UNIVERSIDAD NACIONAL
AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES
A C A T L Á N

"EL ESTÁNDAR ODMG PARA BASES DE DATOS ORIENTADAS A OBJETOS"

TESINA

QUE PARA OBTENER EL TÍTULO DE
LICENCIADO EN MATEMÁTICAS
APLICADAS Y COMPUTACIÓN
PRESENTA

JESÚS SANDOVAL LUGO

ASESORA: DRA. AMPARO LÓPEZ GAONA

MÉXICO, D. F. SEPTIEMBRE 2004

Agradecimientos

A Dios:

Por darme la fuerza para poder terminar este ciclo de mi vida.

A mis Padres (Pablo y Aurelia):

Que han estado conmigo toda la vida y a pesar de todas las cosas, siempre me han apoyado, me han enseñado a su manera a vivir la vida y lo que hay que enfrentar en ella. A ustedes Pablo y Aurelia los seres que me trajeron a este mundo a enfrentar grandes retos. Los AMO.

A mis Hermanos (Rocío, Rosario y Pablo Antonio):

Que han estado conmigo, y a pesar de nuestras diferencias hemos llegado a un punto en el que nos apoyamos y sabemos que contamos los unos con los otros.

A mi Tía Adela, mis primos Fabiola y Rubén y mi sobrino Enrique:

Por que los quiero y siempre están en mi mente y corazón, y se que cuento con ellos y Enrique por venir a darle un toque especial a nuestra familia.

A Verónica:

Por el amor y apoyo que me brindo mientras elaboraba este trabajo, te agradezco de todo corazón cada momento que me diste.

A la Dra. Amparo:

Por el asesoramiento que me dio y la paciencia que tuvo de esperar para que terminara este trabajo.

A Marta:

Por ayudarme a enfrentar y encontrarme a mi mismo, y ayudarme encontrar la fuerza y voluntad para poder llegar a la conclusión de este trabajo.

A todas aquellas personas (amigos, compañeros de trabajo, conocidos, maestros, compañeros de la carrera, etc.) que en algún momento cruzaron por mi vida y me motivaron para que saliera adelante.

El estándar ODMG para Bases de Datos Orientadas a Objetos

Introducción

Este trabajo es el producto de la búsqueda de información sobre la orientación a objetos y las bases de datos, a las cuales se les aplica este paradigma y el estándar ODMG.

La orientación a objetos es un paradigma que se empezó a utilizar a finales de los años sesenta, pero aún no estaba muy bien delimitado, por lo que tardó en tomar fuerza y apenas en la pasada década comenzó a tener gran auge en varias áreas de la computación. Este paradigma se empezó a dar en los lenguajes de programación, aunque no se había planteado como una forma de programación, se utilizó el concepto de objeto para el desarrollo de programas. En la actualidad, la orientación a objetos es la que más se está utilizando para todo tipo de desarrollo de software, desde el modelado hasta la implantación.

Las bases de datos son una herramienta que se ha utilizado desde hace mucho tiempo para el manejo de grandes volúmenes de información. Éstas permiten almacenar toda la información que sea necesaria para solucionar algún problema y después se pueda utilizar del modo que sea más adecuado. A través de los años se han utilizado diferentes modelos para la implementación de una base de datos, comenzando por los sistemas de archivos, que evolucionaron a las bases de datos jerárquicas y de red, las cuales fueron eficientes hasta que llegó el modelo relacional, el cual hace el empleo de las bases de datos más sencillo y práctico. Este modelo es el que se ha venido utilizando hasta nuestros días, pero como el área de la computación siempre está evolucionando, así como los lenguajes de programación que tenían el paradigma estructurado y evolucionaron a la orientación a objetos, las bases de datos se involucraron con el paradigma de la orientación a objetos también, esto se dio por la información y el tipo de datos que se empezaron a utilizar, y que se han vuelto más complejos que los que se utilizaban antes de que aparecieran éstos (enteros, cadenas de caracteres, flotantes, boleanos, etc.), una de las bondades de la orientación a objetos es el manejo de este tipo de datos complejos.

Una vez que se empezaron a desarrollar bases de datos orientadas a objetos, cada desarrollador implementaba éstas con sus propias características, no se tenía un estándar que seguir, y cuando empezaron a surgir más sistemas manejadores de bases de datos orientados a objetos, hubo problemas al querer pasar de un sistema a otro, ya que no se tenían las mismas estructuras, esto provocó que las industrias desarrolladoras de bases de datos bajo el paradigma de la orientación a objetos se unieran en un grupo llamado ODMG (Object Database Managment Group), el cual se creó para desarrollar un estándar que cumpla cualquier sistema manejador de bases de datos orientada a objetos.

Este trabajo está estructurado de la siguiente manera:

En el capítulo 1 se da una definición de lo que es la orientación a objetos, así como los elementos con los que ésta trabaja, la orientación a objetos es la forma más común con que se trabaja en el área de la computación, ya que se emplea en diferentes áreas (Lenguajes de Programación, Ingeniería de Software, Bases de Datos, Interfaces de Usuario Gráficas, etc.), se habla también de las características de las bases de datos, cuáles fueron los antecedentes de las bases de datos; se describen los sistemas

manejadores de bases de datos (SMBD), cómo están compuestos, y cuáles son las partes que los conforman. Una vez que se termina de describir la estructura de un SMBD se pasa a las bases de datos orientadas a objetos: qué son, cuál es su propósito o la necesidad de que hayan surgido, las características que tienen; se describe, además, cómo ha evolucionado este tipo de base de datos, ya que no siempre cumplió con las características propias de una base de datos.

En el capítulo 2 se habla de las características que se determinaron en el estándar ODMG y que deben cumplir los sistemas manejadores de bases de datos que estén bajo el paradigma de la orientación a objetos, éstos deberán cumplir con un modelo de objetos y demás características de la orientación a objetos que se definieron dentro de este estándar. También se habla de OQL (Object Query Language), que es el lenguaje que se definió a semejanza del lenguaje de consulta estructurado SQL que se utiliza en los sistemas manejadores de bases de datos relacionales y se describe como está estructurado para los SMBDO.

En el desarrollo de sistemas, el modelo que se haga de éstos desde un principio es muy significativo para su realización, ya que un buen modelo de cualquier sistema es importante para poder implementarlo, ya que en éste es donde se conceptualizan y definen todos los puntos esenciales del problema a resolver.

Finalmente en el capítulo 3 se habla del lenguaje de modelado UML, que es un proceso que emplea todo el paradigma de la orientación a objetos para el desarrollo de software. Empezando desde la especificación de requerimientos, pasando a través de la etapa de análisis y diseño, desarrollo y pruebas, hasta llegar a la implementación y mantenimiento. UML es un proceso que utiliza una gran variedad de elementos para poder representar todos los componentes a través de las diferentes etapas del desarrollo y que al final integrarán el producto.

Capítulo 1. Bases de Datos Orientadas a Objetos

1.1 Antecedentes de la Orientación a Objetos

El concepto de la orientación a objetos se empezó a conocer con los lenguajes de programación. El primer lenguaje que comenzó a mostrar esta ideología fue SIMULA67 en los años sesenta; en este lenguaje se manejó por primera vez el concepto de objetos, aunque aún no se especificaba bien la tecnología orientada a objetos con este lenguaje, se intentaban modelar los objetos de una simulación como objetos de software. Después de que apareció este lenguaje con la idea de programar creando y utilizando objetos, y como era de esperarse en el ámbito de la computación, empezaron a aparecer más lenguajes con el concepto de orientación a objetos tales como Smalltalk, C++, Eiffel, etc. Éstos ya tenían una noción más específica de la orientación a objetos. A partir de los resultados que generó el concepto objetos en los lenguajes de programación, y la forma en que se facilitaba ver las cosas como simples objetos, ya que vivimos en un mundo creado con éstos, el paradigma de la orientación a objetos empezó a utilizarse en diferentes áreas del ramo de la computación; se comenzó a utilizar en la Ingeniería de Software, en el desarrollo de Interfaces de Usuario Gráficas (GUI Graphic User Interface) y, como era de esperarse, también en las Bases de Datos, entre otras áreas.

1.2 ¿Qué es la orientación a objetos?

La orientación a objetos es un concepto de modelado aplicable a diferentes áreas dentro de la computación (modelado de sistemas, bases de datos, interfaces de usuario, lenguajes de programación, etc.), este concepto sirve para representar un "ente" en la forma de un objeto a través de sus características o atributos junto con métodos que sirven para darle comportamiento al objeto y para la manipulación de sus datos, y de esta forma se puedan comunicar e interactuar unos objetos con otros a través de mensajes, con lo cual llegaríamos a la solución de un problema dado.

El elemento esencial en la orientación a objetos es *el objeto*, el cual va a estar compuesto de atributos y métodos. Los atributos son las características o datos que definen al objeto y las funciones o métodos que servirán para el manejo de los datos que se encuentran en los atributos como para la comunicación con otros objetos.

Además del objeto, la orientación a objetos se basa en otras características fundamentales que son: encapsulamiento, mensajes, métodos, polimorfismo, clase, instancia, subclases herencia, entre los más utilizados.

1.2.1 Encapsulamiento

El encapsulamiento es la forma en que vamos a delimitar y encerrar los datos o atributos y los métodos u operaciones que va a necesitar una clase para poder manipular sus datos y con esto crear un conjunto de objetos de cierto tipo, con características similares. Para ejemplificar este concepto se tiene la clase *VehículosAutomotrices* la cual tiene definidos ciertos atributos y métodos, uno de los atributos de esta clase es *estadoMotor*, este atributo sirve para indicar si el motor está encendido o apagado, el estado de este atributo

sólo podrá ser modificado a través de un método que también se definió llamado *cambiaEstadoMotor()*, el cual podrá cambiar el estado del motor de encendido a apagado y viceversa, con esto sabemos como cambiar el estado del motor, y no necesitamos conocer más que la forma de llamar al método, ya que la forma de cómo quedó implementado no la conocemos, sólo sabemos lo que hace; otro atributo es *velocidad*, este atributo sirve para indicar la velocidad en la que se encuentra nuestro vehículo, el estado de este atributo puede ser modificado por los métodos *acelerar()* y *frenar()*, los cuales aumentan y disminuyen la velocidad del vehículo.

1.2.2 Mensajes

Los mensajes son la forma en que los objetos se pueden comunicar unos con otros, un mensaje consta del nombre de un método y los argumentos que necesita para poder realizar la función para la que se le llamó, además, a través de los mensajes se hace la interoperabilidad entre los objetos, por lo que un mensaje es el que se produce cuando se manda llamar a un método de la clase para ejecutarlo. Siguiendo con la clase *VehículosAutomotrices*, si necesitamos que el vehículo aumente su velocidad mandamos un mensaje con el nombre del método *acelerar()*, con esto se le está indicando al vehículo que tendrá que aumentar la velocidad.

1.2.3 Métodos

Los métodos son la forma en que los objetos van a responder a los mensajes que les mandan otros objetos, dependiendo del nombre del método que venga en el mensaje, este método se ejecutará, realizando la operación para la que fue creado, la operación puede interactuar además de con los datos, con otros objetos enviándoles mensajes para que se efectúe alguna operación. Los métodos contienen un conjunto de operaciones que son las que van a manipular el estado de un objeto, tomando el método *acelerar()*, este modificará el estado en que se encuentra el atributo *velocidad*, aumentando ésta cada vez que se mande llamar al método.

1.2.4 Clases

Las clases son una forma de representar un determinado conjunto de objetos de manera general, la clase actúa como si fuera un molde para la creación de objetos que tienen tanto las mismas características o atributos como los mismos métodos. Una clase es diseñada por el usuario para que cumpla con las características que se deseen, dependiendo de los objetos que se quieran conceptualizar con la clase. En una clase se definen los tipos de datos que van a tener los atributos del objeto y se implementan los métodos que se van a emplear para la manipulación de dichos datos, retomando el ejemplo del punto 2.2.1 mencionamos la clase *Vehículos Automotrices*, la cual encierra a todos aquellos vehículos que cuentan con un motor y las características generales que tienen estos vehículos, como por ejemplo, estado_del_motor, capacidad_de_carga, etcétera.

1.2.5 Instancia (Objeto)

Una instancia es la que se produce al tomar a una clase y asignarle ciertos valores a los atributos de la clase, para que formemos un objeto, a este objeto creado se le conoce como *instancia de la clase*. La información contenida en los atributos de la clase será definida por los métodos realizados sobre la instancia, en este caso siguiendo con el ejemplo de la clase *Vehículos Automotrices*, creamos la instancia *miVehículo* a la cual se le asignan valores a sus atributos, en este caso estado_del_motor=Apagado, capacidad_de_carga=100ton., de esta forma se crea el objeto *miVehículo*.

1.2.6 Herencia

La herencia es una forma de reutilizar clases ya existentes para crear clases con algunas características iguales pero unas más específicas que otras, la herencia ayuda a jerarquizar las clases haciéndolas cada vez más específicas. La superclase tiene los datos comunes de todos los objetos y la subclase hereda los atributos y métodos de la superclase y añade los que la especializan más. De la clase *Vehículos Automotrices* que hemos estado utilizando, se pueden generar otras clases más específicas como *Motocicletas*, *Autos*, *Camiones*, etc., donde éstas van a tener características más específicas para cada una de ellas.

Hay dos tipos de herencia, *la herencia simple* y *la herencia múltiple*, en la primera la subclase sólo hereda sus características de una sola clase y en la segunda la subclase las hereda de diferentes clases.

1.2.7 Subclase

Una subclase es aquella que se crea heredando los atributos y métodos de otra clase a la que se le llama *clase padre* o *superclase*, además de aumentar en ciertas características en atributos y métodos la subclase, lo que la hace más específica, como se ejemplificó en el punto 2.2.6, de la clase *Vehículos Automotrices*, se crearon las subclases *Motocicletas*, *Autos*, *Camiones*, cada una de éstas con características muy particulares, por ejemplo, además de tener los atributos de la clase *Vehículos Automotrices*, la clase *Autos* y *Camiones* tienen el atributo cantidad_de_Ventanas y la clase *Motocicletas* no lo tiene.

1.2.8 Polimorfismo

Esta característica permite implementar a un método de distintas formas, el polimorfismo permite tener métodos con el mismo nombre en la misma clase o en otras clases, pero su implementación va a depender de la clase en que se encuentre, de los argumentos o parámetros que necesite y de la forma en que va a utilizarse el método.

1.3 Ventajas de la Orientación a Objetos

La tecnología orientada a objetos, en términos breves, proporciona un mapeo directo entre un modelo de negocios y la construcción de componentes de software usando lenguajes de programación orientados a objetos. Además, gran parte del código puede ser construido de componentes prefabricados, proporcionados por el vendedor del

lenguaje. Finalmente, los componentes u objetos pueden ser cambiados y extendidos con mínimos efectos sobre otros, esto hace que sea fácil modificar sistemas para reflejar requerimientos cambiantes.

A grandes rasgos, las organizaciones crean bibliotecas de componentes prefabricados, los cuales se han construido como parte de proyectos previos y pueden ser utilizados nuevamente sobre muchos proyectos, esto ayuda en gran parte a la reducción de tiempos en desarrollo y mejora el regreso de la inversión original.

La orientación a objetos tiene la gran ventaja de encapsular tanto datos como métodos en un mismo lugar, llamado clase, para poder realizar en forma independiente operaciones. Con esto es posible atomizar funciones especificas de cada objeto y después hacer una integración completa de diferentes objetos según la necesidad del problema a resolver. Al hacer esta minimización de funcionalidad en diferentes clases u objetos (minimización en el aspecto a funcionalidad específica) se puede hacer una mejor reutilización de componentes como se mencionó en párrafos anteriores, debido al encapsulamiento y a la herencia, la cual se puede utilizar para mejorar las clases ya creadas y adaptarlas a nuestras necesidades para crear objetos más específicos sobre el nuevo problema.

Puede decirse que la orientación a objetos es un paradigma que vino a facilitar la concepción del mundo de la computación a través de la interpretación de todo como objetos. Ya que de esta forma se pueden minimizar los problemas en objetos muy pequeños para que su funcionalidad no sea tan compleja, sino al contrario, sea de lo más sencilla y específica posible.

1.4 Observaciones de la Orientación Objetos

En la actualidad el uso de la orientación a objetos ha tomado una gran fuerza y la mayoría o casi en su totalidad el desarrollo de sistemas se realiza aplicando esta metodología. El problema que se presentó cuando surgió la orientación a objetos fue el cambio de una metodología a otra, ya que los desarrolladores que programaban bajo el paradigma estructurado, al hacer el cambio al paradigma de la orientación a objetos tuvieron problemas de concepción de éste, ya que aquí se salta la funcionalidad entre los objetos existentes, y no siguiendo una secuencia como se hacía en la programación estructurada. El manejo de la operabilidad se cambió a funciones específicas entre los objetos.

Cuando se empezó a manejar el paradigma de la orientación a objetos, a todos los que tenían una cultura sobre el paradigma estructurado se les hizo difícil el paso de uno a otro, lo que dificultó el progreso de la orientación a objetos, algunos recomendaban que para empezar a comprender la orientación a objetos era mejor no saber nada o que nunca hubieran manejado del modelo estructurado, así la aceptación y asimilación del paradigma orientado a objetos era mucho más sencilla y no cruzaba tanto la idea de lo estructurado con los objetos.

Ya en los últimos años el concepto de la orientación a objetos ha sido aceptado por la gran mayoría de los programadores y es lo que más se está utilizando para la realización de grandes desarrollos.

1.5 ¿Qué es una base de datos?

Una base de datos es un conjunto de archivos interrelacionados entre sí donde se va a guardar la definición de la estructura que va a contener a los datos; la estructura donde se van a almacenar los datos está dada por un modelo de datos que cubre las características de los datos que se requieren para un problema en particular en el que se manejarán grandes volúmenes de información, esto con el propósito de tener un mejor control en el procesamiento de los datos y la manipulación de la información.

1.6 Antecedentes de las bases de datos

El avance en el manejo de la información se ha conducido con el paso del tiempo a mejorar las formas de guardar grandes volúmenes de información, desde el uso de los sistemas orientados a archivos hasta los sistemas de bases de datos relacionales.

En los sistemas orientados a archivos se utilizan los conceptos de registro y archivo, los cuales no tienen una relación directa entre ellos para poder accesar los registros de diferentes archivos al mismo tiempo, por lo que se tienen que desarrollar programas independientes para poder accesar los registros de cada archivo por separado, dadas estas circunstancias empezaron a desarrollarse los primeros sistemas de bases de datos, basándose en un modelo jerárquico.

El modelo jerárquico propone que todas las interrelaciones dadas entre los datos puedan estructurarse como jerarquías (de ahí su nombre), donde los datos son representados como registros y las relaciones que se dan entre éstos se representan como los enlaces que existen entre los registros, haciendo la organización como colecciones de árboles. Con esta arquitectura de la base de datos se empezó a obtener un mejor manejo de las relaciones entre los datos, pero estaba limitada en sus interrelaciones, ya que al accesar la base de datos, sólo se podían seguir los enlaces jerárquicos y si se deseaba relacionar otra información que no estuviera en una misma jerarquía causaba muchos problemas en la forma de obtener los datos, ya que se tenían que crear programas especiales para poder accesar la información de la base de datos, los cuales eran demasiado complejos, gracias a este problema de las interrelaciones se creó el modelo de red de base de datos.

El modelo de red se creó debido a que ciertos archivos se necesitaban enlazar con más de un archivo, rompiendo con esto la jerarquía que se tenía, ya que en este caso, un archivo podría tener dos padres o más a diferencia del modelo jerárquico que sólo permite un padre, por lo que se tuvo que ampliar este modelo a una nueva estructura, creando una red entre los archivos que se tenían en la base de datos, esta nueva forma de modelar seguía empleando los conceptos de registros en archivos y las interrelaciones o enlaces (apuntadores), pero añadiéndole el concepto de grafos dirigidos. Este concepto empezó a dar una mejor solución al manejo de la información, creando mejores formas de accesar los datos; por el uso de los grafos dirigidos, se determinaba de mejor manera la navegación entre los archivos; pero, al igual que el modelo jerárquico, las interrelaciones o enlaces con apuntadores se tenían que declarar antes de poner en marcha el sistema, y esto ocasionaba que si en algún momento se llegara a omitir una de éstas o no se contemplara desde el inicio en el modelo, resultaba difícil crear un programa para accesar esta información y, a veces, resultaba hasta imposible.

El empleo de apuntadores físicos se convirtió en una base sólida para la recuperación rápida de datos interrelacionados que se determinaban desde un principio como se mencionó en el párrafo anterior, pero el crear estas interrelaciones antes de que el sistema empezara a utilizarlas debilitaba a la base de datos, ya que no se podrían recuperar datos tan fácilmente o, incluso, no se lograba esta recuperación si no se habían declarado los enlaces correspondientes a estos datos para poder accesarlos. Debido a esta problemática con la definición de las interrelaciones y la declaración de apuntadores, y con la dificultad de poder crear nuevas relaciones que se necesiten posteriormente, se desarrolló un nuevo modelo de bases de datos. "En 1970, Edward F. Codd publicó un artículo revolucionario (Codd, 1970) que desafió fuertemente el juicio convencional de la 'condición' de las bases de datos" [HAHA, 1997]. Codd decía que los datos se deberían relacionar de una forma natural, lógica e inherente a los datos, en lugar de crear una relación con un apuntador físico. Con esta idea, Codd diseñó el modelo relacional, en este modelo, la representación de los datos se realiza por medio de tablas, a las que se les asigna un nombre exclusivo para tener una referencia de éstas y a su vez las tablas están constituidas por filas y columnas, en donde cada fila de una tabla representa una relación entre un conjunto de valores de diferente tipo, y juntos tienen algún sentido, por lo que se le llama relación, y cada columna es la colección de valores de un mismo tipo. Además de proponer esta nueva forma de modelar una base de datos, Codd propone dos lenguajes para poder llevar a cabo la manipulación de los datos de una forma natural, del mismo modo en que se diseñó la relación de los datos en el modelo relacional, estos lenguajes son: *el álgebra relacional* y *el cálculo relacional*. Codd diseñó el modelo relacional pensando en forma de conjuntos y de igual forma en que se pueden realizar operaciones de conjuntos; diseñó los lenguajes pensando en forma matemática para una fácil manipulación de los datos dentro de las tablas. Esta forma de manipulación reemplazó el uso de los apuntadores físicos, que en algún momento funcionaron eficientemente, pero debido a las nuevas necesidades de información que se requerían, su uso se volvió más complejo.

En la actualidad las bases de datos relacionales tienen gran aceptación en el mercado, especialmente en operaciones comerciales, ya que al ver el desempeño del manejo de los datos en este tipo de bases de datos, provocó que se empezaran a migrar de los sistemas jerárquicos y de red a los sistemas relacionales.

Aunque las bases de datos relacionales ayudaron a resolver muchos de los problemas en el manejo de datos, la tecnología no se ha detenido ahí, como ya se sabe, se sigue avanzando tecnológicamente en todas las áreas de la computación y las bases de datos no se podían quedar atrás. En la actualidad, los datos que se desean almacenar dentro de una base de datos se han hecho cada vez más complejos (imágenes, sonido, vídeo, etc.), por lo que a las bases de datos relacionales les resulta complicado poder representar estos tipos de datos, es por esto y por la influencia de la tecnología orientada a objetos que se empezaron a desarrollar las bases de datos orientadas a objetos, ya que la orientación a objetos brinda un gran número de bondades para la creación y manejo de tipos de datos más complejos.

A continuación se presenta una cronología del desarrollo histórico de los métodos de acceso a los datos que se han discutido en párrafos anteriores [HAHA, 1997]:

Cronología del desarrollo histórico de los conceptos para el manejo de datos.

1.7 Componentes de los sistemas de bases de datos

Un sistema de bases de datos se constituye de cuatro componentes:

- Hardware
- Software
- Datos
- Personas

1.7.1 Hardware

El hardware es la parte física o conjunto de dispositivos físicos donde va a residir la base de datos, estos dispositivos consisten en una o más computadoras, las cuales pueden llegar a ser mainframes, minicomputadoras o computadoras personales; unidades de disco, que se utilizan para almacenar las bases de datos y donde se hace el acceso directo a los datos; video_terminales e impresoras, las cuales se utilizan para la recuperación de información contenida en la base de datos; unidades de cinta magnética, las cuales sirven para hacer los respaldos de la información almacenada en las unidades de disco; para mantener y controlar la gran cantidad de datos almacenados en una base de datos, se requiere de una memoria principal y un espacio de almacenamiento en disco muy grande. Además, se necesitan computadoras rápidas, redes y periféricos para efectuar el alto número de accesos requerido para recuperar la información en un tiempo aceptable en un ambiente que tenga una cantidad grande de usuarios.

1.7.2 Software

Hay dos tipos de software en un sistema de base de datos, éstos son:

- El software de propósito general para el manejo de las bases de datos, llamado Sistema Manejador de Bases de Datos (SMBD)
- El software de aplicación, que usa las facilidades del SMBD para poder accesar y manipular la información contenida en la base de datos.

El SMBD es un software que proporciona una serie de servicios para los usuarios finales, programadores y otros, con los que podemos realizar el manejo de una base de datos. Con este propósito, un sistema manejador de bases de datos proporciona los siguientes servicios:

- Un diccionario de datos, que es una herramienta para la definición y el control centralizado de los datos.
- Mecanismos de seguridad e integridad de los datos.
- Acceso concurrente a los datos por varios usuarios.
- Utilidades para la consulta, la manipulación y la elaboración de informes orientados al usuario.
- Utilidades para el desarrollo de sistemas de aplicación orientados al programador.

1.7.3 Datos

Los datos son la parte importante del uso de una base de datos, ya que sin éstos no sería lógico utilizar una base de datos. Estos datos corresponden a las necesidades de información y de procesamiento que en un determinado momento una organización requiere. Estos datos se guardan en la base de datos de una manera bien estructurada lógicamente, para esto se deben analizar las reglas de negocio, los atributos de los datos y las relaciones que se identifiquen al momento de modelar la estructura de la base de datos, para que sean bien definidos, y tales definiciones se deben almacenar en el diccionario de datos de manera precisa. Teniendo bien definida la estructura de la base de datos con sus elementos correspondientes, se podrá obtener e introducir datos, y será una poderosa herramienta para el manejo de la información en una organización.

1.7.4 Personas

En esta parte es posible definir tres tipos de personas diferentes que hacen uso de la base de datos, éstas son:

- Los usuarios.
- Las personas del área de sistemas y bases de datos.
- Las personas encargadas de los procedimientos.

Los usuarios son las personas que se ocuparán de explotar los datos que se almacenen en la base de datos a su conveniencia, éstos pueden ser los ejecutivos, gerentes, administradores, el personal de oficina, por mencionar algunos.

Las personas del área de sistemas son las que se encargarán de dar mantenimiento a la base de datos y las aplicaciones que se desarrollen en torno a ésta, estas personas pueden ser los administradores de la base de datos, analistas, programadores, diseñadores del sistema y de la base de datos y administradores de los sistemas de información.

Las personas encargadas de los procedimientos son las que se encargan que éstos logren las metas del sistema, los procedimientos son instrucciones escritas que describen los pasos necesarios para realizar una tarea determinada en un sistema, ya que no se puede a veces automatizar por completo alguna tarea del usuario.

Estos cuatro componentes (software, hardware, datos y personas) están muy relacionados y podríamos definirlos de la siguiente manera: el personal de sistemas y base de datos junto con los usuarios definen un modelo de datos para la solución de un

problema, este modelo creará una estructura para desarrollar la base de datos que se construirá dentro del SMBD, lógicamente que es parte del software y se guardará físicamente en los discos duros y cintas magnéticas que son parte del hardware y se tendrá acceso a los datos en la base de datos a través de programas de aplicación (software) y procedimientos específicos, éstos interactuarán con diferentes dispositivos (computadoras, impresoras, cintas magnéticas y discos) para la introducción y extracción de datos de la base de datos. Y toda esta unión de componentes es con el fin de alcanzar las metas de alguna organización.

1.8 ¿Qué es un SMBD?

Un sistema manejador de bases de datos SMBD (DBMS por sus siglas en inglés *Data Base Managment System*) es software que permite el manejo de una base de datos, esto es, soporta un modelo de datos para diseñar los datos y relaciones entre ellos para que se almacenen en la base de datos, y una vez creada la estructura a partir del modelo, el SMBD ayudará a cumplir, a través de ciertos mecanismos, con las propiedades de la base de datos, que son: la persistencia, la recuperación de datos, control de la integridad, transparencia de la distribución, control de concurrencia y control de seguridad. Los componentes principales que conforman un SMBD se muestran en la figura 1 [UJWJ, 1999]:

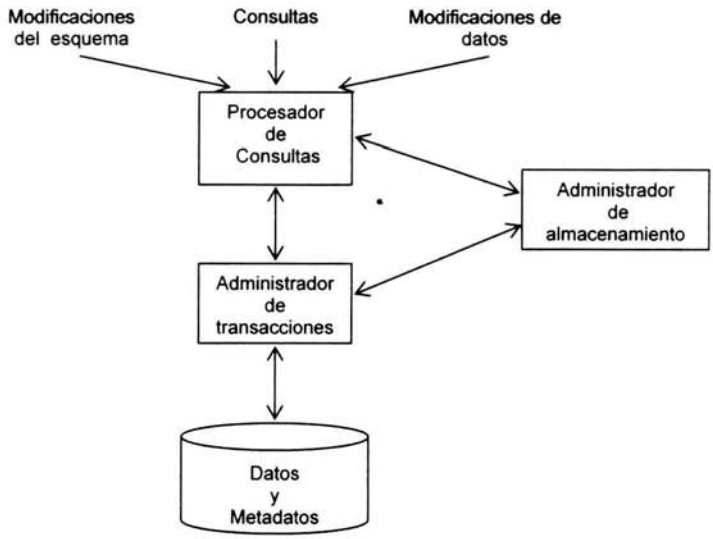


Figura 1. *Estructura de los componentes de un SMBD.*

En la figura 1 se podrán distinguir cuatro componentes. En la parte inferior se encuentra el componente donde se guardan los datos y metadatos (información referente a la estructura de la base de datos). El administrador del almacenamiento es el que se hace cargo de obtener la información del lugar de almacenamiento en donde se encuentra, además de poder modificarla según las peticiones de los niveles de sistemas que se encuentran arriba de él. El procesador de consultas es el responsable de recibir todas las

operaciones de consulta y de manipular la base de datos, así como de transmitir los comandos al administrador de almacenamiento para que se ejecuten.

El administrador de transacciones se encarga de conservar la integridad del sistema. Este componente asegura que las consultas que se ejecuten al mismo tiempo no interfieran entre sí y que el sistema no pierda información cuando sobrevenga alguna falla. Interactúa con el procesador de consultas para saber sobre qué datos se dirigen las consultas y con esto sabrá si están accesando a la misma información dos o más consultas al mismo tiempo, además de conocer la operación que se está haciendo, con esto sabrá qué prioridad le dará a cada operación, las operaciones son de manipulación de datos, de consulta y de modificación de la estructura de la base de datos, dependiendo del tipo de operación dará la prioridad de ejecución a cada una de éstas, además de definir cuando bloquear un recurso que emplea un usuario y liberarlo para que lo sigan utilizando los demás usuarios, con esto asegurará la integridad y consistencia de la base de datos. El administrador de transacciones también interactúa con el administrador de almacenamiento porque los esquemas de protección de información suelen requerir guardar una bitácora de los cambios efectuados en ella.

En la parte superior de la figura 1 se encuentran las operaciones que se pueden realizar en un SMBD, ya mencionadas en el párrafo anterior:

- Consultas.
- Modificaciones de datos.
- Modificaciones del esquema.

1.8.1 Consultas

Este tipo de operación permite recuperar información de la base de datos, ésta se puede aplicar a través de dos formas; una es mediante la interfaz de consultas genéricas, aquí se escriben y hacen las consultas directamente con el SMBD a través del lenguaje SQL, además de que se puede especificar cualquier tipo de consulta según las necesidades de información que el usuario tenga; la segunda forma es mediante una interfaz de algún programa de aplicación, en este caso la consulta sería más específica ya que se tiene que declarar y delimitar sobre qué datos va a interactuar esa interfaz, lo que ocasiona que las consultas no sean abiertas, ya que se limitan a realizar consultas específicas, declaradas dentro de la interfaz.

1.8.2 Modificaciones de datos

Las operaciones de este tipo se utilizan para cambiar, eliminar o insertar más datos dentro de la base de datos, es decir, sirven para afectar el contenido de ésta; se realizan de forma semejante a las operaciones de consulta, mediante la interfaz de consultas genéricas o alguna interfaz de aplicación.

1.8.3 Modificaciones del esquema

Las operaciones de este tipo sirven para modificar la estructura del modelo de la base de datos, ya sea para eliminar, agregar o modificar tablas, clases, índices, etc.; estos

comandos sólo pueden ser emitidos por personal que esté autorizado para hacerlo, a este tipo de personas se les conoce como administradores de la base de datos (DBA, por sus siglas en inglés *DataBase Administrator*).

1.9 Bases de Datos Orientadas a Objetos

Las bases de datos orientadas a objetos (BDO) son la conjunción de un sistema de base de datos (SBD) con el modelo orientado a objetos, esto es, todas las características de un SBD: persistencia, transacciones, control de concurrencia, recuperación, consultas, integridad, seguridad y eficiencia; unidas con las características del modelo orientado a objetos: identidad del objeto, jerarquías, objetos compuestos y tipos de datos abstractos.

Los sistemas manejadores de bases de datos orientados a objetos (SMBDO) se empezaron a desarrollar por el útil manejo de tipos de datos complejos, los cuales a un sistema manejador de base de datos relacional (SMBDR) le resultan complicado, y a veces es hasta imposible su manejo en un sistema de este tipo. Este manejador resulta insuficiente para el desarrollo de aplicaciones multimedia, sistemas de gestión documental, sistemas de diseño asistido por computadora (CAD), ingeniería de software asistida por computadora (CASE), bases de datos hipertexto, etc.; ya que estas aplicaciones requieren definir y manipular entidades abstractas y complejas, cuya representación resulta imposible en los sistemas relacionales. Estos nuevos tipos de aplicaciones no se habían contemplado al inicio de los sistemas de bases de datos en los años setenta, ya que en aquel tiempo sólo se pensaba en tipos de datos sencillos, pero con el avance de la tecnología (incremento de memoria principal y del tamaño de los discos, el aumento de la velocidad de las unidades de procesamiento central, al menor costo de hardware y a la mejor comprensión de la gestión de las bases de datos [SIKS, 1998]) se han podido llevar a cabo estas aplicaciones.

En concreto, los SMBDO se han desarrollado por la gran evolución de la tecnología, la tendencia de la orientación a objetos que ya se aplica en muchas áreas de la computación y de la necesidad de representar tipos de datos más complejos, que sólo se lograrían uniendo un SBD con el paradigma de la orientación a objetos.

1.9.1 La necesidad de las bases de datos orientadas a objetos

El gran énfasis sobre el proceso de integración es una fuerza que impulsa a la adopción de sistemas de bases de datos orientados a objetos. En este caso, el área de Manufactura de Cómputo Integrado (CIM, *Computer Integrated Manufacturing*) está fuertemente enfocada en el uso de la tecnología de bases de datos orientadas a objetos como la estructura de trabajo del proceso de integración. Actualmente en las oficinas se están empleando las bases de datos orientadas a objetos en sus sistemas de automatización para el manejo de datos hypermedia. Los sistemas que siguen el cuidado de los internos en un hospital utilizan la tecnología de las bases de datos orientadas a objetos por su facilidad de empleo. Todas estas aplicaciones están caracterizadas por tener un manejo complejo de la información, altamente relacionada, que es la fuerza de los sistemas de bases de datos orientados a objetos.

La tecnología de los sistemas de bases de datos relacionales ha fallado en las necesidades de manejo de sistemas complejos, el problema con estos sistemas es que requieren desarrollar una aplicación para forzar un modelo de información a tablas donde las relaciones entre las entidades están definidas por los valores. En una comparación que hace Mary Loomis, arquitecto del SMBDO Versant entre las bases de datos relacionales y las orientadas a objetos dice:

> Diseñar una base de datos relacional es realmente un proceso de intentar representar objetos del mundo real dentro de los confines en tal forma que tenga buenos resultados de ejecución y preserve la integridad de los datos como sea posible. El diseño de base de datos del objeto es absolutamente diferente. Para la mayor parte, el diseño de base de datos del objeto es una parte fundamental del proceso total del diseño de la aplicación. Las clases del objeto usadas por el lenguaje de programación son las clases usadas por el SMBDO. Por que sus modelos son consistentes, no hay necesidad de transformar el modelo de objetos del programa a algo único para el administrador de la base de datos [Loo92b].

Algunas de las áreas en las que más se han enfocado varios de los vendedores de bases de datos orientadas a objetos son en aplicaciones para: Diseño Asistido por Computadora CAD (*Computer Aided Design*), Manufactura Asistida por Computadora CAM (*Computer Aided Manufacturing*) e Ingeniería de Software Asistida por Computadora CASE (*Computer Aided Software Engineering*). La característica principal de estas aplicaciones es la necesidad de manejar información muy compleja de forma eficiente. Otra área donde la tecnología de bases de datos orientadas a objetos puede ser aplicada es en la automatización de fábricas y oficinas. Por ejemplo, la manufactura de un avión requiere de varias piezas interdependientes que se puedan ensamblar en diversas configuraciones. Los sistemas de bases de datos orientados a objetos mantienen la promesa de poner soluciones a estos problemas complejos dentro del alcance de los usuarios.

La orientación a objetos es aún otro paso en la búsqueda para expresar soluciones a problemas en una forma más natural y fácil de comprender.

El estudio de la historia de las bases de datos está centrado sobre el problema del modelado de los datos. Un modelo de datos es una colección de conceptos bien definidos matemáticamente que ayudan a considerar y expresar las propiedades estáticas y dinámicas de aplicaciones de datos intensivas. Un modelo de datos consta de:

1) Propiedades estáticas tales como objetos, atributos y relaciones.
2) Reglas de integridad sobre objetos y operaciones.
3) Propiedades dinámicas tales como operaciones o reglas, definiendo nuevos estados de la base de datos sobre cambios de estado aplicados.

Las bases de datos orientadas a objetos tienen la habilidad para modelar estos componentes directamente dentro de la base de datos soportando la capacidad de modelar un problema/solución completo. Antes de las bases de datos orientadas a objetos, las bases de datos fueron capaces de soportar los puntos uno y dos arriba mencionados y confiadas en las aplicaciones por definir las propiedades dinámicas del

modelo. La desventaja de delegar las propiedades dinámicas a las aplicaciones es que estas propiedades no pueden ser aplicadas uniformemente en todos los escenarios del uso de la base de datos, puesto que fueron definidas fuera de la base de datos en aplicaciones autónomas. Las bases de datos orientadas a objetos proporcionan un paradigma de unificación que permite integrar los tres aspectos del modelo de datos y que se puedan aplicar uniformemente a todos los usuarios de la base de datos.

1.9.2 La Evolución de las Bases de Datos Orientadas a Objetos

La investigación y prácticas de las bases de datos orientadas a objetos se empezaron a realizar a finales de los años setenta y han sido un área significativa de investigación a principios de los años ochenta y a finales de ésta década se empezaron a comercializar algunos productos. Ahora, hay muchas compañías de bases de datos orientadas a objetos comerciales que son la segunda generación de productos. El crecimiento en el número de compañías de este tipo de bases de datos ha sido notable. Así como comunidades de usuarios y vendedores crecen, allí el usuario debe lograr madurar en estos productos para proporcionar sistemas de administración de datos muy robustos.

Los mismos SMBDO se han establecido en áreas tales como el comercio electrónico (e-commerce), administración de datos de productos de ingeniería y bases de datos de propósito especial en áreas como seguridad y medicina. La fuerza del Modelo de Objetos está en las aplicaciones donde existe una necesidad subyacente para manejar relaciones complejas entre los datos de los objetos.

Las bases de datos están siguiendo un camino de maduración similar al de las bases de datos relacionales, en la figura 2 se ilustra la evolución que han tenido las tecnologías de las bases de datos orientadas a objetos.

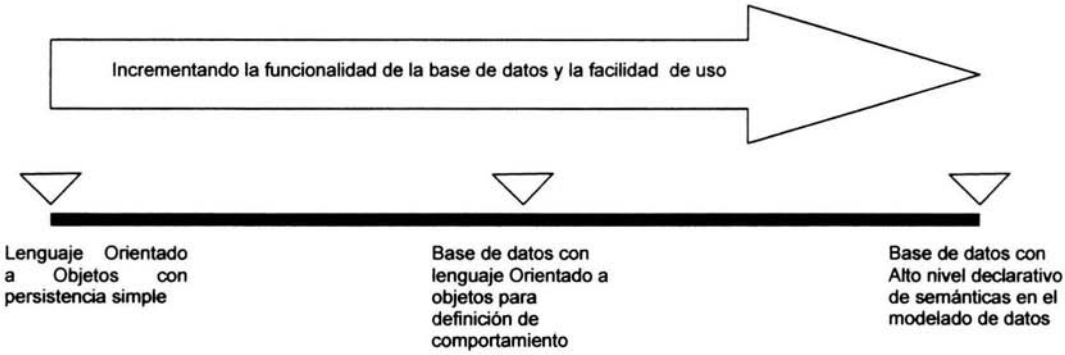


Figura 2.

A la izquierda se tienen los lenguajes orientados a objetos que han sido extendidos para proporcionar persistencia simple con lo que permiten a objetos de aplicación persistir entre sesiones de usuario. La funcionalidad de base de datos mínima es dada en términos de control de concurrencia, transacciones, recuperación, etc. En el punto medio tenemos soporte para muchas de las características de base de datos mencionadas. Productos de base de datos en el punto medio son suficientes para desarrollar razonablemente

aplicaciones de administración de datos complejos. Finalmente, productos de bases de datos con semánticas declarativas tienen la habilidad para reducir enormemente los esfuerzos de desarrollo, así para forzar la uniformidad en la aplicación de estas semánticas. Gran parte de los SMBDO están en la parte central con algunos exhibiendo semánticas declarativas tales como restricciones, reglas de integridad referencial y capacidad de seguridad. En la mayoría de los productos de SMBDO, mucha de la semántica de bases de datos es definida por los programadores usando servicios de bajo nivel proporcionado por la base de datos. El próximo escenario de evolución es más difícil. Conforme se vayan moviendo hacia la derecha, las bases de datos hacen más por el usuario, requiriendo menos esfuerzo para el desarrollo de aplicaciones. Un ejemplo de esto es que los actuales SMBDO proporcionan un mayor número de interfaces de bajo nivel con el propósito de optimizar el acceso a la base de datos. La responsabilidad está completamente sobre el desarrollador para determinar cómo optimizar su aplicación usando estas características. Como los SMBDO evolucionan la tecnología de las bases de datos, los SMBDO deben asumir una mayor parte de la carga para optimización permitiendo al usuario especificar un alto nivel declarativo guiándose sobre qué tipos de optimizaciones necesitan ser ejecutados.

Una guía general para medir la madurez de la base de datos es el grado en el que las funciones tales como optimización de acceso a la base de datos, reglas de integridad, migración de esquemas y base de datos, archivar, operaciones de respaldo y recuperación pueden ser hechas a la medida por el usuario utilizando comandos de alto nivel declarativo para el SMBDO. Hoy en día, la mayoría de los productos de bases de datos orientados a objetos, requieren del desarrollador de aplicaciones para escribir el código para manejar estas funciones.

Otro signo de madurez de una nueva tecnología es el establecimiento de grupos de la industria para estandarizar sobre diferentes aspectos de la tecnología. Ahora observamos un gran interés en el desarrollo de estándares para las bases de datos orientadas a objetos. Por ejemplo, el Grupo de Manejo de Objetos (OMG *Object Managment Group*) es una asociación patrocinada por la industria sin fines de lucro, cuya meta es proporcionar un conjunto de interfaces estándar para la interoperabilidad de componentes de software. Las interfaces están definidas en áreas de comunicaciones (*Object Request Broker*), bases de datos orientadas a objetos, interfaces de usuario orientadas a objetos, etc. Una especificación de un API (*Application Programmers Interface*) de SMBDO está siendo desarrollada por el ODMG (*Object Database Managment Group*, que es un grupo de vendedores de SMBDO) para permitir la portabilidad de aplicaciones entre SMBDO. Otro cuerpo de estándares X3H7, comité técnico bajo X3, ha sido formado para definir estándares en áreas tales como Modelos de Objetos y extensiones de objetos para SQL.

Ahora, vendedores de SMBDO están adicionando más características de bases de datos a sus productos para proporcionar la funcionalidad que debe esperarse de un sistema manejador de base de datos. Esta evolución se mueve hacia el punto medio de la escala evolutiva que se mostró en la figura 2.

1.9.3 Características de las Bases de Datos Orientadas a Objetos

La tecnología de las bases de datos es una conjunción de la programación orientada a objetos y las tecnologías de las bases de datos. La figura 3 ilustra como estos conceptos, juntos, proporcionan lo que conocemos como bases de datos orientadas a objetos.

Quizá la característica más significativa de la tecnología de base de datos orientada a objetos es que combina la programación orientada a objetos con la tecnología de base de datos para proporcionar un sistema de desarrollo de aplicaciones integrado. Hay muchas ventajas para incluir la definición de operaciones con la definición de datos. Primero, las operaciones definidas se emplean ubicuamente y no son dependientes sobre la aplicación de la base de datos particular corriendo en el momento. Segundo, los tipos de datos pueden ser extendidos para soportar datos complejos tales como multimedia (vídeo, sonido, imagen), definiendo nuevas clases de objetos que tienen operaciones para soportar los nuevos tipos de información.

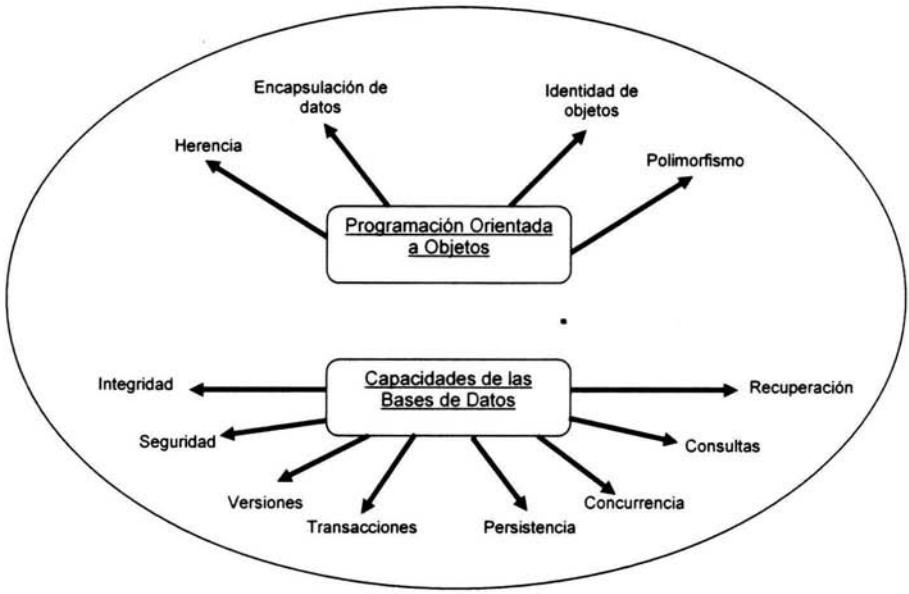


Figura 3. *Compaginación de una Base de Datos Orientada a Objetos.*

Otras ventajas del modelado orientado a objetos son bien conocidas, por ejemplo, la herencia permite desarrollar soluciones a problemas complejos definiendo de forma incremental nuevos objetos en términos de objetos previamente definidos. El polimorfismo y el enlace dinámico permiten definir operaciones para un objeto y después compartir la especificación de la operación con otros objetos. Estos objetos pueden extender más esta operación para proporcionar comportamientos que son únicos para esos objetos. El enlace dinámico determina en tiempo de ejecución, cuál de estas operaciones es la que está siendo ejecutada actualmente, dependiendo las clases de los objetos pedidos para

ejecutar la operación. El polimorfismo y el enlace dinámico son características poderosas de la orientación a objetos que permiten componer objetos para proporcionar soluciones sin tener que escribir un código que es específico a cada objeto. Todas estas capacidades juntas, llegan a proporcionar ventajas productivas para los desarrolladores de aplicaciones de bases de datos.

Una característica única de los objetos es que ellos tienen una identidad que es independiente del estado del objeto. Por ejemplo, si alguien tiene un objeto carro y nosotros volvemos a modelar el carro y cambiamos su apariencia (la máquina, la transmisión, las llantas, etc.) de tal forma que se ve completamente diferente, éste debe ser reconocido aun como el mismo objeto que teníamos originalmente, el identificador sería el número del motor. Dentro de una base de datos orientada a objetos, podemos preguntar siempre, es éste el mismo objeto que tenía previamente, asumiendo recordar la identidad del objeto. La identidad de los objetos les permite estar relacionados, además de compartidos dentro de una red de cómputo distribuida.

Todas estas ventajas apuntan a la aplicación de bases de datos orientadas a objetos para problemas de manejo de información que están caracterizadas por la necesidad de manejar:

- Un gran número de diferentes tipos de datos.
- Un gran número de relaciones entre los objetos.
- Objetos con comportamientos complejos.

Las áreas de aplicación donde este tipo de complejidad existe incluyen la ingeniería, manufactura, simulaciones, automatización de oficinas y sistemas grandes de información.

1.9.4 Aplicación de un SMBDO

El diseño de un modelo de datos orientado a objetos es el primer paso en la aplicación de bases de datos orientadas a objetos para el problema de un área en particular. El desarrollo de un modelo de datos incluye los siguientes pasos principales:

- Identificación de un Objeto.
- Definición del estado de un Objeto.
- Identificación de las relaciones de un Objeto.
- Identificación del comportamiento de un Objeto.
- Clasificación de los Objetos.

Para comenzar a definir el modelo de datos orientado a objetos, el primer paso es simplemente observar y registrar los objetos en el espacio de solución. Hay muchas técnicas que ayudan a este proceso, una de éstas es el lenguaje UML. Por ejemplo, se puede formular una descripción de la solución e identificar los sustantivos que son candidatos para ser objetos en el modelo de datos. Después, se identifican las características de estos objetos; éstas son los atributos del objeto. En forma similar, se examinan las dependencias lógicas entre los objetos identificando los diferentes tipos de asociación. Por ejemplo, la relación de las partes puede ser identificada analizando la

descomposición del sistema en subpartes. Después, se empiezan a enumerar las diferentes respuestas que un objeto tiene para diferentes estímulos. Finalmente, se clasifican los objetos en una estructura de herencia para obtener características de factor común y comportamientos. Estos pasos se efectúan iterativamente hasta que se tenga un modelo de datos completo. En la figura 4 se muestra un modelo de datos para un producto y su descomposición en partes. Cada parte, a su vez, puede descomponerse en subpartes.

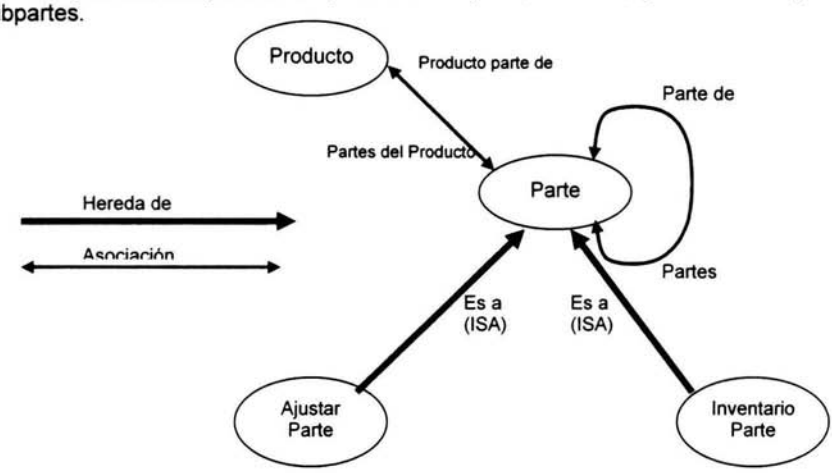


Figura 4. *Esquema de un modelo de datos de partes de un producto.*

Las asociaciones son relaciones bidireccionales entre los objetos. En una base de datos orientada a objetos, las relaciones son mantenidas entre los objetos usando la identidad única del objeto, lo que significa que se pueden cambiar los valores de los atributos de los objetos y no afectar las relaciones entre los objetos.

Una diferencia muy significativa entre las bases de datos y los lenguajes de programación orientados a objetos, es que las bases de datos típicamente proporcionan primitivas de alto nivel para la definición de relaciones entre los objetos. Por lo regular, la implementación de las relaciones es manejada por el SMBDO para mantener la integridad referencial. En adición, los SMBDO permiten definir la cardinalidad de la relación y las restricciones que existen del objeto. La riqueza de semántica de las relaciones está bien controlada por el manejo de complejidad, información altamente relacionada. Desafortunadamente, estas capacidades no son proporcionadas uniformemente por los diferentes productos de bases de datos orientadas a objetos. Un modelo de datos orientado a objetos también permite definir atributos y operaciones para cada uno de los objetos como se muestra en la figura 5.

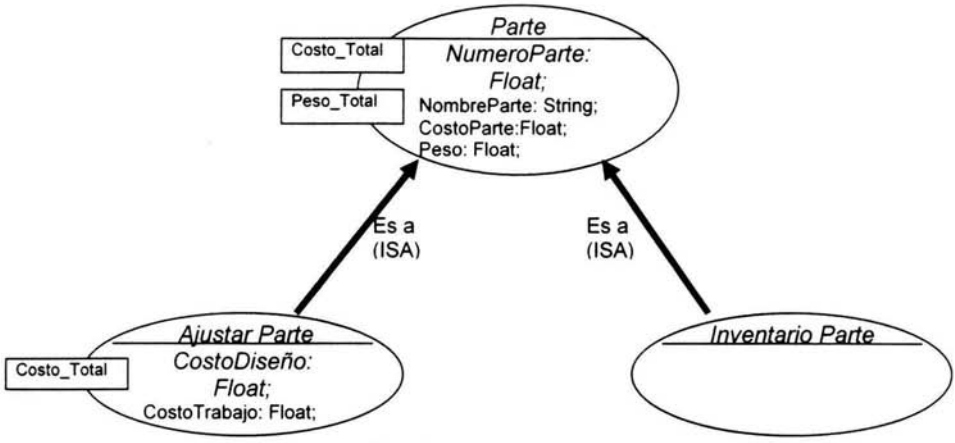


Figura 5. *Atributos y operaciones de los objetos.*

En el ejemplo, los objetos *Ajustar_Parte* e *Inventario_Parte* heredan los atributos cómo las operaciones del objeto *Parte*. El objeto *Ajustar_Parte* define atributos adicionales y escribe otra vez la operación *Costo_Total*. Una ventaja significativa de la herencia es que permite generalizar los objetos al factorizar atributos comunes y operaciones dentro de algún objeto común y, entonces, utilizando la herencia, se comparten estas propiedades comunes. Así si se aumentan más objetos, relaciones y operaciones, la herencia ayuda a reducir la complejidad de definir y mantener esta información.

En el mundo real, el modelo de datos no es estático, debe actualizarse como la información organizacional necesite cambiar y como la información perdida sea identificada. Consecuentemente, la definición de objetos debe ser actualizada periódicamente y las bases de datos que existen migradas para conformar a las nuevas definiciones de objetos. Las bases de datos son semánticamente ricas dando gran cantidad de retos cuando cambian las definiciones de los objetos y la migración de las bases de datos. Las bases de datos orientadas a objetos tienen un mayor reto en el manejo de la migración del esquema porque no es suficiente simplemente migrar la representación de los datos para conformar los cambios en especificaciones de clases. También se debe actualizar el código del comportamiento asociado con cada objeto.

1.10 SMBDO *vs* SMBDR

En la actualidad el uso de bases de datos ha tenido un gran impacto para el manejo de información, y como se ha mencionado en líneas anteriores, los tipos de datos que se desean utilizar para el manejo de información se han vuelto más complejos (imagen, vídeo, sonido, etc.); los SMBDR han sido por mucho tiempo los sistemas que vinieron a solucionar el manejo de la información de forma eficiente, por lo que son de gran valor para las empresas que manejan grandes volúmenes de información, pero su estructura sólo puede soportar tipos de datos simples (enteros, cadenas de caracteres fecha, reales, etc.) que en su momento eran los que se empleaban, los tipos de datos que en la

actualidad se desean utilizar son más complejos como se mencionó anteriormente, por lo que los SMBDR no los pueden soportar, por eso comenzaron a surgir los SMBDO, los cuales tienen todas las propiedades de un SMBD unido con las propiedades de la Orientación a Objetos, y gracias a la fusión que se hizo entre éstas se ha logrado trabajar con tipos de datos complejos que con un SMBDR gracias a la facilidad de la orientación a objetos que permite crea tipos de datos abstractos.

1.10.1 ¿Qué es un SMBDR?

Un sistema manejador de base de datos relacional (SMBDR) es las propiedades de persistencia, recuperación de datos, control de integridad, transparencia de la distribución, control de concurrencia y control de seguridad de un SMBD soportando las características de un modelo de datos Relacional, relaciones, tablas, atributos, vistas e índices.

Un SMBDR permite administrar datos (insertar, actualizar, borrar y consultar) dentro de un esquema relacional, esto se hace a partir de la creación de un modelo relacional, que es el diseño lógico de la base de datos, éste se implementa en el SMBD con funciones de creación, se crean las tablas que formarán la estructura física de la base de datos y con esto se empiezan a administrar los datos que se tienen en la estructura física; se crean índices para mejorar la ejecución de las consultas que se hagan a los datos; se aplican restricciones de Llave Primaria, Nulidad, de Revisión, Unicidad y Referencial (Llave Foránea) a las columnas de las tablas, lo cual permitirá tener consistencia e integridad de la base de datos, se crean vistas a partir de una o varias tablas para representar de diferentes formas los datos que se tienen, de esta manera se manejan de forma eficaz los datos para recuperar la información deseada. El SMBD controla el acceso a los datos de las tablas dependiendo de lo que se desea realizar con ellos, si se manipularán los datos el SMBD no permitirá que otro usuario accese a ellos hasta que se hayan liberado, sólo hasta entonces podrán ser utilizados por los demás usuarios, si los datos solamente son consultados el SMBD permitirá el acceso a todos los usuarios, si algún usuario utiliza un recurso de la base de datos (tabla) para hacerle modificaciones a los datos, y otros usuarios quieren consultar los datos de ese recurso, el SMBD les mostrará los datos que se encontraban antes de que el usuario que lo está manipulando lo utilizara y sólo se mostrarán los cambios hasta que el usuario que los hizo libere el recurso, con esto el SMBDR se encarga de que los datos sean consistentes y no pierdan su integridad.

1.10.2 ¿Qué es un SMBDO?

Un sistema manejador de bases de datos orientado a objetos SMBDO tiene las propiedades de un SMBD mencionadas anteriormente, unidas con los conceptos de la Orientación a Objetos.

Un SMBDO permite administrar una base de datos a partir de un modelo de objetos, una vez que se define el modelo se implementan los objetos que se modelaron a través de clases, las cuales serán el molde para crear los objetos que se van a almacenar en la base de datos, al crearse un objeto el SMBDO le dará un identificador de objeto único (OID) que no depende de los valores de sus atributos, el cual servirá para poder relacionarlo con otros objetos y para localizarlo; en las clases estarán definidos los métodos que utilizará el SMBD para poder manipular los valores de los atributos a través

de mensajes que se manden a los objetos, se podrán crear clases a partir de las ya creadas utilizando el concepto de herencia, esto con la finalidad de hacer más específicas las clases y con esto se podrán implementar de diferente forma las funciones de las nuevas clases a partir de las funciones que heredan. El SMBD controlará el acceso que se haga a los objetos de la base de datos, dependiendo de las operaciones que se desee hacer con ellos, si se manipularan los datos del objeto el SMBD lo aparta y no permite que otro usuario lo utilice hasta que sea liberado por el usuario que lo está manipulando, si sólo se van a consultar los datos del objeto éste no será apartado y lo podrán consultar al mismo tiempo diferentes usuarios. Una de las ventajas de utilizar SMBDO, es que la interacción entre el sistema de base de datos y el lenguaje que se vaya a utilizar para comunicarse con ésta será muy transparente, ya que el lenguaje podrá utilizar las mismas clases de la base de datos.

1.11 Resumen

El principal objetivo de un SMBDO es proporcionar consistencia, datos independientes, seguridad, servicios de administración de datos extensibles y controlados para soportar el paradigma del modelo orientado a objetos. Los SMBDO de hoy proporcionan la mayoría de estas capacidades. Muchos de los productos se encuentran en la segunda generación de SMBDO que han incorporado las lecciones aprendidas de la primera generación de productos. Interpretando el diagrama de evolución de las bases de datos del punto 2.9.2, nos encontramos a la mitad del camino para tener características ricas y poderosos SMBDO en el mercado. Dado el alto grado de interés en las tecnologías orientadas a objetos, existe una estrategia de mercado sustancial para poner a los SMBDO sobre una pista rápida, donde las características y capacidades deben continuar para avanzar en un periodo rápido.

La mayor fuerza de la tecnología de los SMBDO es su habilidad para representar comportamientos complejos. Al incorporar comportamientos dentro de la base de datos, se reduce substancialmente la complejidad de las aplicaciones que las usan. El escenario ideal, la mayoría del código de la aplicación debe repartirse con los datos de entrada y los datos a desplegar. Toda la funcionalidad asociada con la integridad de los datos y el manejo de éstos debe estar definida dentro del modelo de objetos básico. Las ventajas de esta propuesta son:

- Todas las operaciones son definidas una vez y utilizadas de nuevo por todas las aplicaciones.
- Los cambios a una operación afectan a todas las aplicaciones, simplificando el mantenimiento de la base de datos.

Los beneficios de desarrollar aplicaciones de bases de datos orientadas a objetos son el incremento en la productividad resultante del alto grado de código que se puede utilizar de nuevo y la habilidad a enfrentarse con mayor complejidad resultante del aumento del refinamiento de los problemas. También se obtiene aumento en la flexibilidad del diseño debido al polimorfismo y al enlace dinámico. Finalmente, tanto desarrolladores como usuarios resultan beneficiados de la naturalidad y simplicidad de representar datos en objetos.

Capítulo 2. Estándar ODMG

2.1 ODMG

El estándar ODMG fue creado para las bases de datos orientadas a objetos puras, con el objetivo de asegurar la portabilidad de las aplicaciones de un sistema a otro. Uno de los propósitos de ODMG es el de integrar las capacidades de las bases de datos con las capacidades de programación, de tal manera que los objetos creados en la base de datos puedan ser utilizados directamente por los lenguajes de programación, y con esto lograr eliminar la falta de correspondencia que existe entre los sistemas de tipos de ambos lenguajes. El SMBDO extenderá el lenguaje de programación con persistencia, concurrencia, recurrencia, etc. que son las operaciones propias del SMBD.

2.2 Modelo de Objetos

El Modelo de Objetos es importante por que especifica la semántica que se puede definir explícitamente a un SMBDO. Entre otras cosas, la semántica del Modelo de Objetos determina las características de los objetos, cómo pueden ser relacionados con otros objetos y cómo los objetos pueden ser nombrados e identificados.

El Modelo de Objetos especifica los constructores que son soportados por un SMBDO:

- Las primitivas de modelado básicas son el *objeto* y la *literal*. Cada objeto tiene un identificador único. Una literal no tiene identificador.
- Los objetos y las literales pueden ser categorizados por sus *tipos* (types). Todos los elementos de un tipo dado tienen un rango común de estados (*i.e.*, el mismo conjunto de propiedades) y comportamiento común (*i.e.*, el mismo conjunto de operaciones definidas). Un objeto a veces es referido como una *instancia* de su tipo.
- El estado de un objeto es definido por los valores llevados por una serie de *propiedades*. Estas propiedades pueden ser los *atributos* del mismo objeto o *relaciones* entre el objeto y uno o más objetos. Por lo regular los valores de las propiedades pueden cambiar en el tiempo.
- El comportamiento de un objeto es definido por el conjunto de *operaciones* que pueden ser ejecutadas sobre o por el objeto. Las operaciones pueden tener una lista de parámetros de entrada y salida, cada uno con un tipo especificado. Cada operación puede regresar también un resultado tipificado.
- Una *base de datos* que almacena objetos, permite que éstos sean accesados por múltiples usuarios y aplicaciones. Una base de datos está basada sobre un *esquema* que es definido en ODL y contiene instancias de los tipos definidos por su esquema.

2.2.1 Tipos

Hay dos aspectos para definir un tipo. Un tipo tiene una especificación externa y una o más implementaciones. La especificación define las características externas del tipo, éstas son los aspectos visibles al usuario del tipo: las operaciones que pueden ser

invocadas sobre sus instancias, las propiedades, o variables de estado, cuyos valores pueden accesarse, y cualquier excepción que pueda ser levantada por sus operaciones. La implementación define los aspectos internos de los objetos del tipo: la implementación de las operaciones del tipo y otros detalles internos.

Una especificación externa de un tipo consiste en una abstracción, la descripción de la implementación independiente de las operaciones, excepciones y propiedades que son visibles a usuarios del tipo. Una definición de interfaz es una especificación que define sólo el comportamiento abstracto de un tipo de objeto. Una definición de clase es una especificación que indica el comportamiento y estado abstracto de un tipo de objeto. Una literal define sólo el estado abstracto de un tipo de literal.

Una implementación de un tipo de objeto consiste en una representación y un conjunto de métodos. La representación es una estructura de datos que es derivada desde el estado abstracto del tipo mediante un lenguaje de enlace. Para cada propiedad contenida en el estado abstracto hay una variable de instancia de un tipo definido apropiado. Los métodos son cuerpos de procedimientos que son derivados del comportamiento abstracto del tipo mediante el lenguaje de enlace. Para cada una de las operaciones definidas en el comportamiento abstracto del tipo es definido un método. Un método puede leer o modificar la representación del estado de un objeto o invocar operaciones definidas sobre otros objetos. Un tipo puede tener más de una implementación, aunque sólo una de éstas es usualmente utilizada en cualquier programa particular.

2.2.2 Subtipificación y herencia

ODMG incluye la herencia basada en relaciones tipo-subtipo. El supertipo es el tipo más general; el subtipo es el más especializado. Una interfaz del subtipo puede definir características en adición a las ya definidas en sus supertipos. Estos nuevos aspectos de estado o comportamiento se aplican sólo a instancias del subtipo. Una interfaz del subtipo también puede ser refinada para especializar el estado y el comportamiento.

ODMG soporta la herencia múltiple del comportamiento del objeto. Por lo que es posible que un tipo pueda heredar operaciones que tienen el mismo nombre, pero diferentes parámetros, de dos diferentes interfaces.

Las clases son tipos que son directamente instanciables, instancias de éstas pueden ser creadas por el programador, a las interfaces no se les puede generar una instancia directamente.

La subtipificación pertenece sólo a la herencia de comportamiento; así las interfaces pueden heredar de otras interfaces y las clases pueden heredar de las interfaces. Debido a las ineficiencias y ambigüedades de la herencia de estado múltiple, las interfaces no pueden heredar de las clases, ni tampoco pueden las clases heredar de otras clases.

ODMG define una relación EXTENDS para la herencia de estado. Esta relación sólo se aplica a tipos objeto; así sólo las clases y los que no son literales pueden heredar el estado. La relación EXTENDS es una simple relación de herencia entre dos clases, por lo

cual la clase subordinada hereda todo de las propiedades y todo el comportamiento de la clase que ésta extiende. Este tipo de relación es transitiva.

2.2.3 Extents (Extensiones)

El extent de un tipo es el conjunto de todas las instancias del tipo dentro de una base de datos particular.

2.2.4 Keys (Llaves)

En algunos casos las instancias individuales de un tipo pueden ser identificadas de manera única mediante los valores que contienen éstas en alguna propiedad o conjunto de propiedades. Estas propiedades de identificación son llamadas keys.

2.2.5 Objetos

En el capítulo dos se habla de los objetos como entes, que combinan las propiedades de los métodos con los datos de sus atributos, en el sentido de que tengan determinado comportamiento y conserven su estado.

Los objetos son componentes que tienen un comportamiento determinado y se pueden relacionar unos con otros para dar solución a algún problema en particular.

Los objetos son creados invocando operaciones de creación sobre interfaces proporcionadas al programador por la implementación del lenguaje de enlace. Todos los objetos tienen la interfaz Object, que es implícitamente heredada por las definiciones de los objetos de todos lo usuarios.

Porque todos los objetos tienen identificadores únicos (OID's), un objeto puede distinguirse siempre de los demás dentro de su dominio de almacenamiento (base de datos). Todos los identificadores de los objetos de la base de datos son únicos, en relación con otros. Un objeto retiene el mismo identificador durante su tiempo de vida, así el valor de este no debe cambiar. Un identificador es usado comúnmente como punto medio por un objeto para referenciar a otro. Los identificadores del objeto son creados por el SMBDO, no por las aplicaciones.

Además de asignársele a un objeto un identificador por parte del SMBDO, un objeto puede tener uno o más nombres que son significativos para el programador o usuario final. El SMBDO proporciona una función que se usa para mapear el nombre del objeto a un objeto. La aplicación puede referirse a su conveniencia a un objeto por su nombre, el SMBDO aplica la función para determinar el identificador del objeto que localiza al objeto deseado.

El tiempo de vida de un objeto determina cómo la memoria y la asignación de almacenamiento para los objetos son empleados. Dicho tiempo es especificado cuando un objeto es creado. Los tiempos de vida soportados por el Modelo de Objetos son los siguientes:

- *transitorio*. Este tipo de objeto es asignado a la memoria que es empleada por el lenguaje de programación en tiempo de ejecución del sistema. Esa memoria es liberada cuando el tiempo de ejecución del sistema, o un proceso que creó al objeto, se acaba.
- *persistente*. Este tipo de objetos es asignado a memoria y al almacén manejado por el SMBDO en tiempo de ejecución del sistema. Estos objetos continúan existiendo después de que el procedimiento o proceso que los creó termina.

2.2.6 Colección de Objetos

En ODMG, las instancias de colección de objetos están compuestas de distintos elementos, cada uno de los cuales puede ser una instancia de un tipo atómico, otra colección o una literal. Una característica distintiva muy importante de una colección es que todos los elementos de ésta deben ser del mismo tipo. Las colecciones que soporta el Modelo de Objetos ODMG son:

- *Set<t>*. Todos los elementos de esta colección son del mismo tipo t; esta es una colección no ordenada de elementos, sin permitir duplicados.
- *Bag<t>*. Esta es una colección no ordenada de elementos que puede contener duplicados.
- *List<t>*. Esta es una colección ordenada de elementos.
- *Array<t>*. Es una colección de elementos ordenada de tamaño dinámico, y los elementos pueden ser localizados por posición.
- *Dictionary<t,v>*. Es una secuencia no ordenada de pares (clave, valor) con claves no duplicadas.

Todos los objetos estructurados soportan la interfaz Object. El Modelo de Objetos ODMG define los siguientes objetos estructurados: Date, Interval, Time, Timestamp.

2.2.7 Literales

Las literales no tienen identificadores de objetos. El Modelo de Objetos soporta cuatro tipos de literales:

- Literal atómica (long, short, unsigned long, unsigned short, float, double, boolean, octet, char, string, enum).
- Literal de colección (set<t>, bag<t>, list<t>, array<t>, dictionary<t,v>).
- Literal estructurada (date, interval, time, timestamp, y las definidas por el usuario).
- Literal nula.

2.3 Modelando el Estado - Propiedades

Un tipo define un conjunto de propiedades a través de las cuales los usuarios pueden accesar, y en algunos casos manipular directamente el estado de instancias del tipo. Dos especies de propiedades son definidas en el Modelo de Objetos ODMG: atributo y relación. Un atributo es de un tipo. Una relación es definida entre dos tipos, los cuales deben tener cada uno instancias que son referenciables por identificadores de objetos.

Así los tipos de literal, porque no tienen identificadores de objetos, no pueden participar en las relaciones.

Las declaraciones de atributos en una interfaz definen el estado abstracto de un tipo. Es importante notar que un atributo no es lo mismo que una estructura de datos. Un atributo es abstracto, mientras una estructura de datos es una representación física. Asimismo, es común para un atributo ser implementado como estructuras de datos y, a veces, es apropiado para un atributo ser implementado como un método.

Las relaciones están definidas entre tipos. El Modelo de Objetos ODMG soporta sólo relaciones binarias, es decir, relaciones entre dos tipos. Una relación binaria puede ser uno-a-uno, una-a-muchos o muchos-a-muchos, dependiendo sobre como muchas instancias de cada tipo participan en la relación.

Las relaciones en esta versión del Modelo de Objetos no están nombrados y no son "primera clase." Una relación no es un objeto y no tienen identificador de objeto, es definida implícitamente en la declaración de caminos navegables que habilitan aplicaciones para usar las conexiones lógicas entre los objetos participantes en la relación. Los caminos navegables son declarados en pares, uno para cada dirección de navegación de la relación binaria.

El SMBDO es responsable de mantener la integridad referencial de las relaciones. Esto significa que si un objeto que participa en una relación es borrado, entonces cualquier camino navegable a este objeto puede ser también borrado.

2.4 Modelando el Comportamiento - Operaciones

Otra característica de un tipo es su comportamiento, el cual es especificado como un conjunto de operaciones de signaturas o firmas. Cada signatura define el nombre de una operación, el nombre y tipo de cada uno de sus argumentos, los tipos de valores retornados y los nombres de cualquiera de las excepciones que la operación puede levantar. Una operación es definida sólo sobre un tipo simple. Un nombre de operación necesita ser único sólo dentro de una definición de tipo simple. Así, diferentes tipos pueden tener operaciones definidas con el mismo nombre. Los nombres de estas operaciones son mencionados para ser cargados. Cuando se invoque una de éstas, una operación específica debe ser seleccionada para su ejecución.

2.5 Modelo de Excepción

ODMG soporta manejadores de excepciones dinámicamente anidados, usando un modelo de terminación de manejo de excepción. Las operaciones pueden levantar excepciones y las excepciones pueden comunicar algún resultado. Las excepciones en el Modelo de Objetos son asimismo objetos y tienen una interfaz que les permite estar relacionadas con otras excepciones en una jerarquía de generalización-especialización.

2.6 Locking y Control de Concurrencia

ODMG usa un método convencional basado en candados para el control de concurrencia. Este método proporciona un mecanismo para el acceso exclusivo a los objetos. El SMBDO soporta la propiedad de serializabilidad monitoreando solicitudes para candados, y sólo da concesión a un candado si no existe conflicto de candados. Como resultado, el acceso a la base de datos es coordinado a través de múltiples transacciones y una vista consistente de la base de datos es mantenida por cada transacción.

ODMG soporta el control de concurrencia tradicional pesimista como su política por omisión, pero no excluye un SMBDO de soportar un rango amplio de políticas de control de concurrencia.

Los siguientes candados son soportados por ODMG:

- ***Read (Lectura)***. Estos candados permiten el acceso particionado a un objeto.
- ***Write (Escritura)***. Estos candados indican el acceso exclusivo a un candado.
- ***Upgrade.*** Estos candados son utilizados para prevenir una forma de estancamiento que ocurre cuando dos procesos obtienen candados de lectura sobre un objeto y entonces intentan obtener candados de escritura sobre el mismo objeto.

Los candados implícitos son adquiridos durante el curso de la trayectoria de un objeto gráfico. Por ejemplo, los candados de lectura son obtenidos cada vez que un objeto es accesado y los de escritura se obtienen cada vez que un objeto es modificado. En este caso, la operación no especificada es ejecutada en orden para obtener el candado sobre algún objeto. Sin embargo, los candados explícitos, son adquiridos solicitando expresamente un candado específico sobre un objeto en particular. Estos candados son obtenidos usando las operaciones lock y try_lock definidas en la interfaz Object. Los candados upgrade sólo pueden ser solicitados explícitamente por las operaciones, a diferencia de los otros dos tipos de candados.

Por omisión, todos los candados (lectura, escritura y actualización) son tomados hasta que la transacción esté cometida o sea abortada.

2.7 Modelo de Transacción

Los programas que usan objetos persistentes son organizados dentro de transacciones. La administración de transacciones es una importante funcionalidad del SMBDO, fundamental para la integridad, particionalidad y recuperación de una base de datos. Cualquier acceso, creación, modificación y borrado de objetos persistentes debe estar hecho dentro de una transacción.

Una transacción es una unidad de lógica para que un SMBDO garantice:

- *Atomicidad.* Significa que la transacción termina o no tiene efecto en todo.

- *Consistencia.* Significa que una transacción toma la base de datos desde un estado internamente consistente a otro estado internamente consistente.
- *Aislamiento.* Esta garantiza que ningún otro usuario de la base de datos observa los cambios hechos por una transacción hasta que la transacción sea cometida.
- *Durabilidad.* Significa que los efectos de transacciones cometidas son preservados, aun si hay fallas del medio de almacenamiento, pérdida de memoria o caídas del sistema.

Cuando una transacción se compromete, todos los cambios hechos durante ésta son permanentemente instalados en la base de datos y se hacen visibles a otros usuarios de la base de datos. Cuando una transacción es abortada, ninguno de los cambios hechos en ésta, son instalados en la base de datos, incluyendo los cambios hechos previos al momento de abortar. La ejecución de transacciones concurrentes puede producir resultados que son indistinguibles de otros que pudieron haber sido obtenidos si las transacciones hubieran sido ejecutadas serialmente. Esta propiedad es, a veces, llamada serializabilidad.

2.7.1 Transacciones y Procesos

ODMG asume una secuencia lineal de ejecución de transacciones dentro de un hilo de control; esto es, hay exactamente una transacción en curso para un hilo.

Una transacción corre sobre una base de datos lógica. Note que una base de datos lógica puede ser implementada como una o más bases de datos físicas, posiblemente distribuidas sobre una red. El modelo de transacciones no requiere ni previene soporte para transacciones que cruzan múltiples hilos, múltiples espacios de direcciones, o más de una base de datos lógica.

En el Modelo de Objetos actual, los objetos transitorios en un espacio de direcciones no son sujeto a semánticas de transacción. Esto significa que abortando una transacción no restaura el estado de los objetos transitorios modificados.

2.7.2 Operaciones de Transacción

Hay dos tipos que son definidos para soportar la actividad de transacción dentro de un SMBDO: TransactionFactory y Transaction.

El primer tipo es usado para crear objetos transacciones. Las siguientes operaciones son definidas en la interfaz TransactionFactory:

- *new.* Esta operación crea objetos Transaction.
- *current.* Esta operación retorna la transacción asociada con el hilo de control actual. Si no hay tal asociación, la operación retorna *nil.*

Una vez que una transacción es creada, ésta se manipula usando la interfaz Transaction, la cual tiene definidas las siguientes operaciones:

- *begin.* Esta operación se utiliza para abrir una transacción.

- *commit*. esta operación provoca que los objetos persistentes creados o modificados durante una transacción se escriban en la base de datos y seran accesibles para otras transacciones corriendo sobre la base de datos. Todos los candados son liberados.
- *abort*. Esta operación hace que el objeto Transaction termine y sea cerrado. La base de datos regresa al estado previo en que estaba al comienzo de la transacción. Todos los candados son liberados.
- *checkpoint*. Esta operación es equivalente a la operación commit, seguida de una operación begin, excepto que los candados no son liberados. Por lo tanto, esto hace que el objeto sea modificado y dado de alta en la base de datos y retenga todos los candados para la transacción.

Las operaciones son aplicadas siempre a la base de datos durante una transacción. Por lo tanto, para ejecutar cualquier operación, un objeto Transaction debe estar asociado con el hilo actual. La operación join asocia el hilo actual con el objeto Transaction. Si éste es abierto, las operaciones de la base de datos pueden ser ejecutadas; en otro caso se levanta la excepción TransactionNotInProgress.

Si una implementación permite actividades múltiples a objetos Transaction existentes, las operaciones de join y leave permiten al hilo alternar entre éstos. Para asociar el hilo actual con otro objeto Transaction, sólo se ejecuta la operación join sobre el nuevo objeto Transaction. Si es necesario, la operación leave es ejecutada automáticamente para disociar el hilo actual de su objeto Transaction actual. Moviéndose de un objeto Transaction a otro, no comete o aborta un objeto Transaction. Cuando un hilo no tiene un objeto Transaction, la operación leave es ignorada.

Después de que una transacción es completada, para continuar ejecutando las operaciones de la base de datos, otro objeto Transaction es asociado con el hilo actual, o la operación begin debe ser aplicada al objeto Transaction actual para abrirlo otra vez.

2.8 Operaciones de la Base de Datos

Un SMBDO puede manejar una o más bases de datos lógicas, cada una de las cuales puede ser almacenada en una o más bases de datos físicas. Cada base de datos lógica es una instancia del tipo Database, la cual es proporcionada por el SMBDO. Las instancias del tipo Database son creadas usando la interfaz DatabaseFactory.

Una vez que el objeto Database es creado usando la operación new, éste es manipulado usando la interfaz Database, la cual tiene definidas las siguientes operaciones:

- *open*. Esta operación es invocada, con un nombre de base de datos como su argumento, antes de que cualquier acceso pueda ser hecho a los objetos persistentes en dicha base de datos.
- *close*. Esta operación debe ser invocada cuando un programa ha completado todo el acceso a la base de datos.
- *lookup*. Esta operación encuentra el identificador del objeto con el nombre proporcionado como argumento a la operación.
- *bind*. Esta operación se utiliza para enlazar un nombre a un objeto.
- *unbind*. Esta operación les quita el nombre a los objetos.

- *schema*. Esta operación accesa el meta objeto raíz que define el esquema de la base de datos. El esquema de un SMBDO está contenido dentro de un meta objeto Module.

El tipo Database puede soportar también operaciones diseñadas para la administración de la base de datos, por ejemplo, *create*, *delete*, *move*, *copy*, *reorganize*, *verify*, *backup*, *restore*.

2.9 Lenguaje de Consulta de Objetos OQL (Object Query Language)

El diseño está basado en los siguientes principios y suposiciones:

- OQL trabaja con el Modelo de Objetos de ODMG.
- OQL es muy parecido a SQL 92. Las extensiones conciernen a notaciones orientadas a objetos.
- OQL proporciona primitivas de alto nivel para manejar conjuntos de objetos, así como también proporciona primitivas para el manejo de estructuras, listas y arreglos y trata tales constructores con la misma eficiencia.
- OQL es un lenguaje funcional donde los operadores pueden ser compuestos libremente, tan grandes como los operandos respeten el tipo del sistema.
- OQL no está computacionalmente completo. Es un lenguaje de consulta de simple uso que proporciona el acceso fácil a un SMBDO.
- Basado en el mismo tipo del sistema, OQL puede ser invocado dentro de cualquier lenguaje de programación para el cual un enlazador de ODMG es definido, e inversamente, OQL puede invocar operaciones programadas en esos lenguajes.
- OQL no proporciona la actualización de los operadores explícitamente, sino que invoca operaciones definidas sobre los objetos para este propósito, y así no se violan las semánticas de un SMBDO el cual, por definición, es manejado por los "métodos" definidos sobre los objetos.
- OQL proporciona acceso declarativo hacia los objetos.
- Las semánticas formales de OQL pueden ser definidas fácilmente.

2.9.1 Consulta: Entrada y Resultado

Como un lenguaje stand-alone, OQL permite consultar objetos comenzando desde sus nombres, los cuales actúan como puntos de entrada en una base de datos. Un nombre puede denotar cualquier especie de objeto, es decir, atómico, estructura, colección o literal.

Como un lenguaje inmerso, OQL permite consultar objetos que son soportados por el lenguaje nativo a través de expresiones de átomos, estructuras, colecciones y literales. Una consulta OQL es una función que libera un objeto cuyo tipo puede ser inferido desde el operador contribuyendo a la expresión de la consulta.

Se tiene un esquema en el cual se definen los tipos Persona y Empleado.

```
class Persona
(       extent Personas)
```

```
{
        attribute string nombre;
        attribute string sexo;
        attribute date  fechaNacimiento;
        attribute float   salario;
        void edad(in date fechaNacimiento);
};

class Empleado extends Persona
(       extent  Empleados)
{
        relationship set<Empleado> subordinados;
        void antigüedad(in date fechaInicio);
};
```

Estos tipos tienen las extensiones Personas y Empleados respectivamente. El tipo Persona define los atributos nombre, sexo, fechaNacimiento y salario, además de la operación edad. El tipo Empleado, el cual es un subtipo de Persona, define la relación subordinados y la operación antigüedad. Algunos ejemplos son:

Ej. 1
```
select distinct x.edad
from Personas x
where x.nombre = "Pablo"
```

Este ejemplo selecciona el conjunto de edades de todas la personas cuyo nombre es Pablo, regresando una literal de tipo set<integer>.

Ej. 2
```
select distinct struct(a:x.edad, s:x.sexo)
from Personas x
where x.nombre = "Pablo"
```

Este ejemplo hace lo mismo que el anterior, pero para cada persona se construye una estructura conteniendo la edad y el sexo, regresando una literal de tipo set<struct>.

Con OQL se pueden utilizar un select-from-where dentro de la cláusula select, así como en la cláusula from para realizar consultas más complejas.

Además en OQL no siempre se tiene que utilizar una cláusula select-from-where, se puede escribir una consulta dando tan solo el nombre que identifica a un objeto, por ejemplo (del esquema mencionado anteriormente):

```
Personas
```

Este enunciado nos da el conjunto de todas las personas.

2.9.2 Tratando con la Identidad del Objeto

El lenguaje de consulta soporta tanto objetos que tienen OID como literales, dependiendo de la forma en que estos objetos son construidos o seleccionados.

Para crear un objeto con identidad es usado un tipo de constructor de nombre. Por ejemplo, para crear un objeto Persona definido en el ejemplo anterior, simplemente se escribe:

Persona(nombre: "Pablo", sex: "M", fechaNacimiento: "14/10/45", salario: 100,000).

Los parámetros en los paréntesis permiten inicializar ciertas propiedades del objeto. Aquellas que no están inicializadas explícitamente, se les asigna un valor por omisión. Existen además expresiones de construcción que producen objetos sin identidad. Por ejemplo:

```
struct(a:10, b:"Pablo")
```

crea una estructura con dos campos.

También se pueden construir objetos en lugar de solo calcular literales, retomando los ejemplos 1 y 2. Por ejemplo, asumiendo que están definidos los tipos de objetos siguientes:

```
typedef set<integer> vectint;
interface stat{
        attributes
                attribute Short a;
                attribute Char c;
};
typedef bag<stat> stats;
```

se pueden realizar las siguientes consultas:

```
vectint( select distinct x.edad
        from Personas x
        where x.nombre = "Pablo" )
```

la cual regresa un objeto del tipo vectint y

```
stats( select stat( a: x.edad, s: x.sex)
        from Personas x
        where x.nombre = "Pablo" )
```

la cual regresa un objeto del tipo stats.

Las expresiones de extracción pueden regresar:

- Una colección de objetos con identidad.
- Un objeto con identidad.
- Una colección de literales.
- Una literal.

Por lo que el resultado de una consulta es un objeto con o sin identidad; algunos objetos son creados por el intérprete del lenguaje de consulta y otros producidos desde la base de datos actual.

2.9.3 Expresiones de navegación

En las bases de datos se necesita una forma de navegar para poder llegar a los datos que se necesitan a través de los objetos, en OQL se usa la notación de punto "." (o indiferentemente el apuntador "->"), el cual permite llegar a objetos complejos, además de seguir las relaciones simples. Por ejemplo, se tiene un objeto Persona p y se desea conocer el nombre de la ciudad donde está viviendo el cónyuge de la persona.

```
p.conyuge.direccion.ciudad.nombre
```

Esta consulta comienza desde un objeto Persona, obtiene a su cónyuge, después entra al atributo complejo de tipo dirección para obtener el objeto ciudad, cuyo nombre es entonces accesado, esta relación es de 1 a 1. Para una relación del tipo n a p no podemos escribir la consulta de la misma forma. Siguiendo con el ejemplo, ahora se desea obtener los nombres de los hijos de la persona p. Intuitivamente, el resultado debe ser una colección de nombres, en este caso se utiliza la cláusula *select-from-where* para el manejo de las colecciones de igual forma que en SQL.

Ejemplo:
```
select c.nombre
from p.hijos c
```

El resultado de esta consulta es un valor del tipo Bag<String>.

De esta manera podemos navegar de un objeto a otro siguiendo cualquier relación y entrando en subvalores complejos de un objeto. Por ejemplo, se desea conocer el conjunto de direcciones de los hijos de cada persona de la base de datos. Se sabe que la colección llamada Personas contiene todas las personas de la base de datos, se deben recorrer ahora las dos colecciones: Personas y Personas.hijos, éstas deben aparecer en la parte del from. En OQL, una colección en la parte del *from* se puede derivar desde una previa siguiendo una dirección que comienza desde ésta. Por ejemplo:

```
select c.direccion
from Personas p,
     p.hijos c
```

Esta consulta revisa todos los hijos de todas las personas. Su resultado es un valor cuyo tipo es Bag <Direccion>.

La cláusula *where* se puede utilizar para definir un predicado, por lo que sirve para seleccionar sólo los datos que correspondan con el predicado. Del ejemplo anterior podemos restringir el resultado a sólo las personas que viven en Calle Principal y tienen al menos dos hijos. Además, sólo se está interesado en las direcciones de los hijos que no viven en la misma ciudad de sus padres. La sentencia quedaría de la siguiente manera:

```
select c.direccion
from Personas p,
     p.hijos c
where p.direccion.calle = "Calle Principal" and
      count(p.hijos) >=2 and
      c.direccion.city != p.direccion.city
```

En la cláusula *from*, las colecciones que no están directamente relacionadas pueden declararse también. Como en SQL, esto permite el cálculo de joins entre estas colecciones. El siguiente ejemplo selecciona a las personas que tienen el nombre de una flor, asumimos que existe un conjunto de todas las flores llamado Flores.

```
select p
from Personas p, Flores f
where p.nombre = f.nombre
```

2.9.4 Valores Nulos

El resultado de accesar una propiedad del objeto null es UNDEFINED. Las reglas para manejar UNDEFINED son:

- Las operaciones de . y -> (acceso a propiedades) aplicado a un operando hacia la izquierda UNDEFINED produce como resultado UNDEFINED
- Las operaciones de comparación (=, !=, <, >, <=, >=) con uno o ambos operandos como UNDEFINED produce como resultado Falso.
- La función is_undefined(UNDEFINED) regresa Verdadero, la función is_defined(UNDEFINED) regresa Falso.
- Cualquier operación con algún operando UNDEFINED resulta un error en tiempo de ejecución.

2.9.5 Invocación de un Método

OQL permite llamar un método con o sin parámetros donde el tipo del resultado debe concordar con el tipo esperado en la consulta. La notación para llamar a un método es exactamente igual que para accesar un atributo o cruzar una relación, en el caso donde el método no tenga parámetros. Si tienen que llevar parámetros, éstos se ponen entre paréntesis. Esta sintaxis flexible libera al usuario a conocer si la propiedad es almacenada (un atributo) o calculada (un método). El ejemplo siguiente regresa un bag conteniendo la edad de los hijos mayores de todas las personas que se llamen "Paul".

```
select max(select c.edad from p.hijos c)
from Personas p
where p.nombre = "Paul"
```

Un método también puede regresar un objeto complejo o una colección, entonces su llamado puede estar implantado en una expresión de dirección compleja.

2.9.6 Polimorfismo

Una gran contribución de la orientación a objetos es la posibilidad de manipular colecciones polimorfas y gracias al mecanismo late binding se llevan a cabo acciones genéricas sobre los elementos de estas colecciones.

Una consulta es una expresión cuyos operadores trabajan sobre operandos tipificados, ésta es correcta si los tipos de los operandos empalman con aquéllos requeridos por los operadores. Así, una condición necesaria para optimizar consultas eficientes es que OQL sea un lenguaje de consulta tipificado. Cuando una colección polimórfica es filtrada, sus elementos son conocidos estáticamente por ser de esa clase. Esto significa que una propiedad de una subclase (atributo o método) no puede ser empleada para tal elemento, excepto en dos importantes casos: late binding a un método específico o la indicación de clases explícitas.

2.9.6.1 Late Binding

Dar las actividades de cada persona:

select p.actividades
from Personas p

donde actividades es un método que tiene tres encarnaciones, dependiendo de la especie de persona (Empleado, Estudiante o Persona) de la p actual, la encarnación correcta es llamada. Si p es un empleado, OQL llama a ésta operación definida para Empleados; o si p es un estudiante, OQL llama la operación actividades definida para Estudiantes, o si p es una persona, OQL llama el método actividades del tipo Persona.

2.9.6.2 Indicador de Clases

Para recorrer la jerarquía de las clases, un usuario explícitamente puede declarar la clase de un objeto que no puede ser inferido estáticamente. El evaluador tiene que revisar en tiempo de ejecución que este objeto corresponda a la clase indicada. Por ejemplo, asumiendo que sabemos que sólo los estudiantes dedican su tiempo en seguir un curso de estudio, podemos seleccionar estas personas y obtener su grado. Explícitamente indicamos en la consulta que estas personas son de la clase Estudiante. Por ejemplo:

select ((Estudiante)p).grade
from Personas p
where "course of study" in p.actividades

2.9.7 Definición del Lenguaje

OQL es un lenguaje de expresión. Una expresión de consulta es construida de operandos tipificados compuestos recursivamente por los operadores. Usaremos el término expresión para designar una consulta válida. Una expresión regresa un resultado que puede ser un objeto o una literal.

OQL es un lenguaje tipificado, esto significa que cada expresión de consulta tiene un tipo, el cual puede ser derivado de la estructura de la expresión de consulta, del esquema de declaraciones de tipo, y al tipo de los objetos y literales nombrados.

Para cada expresión de consulta, se dan las reglas que permiten (1) revisar para el tipo correcto y (2) substraer el tipo de la expresión desde el tipo de las subexpresiones.

Para las colecciones se necesita la siguiente definición: los tipos t1, t2, ..., tn son compatibles si los elementos de estos tipos pueden ponerse en la misma colección definida en la sección del modelo de objetos.
La compatibilidad es definida recursivamente como sigue:

(1) t es compatible con t.
(2) Si t es compatible con t', entonces
    set(t) es compatible con set(t')
    bag(t) es compatible con bag(t')
    list(t) es compatible con list(t')
    array(t) es compatible con array(t')
(3) Si existe t tal que t es un supertipo de t_1 y t_2, entonces t_1 y t_2 son compatibles. Esto en particular significa que:

- Los tipos literal no son compatibles con los tipo objeto.
- Tipos de literal atómica sólo son compatibles si son de los mismos.
- Tipos de literal estructurada son compatibles sólo si tienen un ancestro común.
- Colecciones de literales son compatibles si son de la misma colección y los tipos de sus miembros son compatibles.
- Los objetos atómicos son compatibles sólo si tienen un ancestro común.
- Colecciones de objetos son compatibles si son de la misma colección y los tipos de sus miembros son compatibles.

Note que si $t_1, t_2, ..., t_n$ son compatibles, entonces existe una t única tal que:

(1) $t > t_i$ para toda i
(2) Para toda t' tal que t' != t y $t' > t_i$ para toda i, $t' > t$.

Esta t es denotada $lub(t_1, t_2..., t_n)$.

2.9.7.1 Consultas

Una **consulta** es una expresión de consulta sin variables relacionadas.

Definición de Consulta Nombrada

La siguiente semántica registra la definición de la función con nombre id en el esquema de la base de datos.

define [query] $id(x_1, x_2, ..., x_n)$ as $e(x_1, x_2, ..., x_n)$,

donde id es un identificador, e es una expresión OQL y (x_1, x_2,..., x_n) son variables libres en la expresión e;

id no puede ser un objeto nombrado, un nombre de un método, un nombre de función o el nombre de una clase en el esquema, en tal caso habría un error.

Una vez que la definición haya sido hecha, cada vez que se compila y evalúa una consulta y se encuentra una expresión de función, si ésta no puede ser directamente evaluada o relacionada a una función o método, el compilador/intérprete reemplaza id por la expresión e. Así actúa como un mecanismo de vista.

Las definiciones de consultas son persistentes, hasta que son anuladas por una nueva definición con el mismo nombre o borradas de la siguiente manera:

delete definition id

Las definiciones de consultas no pueden ser sobrecargadas, si hay una definición de id con n parámetros y redefinimos id con p parámetros, p diferente de n, esto se interpreta como una nueva definición de id y anula la definición previa.

Los paréntesis pueden ser opcionales cuando la definición de una consulta nombrada no tiene parámetros.

2.9.7.2 Expresiones Elementales

Literales atómicas

Si l es una literal atómica, entonces l es una expresión cuyo valor es la literal misma. Las literales tienen la sintaxis usual:

- Literal object: nil.
- Literal boolean: false, trae.
- Literal integer: secuencia de dígitos.
- Literal float: mantissa/exponent. El exponente es opcional.
- Literal character: caracter entre comillas simples.
- Literal string: Cadena de caracteres entre comillas dobles.

Objetos nombrados

Si e es un nombre de un objeto, entonces e es una expresión. Este regresa la entidad atada al nombre. El tipo de e es del tipo del objeto nombrado declarado en el esquema de base de datos.

Ejemplo:
Estudiantes

Esta consulta regresa el conjunto de estudiantes. Se asume que existe un nombre Estudiantes correspondiendo a la extensión de los objetos de la clase Estudiante.

Variable iteradora

Si x es una variable declarada en la parte *from* de un *select-from-where*, entonces x es una expresión cuyo valor es el elemento actual de la iteración sobre la colección correspondiente.
Si x es declarada en la parte *from* de una expresión *select-from-where* en una sentencia de la forma:

e as x
o
e x
o
x in e

donde e es del tipo collection(t), entonces x es del tipo t.

Consultas nombradas

Si $q(x_1, x_2,..., x_n)$ as $e(x_1, x_2,..., x_n)$ es una expresión de definición de consulta donde e es una expresión de tipo t con variables libres $(x_1, x_2,..., x_n)$, entonces $q(x_1, x_2,..., x_n)$ es una expresión de tipo t.

2.9.7.3 Expresiones de Construcción

Construyendo objetos

Si t es un tipo nombre, $p_1, p_2, ..., p_n$ son propiedades de este tipo con los respectivos tipos $t_1, t_2,..., t_n$, entonces $t(p_1:e_1, p_2:e_2, ...,p_n:e_n)$ es una expresión de tipo t.

Esta expresión regresa un objeto nuevo del tipo t cuyas propiedades $p_1, p_2, ...,p_n$ son inicializadas con la expresión $e_1, e_2, ...,e_n$. El tipo de e_i debe ser del tipo de p_i o un subtipo.

Si t es un tipo nombre de una colección y e es una colección literal, entonces t(e) es un objeto colección. El tipo de e debe ser t.

Construyendo estructuras

Si $p_1, p_2,...,p_n$ son nombres de propiedades, si $e_1, e_2,...,e_n$ son expresiones con los tipos respectivos $t_1, t_2,..., t_n$, entonces $struct(p_1:e_1, p_2:e_2,...,p_n:e_n)$ es una expresión del tipo $struct(p_1:t_1, p_2:t_2,...,p_n:t_n)$. Éste regresa la estructura tomando valores $e_1, e_2,..., e_n$ sobre las propiedades $p_1, p_2, ..., p_n$.

Construyendo conjuntos

Si e_1, e_2,..., e_n son expresiones de tipos compatibles t_1, t_2,..., t_n, entonces set(e_1, e_2,..., e_n) es una expresión del tipo set(t), donde t = lub(t_1, t_2,..., t_n). Éste regresa el conjunto conteniendo los elementos e_1, e_2,..., e_n. Éste crea una instancia set.

Construyendo bags

Si e_1, e_2,..., e_n son expresiones de tipos compatibles t_1, t_2,..., t_n, entonces bag(e_1, e_2,..., e_n) es una expresión de tipo bag(t), donde t = lub(t_1, t_2,..., t_n). Ésta regresa el bag teniendo a los elementos e_1, e_2,..., e_n. Esto crea una instancia bag.

Construyendo arreglos

Si e_1, e_2,..., e_n son expresiones de tipos compatibles t_1, t_2,..., t_n, entonces array(e_1,e_2,..., e_n) es una expresión del tipo array(t)m donde t = lub(t_1, t_2, ..., t_n), Ésta regresa un arreglo conteniendo los elementos e_1, e_2,..., e_n. Ésta crea una instancia array.

2.9.7.4 Expresiones de Tipo Atómico

Expresiones unarias

Si e es una expresión y <op> es una operación unaria válida para el tipo de e, entonces <op>e es una expresión, ésta regresa el resultado de aplicar <op> a e con el mismo tipo de e.

Operadores unarios aritméticos: +. -, abs

Operadores unarios boléanos: not (el valor que regresa es de tipo boolean)

Expresiones binarias

Si e_1 y e_2 son expresiones y <op> es una operación binaria, entonces e_1<op>e_2 es una expresión, ésta regresa el resultado de aplicar <op> a e_1 y e_2.

Operadores binarios de aritmética entera: +, -, *, /,mod (módulo)
Operadores binarios de punto flotante: +, -, *, /
Operadores binarios relacionales: =, !=, <, <=, >, >=
Estos operadores están definidos en todos los tipos atómicos
Operadores binarios boléanos: and, or

Expresiones de cadenas

Si s_1 y s_2 son expresiones del tipo string, entonces s_1 || s_2 y s_1 + s_2 son expresiones equivalentes del tipo string, cuyo valor es la concatenación de las dos cadenas.

Si c es una expresión del tipo character y s es una expresión del tipo string, entonces c en s es una expresión del tipo boolean cuyo valor es verdadero si el caracter se encuentra dentro de la cadena, sino es falsa.

Si s es una expresión del tipo string e i es una expresión del tipo integer, entonces Si es una expresión de tipo character cuyo valor es el caracter $i+1^{th}$ de la cadena.

Si s es una expresión del tipo string, y low and up son expresiones del tipo integer, entonces s[low:up] es una expresión de tipo string cuyo valor es la subcadena de s desde el caracter $low+1^{th}$ hasta el caracter $up+1^{th}$.

Si s es una expresión del tipo string y pattern es una cadena literal que puede incluir los caracteres reservados; "?" o "_", que significan cualquier caracter, y "*" o "%", que significan cualquier subcadena, incluyendo la cadena vacía, entonces s como patrón es una expresión del tipo boolean cuyo valor es verdadero si s contiene el patrón, sino es falso.

2.9.7.5 Expresiones de Objetos

Comparación de objetos

Si e_1 y e_2 son expresiones que denotan objetos con identidad de tipos compatibles, entonces $e_1 = e_2$ y $e_1 != e_2$ son expresiones que regresan un valor boleano.

Comparación de literales

Si e_1 y e_2 son expresiones que denotan literales de tipos compatibles, entonces $e_1 = e_2$ y $e_1 != e_2$ son expresiones que regresan un valor boleano.

Extrayendo un atributo o cruzando una relación de un objeto

Si e es una expresión de un tipo (literal u objeto) teniendo un atributo o relación p del tipo t, entonces e.p y e->p son expresiones del tipo t. Éstas son sintaxis para extraer la propiedad p de un objeto e.

Si e está designado a un objeto borrado o que no existe, el acceso a un atributo a la relación debe retornar UNDEFINED.

Aplicando una operación a un objeto

Si e es una expresión de un tipo que tiene un método f sin parámetros y regresa un resultado de tipo t, entonces e->f y e.f son expresiones de tipo t. Éstas son las sintaxis para aplicar una operación sobre un objeto. El valor de la expresión es aquel regresado por la operación, si la operación no regresa nada el objeto es nulo.

Si e esté designado a un objeto borrado o que no existe, el acceso a un atributo a la relación debe retornar UNDEFINED.

Aplicando una operación con parámetros a un objeto

Si e es una expresión del tipo y se tiene un método f con parámetros del tipo t_1, t_2,...,t_n y regresando un resultado de tipo t, si e_1, e_2,..., e_n son expresiones del tipo t_1, t_2,...,t_n, entonces:

e->f(e_1, e_2,...,e_n) y e.f(e_1, e_2,...,e_n) son expresiones del tipo t que aplica la operación f con parámetros (e_1, e_2,...,e_n) hacia el objeto e. El valor de la expresión es aquel regresado por la operación, si la operación no regresa nada el objeto es nulo.

Si e está designado a un objeto borrado o que no existe, el acceso a un atributo a la relación debe retornar UNDEFINED.

2.9.7.6 Expresiones de Colección

Cuantificación universal

Si x es el nombre de variable, e_1 y e_2 son expresiones, e_1 denota una colección y e_2 es una expresión del tipo boleano, entonces para toda x en e_1:e_2 es una expresión de tipo boleano. Ésta regresa verdadero (True) si todos los elementos de la colección e_1 satisfacen a e_2 y falso (False) en otro caso.

Ejemplo:

Para toda x en Estudiantes: x.estudiante_id>0

Ésta regresa verdadero si todos los objetos en el conjunto Estudiantes tienen un valor positivo para su atributo estudiante_id. En otro caso regresa falso.

Cuantificación existencial

Si x es el nombre de variable, si e_1 y e_2 son expresiones, e_1 denota una colección y e_2 es una expresión del tipo boleano, entonces existe x en e_1:e_2 es una expresión de tipo boleano. Ésta regresa verdadero si hay al menos un elemento de la colección e_1 que satisface a e_2 y falso en otro caso.

Ejemplo:

Exists x in Doe.takes: x.taught_by.nombre = "Turing"

Ésta regresa verdadero si al menos un curso de los que toma Doe es enseñado por alguien llamado Turing.

Si e es una expresión de colección, entonces exists(e) y unique(e) son expresiones que regresan un valor boleano. La primera regresa verdadero si existe al menos un elemento en la colección, mientras la segunda regresa verdadero si existe solo un elemento en la colección.

Note que estos operadores aceptan la sintaxis SQL para consultas anidadas como:

Select ... from col *where* exists(*select ... from* col1 *where* predicado)

La consulta anidada regresa un bag para la cual el operador exists es aplicado.

Probando la membresía

Si e_1 y e_2 son expresiones, e_2 es una colección y e_1 es un objeto o una literal que tienen el mismo tipo o subtipo como los elementos de e_2, entonces e_1 en e_2 es una expresión del tipo boleano. Ésta regresa verdadero si el elemento e_1 corresponde a la colección e_2.

Ejemplo.

Doe in Estudiantes

ésta regresa verdadero.

Doe in AT

ésta regresa verdadero si Doe es un Asistente Técnico.

Operadores de agregación

Si e es una expresión que denota una colección, <op> es un operador en {min, max, count, sum, avg}, entonces <op>e es una expresión.

Ejemplo:

max(*select* salario *from* Profesores)

ésta regresa el salario máximo de los profesores.

Si e es del tipo collection(t), donde t es un entero o flotante, entonces <op>e donde <op> es un operador agregado diferente de "count" es una expresión de tipo t.

Si e es de tipo collection(t), entonces count(e) es una expresión de tipo entero.

2.9.7.7 Select From Where

La forma general de una sentencia select es la siguiente:

select [*distinct*] $f(x_1, x_2,..., x_n, x_{n+1}, x_{n+2},..., x_{n+p})$
from x_1 in e_1 $(x_{n+1}, x_{n+2},..., x_{n+p})$
x_2 in e_2 $(x_1, x_{n+1}, x_{n+2},..., x_{n+p})$
x3 in e3 $(x_1, x_2, x_{n+1}, x_{n+2},..., x_{n+p})$
...
x_n in e_n $(x_1, x_2,..., x_{n-1}, x_{n+1}, x_{n+2},..., x_{n+p})$

[*where* $p(x_1, x_2,..., x_n, x_{n+1}, x_{n+2},..., x_{n+p})$]

[*order by* $f_1(x_1, x_2,..., x_{n+p})$, $f_2(x_1, x_2,..., x_{n+p})$,..., $f_q(x_1, x_2,..., x_{n+p})$]

o

select [*distinct*] $f(x_1, x_2,..., x_n, x_{n+1}, x_{n+2},..., x_{n+p})$
from e_1 $(x_{n+1}, x_{n+2},..., x_{n+p})$ as x_1
e_2 $(x_1, x_{n+1}, x_{n+2},..., x_{n+p})$ as x_2
e_3 $(x_1, x_2, x_{n+1}, x_{n+2},..., x_{n+p})$ as x_3
...
e_n $(x_1, x_2,..., x_{n-1}, x_{n+1}, x_{n+2},..., x_{n+p})$ as x_n

[*where* $p(x_1, x_2,..., x_n, x_{n+1}, x_{n+2},..., x_{n+p})$]

[*order by* $f_1(x_1, x_2,..., x_{n+p})$, $f_2(x_1, x_2,..., x_{n+p})$,..., $f_q(x_1, x_2,..., x_{n+p})$]

$x_{n+1}, x_{n+2},..., x_{n+p}$ son variables libres que están enlazadas para evaluar la consulta. Los e_i son de tipo colección, p es del tipo boleano, y las f_i son de tipo atómico.

El resultado de la consulta será una colección de t, donde t es el tipo de resultado de f.

La semántica de la consulta es la siguiente:

Asumiendo que $x_{n+1}, x_{n+2},..., x_{n+p}$ están relacionadas con $X_{n+1}, X_{n+2},..., X_{n+p}$, la consulta es evaluada como sigue:

(1) El resultado de la cláusula from es un bag de elementos del tipo:

struct(x_1:X_1, x_2:X_2, ..., x_n: X_n) donde

X_1 está en el rango de la colección bagof($e_1(X_{n+1}, X_{n+2},..., X_{n+p})$)
X_2 está en el rango de la colección bagof($e_2(X_1, X_{n+1}, X_{n+2},..., X_{n+p})$)
X_3 está en el rango de la colección bagof($e_3(X_1, X_2, X_{n+1}, X_{n+2},..., X_{n+p})$)
...
X_n está en el rango de la colección bagof($e_n(X_1, X_2, ..., X_{n-1}, X_{n+1}, X_{n+2},..., X_{n+p})$)

Donde bagof(C) es definida de la siguiente forma, para una colección C:

Si C es un bag: C.
Si C es un list: el bag consiste de todos los elementos de C.
Si C es un set: el bag consiste de todos los elementos de C.

(2) Filtrar el resultado de la cláusula from reteniendo sólo aquellas tuplas $(X_1, X_2,..., X_n)$ que satisfagan el predicado $p(X_1, X_2,..., X_{n-1}, X_n, x_{n+1}, x_{n+2},..., x_{n+p})$.

(3) Si la palabra clave order by esta allí, clasifica esta colección lexicográficamente usando las funciones $f_1, f_2, ..., f_q$ y transforma ésta en una lista. El orden lexicográfico para un conjunto de funciones es ejecutado de la siguiente forma: primero ordena de acuerdo con la función f_1, entonces para todos los elementos que tienen el mismo valor f_1 ordena de acuerdo con f_2, etcétera.

(4) Aplicar a cada una de estas tuplas la función:

$f(X_1, X_2, \ldots, X_{n-1}, X_n, X_{n+1}, X_{n+2}, \ldots, X_{n+p})$

Si f es sólo "*", entonces conserva el resultado del paso 3 tal cual.

(5) Si la palabra clave "*distinct*" se encuentra, entonces elimina los duplicados eventuales y obtiene un conjunto o una lista sin duplicados.

En resumen, el tipo del resultado de un "*select from where*" es el siguiente:

Este es siempre una colección, el tipo de la colección no depende sobre los tipos de las colecciones especificadas en la cláusula from.

El tipo de colección depende sólo de la forma de la consulta: si se utiliza "*order by*", obtenemos un set; y si no son usadas las cláusulas "*order by*" y "*distinct*", obtenemos un bag.

Ejemplo:

```
select couple(estudiante: x.nombre, profesor:z.nombre)
  from Estudiantes as x,
       x.takes as y,
       y.taught_by as z
 where z.rank = "full professor"
```

Esta consulta regresa un bag de objetos del tipo couple, obteniendo nombres de estudiantes y los nombres de profesores completos con los que toman clases.

Ejemplo:

```
select *
  from Estudiantes as x,
       x.takes as y,
       y.taught_by as z
 where z.rank = "full professor"
```

Esta consulta regresa un bag de estructuras, obteniendo para cada "objeto" Estudiante el objeto sección seguido por el estudiante y el "objeto" profesor completo enseñando en esta sección:

```
bag<struct(x:Estudiante, y:Section, z:Professor)>
```

Las variaciones sintácticas son aceptadas para la declaración de variables en la parte from, exactamente como con SQL. La palabra reservada as puede ser omitida. Sin embargo, la variable misma puede ser omitida también, en este caso, el nombre de la colección misma sirve como un nombre de variable sobre ésta.

Ejemplo:

select couple(estudiante:Estudiantes.nombre, professor: z.nombre)
from Estudiantes,
Estudiantes.takes y,
y.taught_by = "full professor"

En un *select-from-where*, la cláusula *where* puede ser omitida, con el significado de un predicado verdadero.

2.9.7.8 Operador Group-By

Si select_query es una consulta *select-from-where*, partition_attributes es una expresión de estructura, y predicate una expresión boleana, entonces:

select_query *group by* partition_atributes
es una expresión, y

select_query *group by* partition_atributes having predicate
es una expresión.

El producto cartesiano visitado por el operador select es dividido en particiones. Para cada elemento del producto cartesiano la partición de atributos son evaluados. Todos los elementos que correspondan a los mismos valores de acuerdo con la partición de atributos dada que correspondan a la misma partición. Así el conjunto particionado, después de la operación de agrupación, es un conjunto de estructuras: cada estructura tiene las propiedades evaluadas para esta partición (el valor partition_attributes), completada por una propiedad que es convencionalmente llamada partition y que es un bag de todos los elementos del producto cartesiano que correspondieron a esta particular partición evaluada.

Si la partición de atributos son att_1:e_1, att_2:e_2,..., att_n:e_n, entonces el resultado de la agrupación es del tipo:

set<struct(att_1:type_of(e_1), att_2:type_of(e_2), ..., att_n:type_of(e_n),
partition:bag<type_of(grouped elements)>)>

El tipo de elementos agrupados es definido de la siguiente manera:

Si la cláusula from declara las variables v_1 sobre la colección col_1, v_2 sobre col_2, ..., v_n sobre col_n, los elementos agrupados es una estructura con un atributo, vk, para cada colección que tiene el tipo de los elementos de la partición de la colección correspondiente:

bag<struct(v_1:type_of(col_1, elements),..., v_n:type_of($vcol_n$, elements))>

Si una colección col_k no tiene variable declarada, el atributo correspondiente tiene un nombre de sistema interno.

Este set particionado puede ser entonces filtrado por el predicado de la cláusula having. Finalmente el resultado es calculado evaluando la cláusula select para este set particionado y filtrado.

La cláusula having puede aplicar funciones agregadas sobre la partición, además, la cláusula select puede referirse a la partición para calcular el resultado final. Ambas cláusulas pueden referirse también a la partición de atributos.

Ejemplo:

```
select *
  from Empleados e
 group by low:    salario < 1000,
           medium: salario >= 1000 and salario < 10000,
           high:   salario >= 10000
```

Ésta da un set de tres elementos, cada uno de ellos tiene una propiedad llamada partición que contiene el bag de empleados que entran en esta categoría. Así el tipo de resultado es:

```
set<struct(low:boolean, medium:boolean, high:boolean,
partition:bag<struct(e:Empleado)>)>
```

La segunda forma realza a la primera con una cláusula having que habilita filtrar el resultado usando funciones agregativas que operan sobre cada partición.

Ejemplo:

```
select department,
   avg_salario:avg(select x.e.salario from partition x)
  from Empleados e
group by department
having avg(select x.e.salario from partition x) > 30000
```

Ésta consulta obtiene un set de parejas: departamento y promedio de los salarios de los empleados trabajando en ese departamento, cuando este promedio sea mayor de 30000. Así el tipo del resultado es:

```
bag<struct(department: integer, avg_salario: float)>
```

2.9.7.9 Operador Order-by

Si select_query es una consulta *select-from-where* o *select-from-where-group_by*, y si e_1, e_2,..., e_n son expresiones, entonces select_query *order by* e_1, e_2,..., e_n es una expresión. Ésta regresa una lista de los elementos seleccionados clasificados por la función e1, y dentro de cada subset producido por la misma e_1, clasificado por e_2,..., y el subsub...set final, clasificado por e_n.

Ejemplo:

select p *from* Personas p *order by* p.edad, p.nombre

Ésta consulta clasifica al conjunto de personas por su edad, y como segundo criterio por su nombre, y el resultado de la clasificación de los objetos es puesto dentro de una lista.

Cada expresión como criterio de clasificación puede estar seguida por la palabra reservada ASC o DESC, especificando respectivamente el orden ascendente o descendente. El orden por omisión es el de la declaración previa. Para la primera expresión, el orden es ascendente.

Ejemplo:

select * *from* Personas *order by* edad desc, nombre asc, department

2.9.7.10 Expresiones de Colección Indexada

Obteniendo el i-ésimo elemento de una Colección Indexada

Si e_1 es una expresión del tipo list(t) o array(t) y e_2 es un expresión del tipo Integer, entonces e_1 [e_2] es una expresión del tipo t. Ésta extrae el elemento e_2+1 de la colección indexada e_1. Note que el primer elemento tiene el rango 0.

Ejemplo:

list(a,b,c,d)[1]

Ésta regresa b.

Ejemplo:

element(*select* x
from Courses x
where x.nombre = "Math" and x.number = "101")requires[2]

Ésta regresa el tercer prerrequisito de Math 101.

Extrayendo una subcolección de una Colección Indexada

Si e_1 es una expresión del tipo list(t) (resp.array(t)), y e_2 y e_3 son expresiones del tipo Integer, entonces e_1[e_2:e_3] es una expresión del tipo list(t) (resp.array(t)). Ésta extrae la subcolección e_1 empezando en la posición e_2 y finalizando en la posición e_3.

Ejemplo

list(a, b, c, d)[1:3]

Ésta operación regresa list(b, c, d).

Ejemplo:

element(*select* x
from Courses x
where x.nombre = "Math" and x.number ="101").requiere[0:2]

Ésta consulta regresa una lista que consiste en los primeros tres requisitos de Math 101. Obteniendo el Primer y Último elemento de una Colección Indexada.

Si e es una expresión del tipo list(t) o array(t), <op> es un operador de {first, last}, entonces <op>(e) es una expresión del tipo t. Ésta extrae el primer y último elemento de una colección.

Ejemplo

first(element(*select* x
from Courses x
where w.nombre = "Math" and x.number = "101").requiere)

Ésta consulta regresa el primer prerrequisito de Math 101.

Concatenando dos Colecciones Indexadas

Si e_1 y e_2 son expresiones del tipo list(t_1) y list(t_2) (resp. array(t_1) and array(t_2)) donde t_1 y t_2 son compatibles, entonces e_1+e_2 es una expresión del tipo list(lub(t_1, t_2)) (resp. array(lub(t_1, t_2))). Ésta calcula la concatenación de e_1 y e_2.

list(1, 2) + list(2, 3)

Esta consulta genera la lista (1, 2, 2, 3).

Accesando a un elemento de un diccionario desde su clave

Si e_1 es una expresión del tipo dictionary(k, v) y e_2 es una expresión de tipo k, entonces $e_1[e_2]$ es una expresión del tipo v. Ésta extrae el valor asociado con la llave e_2 en el diccionario e_1.

Ejemplo:

theDict["foobar"]

Ésta consulta regresa el valor que está asociado con la llave "foobar" en el diccionario theDict.

2.9.7.11 Expresiones de Conjuntos Binarios

Unión, Intersección y Diferencia

Si e_1 es una expresión del tipo set(t_1) o bag(t_1) y e_2 es una expresión del tipo set(t_2) o bag(t_2) donde t_1 y t_2 son tipos compatibles, si <op> es un operador de {union, except,

intersect}, entonces e_1<op>e_2 es una expresión del tipo set(lub(t_1,t_2)) si ambas expresiones son del tipo set, o bag(lub(t_1,t_2)) si cualquiera de ellos es del tipo bag.

Cuando los tipos de colección del operando son diferentes (bag y set), el set es convertido primero a un bag y el resultado es un bag.

Ejemplo:

Estudiante except AE

Ésta consulta regresa al conjunto de estudiantes que no son asistentes de enseñanza.

bag(2, 2, 3, 3, 3) *union* bag(2, 3, 3, 3)

Ésta expresión de bag regresa: bag(2, 2, 3, 3, 3, 2, 3, 3, 3).

bag(2, 2, 3, 3, 3) *intersect* bag(2, 3, 3, 3)

La intersección de dos bags produce un bag que contiene lo mínimo para cada uno de los valores múltiples. Así el resultado es: bag(2, 3, 3, 3).

bag(2, 2, 3, 3, 3) *except* bag(2, 3, 3, 3)

Esta expresión de bag regresa: bag(2).

Inclusión

Si e_1 y e_2 son expresiones que denotan sets o bags de tipos compatibles y si <op> es un operador de (<, <=, >, >=), entonces e_1<op>e_2 es una expresión de tipo boleano.

Cuando los operandos son de diferente género de colecciones (bag y set), el set es primero convertido en un bag.

e_1<e_2 es verdadero si e_1 está incluido en e_2 pero no igual a e_2,

e_1<=e_2 es verdadero si e_1 está incluido en e_2,

Ejemplo:
set(1, 2, 3) < set(3, 4, 2, 1) es verdadera (true).

2.9.7.12 Expresiones de Conversión

Extracción del elemento de un Singleton

Si e es una expresión del tipo collection(t), element(e) es una expresión del tipo t. Ésta toma el singleton e y regresa su elemento. Si e no es un singleton, ésta levanta una excepción.

Ejemplo:

element(*select* x *from* Professors x *where* x.nombre = "Turing").

Ésta consulta regresa el profesor cuyo nombre es Turing (si solo hay uno).

Cambiando un List a un Set

Si e es una expresión del tipo list(t), listtoset(e) es una expresión del tipo set(t). Ésta convierte el list en un set, formando el set que contendrá todos los elementos del list.

Ejemplo:

listtoset(list(1, 2, 3, 2))

Ésta regresa el set que contendrá 1, 2 y 3.

Removiendo duplicados

Si e es una expresión del tipo col(t), donde col es set o bag, entonces distinct(e) es una expresión del tipo set(t) cuyo valor es la misma colección después de remover todos los elementos duplicados. Si e es una expresión del tipo col(t), donde col es una lista o un arreglo, entonces distinct(e) es una expresión del tipo col(t) obtenido para conservar la primera ocurrencia para cada elemento de la lista.

Ejemplo:

distinct(list(1, 4, 2, 3, 2, 4, 1))

Ésta operación regresa: list (1, 4, 2, 3).

Aplanando una colección de colecciones

Si e es una expresión de colección-valuada, flatten(e) es una expresión. Ésta convierte una colección de colecciones de t en una colección de t. Así, el aplanamiento opera sólo en el primer nivel.

Asumiendo el tipo de e para ser col_1<col_2<t>>, el resultado de flatten(e) es:

- Si col_2 es un set (resp. un bag), la unión de todo col_2<t> es hecha y el resultado es set<t> (resp. Bag<t>).

- Si col_2 es un list o un array y col_1 es un list o un array, la concatenación de todo col_2<t> es hecha siguiendo el orden en col_1 y el resultado es col_2<t>, el cual es una lista o un arreglo. Por supuesto los duplicados, si los hay, son mantenidos por esta operación.

- Si col_2 es un list o un arrray y col_1 s un set(resp. un bag), las listas o arreglos son convertidos en sets (resp. bags), la unión de todos estos sets (resp. bags) es realizada, y el resultado es un set<t> (resp. bag<t>).

Ejemplo:

flatten(list(set(1, 2, 3), set(3, 4, 5, 6), set(7)))

Ésta operación regresa un conjunto (set) conteniendo 1, 2, 3, 4, 5, 6, 7.

flatten(list(list(1,2), list(1, 2, 3)))

Ésta operación regresa list(1, 2, 1, 2, 3).

flatten(set(list(1, 2), list(1, 2, 3)))

Ésta operación regresa el conjunto (set) conteniendo 1, 2, 3.

Escribiendo una expresión

Si e es una expresión del tipo t y t' es un nombre de tipo, y t y t' son comparables (t >= t' o t <= t'), entonces (t')e es una expresión de tipo t'. Ésta expresión tiene dos impactos:

(1) En tiempo de compilación, ésta es una sentencia para el verificador de escritura interprete/compilador para notificar que **e** debe ser comprendida como el tipo **t'**.
(2) En tiempo de ejecución ésta expresión mantiene que **e** es realmente del tipo **t'** (o un subtipo de éste) y debe regresar el resultado de **e** en este caso, o una excepción en los demás casos.

Este mecanismo permite al usuario ejecutar consultas que deberán de algún modo ser rechazadas como escritas incorrectamente. Por ejemplo:

select s.salario
from Estudiante s
where s in (*select* sec.asistente *from* Secciones sec)

Porque s está restringida en la cláusula where para asistentes de enseñanza que enseñan en una sección, ésta consulta debe regresar realmente los salarios de estas personas. Sin embargo, el verificador de escritura no tiene la intención de revisar que la s en Estudiante tiene siempre un campo de salario, y la consulta debe ser rechazada en tiempo de compilación.

Si se escribe

select ((Empleado)s).salario
from Estudiante s
where s in (*select* sec.asistente *from* Secciones sec)

entonces el verificador de escritura conoce que s es del tipo Empleado y la consulta es aceptada como escritura correcta. En tiempo de ejecución, cada ocurrencia de s en la cláusula select debe ser revisada por su tipo.

2.9.7.13 Llamado de una Función

Si f es una función del tipo (t_1, t_2,...,t_n -> t), si e_1, e_2,..., e_n son expresiones del tipo t_1, t_2, ...,t_n, entonces f(e_1, e_2,..., e_n) es una expresión del tipo t cuyo valor es el que regresa la función, o el objeto Null, cuando la función no regresa ningún valor. La primer forma llama una función sin un parámetro, mientras que la segunda llama a una función con los parámetros e_1, e_2,..., e_n.

OQL no define en que lenguaje es escrito el cuerpo de la función. Esto permite extender la funcionalidad de OQL sin cambiar el lenguaje.

2.9.7.14 Reglas de Ámbito

La parte from de una consulta select-from-where introduce variables explícitas o implícitas al rango sobre las colecciones filtradas. Un ejemplo de una variable explícita es:

select ... *from* Personas p ...

mientras una declaración implícita debe ser:

select ... *from* Personas ...

El ámbito de estas variables se extiende sobre todas las partes de la expresión *select-from-where*, incluyendo expresiones anidadas.

La parte *group by* de una consulta *select-from-where-group_by* introduce el nombre partition junto con los nombres de atributos explícitos posibles que caracterizan a la partición. Estos nombres son visibles en las partes *select* y *having* correspondientes, incluyendo subexpresiones anidadas dentro de estas partes.

En lugar de un ámbito, se utilizan nombres de variables para construir expresiones de camino y alcanzar las propiedades (atributos y operaciones) cuando estas variables denotan objetos complejos. Por ejemplo, en el ámbito de la primer cláusula *from* de arriba, se accesa a la edad de una persona mediante p.edad.

Cuando la variable es implícita, como en la segunda cláusula *from*, se utiliza directamente el nombre de la colección de la siguiente forma Personas.edad.

Sin embargo, cuando no existe ambigüedad, se puede utilizar el nombre de la propiedad directamente como acceso directo, sin utilizar el nombre de variable para abrir el ámbito (esto es hecho implícitamente), escribiendo simplemente: edad. No hay ambigüedad cuando un nombre de propiedad es definido para uno y sólo un objeto denotado por una variable visible.

2.10 Generalidades de SMBDO

2.10.1 Sistema O2

Definición de datos

En O2, la definición del esquema emplea las ligaduras del lenguaje C++ o Java para ODL, como han sido definidas por ODMG.

Manipulación de datos

Las aplicaciones de O2 pueden crearse mediante la ligadura de O2 para C++ o Java, que proporciona a la base de datos una ligadura de lenguaje acorde con ODMG. La ligadura amplía el lenguaje de programación ofreciendo los siguientes elementos: punteros persistentes, colecciones genéricas, objetos persistentes con nombre, relaciones, consultas y soporte del sistema de base de datos para sesiones, bases de datos y transacciones.

El modelo ODMG indica para C++, que la persistencia es declarada cuando se crea el objeto. Ésta propiedad es inmutable, ya que un objeto transitorio no puede convertirse en objeto persistente, y la integridad referencial no está garantizada; si los subobjetos de éste no son persistentes, la aplicación fallará al seguir las referencias. Asimismo, si se elimina un objeto, las referencias que se le hagan fallarán cuando se acceda a través de ellas. O2 soporta la persistencia mediante alcanzabilidad, lo que simplifica la programación de la aplicación y garantiza la integridad referencial. Cuando un objeto o valor se hace persistente, también lo hacen sus subobjetos, esto hace que el programador no tenga que hacerlo de forma explícita. En cualquier momento un objeto puede cambiar de ser persistente a transitorio y viceversa. Los objetos se hacen persistentes cuando son instanciados y siguen manteniendo su identidad; a los objetos que ya no se hace referencia se desechan automáticamente (garbage collection).

O2 soporta el lenguaje de consultas de objetos OQL, cuando se transforman consultas de este lenguaje a C++ existen dos alternativas; el primer método consiste en usar una función miembro para realizar la consulta sobre una colección, en este caso se especifica un predicado de selección, usando la sintaxis de la cláusula *where* de OQL, de modo que se filtre la colección seleccionando aquellas tuplas que satisfagan la condición *where*. El segundo método permite utilizar la funcionalidad de OQL desde un programa C++ mediante el empleo de la función *d_oql_execute*. Esta función ejecuta la consulta de tipo *d_OQL_Query* que aparece en su primer argumento y obtiene como resultado la colección C++ especificada en su segundo argumento.

El núcleo del sistema O2, llamado O2Engine, se encarga de una buena parte de la funcionalidad del SMBD, como proporcionar recursos de almacenamiento, obtención y actualización de objetos almacenados persistentemente, que pueden ser compartidos por múltiples programas. O2Engine implementa los mecanismos de concurrencia, recuperación y seguridad comunes en los sistemas de bases de datos, además, implementa un modelo de gestión de transacciones, gestión de versiones, gestión de notificaciones, mecanismos de evolución de esquemas y un mecanismo de replicación. La

implementación de O2Engine en el nivel de sistema se basa en una arquitectura cliente/servidor.

A nivel funcional, O2Engine tiene tres componentes principales:

1. Componente de almacenamiento: la implementación de esta capa está dividida entre el cliente y el servidor. El proceso servidor se encarga de la gestión del disco, del almacenamiento y recuperación de páginas, del control de concurrencia y de la recuperación. El proceso cliente localiza las páginas y bloqueos proporcionados por el servidor y los pone a disposición de los módulos funcionales de mayor nivel del cliente O2.
2. Gestor de objetos: se encarga de estructurar objetos y valores, formar grupos de objetos relacionados en páginas de disco, indexar objetos, mantener la identidad de los objetos, efectuar operaciones con objetos, etcétera.
3. Gestor de esquemas: lleva el control de las definiciones de clases, tipos y métodos, proporciona los mecanismos de herencia, verifica la consistencia de las declaraciones de clases y hace posible la evolución de esquemas, lo que incluye la creación, modificación y la eliminación incrementales de declaraciones de clases.

2.10.2 Sistema ObjectStore

Definición de datos

ObjectStore está integrado por diferentes paquetes que se pueden adquirir por separado. Un paquete proporciona almacenamiento persistente para Java y otro para C++.

El paquete para C++ está muy integrado con el lenguaje C++ y proporciona capacidades de almacenamiento persistente de C++, ObjectStore utiliza las declaraciones de clase de C++ como lenguaje de definición de datos, con una sintaxis C++ extendida que incluye constructores adicionales que son útiles específicamente en aplicaciones de bases de datos. Los objetos persistentes pueden ser compartidos por varios programas.

El compilador de C++ extendido de ObjectStore maneja declaraciones de relaciones inversas y funciones adicionales. En C++, un asterisco (*) especifica una referencia y el tipo de campo aparece antes del nombre del atributo. Los tipos básicos de C++ son char, int, float.

ObjectStore tiene un constructor de conjuntos a C++ valiéndose de la palabra clave *os_Set*, también cuenta con constructores de bolsas y de listas llamados *os_Bag* y *os_List*, respectivamente.

ObjectStore tiene un mecanismo para la especificación de relaciones que permite especificar los atributos inversos que representan una relación binaria.

Manipulación de datos

Se pueden aplicar funciones adicionales a los tipos *collection* de ObjectStore, entre ellas se incluyen las funciones insert(e), remove(e) y create, que sirven para insertar y borrar un elemento e en una colección, y crear una colección nueva, respectivamente.

En C++, la referencia funcional a los elementos de un objeto *o* emplea la notación de flechas cuando se proporciona un apuntador a *o*, y usa la notación de punto cuando se provee una variable cuyo valor es el propio objeto *o*. Con cualquiera de ellas podemos referirnos tanto a los atributos como a las funciones del objeto.

Si el programador desea crear objetos y colecciones persistentes en ObjectStore, deberá asignarles un nombre, al que también se le denomina *variable persistente*. Ésta puede considerarse como una referencia abreviada al objeto.

ObjectStore cuenta también con un recurso de consultas, que sirve para seleccionar un conjunto de objetos de una colección especificando una condición de selección. El resultado de una consulta es una colección de apuntadores a los objetos que satisfacen la consulta. Las consultas pueden incorporarse en un programa C++, y considerarse como un medio de acceso asociativo de alto nivel a los objetos seleccionados que hace innecesario crear una construcción cíclica explícita.

2.11 Resumen

El estándar ODMG se desarrolló con el propósito de que los SMBDO estandaricen sus características, con la finalidad de que las aplicaciones sean portables entre los sistemas, y el manejador de bases de datos extienda al lenguaje de programación con las propiedades de persistencia, concurrencia, recurrencia, etcétera.

El estándar se basa en un modelo de objetos, en donde se definen las características de los objetos y las relaciones que existirán entre ellos, los tipos, la herencia, extensiones, llaves, objetos, colecciones y literales. En este modelo también se indican el estado, comportamiento y manejo de excepciones; el estado se da por el conjunto de propiedades de cada tipo definido en el modelo de objetos, las propiedades pueden atributos o relaciones, los primeros se definen dentro de un tipo y las segundas se dan entre dos tipos; el comportamiento se da por el conjunto de operaciones de cada tipo, a cada operación se le definen nombre de ésta, nombre de los argumentos, tipos de los argumentos, los tipos de valores que regresa la operación y los nombres de las excepciones que la operación pueda levantar; el manejo de excepciones permite controlar los errores que se puedan producir durante la ejecución de alguna operación, esto permitirá que el sistema termine la función de una manera optativa para que no se afecte la operatividad de este.

Se cuenta además con un método de control de concurrencia a través de candados, esto da una forma de acceso exclusiva a los objetos, existen tres tipos de candados: Read (lectura), Write (escritura), Upgrade (actualización); los dos primeros son candados implícitos que se aplican directamente cuando se hace una consulta o se modifica algún

objeto respectivamente; el tercer candado es explicito, si se desea utilizar se indica sobre que objeto se utilizará.

Se define un modelo de transacciones, este permite que los programas que usan objetos persistentes se organicen dentro de transacciones, éstas permiten al SMBDO garantizar atomicidad, consistencia, aislamiento y durabilidad, así cualquier operación que se haga a la base de datos debe hacerse dentro de una transacción.

Una base de datos es una instancia del tipo Database, la interfaz de éste tiene las operaciones open, close, lookup, bind, unbind y schema además de soportar operaciones administrativas como create, delete, move, copy, reorganize, verify, backup and restore.

El estándar ODMG define un lenguaje para poder consultar los objetos de la base de datos llamado OQL, éste nos da las sentencias que accedan los conjuntos de objetos así como el manejo de estructuras listas y arreglos.

Capítulo 3. Lenguaje de Modelado UML

Una de las herramientas que más se está utilizando en el paradigma de la orientación a objetos es UML, ya que en el desarrollo de un programa, de un sistema, de una base de datos, etc., se necesita primero hacer un modelo de lo que se desea hacer. El UML en este caso lo admitimos porque es un lenguaje para modelar dentro del paradigma de la orientación a objetos, a continuación se describe a grandes rasgos.

3.1 ¿Qué es UML?

El lenguaje unificado de modelado o UML (Unified Modeling Language) es el producto que conjunta toda la oleada de métodos de análisis y diseño orientados a objetos que surgió a finales de la década de los años ochenta y principios de los noventa. El UML unifica, sobre todo, los métodos que desarrollaron Booch, Rumbaugh (OMT) y Jacobson, pero su alcance llega a ser mucho más amplio.

Decimos pues, que el UML es un lenguaje de modelado, y no un método. La mayor parte de los métodos consisten, al menos en principio, en un lenguaje y un proceso para modelar. El lenguaje de modelado es la notación (principalmente gráfica) de que se valen los métodos para expresar los diseños. "El UML se define como un lenguaje que permite especificar, visualizar y construir los artefactos de los sistemas de software... [BJR97]" [LARM, 1999]. El proceso es la orientación que nos dan sobre los pasos a seguir para hacer el diseño de sistemas que utilizan conceptos orientados a objetos. A continuación se describirá el alcance y la estructura de UML, refiriéndose a sus elementos. Las partes que conforman al UML son:

- Vistas
- Diagramas
- Elementos del Modelo
- Mecanismos Generales

3.2 Vistas

Las vistas nos ayudan a mostrar diferentes aspectos del sistema que se está modelando, una vista no es un gráfico, sino una abstracción que consiste de un número de diagramas. El modelado de un sistema suele ser una tarea difícil y elaborada, y querer representar en una sola imagen toda la funcionalidad y requerimientos del sistema sería imposible, por lo tanto se recurre a las vistas, donde cada vista representa una proyección de la descripción completa del sistema, mostrando un aspecto particular de éste. Cada vista es descrita en un número de diagramas que contienen información que enfatiza un aspecto particular del sistema. Un diagrama contiene símbolos gráficos que representan a los elementos del modelo del sistema. Las vistas de UML son:

- Vista de casos de uso
- Vista lógica
- Vista de componentes
- Vista de concurrencia
- Vista de despliegue (Deployment)

3.2.1 Vista de casos de uso

Esta vista describe la funcionalidad que debe tener el sistema, según la percepción de actores externos. Un actor interactúa con el sistema, éste puede ser un usuario u otro sistema. La vista de casos de uso es para clientes, diseñadores, desarrolladores y probadores. Se describe a través de diagramas de casos de uso y ocasionalmente diagramas de actividad. El uso que se desea del sistema es descrito mediante un número de casos de uso en la vista de casos de uso, donde cada uno es una descripción genérica de un uso del sistema. (Una función requerida.)

La vista de casos de uso es central, ya que su contenido conduce al desarrollo de otras vistas. EL objetivo final del sistema es proporcionar la funcionalidad descrita en éstas, por lo tanto, esta vista afecta a todas las demás. Esta vista también es usada para validar y finalmente verificar el sistema mediante la prueba de los casos de uso contra los clientes (preguntando "¿Es esto lo que querías?") y contra el sistema terminado (preguntando "¿Trabaja el sistema como se especificó?").

3.2.2 Vista lógica

La vista lógica describe cómo se proporciona la funcionalidad al sistema. Esta vista está orientada principalmente para los diseñadores y desarrolladores. En contraste con la vista de casos de uso, la vista lógica se involucra dentro del sistema. Ésta describe tanto la estructura estática (clases, objetos y relaciones) y las colaboraciones dinámicas que ocurren cuando los objetos se envían mensajes entre sí para desempeñar alguna función determinada. Se definen también propiedades tales como persistencia y concurrencia, así como interfaces y estructura interna de las clases.

La estructura estática es descrita en diagramas de clases y de objetos. El modelado dinámico se describe en diagramas de estado, secuencia, colaboración y actividad.

3.2.3 Vista de componentes

La vista de componentes es una descripción de los módulos de implementación y sus dependencias. Sirve principalmente para desarrolladores y consiste en diagramas de componentes. Los componentes son de diferentes tipos de módulos de código, se muestran con su estructura y dependencias. Información adicional sobre los componentes, tal como la asignación de recursos (responsabilidad para un componente), u otra información administrativa, tal como un reporte de progreso del trabajo de desarrollo, también puede ser agregada.

3.2.4 Vista de concurrencia

La vista de concurrencia trata con la división del sistema en procesos y procesadores. Este aspecto, el cual es una propiedad no funcional del sistema, permite un uso eficiente de recursos, ejecución paralela y manejo de eventos asíncronos del ambiente. Además, dividiendo el sistema en hilos de control de ejecución concurrente, la vista debe tratar también con la comunicación y sincronización de dichos hilos.

La vista de concurrencia es para desarrolladores e integradores del sistema, y consiste en diagramas dinámicos (estado, secuencia, colaboración y actividad) y diagramas de implantación (diagramas de componentes y de despliegue "deployment").

3.2.5 Vista de despliegue

Finalmente, la vista de despliegue muestra el despliegue físico del sistema, abarcando computadoras y dispositivos (nodos) y cómo se conectan entre sí. La vista de despliegue es para desarrolladores, integradores y probadores, y es representada por los diagramas de despliegue. Esta vista incluye también un mapeo que muestra como son desplegados los componentes en la arquitectura física; por ejemplo, qué programas u objetos se ejecutan en cuál computadora.

3.3 Diagramas

Los diagramas son las gráficas que muestran símbolos y elementos de modelado organizados para ilustrar una parte o aspecto particular del sistema. Un modelo de sistema típicamente tiene varios diagramas de cada tipo. Un diagrama es parte de una vista específica; y cuando se dibuja, es ubicado usualmente en una vista. Algunos tipos de diagramas pueden ser parte de varias vistas, dependiendo de los contenidos del diagrama.

Los diagramas que utiliza UML son:

- Diagramas de casos de uso
- Diagrama de clases
- Diagrama de objetos
- Diagrama de estados
- Diagrama de secuencia
- Diagrama de colaboración
- Diagrama de actividad
- Diagrama de componentes
- Diagrama de despliegue

A continuación se describen en forma muy general estos diagramas.

3.3.1 Diagramas de casos de uso

Un diagrama de caso de uso muestra un número de actores externos y su conexión con los casos de uso que el sistema brinda. Un caso de uso es una descripción de una funcionalidad (un uso específico del sistema) que el sistema proporciona. Por ejemplo, en la figura de abajo se muestran dos casos de uso que describen las funciones que realizan en una empresa de Seguros, un *Cliente* y un *Vendedor de Seguros* (Actores), se muestra en este caso que los dos actores participan o utilizan el caso de uso *Firma de Una Póliza de Seguros*, y en el caso de uso *Estadísticas* sólo participa el Vendedor de Seguros. Con este diagrama se definen y se muestran de forma gráfica dos de las operaciones que se llevan a cabo en una Compañía Aseguradora.

Figura 6. *Diagrama de casos de uso para un negocio de seguros.*

La descripción de un caso de uso se lleva a cabo normalmente en texto plano como una propiedad de documentación del símbolo de caso de uso, pero también puede ser descrito usando un diagrama de actividad. Los casos de uso son descritos solamente como son vistos por el actor externo (el comportamiento del sistema como el usuario lo percibe), y no describe cómo se realiza la funcionalidad dentro del sistema. Los casos de uso definen los requerimientos de funcionalidad del sistema.

3.3.2 Diagrama de clases

Un diagrama de clases muestra la estructura estática de clases en el sistema. Las clases representan las "cosas" que son manejadas por el sistema. Las clases pueden relacionarse entre sí, de diversas maneras: asociadas (conectadas entre sí), dependientes (una clase depende/usa a otra), especializadas (una clase es una especialización de otra), o empaquetadas (agrupadas como una unidad). Todas estas relaciones se muestran en un diagrama de clases junto con la estructura interna de las clases en términos de atributos y operaciones. El diagrama se considera estático, en tanto que la estructura descrita sea válida en todo punto del ciclo de vida del sistema.

Un sistema, típicamente tiene un número de diagramas de clases –no todas las clases se insertan en un solo diagrama– y una clase puede participar en varios diagramas de clase.

En el diagrama de clases siguiente se muestran las clases que se necesitarían para el desarrollo de un sistema para el Comercio Financiero, además de cómo se asocian unas con otras, como se muestra en las asociaciones que hay entre las clases *Cliente* y *Portafolio*; *Comerciante* y *Portafolio*; y *Portafolio* e *Instrumento*, así como las generalizaciones que se hacen de la clase *Instrumento*; *Enlace*, *Acción* y *Opción de Acción*.

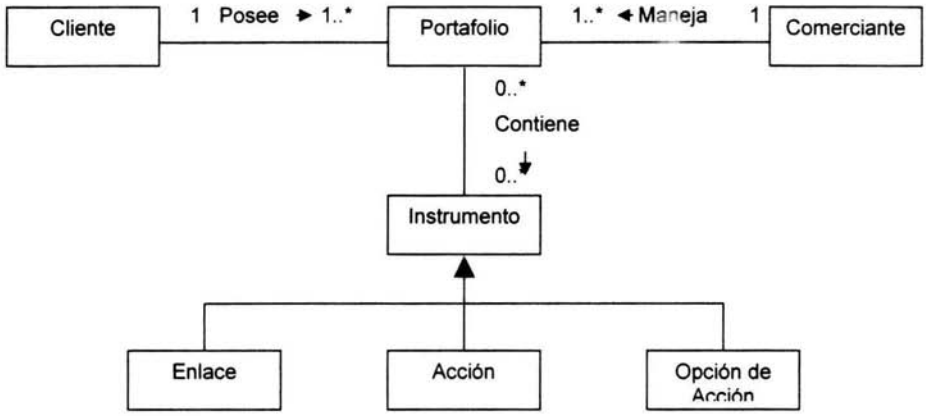


Figura 7. *Diagrama de clases para comercio financiero.*

3.3.3 Diagrama de objetos

Un diagrama de objetos es una variante de un diagrama de clases y usa una notación casi idéntica. La diferencia entre los dos, es que un diagrama de objetos muestra una o varias instancias (objetos) de las clases, en lugar de las clases en sí. Un diagrama de objetos es un ejemplo del diagrama de clases que muestra una posible ejecución del sistema, se puede observar cómo será el sistema en algún momento del tiempo. Se usa la misma notación del diagrama de clases, con dos excepciones: Los objetos son escritos con sus nombres subrayados, y se muestran todas las instancias en una relación.

Los diagramas de objetos no son tan importantes como los diagramas de clases, pero pueden ser usados para ejemplificar diagramas de clases complejos, mostrando las instancias y cómo podrían ser las relaciones. Los diagramas de objetos son también usados como parte de los diagramas de colaboración, donde se muestra la colaboración dinámica entre un conjunto de objetos.

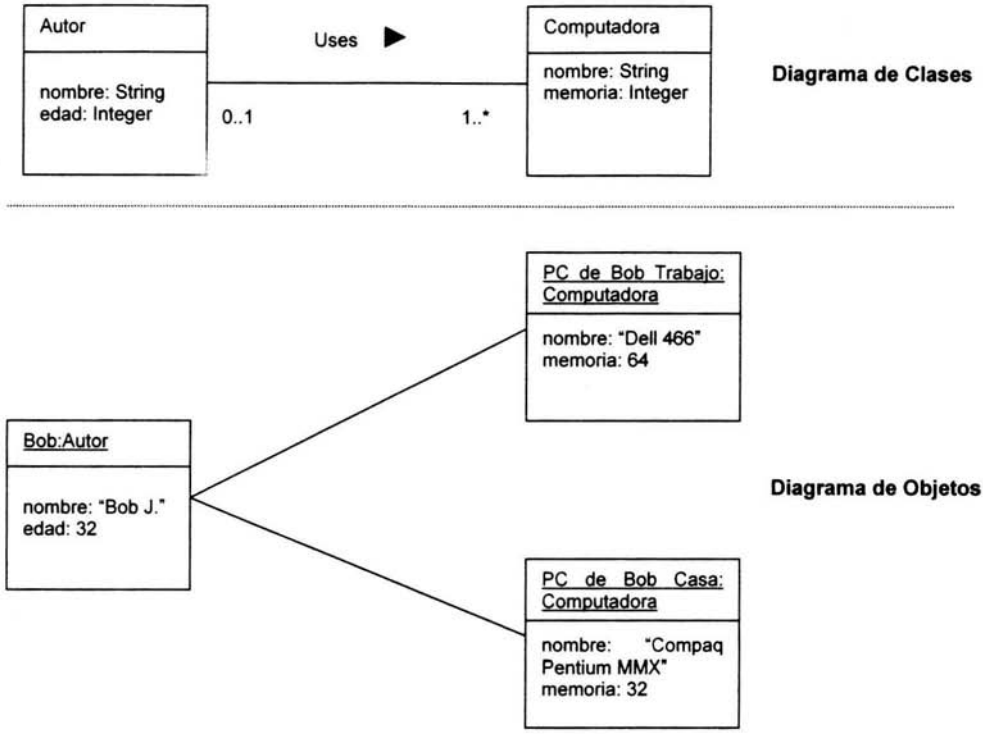


Figura 8. *Diagrama de objetos mostrando las instancias de las clases.*

... en el diagrama anterior se muestra un diagrama de clases que contiene las clases *Autor* y *Computadora* cada una de éstas contiene un conjunto de atributos que las caracteriza y con los cuales guardarán información para realizar sus instancias; al crear sus instancias dándoles valores a sus atributos, se hace una representación de ellas con el diagrama de objetos, el cual nos mostrará las asociaciones que hay entre los objetos. En este caso se presenta al objeto Bob de la clase Autor que está asociado con los objetos *PC_de_Bob_Casa* y *PC_de_Bob_Trabajo* de la clase Computadora.

3.3.4 Diagrama de estados

Un diagrama de estados es típicamente un complemento de la descripción de una clase; muestra todos los posibles estados que los objetos de una clase puedan tener, y qué eventos causan un cambio de estado. Un evento puede ser otro objeto que envía un mensaje, o alguna condición alcanzada. Un cambio de estado es conocido como una transición. Una transición también puede tener una acción conectada que especifica que deberá realizarse en conexión con la transición de estado.

El diagrama de estados no se dibuja para todas las clases, solamente para aquellas que tienen un número de estados bien definidos y donde el comportamiento de la clase es afectado y cambiado por los diferentes estados. Los diagramas de estados pueden también dibujarse para el sistema como un todo.

El diagrama de estados siguiente muestra los estados de un Elevador, aquí se muestran cinco estados posibles en los que se puede encontrar un elevador, empezando por el estado *En_Planta_Baja*, de ese estado sólo se puede ejecutar la operación de *Sube(piso)* y hace que se pase al estado *Subiendo*, una vez que se termina de subir se pasa al estado *Quieto*, en el cual se tendrán tres opciones, *Baja(piso)*, *Sube(piso)* y *Finaliza_Tiempo*, al ejecutarse la operación *Baja(piso)* se pasa al estado *Bajando* y una vez que finalice la operación se regresa al estado *Quieto*, la operación *Sube(piso)* nos regresa al estado *Subiendo* y la operación *Finaliza_Tiempo* nos lleva al estado *Moviéndose_a_Planta_Baja* y después al estado *En_Planta_Baja*, donde empezaría otra vez el elevador.

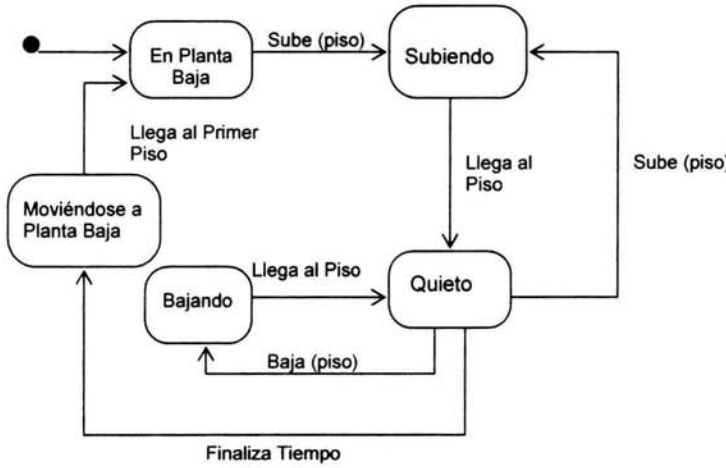


Figura 9. *Diagrama de estados para un elevador.*

3.3.5 Diagrama de secuencia

Un diagrama de secuencia muestra una colaboración dinámica entre un número de objetos. El aspecto importante de este diagrama es mostrar la secuencia de los mensajes que son enviados entre los objetos, así se muestra una interacción entre los objetos, y lo que sucede en un punto específico de ejecución del sistema. El diagrama consiste en un número de objetos mostrados con líneas verticales. El tiempo transcurre hacia abajo en el diagrama, y éste muestra el intercambio de mensajes entre los objetos conforme pasa el tiempo en la secuencia o la función. Los mensajes se muestran como líneas con flechas de mensaje entre las líneas verticales de objetos. Las especificaciones de tiempo y otros comentarios se agregan en un script al margen del diagrama. En el diagrama de secuencia siguiente se muestran cuatro clases (Computadora, Servidor_de_Impresión, Impresora, Cola) que representan a los objetos con los que se va a interactuar, de entrada

se le manda una función al objeto Computadora, este objeto hace la petición al objeto Servidor_de_Impresión, este último verifica si el objeto Impresora está disponible; si es así, manda a imprimir el archivo al objeto Impresora, en caso contrario, es decir, que la impresora esté ocupada, se manda el archivo al objeto Cola, donde se almacenara hasta que se desocupe la impresora para que el archivo se imprima, una vez que se termina de imprimir un archivo, se regresa la secuencia al objeto Computadora.

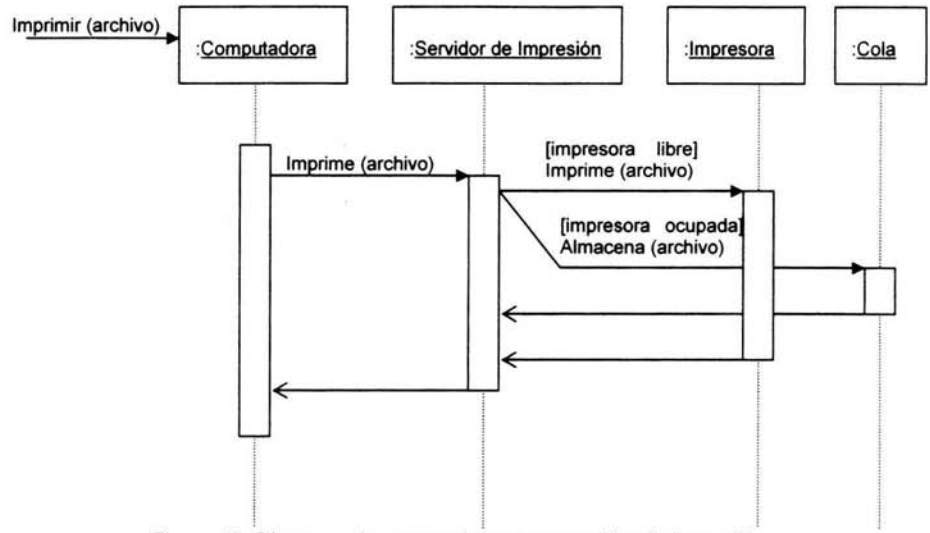


Figura 10. *Diagrama de secuencia para un servidor de impresión.*

3.3.6 Diagrama de colaboración

Un diagrama de colaboración muestra una colaboración dinámica, tal como el diagrama de secuencia. Una colaboración puede mostrarse como un diagrama de secuencia o como un diagrama de colaboración. Además, para mostrar el intercambio de mensajes (llamado la *interacción*), el diagrama de colaboración muestra los objetos y sus relaciones (algunas veces referidas como el contexto). El uso de un diagrama de secuencia o de un diagrama de colaboración puede decidirse con base en los siguientes criterios: si el tiempo o secuencia es el aspecto más importante a resaltar, debe elegirse un diagrama de secuencia; si lo importante es enfatizar el contexto, deberá elegirse un diagrama de colaboración. La interacción entre los objetos se muestra en ambos diagramas.

El diagrama de colaboración se dibuja como un diagrama de objetos, donde se muestra un número de objetos junto con sus relaciones (usando la notación del diagrama de clases/objetos). Las flechas de mensaje se dibujan entre los objetos para mostrar el flujo de mensajes entre los objetos. Las etiquetas se colocan sobre los mensajes, que entre otras cosas, muestran el orden en que los mensajes son enviados. Pueden también mostrar condiciones, iteraciones, valores de retorno, etc. Una vez familiarizado con la

sintaxis de las etiquetas de mensaje, un desarrollador puede leer la colaboración y seguir el flujo de ejecución y el intercambio de mensajes. Un diagrama de colaboración puede también contener objetos activos, aquellos que se ejecutan concurrentemente con otros objetos activos.

En el siguiente diagrama de colaboración de un servidor de impresión, se muestran, como en el diagrama de secuencia anterior, cuatro clases (*Computadora, Servidor_de_Impresión, Impresora, Cola*), en este diagrama se muestra en forma numerada la manera en que se deben realizar las operaciones. La primer función es *Imprime(archivo)*, esta es enviada por el objeto *Computadora* al objeto *Servidor_de_Impresión*, de aquí el objeto *Servidor_de_Impresión* puede ejecutar una de dos operaciones según el estado del objeto *Impresora*, si la impresora está libre se realiza la operación *Imprime(archivo)* que se envía al objeto *Impresora*, en caso contrario se realiza la operación *Almacena(archivo)* y se envía al objeto *Cola*, así finaliza la colaboración de los objetos.

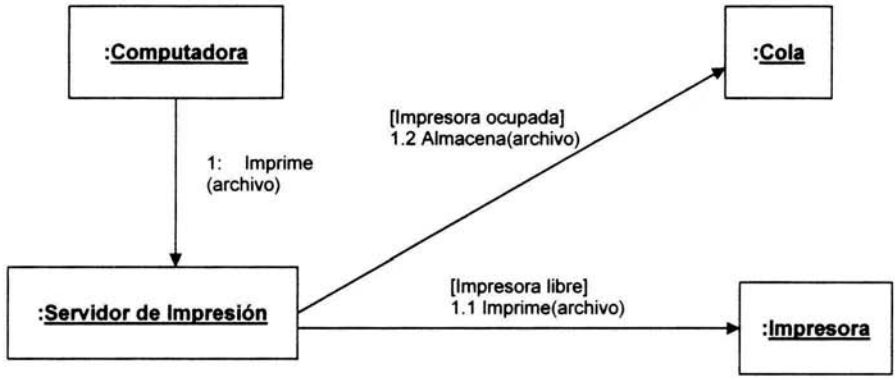


Figura 11. *Diagrama de colaboración para un servidor de impresión.*

3.3.7 Diagrama de actividad

Un diagrama de actividad muestra un flujo secuencial de actividades. El diagrama de actividad es típicamente usado para describir las actividades desempeñadas en una operación, pero puede aplicarse para describir otros flujos de actividades, tal como un caso de uso o una interacción. El diagrama de actividad consiste en estados de acción, que contienen la especificación de una actividad a ser desempeñada (una acción). Un estado de acción dejará dicho estado cuando la acción haya sido realizada (un estado en un diagrama de estados necesita la ejecución de un evento explícitamente, para abandonar el estado). Así, el control fluye entre los estados de acción, que son conectados entre sí. Decisiones y condiciones, así como ejecución paralela de estados de acción, puede también contener especificaciones de mensajes siendo enviados o recibidos como parte de las acciones desempeñadas. En la figura 12 se muestra el diagrama de actividades de un servidor de impresión, aquí se muestra el envío de la acción ImprimirArchivo(), se tienen dos opciones si no hay espacio en el disco se muestra un mensaje de disco lleno y termina la actividad, en caso contrario, se muestra un

mensaje de Imprimiendo, se genera un archivo postcript y se ejecuta la función de imprimir, termina de imprimirse el archivo, se remueve el mensaje y se termina la actividad.

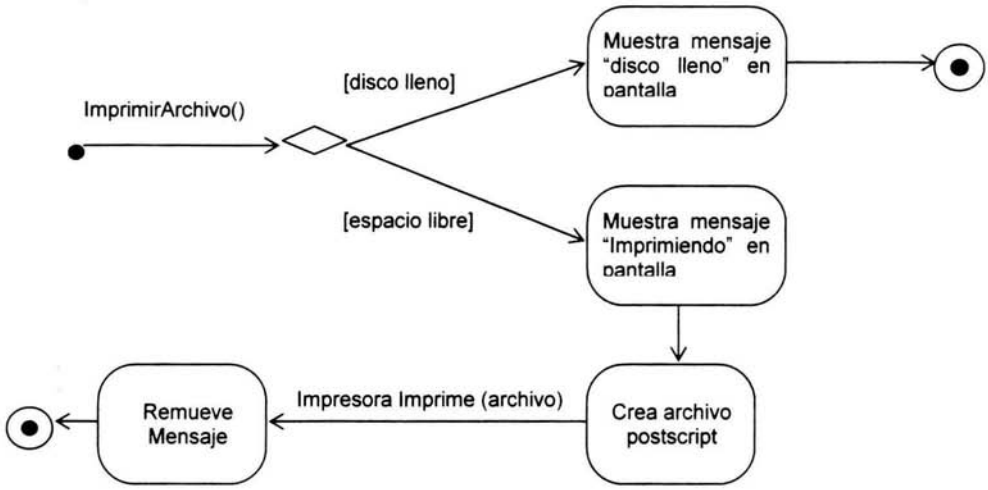


Figura 12. *Diagrama de actividad para un servidor de impresión.*

3.3.8 Diagrama de componentes

Un diagrama de componentes muestra la estructura física del código en términos de componentes de código. Un componente puede ser un componente de código fuente, un componente binario o un componente ejecutable. Un componente contiene información sobre la clase lógica o clases que implementa, creando así un mapeo de la vista lógica, a la vista de componentes. Se muestran las dependencias entre componentes, haciendo más fácil el análisis de cómo son afectados otros componentes al hacer algún cambio en cierto componente. Los componentes también pueden mostrarse con cualquiera de las interfaces que usan, así como ser agrupados en paquetes. El diagrama de componentes es usado en trabajos de programación práctica. En la siguiente figura se muestra un ejemplo de un diagrama de componentes, con los tres tipos de componentes, se tienen los archivos de códigos fuente (whnd.cpp, comhnd.cpp y main.cpp), códigos binarios (whnd.obj, comhnd.obj, main.obj y graphic.dll) y el código ejecutable (client.exe).

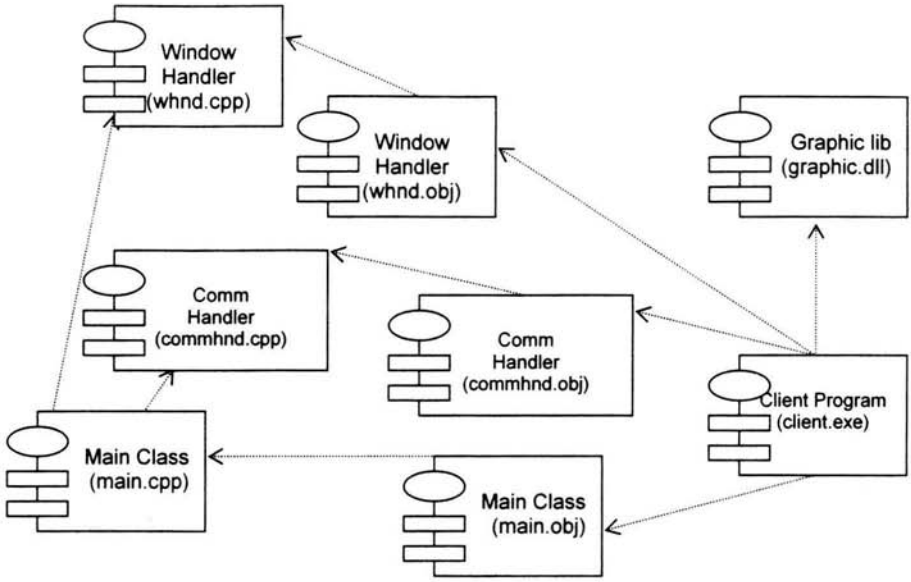


Figura 13. *Diagrama de componentes mostrando dependencias entre componentes de código.*

3.3.9 Diagrama de despliegue

El diagrama de despliegue, muestra la arquitectura física del hardware y software en el sistema. Es posible mostrar las computadoras y dispositivos (nodos), junto con las conexiones que tienen entre sí; también es posible mostrar el tipo de conexiones. Dentro de los nodos, los componentes ejecutables y los objetos son asignados para mostrar qué unidades de software se ejecutan en cuáles nodos. También pueden mostrarse dependencias entre componentes.

Como ya se ha establecido, el diagrama de despliegue, muestra la vista de despliegue, describiendo la arquitectura física del sistema. Esto está muy lejos de la descripción funcional de la vista de casos de uso. Sin embargo, con un modelo bien definido, es posible navegar todo el camino desde un nodo en la arquitectura física hacia sus componentes, hacia las clases que lo implementan, hacia las interacciones de los objetos de la clase, y finalmente hacia el caso de uso en el que se dan dichas interacciones. En la figura 14 se muestra el diagrama de despliegue de un sistema que está compuesto por cuatro componentes, cada componente es un equipo diferente, los componentes *ClienteA* y *ClienteB* (equipo Compaq Pro PC) son los que harán las peticiones hacia el componente *Servidor_de_Aplicaciones* (equipo Silicon Graphics O2) el cual hace peticiones a la base de datos a través del componente *Servidor_de_Base_de_Datos* (equipo VAX).

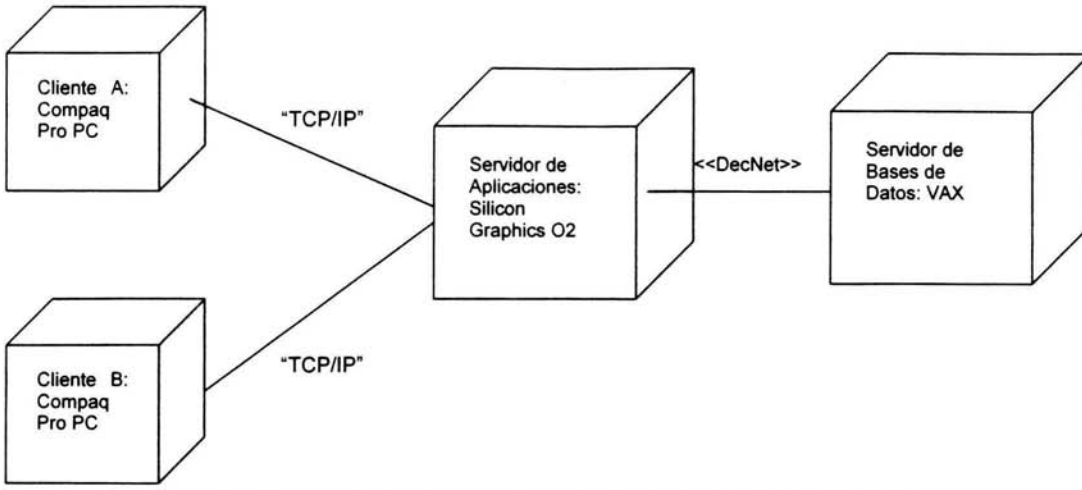
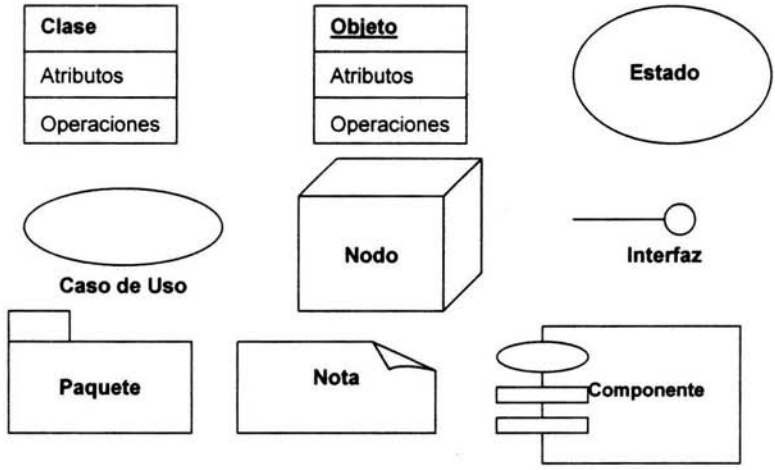


Figura 14. *Diagrama de despliegue que muestra la arquitectura física de un sistema.*

3.4 Elementos de Modelado

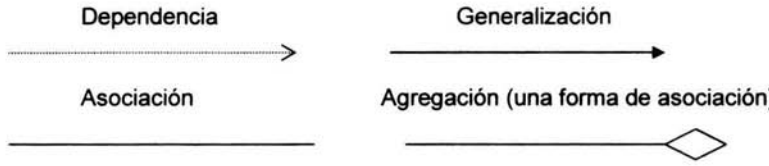
Los conceptos usados en los diagramas son llamados elementos de modelado. Un elemento de modelado es definido con semántica, una definición formal del elemento, o el significado exacto de lo que representa en enunciados no ambiguos. Un elemento de modelado tiene también su correspondiente elemento de vista, que es la representación gráfica del elemento o el símbolo gráfico usado para representar al elemento en diagramas. Un elemento puede existir en varios tipos diferentes de diagramas, pero bajo ciertas reglas. Ejemplos de elementos de modelado son:



Otros elementos de modelado son también las relaciones, que son usadas para conectar otros elementos de modelado entre sí

Ejemplos de estas relaciones son:

- *Asociación:* conecta elementos y liga instancias.
- *Generalización:* también llamada herencia, significa que un elemento puede ser una especialización de otro.
- *Dependencia:* muestra que un elemento depende en algún sentido, de otro elemento.
- *Agregación:* una forma de asociación en la cual un elemento contiene a otros elementos.



Otros elementos de modelado, además de los descritos, incluyen mensajes, acciones y estereotipos.

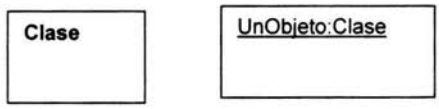
3.5 Mecanismos Generales

UML utiliza ciertos mecanismos generales en todos los diagramas, para información adicional en dicho diagrama, por lo regular para aquello que no se puede representar usando los elementos de modelado, estos mecanismos son:

- Tipografía
- Notas
- Especificaciones

3.5.1 Tipografía

El nombre de una clase se escribe en negritas, y cuando se trata de un objeto, instancia de una clase, se escribe subrayado y puede especificar tanto el nombre de la clase, como el nombre de la instancia. A continuación se muestra un ejemplo.

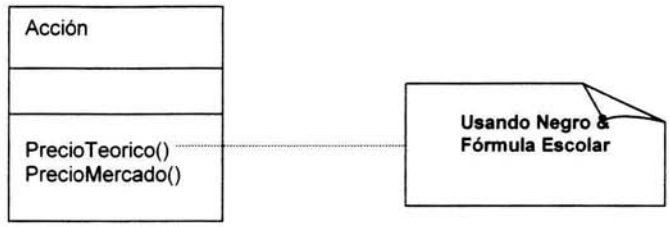


3.5.2 Notas

No todo puede ser definido en un lenguaje de modelado, sin importar que tan extenso sea el lenguaje. Para poder agregar información extra a un modelo que de otro modo no podría mostrarse, UML cuenta con la capacidad de manejar notas. Una nota puede

colocarse en cualquier parte en un diagrama y puede contener cualquier tipo de información que no sea interpretada por UML. La nota es típicamente agregada a algún elemento en el diagrama con una línea punteada que especifica cuál elemento es explicado o detallado, junto con la información en la nota.

Una nota generalmente contiene comentarios o preguntas que el modelador plantea para su posterior solución. Las notas también pueden tener estereotipos que describan el tipo de nota.



3.5.3 Especificaciones

Los elementos de modelado tienen propiedades que contienen datos sobre el elemento. Una propiedad se define con un nombre y un valor llamado "tagged value" que es de un tipo específico como un entero o un String. Existe un número de propiedades predefinidas tales como Documentación, Responsabilidad, Persistencia y Concurrencia. Las propiedades se utilizan para agregar especificaciones adicionales sobre instancias de elementos que normalmente no se muestran en el diagrama. Se manejan generalmente como ventanas de texto con pestañas, que surgen dando un doble click al elemento del diagrama en cuestión. Estas especificaciones se dan en las herramientas que se utilizan para realizar el modelado.

3.6 Extendiendo el UML

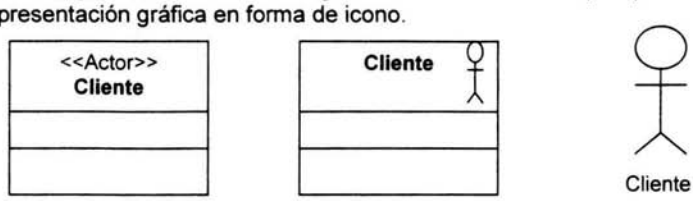
UML puede ser extendido o adaptado a un método específico, organización o usuario. Ejemplos de estas extensiones son:

- Estereotipos
- Valores Etiquetados (Tagged Values)
- Restricciones (Constraints)

3.6.1 Estereotipos

Un estereotipo es un mecanismo de extensión que define un nuevo tipo de elemento del modelo basado en un elemento ya existente. Así, un estereotipo es "como" un elemento existente, más alguna semántica extra no presente en el original. Un estereotipo de un elemento puede ser usado en las mismas situaciones en las cuales el elemento original es usado. Los estereotipos están basados en todo tipo de elementos –clases, nodos, componentes, y notas, así como en relaciones tales como asociaciones, generalizaciones

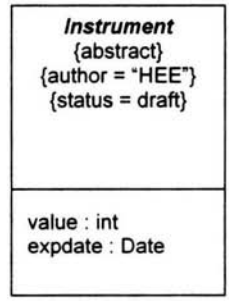
y dependencias. Un número de estereotipos están predefinidos en el UML, y son usados para ajustar un elemento de modelo existente en vez de definir uno nuevo, lo cual mantiene el UML simple. Un estereotipo es descrito colocando su nombre como una cadena –por ejemplo, <<NombreEstereotipo>>– sobre el nombre del elemento. Los paréntesis angulares son llamados "guillemets". Los estereotipos pueden tener también una representación gráfica en forma de icono.



El cliente es una clase con el estereotipo <<Actor>>. El estereotipo agrega semántica extra a la clase; en este caso, significa que la clase representa un usuario externo del sistema.

3.6.2 Valores Etiquetados (Tagged Values)

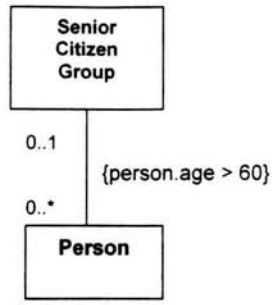
Como ya se describió antes, los elementos pueden tener propiedades que contienen pares nombre-valor de información sobre ellos. Estas propiedades son también llamadas "tagged values". Un cierto número de propiedades están definidas en UML, pero pueden definirse también por el usuario para mantener información adicional sobre los elementos. Cualquier tipo de información puede agregarse a los elementos: información específica de métodos, información administrativa sobre el progreso del modelado, información usada por otras herramientas, o cualquier tipo de información que el usuario quiera agregar a los elementos.



Propiedades sobre una clase Instrument. *abstract* es una propiedad predefinida; *author* y *status* son tagged values definidos por el usuario.

3.6.3 Restricciones (Constraints)

Un constraint es una restricción sobre un elemento que limita el uso de los elementos o la semántica (significado) del elemento. Un constraint puede ser declarado en una herramienta y usado repetidamente en diferentes diagramas, o puede ser definido y aplicado según se necesite en un diagrama.



Un constrain restringe qué objetos Persona pueden participar en la asociación.

3.7 Modelando con UML

Al construir sistemas con UML, no se construye solamente un modelo. Existen diferentes modelos en diferentes fases del desarrollo, y el propósito de cada modelo es diferente. En la fase de análisis, el propósito del modelo es capturar los requerimientos del sistema y modelar las clases básicas del "mundo-real" y sus colaboraciones. En la fase de diseño, el propósito del modelo es expandir el modelo del análisis a una solución técnica funcional, considerando el ambiente de implantación. En la fase de implantación, el modelo es la fuente misma del código que se convierte en aplicaciones. Y, finalmente, en el modelo de despliegue (deployment) se explica cómo se va a desplegar el sistema en la arquitectura física. El rastreo entre las fases y los modelos es mantenido a través de propiedades o relaciones de refinamiento. Aun cuando los modelos son diferentes, son normalmente construidos expandiendo los contenidos de modelos anteriores. A causa de esto, todos los modelos deberían mantenerse, de modo que sea fácil regresar y rehacer o expandir el modelos de análisis inicial, e introducir gradualmente los cambios en los modelos de diseño e implantación. UML es independiente de la fase, lo cual significa que el mismo lenguaje genérico y los mismos diagramas son usados para modelar diferentes cosas en diferentes fases. Depende del modelador el decidir el propósito y el alcance de un modelo. El lenguaje de modelado sólo proporciona la habilidad de crear modelos de una manera consistente y expresiva.

Al modelar con UML, el trabajo debería regirse por un método o un proceso que tome en cuenta los diferentes pasos a seguir y como deben ser llevados a cabo. Tal proceso divide típicamente el trabajo en iteraciones sucesivas de:

- Análisis de requerimientos.
- Análisis.
- Diseño.
- Implantación.
- Instalación (Deployment).

Sin embargo, también hay un proceso más pequeño que concierne con el trabajo de modelado. Normalmente, cuando se produce un modelo o un diagrama simple, el trabajo se inicia al reunir un grupo de gente adecuado que presente el problema y los objetivos;

este grupo se enfrasca en una sesión informal de lluvia de ideas y de bosquejos iniciales, durante la cual se intercambian ideas posibles de modelado. Las herramientas usadas son muy informales -usualmente notas Post-it y un pizarrón blanco. Esta sesión continúa hasta que los participantes sienten que tienen una base práctica para las bases del modelo (una hipótesis inicial). El resultado es puesto entonces en una herramienta; el modelo de la hipótesis se organiza y un diagrama se construye con base en las reglas del lenguaje de modelado. A continuación, el modelo se detalla a través de ciclos iterativos durante los cuales se descubren y se documentan más detalles sobre la solución. Mientras más información se reúna sobre el problema y su solución, la hipótesis gradualmente se convierte en un diagnóstico utilizable de solución. Cuando el modelo está casi terminado, se lleva a cabo una etapa de integración y verificación que lleva a integrar el modelo o diagrama con otros modelos o diagramas en el mismo proyecto para asegurar que no existan inconsistencias. El modelo también se valida para verificar si resuelve el problema.

Finalmente, el modelo es implantado en algún tipo de prototipo que se evalúa para encontrar deficiencias en la solución. Las deficiencias incluyen cosas tales como omitir funcionalidad, mal rendimiento, o altos costos de desarrollo. Las deficiencias deben llevar a los desarrolladores de regreso a los pasos respectivos con el fin de eliminarlas. Si los problemas son mayores, los desarrolladores pueden tener que recorrer de nuevo todo el camino desde cero. Si el problema es menor, los desarrolladores probablemente sólo tengan que cambiar partes de la organización o la especificación del modelo.

3.8 Resumen

UML es una herramienta que se ha desarrollado para facilitar el modelado bajo el paradigma orientado a objetos, ayuda a modelar en todas las fases de desarrollo de un sistema, desde el análisis hasta la implementación. UML esta constituido por cuatro partes generales que son: vistas, diagramas, elementos de modelado y mecanismos generales.

Las vistas nos permiten mostrar diferentes partes del sistema que se va desarrollar, cada vista se apoya en un grupo de diagramas para visualizar algún aspecto del sistema, existen 5 tipos, las cuales son: vista de casos de uso, vista lógica, vista de componentes, vista de concurrencia y vista de despliegue; cada una de éstas sirve a un determinado grupo de personas dependiendo su perfil: clientes, diseñadores, desarrolladores, probadores e integradores entre otros.

Los diagramas son parte de una vista y muestran gráficamente aspectos del sistema a desarrollar, los diagramas que existen en UML son: De casos de uso, de clases, de objetos, de estados, de secuencia, de colaboración, de actividad, de componentes y de despliegue, y dependiendo del tipo de información que expresan se pueden utilizar en diferentes vistas.

Los elementos de modelado son conceptos específicos usados en los diagramas, ya que tienen una semántica y expresan algo en específico, ejemplos de estos son: clase, objeto, estado, caso de uso, nodo, interfaz, paquete, nota, componente, relaciones, mensajes y

acciones. Cada uno de estos elementos tiene una forma de representación gráfica y algunos pueden usarse en diferentes diagramas bajo ciertas reglas.

Los mecanismos generales son utilizados para transmitir algún tipo de información que no puedan ser representados por los elementos de modelado, los mecanismos son: tipografía, diferentes tipos de notas y especificaciones.

Además de los cuatro elementos que conforman el UML mencionados, se pueden extender algunos de ellos para hacerlos más específicos o den información extra, algunos ejemplos de estas extensiones son: estereotipos, valores etiquetados y restricciones.

En general UML permite desarrollar diferentes modelos durante la realización de un sistema, cada uno de éstos representan diferentes aspectos del sistema y van sufriendo modificaciones durante el proceso de creación del sistema, algunos de los modelos pueden estar desde la primera fase de desarrollo del sistema hasta el final, otros van creándose de acuerdo a la fase en la que vaya el proceso y la información que se requiera mostrar.

Conclusiones

En la actualidad la orientación a objetos es el paradigma que se está utilizando para el desarrollo de la mayoría de las aplicaciones de software. La orientación a objetos es una forma de pensar de analistas y programadores para encontrar la solución a algún problema en el área de la computación, y representar de forma más natural la solución a éste. La orientación a objetos permite crear un ambiente similar al que se lleva en la vida real a través de la concepción de todas las cosas como objetos y la relación que existe entre éstos. La orientación a objetos vino a facilitar y a mejorar la programación y con esto a involucrarse en más áreas de la computación, además de los lenguajes de programación, una de estas áreas es la de las bases de datos.

Las bases de datos, en especial las relacionales, empezaron a simplificar el manejo de los datos para los sistemas que se desarrollaban, empezaron a tener problemas con los tipos de datos que se están o se desean utilizar en la actualidad, que son los de dispositivos multimedia (vídeo, sonido, imágenes, etc.). Con este problema de los tipos de datos abstractos y al avanzado manejo que tiene la orientación a objetos sobre éstos se empezaron a crear bases de datos bajo el paradigma de la orientación a objetos.

Las bases de datos orientadas a objetos se crearon para mejorar el manejo de los tipos de datos más complejos, que las bases de datos relacionales no los manejan. Éstas se empezaron a utilizar más en las siguientes áreas de la computación: desarrollo de aplicaciones multimedia, sistemas de diseño asistido por computadora CAD, ingeniería de software asistida por computadora CASE, etc.; esto se dio por el uso de objetos que dan estas bases de datos y por la necesidad de las herramientas para utilizar objetos dentro de su funcionalidad, por lo que las bases de datos relacionales no podían utilizarse, ya que no ofrecían la funcionalidad requerida por las herramientas.

Aunque la orientación a objetos es lo más usado en diferentes áreas de la computación, en las bases de datos hay una contienda por la implantación de bases de datos orientadas a objetos y las bases de datos relacionales, ya que estas últimas son las que se han empleado desde hace mucho tiempo en empresas importantes y su funcionalidad ha resuelto los problemas de manejo de información que han tenido. La implementación de nuevos tipos de datos ha hecho que se ponga en disputa qué tipo de manejador de base de datos se pueda utilizar, ya que la funcionalidad de los manejadores relacionales ha sido sencilla, eficaz, pero con estos tipos se vio limitada dicha funcionalidad; los manejadores de bases de datos orientadas a objetos se crearon para solucionar el problema de los tipos de datos.

Por eso, así como evolucionaron las bases de datos, desde el manejo de archivos pasando por el modelo jerárquico y de red hasta llegar al modelo relacional, ahora les toca evolucionar de éste último al modelo de objetos, lo que se pretende ahora es crear sistemas manejadores de bases de datos orientados a objetos, ya que es el siguiente paso de la evolución en bases de datos, y así como el modelo relacional sustituyó a los modelos de red y jerárquico, lo que se busca es reemplazar las bases de datos relacionales con las bases de datos orientadas a objetos. Este propósito aún no se puede realizar debido a que las bases de datos relacionales han tenido un gran desempeño y es muy difícil su reemplazo, aunque han salido deficiencias por el manejo de los tipos de

datos, esto no es motivo suficiente para reemplazarlas y lo que se ha hecho, para poder aumentar su funcionalidad, es adaptarles las siguientes características de la orientación a objetos:

- Relaciones anidadas
- Generación de tipos complejos (Tipos estructurados, de referencia y colecciones)
- Herencia
- Identidad de objetos
- Ampliación de SQL en SQL3 para soportar las características orientadas a objetos

Las bases de datos orientadas a objetos se han empleado más para el desarrollo de herramientas, como se mencionó anteriormente.

La creación de sistemas manejadores de bases de datos orientados a objetos se empezó a dar en diferentes compañías desarrolladoras de software, y esto dio la pauta a que se crearan diferentes estructuras de los manejadores, ya que no se tenía un estándar en el que se podrían basar, ya que cada quien creaba su propia estructura; para esto se formó el ODMG, el cual creó un estándar basándose en un modelo de objetos, de funciones determinadas que deberán cumplir los manejadores que se creen bajo el paradigma de la orientación a objetos, y de un lenguaje para la creación y manipulación de los objetos similar a SQL llamado OQL.

La orientación a objetos se ha convertido en la mejor forma de desarrollo de software, por eso se creó el proceso UML, lenguaje de modelado unificado, el cual nos permite manejar un problema desde el inicio, y proporciona muchas herramientas a través de una gran variedad de diagramas, elementos de modelado y mecanismos para poder resolver dicho problema. UML permite representar el problema y la solución a través de elementos gráficos, esto con la finalidad de que sea más fácil la percepción del problema y su solución para los desarrolladores, así como para explicar al usuario de forma gráfica y para que no sea tan técnica, la percepción del problema y la solución a éste.

Bibliografía

- [BEDD, 1996] Bertino, E.; Ditrich, K. y Díaz, O.; *Seminario Sistemas Gestores de Bases de Datos Orientadas a Objetos*, Fundación Universidad Empresa de Murcia, 1996.

- [BLPR, 1998] Blaha, M. y Premerlani, W.; *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, 1998.

- [BOOC, 1996] Booch, Grady; *Análisis y Diseño Orientado a Objetos*, Addison-Wesley/Diaz de Santos, 1996.

- [CABA, 1997] Cattell, R.G.G.; Barry, Douglas K.; *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers, Inc., 1997.

- [CEBA, 1998] Ceballos, J.; *Programación Orientada a Objetos con C++*, De. Ra-Ma, 1998.

- [HAHA, 1997] Hansen, G. y Hansen, J.; *Diseño y Administración de Bases de Datos*, Prentice Hall, 1997.

- [JCÖJ, 1992] Jacobson, I.; Christerson, M.; Jonson, P. y Övergaard, G.; *Object-Oriented Software Engineering* Addison-Wesley, 1992.

- [JORD, 1198] Jordan, David; *C++ Object DataBases: Programming with the ODMG Standard*, Addison-Wesley, 1998.

- [LARM, 1999] Larman, Craig; *UML y Patrones Introducción al análisis y diseño orientado a objetos*, Prentice Hall, 1999.

- [SIKS, 1998] Silberchatz, A.; Korth, H. y Sudarshan, S.; *Fundamentos de Bases de Datos*, McGrawHill, 1998.

- [UJWJ, 1999] Ullman, Jefrey D. y Widom, Jennifer; *Introducción a los Sistemas de Bases de Datos*, Prentice Hall, 1999.

- [WIWW, 1990] Wirfs-Brock, R.; Wilkerson, B. y Wiener, L.; *Designing Object-Oriented Software*, Prentice Hall, 1990.