera iteración. Es posible identificar *actores primarios* durante la primera iteración y *actores secundarios* al aprender más sobre el sistema. Los actores primarios interactúan para lograr el funcionamiento requerido del sistema y obtener, de él, el beneficio propuesto. Ellos trabajan directa y frecuentemente con el software. Los actores secundarios existen para dar soporte al sistema de manera tal que los primarios puedan realizar su trabajo.

Una vez que los actores primarios han sido identificados, pueden desarrollarse casos de uso.[12]

En general, un caso de uso es simplemente la narración escrita que describe el papel de un actor al ocurrir la interacción con el sistema.

Los casos de uso describen escenarios que pueden percibirse de manera diferente por diferentes actores. Wyder sugiere el uso del despliegue de función de calidad (DFC) para desarrollar un valor de prioridad ponderado para cada caso de uso. Para lograr esto, se evalúan los casos de uso desde el punto de vista definido por todos los actores definidos para el sistema. Se asigna un

[12] *El caso de uso* describe la forma en la cual un actor interactúa con el sistema.

valor de prioridad para cada caso por cada actor.[13] Cuando se usa un modelo de proceso iterativo o incremental para la ingeniería del software orientado a objetos, las prioridades pueden influenciar determinando qué funcionalidad del sistema será la primera en liberarse.

2.3.2 Modelado de clases-responsabilidades-colaboraciones (CRC)

Una vez que se han desarrollado los escenarios de uso básicos para el sistema, es tiempo de identificar las clases candidatas, e indicar sus responsabilidades y colaboraciones. El modelado de clases-responsabilidades-colaboraciones (CRC) aporta un medio sencillo de identificar y organizar las clases que resulten relevantes al sistema o requisitos del producto. Ambler[14] describe el modelado CRC de la siguiente manera:

"Un modelo CRC es realmente una colección de tarjetas índice estándar que representan clases".

Clases

Los objetos se manifiestan en una variedad de formas: entidades externas, cosas, ocurrencias o eventos, roles, unidades organizacionales, lugares, o estructuras. Todos los nombres se transforman en objetos potenciales. Existen seis características de selección: información retenida, servicios necesarios, múltiples atributos, atributos comunes, operaciones comunes y requisitos esenciales.

Información retenida, el objeto potencial será de utilidad durante el análisis solamente si la información acerca de él debe recordarse para que el sistema funcione. Servicios necesarios, el objeto potencial debe poseer un conjunto de operaciones identificables que pueden cambiar de alguna manera el valor de sus atributos. Atributos múltiples, durante el análisis de requisitos se debe centrar la atención en la información principal(un objeto con un solo atributo puede ser útil durante el diseño, pero será mejor presentado como un atributo de otro objeto durante la actividad del análisis.

[13] Idealmente esta evaluación debe realizarse por individuos de la organización o función de negocio representada por un actor.

[14] AMBLER, S., ***Software Development: Using Use Clases***, julio 1995,pp.53-61

Atributos comunes, puede definirse un conjunto de atributos para el objeto potencial, los cuales son aplicables a todas las ocurrencias del objeto. Operaciones comunes, puede definirse como un conjunto de operaciones para el objeto potencial, las cuales son aplicable a todas las ocurrencias del objeto. Requisitos esenciales, son entidades externas que aparecen en el espacio del problema y producen o consumen información esencial para la producción de cualquier solución para el sistema, serán casi siempre definidas como objetos en el modelo de requisitos.

Un objeto potencial debe satisfacer estas seis características para poder ser considerado como posible miembro del modelo CRC.

Firesmith[15] extiende los tipos de clases:

Clases dispositivo. Modelan entidades externas tales como sensores, motores y teclados.

Clases propiedad. Representan alguna propiedad importante del entorno del problema (por ejemplo, establecimiento de créditos dentro del contexto de una aplicación de préstamos hipotecarios).

Clases interacción. Modelan interacciones que ocurren entre otros objetos (por ejemplo, una adquisición o una licencia).

Adicionalmente, los objetos y clases pueden clarificarse por un conjunto de características:

- ✓ Tangibilidad. Representa la clase algo tangible o palpable (por ejemplo, un teclado o sensor), o representa información más abstracta (por ejemplo, una salida prevista).
- ✓ Inclusividad. Es la clase atómica (es decir, no incluye otras clases) o es agregada (incluye al menos un objeto anidado).
- ✓ Secuenciabilidad. Es la clase concurrente (es decir posee su propio hilo de control) o secuencial (es controlada por recursos externos).
- ✓ Persistencia. Es la clase transitoria (es decir, es creada y eliminada durante la ejecución del programa), temporal (es creada durante la ejecución del programa y eliminada una vez que éste termina), o permanente (es almacenada en una base de datos).

[15] FIRESMITH, D. G., Object OrientedRequirements Analysis and Logical Design, Ed. Wiley, 1993.

✓ *Integridad*. Es la clase corrompible (es decir, no protege sus recursos de influencias externas) o es segura (la clase refuerza los controles de accesos de recursos).

Usando estas categorías de clases, las tarjetas índice creadas como parte del modelo CRC pueden extenderse para incluir el tipo de la clase y sus características (figura 2.3).

Figura 2.3 Un modelo CRC de tarjeta índice

Nombre de la clase:	
Tipo de la clase: (dispositivo, propiedad, rol, evento, ...)	
Características de la clase: (tangible, atómica, concurrente, ...)	
Responsabilidades:	Colaboradores:

Fuente: FIRESMITH, D. G., Object OrientedRequirements Analysis and Logical Design, Ed. Wiley, 1993.

Responsabilidades

Responsabilidades (atributos y operaciones), los atributos representan características estables de una clase (ver figura anterior), esto es, información sobre la clase que debe retenerse para llevar a cabo los objetivos del software especificados por el cliente.

Wirfs-Brock[16] y sus colegas sugieren cinco pautas para especificar responsabilidades para las clases:

1. *La inteligencia del sistema debe distribuirse de manera igualatoria.* Toda aplicación encierra un cierto grado de inteligencia. Esta inteligencia puede distribuirse entre las clases de varias maneras. Las clases <<tontas>> (aquellas con pocas responsabilidades) pueden modelarse de

[16] WIRFS-BROCK, R., B. Wilkerson y L. Weiner., Designing Object-Oriented Software, Ed. Prentice Hall, 1990

manera que actúen como sirvientes de unas pocas clases <<listas>> (aquellas con muchas responsabilidades).

Desventajas:

- Concentra toda la inteligencia en pocas clases, haciendo los cambios más difíciles.
- Tiende a necesitar más clases y por lo tanto el esfuerzo de desarrollo aumenta.

Por esta razón, la inteligencia del sistema debe distribuirse de manera igualatoria entre las clases de una aplicación.

Para determinar si la inteligencia del sistema está distribuida equitativamente, las responsabilidades definidas en cada tarjeta índice del modelo CRC deben ser evaluadas para determinar si cada clase posee una lista de responsabilidad es extraordinariamente grande. Esto indica una concentración de inteligencia.

2. *Cada responsabilidad debe establecerse lo más general posible.* Tanto los atributos como las operaciones deben residir en la parte alta de la jerarquía de clases (puesto que son genéricas, se aplicarán a todas las subclases).

3. *La información y el comportamiento asociado a ella, debe encontrarse dentro de la misma clase.* Los datos y procesos que manipulan estos datos deben empaquetarse como una unidad cohesionada.

4. *La información sobre un elemento debe estar localizada dentro de una clase, no distribuida a través de varias clases.* Una clase simple debe asumir la responsabilidad de almacenamiento y manipulación de un tipo específico de información. Esta responsabilidad no debe compartirse, de manera general, entre un número de clases. Si la información está distribuida, el software se torna más difícil de mantener y probar.

5. *Compartir responsabilidades entre clases relacionadas cuando es apropiado.* Existen muchos casos en los cuales una gran variedad de objetos exhibe el mismo comportamiento al mismo tiempo.

Colaboradores

Las clases cumplen con sus responsabilidades en una o dos maneras:

1) Una clase puede usar sus propias operaciones para manipular sus propios atributos, cumpliendo por lo tanto con una responsabilidad particular.
2) Una clase puede colaborar con otras clases.

Wirfs-Brock[17] y sus colegas definen las colaboraciones de la siguiente forma:

"Las colaboraciones representan solicitudes de un cliente a un servidor en el cumplimiento de una responsabilidad del cliente. Una colaboración es la realización de un contrato entre el cliente y el servidor. Decimos que un objeto colabora con otro, si para ejecutar una responsabilidad, necesita enviar cualquier mensaje al otro objeto. Una colaboración simple fluyen en una dirección, representando una solicitud del cliente al servidor. Desde el punto de vista del cliente, cada una de sus colaboraciones está asociada con una responsabilidad particular implementada por el servidor".

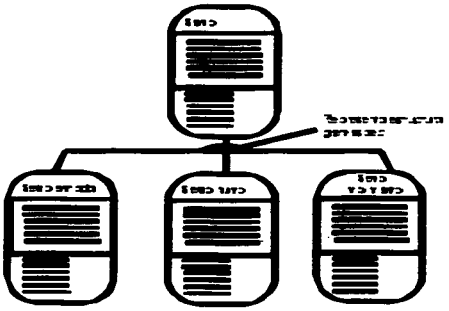
Las colaboraciones se identifican determinando si una clase puede satisfacer cada responsabilidad. Si no puede, entonces necesita interactuar con otra clase. Por consiguiente, una colaboración.

2.3.3 Definición de estructuras y jerarquías

Una vez que se han identificado las clases y objetos usando el modelo CRC, el analista comienza a centrarse en la estructura del modelo de clases y las jerarquías resultantes que surgen al emerger clases y subclases. Por ejemplo, el objeto sensor (figura 2.4), se refina en un conjunto de especializaciones: sensor de entrada, sensor de humo y sensor de movimiento. Se ha creado una jerarquía de clases simples.

[17] Veáse la nota anterior.

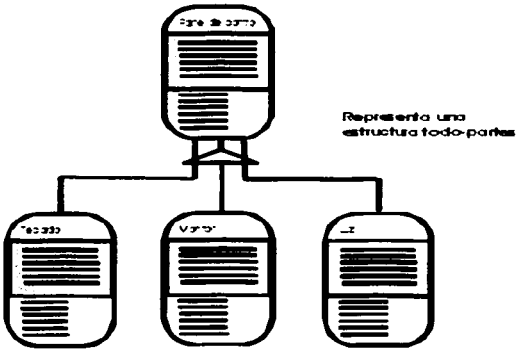
Figura 2.4 Ejemplo de un objeto sensor con la estructura gen-espec



Fuente: FIRESMITH, D. G., Object OrientedRequirements Analysis and Logical Design, Ed. Wiley, 1993.

En otros casos, un objeto representado según el modelo inicial puede estar compuesto de un número de partes las cuales pueden definirse a su vez como objetos. Estos objetos agregados pueden representarse como una estructura *todo-partes* y se definen usando la notación representado en la figura 2.5. El triángulo implica el sentido de una relación de ensamblaje.

Figura 2.5 Notación de la estructura todo-partes



Fuente: FIRESMITH, D. G., Object OrientedRequirements Analysis and Logical Design, Ed. Wiley, 1993.

Las representaciones estructurales proveen al analista de los medios para particionar el modelo CRC y para representar esta partición gráficamente.

2.3.4 Definición de temas y subsistemas

Un tema o subsistema puede tratarse como un conjunto de responsabilidades y que posee sus propios colaboradores (externos). Un subsistema implementa uno o más *contratos* con sus colaboradores externos. Un contrato es una lista específica de solicitudes que los colaboradores pueden hacer a un subsistema.[18]

Los temas son idénticos a los subsistemas en intención y contenido, pero se representan gráficamente. Por ejemplo, un panel de control (figura 2.5).

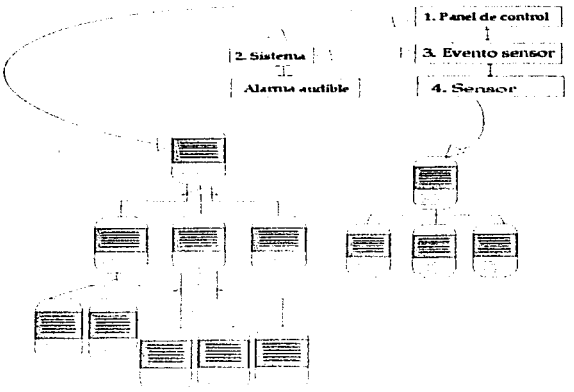
Definiendo una referencia de temas, como se muestra en la figura, pudiera referenciarse toda la estructura a través de un simple icono (el rectángulo).

Las referencias de temas se crean generalmente para cualquier estructura que posea más de cinco o seis objetos.

Al nivel más alto, se crearán también si estos objetos necesitan más de cinco o seis clasificaciones u objetos ensamblados (Figura 2.6).

[18] Las clases interactúan usando una filosofía *cliente/servidor*. En este caso el *subsistema* es el servidor y el colaborador externo de los clientes.

Figura 2.6 Un modelo de AOO con referencias de sujeto



Fuente: FIRESMITH, D. G., Object OrientedRequirements Analysis and Logical Design, Ed. Wiley, 1993.

Las flechas con dos puntas mostradas en la figura representan caminos de comunicación (mensajes) entre objetos contenidos dentro de las referencias de temas.

2.4 El modelo Objeto- Relación[19]

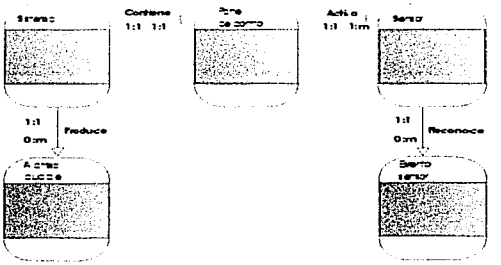
El primer paso en el establecimiento de las relaciones es comprender las responsabilidades de cada clase. El siguiente paso es definir aquellas clases colaboradoras que ayudan en la realización de cada responsabilidad. Debido a esto los colaboradores siempre están relacionados de alguna manera. El tipo de relación más común es la binaria (existe una relación entre dos clases). Una relación binaria posee una dirección específica que se define a partir de que clase desempeña el papel del cliente y cuál actúa como servidor.

[19] RUBIN, K. S. y A. Goldberg, "Object Behavior Analysis", *Comunications of the ACM*, vol. 35, núm. 9, septiembre 1992, págs. 48-62.

El modelo objeto relación puede derivarse en tres etapas:

1) Usando las tarjetas índice CRC, puede dibujarse una red de objetos colaboradores. Primero se dibujan los objetos conectados por líneas sin etiquetas que indican la existencia de alguna relación entre los objetos conectados.
2) Evaluar responsabilidades y colaboradores y cada línea de conexión sin etiquetar recibe un nombre. Para evitar ambigüedades, una punta de flecha indica la <<dirección>> de la relación (figura 2.7).

Figura 2.7 Definición de la cardinalidad



Fuente. FIRESMITH, D. G., Object OrientedRequirements Analysis and Logical Design, Ed. Wiley, 1993.

3) Una vez que se han establecido y nombrado las relaciones, se evalúa cada extremo para determinar la cardinalidad (figura 2.7). Existen cuatro opciones: 0 a 1, 1 a 1, 0 a muchos, o 1 a muchos.

2.5 El modelo objeto-comportamiento[20]

El modelo objeto-comportamiento indica cómo responderá un sistema OO a eventos externos o estímulos. Para crear el modelo se deben ejecutar los siguientes pasos:

- Evaluar todos los casos de uso para comprender la secuencia de interacción dentro del sistema.
- Identificar eventos que dirigen la secuencia de interacción y comprender como estos eventos se relacionan con objetos específicos.
- Crear una traza de eventos para cada caso de uso.
- Construir un diagrama de transición de estados para el sistema.
- Revisar el modelo objeto-comportamiento para verificar exactitud y consistencia.

2.5.1 Identificación de eventos con casos de uso

El caso de uso representa una secuencia de actividades que incluyen a actores y al sistema. Un evento ocurre cada vez que un sistema OO y un actor intercambian información.

Como ejemplo de un evento típico, considere la frase subrayada del caso de uso propietario usa el tecleado para teclear una contraseña de cuatro dígitos. En el contexto del modelo de análisis OO, el actor propietario transmite un evento al objeto panel de control . El evento puede llamarse entrada de contraseña, la información transferida son los cuatro dígitos que forman la contraseña, pero ésta no es una parte esencial del modelo de comportamiento.

2.5.2 Representaciones de estados

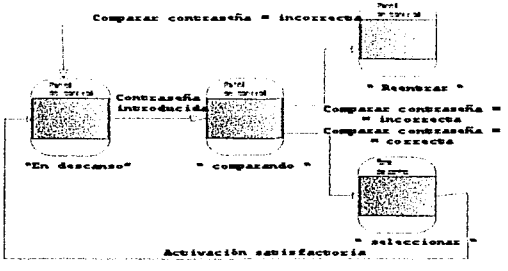
Deben considerarse dos caracterizaciones de estados:

[20] Véase nota anterior.

- El estado de cada objeto cuando el sistema ejecuta su función.
- El estado del sistema observado desde el exterior cuando éste ejecuta su función.

El estado de un objeto adquiere en ambos casos características pasivas y activas. Un estado pasivo es simplemente el estado actual de todos los atributos de un objeto. El estado activo de un objeto indica el estado actual cuando éste entra en una trasformación continua o proceso. Para forzar la transición de un objeto de un estado activo a otro debe ocurrir un evento (a veces llamado disparador). Un componente de un modelo objeto-comportamiento es una representación simple de los estados activos de cada objeto y los eventos (disparadores) que producen los cambios entre estos estados activos. Cada flecha (figura 2.8) representa una transición de un estado activo del objeto a otro. Las etiquetas mostradas en cada flecha representan los eventos que disparan la transición. Es posible especificar información adicional para aportar más profundidad en la comprensión del comportamiento objeto. El análisis puede también especificar un guardián y una acción. Un guardián es una condición Booleana, que debe satisfacerse para posibilitar la ocurrencia de una transición.

Figura 2.8 Representación de las transiciones entre estados activos para el objeto panel de control



Fuente: FIRESMITH, D. G., Object OrientedRequirements Analysis and Logical Design, Ed. Wiley, 1993.

En general, el guardián de una transición depende usualmente del valor de uno o más atributos de un objeto. En otras palabras, el guardián depende del estado pasivo del objeto.

Una acción ocurre concurrentemente con la transición o como una consecuencia de ella y generalmente implica una o más operaciones —— (responsabilidades) del objeto.

El segundo tipo de representación de comportamiento para el AOO considera una representación de estados para el producto general o sistema. Está representación abarca un modelo simple de traza de eventos que indica cómo los eventos causan las transiciones de objeto a objeto y un diagrama de transición de estados que ilustra el comportamiento de cada objeto durante el procesamiento.

El analista crea una representación acerca de cómo los eventos provocan el flujo desde un objeto a otro. Esta representación, llamada traza de eventos (sucesos), es una versión abreviada del caso de uso. Ella representa objetos claves y los eventos que provocan el comportamiento de pasar de objeto a objeto.

Una vez que se ha desarrollado una traza completa de los eventos, todos aquellos que provoquen transiciones entre objetos del sistema pueden incluirse en un conjunto de eventos de entradas y eventos de salida.

Capítulo 3

Diseño orientado a objetos (DOO)

El diseño orientado a objetos **(DOO)** es un modelo de diseño que sirve como un anteproyecto para la construcción del software. A diferencia de los métodos convencionales de diseño del software, el DOO constituye un tipo de diseño que logra un cierto número de diferentes niveles de modularidad. Las componentes principales del sistema están organizados en módulos denominados subsistemas. Los datos y las operaciones que manipulan los datos están encapsulados en objetos, una forma modular que es el bloque de construcción de un sistema OO. En suma, el DOO debe describir la organización de datos específicos, de atributos y los detalles procedimentales de las operaciones individuales. Está representación fragmentada de datos y algoritmos de un sistema OO colaboran a una modularidad general.

La naturaleza única del diseño orientado a objetos descansa en su capacidad de apoyarse en cuatro importantes conceptos de diseño del software: abstracción, ocultación de la información, independencia funcional y modularidad.[21]

Abstracción, permite concentrarse en un problema donde se establece una solución usando el lenguaje del entorno del problema usando conceptos y términos que resultan familiares al cliente.

Ocultación de la información, se especifica y diseña los módulos a utilizar para que la información (procedimiento y datos) contenida dentro de un módulo sea inaccesible a otros módulos que no necesiten esa información.

Independencia funcional, se diseña software de tal manera que cada módulo lo trate una subfunción específica de los requisitos y tenga una sencilla interfaz cuando se vea desde otras partes del programa.

Modularidad, tanto en el programa como en los datos y el concepto de abstracción permiten al diseñador simplificar y reutilizar componentes del software.

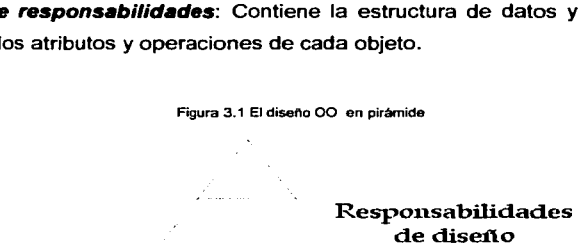
3.1 Diseño de sistemas orientados a objetos[22]

Para sistemas orientados a objetos podemos definir un diseño en pirámide **(figura 3.1)**, las cuatro capas del diseño OO son:

[21] MYERS, G., ***Composite Structured Design***, Ed. Van Nostrand Reinhold, 1978.
[22] GAMMA, E., ***Design Paterns***, Ed. Adisson Wesley, 1995.

- ***La capa del subsistema***. Contiene una representación de cada uno de los subsistemas que le permiten al software conseguir los requisitos definidos por el cliente e implementar la infraestructura técnica que los soporta.
- ***La capa de clases y objetos***. Contiene las jerarquías de clases que permiten crear el sistema usando generalizaciones y especializaciones mejor definidas incrementalmente. Esta capa también contiene representaciones de diseño para cada objeto.
- ***La capa de mensajes***: Establece las interfaces externas e internas para el sistema.
- ***La capa de responsabilidades***: Contiene la estructura de datos y el diseño algorítmico para todos los atributos y operaciones de cada objeto.

Figura 3.1 El diseño OO en pirámide



Fuente: GAMMA, E., ***Design Paterns***, Ed. Adisson Wesley, 1995

3.1.1 El enfoque convencional y el enfoque OO

Los enfoques convencionales para el diseño del software aplican notaciones y conjuntos diferentes para establecer correspondencias entre el modelo de análisis y el diseño. El DOO aplica diseño de datos (cuando se representan atributos), diseño de interfaces (cuando se desarrolla un modelo de intercambio de mensajes), y diseño procedimental (en el diseño de operaciones).

Aunque existe similitud entre los modelos convencionales y los de diseño OO, se ha decidido por renombrar las capas de la pirámide de diseño para reflejar más exactamente la naturaleza de un diseño OO.

El diseño del subsistema se deriva considerando requisitos generales del cliente (representados con casos de uso), los eventos y estados observables externamente (el modelo objeto-comportamiento).

Fichman y Kemerer[23] sugieren 10 componentes de modelado para el diseño:

1. Representación de jerarquías de módulos.
2. Especificación de definiciones de datos.
3. Especificación de la lógica procedimental.
4. Indicación de secuencias de procesos fin-a-fin.
5. Representaciones de estados de objetos y transiciones.
6. Definición de clases y jerarquías.
7. Asignación de operaciones a clases.
8. Definición detallada de las operaciones.
9. Especificación de conexiones de mensajes.
10. Identificación de servicios exclusivos.

3.1.2 Asuntos de diseño

Bertrand Meyer[24] sugiere cinco criterios para juzgar la capacidad que posee un método de diseño en lograr la modularidad:

[23] FICHMAN, R. Y C. Kemerer. "Object-Oriented and Conceptual Design Methdologies". ***Computer***, vol. 25, núm. 10, octubre 1992, págs. 22-39.
[24] MEYER, Bertrán, ***Object-Oriented Databases: Technology, Aplications and Products***, Ed. McGraw Hill, 1994.

- *Descomponibilidad*: Facilidad para descomponer un gran problema en subproblemas más sencillos de resolver.
- *Componibilidad*: Método de diseño asegura que los componentes de un programa (módulos), una vez diseñados y construidos, pueden reusarse para crear otros sistemas.
- *Comprensibilidad*: Facilidad de comprensión de un componente de programa sin referencia a otra información o módulos.
- *Continuidad*: Facilidad de hacer pequeños cambios en un programa y hacer que estos se manifiesten por sí mismos.
- *Protección*: Reduce la propagación de efectos colaterales si ocurre un error en un módulo dado.

A partir de estos criterios Meyer sugiere cinco principios de diseño básicos que pueden derivarse para arquitecturas modulares:

- Unidades modulares lingüísticas.
- Pocas interfaces.
- Pequeñas interfaces.
- Interfaces explícitas.
- Ocultación de la información.

Para lograr un bajo acoplamiento, debe minimizarse el número de interfaces entre módulos (pocas interfaces) y la cantidad de información que se mueve a través de una interfaz (interfaces pequeñas). Cada vez que se comuniquen los módulos, deben realizarlo de una manera obvia y directa (interfaces explícitas).

Finalmente, logramos el principio de ocultar la información cuando toda la información sobre un módulo está oculta al acceso exterior, a menos que la información se defina explícitamente como <<información pública>>.

3.1.3 Visión del DOO

3.1.3.1 Autores del método:

El método de Booch.[25] Abarca un <<proceso de micro-desarrollo>> y un <<proceso de macro-desarrollo>>. El nivel de micro-desarrollo define un conjunto de tareas de diseño que se reaplican en cada etapa del proceso macro-desarrollo.

Descripción del proceso de micro-desarrollo de Booch:

Planificación arquitectónica:

- Agrupar objetos similares en particiones arquitectónicas separadas.
- Distribuir objetos en capas por niveles de abstracción.
- Identificar escenarios relevantes.
- Crear un prototipo de diseño.
- Validar el prototipo de diseño aplicándolo en escenarios de uso.

Diseño táctico:

- Las reglas gobiernan el uso de operaciones y atributos.
- Definir políticas específicas al dominio para la administración de memoria, la gestión de errores y otras funciones de la infraestructura.
- Desarrollar un escenario que describa la semántica de cada política.
- Crear un prototipo para cada política.
- Instrumentar y refinar el prototipo.
- Revisar cada política para asegurar que <<transmite su visión arquitectónica>>

Planificación de la realización:

- Organizar escenarios desarrollados durante el AOO por prioridad.

---

[25] BOOCH, G. Y J. Rumbaugh, ***Unified Method for Object-Oriented Development***, Rational Software Corp, 1996.

- Asignar las correspondientes realizaciones arquitectónicas a los escenarios.
- Diseñar y construir cada realización arquitectónica de manera incremental.
- Ajustar los objetivos y el plan de la realización incremental como se requiera.

El método de Coad y Yourdon.[26] El enfoque de diseño se dirige no solamente a la aplicación, sino también a la infraestructura para la aplicación.

Componentes del dominio del problema:

- Agrupar todas las clases especificas al dominio.
- Diseñar una jerarquía de clases apropiada para las clases de aplicación.
- Trabajar, para simplificar la herencia.
- Refinar el diseño para mejorar el rendimiento.
- Desarrollar una interfaz con el componente de gestión de datos.
- Refinar y añadir objetos en bajo nivel.
- Revisar el diseño y proponer adiciones al modelo de análisis.

Componente de interacción humana:

- Definir los actores humanos.
- Desarrollar escenarios para las tareas.
- Diseñar una jerarquía de órdenes de usuario.
- Refinar la secuencia de interacción del usuario.
- Diseñar clases relevantes y la jerarquía de clases.
- Integrar las clases de IGU (Interfaz Gráfica de Usuario).

Componentes para la gestión de tareas:

- Identificar tipos de tareas.

---

[26] S. PRESSMAN, Roger , ***Ingeniería de software*** , 4ª edición, Editorial Mac Graw Hill, México,1998, 581 pp.

- Estableces prioridades.
- Identificar la tarea que servirá de coordinadora para otras tareas.
- Diseñar objetos apropiados para cada tarea.

La componente para la gestión de datos:

- Diseñar las estructuras de datos y su distribución.
- Diseñar servicios necesarios para manejar las estructuras de datos.
- Identificar las herramientas que puedan ayudar en la implementación de la gestión de datos.
- Diseñar clases apropiadas y la jerarquía de clases.

El método de Jacobson.[27] El modelo de diseño hace hincapié en el seguimiento al modelo de análisis ISOO.

- Considerar adaptaciones para hacer que el modelo de análisis ideal cumpla con el entorno del mundo real.
- Crear bloques como objetos de diseño primario.
- Crear un diagrama de interacción que muestre cómo se pasan los estímulos entre bloques.
- Organizar los bloques en subsistemas.
- Revisar el trabajo de diseño.

El método de Rumbaugh.[28] El diseño del sistema se centra en la distribución de los componentes necesarios para construir un producto o sistema completo.

- Realizar un diseño del sistema.
- Conducir un diseño de objeto.
- Implementar mecanismos de control definidos en el diseño del sistema.

[27] véase nota anterior.
[28] BOOCH, G. Y J. Rumbaugh, ***Unified Method for Object-Oriented Development***, Rational Software Corp, 1996.

- Ajustar la estructura de clase a una herencia fuerte.
- Diseñar el intercambio de mensajes para implementar relaciones entre objetos.
- Empaquetar las clases y asociaciones en módulos.

El método de Wirfs-Brock.[29] Define una secuencia de tareas técnicas en el cual el análisis conduce ineludiblemente al diseño.

- Construir un protocolo para cada clase.
- Crear una especificación de diseño para cada clase.
- Crear una especificación de diseño para cada subsistema.

Para realizar un diseño orientado a objetos, el ingeniero de software debe seguir las siguientes etapas genéricas:

- Describir cada uno de los subsistemas de manera que sean implementables.
- Diseño de objetos.
- Diseño de mensajes.
- Revisión del modelo de diseño e iterar siempre que sea necesario.

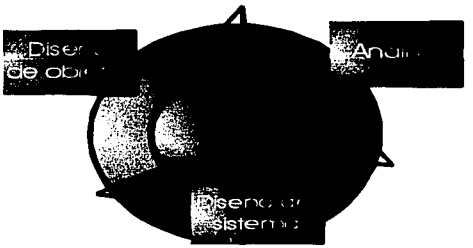
3.2 Componentes genéricos del modelo de diseño OO

Una distinción clara entre el análisis orientado a objetos y el diseño orientado a objetos es que el *análisis* es una actividad de clasificación, además, determina también las relaciones y el comportamiento del objeto. El *diseño* indica los objetos que se derivan de cada clase y cómo estos se interrelacionan unos con otros. Adicionalmente ilustra cómo se desarrollan las relaciones entre

[29] WIRFS-BROCK, R., B. Wilkerson y L. Weiner., Designing Object-Oriented Software, Ed. Prentice Hall, 1990

objetos, cómo se debe implementar el comportamiento y cómo implementar la comunicación entre objetos. El flujo del proceso genérico desde el análisis hasta el diseño se ilustra en la figura 3.2.

Figura 3.2 Flujo de proceso del DOO



Fuente: GAMMA, E., ***Design Paterns***, Ed. Adisson Wesley, 1995

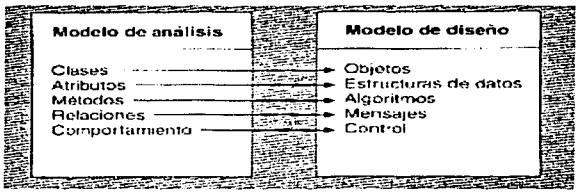
Durante el diseño de los subsistemas, es necesario definir cuatro importantes componentes del diseño:

- *Dominio del problema*: los subsistemas responsables de la implementación de los requisitos del cliente directamente.
- *Interacción humana*: los subsistemas, que implementan la interfaz del usuario.
- *Gestión de tareas*: los subsistemas responsables de control y coordinación de tareas concurrentes que pueden empaquetarse dentro de uno o varios subsistemas.
- *Gestión de datos*: el subsistema que es responsable del almacenamiento y recuperación de objetos.

Cada uno de estos componentes genéricos pueden modelarse a través de una serie de clases así como las relaciones y comportamientos que constituyen requisitos. Los componentes de diseño se implementan definiendo el protocolo que describe formalmente el modelo de mensajes para cada uno de los componentes.

Una vez que se ha definido un subsistema y comienza el diseño de cada uno de los componentes mencionados anteriormente, se traslada el enfoque al diseño de objetos. A este nivel los elementos del modelo CRC se convierten en una realización del diseño (se realiza la conversión del modelo de análisis que es un modelo de diseño durante el diseño de objetos mostrada en la figura 3.3).

Figura 3.3 Conversión del modelo de análisis



Fuente: GAMMA, E., ***Design Paterns***, Ed. Adisson Wesley, 1995

3.3 El proceso de diseño del sistema

La secuencia de actividades propuesta por Rumbaugh y sus colegas[30] es uno de los tratamientos más definitivos a este tema llegando a las siguientes etapas:

- Dividir el modelo de análisis subsistemas.
- Identificar la concurrencia dictada por el problema.
- Asignar subsistemas a procesadores y tareas.
- Elegir una estrategia básica para la implementación de la gestión de datos.
- Identificar los recursos globales y los mecanismos de control necesarios para acceder a ellos.
- Diseñar un mecanismo de control apropiado para el sistema.
- Considerar como manipular las condiciones límite.
- Revisar y considerar los intercambios.

[30] véase notas al pie de página núm. 25,26,27,28,29.

3.3.1 Partición del modelo de análisis

Uno de los principios de análisis es la partición, se divide el modelo para definir colecciones cohesivas de clases, relaciones y comportamientos. Estos elementos se empaquetan como subsistema.

Los elementos de un subsistema comparten alguna propiedad en común. Se caracterizan por sus responsabilidades; esto es, un subsistema puede identificarse por los servicios que realiza.

Al definir (y diseñar) subsistemas, éstos deben cumplir con los siguientes criterios de diseño:

- Debe poseer una interfaz bien definida a través de la cual ocurre toda la comunicación con el resto del sistema.
- Las clases deben colaborar únicamente con otras clases dentro del subsistema.
- El número de subsistemas debe mantenerse pequeño.
- Los subsistemas pueden dividirse internamente para ayudar a reducir la complejidad.

Cuando dos subsistemas se comunican entre sí, se puede establecer un enlace cliente/servidor o punto-a-punto. En un enlace *cliente/servidor* los servicios fluyen en una única dirección. En un enlace *punto-a-punto*, los servicios pueden fluir en cualquier dirección.

3.3.2 Concurrencia y asignación de subsistemas

Si los objetos (o subsistemas) deben actuar sobre eventos asíncronamente y al mismo tiempo, se toman como concurrentes. Cuando son concurrentes, existen dos opciones de asignación:

- Asignar cada subsistema a un procesador independiente.
- Asignar los subsistemas al mismo procesador y ofrecer soporte de concurrencia a través de las capacidades del sistema operativo.

3.3.3 El componente para la gestión de tareas

Este subtema se refiere al momento de ejecutar una tarea o aplicación pueden surgir eventos externos que pueden interrumpir la aplicación.

Coad y Yourdon sugieren la siguiente estrategia para el diseño de los objetos que manejan tareas concurrentes:

- Se determinan las características de la tarea.
- Se define una tarea coordinadora y los objetos asociados.
- El coordinador y las otras tareas se integran.

Las tareas dirigidas por eventos o por reloj son las que más comúnmente encontramos. Ambas se activan por una interrupción de alguna fuente exterior (procesador o un sensor) mientras que la última esta gobernada por el reloj del sistema.

Adicionalmente a la manera en que una tarea se inicia, se debe determinar la prioridad y sentido crítico de la tarea. Tareas con una alta prioridad deben tener un acceso inmediato a los recursos del sistema. Aquellas con un alto grado de sentido crítico deben continuar su operación aún cuando la disponibilidad de recursos está reducida o el sistema esté operando con cierto nivel de degradación.

Al terminar de definir las características de la tarea, se definen los atributos y operaciones de los objetos requeridos para lograr comunicación y coordinación con otras tareas. La plantilla básica de la tarea toma la forma:

- *Nombre de tarea*: el nombre del objeto.
- *Descripción*: una descripción narrativa acerca del propósito del objeto.
- *Prioridad*: prioridad de la tarea.
- *Servicios*: una lista de operaciones que constituyen las responsabilidades del objeto.
- *Coordinado por*: la manera según la cual se invoca el comportamiento del objeto.
- *Comunicar vía*: valores de datos de entrada y salida relevantes a la tarea.

3.3.4 El componente para la gestión de datos

Dentro del contexto del sistema, un sistema de gestión de datos se usa a menudo como un almacén común de datos para todos los subsistemas. Incluye el diseño de los atributos y operaciones necesarias para la gestión de datos.

La gestión de datos abarca dos áreas distintas:

1) La gestión de datos críticos para la propia aplicación.
2) La creación de una infraestructura para el almacenamiento y recuperación de objetos.

Para entenderlo mucho mejor, imagine cuando esta trabajando en su computadora y se va la luz, automáticamente pierde la información que todavía no se había guardado, pero su ventaja fue que gran parte ya estaba almacenada y no tendría que realizar su trabajo completo sino solo una parte, por que usted puede recuperar su información de donde lo tenía almacenado. De manera similar ocurre para los sistemas solo que en este caso utilizan una base de datos relacional o una base de datos orientada a objetos.

3.3.5 El componente para la gestión de recursos

Existe una gran variedad de recursos para los sistemas y en muchas instancias, los subsistemas compiten por estos recursos al mismo tiempo, para poder ejecutar su tarea o aplicación. Los recursos globales del sistema pueden ser entidades externas (ejemplo, una torre de discos, procesador, o línea de comunicación) o abstracciones (base de datos, un objeto). Rumbaugh y sus colegas sugieren que cada recursos debería ser propiedad de un <<objeto guardián>>. El objeto guardián es el portero del recurso, controlando los accesos a él y moderando las solicitudes en conflicto sobre él.

3.3.6 El componente de interfaz hombre-máquina

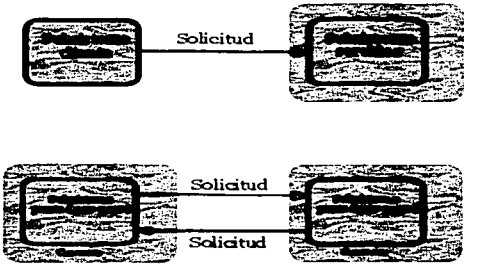
El modelo de análisis OO contiene escenarios de uso (llamados casos de uso) y una descripción del papel que tienen los usuarios (llamados actores) al interactuar con el sistema. Esto sirve como dato de entrada al proceso de diseño de la **IHM** (Interfaz Hombre-Máquina).

Una vez que se han definido el actor y su escenario de uso, se identifica una jerarquía de órdenes, la jerarquía de comandos define las principales categorías de menús del sistema. La jerarquía de comandos se refina iterativamente hasta que cada caso de uso pueda implementarse navegando a través de la jerarquía de funciones.

3.3.7 Comunicación entre subsistemas

La comunicación puede ocurrir al establecer un enlace cliente/servidor o punto-a-punto (figura 3.4), debemos especificar el contrato que existe entre los subsistemas. Recuerde que un contrato brinda indicaciones acerca de las maneras en que un subsistema puede interactuar con otro.

Figura 3.4 Modelo de colaboración entre subsistemas



Fuente: GAMMA, E., ***Design Paterns***, Ed. Adisson Wesley, 1995

Pasos para especificar contrato a un subsistema:

1. Listar cada solicitud que pueda realizarse por parte de los colaboradores del subsistema. Organizar dichas solicitudes según el subsistema y definirlas dentro del marco de uno o más contratos apropiados.

2. Para cada contrato, marque las operaciones (heredadas y privadas) requeridas para implementar las responsabilidades que implica el contrato. Asegúrese de asociar las operaciones con clases específicas que residan dentro del subsistema.

3. Cree una tabla de la misma forma a la mostrada por la figura 3.5 Para cada contrato, se crearán las siguientes entradas en la tabla:

- ✓ *Tipo*: el tipo de contrato (cliente/servidor o punto-a-punto).
- ✓ *Colaboradores*: los nombres de los subsistemas que son parte del contrato.
- ✓ *Clase*: los nombres de las clases que soportan servicios implicados por el contrato.
- ✓ *Operación*: los nombres de las operaciones que implican los servicios.
- ✓ *Formato del mensaje*: el formato de mensaje requerido para implementar la interacción entre colaboradores.

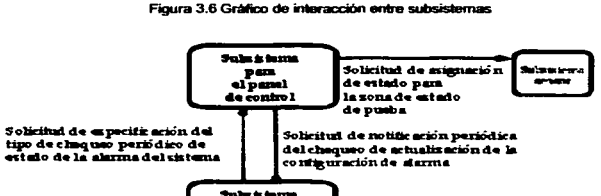
4. Si los modos de interacción entre los subsistemas son complejos se creará un grafo de intersección entre subsistemas (figura 3.6).

Figura 3.5 Tabla de colaboraciones del subsistema

Contrato	Tipo	Colaboradores	Clase(S)	Operación(es)	Formato del mensaje

Fuente: Tabla realizada por la autora de esta tesis

Figura 3.6 Gráfico de interacción entre subsistemas



Fuente: GAMMA, E., ***Design Patterns***, Ed. Adisson Wesley, 1995

3.4 El proceso de diseño de objetos

Debemos desarrollar un diseño detallado de los atributos y operaciones que incluye cada clase, y una especificación completa de los mensajes que conectan la clase con sus colaboradores.

3.4.1 Descripción de objeto

Una descripción de un objeto (una instancia de una clase o subclase) puede tomar una de estas dos formas:

1. Una descripción del protocolo que establece la interfaz de un objeto definiendo cada mensaje que el objeto puede recibir y la correspondiente operación que el mismo ejecuta al recibir el mensaje.

2. La descripción del protocolo que no es nada más que un conjunto de mensajes y el comentario correspondiente para cada uno de ellos.

Para un gran sistema con muchos mensajes, es posible a menudo crear categorías de mensajes. Una descripción de la implementación de un objeto aporta los detalles internos (ocultos) necesarios para la implementación pero que no son necesarios para su invocación. El diseñador del objeto debe, por tanto, crear los detalles internos del objeto.

Una descripción de la implementación se compone de la siguiente información:

- Especificación del nombre del objeto y referencia a su clase.
- Especificación de las estructuras de datos privadas indicando los elementos de datos y de tipos.
- Descripción procedimental de cada operación o alternativamente, punteros a tales descripciones procedimentales.

3.4.2 Diseño de algoritmos y estructuras de datos

Se crea un algoritmo para implementar la especificación de cada operación. El algoritmo es una simple secuencia computacional o procedimental que puede implementarse como un módulo de software autocontenido.

Aunque existen muchos tipos diferentes de operaciones, estas pueden generalmente dividirse en tres amplias categorías:

I. Operaciones que manipulan datos de alguna manera.
II. Operaciones que realizan cálculos.
III. Operaciones que monitorizan un objeto debido a la ocurrencia de un evento controlador.

Una vez que se ha creado el modelo de objeto básico, debe ocurrir la optimización. Rumbaugh sugiere tres etapas principales para la optimización del DOO:

- Revisar el modelo de objeto-relación para asegurarse que el diseño implementado lleva una utilización eficiente de los recursos y a una facilidad de implementación.
- Revisar las estructuras de datos atributos y las operaciones algorítmicas correspondientes para aumentar el rendimiento.
- Crear nuevos atributos para preservar información derivada, evitando para esto la repetición de computaciones previas.

3.4.3 Componentes de programas e interfaces

Un aspecto importante de la calidad del diseño del software es la modularidad, esto es, la especificación de componentes del programa (módulos) que se combinan para formar un programa completo. El enfoque orientado a objetos define al objeto como una componente del programa la cual está auto-enlazada con otras componentes.

3.4.4 Patrones de diseño

Los mejores diseñadores poseen una habilidad para ver patrones que caracterizan un problema, y los patrones correspondientes que pueden combinarse para crear una solución. Gamma y sus colegas examinan esto cuando expresan:

"Se encontraran patrones recurrentes de clases y objetos en comunicación en muchos sistemas orientados a objetos. Estos patrones resuelven problemas de dueño específicos y hacen el diseño orientado a objetos más flexible, elegante y en última instancia reusable".

3.4.5 Descripción de un patrón de diseño

Todos patrones de diseño pueden describirse a través de la especificación de cuatro piezas de información:

- El nombre del patrón.

- El problema al cual se aplica generalmente el patrón.
- Las características del patrón de diseño.
- Las consecuencias de la aplicación del patrón de diseño.

El nombre de patrón de diseño es una abstracción que aporta un significado acerca de su aplicabilidad y objetivos. La descripción del problema indica el entorno y las condiciones que deben existir para hacer aplicable el patrón de diseño.

Las características del patrón indican los atributos del diseño que deben ajustarse para permitirle al patrón acomodarse a una variedad de problemas. Finalmente, las consecuencias asociadas con el uso del patrón de diseño aportan una indicación de las ramificaciones de las decisiones de diseño.

3.4.6 Uso de patrones en el diseño

Los patrones de diseño pueden usarse aplicando mecanismos diferentes: ***herencia y composición***. La herencia es un concepto OO fundamental. A través del uso de la herencia un patrón de diseño existente se convierte en una plantilla para una subclase nueva. Los atributos y operaciones que existen en el patrón pasan a ser parte de la clase.

La composición es un concepto que nos lleva la de objetos agregados. El objeto complejo puede ensamblarse a partir de la selección de un conjunto de patrones de diseño y la composición del objeto seleccionado (o subsistema).

Gamma y sus colegas sugieren que la composición de objetos debe ser favorecida por encima de la herencia cuando existen ambas opciones.

3.5 Desarrollo del software

El objetivo principal de la programación orientada a objetos extiende el modelo de diseño al dominio ejecutable (es la producción de un código fuente que sea fácil de leer y comprender).

La claridad del código fuente facilita la depuración, prueba y modificación, y estas actividades consumen una gran porción de los presupuestos de la programación. El estilo de codificación se manifiesta en los patrones de elecciones hechas entre modos alternos de expresar un algoritmo. Un estilo de codificación existe entre distintos programadores. Los lenguajes de programación son los vehículos notacionales que se usan para instrumentar productos de programación. Las características disponibles en el lenguaje de instrumentación ejercen una fuerte influencia sobre la estructura arquitectónica y los detalles algoritmicos escritos en ese lenguaje. La programación orientada a objetos traduce las clases, atributos, operaciones y mensajes en una forma que pueda ejecutarse en una máquina.

La creación de un programa legible y confiable es un proceso creativo, por lo que es imposible imponer reglas rígidas que gobiernen el estilo de programación. Sin embargo, se pueden establecer varios principios generales que mejoran la legibilidad de los programas en diferentes lenguajes, como lo son :

- **Nombres de los programas**: los objetos de un programa como lo son constantes, variables, procedimientos, funciones y tipos, representan entidades en el mundo real. De acuerdo con esto, los nombres de los objetos deben estar estrechamente relacionados con los nombres de las entidades del mundo real que modelan.
- **Construcciones de control en los programas**: en un programa deben usarse construcciones de control para que el flujo del programa sea descendente. Al poder considerar a los ciclos como una proposición compuesta, la ejecución debe empezar por la primera proposición de programa, cada proposición debe ejecutarse por turno y la ejecución debe terminar con la última proposición. Las unidades de programa, los ciclos y las proposiciones de decisión deben tener solo un punto de entrada y una sola salida. Ejemplo :

```
if C1 then
        S1
else
        if C2 then
                S2
        else
                if C3 then
                        S3
                else . . . . . .
```

- **Distribución del programa**: la distribución afecta a la legibilidad de los programas. El uso de líneas en blanco, el énfasis en las palabras reservadas y la agrupación consistente en párrafos que hacen que el programa sea más elegante y fácil de leer, y actúan como separadores que distinguen una parte del programa con otra. Ejemplo :

```
procedure Cuenta_ocurrencias (var arreglo_ent: arreglo_entero:
tamaño_arreglo : integer);
var I, cont : integer;
begin
        cont: = 1
        for I= 1 to tamaño_arreglo-1 do
                if arreglo_ent =arreglo_entero then
                else . . . . . .
```

La rapidez de los cambios en la tecnología del hardware de computadoras es tal que la maquinaria de computación se queda anticuada mucho antes que los programas que se ejecutan en esas máquinas. Por tanto, es muy importante realizar los programas de manera que se puedan aplicar en más de una configuración del sistema de cómputo y del sistema operativo. Esto es de vital importancia si un sistema de programación tiene un mercado amplio como producto: *cuanto mayor sea el número de máquinas en las que se aplica el sistema, mayor será su mercado de potencial.* Es aquí donde utilizamos el término ***transportabilidad*** de los programas.

Los programas transportables deben ser autónomos. El programa no debe depender de la existencia de agentes externos que le proporcionen las funciones requeridas. Ya en la práctica la autonomía completa es casi imposible de lograr, y el programador que intente producir un programa transportable debe arreglárselas para aislar las referencias necesarias al ambiente externo. Cuando cambia ese ambiente externo, las partes del programa que son dependientes se pueden modificar.

Es convincente la observación de que un programa se lee más veces de las que se escribe, y que es responsabilidad de los diseñadores crear y construir programas legibles con lenguajes de programación que así lo permitan.

Sin embargo la legibilidad de los programas no depende de las características del lenguaje sino de el estilo en que un programa está escrito .

Finalmente, después de completar el sistema, los clientes llevaran a cabo una serie de pruebas (revisiones) de aceptación para verificar que el sistema se desempeña conforme a lo especificado. De aquí parte lo que son las pruebas orientadas a objetos que se explican en el siguiente capítulo.

Capítulo 4

Pruebas orientadas a objetos (PrOO)

4.1 Implementación del software

Las clases de objetos y relaciones desarrolladas durante el análisis de objetos se traducen finalmente a una implementación concreta. Durante la fase de implementación es importante tener en cuenta los principios de la ingeniería del software de forma que la correspondencia con el diseño sea directa y el sistema implementado sea flexible y extensible. No tiene sentido que utilicemos AOO y DOO de forma que potenciemos la reutilización de código, el dominio del problema y el sistema informático, si luego perdemos todas estas ventajas con una implementación de mala calidad. La especialización al lenguaje de programación o bases de datos describe cómo traducir los términos usados en el diseño a los términos y propiedades del lenguaje de implementación. Aunque el diseño de objetos es bastante independiente del lenguaje actual, todos los lenguajes tendrán sus especialidades durante la implementación final incluyendo las bases de datos.

En el modelo de implementación, el concepto de rastreabilidad es también muy importante, dado que al leer el código fuente se debe poder rastrear directamente del modelo de diseño y análisis.

Un aspecto importante durante el diseño de objetos es la selección del **lenguaje de programación**. La elección del lenguaje influye en el diseño pero el diseño no debe depender de los detalles del lenguaje. Si se cambia de lenguaje de programación no debe requerirse el re-diseño del sistema. Los lenguajes de programación y sistemas operativos difieren mucho en su organización, aunque la mayoría de los lenguajes tienen la habilidad de expresar los aspectos de la especificación del software que son las estructuras de datos (objetos), flujo dinámico de control secuencial (ciclos, condiciones) o declarativo (reglas, tablas), además de contar con modificaciones en las funciones. En el caso de un lenguaje orientado a objetos las estructura básicas son las propias clases. Dependiendo del lenguaje de programación esto puede hacerse más sencillo o complicado. Los lenguajes más utilizados varían desde los "semi" orientado a objetos como Ada y Modula-2 hasta los estructurados tradicionales, como C, Pascal, Fortran y COBOL.

De manera similar a las pruebas la documentación debe ocurrir a lo largo del desarrollo del sistema y no como una etapa final del mismo. Existen diferentes tipos de documentos que deben ser

generados como apoyo al sistema. Cada uno de estos documentos tiene diferentes objetivos y está dirigidos a distintos tipos de personas, desde los usuarios no técnicos hasta los desarrolladores más técnicos. Los siguientes son algunos de los documentos o manuales más importantes:

- **Manual del usuario**, que le permite a un usuario comprender como utilizar el sistema.
- **Manual del programador**, que le permite a un desarrollador entender los aspectos de diseño considerados durante su implementación.
- **Manual del operador**, que le permite al encargado de operar el sistema comprender más generales como son los modelos de requisitos y análisis.

Realmente no hay límite al número y detalle que se puede lograr mediante la documentación, de manera similar a que no hay límite a que tanto se puede extender y optimizar un sistema. La idea básica es mantener un nivel de documentación que sea útil aunque es necesario adaptarlo al proceso de la organización.

Una visión equivocada del mantenimiento de un sistema es que esto involucra únicamente la corrección de errores. El mantenimiento realmente va más allá de corregir problemas, se basa en generar nuevos desarrollos pero tomando como punto de partida el sistema ya existente. En cierta manera es regresar al resto de las actividades pero sin partir de cero, en la mayoría de los casos, le resulta difícil al cliente establecer explícitamente al principio todos los requisitos para su sistema.

Debido a que pueden existir errores se realizan una serie de pruebas que se mencionan a continuación.

4.2 Pruebas orientadas a objetos[31]

Para muchos sistemas, los programadores tienen la responsabilidad de probar su propio código (módulos u objetos). Una vez que lo hacen, el trabajo se pasa a un equipo de integración que

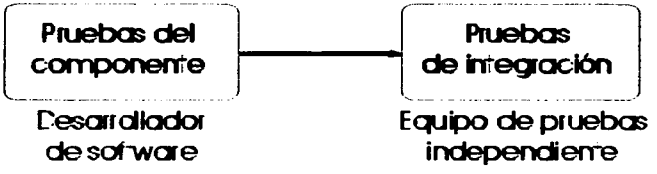
[31] SOMM ERVILLE, Ian, Ingeniería de software, 6ª edición, Ed. Addison Wesley, México.

integra los módulos de los diferentes desarrolladores, construye el software y prueba el sistema en conjunto. Sin embargo, para sistemas independientes se utiliza un proceso más formal donde los probadores son responsables de todas las etapas del proceso de prueba. Las pruebas se desarrollan de forma independiente y se lleva a cabo un registro detallado.

Cuando se prueban sistemas independientes, una especificación detallada de cada componente de software es utilizada por el equipo independiente para generar las pruebas del sistema. Sin embargo, en muchos casos, llevar a cabo las pruebas es un proceso más intuitivo puesto que no existe tiempo para redactar especificaciones detalladas de cada parte de un sistema de software.

En la figura 4.1, señala que existen dos actividades fundamentales en el proceso de pruebas. Éstas son ***pruebas de componentes***, en las que los componentes del sistema se pruebas de forma individual, y ***pruebas de integración***, en las que las colecciones de componentes se integran en subsistemas y se prueba el sistema final.

Figura 4.1 Proceso de pruebas



Fuente: SOMM ERVILLE, Ian, Ingeniería de software, 6ª edición, Ed. Addison Wesley, México.

Estas actividades se aplican de igual forma a los sistemas orientados a objetos. Sin embargo, existen diferencias importantes entre los sistemas orientados a objetos y los sistemas convencionales:

1. Los objetos como componentes individuales son a menudo más grande que una sola función, puesto que utilizan la reutilización de código.

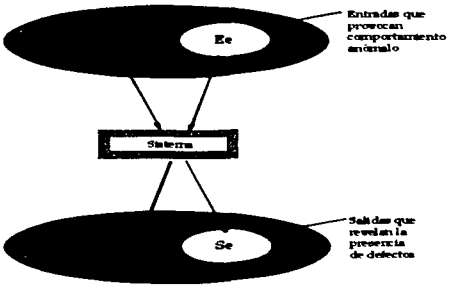
2. Los objetos integrados en los subsistemas son por lo general débilmente acoplados y no existe un "máximo" para el sistema, ya que si se escala el sistema utiliza más funciones y se hace mucho más grande.

En un sistema orientado a objetos, se pueden identificar cuatro niveles de pruebas:

1. ***Probar las operaciones individuales asociadas con los objetos***. Éstas son funciones o procedimientos y utiliza el enfoque de *caja negra*. Definida posteriormente:

   El sistema es una "caja negra" cuyo comportamiento solo se puede determinar estudiando las entradas y salidas relacionadas. Otro nombre es *pruebas funcionales*. La siguiente figura muestra el modelo de un sistema al que se le aplica una prueba de caja negra, el cual el probador (ya sea cliente o programador) introduce las entradas en los componentes del sistema y examina las salidas correspondientes. Si la salida no es la correspondiente, entonces existe un problema en el software.

Figura 4.2 Pruebas de caja negra



Fuente: SOMMERVILLE, Ian, Ingeniería de software, 6ª edición, Ed. Addison Wesley, México.

2. ***Probar clases de objetos individuales***. El principio de las pruebas de caja negra no cambia pero la noción de una clase de equivalencia se debe extender para cubrir las secuencias de operaciones relacionadas. Se especifica en la sección 4.2.1.
3. ***Probar clústers de objetos***. La integración descendente o ascendente no es apropiada para crear grupos de objetos relacionados. Se utilizan otros enfoques como las pruebas basadas en escenarios. Descritas en la sección 4.2.2.
4. ***Probar el sistema orientado a objetos***. La verificación y validación en comparación con la especificación de requerimientos del sistema se llevan a cabo exactamente de la misma forma que para cualquier otro sistema.

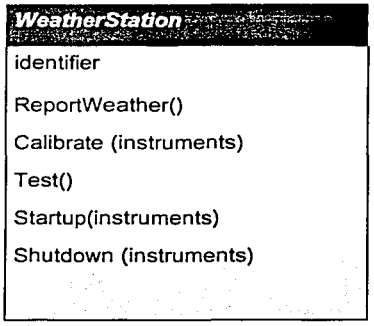
Las siguientes secciones dan un panorama general de un enfoque básico para las pruebas de sistemas orientado a objetos.

4.2.1 Prueba de clases de objetos

Cuando se prueban los objetos deben incluir:

- Pruebas aisladas de todas las operaciones asociadas con el objeto.
- La selección e interrogación de todos los atributos asociados con el objeto.
- La ejecución de los objetos en todos los estados posibles. Esto significa que se simulan todos los eventos que provocan un cambio de estado en el objeto.

Para entender mucho mejor estos puntos veamos éste ejemplo:



Supóngamos que esta es una interfaz de alguna estación climática el cual tiene un identificador (constante establecida), que sólo necesita una prueba para verificar que se estableció. Por otro lado, se necesitan definir casos de prueba para ReportWeather, Calibrate, Test, Startup, Shutdown de forma independiente, en algunos casos, las secuencias de prueba son necesarias. Por ejemplo, shutdown necesita haber ejecutado el método startup, ya que en estos casos se pudo utilizar la herencia. Cuando una superclase provee operaciones que son heredadas por varias subclases , todas estas subclases se prueban con todas las operaciones heredadas. La razón de esto es que la operación heredada hace suposiciones acerca de otras operaciones y atributos y éstos cambian cuando se heredan. De igual forma, cuando una operación de la superclase se sustituye, la operación sobrescrita se debe probar.

4.2.2 Integración de objetos

Cuando se desarrollan sistemas orientados a objetos, las operaciones de los datos se integran para formar objetos y clases de objetos. Probar estas clases de objetos corresponde a probar módulos, pero no existe una equivalencia para probar esto. Sin embargo, Murphy (1994)

sugiere que se prueben juntos los grupos y clases que actúan en combinación para proveer un conjuntos de servicios, a esto se le llama *pruebas de clústers.*

Los clústers deben crearse conociendo que operación realizan y sus características en el sistema. Pueden existir tres posibles enfoques para las pruebas que se tengan que utilizar:

- ✓ ***Pruebas de casos de uso o basados en escenarios***, estos describen una manera de utilización del sistema. Las pruebas se basan en estas descripciones de escenarios y en los clústers de objetos creados para apoyar a los casos de uso.
- ✓ ***Pruebas de cadenas de eventos***, se basan en probar la respuesta del sistema a una entrada en particular o a un conjuntos de eventos de entrada. A menudo los sistemas orientados a objetos son conducidos por eventos por lo que ésta es una forma apropiada para probar su utilización. Para utilizar este enfoque, se tiene que identificar cómo se lleva a cabo el procesamiento de cadenas de eventos en el sistema.
- ✓ ***Probar la interacción de los objetos***, se basa en identificar las trayectorias de los mensajes de los métodos o la secuencia de interacción de los objetos que se detienen cuando una operación del objetos no llama a los servicios de cualquier otro objeto. También se identifica llamada **ASF**(Función Atómica del Sistema), la cual esta compuesta por un evento de entrada seguido de una secuencia de trayectorias que finalizan en un evento de salida.

Estas pruebas son apropiadas ya que se pueden organizar de tal forma que los escenarios más probables se prueben primero, y los no comunes se prueben más adelante. Esto satisface el *principio fundamental de las pruebas* que señala que el esfuerzo en las pruebas se debe dedicar a aquellas partes del sistema que se utilizan más.

Cuando se eligen escenarios para formular las pruebas del sistema, es importante asegurar que cada método se ejecuta en cada clase al menos una vez. Por lo tanto, se puede crear una lista de verificación de clases de objetos y métodos y, cuando se elige un escenario, se pueden marcar los métodos que se ejecutan. Claro que no se pueden ejecutar todas las combinaciones de métodos, pero esto al menos asegura que todos los métodos tradicionales se prueben como parte de una secuencia.

Capítulo 5

Métricas orientadas a objetos

5.1 Objetivos y características de las métricas orientadas a objetos

El Software Orientado a Objetos (OO) es fundamentalmente distinto del software que se desarrolla utilizando métodos convencionales. Las métricas para sistemas OO deben de ajustarse a las características que distinguen el software OO del software convencional.

Estas métricas hacen hincapié en el ***encapsulamiento, la herencia, complejidad de clases y polimorfismo.*** Por lo tanto las métricas OO se centran en métricas que se pueden aplicar a las *características* de encapsulamiento, localización, ocultamiento de información, herencia y técnicas de abstracción de objetos que hagan única a esa clase.

Como en todas las métricas los ***objetivos*** principales de las métricas OO se derivan del software convencional: comprender mejor la calidad del producto, estimar la efectividad del proceso y mejorar la calidad del trabajo realizado a nivel del proyecto.

Se conoce que las medidas y las métricas son componentes clave de cualquier disciplina de la ingeniería. A medida que los sistemas OO van siendo más habituales, resulta fundamental que los ingenieros del software estimen la calidad de los diseños y la efectividad de los programas 00.

5.1.1 Encapsulamiento

Se define el encapsulamiento como "el empaquetamiento (o enlazado) de una colección de elementos.

Entre los ejemplos de encapsulamiento de bajo nivel (software convencional) se cuentan los registros y matrices. Los mecanismos de nivel medio para el encapsulamiento son los subprogramas (por ejemplo, procedimientos, funciones, subrutinas y párrafos). Para los sistemas 00, el encapsulamiento comprende las responsabilidades de una clase, incluyendo sus atributos (y otras clases para objetos agregados), operaciones, y los estados de la clase, según se definen mediante valores específicos de atributos.

El encapsulamiento influye en las métricas cambiando el objetivo de la medida, que pasa de

ser un único módulo a ser un paquete de datos (atributos) y de módulos de procesamiento (operaciones).

5.1.2 Localización

La localización es una característica del software que indica la forma en que se concentra la información dentro de un programa mediante el encapsulamiento tanto de datos como de procesos dentro de los límites de una clase u objeto.

Las métricas de software se han centrado en la estructura interna o complejidad de las funciones (por ejemplo, longitud del módulo, cohesión, o complejidad) o bien en la forma en que las funciones se conectan entre sí (por ejemplo, acoplamiento de módulos).

Dado que las clases constituyen la unidad básica de los sistemas 00, la localización está basada en los objetos. Por tanto, las métricas deberían de ser aplicables a la clase (objeto) como si se tratara de una entidad completa. Además, la relación entre operaciones (funciones) y clases no es precisamente uno-a-uno. Por tanto, las métricas que reflejan la forma en que colaboran las clases deben de ser capaces de adaptarse a las relaciones uno-a-muchos y muchos-a-uno.

5.1.3 Ocultamiento de información

El ocultamiento de información suprime los detalles operativos de un componente de un programa. Tan sólo se proporciona la información necesaria para acceder a ese componente o a aquellos otros componentes que deseen acceder a él.

Un sistema 00 bien diseñado debería de impulsar al ocultamiento de información. Por tanto, aquellas métricas que proporcionen una indicación del grado en que se ha logrado el ocultamiento proporcionarán una indicación de la calidad del diseño 00.

5.1.4 Herencia

La herencia es un mecanismo que hace posible que los compromisos de un objeto se difundan a otros objetos. La herencia se produce a lo largo de todos los niveles de la jerarquía de clases.

Las métricas OO se centran mucho en esta característica. Por ejemplo, en el número de descendientes (número de instancias inmediatas de una clase), número de predecesores (número de generalizaciones inmediatas), y grado de anidamiento de la jerarquía de clases (profundidad de una clase dentro de una jerarquía de herencia).

5.1.5 Abstracción

La abstracción permite al diseñador centrarse en los detalles esenciales de algún componente de un programa (tanto si es un dato como si es un proceso) sin preocuparse por los detalles de nivel inferior. Cuando los niveles de abstracción van elevándose, se ignoran más y más detalles, por lo tanto, se proporciona una visión más general de un concepto u objeto. A medida que pasamos a niveles mas reducidos de abstracción, se muestran más detalles, esto es, se proporciona una visión más específica de un concepto u objeto.

Dado que una clase es una abstracción que se puede visualizar con muchos niveles distintos de detalles, y de muchas maneras diferentes, las métricas OO representan la abstracción en términos de medidas de una clase (por ejemplo, número de instancias por clase por aplicación).

5.2 Métricas para el modelo de diseño orientado a objetos

Un diseñador experimentado sabe como puede caracterizar un sistema 00 para que se implemente de forma efectiva en los requisitos del cliente. Pero a medida que los modelos de diseño 00 van creciendo de tamaño y complejidad, puede resultar beneficiosa una visión más objetiva de las características del diseño, tanto para el diseñador experimentado como para el menos experimentado.

Una visión objetiva del diseño debería de tener un componente cuantitativo y esto nos lleva a las métricas 00, en donde se pueden aplicar no solo al modelo de diseño, sino también al modelo de análisis.

5.3 Métricas orientadas a clases

Se sabe que la clase es la unidad principal de todo sistema 00. Por consiguiente, las medidas y métricas para una clase individual, la jerarquía de clases, y las colaboraciones de clases resultarán sumamente valiosas para un ingeniero de software que tenga que estimar la calidad de un diseño. Se ha visto que la clase encapsula a las operaciones (procesamiento) y a los atributos (datos).

La clase suele ser el "predecesor" de las subclases (que a veces se denominan "descendientes") que heredan sus atributos de operaciones. La clase suele colaborar con otras clases. Todas estas características se pueden utilizar como bases de las métricas explicadas a continuación:

5.3.1 El conjunto de métricas CK

Uno de los conjuntos de métricas de software 00 a los que se hace más ampliamente referencia es el propuesto por Chidamber y Kemener.[32] Estas métricas propuestas de diseño basadas en clases, a las cuales suele llamarse con el nombre de conjunto de métricas CK para sistemas 00.[33]

Los métodos ponderados por clase (MPC), son aquellos en donde se definen *n* métodos de complejidad C_1, C_2..., C_n, para una clase *C*. El número de métodos y su complejidad es un indicador razonable de la cantidad de esfuerzo necesaria para implementar y comprobar una clase.

[32] CHIDAMBER, S. R. Y F. Kemerer, "A Metrics Suite for Object-Oriented Design", ***IEEE Transf. Software Engineering***, volumen 20, núm. 6, junio 1994, págs. 476-493.
[33] Chidamber y Kemerer utilizan el término *método* en lugar de *operaciones*.

Además, cuanto mayor sea el número de métodos, más complejo será el árbol de herencia (todas las subclases heredan el método de sus predecesores).

A medida que el número de métodos crece para una clase dada, se vuelve cada vez más específico. El desarrollar un contador del número de métodos de una clase no es sencillo. Sin embargo, las implicaciones para las métricas pueden ser significativas.

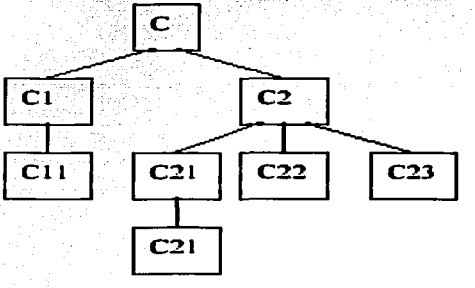
Una posibilidad es restringir los miembros heredados. Esto sería que los miembros heredados ya habrán sido contados en la clases en que fueran definidos, así que el incremento de clase es la mejor medida de su funcionalidad. Con objeto de entender lo que hace una clase, la fuente de información son sus propias operaciones. Si una clase no puede responder a un mensaje, entonces esa clase pasará el mensaje a su predecesor(es).

Árbol de Profundidad de Herencia (APH). Esta métrica se define como la longitud máxima desde el nodo hasta la raíz del árbol (figura 6.1).

A medida que crece el APH, es más probable que las clases de niveles inferiores hereden muchos métodos. Esto da lugar a posibles dificultades cuando se intenta definir el comportamiento de una clase. Una jerarquía de clases profunda (con un valor grande de APH) lleva también a una mayor complejidad de diseño.

Una ventaja es que los valores grandes de APH implican que se pueden reutilizar muchos métodos.

Figura 6.1 Jerarquía de clases



Fuente: CHIDAMBER, S. R. Y F. Kemerer, "A Metrics Suite for Object-Oriented Design", *IEEE Transf. Software Engineering*, volumen 20, núm. 6, junio 1994, págs. 476-493.

Número de Descendientes (NDD). Las subclases que son subordinadas a una clase son denominan descendientes. En la figura 6.1 la clase *C2* tiene tres descendientes las subclases *C21, C22 y C23.*

A medida que crece el número de descendientes, se incrementa la reutilización, pero también es cierto, que a medida que crece NDD, existe la posibilidad de que algunos de los descendientes no sean realmente miembros propios de la clase predecesora. A medida que NDD va creciendo, la cantidad de pruebas crecerá también.

Respuesta Para una Clase (RPC). Es un conjunto de métodos que pueden ser ejecutados en respuesta a un mensaje recibido por un objeto de esa clase. RPC se define como el número de métodos existentes en el conjunto de respuestas.

A medida que crece RPC, el esfuerzo necesario para la comprobación crece, porque la sucesión de comprobación va creciendo y también la complejidad global de diseño de la clase crece.

Carencia de Cohesión en los Métodos (CCM). Es el número de métodos que acceden a uno o más de los mismos atributos. Por ejemplo, si ningún método accede a los mismos atributos, entonces CCM será 0, es distinto de 0 suponiendo que existe una clase con 6 métodos, en donde cuatro de los métodos tienen en común uno o más atributos, por consiguiente, CCM=4.

Esto incrementará la complejidad del diseño de clases.

5.3.2 Métricas propuestas por Lorenz y Kidd

En el libro de métricas realizado por Lorenz y Kidd[34], dividen las métricas basadas en clases en cuatro categorías: tamaño, herencia, valores internos y valores externos.

Tamaño de Clase (TC). El tamaño general de una clase se puede determinar empleando las medidas siguientes:

- El número total de operaciones (tanto operaciones heredadas como privadas de la instancia) que están encapsuladas dentro de la clase.

[34] LORENZ, M. y J. Kidd, Object-Oriented Software Metrics, Ed. Prentice Hall, 1994.

- El número de atributos (tanto atributos heredados como atributos privados de la Instancia) que están encapsulados en la clase.

Si existen valores grandes de TC una clase puede tener demasiada responsabilidad, lo cual reducirá la reutilización de la clase y complicará la implementación y la comprobación, cuanto menor sea el valor, más probable es que las clases existentes dentro del sistema se puedan reutilizar ampliamente.

Número de Operaciones Invalidadas por una subclases (NOI). Existen casos en que una subclase sustituye una operación heredada de su superclase a esto se le denomina *invalidación*. Esto da lugar a una jerarquía de clases débil y a un software 00 si el NOI es elevado ya que puede resultar difícil de comprobar y modificar.

Índice de Especialización (IE). Proporciona una indicación aproximada del grado de especialización de cada una de las subclases existentes en un sistema orientado a objetos.

La especialización se puede alcanzar añadiendo o borrando operaciones, o bien por invalidación.

$$IE = [NOI \times nivel] M_{total}$$

En donde *nivel* es el nivel de la jerarquía de clases en que reside la clase, y M_{total} es el número total de métodos para la clase. Cuanto más elevado sea el valor de IE es más probable que la jerarquía de clases tenga clases que no se ajustan a la superclase.

5.4 Métricas orientadas a operaciones

A continuación tres métricas sencillas propuestas por Lorenz y Kidd:

1. ***Tamaño medio de operación (TOavg).*** El número de mensajes enviados por la operación proporciona una alternativa para el tamaño de la operación. A medida que

asciende el número de mensajes enviados por una única operación, es posible que las responsabilidades no hayan sido bien estipuladas dentro de la clase.

2. ***Complejidad de operación (CO).*** La complejidad de una operación se calcula empleando cualquiera de las métricas propuestas para el software convencional. En donde el diseñador debería esforzarse por mantener el valor de CO tan bajo como sea posible.
3. ***Número Medio de Parámetros por operación (NPavg).*** En cuanto sea más grande el número de parámetros de la operación, será más compleja la colaboración entre objetos. En general, NPavg debería de mantenerse tan bajo como sea posible.

5.5 Métricas para pruebas orientadas a objetos

Las métricas de diseño proporcionarán una predicción de la calidad del diseño, también proporcionan una indicación general de la cantidad de esfuerzo de pruebas necesario para aplicarlo en un sistema 00.

Binder[34] sugiere una amplia gamma de métricas de diseño que tienen influencia directa en la "comprobabilidad" de un sistema 00. Las métricas se crean en categorías que reflejan características de diseño importantes:

➢ ***Encapsulamiento:***

Carencia de Cohesión en Métodos (CCM). Cuando sea más alto el valor de CCM, más estados tendrán que ser probados para asegurar que los métodos no den lugar a efectos secundarios.

Porcentaje Público y Protegido (PPP). Los atributos públicos se heredan de otras clases, y por tanto son visibles para esas clases. Los atributos protegidos son una especialización y son privados de alguna subclase específica. Esta métrica indica el porcentaje de atributos de clase que son públicos. Unos valores altos de PPP incrementan la probabilidad de efectos colaterales entre

[34] BINDER, R., "Testing Object Oriented Systems", American Programmer, volumen 7, núm. 4, abril 1994, págs. 22-29.

clases, y por lo tanto es preciso diseñar comprobaciones que aseguren que se descubran estos efectos colaterales.

Acceso Público a Datos miembros (APD). Esta métrica muestra el número de clases (o métodos) que pueden acceder a los atributos de otra clase, violando así el encapsulamiento. Es preciso diseñar comprobaciones que descubran efectos colaterales a niveles altos de APD entre clases.

- ***Herencia***

Número de Clases Raíz (NCR). Es un conjunto de jerarquías de clases distintas (que se describen en el modelo de diseño), en donde será preciso desarrollar conjuntos de pruebas para cada una de las clases raíz, y para la correspondiente jerarquía de clases. A medida que crece NCR crece también el esfuerzo de comprobación.

AdMisión (ADM). la admisión es una indicación de herencia múltiple. Cuando la ADM sea mayor a 1, indica que una clase hereda sus atributos y operaciones de más de una clase raíz. Siempre es preciso evitar un valor de ADM mayor a 1.

5.6 Métricas para proyectos orientados a objetos

El trabajo del administrador de un proyecto es planificar, coordinar, seguir y controlar un proyecto de software.

La primera actividad que desarrolla el administrador del proyecto es la planificación y una de las primeras tareas de la planificación es la estimación. Por tanto, el plan y sus estimaciones de proyecto se revisan después de cada iteración de Análisis Orientado a Objetos(A00), Diseño Orientados a Objetos (D00), e incluso la Programación Orientada a Objetos (POO).

Uno de los problemas a los que se enfrenta un administrador de proyectos durante la planificación es la estimación del tamaño del software, ya que se conoce que el tamaño es directamente proporcional al esfuerzo y la duración. Las métricas 00 siguientes pueden aportar ideas acerca del tamaño del software:

Número de Clases Clave (NCC). Las clases clave se centran directamente en el dominio del negocio para el problema que se estará analizando, y tendrán menor probabilidad de ser implementadas mediante reutilización. Por esta razón, unos valores elevados NCC indican que en nuestro camino encontraremos una cantidad notable de trabajo de desarrollo. Lorenz y Kidd sugieren que entre el 20 y el 40 por ciento de todas las clases de un sistema 00 típico son clases clave. El resto sirve como infraestructura de apoyo (IGU, comunicaciones, bases de datos, etc.).

Número de Subsistemas (NSUB). El número de subsistemas proporciona una idea general de la asignación de recursos, de la planificación, y del esfuerzo global.

Las métricas NCC y NSUB se pueden escoger para otros proyectos OO anteriores, y se pueden relacionar con el esfuerzo invertido en el proyecto y también con las actividades de proceso individuales. Estos datos se pueden utilizar también junto con métricas de diseño, con el objeto de calcular "métricas de productividad" tales como el número medio de clases por desarrollador o el promedio de métodos por persona y mes. En su conjunto, estas métricas se pueden utilizar para estimar el esfuerzo, la duración, el personal y otras informaciones acerca del proyecto para el proyecto actual.

Capítulo 6

Propuesta metodológica orientada a objetos

6.1 La visión orientada a objetos

En primer término usted se preguntará el porqué se propone que al desarrollar un software de computadora se haga con el enfoque de programación orientada a objetos. Bueno esto se debe a que actualmente una de las áreas más mencionadas en la industria y en el ámbito académico es la orientación a objetos, lenguajes como Java, VB.Net, C (por mencionar algunos), utilizan la orientación a objetos ya que promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo en el desarrollo de software, como lo es:

- la falta de portabilidad del código y reusabilidad
- código que es difícil de modificar
- ciclos de desarrollo largos
- y técnicas de codificación

Un lenguaje orientado a objetos ataca estos problemas.

Nosotros como personas nos formamos conceptos desde temprana edad. Cada concepto es una idea particular o una comprensión de nuestro mundo. Los conceptos adquiridos nos permiten sentir y razonar acerca de las cosas en el mundo. A estas cosas a las que se aplican nuestros conceptos se llaman objetos.

Un objeto consta de tres características básicas: *debe estar basado en objetos, basado en clases y capaz de tener herencia de clases.* Muchos lenguajes cumplen uno o dos de estos puntos, pero difícilmente abarca estos tres. La barrera más difícil de cumplir es usualmente la herencia.

También es cierto que el programar es todavía una de las tareas más difíciles emprendidas por la humanidad, se llega a ser un experto a base de talento, creatividad, inteligencia, lógica, habilidad y experiencia, aún cuando se disponga de las mejores herramientas.

El concepto de programación orientada a objetos (OOP) no es nuevo, lenguajes clásicos como SmallTalk se basan en ella. Dado que la OOP se basa en la idea de la existencia de un mundo lleno de objetos y que la resolución del problema se realiza en términos de objetos. Viéndolo de una

manera menos informal se puede decir que la programación orientada a objetos es una nueva forma de pensar acerca de cómo podemos estructurar la información dentro de una computadora.

Muchas veces se le llama a la programación orientada a objetos como un nuevo paradigma de programación, la definición de paradigma abarca un conjunto de teorías, estándares y métodos que juntos representan una forma de organizar nuestro conocimiento. Por ello es que se alude como un paradigma. La visión orientada a objetos sigue con frecuencia el mismo método que aplicamos en la resolución de problemas en la vida diaria. De esta manera, se pueden captar fácilmente las ideas básicas de la programación orientada a objetos.

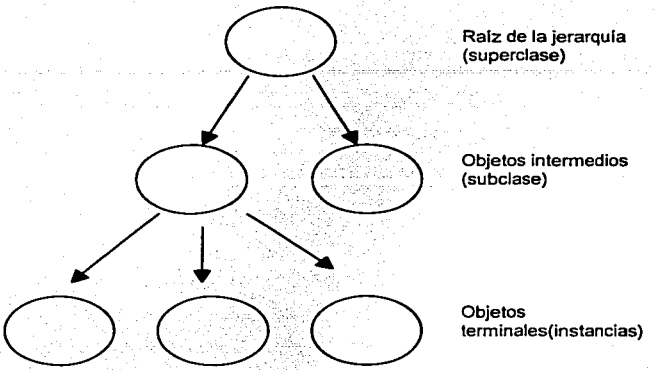
6.2 El objeto y sus características

El elemento fundamental de la OOP es, como su nombre lo indica, el *objeto*. Podemos definir un objeto como *un conjunto complejo de datos y programas que poseen estructura y forman parte de una organización* o dicho de otra manera se define un objeto como cualquier entidad que juega un rol visible al proveer servicios a clientes. Es decir, basa el concepto de objeto en la relación existente entre clientes y proveedores de servicios

Esta definición especifica varias propiedades importantes de los objetos. En primer lugar, un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados. En segundo lugar, cada objeto no es un ente aislado, sino que forma parte de una organización jerárquica o de otro tipo. Cada objeto tiene un papel específico en un programa, y todos los objetos pueden funcionar con otros objetos en maneras definidas.

En principio, los objetos forman siempre una organización jerárquica, en el sentido de que ciertos objetos son superiores a otros de cierto modo, puede que existan jerarquías muy simples o muy complejas En cualquier caso, sea la estructura simple o compleja, podrán distinguirse en ella tres niveles de objetos.

Figura 6.1 Niveles de objetos



Fuente: figura realizada por la autora de esta tesis

1. *La raíz de la jerarquía* la cual trata de un objeto único y especial. Este se caracteriza por estar en el nivel más alto de la estructura y suele recibir un nombre muy genérico, que indica su categoría especial, como por ejemplo objeto madre, raíz o entidad.
2. *Los objetos intermedios* que descienden directamente de la raíz y que a su vez tienen descendientes. Representan conjuntos o clases de objetos, que pueden ser muy generales o muy especializados, según la aplicación. Normalmente denotan al conjunto de objetos que representan.
3. *Los objetos terminales* descienden de una clase o subclase y no tienen descendientes. Suelen llamarse casos particulares, instancias o ítems porque representan los elementos del conjunto representado por la clase o subclase a la que pertenecen.

6.3 Diferencia entre la técnica de programación orientado a objetos y la técnica convencional

La característica única de los sistemas de software desarrollados mediante técnicas convencionales, que los colocan entre los sistemas más complejos desarrollados por la gente, es su alto grado de interconexión que no es más que la dependencia de una parte del código de otra sección de código. Lo contrario a la programación orientada a objetos, la cual nos permite reducir la interdependencia entre los componentes de software ya que concede el desarrollo de sistemas de software reutilizables. Tales componentes pueden crearse y probarse como unidades independientes, aislados de otras partes de una aplicación de software.

6.3.1 Procedimientos, módulos, datos abstractos

Las técnicas orientadas a objetos va de los procedimientos, a los módulos, los tipos de datos abstractos y los objetos. Los *procedimientos* permiten que las tareas que se ejecutaban en forma repetida, o que se realizaban sólo con ligeras variaciones, fueran agrupadas en un lugar y reutilizadas, en vez de duplicar el código varias veces. Este procedimiento otorgó la posibilidad para la ocultación de información, ya que un programador podía escribir un procedimiento, o un conjunto de procedimientos que podían se utilizados por muchos otros. Los demás programadores no necesitaban conocer los detalles exactos de la implantación, sólo necesitaban la interfaz necesaria para que pudiera realizar alguna tarea especifica. Claro que los procedimientos no resolvían todos los problemas ya que podían existir variables locales y globales que no podían utilizarse en todo el sistema. Por ellos los *módulos* pueden considerarse como una técnica mejorada para crear o manejar espacios con nombres, la cual brinda la posibilidad de dividir un espacio de nombres en dos partes, La parte pública que es accesible desde fuera del módulo, mientras que la privada es accesible sólo desde dentro del módulo. Los módulos por sí mismos ofrecen un método efectivo para la ocultación de información, pero no nos permite realizar copias múltiples de las áreas de datos.

Los tipos de *datos abstractos* elimina algunas distinciones para que podamos ver los aspectos comunes entre los objetos. Sin la abstracción, sólo sabríamos que cada cosa es diferente a las demás. Con la abstracción se omiten de manera selectiva varias características distintivas de uno o más objetos, lo que permite concentrarnos en las características que comparten.

Los usuarios pueden crear variables con valores que puedan fluir dentro del conjunto aceptado y actuar sobre dichos valores por medio de la operaciones definidas, esto se refiere a que la programación mediante abstracciones de datos es un enfoque para la solución de problemas en el que la información se oculta en una pequeña parte del programa. Los programadores desarrollamos una serie de tipos de datos abstractos, con ello se entiende que una abstracción es el acto o resultado de eliminar diferencias entre los objetos, lo cual permite un mejor manejo de la complejidad del problema, al permitir suprimir o posponer detalles irrelevantes en cierto momento, y concentrarse más bien, en detalles verdaderamente esenciales. Cada tipo de datos abstracto se puede ver como si tuviera dos caras. Desde el exterior, el usuario de un tipo de datos abstracto ve nada más un conjunto de operaciones que definen el comportamiento de la abstracción. En la otra, el programador que define la abstracción ve las variable de datos que se usan para mantener el estado interno del objeto.

6.3.2 Paso de mensajes

Las técnicas de la programación orientada a objetos agrega algunas ideas nuevas al concepto de tipo de datos abstracto. La primera de ellas es la idea de *paso de mensajes*, esta actividad se inicia con una solicitud hecha a un objeto específico, no por la invocación de una función que use datos específicos. El software convencional concede importancia a la operación, mientras que el orientado a objetos la otorga al valor mismo. Por ejemplo, ¿Qué haría usted?, ¿llamar a una rutina *meter* con una pila y un valor o pide usted a la pila que meta un valor en si misma?. Esto sólo es una cuestión de enfoque. En el paso de mensajes el comportamiento y la respuesta que el mensaje provoca dependerán del objeto que lo recibe. En este caso, *meter* puede significar una cosa para una pila y una idea muy diferente para el controlador de un brazo mecánico, etc. Puesto que los nombres para las operaciones no necesitan ser únicos, pueden usarse formas simples y directas, lo que llevará a un código más legible y comprensible.

6.3.3 Herencia y polimorfismo

Por último la programación orientada a objetos agrega los mecanismos de herencia y polimorfismo. La *herencia* permite que diferentes tipos de datos compartan el mismo código, lo que conduce a una reducción del tamaño del código y a un incremento en la funcionalidad. El *polimorfismo* permite que este código compartido sea adaptado pero con relación a la clase a la que pertenece cada uno, con comportamientos diferentes, por ejemplo, un mensaje "+" a un objeto ENTERO significaría suma, mientras que para un objeto STRING significaría concatenación ("pegar" strings uno seguido al otro). Esto conlleva la habilidad de enviar un mismo mensaje a objetos de clases diferentes. Estos objetos recibirían el mismo mensaje global pero responderían a él de formas diferentes

El enfoque que da la programación orientada a objetos en la independencia de componentes individuales permite un proceso de desarrollo incremental, en el cual unidades de software individuales se diseñan, programan y prueban antes de combinarse en un sistema grande.

En el pasado, la reutilización de software era una meta muy buscada y pocas veces alcanzada. Una razón para ello era la interconexión de la mayoría de los programas, como ya lo mencione anteriormente. Y si por ejemplo en la vida diaria para construir un automóvil, un aparato eléctrico o una computadora por lo general se ensambla cierto número de componentes prefabricados, en vez de partir de uno nuevo. Si eso se hace con las cosas que utilizamos a diario ¿Por qué no hacer lo mismo con los sistemas de software?¿Usted que prefiere empezar de cero o utilizar algo que ya tiene?. Por supuesto la mejor elección es disponer de algo con el cual ya contamos.

6.4 El objeto y sus componentes

Un objeto puede considerarse como una especie de encapsulación del estado (valores de datos) y del comportamiento (operaciones). Así, un objeto se parece en muchas formas a un módulo o a un tipo de datos abstracto dividida en tres partes:

1) RELACIONES
2) PROPIEDADES
3) METODOS

Cada uno de estos componentes desempeña un papel totalmente independiente:

Las *relaciones* entre objetos son los enlaces que permiten a un objeto relacionarse con aquellos que forman parte de la misma organización y están formadas esencialmente por punteros a otros objetos.

Las *propiedades* distinguen un objeto determinado de los restantes que forman parte de la misma organización y tiene valores que dependen de la propiedad de la que se trate. Las propiedades de un objeto pueden ser heredadas a sus descendientes en la organización.

Los *métodos* son las operaciones que pueden realizarse sobre el objeto, que normalmente estarán incorporados en forma de programas (código) que el objeto es capaz de ejecutar y que también pone a disposición de sus descendientes a través de la herencia.

Como hemos visto, cada objeto es una estructura compleja en cuyo interior hay datos y programas, todos ellos relacionados entre sí, como si estuvieran encerrados conjuntamente en una cápsula. Los objetos son inaccesibles, e impiden que otros objetos, usuarios, o incluso los programadores conozcan cómo está distribuida la información o qué información hay disponible. Esto no quiere decir, sin embargo, que sea imposible conocer lo necesario respecto a un objeto y a lo que contiene. Si así fuera no se podría hacer gran cosa con él. Lo que sucede es que las peticiones de información a un objeto. deben realizarse a través de mensajes dirigidos a él, con la orden de realizar la operación pertinente. La respuesta a estas ordenes será la información requerida, siempre que el objeto considere que quien envía el mensaje está autorizado para obtenerla.

El hecho de que cada objeto sea una cápsula facilita enormemente que un objeto determinado pueda ser transportado a otro punto de la organización, o incluso a otra organización totalmente diferente que precise de él. Si el objeto ha sido bien construido, sus métodos seguirán

funcionando en un nuevo entorno sin problemas. Esta cualidad hace que la OOP sea muy apta para la reutilización de programas.

Existen muchísimos beneficios el utilizar el desarrollo de OOP ya que día a día los costos del Hardware decrecen. Así surgen nuevas áreas de aplicación cotidianamente: procesamiento de imágenes y sonido, bases de datos, automatización de oficinas, etc. Lamentablemente, los costos de producción de software siguen aumentando, el mantenimiento y la modificación de sistemas complejos suele ser una tarea trabajosa, cada aplicación, (aunque tenga aspectos similares a otra) suele encararse como un proyecto nuevo, etc.

Todos estos problemas aún no han sido solucionados en forma completa. Pero como los objetos son *portables* mientras que la herencia permite la reusabilidad del código orientado a objetos, es más sencillo modificar código existente porque los objetos no interaccionan excepto a través de mensajes, en consecuencia un cambio en la codificación de un objeto no afectará la operación con otro objeto siempre que los métodos respectivos permanezcan intactos.

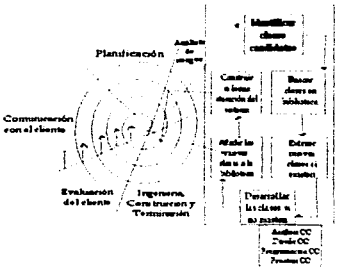
La introducción de tecnología de objetos como una herramienta conceptual para analizar, diseñar e implementar aplicaciones permite obtener aplicaciones más modificables, fácilmente extendibles y a partir de componentes reusables. Esta reusabilidad del código disminuye el tiempo que se utiliza en el desarrollo y hace que el desarrollo del software sea mas intuitivo porque la gente piensa naturalmente en términos de objetos más que en términos de algoritmos de software.

Una vez revisado en forma general la visión que se le da al lenguaje orientado a objetos y varios términos importantes que se manejan, ahora nuestro interés es dar seguimiento al ciclo de vida en el desarrollo de un sistema de software orientado a objetos que se muestra a continuación.

6.5 Ciclo de vida propuesto en el desarrollo de software orientado a objetos

El ciclo de vida que se muestra a continuación ya se había especificado en el capitulo uno de esta tesis, pero el motivo por el cual ha sido retomado, es que la misma secuencia que se muestra en la figura será el de éste capítulo (más no en el contenido).

Figura 6.2 Modelo de proceso OO



Fuente: S. PRESSMAN, Roger , Ingeniería de software , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

De acuerdo con esta figura esta propuesta va ha ser dividida en cinco fases,:

1. Comunicación con el cliente
2. Planificación
3. Análisis de riesgos
4. Ingeniería construcción y terminación
5. Evaluación del cliente

De las cuales se hará su respectiva división donde así se requiera.

De las cuales se hará su respectiva división donde así se requiera.

6.5.1 Primera fase:

Comunicación con el cliente

Según vimos existen diversos modelos de ciclo de vida para el desarrollo de software, pero en todos ellos se puede observar la existencia de una fase denominada comunicación con el cliente o análisis de requisitos del cliente.

El principal problema que se nos presenta es que, en estos momentos iniciales, es difícil tener una idea clara o, al menos, es difícil llegar a expresar cuáles son los requisitos del sistema o del software, y llegar a comprender en su totalidad la función que el software debe realizar. Por esto, algunos de los modelos de ciclo de vida proponen enfoques cíclicos de refinamiento de los requisitos o incluso de todo el proceso de desarrollo de software, tal como se muestra en el ciclo de vida de la programación orientada a objetos.

El análisis de requisitos del sistema constituye el primer intento de comprender cuál va a ser la función y ámbito de información de un nuevo proyecto. El objetivo es conseguir representar un sistema en su totalidad, incluyendo hardware, software y personas, mostrando la relación entre los diversos componentes y sin entrar en la estructura interna de los mismos. En algunos casos se nos plantearán diferentes posibilidades y tendremos que realizar un estudio de cada una de ellas.

La forma de especificar un sistema tiene una gran influencia en la calidad del software que se desea implementar. Tradicionalmente los ingenieros de software han venido trabajando con especificaciones incompletas, inconsistentes o erróneas, lo que lleva a la confusión y a la frustración en todas las etapas del ciclo de vida. Como consecuencia de esto, la calidad y la completitud del software disminuyen.

Los esfuerzos realizados en esta fase producen beneficios en las fases posteriores. Sin embargo se nos pueden plantear las siguientes dudas:

- **¿Cuánto tiempo debe dedicarse al análisis del sistema?** Dependerá del tamaño y la complejidad de la aplicación. En la mayoría de los proyectos la mayor parte del esfuerzo se va en la codificación, precisamente por la dificultad de realizar la codificación cuando no se ha hecho un buen análisis previo.
- **¿Quién debe hacerlo?** La respuesta es fácil: el analista de sistemas. Este debe ser una persona con buena formación técnica y con experiencia. No obstante, el analista no trabaja de forma aislada sino en estrecho contacto con el personal del cliente como del que desarrolla el software.
- **¿Por qué es una tarea tan difícil?** Básicamente porque consiste en la traducción de unas ideas vagas de necesidades de software en un conjunto concreto de funciones y restricciones. Además el analista debe extraer información dialogando con muchas personas y cada una de ellas se expresará de una forma distinta, tendrá conocimientos informáticos y técnicos distintos, y tendrá unas necesidades y una idea del proyecto muy particulares.

El papel del analista de sistemas es el de definir los elementos de un sistema informático dentro del entorno del sistema en que va a ser usado. Hay que identificar las funciones del sistema y asignarlas a alguno de sus componentes. Para ello, el analista de sistemas parte de los objetivos y restricciones definidos por el usuario y realiza una representación de la función del sistema, de la información que maneja, de sus interfaces, del rendimiento y las restricciones del mismo.

Como en la mayoría de los casos, el proyecto empieza con un concepto más bien vago y ambiguo de cuál es la función deseada. Entonces el analista debe delimitar el sistema, indicando el funcionamiento y el rendimiento deseados. Por ejemplo, en el caso de un sistema de control, no basta con decir algo así como *'el sistema debe reaccionar rápidamente en caso de que la señal de entrada supere los límites de seguridad'* sino que hay que definir cuáles son los límites de seguridad de la señal de entrada, en cuánto tiempo debe producirse la reacción y cómo ha de ser esta reacción.

Una vez que se ha logrado **delimitar la función, el ámbito de información, las restricciones, el rendimiento y las interfaces del sistema, el analista debe proceder a la asignación de funciones**. Las funciones del sistema deben de ser asignadas a alguno de sus componentes (ya sean éstos software, hardware o personas). El analista debe estudiar varias opciones de asignación (considerando, por ejemplo, la posibilidad de automatizar o no alguna de estas funciones), teniendo en cuenta las ventajas e inconvenientes de cada una de ellas (en cuanto a costes de desarrollo, funcionamiento y fiabilidad) y decidirse por una de ellas, o bien presentar un estudio razonado de las opciones a quienes tengan que tomar la decisión.

Para explicar todo lo anterior podemos poner el siguiente ejemplo para una mejor comprensión:

Ejemplo:

"Especificación del sistema de clasificación de paquetes".

El sistema de clasificación de paquetes debe realizarse de forma que los paquetes que se mueven a lo largo de una cinta transportadora, sean identificados (para lo cual van provistos de un código numérico) y clasificados en alguno de los seis contenedores situados al final de la cinta. Los paquetes de cada tipo aparecen en la cinta siguiendo una distribución aleatoria y están espaciados de manera uniforme. La velocidad de la cinta debe ser tan alta como sea posible; como mínimo el sistema debe ser capaz de clasificar 10 paquetes por minuto. La carga de paquetes al principio de la cinta puede realizarse a una velocidad máxima de 20 paquetes por minuto. El sistema debe funcionar las 24 horas del día, siete días a la semana.

Función del sistema.

Realizar la clasificación de paquetes que llegan por una cinta transportadora en seis compartimentos distintos, dependiendo del tipo de cada paquete.

Información que se maneja.

Los paquetes disponen de un código numérico de identificación.

Debe existir una tabla o algoritmo que asigne a cada número de paquete el contenedor donde debe ser clasificado.

Interfaces del sistema.

El sistema de clasificación se relaciona con otros dos sistemas. Por un lado tenemos la cinta transportadora. Parece conveniente que el sistema de clasificación pueda parar el funcionamiento de la cinta o del sistema de carga de paquetes en la cinta, en caso de que no pueda realizar la clasificación (por ejemplo si se produce una avería). Por otro lado, el sistema deposita paquetes en los contenedores, pero no se establece ningún mecanismo de vaciado o sustitución de los contenedores si se llenan. El sistema debería ordenar la sustitución o vaciado del contenedor o esperar mientras un contenedor esté lleno.

Como la descripción realizada por el cliente no establece los mecanismos para solventar estas dos situaciones estos detalles deben ser discutidos con el cliente.

Rendimiento.

El sistema debe ser capaz de clasificar al menos 10 paquetes por minuto. No es necesario que el sistema sea capaz de clasificar más de 20 paquetes por minuto.

Restricciones.

El sistema debe tener un funcionamiento continuo. Por tanto, debemos evitar la parada del sistema incluso en el caso de que para alguno de los componentes del mismo se averien.

El documento no indica restricciones sobre la eficacia del sistema, es decir, sobre cuál es el porcentaje máximo que se puede tolerar de paquetes que pueden ser clasificados de forma errónea. Estos detalles también deben ser aclarados con el cliente.

Asignación de funciones.

Podemos considerar tres asignaciones posibles:

Asignación 1.

Esta asignación propone una solución manual para implementar el sistema. Los recursos que utiliza son básicamente personas, y se requiere además algo de documentación, definiendo las características del puesto de trabajo y del sistema de turnos y una tabla que sirva al operador para relacionar los códigos de identificación de los paquetes con el contenedor donde deben ser depositados.

La inversión necesaria para poner en marcha este sistema es mínima, pero requiere una gran cantidad de mano de obra (varios turnos de trabajo y operadores de guardia) con lo que los costes de funcionamiento serán elevados. Además hay que tener en cuenta que lo rutinario del trabajo provocará una falta de motivación en los operarios, lo que a la larga se acabará traduciendo en un mayor ausentismo laboral. Todos estos factores deben de tenerse en cuenta a la hora de elegir esta u otra opción.

Asignación 2.

En este caso, la solución es automatizada. Los recursos que se utilizan son: hardware (el lector de códigos de barras, el controlador, el mecanismo de distribución), software (para el lector de códigos de barras y el controlador, y una base de datos que permita asignar a cada código su contenedor) y personas (si en caso de avería la distribución se va a hacer manualmente). Cualquiera de los elementos hardware y software tendrán la correspondiente documentación sobre cómo han sido construidos y un manual de usuario.

Aquí si hay que realizar una cierta inversión, para comprar o desarrollar los componentes del sistema, pero los costes de funcionamiento serán sin duda menores (sólo el consumo de energía

eléctrica). Hay que tener en cuenta que el uso de dispositivos mecánicos (el mecanismo de distribución) va a introducir unos costes de mantenimiento y paradas por avería o mantenimiento con una cierta frecuencia.

Asignación 3.

Los recursos que utilizamos aquí son: hardware (el lector, el brazo robot), software (el del lector, el del robot, incluyendo la tabla o algoritmo de clasificación), la documentación y manuales correspondientes a estos elementos.

En este caso la inversión inicial es, sin duda, la más elevada. Los costes de funcionamiento son bajos pero hay que considerar también el coste de mantenimiento del robot, que posiblemente tenga que ser realizado por personal especializado. Los únicos motivos que nos harían decidir por esta opción en vez de la anterior vendrían dados por una mayor velocidad, un menor número de errores o unas menores necesidades de mantenimiento o frecuencia de averías.

Por otra parte esta solución puede tener problemas de viabilidad (si no encontramos un brazo robot que sea capaz de atrapar los paquetes según pasan por la cinta).

Además de las tres opciones propuestas en este ejemplo, el ingeniero de sistemas debe estudiar si existe ya un producto comercial que realice la función requerida para el sistema o si alguna de las partes del mismo pueden ser adquiridas a un tercero. Aparte de considerar el precio de estos productos habrá que tener también en cuenta los costes del mantenimiento y el riesgo que se asocia al depender de una tecnología que no es propia (¿es la empresa proveedora estable?, ¿cuál es la calidad de sus productos?, etc.) valorando todo esto frente a los riesgos asociados a realizar el desarrollo nosotros mismos.

La labor del ingeniero o analista de sistemas consiste, en definitiva, en asignar a cada elemento del sistema un ámbito de funcionamiento y de rendimiento. Después, el ingeniero del software se encargará de refinar este ámbito para el componente software del sistema y de producir un elemento funcional, que sea capaz de ser integrado con el resto de los elementos del sistema.

En base a este ejemplo podemos definir varias tareas del análisis de sistemas.

Primera: Hacer una completa identificación de las necesidades del cliente.

Para identificar las necesidades, el analista del sistema debe reunirse con el cliente o con su representante. Juntos, proceden a definir los objetivos del sistemas, la información que se va a suministrar, la información que se va a obtener, las funciones y el rendimiento requerido.

El analista debe ser capaz de distinguir entre lo que *necesita* el cliente (lo que es para él imprescindible), y lo que *quiere* el cliente (aquello que le sería útil pero no imprescindible).

Segunda: Hacer una evaluación de la viabilidad del sistema.

Los recursos son siempre limitados: existen restricciones sobre el número de personas que se pueden dedicar (especialmente si se trata de personal del cliente), sobre cuanto dinero nos podemos gastar en el proyecto (si la inversión necesaria para el desarrollo es demasiado alta el sistema no compensará los ahorros que se produzcan con su uso o viceversa) y sobre los tiempos de entrega (nadie compra software a cinco años vista, además, según pasa el tiempo aumentan las posibilidades de que cambien los requisitos).

Por esto, es conveniente estudiar la viabilidad del proyecto lo antes posible, puesto que así se pueden ahorrar meses de esfuerzos en el desarrollo de un proyecto que al final se muestre como un proyecto no viable

Nos centraremos fundamentalmente en tres aspectos:

- **Viabilidad económica.** Consiste en comparar los beneficios futuros de la utilización del sistema con los costes de su desarrollo. La justificación económica es normalmente la principal consideración a la hora de decidir realizar o no cualquier proyecto. Por esto, aparte de decidir la viabilidad o no viabilidad se necesita también un análisis económico completo (no sólo si va a producir beneficios sino qué beneficios va a producir, a qué plazo, etc.).

- **Viabilidad técnica.** Consiste en determinar si es posible desarrollar o no el producto, teniendo en cuenta sus restricciones, basándonos en los recursos humanos y técnicos a nuestro alcance. Como los objetivos, funciones y rendimiento son confusos al inicio del proyecto, cualquier cosa puede parecer inicialmente viable. Debido a esto, es conveniente revisar el estudio de viabilidad técnica una vez que las especificaciones estén más claras. Además, el análisis de viabilidad técnica debe ser completado con un estudio técnico del sistema en proyecto, que permita determinar las características técnicas del nuevo sistema y qué mejoras introduce sobre el proceso actual.

- **Viabilidad legal.** Consiste en determinar si el proyecto infringe alguna disposición legal sobre las normas de seguridad en el trabajo, de calidad de los productos, las leyes de Copyright si se van a utilizar componentes comprados a terceros para integrarlos en el sistema o basarnos en ellos para el desarrollo del producto software, y otras.

Tercero: Realizar un análisis económico.

El análisis económico del proyecto consistirá en señalar los costes de desarrollo del proyecto (inversiones en equipos, horas-hombre necesarias en las fases de análisis, diseño y codificación) y compararlos con los beneficios que producirá una vez desarrollado (disminución del tiempo necesario para realizar determinadas tareas, etc.).

Los costes de desarrollo pueden ser cuantificados fácilmente. Sin embargo, suele ser más difícil determinar los beneficios futuros que producirá el proyecto una vez listo para ser usado.

La única forma de demostrar estos beneficios será comparando los modos de trabajo con el nuevo sistema con los modos de trabajo actuales. El nuevo sistema, cambiará los modos de trabajo de la empresa u organización en la que se instale de forma que se producirá:

- una disminución del tiempo necesario para realizar determinadas tareas.
- una menor necesidad de mano de obra para realizar el trabajo actual.
- un aumento de productividad, que permitirá aumentar la producción con la mano de obra actual.

Cualquiera de los tres puntos anteriores puede ser evaluado cuantitativamente, de forma que se puede determinar el ahorro que se producirá con la puesta en marcha del nuevo sistema.

Cuarto: Realizar un análisis técnico del proyecto.

Con el análisis técnico se pretende estudiar las características técnicas del nuevo sistema: capacidad, rendimiento, fiabilidad, seguridad, etc., de forma que se complemente el análisis de coste-beneficio (análisis económico) con las mejoras técnicas que pueda proporcionar el sistema a los modos de trabajo de la empresa u organización donde se implante.

Para hacer un buen análisis técnico tendremos que modelar el sistema de alguna forma, y realizar un estudio a fondo de las características del modelo propuesto o bien realizar algún tipo de simulación con el modelo.

Los resultados del análisis técnico son otro de los criterios que permitirán decidir si seguir o no con el proyecto. Si el riesgo técnico es alto, o si el modelo o las simulaciones muestran que el sistema en proyecto no va a conseguir substanciales mejoras sobre el sistema actual, podemos cancelar el proyecto.

Quinto: Asignación de funciones a cada uno de los elementos del sistema.

Como ya habíamos visto, para cada elemento del sistema puede haber una o varias asignaciones posibles. Es necesario estudiar cada una de las alternativas desde el punto de vista económico y técnico y elegir cuál es la más adecuada.

Para elegir entre las distintas alternativas debemos fijar una serie de parámetros de evaluación. Normalmente los parámetros de evaluación serán de orden económico (coste de las inversiones, ahorro que se producirá con el sistema nuevo), pero también pueden tenerse en cuenta parámetros relacionados con la fiabilidad del sistema, su capacidad, su rendimiento o la mejora de calidad de los productos, si se trata de un sistema de producción. Hay que ordenar cada uno de estos parámetros de acuerdo a su importancia, lo que dependerá de los objetivos generales de cada proyecto, y ver cuál de las distintas alternativas obtiene una mejor evaluación de acuerdo con los

parámetros establecidos. Cuando dos o más alternativas satisfagan los parámetros de evaluación principales, optaremos entre ellas observando los parámetros de segundo orden y así sucesivamente.

Aquí cabe señalar que esta primer fase siguiendo tal cual se ha descrito anteriormente, no sólo puede funcionar para el desarrollo de un software orientado a objetos sino para un software convencional, ya que desde mi punto de vista abarca todo los aspectos necesarios para un buen estudio de software a realizar.

6.5.2 Segunda fase:

Planificación

Es la etapa en la que el desarrollador de software determina si el proyecto es o no factible de realizar, se determinan tiempos y costos aproximados, estableciendo así la ruta crítica de cada actividad. Este punto es apropiado porque la falta de planeación de un sistema es la causa principal de retrasos en programación, incremento de costos, poca calidad, y altos costos de mantenimiento en los desarrollos de productos de software.

Con frecuencia se dice que es imposible realizar una planeación inicial, porque la información precisa sobre las metas del proyecto, necesidades del cliente y restricciones del producto, no se conocen a fondo al comenzar el proyecto, sin embargo, uno de los principales propósitos de esta fase es aclarar los objetivos, problemas o necesidades y restricciones.

La dificultad de la planeación no debe desalentarnos como desarrolladores a efectuar tan importante actividad, claro, en este punto el que toma la última palabra es el que va a desarrollar el proyecto, siempre es mejor hablar claro desde el momento que solicitan este servicio, puede que tengamos alguna ocupación o cualquier otra causa que nos pueda restringir como para invertir nuestro tiempo en el desarrollo de un proyecto de software. En cualquiera de estos casos se debe actuar con honestidad y por que no, recomendar a alguien este trabajo. Recuerde que la decisión es suya.

Ahora que si ya ha decidido realizar este proyecto, lo que puedo sugerirle para lograr un proyecto bien ejecutado es pasar por tres etapas básicas para crear una planificación software. Primero se estima el tamaño del producto, luego se estima el esfuerzo necesario para construir un producto con ese tamaño del producto, y por último se realiza una planificación, basándose en la estimación del esfuerzo. Ya que el desarrollar una estimación incorrecta reduce la eficiencia en el desarrollo.

Claro que, también implican ciertas actividades como lo son:

- Determinar el número de personas que deben participar en el equipo del proyecto, qué habilidades técnicas son necesarias, cuándo aumentar el número de personas y quién participará.
- Decidir cómo organizar el equipo.
- Desarrollar una gestión de riesgos.
- Tomar decisiones estratégicas, tales como controlar y decidir si hay que comprar o crear algunas partes del producto.

También es muy importante darle seguimiento una vez que se ha planificado el proyecto, hay que comprobar que se está siguiendo el plan previsto: que satisface sus objetivos de planificación , coste y calidad. Se deben incluir listas de tareas, reuniones e informes sobre el estado, revisiones de presupuestos, etc.

6.5.3 Tercera fase:

Análisis de riesgos

Bueno, cuando uno trabaja un proyecto se debe establecer una base o una trayectoria en cuanto a la realización de trabajo que hay que desempeñar. Lo ideal es seguir ese plan al pie de la letra para no tener demora al entregar un software, básicamente a esto se refiere con planificación y análisis de riesgos.

La planeación de proyectos se considera por tanto como un método y un conjunto de técnicas basadas en los principios de dirección que han sido aceptados por todos los integrantes del proyecto OO y que se emplean para planear, estimar y controlar actividades de trabajo para alcanzar un resultado final deseado a tiempo, dentro del presupuesto y conforme a una especificación.

Al no seguir un plan de trabajo puede que tengamos diferentes percances que puedan atrasar nuestro proyecto, por eso es muy importante que cuando se trabaje en equipo exista una comunicación primordial, ya que puede que el trabajo de otro o el de nosotros pueda repercutir en el desempeño de diversas actividades en el desarrollo de software. Si un proyecto no sigue su trayectoria entonces las probabilidades de que no alcance su meta o por lo menos no se va a poder alcanzar todas las metas que se propusieron en el proyecto.

Los proyectos que no siguen método alguno tienen el riesgo de fracasar a menudo por las razones que siguen:

- Nadie está a cargo.
- El plan del proyecto carece de estructura.
- El plan del proyecto carece de detalle.
- El presupuesto del proyecto es inferior al requerido.
- Los recursos asignados son insuficientes.

Quienquiera que haya trabajado en un proyecto ha tenido indudablemente experiencias que conforman todas estas causas.

También es muy importante o necesario que con frecuencia se venda la idea del proyecto a "los de arriba", así como al personal y a los grupos de supervisión. Por lo tanto, el panorama del proyecto deberá ser un planteamiento persuasivo. No hay que olvidar que, para vender eficazmente una idea, producto o servicio, debe creerse en él y tener confianza en su valor. Al mismo tiempo deberá ser realista.

6.5.4 Cuarta fase:

Ingeniería, construcción y terminación

6.5.4.1 Identificar clases candidatas:

Como ya lo había mencionado una de las gran ventajas que posee la programación orientada a objetos es la reutilización de código, ya que no es necesario escribir de nuevo clases de objetos que cubran los requisitos que estamos buscando, ya que cuando se escribe un programa en un lenguaje orientado a objetos, uno no define objetos individuales sino que define clases de objetos. Cuando su programa se esta ejecutando, los objetos se crean desde estas clases y se usan conforme se van necesitando. La tarea de un buen programador es crear un conjunto de clases que sea adecuado para llevar a cabo lo que su programa requiere, sin partir de cero por supuesto, ya que las clases engloban todas las características particulares de un objeto. Así es que si existe algún programa ya realizado con el enfoque orientado a objetos, de ese mismo programa se podrían utilizarlas clases de objetos para el desarrollo de un proyecto nuevo . Esto quiere decir que el primer paso a seguir es identificar clases candidatas ya disponibles en bibliotecas asociadas con cualquier lenguaje de programación que se esté usando para la aplicación, y así podamos utilizarla(s) para avanzar más rápido en el proyecto de software que deseemos realizar aprovechando la gran ventaja de reusabilidad.

Cabe destacar que la estructura de clases se espera que sea fácilmente reutilizable cuando sea necesario. Las clases pueden ser agrupadas para formar bibliotecas, y al desarrollar un nuevo sistema cada vez se cuenta con una mayor cantidad de ellas. La idea es tomar ese conjunto de clases ya depuradas y construir a partir de ellas nuevas soluciones, modificando la jerarquía de clases según se requiera.

6.5.4.2 Buscar clases en biblioteca:

En primer lugar ¿a que se refiere con biblioteca?, el término biblioteca se refiere a que existe un grupo de clases diseñadas para su uso con otros programas, como lo son palabras clave y operadores que son reconocidos por el lenguaje que se esta utilizando. Por ejemplo el lenguaje Java maneja tareas como funciones matemáticas, gráficos, sonido, interacción con el usuario, manejo de textos y conectividad de redes.

Por ejemplo:

```
Class Dragon {
String color;
String sex;
Boolean hungry;
}
```

Este ejemplo muestra en la primera línea una clase denominada Dragón en el lenguaje Java, las siguientes tres líneas son atributos que describen a la clase de objetos con otros. Dos de ellas, color y sex, pueden contener objetos String. Una cadena de texto es un término general que significa un grupo de caracteres, pero en Java *un objeto String se crea mediante una de las clases estándar en la biblioteca de clases de Java.* La clase String se usa para guardado de texto y funciones de manejo de texto. El tercer objeto, hungry, es una variable boolean que sólo puede guardar cierto o falso. En este caso se usa para saber si Dragon se encuentra hambriento o no.

Este ejemplo trata de describir que al crear un objeto, esté se crea desde estas clases y las vamos a estar utilizando conforme las vayamos utilizando para describir las características particulares de este objeto, al crear una clase utilizamos palabras claves para nosotros como programadores que son fáciles de entender en el lenguaje que estamos utilizando, para que así el programa las identifique como alguna función o descripción en especifico, es como utilizar un lenguaje universal que puede ser entendidos por todos para no tener ningún problema con la descripción de alguna de las clases. Si ya existen, utilizarlas, y si no, la tarea del programador al crear alguna clase consistirá en crear alguna clase sencilla para crear objetos a partir de las clases

estándar del lenguaje que este utilizando y manejar su interacción. Con este ejemplo se puede ver claramente a que se refiere el buscar clases en biblioteca.

6.5.4.3 Extraer nuevas clases si existen:

Este punto es bastante claro, lo que trata de decir es que al buscar clases candidatas e identificar si nos sirven para crear algún proyecto nuevo, simplemente las utilicemos para ahorrarnos tiempo y aprovechar las ventajas que nos brinda este enfoque orientado a objetos.

En caso contrario a este punto, el paso a seguir es desarrollar las clases si no existen con las siguientes etapas que se muestran a continuación:

6.5.4.4 Desarrollar las clases si no existen:

6.5.4.4.1 Análisis OO

Como ya vimos, el objetivo del análisis es generar modelos del dominio del problema, es decir, entender, haciendo abstracción hacia objetos, en qué consiste el problema mismo y el del diseño es extender ese entendimiento con otras clases de otros objetos, que permitan poner en práctica un sistema en un ambiente de cómputo.

La Ingeniería de Software define métodos para la construcción de sistemas de cómputo complejos. Estos métodos son técnicas claramente definidas, fácilmente enseñadas, repetidas y utilizadas por grupos de personas que cooperan entre sí para desarrollar software. Esto implica que el enfoque de la Ingeniería de Software no sea personal. Prácticamente hoy en día cualquier sistema de software que se desarrolla tiene un cierto grado de complejidad.

Aún en nuestros días, al hablar de programación puede pensarse que se trata de un acto personal o individual, pero cuando se hace el desarrollado de software en grupos, estamos hablando de Ingeniería de Software ya que necesitamos trabajar de una forma más organizada y con mejores técnicas, ya que dos cabezas piensan mejor que una.

En la Ingeniería de Software se propone el proceso que hay que seguir para desarrollar software donde su último enfoque es hacerlo con calidad. Pero, ¿qué es el proceso de software? El proceso de software se define cuando señalamos el orden de pasos y actividades que hay que realizar para desarrollar sistemas de software. Las definiciones más recientes tratan de incluir otros aspectos, especificando ¿qué hacer?, ¿cuáles pasos seguir?, ¿y quién debe hacerlos?. Con esto introduce el concepto de diferentes roles para el proceso de software y de quienes se responsabilizan de la realización de las actividades. El proceso de software define el orden especificando el qué se va a hacer, el quién lo va a hacer y el cuándo se va a hacer. Otro aspecto importante dentro de la definición del proceso de software, es que establece criterios para poder decir cuándo hemos terminado una actividad y podemos empezar otra.

La parte de análisis tiene como objetivo proporcionar el modelo del sistema enfocándose en su comportamiento no en su estructura. Si nos enfocamos en el dominio del problema o sea en la funcionalidad que el usuario desea. Tratamos de entender mejor sus requerimientos pasando la estructura a un segundo plano, y comenzamos a modelar las funciones del sistema, es decir entendemos en qué consiste el problema.

En lo personal para generar una documentación del análisis o realizar un buen análisis, se deberán crear tres productos, los cuales son:

Escenarios. El concepto de escenario fue adquirido de Rumbaugh para reforzar el análisis. Para cada requerimiento, (que después Jacobson los llamaría Casos de Uso), se define su escenario. Un Caso de Uso es como una función completa que el usuario espera obtener del sistema, desde que empieza por alguna interacción del usuario con el sistema, hasta que éste realiza una operación completa. Los Casos de Uso sirven para capturar, describir y denotar una función, un requerimiento funcional completo, que se espera del sistema. Los escenarios son pasos que el sistema debe realizar para ejecutar algún Caso de Uso. Los escenarios primarios son aquellos pasos que el sistema tendrá que realizar para satisfacer un requerimiento de los más importantes; los secundarios, por lo general, son los casos que suceden sólo de vez en cuando (por ejemplo cuando el usuario cancela alguna operación).

La técnica de los escenarios fomenta un enfoque experimental hacia el diseño. Puesto que las tarjetas se modifican con tanta facilidad que pueden evaluarse rápidamente diseños diferentes.

Tarjetas CRC (Class Responsability and Cooperation). Esta es una técnica para hacer análisis basado en tarjetas de papel que usan las secretarias para hacer anotaciones. Se usan como apoyo cuando estamos definiendo con más detalle el escenario de un Caso de Uso o requerimiento del usuario, (por ejemplo: hacer una compra, su escenario podría ser: Generar una Factura, Actualizar Inventarios, Actualizar Contabilidad).

En el modelado de objetos, el escenario se convierte en una colección de clases. Se identifica: qué clases del sistema van a participar en el escenario, cuáles serán sus responsabilidades (lo que sabe hacer, es decir sus métodos) y la cooperación de esta clase con otras para poder realizar dichas responsabilidades, anotándose el nombre de aquélla clase o clases con la que se coopera enviando mensajes para que la responsabilidad se realice.

Entonces las tarjetas CRC nos ayudan a analizar los escenarios identificando las clases que aparecen en el dominio del problema, con la responsabilidad representamos los métodos asociados a la clase y con la cooperación encontramos las asociaciones y relaciones entre las clases.

Los diagramas de objetos y clases. Sirven para documentar el análisis (el dominio del problema). Son las herramienta de documentación que propuso Booch, mencionadas previamente.. El diagrama de clases representa la vista estática del sistema y el de objetos la dinámica.

Las actividades que conforman este punto son la planeación de escenarios y la identificación de funciones primarias del sistema, haciendo escenarios para cada una. Como ya se mencionó, los escenarios sirven para documentar los requerimientos del sistema, cada escenario tiene sus productos y cuando se tengan todos los escenarios para todas las funciones o casos de uso básicos del sistema, la fase de análisis se da por terminada.

Con este método se puede obtener un análisis y un método mucho más claro. Si se realiza de manera conjunta, tal cual se menciona anteriormente.

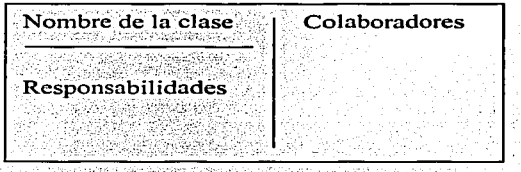
6.5.4.4.2 Diseño OO

El punto más importante de la programación orientada a objetos es la técnica de diseño dirigido por responsabilidades. Cuando hacemos que los objetos se responsabilicen de acciones específicas, esperamos de ellas cierto comportamiento, al menos cuando se obedecen las reglas, y en general se espera un comportamiento razonable aún cuando se desobedezcan. El diseño dirigido

por responsabilidades resulta útil cuando se está diseñando una aplicación orientada a objetos ya que si ha de ocurrir alguna acción especifica, alguien tiene que responsabilizarse de desempeñarla. Ninguna acción puede tener lugar sin que haya un agente que la ejecute. De este modo un buen diseño orientado a objetos radica en establecer primero quién es el responsable de cada acción que se va a ejecutar.

El uso de fichas para representar las clases individuales es una técnica muy útil para llevar a cabo físicamente tanto la necesidad de deslindar responsabilidades como la necesidad de asignar una responsabilidad a cada acción. Tales fichas se conocen como tarjetas CRC (figura 6.1) por que están divididas en tres partes: clase, responsabilidad y colaboración.

Figura 6.1 Tarjeta CRC



Fuente: Ilustración realizada por la autora de esta tesis

Cada clase de nuestro diseño es extremadamente importante, ya que la selección de nombres significativos crean el vocabulario con el cual se formulará el diseño, si se utiliza algún nombre raro, puede hacer que la operación más sencilla suene intimidante o complicada, por ejemplo en un banco, si en vez de llamar a un "lector de tarjetas" se llamará "procesador de identificación de usuarios" podría resultar confuso. Los nombres deben ser consistentes, significativos, de preferencia cortos. En muchos casos los programadores destinan gran tiempo a encontrar un conjunto correcto de términos que describan los objetos que se están manejando. La selección de nombres adecuados en el proceso de diseño simplifica y facilita mucho los pasos siguientes.

Mi recomendación es utilizar nombres que sean de fácil pronunciación, usé mayúsculas o subrayado, si quiere utilizar abreviaturas procure que sean lo más claras posibles, no utilice dígitos

ya que una "O" puede confundirse con un "0", si maneja valores booleanos nombre variables y funciones que describan la interpretación de un valor falso de uno verdadero, por ejemplo ListaImpresion o EstadoImpresion resulta menos impreciso el último término. Y por último no utilice acentos, además de que pierde tiempo puede no recordar si utilizó al principio acento o no.

Debajo del nombre de la clase en la tarjeta CRC se ennumeran las responsabilidades de la clase. Éstas describen el problema que se va a resolver. Las frases deben expresarse mediante frases cortas por ejemplo, describir *qué* se debe hacer y deben evitar las especificaciones de *cómo* se va a llevar a cabo la tarea. De preferencia utilice frases que contenga algún verbo especifico como los que se señalaron anteriormente. Esta sección solo debe describir los detalles más importantes, dejando la información no tan importante para que se complete más tarde. Las medidas de las tarjetas que son de diez por quince centímetros imponen un buen límite a la complejidad. Si se desarrolla una clase que abarque mucho más que este espacio puede que sea muy compleja y la solución más sencilla es trasladar responsabilidades a otra parte o dividir la tarea entre dos o más objetos separados. El tiempo que se dedique a encontrar responsabilidades más cortas se ve recompensado más tarde en el proceso de desarrollo.

La tercera parte en la tarjeta es una lista de los colaboradores. Pocos objetos puede realizar operaciones solos, casi todos tienen relación con otros, ya sea como prestadores o como solicitantes de un servicio o de un recurso. La lista de colaboradores debe incluir todas las clases de las que deba estar al tanto la clase que se esta describiendo, además deberá incluir las clases que presten los servicios para que cumpla las responsabilidades dicha clase. Podria incluir también las clases que requieren de los servicios de esta clase, aunque esto puede no ser tan necesario.

Antes de utilizar las tarjetas CRC se debe crear las clases correspondientes para poder elaborar una aplicación orientada a objetos. Las clases pueden tener diferentes responsabilidades, por ejemplo: *clases manejadores de datos o de estado* su principal responsabilidad es la de mantener información de datos o de estado como su nombre lo indica, *fuentes de datos* estas clases generan datos o aceptan datos y luego los procesan, a diferencia de las clases manejadores de datos este último no retiene los datos por un periodo largo sino que los genera sobre demanda o los procesa cuando se le llama, *clases vista u observadoras* estas se encargan de presentar la información en algún dispositivo de salida, como por ejemplo la pantalla de una terminal, en este caso los datos base son llamadas clase modelo y la clase que se exhibe es la vista, la modelo puede utilizarse en diversas aplicaciones y no debe requerir ninguna información de la vista, por ejemplo

alguna información financiera de alguna empresa que desea representar su información por medio de gráficas de barras o en forma de pastel, etc., no debe cambiar su modelo base (información) sino su presentación, *clases auxiliares* estas son clases que asisten en la ejecución de tareas complejas.

Estas clases pueden servir de guía en la fase de diseño de la programación orientada a objetos aunque estas suelen ser las más importantes más no las únicas. Si una clase abarcará más dos o más de estas categorías podrá dividirse en dos o más clases.

Al hacer una clara identificación de clases, el diseño con tarjetas CRC debe comenzar sólo con las clases más obvias o necesarias para iniciar la aplicación. Si diversos programadores trabajan de manera conjunta, las tarjetas CRC ayudan como un proceso de simulación en el que se podrán intercambiar las tarjetas que se relacionan de alguna manera en la que pueden agruparse y las que son independientes pueden separarse. Al identificar cada acción del escenario se asigna como una responsabilidad en la tarjeta correspondiente a ese objeto, y se añade respectivamente en la tarjeta. Deben probarse diferentes escenarios ya que pueden generar más responsabilidades. No es conveniente anticiparse a ninguna acción que no haya sido representada con las tarjetas, estas tarjetas la ventaja que tienen es que pueden ser diseñadas por los programadores o comprarlas a un costo muy bajo y que pueden ser remplazadas cuando se haya cometido un error, que estén muy desordenadas o que se dividan en otras clases. Al trabajar a partir de escenarios con estas tarjetas se puede asegurar que el diseño contiene toda la información necesaria para la ejecución de alguna tarea, ya que la manera de pensar es a través de experiencias que el diseñador haya experimentado en la vida diaria, siempre con la mentalidad de que *"para conseguir esto debo de hacer esto"*.

Por ejemplo, si necesitamos retirar dinero de un cajero automático, lo primero es ir a un cajero automático (lector de tarjetas). La primera responsabilidad del LectorDeTarjetas (figura 6.2) es mostrar un mensaje de bienvenida , después se espera a que inserte la tarjeta, una vez que se introduce, el LectorDeTarjetas decodifica la información de la cuenta a partir de la banda magnética y pasa este número al VerificadorDeNip (figura 6.3). El VerificadorDeNip devolverá ya sea un valor verdadero, si es así continuará el proceso, o un valor falso, en el cual se expulsará la tarjeta., se mostrará de nuevo la ventana de bienvenida y se repetirá el ciclo. El VerificadorDeNip pasa primero el número de cuenta al ManejadorDeCuenta (figura 6.4), para comprobar que ésta sea válida y solicitar el NIP (número de identificación personal) registrado en el banco. Entonces se muestra una ventana que pide al usuario que introduzca los dígitos de su número NIP. Si los dos valores coinciden, el VerificadorDeNip devuelve un valor verdadero, de lo contrario devuelve un valor falso.

Si el paso del NIP se completa con éxito, el LectorDeTarjetas pasa el control al SelectorDeActividad (figura 6.5). El usuario selecciona un elemento del menú, por ejemplo, depósito, retiro o terminación. Si selecciona la última actividad se devuelve el control al LectorDeTarjetas, el cual libera la tarjeta y muestra nuevamente la ventana de bienvenida (figura 6.2).

Este ejemplo puede ilustrarse de la siguiente manera, por supuesto utilizando las tarjetas CRC.

Figura 6.2 Responsabilidades del lector de tarjetas

LectorDeTarjetas	Colaboradores
Muestre mensaje de bienvenida, espere tarjeta	VerificadorDeNip
Pida al VerificadorDeNip que compruebe validez	SelectorDeActividad
Llame al Selector de Actividad	
Devuelva tarjeta al usuario	

Fuente: ilustración realizada por la autora de esta tesis

Figura 6.3 Responsabilidades del VerificadorDeNip

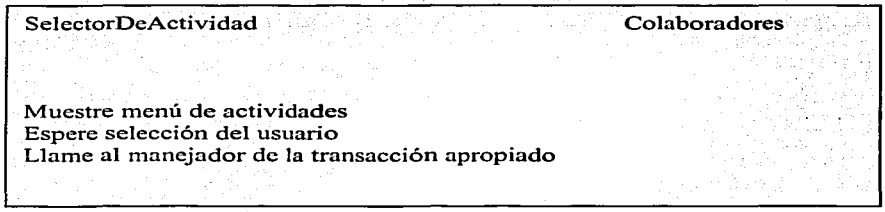
VerificadorDeNip	Colaboradores
Reciba número NIP del ManejadorDeCuenta: devuelva falso si no hay cuenta	ManejadorDeCuenta
Presente ventana de solicitud de NIP	
Reciba número NIP del Usuario	
Compare números NIP: devuelva el resultado	

Fuente: ilustración realizada por la autora de esta tesis

Figura 6.4 Responsabilidades del ManjeadorDeCuenta

Fuente: ilustración realizada por la autora de esta tesis

Figura 6.5 Responsabilidades del SelectorDeActividad



Fuente: ilustración realizada por la autora de esta tesis

Así como se ilustro este ejemplo se pueden ilustrar más, ya que esta actividad se realiza en la vida diaria y el programador puede basarse en ello para otorgar las responsabilidades pertinentes a cada tarjeta y poder dividir las clases en otras. Como se dará cuenta este método es sencillo de llevar a cabo y si ocurre algún error puede desechar la tarjeta pertinente para crear una nueva sin algún costo que no pueda se redituado fácilmente. El utilizar tarjetas CRC es un proceso de refinamiento que es iterativo, cada responsabilidad es examinada y se pueden generar especificaciones más precisas. A tal grado de lograr una especificación detallada para poder recurrir al código.

El siguiente paso para el refinamiento de este diseño, ya que se han asignado las responsabilidades a cada clase, sólo hasta este momento tenemos que resolver el cómo se habrán de alcanzar dichas responsabilidades, es decir, consiste en el manejo de los valores de datos, una sola clase debe tener la responsabilidad de las acciones tomadas para consultar o alterar los valores. Todas las demás clases que necesiten tener los valores deben hacer solicitudes al manejador para tales acciones, más que lograr el acceso de datos por sí mismas. Logrando este punto implicaría proteger cualquier acceso a los datos dentro de una llamada a un procedimiento (el mensaje al manejador). He aquí donde se creará un cuarto campo en la tarjeta CRC, la cual describe los valores de datos manejados por la clase. Cada responsabilidad se examina para determinar que valores necesita para llevar a cabo la tarea asignada, aquí pueden surgir nuevas responsabilidades ya que puede existir que una clase solicite valores a otra clase.

A pesar de estos cuatro pasos existe una segunda etapa que no deja de ser menos o más importante: *la herencia,* como se mencionó antes la programación orientada a objetos maneja la reutilización de código existente, por tal motivo la herencia es una técnica útil de implantación aunque no precisamente de diseño. En esta parte existen dos relaciones importantes que son: es-un y tiene-un (o parte-de). La relación *es–un* define jerarquías de clase-subclase, mientras que la relación *tiene-un* describe datos que se deben mantener dentro de una clase.

La mayoría de las veces, la distinción es muy clara. Algunas veces, sin embargo, la diferencia puede ser muy sutil o puede depender de las circunstancias. Al buscar código que pueda ser reutilizado o generalizado, enfatizamos el reconocimiento de estas dos relaciones.

Empezamos por examinar las clases ya disponibles en bibliotecas asociadas con cualquier lenguaje de programación que se esté usando para la aplicación, por ejemplo, las ventanas se proporcionan habitualmente en las bibliotecas de software existente. El menú de actividades mostrado en el SelectorDeActividad (figura 6.5) es un tipo especial de ventana. Dependiendo del sistema, podemos tener acceso a una gran funcionalidad gratuita con sólo hacer que el Menú DeActividad sea una subclase de una clase existente.

6.5.4.4.3 Consejos prácticos

Por último, una serie de consejos prácticos para la creación de objetos. Muchos de estos consejos son muy similares a los de los métodos estructurados aunque no iguales:

- *No lanzarse a crear clases y asociaciones sin sentido.* Primero hay que entender el problema. Los métodos o pasos no hacen análisis. El análisis lo hacemos nosotros ayudándonos con estos pasos.

- *Intentar que el modelo sea simple, evitando complicaciones innecesarias.* De lo que se trata es de que el modelo sea claro, que alguien más pueda entenderlo, aceptarlo o criticarlo.

- *Los nombres de objetos, asociaciones, atributos y operaciones deben ser significativos.* Antes de poner un nombre hay que pensarlo un poco: el nombre representa al elemento. Las clases deben nombrarse con sustantivos y las asociaciones con verbos. Por las asociaciones se envían mensajes, pero no objetos. Un nombre largo puede ser más significativo pero no será más claro.

- *Nuestro modelo debe ser independiente de la implementación.*

- *Utilizar, si es posible, asociaciones binarias.* Las asociaciones de orden superior son más difíciles de nombrar, entender e implementar. De todas formas hay asociaciones que no podemos descomponer en binarias sin perder información.

- *Evitar las jerarquías de composición o generalización de muchos niveles.* Dificultan la legibilidad del modelo.

- *Revisar el modelo hasta que sea satisfactorio.* No podemos pretender que la primera versión sea correcta. Es conveniente mostrar el modelo a otras personas.

- *Documentar el modelo.* Una imagen vale más que mil palabras, pero una imagen con texto es siempre mucho más explicativa que otra sin él.

- *Utilizar sólo los elementos necesarios.* No se trata de lucirse y mostrar que se dominan todos los conceptos del modelo, sino de que el modelo sea correcto y claro.

6.5.4.4.4 Programación OO

El objetivo principal de la programación es la producción de un código fuente que sea fácil de leer y comprender. La claridad del código fuente facilita la depuración, prueba y modificación, estas actividades consumen una gran porción de la mayor parte de los presupuestos de la programación. Un estilo de codificación varía entre distintos programadores, pero de lo que se trata es de obtener una mejor comunicación entre el proyecto y el usuario.

La creación de un programa legible y confiable es un proceso creativo, por lo que es imposible imponer reglas que gobiernen el estilo de programación. Sin embargo, se pueden establecer varios principios que mejoran la legibilidad de los programas, como lo son :

Nombres de los programas: los objetos de un programa como lo son constantes, variables, procedimientos, funciones y tipos, representan entidades en el mundo real. De acuerdo con esto, los nombres de los objetos deben estar estrechamente relacionados con los nombres de las entidades del mundo real que modelan.

Construcciones de control en los programas: en un programa deben usarse construcciones de control para que le flujo del programa sea descendente. Esto se refiere a que la ejecución debe empezar por la primera proposición de programa, cada proposición debe ejecutarse por turno y la ejecución debe terminar con la última proposición. Las unidades de programa deben tener solo un punto de entrada y una sola salida.

Distribución del programa: la distribución afecta a la legibilidad de los programas. El uso de las líneas en blanco, el énfasis en las palabras reservadas y la agrupación consistente en párrafos hacen que el programa sea más elegante y fácil de leer, y actúan como separadores que distinguen una parte del programa con otra

Es conveniente la observación de que un programa se lee más veces de las que se escribe, y que es responsabilidad de los diseñadores crear y construir programas legibles con lenguajes de programación que así lo permitan.

Sin embargo la legibilidad de los programas no depende de las características del lenguaje sino de el estilo en que un programa está escrito .

Además debe tomar en cuenta que la documentación es importante ya que permite contar con un sistema amigable y su utilización sirve de apoyo al mismo. Conduciendo a un buen uso del sistema desarrollado y otorga al usuario respuestas a sus posibles dudas.

6.5.4.4.5 Pruebas OO

Como los modelos de análisis, diseño y código fuente están enlazados, las pruebas en forma de revisiones técnicas comienzan durante el desarrollo de estas actividades. Una vez que se ha realizado el código fuente la prueba de unidad se aplica a cada clase. Ya que el proceso de prueba se realiza de manera independiente o al obtener un sistema final en el cual los componentes se integran en subsistemas.

La prueba se centra en que todas las sentencias se han probado, y realizando pruebas que aseguren que la entrada definida produce los resultados que realmente se requieren. Es importante asegurar que todas las instrucciones de un programa han sido ejecutadas al menos una vez y que se han ejecutado todas las trayectorias en el programa.

Cuando se prueban objetos esto incluye realizar pruebas aisladas de todas las operaciones asociadas con los objetos y la ejecución de los objetos en todos los estados posibles. Esto significa que se simulan todos los eventos que provocan un cambio de estado en el objeto.

Si se utiliza la herencia, esto hace más difícil diseñar las pruebas de clases de objetos. Cuando una superclase provee operaciones que son heredadas por varias subclases, todas estas subclases se prueban con todas las operaciones heredadas. La razón de esto es que la operación heredada hace suposiciones acerca de otras operaciones y atributos y éstos cambian cuando se heredan. De igual forma, cuando una operación de la superclase se sustituye, la operación sobreescrita se debe probar.

A mi consideración se pueden elegir escenarios para formular las pruebas del sistema, puede crearse una lista de verificación de clases de objetos y métodos, cuando se elige un escenario se pueden marcar los métodos que se ejecutan. Claro que no va a ser posible ejecutar todas las combinación de métodos ya que se llevaría mucho tiempo, pero esto al menos asegura que todos los métodos se prueben como parte de una secuencia. Puesto que pueden organizarse de tal forma que los escenarios más probables se prueben primero y los escenarios no comunes o excepcionales se consideran más adelante en el proceso de prueba.

Al confirmar que la clase candidata o que la creación de una clase cumple con cada uno de estos pasos el siguiente paso sería el:

6.5.4.5 Añadir las nuevas clases a la biblioteca:

Como lo mencioné la tarea del programador al crear alguna clase consistirá en crear alguna clase sencilla para crear objetos a partir de las clases estándar del lenguaje que este utilizando y manejar su interacción. Al haber creado una nueva clase, ésta se pone en la biblioteca de tal manera que pueda ser reutilizada en el futuro.

A medida que el análisis OO y los modelos de diseño evolucionan, se vuelve aparente la necesidad de clases adicionales; por ello, el paradigma arriba descrito trabaja mejor para la OO.

6.5.4.6 Construir n-ésima iteración del sistema:

El tener ya una clase existente que pueda ayudarnos para poder continuar con nuestro proyecto, debe ser incorporada al procedimiento del cual necesitamos de ésta clase, para que así podamos manejar su funcionamiento en el programa y seguir continuación con nuestro proyecto.

6.5.5 Quinta fase:

Evaluación del cliente

Esta fase como cualquiera de las otras tiene su mérito propio, ya que cada uno de ellos nos ayuda a seguir evolucionando en nuestro trabajo.

Esta etapa tiene dos propósitos: extraer de los usuarios la especificación de los requerimientos adicionales del sistema y verificar que el prototipo desarrollado lo haya sido en concordancia con la definición de requerimientos del sistema. Si los usuarios identifican fallas en el prototipo, entonces el desarrollador simplemente corrige el prototipo antes de la siguiente evaluación. El prototipo es repetidamente modificado y evaluado hasta que todos los requerimientos del sistema han sido satisfechos. El proceso de evaluación puede ser dividido en cuatro pasos separados: preparación, demostración, uso del prototipo y discusión de comentarios. En esta fase se decide si el prototipo es aceptado o modificado.

Cabe aclarar que el desarrollo de estas cinco fases es sólo el comienzo para la terminación de un desarrollo orientado a objetos ya que como me mostró en la figura 6.1 este proceso tiene forma de espiral lo que quiere decir que es cíclico, al no cumplir con alguna especificación del cliente hay que detallarlo de manera que cumpla con las necesidades del cliente cuantas veces sea necesario, por ello la ventaja de la reusabilidad es muy importante, ya que nos facilita el desarrollo de todo este proceso y se pueda avanzar de una manera más rápida y eficaz para cumplir con el desarrollo de esta tarea tan ardua y creativa.

Otro punto que es verdaderamente importante es el tema de la calidad, la cual expresa el nivel con que un producto o un servicio cumple con requerimientos precisamente definidos, otras definiciones destacan aspectos como minimización de perdidas para clientes, proveedores, etc.

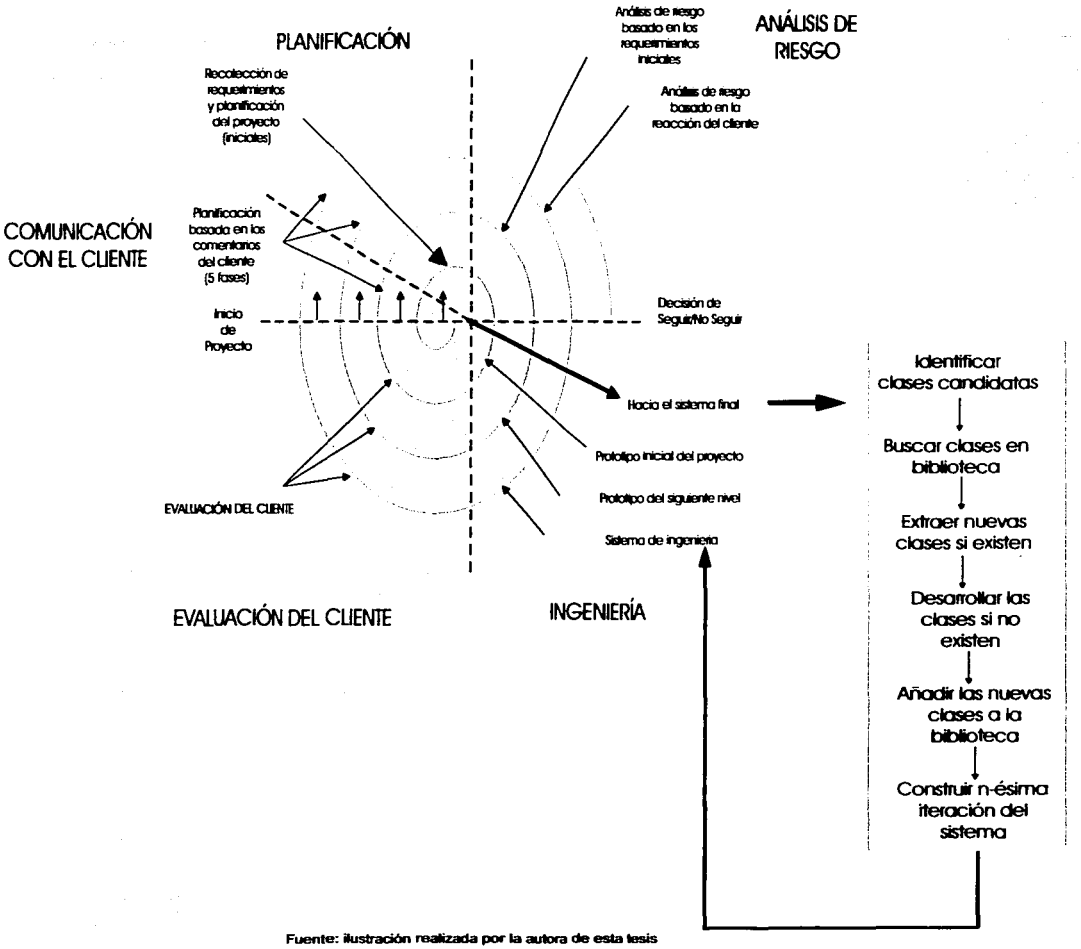
Cuando nosotros proveamos un servicio requerido siempre hay que tener presente este concepto ya que:

Existe calidad cuando vuelve el cliente y no el producto.

Con todo lo expuesto durante este capítulo, la propuesta que propongo presenta diversas variantes con respesto al modelo de proceso OO que propone el autor PRESSMAN (figura 6.2). Dando por resultado la figura que se presenta a continuación:

CICLO DE VIDA PROPUESTO POR LA AUTORA DE ESTA TESIS

Figura 6.6 Ciclo de vida propuesto para el desarrollo de software orientado a objetos



Fuente: ilustración realizada por la autora de esta tesis

Conclusión

Mediante esta tesis y desde mi punto de vista, el software orientado a objetos es el mejor enfoque que puede escoger un desarrollador de software, ya que ofrece múltiples beneficios sin utilizar un lenguaje complicado de una manera rápida y eficaz, obteniendo un producto de buena calidad y que puede ser utilizado para otros proyectos a futuro. Además de que cumple con todas las fases implementadas en un software convencional, consta de su propio ciclo de vida el cual es interpretado mediante un enfoque orientado a objetos que trata de ser lo más natural y sencillo para el desarrollador como para el usuario que requiera de este servicio informático.

Ya que el paradigma orientado a objetos ofrece múltiples ventajas a futuro, se pueden mencionar algunas de las que cuenta el desarrollador al implementar un software orientado a objetos:

- **Reutilización.** Las clases están diseñadas para que se reutilicen en muchos sistemas
- **Estabilidad.** Las clases diseñadas para una reutilización repetida se vuelven estables, de la misma manera que los microprocesadores y otros chips.
- **El diseñador piensa en términos del comportamiento de objetos y no en detalles de bajo nivel.** El encapsulado oculta los detalles y hace que las clases complejas sean fáciles de utilizar.
- **Se construyen clases cada vez más complejas.** Se construyen clases a partir de otras clases, las cuales a su vez se integran mediante clases.
- **Confiabilidad.** Es probable que el software construido a partir de clases estables y probadas tenga menos fallas que el software elaborado a partir de cero.

- **Un diseño más rápido.** Las aplicaciones se crean a partir de componentes ya existentes.
- **Diseño de mayor claridad.** Los diseños suelen tener mayor calidad, puesto que se integran a partir de componentes probados, que han sido verificados y pulidos varias veces.
- **Integridad.** Las estructuras de datos sólo se pueden utilizar como métodos específicos.
- **Programación más sencilla.** Los programas se conjuntan a partir de piezas pequeñas, cada una de las cuales, en general, se pueden crear fácilmente.
- **Mantenimiento más sencillo.** El programador encargado del mantenimiento cambia un método de clase a la vez.
- **Invención.** Los implantadores eficientes encuentran que se puede generar ideas con rapidez.
- **Ciclo de vida dinámico.** El objetivo del desarrollo de un sistema cambia con frecuencia durante la implementación.
- **Esmero durante la construcción.** Las personas creativas, como lo son los desarrolladores de software modifican la manera constante de su trabajo durante la implantación.
- **Modelo más realista.** El análisis OO modela la empresa o área de aplicación de manera que sea lo más cercana posible a la realidad de lo que se logra con el análisis convencional.
- **Mejor comunicación entre los profesionales de los sistemas de información y los empresarios.** Los empresarios comprenden más fácilmente el paradigma OO. Piensan en términos de eventos, objetos y políticas empresariales que describen el comportamiento de los objetos.
- **Modelos empresariales más inteligentes.** Los modelos empresariales deben describir las reglas con las que los ejecutivos desean controlar su empresa.
- **Interacción.** El software de varios proveedores puede funcionar como conjunto.

- **Computación de distribución masiva.** Las redes a nivel mundial utilizarán directorios de software de objetos accesibles.

Como puede darse cuenta el desarrollo de software orientado a objetos cuenta con muchísimas ventajas a muy largo plazo. A diferencia del software convencional se puede establecer una clara ventaja en cada uno de ellos.

La ***ventaja*** **del AOO** es que se basa en la utilización de objetos utilizados en el mundo real. Esto nos permite centrarnos en los aspectos significativos del dominio del problema (en las características de los objetos y las relaciones que se establecen entre ellos) y este conocimiento se convierte en la parte fundamental del análisis del sistema software, que será luego utilizado en el diseño y la implementación.

El DOO constituye un tipo de diseño que logra un cierto número de diferentes niveles de modularidad. el DOO describe la organización de datos específicos, de atributos y los detalles procedimentales de las operaciones individuales. Está representación fragmentada de datos y algoritmos de un sistema OO colaboran a una modularidad general.

Reduce líneas de código en un sistema y facilita los cambios en caso de que se produzcan. Cuenta también con el beneficio de escalar un sistema, además de que evoluciona iterativamente en base a una serie de incrementos.

Las interfaces entre objetos encapsulados están simplificadas. Un objeto que envía un mensaje no se tiene que preocupar por los detalles de estructuras de datos internas en el objeto receptor. Por tanto, se simplifica la interacción y el acoplamiento del sistema tiende a reducirse.

En el desarrollo orientado a objetos extiende el modelo de diseño al dominio ejecutable, es básicamente la producción de un código fuente que sea fácil de leer y comprender. La claridad del código fuente facilita la depuración, prueba y modificación del software.

Un aspecto importante es la selección del lenguaje de programación. La elección del lenguaje influye en el diseño pero el diseño no depende de los detalles del lenguaje. Si se cambia de lenguaje de programación no se requiere el re-diseño del sistema. En el caso de un lenguaje orientado a objetos las estructura básica son las propias clases. Dependiendo del lenguaje de programación esto puede hacerse más sencillo o complicado.

Durante la fase de implementación la correspondencia con el diseño es directa y el sistema implementado es flexible y extensible. La especialización al lenguaje de programación o bases de datos describe cómo traducir los términos usados en el diseño a los términos y propiedades del lenguaje de implementación. Aunque el diseño de objetos es bastante independiente del lenguaje actual, todos los lenguajes tendrán sus especialidades durante la implementación final incluyendo las bases de datos. Por tal motivo queda a consideración del desarrollador la elección del lenguaje que mejor le convenza y sea fácil de implementar.

Los conceptos del paradigma OO pueden aplicarse durante todo el ciclo de desarrollo del software, desde el análisis a la implementación sin cambios de notación, sólo añadiendo progresivamente detalles al modelo inicial.

Los sistemas OO tienden a evolucionar con el tiempo. Por esto el modelo evolutivo de proceso que fomenta el ensamblaje (reutilización) de componentes es el mejor paradigma para ingeniería de software OO. Ya que sus pruebas pueden realizarse de manera individual o hasta que se obtenga un sistema final.

Las métricas OO se centran en métricas que se pueden aplicar a las características de encapsulamiento, localización, ocultamiento de información, herencia y técnicas de abstracción de objetos que hagan única a esa clase.

Por tales motivos propongo el utilizar esta Propuesta Metodológica de Ingeniería de Software Orientado a Objetos ya que ofrece ventajas competitivas en el campo informático y consta de un ciclo de vida más perfeccionado, de fácil comprensión y aplicable de una forma mucho más sencilla que las demás metodologías.

Bibliografía

BOOCH, G. Y J. Rumbaugh, ***Unified Method for Object-Oriented Development***, Rational Software Corp, 1996, 581 pp.

COAD, P. y E. Yourdon, ***Object-Oriented Analysis***, 2ª Edición , ed. Prentice-Hall, México, 1991, 318 pp.

FIRESMITH, D. G., ***Object Oriented Requirements Analysis and Logical Design***, Ed. Wiley, 1993, 249 pp.

G. BOOCH, Benjamin, ***Object-Oriented Analysis and Design with Applications***, 2ª Edición, 1993, 211 pp.

GAMMA, E., ***Design Paterns***, Ed. Adisson Wesley, 1995, 246 pp.

JACOBSON, Ivar, ***Object-Oriented Software Engineering: A Use Case Driven Approach***, Ed. Addison Wesley, 1992, 309 pp.

LORENZ, M. y J. Kidd, ***Object-Oriented Software Metrics***, Ed. Prentice Hall, 1994, 256 pp.

MEYER, Bertrán, ***Object-Oriented Databases: Technology, Aplications and Products***, Ed. McGraw Hill, 1994, 548 pp.

MYERS, G., ***Composite Structured Design***, Ed. Van Nostrand Reinhold, 1978, 129 pp.

PIATTINI, M. G., ***Aplicaciones Informáticas de Gestión***,Ed. Ra-Ma, 1996, 273 pp.

PRESSMANN, Roger, ***Ingeniería del Software: un enfoque práctico***, 3ª Edición Ed. Mc Graw Hill, 1993, 428 pp.

RUMBAUGH, J., ***Object-Oriented Modeling and Design***, Ed. Prentice Hall, 1991, 239 pp.

S. PRESSMAN, Roger , ***Ingeniería de software*** , 4ª edición, Editorial Mac Graw Hill, México, 1998, 581 pp.

SOMM ERVILLE, Ian, ***Ingeniería de software***, 6ª edición, Ed. Addison Wesley, México, 349 pp.

WIRFS-BROCK, R., B. Wilkerson y L. Weiner, ***Designing Object-Oriented Software***, Ed. Prentice Hall, 1990, 409 pp.

Otras fuentes:

AMBLER, S., ***Software Development: Using Use Clases***, julio 1995, págs. 53-61.

BINDER, R., "Testing Object Oriented Systems", ***American Programmer***, volumen 7, núm. 4, abril 1994, págs. 22-29.

CHIDAMBER, S. R. Y F. Kemerer, "A Metrics Suite for Object-Oriented Design", ***IEEE Transf. Software Engineering***, volumen 20, núm. 6, junio 1994, págs. 476-493.

FICHMAN, R. Y C. Kemerer, "Object-Oriented and Conceptual Design Methdologies", ***Computer***, vol. 25, núm. 10, octubre 1992, págs. 22-39.

RUBIN, K. S. y A. Goldberg, "Object Behavior Analysis", **Comunications of the ACM**, vol. 35, núm. 9, septiembre 1992, págs. 48-62.

http://osm7.cs.byu.edu

http://www.amazon.com

http://www.google.com

http://www.rational.com

http://www.soft-design-com/softinfo/objects-html

http://www.terra.com

http://www.toa.com