



**ASOCIACIÓN DE INVESTIGACIÓN
EN SOFTWARE INTELIGENTE**

MODELADO AVANZADO E IMPLEMENTACIÓN DE OBJETOS CON UML Y JAVA



**INGENIERÍA DEL SOFTWARE
ORIENTADA A OBJETOS**

Mayo 2007
La Paz - Bolivia
1

CAPÍTULO I

1. INTRODUCCIÓN

La tecnología orientada a objetos propone una forma de pensar de modo abstracto acerca de problemas a resolver empleando conceptos del mundo real. El Modelado y Diseño Orientado a Objetos se funda en pensar acerca de problemas a resolver empleando modelos que se han organizado, tomando como base, conceptos del entorno. La unidad básica es el objeto que combina las estructuras de datos con los comportamientos en una entidad única (Rossi *et al* 2006).

Pressman (2002) afirma que se vive en un mundo de objetos. Estos objetos existen en la naturaleza, en entidades hechas por el hombre, en los negocios y en los productos. Pueden ser clasificados, descritos, organizados, combinados, manipulados y creados. Por esto no es sorprendente que se proponga una visión orientada a objetos para la creación de productos software, una abstracción que modela el mundo de forma tal que ayuda a entenderlo y gobernarlo mejor manera.

1.1 PARADIGMA ORIENTADO A OBJETOS

Greiff (1994), señala que la Programación Orientada a Objetos (POO) como paradigma, es una forma de pensar, una filosofía, de la cual surge una cultura nueva que incorpora técnicas y metodologías diferentes. La POO como paradigma es una postura ontológica: el universo computacional está poblado por objetos, cada uno responsabilizándose por sí mismo, y comunicándose con los demás por medio de mensajes (Zavala 2002).

Se debe distinguir que la POO como paradigma y como metodología, no son la misma cosa. Sin embargo, la publicidad confunde asociando la POO más a una metodología, que al paradigma. De aquí que el interés en la POO radica más en los mecanismos que aporta para la construcción de programas que en aprovechar un esquema alterno para el modelado de procesos computacionales (Zavala 2002).

La Programación Orientada a Objetos desde el punto de vista computacional, es un método de implementación en el cuál los programas son organizados como grupos cooperativos de objetos, cada uno de los cuales representa una instancia de alguna clase, y estas clases son miembros de una jerarquía de clases unidas mediante relaciones de herencia (Zavala 2002).

A continuación se describen los conceptos más importantes del paradigma orientado a objetos:

a) Abstracción

Pressman (2002), afirma que la abstracción es un mecanismo que permite al diseñador concentrarse en los detalles esenciales de un componente de programa (ya sean datos o procesos). La abstracción es un concepto relativo. A medida que se mueve a niveles más altos de abstracción, se ignoran más detalles, es decir, se tiene una visión más general de un concepto o elemento. A medida que se mueve a niveles de abstracción más bajos, se introducen más detalles, es decir, se tiene una visión más específica de un concepto o elemento. Rumbaugh y sus colegas (2000), afirman que es la identificación de las características esenciales de una entidad que la distinguen de cualquier otra, definiendo un entorno relativo a la perspectiva del observador.

b) Encapsulamiento

Consiste en empaquetar en una clase los atributos (propiedades) y los métodos (comportamiento) que operan sobre los datos. El acceso a los datos sólo es permitido a través de los métodos del objeto. El encapsulamiento permite el uso de objetos como componentes modulares en cualquier lugar del sistema, puesto que los objetos envían y reciben mensajes pero no alteran los métodos internos de otros objetos. Un ejemplo de encapsulamiento de datos es el cálculo de operaciones que realiza la calculadora de manera interna, mostrando solamente el resultado (Terrazas 2004).

c) Modularidad

Es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos coherentes e independientes, es decir, el software se divide en componentes nombrados y abordados por separado, llamados frecuentemente *módulos*, que se integran para satisfacer los requisitos del problema (Pressman 2002).

d) Jerarquía

La jerarquía de clases está representada por un árbol, donde la clase de más alto nivel es denominada *clase padre* y sus descendientes *clases hijas*. Dichas clases se encuentran conectadas mediante líneas.

Al interior de la jerarquía de clases se establece el concepto de *polimorfismo*: que es la habilidad de enviar el mismo mensaje a objetos de diferentes clases y que cada objeto responda de manera particular. Esto sucede en tiempo de ejecución y no en la compilación (Terrazas 2004). En la Figura 1 se muestra un ejemplo de polimorfismo, existe un método *abrir* que para cada objeto (puerta, candado, maletín) tiene un proceso de ejecución distinto.

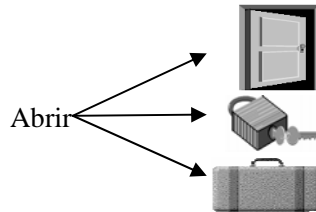


Figura 1: Ejemplo de polimorfismo
Fuente: (Terrazas 2004)

e) Tipificación

Es la definición precisa de un objeto de tal forma que objetos de diferentes tipos no puedan ser intercambiados o, solamente puedan intercambiarse de manera muy restringida (Zavala 2002).

f) Concurrencia

Es la propiedad que distingue un objeto que está activo de uno que no lo está.

g) Persistencia

Es la propiedad de un objeto a través de la cual su existencia trasciende en el tiempo (es decir, el objeto continua existiendo después de que su creador ha dejado de existir) y en el espacio (es decir, la localización del objeto se mueve del espacio de dirección en que fue creado) (Zavala 2002)

h) Beneficios del modelo de objetos

El modelo de objetos asegura que los componentes desarrollados en distintos lenguajes de programación que residan en distintas plataformas pueden ser interoperables. Esto es, los objetos deben ser capaces de comunicarse en una red. Para lograr esto, el modelo de objetos define un estándar para la interoperabilidad de los componentes (Pressman 2002).

Otro beneficio radica en la promoción de la reutilización no solo de software, sino de diseños enteros, conduciendo a la creación de marcos de desarrollo de aplicaciones reutilizables.

También, reduce los riesgos inherentes al desarrollo de sistemas complejos más que nada por que la integración se distribuye a lo largo del ciclo de vida en lugar de ocurrir como un evento principal (Booch 1996).

1.2 PROCESO UNIFICADO RACIONAL

El proceso unificado racional¹ (RUP) es un proceso de desarrollo de software que tiene la finalidad de asegurar la producción de software de calidad dentro de los plazos y presupuestos establecidos. El RUP está dirigido por casos de uso, centrado en la arquitectura y es iterativo e incremental. Los casos de uso, son artefactos² utilizados para establecer el comportamiento deseado del sistema, verificar y validar la arquitectura del sistema mediante las pruebas de software y garantizar la comunicación entre los participantes del desarrollo del proyecto. La arquitectura del sistema es la parte estable del sistema, es el artefacto utilizado para conceptualizar, construir, gestionar y evolucionar dicho sistema. La arquitectura se documenta, especifica y construye con modelos. Finalmente el proceso iterativo e incremental conlleva la gestión de un conjunto de ediciones y la integración continua de las nuevas funcionalidades y mejoras (Pavón 2004).

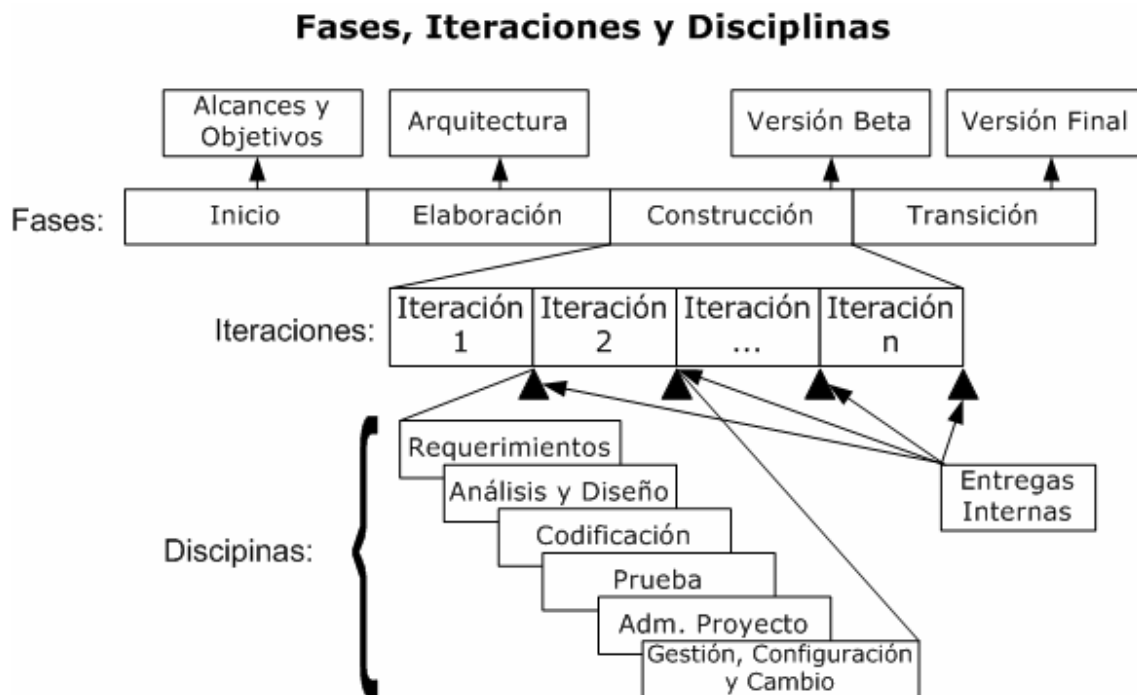


Figura 2: Ciclo de vida del proceso unificado
Fuente: (Torossi 2006)

¹ Del término original "Rational Unified Process (RUP)"

² Es todo producto subproducto del proceso de desarrollo

La figura 2, muestra el ciclo de vida del proceso unificado, dicho proceso se repite a lo largo de una serie de ciclos que constituyen la vida del sistema. Cada ciclo consta de cuatro fases:

- a) Inicio**, define el ámbito del proyecto.
- b) Elaboración**, planifica el proyecto, especifica características y elabora una arquitectura base.
- c) Construcción**, construcción del producto
- d) Transición**, entrega del producto al cliente y mantenimiento del software

Cada fase se subdivide en iteraciones. En cada iteración se desarrolla en secuencia un conjunto de disciplinas, siendo estas, actividades relacionadas, vinculadas a un área específica dentro del proyecto, denominada también *flujos de trabajo*. Los más importantes son: Requerimientos, análisis, diseño, codificación y pruebas.

Cada disciplina está asociada a un conjunto de modelos que se desarrollan. Estos modelos están compuestos por artefactos. Los artefactos más importantes son los modelos que cada disciplina realiza: modelo de casos de uso, modelo de diseño, modelo de implementación y modelo de pruebas (Torossi 2006).

1.3 LENGUAJE DE MODELADO UNIFICADO

El Lenguaje de Modelado Unificado³ (UML) es la creación de Grady Booch, James Rumbaugh e Ivar Jacobson. UML es un lenguaje de modelado estandarizado que consiste en un conjunto de diagramas, tiene el objetivo de colaborar a los diseñadores de productos software en el desarrollo de sistemas de información, realizando las siguientes tareas: especificación, visualización, plan de la arquitectura, construcción, simulación y documentación. UML se desarrolló originalmente con la idea de promover la comunicación y productividad entre los diseñadores de sistemas orientados a objetos logrando rápidamente incursionar dentro de cada tipo y desarrollo del producto software (Larman 1999).

UML es un estándar de facto para el modelado orientado a objetos y continua siendo refinado en nuevas versiones por el Grupo de Gestión de Objetos⁴ (OMG). Cabe mencionar que UML es

³ Del término original "Unified Modeling Language (UML)"

⁴ Del término original Object Management Group (OMG)

sólo un lenguaje y por tanto es parte de un método de desarrollo de software, independiente del proceso (Sommerville 2005)(Larman 2004)(Booch et al 1999).

UML está compuesto por diversos elementos gráficos que se combinan para conformar diagramas, debido a que es un lenguaje de modelado, cuenta con reglas para combinar tales elementos. La finalidad de los diagramas es presentar diversas perspectivas de un sistema, a los cuales se los conoce como *modelo*. Los diagramas UML de un sistema son similares a un modelo a escala de un edificio junto con la interpretación del artista acerca del edificio (Schmuller 1997).

A continuación se listan los diagramas que componen el UML:

- a) Diagrama de casos de uso
- b) Diagrama de actividades
- c) Diagrama de despliegue
- d) Diagrama de estados
- e) Diagrama de interacción
- f) Diagrama de colaboración
- g) Diagrama de secuencia
- h) Diagrama de clases
- i) Diagrama de distribución

El modelado de objetos se encuentra descrito en el diagrama de clases, este diagrama es considerado como el principal para capturar todas las reglas que gobiernan la definición y uso de los objetos. Sin embargo para modelar un producto software no es suficiente definir la herramienta, es necesario definir en primera instancia el método de desarrollo, que en este caso particular es RUP.

1.4 LENGUAJE JAVA

Froufe (2005), enumera las características principales que ofrece el lenguaje Java con respecto a cualquier otro lenguaje de programación, siendo estas características las siguientes:

a) Simple

Los lenguajes de programación como C y C++ son lenguajes muy conocidos por los desarrolladores, por esta razón, Java es similar a estos lenguajes en su sintaxis con el objetivo de facilitar el aprendizaje.

En Java no es necesario realizar procedimientos para recuperar memoria, el *garbage collector* o reciclador de memoria dinámica, se encarga de eso y cuando se encuentra en funcionamiento permite liberar bloques de memoria lo que limita su fragmentación.

b) Orientado a objetos

Java soporta las tres características propias del paradigma orientado a objetos: encapsulamiento, herencia y polimorfismo. Las plantillas de objetos son llamadas *clases* y sus copias, *instancias*.

c) Distribuido

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como *http* y *ftp*. Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos.

d) Robusto

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error.

e) Arquitectura neutral

Para establecer Java como parte integral de la red, el compilador de este lenguaje de programación compila su código a un fichero objeto independiente de la arquitectura de la máquina en que se ejecutará. Cualquier computador que tenga el sistema de ejecución puede ejecutar ese código objeto.

f) Seguro

En Java los punteros o el *casting* que hace el compilador de C y C++ se eliminan para prevenir el acceso ilegal a la memoria. El código Java pasa por comprobaciones antes de su ejecución en la máquina. El código se pasa a través de un verificador de *ByteCode* que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal. Java imposibilita abrir ficheros de la máquina local, no permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otras computadoras como puente, es decir, nadie puede utilizar una máquina para realizar peticiones o realizar operaciones con otra.

g) Portable

Más allá de la portabilidad básica por ser de arquitectura independiente, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que éstas puedan ser implementadas en entornos Unix, Pc o Mac.

h) Interpretado

El intérprete Java (sistema *run-time*) puede ejecutar directamente el código objeto. Enlazar (*link*) un programa normalmente consume menos recursos que compilarlo, Java es más lento que otros lenguajes de programación debido a que debe ser interpretado y no ejecutado.

i) Multitarea

Al ser multitarea o multihilo (*multithreaded*), Java permite realizar actividades simultáneas en un programa. El beneficio de ser multitarea consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente de la plataforma, aun supera a los entornos de flujo único de programa tanto en facilidad de desarrollo como en rendimiento.

j) Dinámico

Las librerías nuevas o actualizadas no paralizarán la ejecución de las aplicaciones actuales siempre que mantengan el API anterior.

CAPÍTULO II

2. OBJETOS Y CLASES

Cuando se usan métodos orientados a objetos para analizar o diseñar un sistema de software complejo, los bloques básicos de construcción son las clases y los objetos. En este capítulo se muestra la naturaleza e identificación de objetos y clases para luego detallar sus componentes.

2.1 IDENTIFICACIÓN DE OBJETOS

La capacidad de reconocer objetos físicos es una habilidad que los humanos aprenden en edades muy tempranas. Normalmente, hasta la edad de un año un niño no desarrolla lo que se denomina el concepto de objeto, una habilidad de importancia crítica para el desarrollo cognitivo futuro. A través del concepto de objeto, un niño llega a darse cuenta de que los objetos tienen una permanencia e identidad además de operaciones sobre ellos.

Desde la perspectiva de la cognición humana, un objeto puede ser:

- Una cosa tangible y/o visible.
- Algo que puede comprenderse intelectualmente.
- Algo hacia lo que se dirige un pensamiento o acción.

Se añade a la definición informal la idea de que un objeto modela alguna parte de la realidad y es, por tanto, algo que existe en el tiempo y el espacio. Los objetos del mundo real no son el único tipo de objeto de interés en el desarrollo del software. Otros tipos importantes de objetos son invenciones del proceso de diseño cuyas colaboraciones con otros objetos semejantes sirven como mecanismos para desempeñar algún comportamiento de nivel superior (Booch 1996).

Por tanto, un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en una clase común; los términos instancia y objeto son similares o intercambiables.

A continuación se definen las propiedades que identifican a un objeto:

- Se entiende por estado del objeto a todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas

propiedades, una propiedad es una característica inherente o distintiva, un rasgo o cualidad que contribuye a hacer que un objeto sea ese objeto y no otro.

- Ningún objeto existe de forma aislada. Ya que los objetos reciben acciones, y ellos mismos actúan sobre otros objetos. El comportamiento de un objeto es la acción y reacción, en términos de sus cambios de estado y paso de mensajes. El estado de un objeto representa los resultados acumulados de su comportamiento. En otras palabras, el comportamiento de un objeto representa su actividad visible y comprobable exteriormente.
- La identidad de un objeto es aquella propiedad que lo distingue de todos los demás objetos.

Un **objeto** es la representación específica de una entidad del mundo real, es una instancia⁵ de una clase que posee: estructura, atributos y comportamiento. Un objeto puede ser una entidad tangible (cajero, tarjeta bancaria, etc.) o una entidad intangible (cuenta, etc).

Para abstraer⁶ las características de un objeto se toman en cuenta: la información y el comportamiento. Cada objeto define tres tipos básicos de información y dos tipos de comportamiento (Pender 2003) (Terrazas 2004).

a) Información

- Un objeto describe las características de una entidad del mundo real.
- Un objeto se describe a si mismo.
- Un objeto describe su estado o condición actual. Un estado se encuentra representado por los valores de sus atributos.

b) Comportamiento

En la programación orientada a objetos, el comportamiento (método) de un objeto se debe definir una vez y ser almacenado, sin tener en cuenta quién lo utiliza, así el momento que se requiera utilizarlo simplemente se debe observar el comportamiento definido en el mismo, así los programas invocan simplemente conductas ya definidas dentro el objeto. Así para manipular un objeto se debe comprender:

⁵ Entidad individual con su propio valor e identidad

⁶ Mecanismo que permite al diseñador concentrarse en los detalles esenciales de un componente del programa; sean estos datos o procesos (Pressman 2002).

- Lo que puede hacer.
- Lo que puede hacerse a él.

Según Ferré y Sánchez (2006), la representación de un objeto viene dada en forma similar a la representación gráfica de una clase. En la parte superior aparece el nombre del objeto junto con el nombre de la clase, según la siguiente sintaxis:

nombre_del_objeto: nombre_de_la_clase

Existen distintas maneras de representar gráficamente un objeto, la Figura 3 muestra dichos tipos:

	<p>Objeto <i>cliente</i> de la clase Persona, con información (atributos) y comportamiento (métodos)</p>
	<p>Objeto de la clase Persona sin nombre específico. con información (atributos) y comportamiento (métodos)</p>

Figura 3: Tipos de representación de un objeto
Fuente: Modificado de (Ferré y Sánchez 2006)

2.2 IDENTIFICACIÓN DE CLASES

Es un concepto orientado a objetos que encapsula las abstracciones de datos y procedimientos que se requieren para describir el contenido y comportamiento de alguna entidad del mundo real. Una clase define uno o más objetos que pertenecen a la clase y que tienen características comunes (Pressman 2002) (Weitzenfeld 2005)

2.2.1 Atributos

Los atributos definen las características o propiedades de una clase u objeto, estos atributos pueden ser manejados a través de las operaciones o métodos. Cada clase debe tener definido

Cliente:Persona

-Cí

-apellidoPaterno

-apellidoMaterno

-nombre

-fechaNacimiento

+Adicionar()

+Eliminar()

Persona

un conjunto de atributos que guardarán el estado de cada instancia del objeto de la clase (Weisfeld 2004).

2.2.2 Método

Es el mecanismo de acceso y administración de los datos del objeto. Un objeto encapsula datos (representados como una colección de atributos) y los algoritmos que procesan estos datos. Estos algoritmos son denominados operaciones, métodos o servicios. Los métodos constituyen la interfaz del objeto con los demás objetos. Cada método está conformado por un conjunto de instrucciones escritas en un lenguaje de programación determinado (Pressman 2002).

Rumbaugh y sus colegas (2000) afirman que un método es la implementación de una operación, mediante el cual se da a conocer, el algoritmo o procedimiento que da lugar a los resultados de una operación.

Por lo visto en los párrafos precedentes, un método es la implementación de una operación de un proceso disciplinado, basado en notaciones generadas a partir de modelos que describen aspectos de productos software en desarrollo. La notación es un conjunto de diagramas normalizados que posibilitan al analista o desarrollador el describir el comportamiento del sistema (análisis) y los detalles de una arquitectura (diseño) de forma no ambigua (Cueva 1999). La figura 4 muestra los tipos de representación gráfica de una clase en UML:

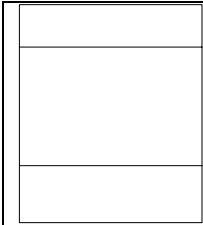
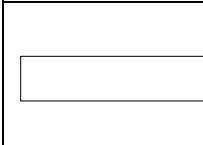
	<p>Representación de la clase Persona con sus respectivos atributos y métodos.</p> <p>Los atributos y métodos en la representación gráfica pueden especificar los valores de los mismos.</p>
	<p>Representación gráfica de la clase Persona sin especificación de sus atributos y métodos</p>

Figura 4: Tipos de representación de una clase
Fuente: Modificado de (Ferré y Sánchez 2006)

Un tipo especial de método es el denominado *constructor*. Se encuentra definido en el lenguaje de programación Java. Dicho lenguaje de programación tiene métodos especiales llamados

constructores que se utilizan para inicializar un objeto nuevo de ese tipo. Los constructores tienen el mismo nombre que la clase.

Java soporta la sobrecarga de los nombres de métodos, por lo que una clase puede tener cualquier número de constructores, todos los cuales tienen el mismo nombre. Al igual que otros métodos sobrecargados, los constructores se diferencian unos de otros en el número y tipo de sus argumentos (Deitel y Deitel 1998).

En Java si no se definen constructores para una clase, el compilador crea un constructor por omisión que no recibe argumentos. El constructor por omisión de una clase invoca al constructor de la clase, que esta clase extiende y procede a inicializar las variables (es decir, las variables de tipos de datos numéricos primitivos, boolean y las referencias nulas) también se puede incluir un constructor sin argumentos (Deitel y Deitel, 1998).

2.3 CONTROL DE ACCESO

Los signos (+), (-), (#) y (~) ubicados en la parte izquierda tanto de los atributos como de los métodos de la clase tienen el significado de que son atributos o métodos públicos (public), privados(private), protegidos (Protected) o paquetes (Package) respectivamente.

La designación de acceso se aplica a nivel de clase y no a nivel de objeto. Los métodos de instancia de un objeto de una clase determinada tienen acceso directo a los miembros privados de cualquier otro objeto de la misma clase. A continuación se describen cada uno de los tipos de acceso explicados por Froufe (2005).

- a) **Private**, las variables y métodos de instancia privados sólo pueden ser accedidos desde dentro de la clase. No son accesibles desde las subclases de esa clase. Un método de instancia de un objeto de una clase puede acceder a todos los miembros privados de ese objeto o miembros privados de cualquier otro objeto de la misma clase.
- b) **Public**, todas las clases pueden acceder a las variables y métodos de instancia públicos.
- c) **Protected**, solo las subclases de la clase pueden acceder a las variables y métodos de instancia protegidos. Los métodos protegidos pueden ser vistos por las clases derivadas por los paquetes. Todas las clases de un paquete pueden acceder a los métodos protegidos de ese paquete.

d) Package (friendly), por defecto, si no se especifica el control de acceso, las variables y métodos de instancia se declaran package, lo que significa que son accesibles por todos los objetos dentro del mismo paquete, pero no por los externos al paquete. Aparentemente este tipo de designación de acceso es similar al tipo *protected*; la diferencia radica, en que la designación de este último es heredada por las subclases de un paquete diferente, mientras que la designación no es heredada por subclases de paquetes diferentes.

2.4 TIPOS DE CLASES

2.4.1 Estereotipos

Los estereotipos son un nuevo tipo de elementos de modelado definidos dentro del modelo basado en un tipo de elemento de un modelo existente.

Los estereotipos son el mecanismo de extensibilidad incorporado, más utilizado dentro de UML. Un estereotipo representa una distinción de uso. Puede ser aplicado a cualquier elemento de modelado, incluyendo clases, paquetes, relaciones de herencia, etc. Por ejemplo, una clase con el estereotipo *actor* es una clase usada como un agente externo en el modelado de negocio. Una clase patrón es modelada como una clase con estereotipo parametrizado, lo que significa que puede contener parámetros.

2.4.2 Clase abstracta

Una clase abstracta no se instancia, se utiliza como clase base para la herencia, tiene al menos un método abstracto, es decir, uno de sus métodos no tiene implementación. La clase abstracta se usa cuando se define una abstracción que engloba objetos de distintos tipos y se desea hacer uso del polimorfismo. (Galiano 2007) (Azorín 2007) (Froufe 2005).

Froufe (2005) afirma que una de las características más útiles de cualquier lenguaje orientado a objetos es la posibilidad de declarar clases que definen cómo se utilizan, sin tener que implementar métodos, estas son las *clases abstractas*. Mediante una clase abstracta se intenta fijar un conjunto mínimo de métodos (el comportamiento) y de atributos, que permiten modelar un cierto concepto, que será redefinido y especializado mediante el mecanismo de la herencia. Como consecuencia, la implementación de la mayoría de los métodos de una clase abstracta podría no tener significado. Para resolverlo, Java proporciona los *métodos abstractos*. Estos

métodos se encuentran incompletos, solo cuentan con la declaración y no poseen cuerpo de definición.

Cuando una clase contiene un método abstracto tiene que declararse abstractamente. Sin embargo, no todos los métodos de una clase abstracta tienen que ser abstractos. Las clases abstractas no pueden tener métodos privados (no se podrían implementar) ni tampoco estáticos. Una clase abstracta tiene que derivarse obligatoriamente, (Froufe 2005).

2.4.3 Clase interfaz

Una interfaz es una clase completamente abstracta cuya clase contiene una colección de métodos que se implementan en otro lugar. En Java las interfaces se declaran con la palabra reservada *interface* de manera similar a como se declaran la clase *abstract* para indicar que una clase implementa una interfaz se utiliza la palabra reservada *implements*. Una interfaz no encapsula datos, solo define cuales son los métodos que han de implementar los objetos de aquellas clases que implementan la interfaz (Galiano 2007) (Azorín 2007) (Froufe 2005)

Los métodos abstractos son útiles cuando se quiere que cada implementación de la clase parezca y funcione igual, pero necesita que se cree una nueva clase para utilizar esos métodos abstractos. Las interfaces proporcionan un mecanismo para abstraer los métodos a un nivel superior, lo que permite simular la herencia múltiple de otros lenguajes (Froufe 2005).

CAPÍTULO III

3. RELACIONES ENTRE CLASES

Las relaciones se encuentran representadas por líneas entre dos o más clases que implica conexiones entre sus instancias. Existen cuatro tipos de relaciones: dependencia, herencia, asociación y realización. En los párrafos posteriores se describen cada uno de ellos.

3.1 DEPENDENCIA

Una dependencia es una relación de uso entre elementos, en el que un cambio en un elemento afectaría al otro que la utiliza y no requiere un conjunto de instancias para su significado; en otras palabras, una dependencia indica una situación en la cual un cambio al elemento origen puede requerir o indicar un cambio en el significado del elemento destino en la dependencia (Rational 2001) (Rumbaugh 2000) (Booch 1999).

Una relación de dependencia indica que las operaciones del elemento destino involucran a las operaciones del elemento origen (Rational 2001). *“Estas relaciones de dependencia son conducidas a través de los flujos de información o control dentro del sistema”* (Pressman 2005).

UML proporciona la representación gráfica para este tipo de relaciones, de forma que permite destacar las partes más importantes de una relación: su nombre y los elementos que conecta.

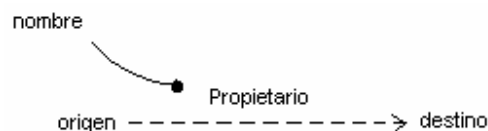


Figura 5: Relaciones de dependencia
Fuente: (Rumbaugh 2000)

Una dependencia se representa por medio de una línea de trazo discontinuo entre dos elementos con una flecha en su extremo. El elemento dependiente es el origen de la flecha y el elemento del que depende es el destino. Booch (1999) afirma que no se utilizan con frecuencia los nombres, a menos que se tenga un modelo con muchas dependencias y haya que referirse a ellas o distinguirlas. Normalmente se utilizan estereotipos para distinguir diferentes variedades

de dependencias. De modo similar la OMG⁷ indica cómo debe ser el uso de varias dependencias: La flecha puede etiquetarse con un estereotipo opcional y un nombre opcional. Es posible tener una lista de elementos origen o destino. En este caso, se conectan una o más flechas apuntadas a los elementos destino la cola de uno o más flechas con sus cabezas en los proveedores (OMG 2003).

Mayormente las dependencias se utilizarán en el contexto de las clases, para indicar que una clase utiliza a otra como argumento en la signatura de una operación (Booch, 1998). La figura 6 muestra un ejemplo de la clase *depósito* que depende de la clase *cuenta* en varios de sus métodos, pues al modificar la estructura de *cuenta*, *depósito* también necesitaría un cambio.



Figura 6: Dependencia entre clases
Fuente: Modificado de (Rumbaugh 2000)

3.2 HERENCIA

La herencia define una relación entre clases, en la que una clase comparte la estructura de comportamiento definida en una o más clases (lo que se denomina herencia simple o herencia múltiple, respectivamente). A medida que se desarrolla la jerarquía de herencias, la estructura y comportamiento comunes a diferentes clases tenderá a migrar hacia superclases comunes. Por esta razón se habla de la herencia como una jerarquía de generalización o especialización. La herencia entre clases se puede entender como una relación del tipo “es un”, “es del tipo” o “es como” lo que permite el ahorro tiempo de codificación, tiempo de prueba, mantenimiento y reutilización de datos y código (Booch 1996, 1999) (Ambler 2004) (Weisfeld 2003).

La herencia se considera un sinónimo de generalización, debido a que este es el proceso que organiza las características de diferentes clases que comparten un mismo propósito. Una generalización es una descripción de las características compartidas por un conjunto de clases, se utiliza este proceso para organizar grandes cantidades de información. Por otra parte una especialización se considera un proceso inverso a la generalización debido a que una clase hereda todas las propiedades generalizadas y añade algunas propiedades únicas que hacen a una clase especial o única en un grupo de clases. Para entender el concepto de generalización,

⁷ Del término original Object Management Group (OMG)

se necesita definir otros conceptos como superclase, subclase, clase abstracta, clase concreta y discriminador (Pender 2003).

Las *superclases* representan abstracciones generalizadas, contienen características que son comunes a dos o más tipos de objetos que comparten el mismo propósito y las *subclases* representan especializaciones de un caso más específico de una superclase, contienen alguna combinación de características que son únicas a un tipo de objeto que está parcialmente definido por la superclase. Una *clase abstracta* es una clase que no tiene una definición completa. Si una clase define una operación u operaciones que no tiene un método, se dice que es abstracta. Debido a que una clase abstracta no esta implementada completamente, no puede crear objetos (no puede ser instanciada). Las clases abstractas deben tener subclases que proveen métodos para cualquier operación sin implementar. Por tanto, solo las superclases pueden ser abstractas (Pender 2003).

Una *clase concreta* es una clase que tiene un método para cada operación así que se pueden crear objetos. Los métodos pueden ser definidos en la clase o heredados de la superclase. Todas las clases debajo de la jerarquía de generalización son llamadas clases hoja. Un *discriminador* es un atributo o regla que describe como se elige el conjunto de subclases para una superclase (Pender 2003).

En las relaciones de generalización no se utilizan ninguna de las reglas aplicadas a la asociación, como la multiplicidad, roles, etc. Estas características son irrelevantes en la generalización.

La herencia es representada en UML, por una línea con una flecha apuntando hacia arriba en la dirección de la superclase, como se muestra en la Figura 6.

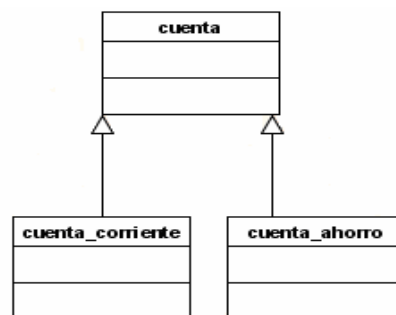


Figura 6: Jerarquía de herencia en UML
Fuente: Modificado de (Prender 2003)

3.3 ASOCIACIÓN

Definir objetos individuales es solo el inicio del modelado del objetos. Para que los objetos del software funcionen, necesitan trabajar juntos de modo que los objetos que representen trabajen también a la par. Es útil encontrar y mostrar asociaciones que son necesarias para satisfacer los requerimientos de escenarios actuales bajo desarrollo y que ayuden a entender el dominio (Pender 2003) (Larman 2004).

Una asociación es un tipo de relación entre clases⁸, que indica algún significado o conexión física o conceptual de interés. Se llama asociación debido a que se muestra que instancias de ciertas clases se asocian, es decir se comunican y trabajan unas con otras. (Larman 2004)(Booch 1996)(Chonoles 2003).

La asociación entre dos clases se representa mediante una línea que las une. La línea puede tener una serie de adornos gráficos que expresan características particulares de la asociación. Por ejemplo, en un sistema de entidad financiera la relación de asociación entre *Cuenta*⁹ y *Usuario*.

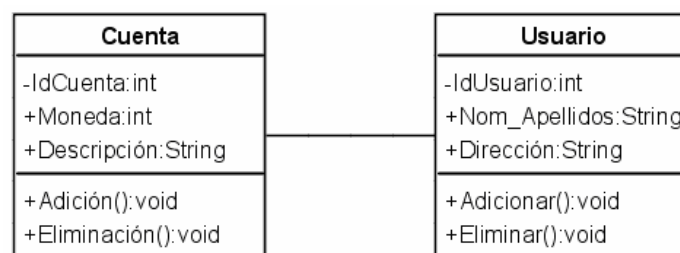


Figura 7: Jerarquía de asociación en UML
Fuente: Modificado de (Larman 2004)

La clase *Usuario* de la figura colabora con otras a través de una relación de asociación entre estas. Dicho de otro modo, una asociación especifica que un *Usuario* utiliza los servicios de otra *Cuenta*.

El propósito de una asociación es establecer la razón que existe entre dos clases y las reglas que gobiernan la relación. Por ejemplo de la figura 7, un *Usuario* puede abrir *cuenta*. La *Cuenta* necesita saber del *Usuario* para que se pueda abrir una cuenta. Estas son las perspectivas de la asociación.

⁸ Específicamente, instancias de clases.

⁹ Cuenta, el usuario puede tener a plazo fijo o caja de ahorro.

3.3.1 Asociaciones binarias

En un diagrama de Clase, una asociación binaria documenta las reglas que gobiernan una relación entre dos clases o entre dos objetos. Por ejemplo la figura 7 muestra una asociación binaria.

3.3.2 Asociaciones n-arias

En el caso de una asociación en la que participan más de dos clases, las clases se unen con una línea a un diamante central. Si se muestra multiplicidad en un rol, representa el número potencial de tuplas de instancias¹⁰ en la asociación cuando el resto de los N-1 valores están fijos. En la figura 8 se ha impuesto la restricción de que un jugador no puede jugar en dos equipos distintos a lo largo de una temporada, porque la multiplicidad de “Equipo” es 1 en la asociación ternaria (Ferré y Sánchez 2006).

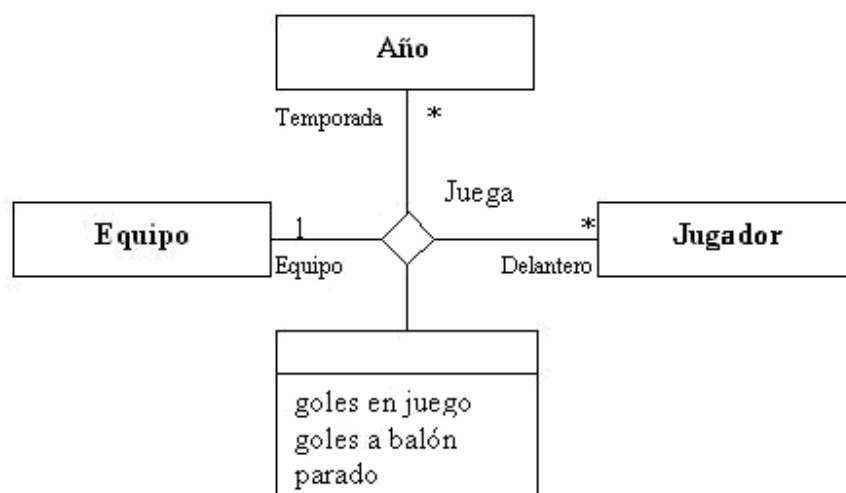


Figura 8: Asociación n-arias
Fuente: (Ferré & Sánchez 2006)

La asociación puede ser complementada con dos partes importantes dentro de la relación de clases (Booch et al 1999):

- Adornos básicos de la asociación.
- Características avanzadas de la asociación.

¹⁰ Manifestación concreta de una abstracción; entidad a la que se puede aplicar un conjunto de operaciones y que tiene un estado que almacena el efecto de las operaciones (Booch et al 1999, Pág. 414).

Aparte de otras formas básicas hay tres adornos básicos de la asociación (Booch et al. 1999) (Ferre & Sánchez 2006):

- *Nombre de la asociación y dirección*
- *Rol.*
- *Multiplidad.*

a) Nombre de la asociación y dirección.

El nombre de la asociación es opcional; pero es importante, porque expresa al lector el intento de la relación. Cada asociación representa tiempo y esfuerzo para establecer la relación, conservando su integridad, y asegura su uso apropiado. Para que no exista ambigüedad se muestra como un texto que está próximo a la línea de asociación. Se puede añadir un pequeño triángulo negro sólido que indique la dirección en la cual se lea el nombre de la asociación. En la Figura 9 se puede leer la asociación como “*Usuario abre cuenta*”.

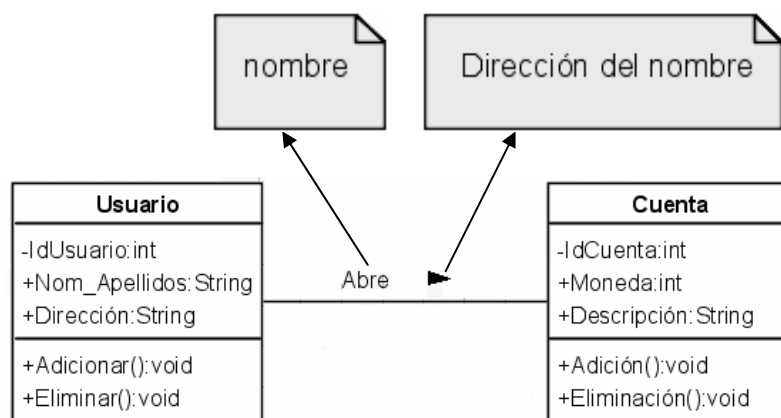


Figura 9: Nombre de asociaciones
Fuente: Modificado de (Ferre y Sánchez 2006)

El nombre de la asociación normalmente se incluye en el modelo para aumentar la legibilidad. Sin embargo, en ocasiones pueden hacer demasiado abundante la información que se presenta, con riesgo de saturación. En ese caso se puede suprimir el nombre de la asociación considerada como suficientemente conocidas.

b) Rol

Al participar una clase en la asociación, tienen un rol específico que juega en la asociación; el contexto del rol¹¹ es simplemente el papel que juega la clase de un extremo, que la asociación presenta a la clases del otro extremo.

En la figura 10 se observa que el *Usuario* juega el papel de cliente de la entidad financiera y la *cuenta* genera ganancia para el usuario.

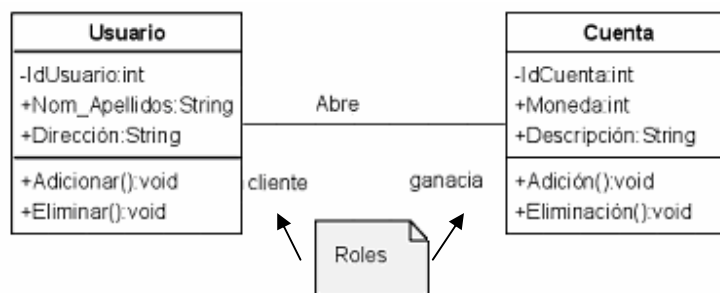


Figura 10: Roles en la asociación
Fuente: Modificado de (Ferré y Sánchez 2006)

c) Multiplicidad

En muchas situaciones de modelado, es importante señalar cuántos objetos pueden conectarse a través de una instancia de una asociación. A este número se le denomina *multiplicidad del rol de la asociación* y se evalúa a un rango de valores o a un valor explícito como se muestra se detalla en la tabla 1 (Booch et al 1999).

La multiplicidad es una restricción que se pone a una asociación, que limita el número de instancias de una clase que pueden tener esa asociación con una instancia de la otra clase. (Larman 1999)

Tabla 1: Tabla de Multiplicidad
Fuente: (Booch et al. 1999)

Valor	Significado
1	Uno y sólo uno
*	Muchos números
0...1	De unos o cero
0...n	De cero o n
0...*	De cero o machos

La multiplicidad define cuántos casos de la clase *A* se asocian con los casos de la clase *B*, es decir para el sistema financiero se dice lo siguiente: un *Usuario* de la entidad bancaria puede

¹¹ La misma clase puede jugar el mismo o diferentes roles en otras asociaciones.

tener más de una *Cuenta* y una *Cuenta* solo puede pertenecer a un sólo *Usuario* o un grupo de *Usuario* (vea la Figura 11).

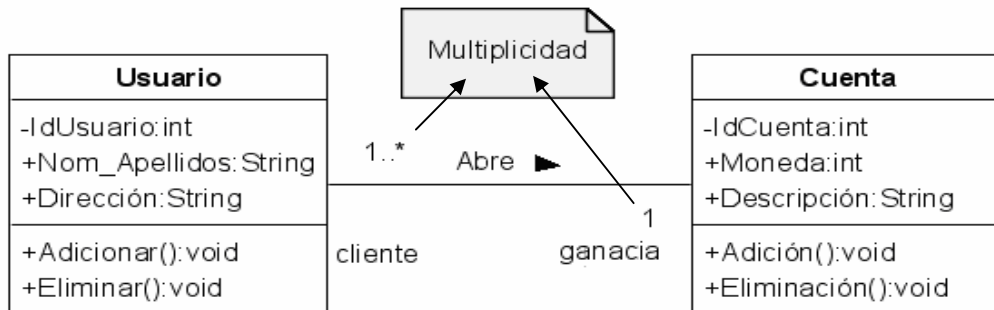


Figura 11: Multiplicidad
Fuente: Modificado de (Ferré y Sánchez 2006)

3.3.3 Agregación y composición

La agregación es una forma especial de asociación que especifica una relación Todo-Parte entre el agregado (el todo) y un componente (la parte).

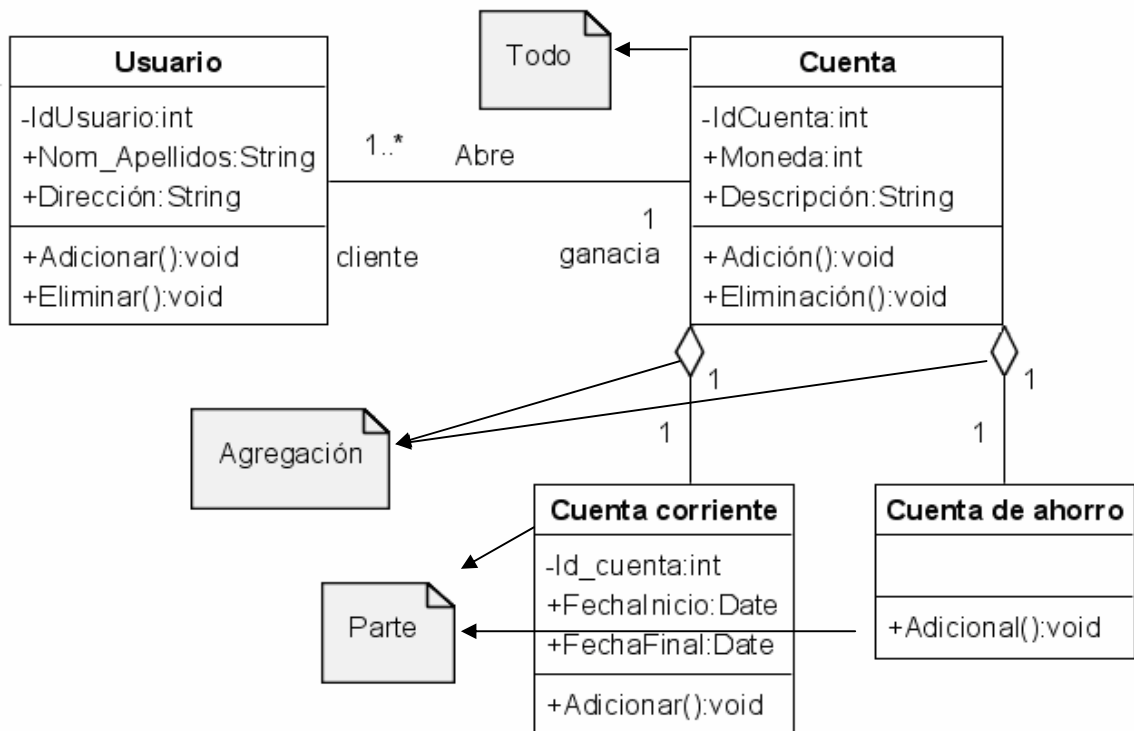


Figura 12: Agregación
Fuente: Modificado de (Ferré y Sánchez 2006)

El significado de esta forma simple de agregación es completamente conceptual. El rombo vacío distingue el "todo" de la "parte" (Booch et al 1999).

Una forma más fuerte de agregación es la agregación de composición: se encuentra representado por un rombo pintado de color negro. Si el padre de una agregación de composición se anula, normalmente todas sus partes se anulan con él; estos tipos de relaciones son transitivas, asimétricas y pueden ser recursivas.

3.3.4 Elementos descriptivos

Se vieron cuatro tipos básicos que se aplican a las asociaciones; pero también debemos conocer otros usos avanzados que hay en las propiedades que permiten modelar detalles como son:

- Clases de Asociación
- Navegación
- Visibilidad.

a) Clases de asociación

Cuando una asociación tiene propiedades se representa como una clase unida a la línea de la asociación por medio de una línea a trazos. Tanto la línea como el rectángulo de clase representan el mismo elemento conceptual: la asociación. Por tanto ambos tienen el mismo nombre de la asociación. Cuando la clase asociación sólo tiene atributos el nombre suele ponerse sobre la línea. Por el contrario, cuando la clase asociación tiene alguna operación o asociación propia, entonces se pone el nombre en la clase asociación y se puede quitar de la línea (Figura 13) (Ferré & Sánchez 2006)

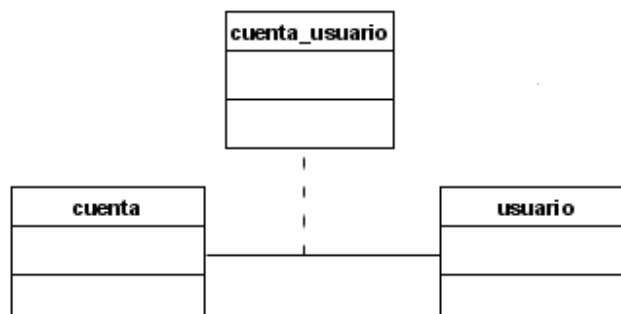


Figura 13: Clases de asociación
Fuente: Basado en (Ferré y Sánchez 2006)

b) Navegación

Se usa una flecha para representar la navegación de la asociación. Especificando una dirección de recorrido no significa necesariamente que no se pueda llegar nunca de los objetos de un extremo a los del otro. En vez de ello, la navegación es una afirmación sobre la eficiencia de recorrido. Significa que es posible "navegar" desde el objeto de la clase origen hasta el objeto de la clase destino. Se trata de un concepto de diseño, que indica que un objeto de la clase origen conoce a los objetos de la clase destino, y por tanto puede llamar a alguna de sus operaciones.

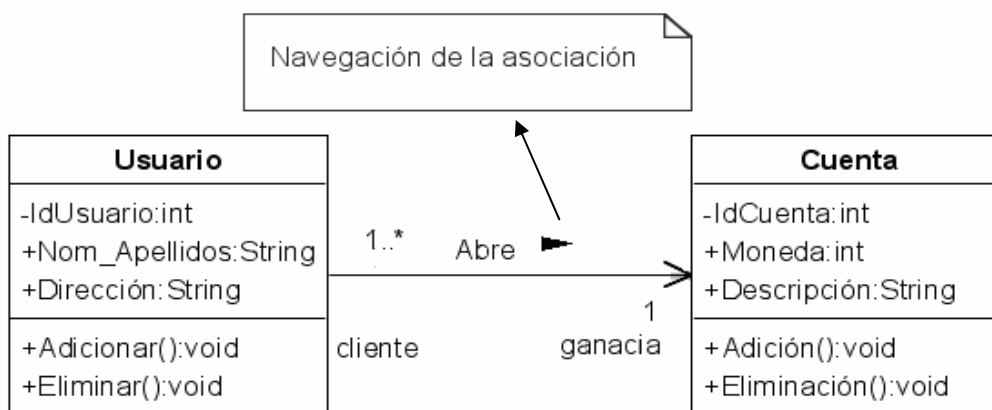


Figura 14: Navegación
Fuente: Modificado de (Ferré y Sánchez 2006)

c) Visibilidad

Dada una asociación entre dos clases, los objetos de una clase pueden ver y navegar hasta los objetos de la otra, a menos que se restrinja por enunciado explícito de navegación. Sin embargo, hay circunstancias en las que se requiere limitar la visibilidad a través de esta asociación relativa a los objetos extremos a ella. En UML se puede especificar una característica con cualquiera de los tres niveles de visibilidad disponibles.

Tabla 2: Características de visibilidad
Fuente: (Booch et al 1999)

Características	Símbolo
Public	+
Protected	#
Private	-

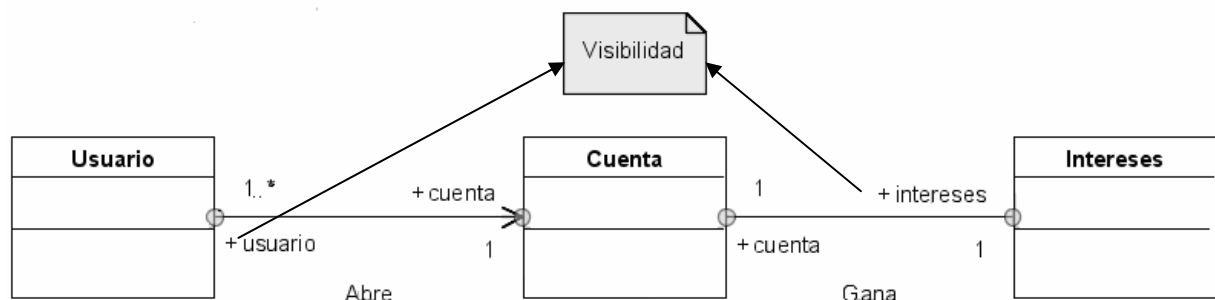


Figura 15: Visibilidad
Fuente: (Booch et al 1999)

3.4 REALIZACIÓN

La realización es una relación semántica entre clases, en la cual una clase especifica un contrato que otra clase garantiza que cumplirá. Semánticamente, la realización es una mezcla entre las relaciones de dependencia y generalización, así como su notación es una combinación de estas notaciones.

La generalización es una forma de lograr el polimorfismo. Otra forma es definir un tipo especial de clase abstracta denominada *clase interfaz*, que sigue algunas de las mismas reglas de una clase común. Una interfaz puede declarar solo las definiciones de las operaciones, los detalles de la operación que dicen a otros objetos como invocar el comportamiento. Esto típicamente consiste del nombre, parámetro y el resultado.

Implementar una clase de interfaz se denomina realizar la clase de interfaz. Esto quiere decir que las reglas establecidas por la interfaz son hechas realidad añadiendo su implementación. La realización permite a una clase heredar de una interfaz sin ser una subclase de la interfaz. Pero las únicas características que la clase puede heredar son las operaciones o métodos, no los atributos, ni las asociaciones (Pender 2003).

3.4.1 Interfaces

Las interfaces definen una línea entre la especificación de lo que un abstracción hace y la implementación de cómo lo hace. Una interfaz es una colección de operaciones que sirven para especificar un servicio de una clase o un componente.

Las interfaces se utilizan para visualizar, especificar, construir y documentar las líneas de separación dentro de un sistema, proporcionan una clara separación entre las vistas externa e interna de una abstracción, haciendo posible comprender una abstracción sin tener que sumergirse en los detalles de su implementación. La visión externa de una clase representada por una interfaz realiza por esta clase, enfatiza la abstracción a la vez que oculta su estructura y el comportamiento. La visión interna de una clase representada por su implementación engloba los secretos de su comportamiento. La implementación de una clase se compone principalmente de la implementación de todas las operaciones definidas en la interfaz de la misma (Booch 1996, 1999).

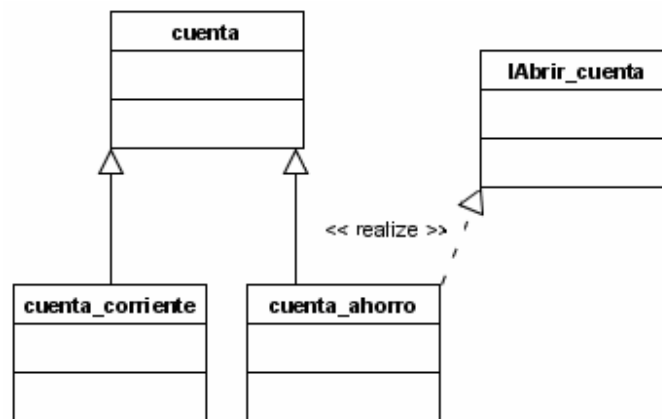


Figura 16: Realización de una interfaz
Fuente: Elaboración propia

En la figura 16, se muestra la realización de una interfaz, que indica la separación de la operación de abrir una cuenta.

CAPÍTULO IV

4. TRANSFORMACIONES DE CLASES A TABLAS

El Lenguaje de Modelado Unificado (UML) es un lenguaje de modelado para el análisis y diseño orientado a objetos. Una parte fundamental es el diagrama de clases que captura la vista estática de un sistema, modela clases en el mundo real y especifica las relaciones entre estas.

El concepto subyacente de los diagramas de clase es que pueden parecer ser similares a los diagramas ER, sin embargo, existen algunas diferencias fundamentales entre estos lenguajes de modelado. Generalmente el modelo ER es utilizado con el método de Análisis y Diseño Estructurado que está centrado en el proceso. Por otra parte, el modelado orientado a objetos como parte del Análisis y Diseño Orientados a Objetos está centrado en funciones y datos.

Transformar un diagrama de clases a un diagrama ER puede ser una tarea no trivial, debido a que muchos símbolos y notaciones que se usan en un diagrama de clases (relaciones n-arias, agregación, composición) no tienen una correspondencia directa en un diagrama ER. Un diseño lógico y físico de bases de datos relacionales requiere de un proceso sistemático para transformar un diagrama de clase a un diagrama ER.

El diagrama de clases de UML conforma la vista conceptual del sistema que necesita ser convertido a una vista lógica para diseñar el esquema de bases de datos relacional, a continuación se detalla el proceso que se puede utilizar para transformar los elementos del diagrama de clases a una vista lógica (Favre 2003).

4.1 Correspondencia de clases a tablas

La forma más fácil de transformar clases en entidades es hacer una correspondencia de uno a uno. A un nivel físico una tabla puede ser creada para cada clase persistente de la vista conceptual. Sin embargo, este paso se puede complicar cuando se considera la generalización y asociación con otras clases.

La correspondencia de uno a uno puede generar los siguientes problemas:

- Varias tablas: Se producen más tablas de las que son necesarias.

- Ausencia de tablas: En las asociaciones de muchos a muchos requiere una tabla que represente los objetos particulares de la asociación.

4.2 Correspondencia de atributos

Un atributo muestra las diferentes propiedades o características de una clase. Cuando se hace la correspondencia de atributos a columnas, un atributo puede hacer correspondencia a cero o más columnas. En un diagrama UML, se puede especificar diferentes propiedades de los atributos a este nivel:

- Visibilidad: A un nivel de base de datos, todas las columnas de una tabla son públicas. Por tanto, cuando se hace esta correspondencia, esta propiedad de acceso puede ser ignorada.
- Multiplicidad: Una multiplicidad de 0..1 denota un atributo como no obligatorio o valor univaluado, por otra parte, una multiplicidad de 1..* denota un atributo como obligatorio, o multivaluado. Una propiedad de un atributo de ser o no ser obligatorio puede ser transformado a una propiedad de ser null o not null a nivel de la base de datos. Sin embargo, transformar un atributo multivaluado no es una tarea simple. Esto se debe a que las bases de datos relacionales no pueden almacenar múltiples valores en una columna. Dependiendo del rango de valores del atributo multivaluado se puede crear o no una tabla para almacenar los diferentes valores y enlazar esta tabla con la tabla original utilizando una referencia de llave foránea.

4.3 Correspondencia de asociaciones simples

Asociación Binaria:

Una asociación es una relación entre dos o más clasificadores. El descriptor de multiplicidad especifica el rango de cardinalidades permitidas que un conjunto puede asumir.

Asociaciones de uno a uno

Se pueden crear tablas separadas para cada clase involucrada en la asociación o se puede combinar las clases para formar una sola tabla. La decisión depende del dominio de aplicación y el grado de emparejamiento que existe entre las dos tablas.

Asociaciones de uno a muchos

Una tabla separada puede ser creada para cada clase involucrada en la asociación. Una asociación es realizada cuando se almacena la llave primaria de la clase (la clase cuya instancia participa en la relación) como una llave foránea en la segunda clase (la clase cuya instancia participa en la relación).

Asociaciones de muchos a muchos

Una tabla separada puede ser creada para cada clase involucrada en la asociación. Además, una tabla puede ser creada para almacenar la asociación por sí misma. Esta tabla también mantendrá las llaves primarias de las tablas asociadas y los atributos asociados.

Asociaciones n-arias

Una notación n-aria es una asociación entre tres o más clasificadores. Cada instancia de una asociación es una n-tupla de valores del respectivo clasificador. Una asociación binaria es un caso especial que tiene su propia notación. La multiplicidad de un rol representa el número potencial de tuplas de instancia en la asociación cuando los otros n menos 1 valores son fijos. Una asociación puede tener también una clase de asociación que puede ser unida al diamante por una línea discontinua. Esto indica que la asociación n-aria tiene atributos, operaciones, y/o asociaciones. A medida que se convierte el diagrama de clases a un esquema lógico, las relaciones n-arias pueden ser divididas en una serie de relaciones binarias. Sin embargo, dependiendo de la cardinalidad de la relación, cada relación puede continuar siendo dividida de acuerdo a otras reglas.

4.4 Correspondencia de agregación y composición

Cuando es colocada en el lado del destino, la relación especifica si el destino es una agregación con respecto al lado del origen. Solo un lado puede ser una agregación. Si un lado es un agregado, el otro lado es una parte y debe tener un valor de agregación de ninguno. La parte puede ser contenida en otros agregados. Si un lado es una composición, el otro lado es una parte y debe tener el valor de agregación de ninguno. La parte es fuertemente poseída por la composición y puede no ser parte de otras composiciones. La composición es una forma de agregación con fuerte propiedad y tiempo de vida coincidental de la parte con el todo. La multiplicidad del lado agregado puede no exceder uno (es no compartida). La composición se muestra como un diamante relleno como un adorno de la asociación.

Dependiendo de la multiplicidad de la asociación, las clases pueden ser convertidas a tablas. Por ejemplo, si existe una relación de 1:N entre el agregado y la parte, entonces dos tablas pueden ser creadas: una para el agregado y otra para la parte. Adicionalmente, la relación puede ser almacenada en la tabla de la parte como una llave foránea. Si existe una relación de M:N entre el agregado y la parte (se debe notar que este caso no es posible para la composición porque una instancia de la parte no puede ser compartida por más de una instancia del agregado), entonces se crean tres tablas: una para cada clase y otra para almacenar la relación.

4.5 Correspondencia de Generalización

Implementar la jerarquía de generalización es una de las tareas más difíciles cuando se convierte la vista conceptual a una vista lógica. Generalmente existen tres alternativas: la primera es asignar una tabla para cada clase, incluyendo la clase padre; la segunda alternativa es copiar los atributos de la clase padre en las clases hijo durante la conversión de clases concretas a tablas; una tercera alternativa es asignar una sola tabla de compuesta que contiene atributos de la clase padre así como los atributos de las clases hijo. Se debe notar que ninguna de las tres opciones es perfecta pero cada opción ofrece alguna ventaja como desventaja sobre cada uno de las otras alternativas.

4.6 Selección de claves primarias

En las bases de datos relacionales, es necesario definir un identificador único para cada fila en una tabla, así como para almacenar las relaciones. Un identificador debe ser asignado para cada objeto (fila, en la terminología relacional) para identificarlo de manera única. Existen varias opciones para seleccionar las llaves primarias:

- Seleccionar un atributo que tiene un significado funcional como llave primaria: Se puede escoger un atributo que tenga un significado funcional en el negocio. Una ventaja de este enfoque es que los usuarios del sistema son o están familiarizados con la llave, la desventaja es que si el dominio del negocio cambia, el significado de la llave también debe cambiar.
- Asignar un atributo separado para almacenar el identificador de objetos: Se puede añadir un atributo a la definición de la clase para identificar las llaves únicas, estos atributos separados pueden ser generados automáticamente por el sistema, la desventaja de este enfoque es que como la llave primaria no tiene un significado funcional, la tabla necesitará un campo más que cumpla la función de llave primaria.

BIBLIOGRAFÍA

(Rossi et al 2006)

Rossi Bibiana, Britos Paola, Garcia Ramón, 2006 *Modelado de Objetos*. Centro de actualización permanente en ingeniería del software (CAPIS), Escuela de Postgrado. Disponible en: <http://www.itba.edu.ar/capis/webcapis/RGMITBA/articulosrgm/R-ITBA-21-modeladodeobjetos.pdf>
[Acceso: abril 2007]

(Pavón 2004)

Pavón Mestras Juan, 2004 *El Proceso Unificado*. Dep. Sistemas Informáticos y Programación Universidad Complutense Madrid. Disponible en: <http://www.fdi.ucm.es/profesor/jpavon/is2>
[Acceso: noviembre 2006]

(Torossi 2006)

Torossi Gustavo, 2006 *El Proceso Unificado de Desarrollo de Software*. Disponible en: <http://www.ecomchaco.com.ar/UTN/disenodesistemas/apuntes/oo/ApunteRUP.pdf>
[Acceso: febrero 2006]

Ambler, S., 2004, *The Object Primer*, 3ra. Edición, Cambridge University Press.

Booch, G., 1996, *Análisis y diseño orientado a objetos con aplicaciones*, 2da. Edición, Addison-Wesley.

Booch, G., Rumbaugh, J., Jacobson, I., 1999, *El Lenguaje Unificado de Modelado*, Addison-Wesley.

Deitel H., Deitel P. 1998, *Como Programar en JAVA*, Primera edición, Pearson Education, México.

Ferré, X., Sánchez, M. I. 2006, *Desarrollo Orientado a Objetos con UML*. Disponible en: <http://www.elquintero.net/Manuales/UML/umlTotal.pdf>
[Acceso: Abril 2007]

Jiménez Pacheco Javier, 2006 *Programación Orientada a Objetos*. Disponible en: http://www.esimez.ipn.mx/acadcompu/apuntes_notas%20breves/programacion_orientada_objetos.pdf
[Acceso: Marzo 2007]

Larman, C. 1999, *UML y Patrones Introducción al Análisis y Diseño Orientado a Objetos*, Primera edición, Prentice Hall Hispanoamericana S.A., México.

Larman, C. 2004, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Tercera Edición, Addison Wesley Professional, New Jersey.

Laski, Stanley, Hurst, *Dependency Analysis of Ada Programs*, disponible en: <http://www.iowa.edu/~articulos/laski.pdf>
[Acceso: Abril 2007]

Object Management Group OMG 2003, *UML 2.0 Superstructure Specification.*, Disponible en: <http://www.sparxsystems.com/bin/UML2SuperStructure.pdf>
[Acceso: abril 2007]

Pender, T. 2003, *UML Bible*, John Wiley & Sons.

Pressman, R. 2002, *Ingeniería del Software un Enfoque Práctico*, 5^{ta} Edición, McGraw Hill, España.

Pressman, R. 2005, *Ingeniería del Software un Enfoque Práctico*, 6^{ta} Edición, McGraw Hill, España.

Rational Rose 2001, *Using Rose*, disponible en: <http://www.rational.com>
[Acceso: Abril 2007]

Rumbaugh, J., Jacobson, I., Booch, G. 2000, *El Lenguaje Unificado de Modelado. Manual de Referencia*, Pearson Educación S.A., Madrid.

Schmuller, J. 1997, *Aprendiendo UML en 24 horas*, Primera edición, Editorial División computación, México.

Sommerville, I. 2005, *Ingeniería del Software*, Séptima Edición, Pearson Educación S.A., Madrid.

Terrazas, N., *Análisis y Diseño Orientado a Objetos*, Disponible en:
http://www.tevasoft.com/UMSA_ANT/ADOO/adoo-1.pdf#search=%22nelson%20terrazas%22
[Acceso: mayo, 2006]

Weisfeld, M. 2004, *The Object-Oriented Thought Process*, Second Edition, Sams Publishing, United States of America.

Weitzenfeld, A. 2005, *Ingeniería del Software Orientada a Objetos con UML. Java e Internet*, 1^{ra} Edición, Thomson Editores S.A., México.

Zavala, J. 2002, *Ingeniería del Software*, Disponible en:
<http://www.angelfire.com/scifi/jzabalar/apuntes/apuntes.html>
[Acceso: septiembre, 2006]

Favre, L. 2003, *UML and the Unified Process*, Idea Group Publishing.