UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA

ESTRUCTURA INTERNA DE LAS BASES DE DATOS ORIENTADAS A OBJETOS

TESIS

PRESENTADA A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

CLAUDIA LICETH ROJAS MORALES
AL COFERÍRSELE EL TITULO DE
INGENIERA EN CIENCIAS Y SISTEMAS

GUATEMALA, OCTUBRE DE 1997.

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de tesis titulado:

ESTRUCTURA INTERNA DE LAS BASES DE DATOS ORIENTADAS A OBJETOS

tema que me fuera asignado por la coordinación de la carrera de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería, con fecha 5 de septiembre de 1996.

CLAUDIA LICETH ROJAS MORALES

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA

MIEMBROS DE LA JUNTA DIRECTIVA

DECANO	Ing. Herbert René Miranda Barrios
VOCAL 1ero.	Ing. Miguel Ángel Sánchez Guerra
VOCAL 2do.	Ing. Jack Douglas Ibarra Solórzano
VOCAL 3er.	Ing. Juan Adolfo Echeverría Méndez
VOCAL 4to.	Br. Victor Rafael Lobos Aldana
VOCAL 5to.	Br. Wagner López Cáceres
SECRETARIO	Ing. Gilda Marina Castellanos de Illescas

TRIBUNAL QUE PRACTICO EL EXAMEN
GENERAL PRIVADO

DECANO	Ing. Herbert René Miranda Barrios
EXAMINADOR	Ing. Jorge Luis Álvarez Mejía
EXAMINADOR	Ing. Aldo Alexander Sagastume Reyes
EXAMINADOR	Ing. Byron Wosbelí López López
SECRETARIO	Ing. Gilda Marina Castellanos de Illescas

Guatemala, 11 de Agosto de 1,997

Ingeniero
Jorge Luis Alvarez Mejía
Coordinador de Area
Escuela de Ciencias y Sistemas
Facultad de Ingeniería
Universidad de San Carlos de Guatemala

Ingeniero Alvarez:

Por medio de la presente, hago de su conocimiento que he revisado completamente el trabajo de tesis del estudiante Claudia Liceth Rojas Morales, titulado ***Estructura Interna de las Bases de Datos Orientadas a Objetos.*** Puedo concluir que la misma llena los objetivos propuestos en el anteproyecto de tesis.

Por lo tanto, el autor de esta tesis y yo como su asesor, nos hacemos responsables por el contenido y conclusiones de la misma.

Atentamente,

Jorge Luis Alvarez Mejía
Ingeniero en Ciencias y Sistemas
Colegiado No. 3731

Guatemala, 24 de septiembre de 1997

Ingeniero
Jorge Luis Alvarez Mejía
Coordinador Ingeniería en Ciencias y Sistemas
Facultad de Ingeniería
Universidad de San Carlos de Guatemala
Presente

Ingeniero Alvarez:

Por este medio me permito hacer de su conocimiento, que he procedido a revisar el trabajo de tesis titulado **"Estructura Interna de las Bases de Datos Orientadas a Objetos"**, elaborado por el estudiante Claudia Liceth Rojas Morales, a mi juicio, el mismo cumple con los objetivos propuestos para su desarrollo.

Sin otro particular me suscribo de usted.

Atentamente,

Francisco Javier Guevara Castillo
Ingeniero en Ciencias y Sistemas
Revisor.

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERIA

Escuelas de Ingeniería Civil, Ingeniería Mecánica Industrial, Ingeniería Química, Ingeniería Mecánica Eléctrica, Técnica y Regional de Post-grado de Ingeniería Sanitaria.

Ciudad Universitaria, zona 12
Guatemala, Centroamérica

Guatemala, 26 de Septiembre 1,997

Ingeniero
Herbert Renè Miranda Barrios
Decano
Facultad de Ingenierìa

Estimado Sr. Decano:

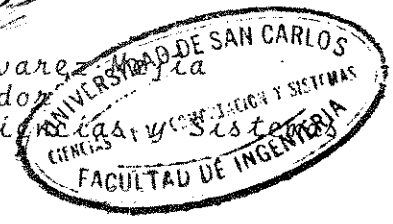
Atentamente me dirijo a usted, para informarle que después de conocer el dictamen del Asesor del trabajo de tesis del estudiante Claudia Liceth Rojas Morales, titulado "Estructura interna de las bases de datos orientadas a objetos", procedo a la autorización del mismo.

Sin otro particular, me suscribo con las muestras de mi consideración y estima,

Atentamente,

ID Y ENSEÑAD A TODOS

Ing. Jorge Luis Alvarez Mejía
Coordinador
Carrera de Ingeniería en Ciencias y Sistemas



UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERIA

Escuelas de Ingeniería Civil, Ingeniería Mecánica Industrial, Ingeniería Química, Ingeniería Mecánica Eléctrica, Técnica y Regional de Post-grado de Ingeniería Sanitaria.

Ciudad Universitaria, zona 12
Guatemala, Centroamérica

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la autorización por parte del Coordinador de la Carrera de Ingeniería en Ciencias y Sistemas, al trabajo de tesis titulado ESTRUCTURA INTERNA DE LAS BASES DE DATOS ORIENTADAS A OBJETOS, presentado por el estudiante universitario, CLAUDIA LICETH ROJAS MORALES, procede a la autorización para la impresión de la misma.

IMPRIMASE

Ing. Herbert René Miranda Barrios
DECANO

UNIVERSIDAD DE SAN CARLOS
DECANO
FACULTAD DE INGENIERIA
DE GUATEMALA

Guatemala, octubre de 1,997

DEDICATORIA

A DIOS:

Si tengo una profesión es porque Tú me la diste, Tú me has permitido administrarla para Ti. Tú que eres Dios y honra perfecta. Gracias Señor por permitirme poner la última piedra.

A LA VIRGENCITA:

Señora, Madre Mía, si tu no me hubieras acompañado en todo momento como acompañaste a tu Santísimo Hijo, nada me hubiera sido posible.

A MIS PADRES:

Por la maravillosa herencia que me han dado: "mis estudios". Por su amor, abnegación, entrega y compresión.

A MIS HERMANITOS:

Lily y Roberto, por todo su apoyo, su ayuda, su comprensión. Con mucho amor para ustedes.

AGRADECIMIENTOS

A mi asesor:

Jorge Luis Álvarez, por su amistad, y por brindarme sus conocimientos, experiencia y tiempo, de manera incondicional.

A mis familiares, amigos y compañeros:

Isa, Amanda, Cristy, por su apoyo, entusiasmo y optimismo incondicionales.

Carol, por la ayuda, que con tanto cariño me brindaste.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

GLOSARIO

Atomicidad	Se refiere a algo que no posee estructura y que no puede subdividirse más. Es algo que o bien existe o no existe.
Byte	Grupo pequeño de bits de datos que se tratan como una unidad. Un byte contiene ocho bits, y se le conoce también como octeto de bits.
Clases recursivas	Una clase recursiva presenta circularidad con respecto a su padre o a sus atributos.
Hashing	Hashing o Hash, en este contexto interprétese como mapeo o direccionamiento.
Indexamiento	En este contexto, indexamiento se refiere a indización (de índices), se ha preferido usar este término pues es más común en el ambiente de las bases de datos orientadas a objetos.
Interfaz	Se refiere a interface. Es decir, el ambiente que se le presenta al usuario al trabajar con objetos.
Navegar	Acción de seguir trayectorias o caminos entre clases para realizar el proceso deseado.
Objetos Primitivos	Objetos que no contienen más objetos dentro de ellos, es decir objetos que no son complejos.
Overloaded	Cuando diferentes tipos pueden tener operaciones definidas con el mismo nombre, se dice que estas operaciones son overloaded.
Predicado	Los predicados son condiciones o limitaciones hechas sobre los objetos.
Signatura	En el presente trabajo signatura se entiende en sentido figurado como identificación.
Trigger	Regla que se activa al ejecutar alguna acción dentro de la base de datos, para resguardad integridad y consistencia de la misma.

INTRODUCCION

Los sistemas de bases de datos orientados a objetos tienen sus orígenes en los lenguajes de programación orientados a objetos. La idea en ambos casos es que el usuario no tendrá que batallar con construcciones orientadas al computador tales como registros y campos, sino más bien debería poder manejar objetos que se asemejen más a sus equivalentes en el mundo real.

Son muchos los investigadores que proponen a los sistemas orientados a objetos como el futuro en la tecnología de bases de datos. En particular, muchos son de la opinión de que las técnicas orientadas a objetos son la estrategia más recomendable en nuevas áreas de aplicación, tales como el diseño y la manufactura asistidos por computador (CAD/CAM), la manufactura integrada por computador (CIM), la ingeniería de software asistida por computador (CASE), el almacenamiento y recuperación de documentos.

La orientación a objetos es utilizada en muchos ambientes. Los sistemas orientados a objetos en ambientes comerciales son relativamente nuevos, y pueden ser usados para construir sistemas de información.

Existe actualmente una tendencia hacia construir sistemas de gestión de bases de datos que directamente apoyen un modelo semántico, permitiendo a los usuarios referirse directamente a objetos desde el punto de vista de un modelo semántico más bien que mediante el acceso a registros.

En general, no existe aún un enfoque estándar de sistemas orientados a objetos del cual se pueda afirmar ningún uso amplio, pero las tendencias futuras son bastante alentadoras; por lo que se ha considerado importante llevar a cabo un trabajo de este tipo que ayudara a disipar dudas y a enriquecer el conocimiento actual sobre este tema en particular, del cual aún no se ha tratado mucho y en particular aclarar un poco el panorama.

Este trabajo trata especialmente de las bases de datos orientadas a objetos, y de sus principios generales de arquitectura interna que constituye su principal enfoque; por lo que se hace necesario conocer la jerarquía entre objetos, clases, las estructuras de indexamiento, una evaluación del procesamiento de consultas, desempeño, estructuras de datos, cómo se encuentran internamente los archivos, la manipulación de datos, etc.

Este documento se encuentra organizado en capítulos, que están construidos en forma secuencial, lo que implica que es conveniente que el lector conozca los capítulos anteriores antes de pasar a un nuevo capítulo.

El primer capítulo resume el manejo de datos desde sus inicios; para que el lector se de una idea de la trayectoria que a sufrido dicho manejo hasta nuestros días, a la vez que permite visualizar cómo se ha venido simplificando la manipulación de los mismos.

La lectura del capítulo dos es de vital importancia para el lector, ya que trata toda la conceptualización del ambiente orientado a objetos, contiene un preámbulo al tema de las bases de datos orientadas a objetos.

En el capítulo tres se intenta dar a conocer a las bases de datos orientadas a objetos, y se presenta un modelo orientado a objetos que pretende ser una guía que apoye al diseñador en sistemas de bases de datos orientadas a objetos. El capítulo busca que el lector se familiarice con las bases de datos orientadas a objetos así como despertar el interés hacia su manejo interno.

En el capítulo cuatro se presentan los principios de arquitectura en los sistemas de bases de datos orientados a objetos, es decir, se pretende dar a conocer la estructura interna de las bases de datos orientadas a objetos: características de almacenamiento, manejo de índices, etc.

Por último, el capítulo cinco presenta el desarrollo e implementación de un manejador de bases de datos orientado a objetos (ODBMS), en el capítulo se unifican y se llevan a la práctica los contenidos de los capítulos dos, tres y cuatro, presenta un enfoque a nivel de bases de datos orientadas a objetos y también presenta el lado de la arquitectura interna de un ODBMS.

Es necesario indicar que ésta tesis tiene un contenido teórico-práctico a nivel de conceptualización y no un enfoque a nivel de desarrollo de aplicaciones

1. CUATRO GENERACIONES DEL MANEJO DE DATOS

1.1 Introducción

Se sabe que una base de datos tradicional solo almacena datos, de modo que resulten independientes de los procedimientos. Dicho de otra forma, una base de datos es una especie de archivero electrónico, en el cual los datos son accedidos por diferentes usuarios, de diferente manera y con distintos propósitos. Contrariamente una base de datos orientada a objetos se compone únicamente de objetos (es decir almacena sólo objetos); los datos se almacenan junto con los métodos que procesan dichos datos.

En las bases de datos orientadas a objetos NO se tiene acceso a dato alguno si no es a través de los métodos almacenados en la base de datos. Estos métodos están listos para entrar en acción en el momento que reciban alguna solicitud; de manera que los datos de todos los objetos quedan encapsulados.

Si se hace una analogía con el cuerpo humano: los objetos de bajo nivel, son algo así como las células del cuerpo, se agrupan en objetos de mayor nivel, que realizan funciones útiles para las empresas y sus empresarios. Las bases de datos orientadas a objetos serán una importante tecnología en las empresas. A continuación, en este capítulo haremos un recorrido por cuatro generaciones del manejo de datos que se ilustran en la figura 1.1.

1.2 Sistemas de archivos

Al principio, los lenguajes y las instrucciones de máquina eran muy similares, lo que producía un modelo de programación orientado por procesos. Con el tiempo la mayoría de los programas utilizaron este nuevo tipo de almacenamiento en disco, pero era un tanto difícil organizar y administrar los datos en este medio, así que los diseñadores comenzaron a construir paquetes que facilitaran su manejo; y así nacieron los sistemas de administración de archivos.

El acceso de datos (lectura y escritura) en archivos requiere de mucha actividad que es transparente para el programador de la aplicación. Actualmente los lenguajes de programación permiten a los programadores definir técnicas de organización de archivos bastante complejas con instrucciones bastante simples. Un sistema de archivo proporciona el apoyo que permite al programador accesar archivos sin preocuparse de los detalles sobre las características de almacenamiento y tiempos de acceso. Las responsabilidades de un sistema de archivo son muchas y muy variadas, incluyen las siguientes:

- Mantener un directorio de identificación de archivos y localización de información.
- Establecer rutas del flujo de datos entre la memoria principal y los dispositivos de almacenamiento secundario.
- Coordinar la comunicación entre la unidad central de procesamiento (UCP) y los dispositivos de almacenamiento secundario y viceversa.
- Preparar archivos para usarse como entrada o salida.
- Manipular los archivos cuando su uso de entrada o salida haya terminado.

Por lo general, los programas seguían la orientación a los procesos; la disponibilidad de lenguajes de alto nivel (principalmente COBOL que tenía una orientación comercial), llevó al desarrollo de grandes programas para las empresas. Estos programas veían los datos organizados en archivos, ordenados en categorías o indexados por alguna llave lógica por ejemplo, el código de cliente. Se descubría entonces el concepto de registro y un archivo como un grupo de registros.

Aún con todo esto, el sistema seguía siendo complicado, el acceso aleatorio requería que la aplicación conociese la dirección física de los datos en el disco. El cálculo de esta única dirección requería de algoritmos de dispersión conocidos como hashing; a raíz de esto surgió el archivo indexado (la primera ayuda principal independiente de la implantación). En lugar de pedirle a una aplicación la colocación exacta de una parte de los datos, sólo se requeriría de una llave simbólica. Entre los sistemas más utilizados estaban (y aún actualmente): el ISAM (método de acceso secuencial indexado) y el VSAM (método de acceso secuencial virtual).

1.3 Los primeros sistemas de administración de bases de datos

La demanda de mayor capacidad de aplicaciones seguía aumentado, y entonces se hizo evidente que incluso los sistemas de archivos indexados eran instrumentos rústicos. Esta demanda impulsó la creación de los primeros sistemas construidos sobre el sistema de archivos, conocidos como sistemas de administración de bases de datos (DBMS en inglés). Algunos procesos seguían una naturaleza jerárquica es decir tenían varios elementos de línea; el DBMS jerárquico más conocido era IMS (siglas en ingles del sistema de administración de la información).

Los DBMSs se basaban en un modelo de datos independiente de cualquier aplicación particular. Este nuevo paso en la independencia de la aplicación permitió a los diseñadores de aplicaciones disponer de más tiempo para la arquitectura de su aplicación. Con los DBMSs el diseño de datos se convirtió en una actividad importante, lo que provocó un cambio fundamental dirigido hacia un modelo de aplicaciones orientado por datos; esto llevó al surgimiento de otras necesidades , como disponer de varias aplicaciones que interactuaran en forma simultánea con el

DBMS y crear aplicaciones de utilerías independientes para el manejo del DBMS; de esta forma nacieron los conceptos de control de concurrencia, y la mayoría de las actividades conocidas como la administración de la base de datos: respaldo, recuperación, distribución de recursos, seguridad, etc.

Las jerarquías ofrecían un buen modelo para muchos problemas, pero pronto se dieron cuenta que el mundo no es inherentemente jerárquico y que resultaba difícil implantar los modelos no jerárquicos a partir de IMS; así que se desarrolló CODASYL. El DBMS venía con un modelo de datos utilizado para describir la empresa de forma independiente a cualquier aplicación. Se implementó un lenguaje de definición de datos (DDL en inglés) , y un lenguaje para el manejo de datos (DML en inglés); un conjunto de funciones auxiliares administraba el espacio de almacenamiento físico del DBMS, la seguridad y otros servicios.

En la medida en que los sistemas construidos en base a los DBMSs crecían, crecían también los problemas, centrados estos en la reorganización y la navegación. La reorganización era el mayor de los males, ya que el modelo conceptual y la implantación física estaban íntimamente ligados; esto implicaba solo por mencionar un caso que cualquier cambio a la organización del registro en el DBMS requería como mínimo, volver a compilar todos los programas; y peor aún si la dirección física de un registro cambiaba, había que encontrar todas las referencias a él.

La navegación era el segundo problema, y se refiere a la forma en que las aplicaciones están restringidas por el DBMS.

1.4 La base de datos relacional

El modelo relacional no se basaba en un paradigma particular para la estructuración de los datos, sino en ciertos fundamentos matemáticos.

El modelo de datos por relación cambió el centro del proceso de desarrollo del sistema de las estructuras de datos y las implantaciones de computación hacia el modelado del dominio de la aplicación empresarial. Las bases de datos por relación representaban los primeros sistemas que proporcionaban una interfaz de aplicación en la cual se eliminaban del proceso los aspectos de la implantación. Se había logrado uno de los objetivos principales: los datos eran independientes del proceso.

Se define una base de datos relacional a aquel conjunto de datos que están organizados en tablas; es decir, se sabe que una base de datos es una especie de archivero electrónico; dicho de otra forma es un lugar donde se almacena un conjunto de archivos de datos computarizados. Haciendo una analogía, a los archivos de computadora se les denomina tablas (tablas relacionales), las filas de

una tabla representan los registros del archivo y las columnas representan los campos.

El modelo relacional se representa gráficamente por el modelo Entidad-Relacion, el cual se basa en una percepción del mundo real que consiste en un conjunto de objetos básicos llamados entidades y relaciones. A continuación se describen los principales términos que se manejan en el contexto del modelo relacional (lo que comúnmente se denomina estructura del modelo relacional):

1.4.1 Tupla: una tupla es una fila de una tabla y un atributo es una columna. Al número de tuplas se le llama cardinalidad y al número de atributos se le denomina grado.

1.4.2 Dominio: se entiende por dominio a un conjunto de valores, todos del mismo tipo de los cuales un conjunto de atributos toman sus valores reales. Una definición más formal es que dominio es la menor unidad semántica de información. Los valores escalares representan la menor unidad semántica de información en el sentido de que son atómicos; es decir no poseen estructura interna.

1.4.3 Entidad: este es un concepto útil para trabajar con el modelo Entidad-Relacion. Una entidad es un objeto claramente distinguible de otros objetos, que existe, y del cual deseamos llevar información. Una entidad puede ser concreta o abstracta, un material químico, una persona son entidades del primer tipo, un concepto puede ser una entidad del segundo tipo. Una entidad esta representada por un conjunto de atributos (los atributos también se pueden identificar como características que describen a un objeto); por ejemplo: el nombre de la persona es un atributo de la entidad persona, el tipo de material es un atributo de la entidad material químico.

1.4.4 Relación: relación es lo que conocemos como tabla; pero más bien una relación es una especie bastante abstracta de objeto, está dada sobre un conjunto de dominios y básicamente se compone de dos partes: una cabecera y un cuerpo.

La cabecera esta formada por un conjunto fijo de atributos, y el cuerpo esta formado por un conjunto de tuplas, el cual varia con el tiempo. Dentro de una relación dada existen cuatro propiedades muy importantes:

- No existen tuplas repetidas.
- Las tuplas no están ordenadas (de arriba hacia abajo).
- Los atributos no están ordenados (de izquierda a derecha).

Todos los valores de los atributos son atómicos (que no se pueden descomponer). Un concepto de relación visto desde el punto de vista del modelo entidad-relación es una asociación entre varias entidades.

Los dos conceptos anteriores podrían crear un poco de confusión; es por eso importante poder especificar como se distinguen las entidades y las relaciones. Como se ha visto, conceptualmente son diferentes (desde el punto de vista del modelo entidad-relación), pero desde la perspectiva de la base de datos, la diferencia entre ellos debe especificarse únicamente en términos de atributos; el siguiente concepto ayuda a aclarar el panorama:

1.4.5 Claves: son llamadas también llaves. La clave primaria es un identificador único para la tabla, es decir un conjunto de uno o más atributos que, considerados conjuntamente, nos permiten identificar de forma única a una tabla (entidad: Modelo E-R), dicho de otra manera, nunca existen dos filas de la tabla con el mismo valor en esa columna o combinación de columnas.

1.4.6 El Algebra Relacional:

A esta parte del modelo relacional se le conoce como la parte manipulativa. El álgebra relacional es un lenguaje de consulta procedimental. Consta de un conjunto de operaciones que toman una o dos relaciones como entrada y producen una nueva relación como resultado.

Operaciones Fundamentales:

A continuación se describen a grandes rasgos los principales operadores:

1.4.6.1 Restricción: también llamada seleccionar. Extrae tuplas especificadas de una relación dada; es decir restringe la relación sólo a tuplas que satisfagan una condición dada.

1.4.6.2 Proyección: extrae los atributos especificados de una relación dada, es decir no nos preocupamos de los atributos que no nos interesan.

1.4.6.3 Operación Renombrar: devuelve a una relación R con el nombre X. Se puede hacer referencia a la relación R dos veces sin ambigüedad.

Las operaciones de restricción (seleccionar), proyectar y renombrar se denominan operaciones unitarias, ya que operan sobre una relación.

1.4.6.4 Producto: a partir de dos relaciones especificadas, construye una relación que contiene todas las combinaciones posibles de tuplas, una de cada una de las dos relaciones.

1.4.6.5 Unión: construye una relación formada por todas las tuplas que aparecen en cualquiera de las dos relaciones especificadas. Análogo al concepto de unión en teoría de conjuntos.

1.4.6.6 Intersección: construye una relación formada por aquellas tuplas que aparezcan en las dos relaciones especificadas.

1.4.6.7 Diferencia: construye una relación formada por todas las tuplas de la primera relación que no aparezcan en la segunda de las dos relaciones especificadas.

1.4.6.8 Reunión: conocida como Join. Construye una relación (de dos relaciones dadas) que contiene todas las posibles combinaciones de tuplas, una de cada una de las dos relaciones, tales que las dos tuplas participantes en una combinación dada satisfagan alguna condición especificada.

1.4.6.9 División: toma dos relaciones, una binaria y una unaria, y construye una relación formada por todos los valores de un atributo de la relación binaria que concuerdan con todos los valores en la relación unaria.

1.4.6.10 Asignación: la operación asignación, funciona de forma parecida a la operación asignación en un lenguaje de programación. Su misión es asignar el valor de alguna expresión arbitraria del álgebra a una relación nombrada.

1.4.7 Reglas de integridad relacional:

1.4.7.1 Regla No. 1: la regla No.1 es la regla de la integridad de las entidades y dice así: Ningún componente de la llave primaria de una relación base puede aceptar nulos.

1.4.7.2 Regla No. 2: esta es la regla de integridad referencial; y dice así: La base de datos no debe contener valores de llave extranjera sin concordancia.

Estas reglas se refieren a las llaves primarias y a las llaves alternas respectivamente; las primeras ya las conocemos; sabemos que una llave primaria de una relación es un identificador único de esa relación, las segundas las podemos definir más o menos así: una llave extranjera o clave ajena es un atributo (tal vez compuesto: llave formada por más de un atributo) de una relación R2 cuyos valores deben concordar con los de la llave primaria de alguna relación R1.

Las bases de datos por relación trabajan con datos pasivos es decir que se pueden activar de forma automática ciertas operaciones limitadas al utilizarlos. En general la base de datos almacena precisamente datos.

El modelo de datos por relación ofrece varias ventajas significativas sobre sus predecesores:

- Independencia de los datos: la representación de los datos en la computadora es independiente de la interfaz con la aplicación.
- Manejo declarativo: el SQL, es el lenguaje más común utilizado para expresar el modelo por relación, es un lenguaje declarativo no por procedimiento que expresa el tipo de datos deseados y no la forma de obtenerlos.
- Eliminación de la redundancia: al diseñar los datos por relación, se puede aplicar el proceso de normalización. La normalización total de un modelo de datos produce la eliminación de la redundancia.
- Sencillez: el modelo por relación es más fácil de aprender y de utilizar. La mayoría de personas (fuesen programadores o no) ya están familiarizadas con los conceptos básicos, ya que han trabajado con tablas, filas y columnas.
- Tablas como vehículos de presentación: el resultado de todos los operadores por relación es una tabla.

Debido a todas estas características, dieron la popularidad actual a las bases de datos relacionales, también dieron pie a los lenguajes de cuarta generación (4gl) que se caracterizan por ser lenguajes con operadores de una base de datos ya integrados. Todo esto volvió más simple el proceso de desarrollo de aplicaciones.

1.5 Bases de datos orientadas a objetos

Las bases de datos orientadas objetos surgieron para soportar la programación orientada a objetos. Los programadores necesitaban almacenar lo que llamaban datos persistentes[1]. Las bases de datos orientadas a objetos se volvieron importantes para aplicaciones con datos complejos como por ejemplo CAD[2] y CAE[3]; y también se volvieron importantes para el manejo de los objetos binarios de gran tamaño como las imágenes, el sonido, el video y el texto sin formato. Las bases de datos activas[4] se implantan mejor con las técnicas orientadas a objetos. Las bases con conocimiento[5] se orientaron a los marcos donde el marco es un objeto con un conjunto de reglas asociadas con él. Los tipos de datos necesitaban un soporte que incluyera habla, imágenes y vídeo. Los datos complejos necesitaban técnicas que mejorasen el desempeño sobre las bases de datos por relación.

El objetivo principal de las bases de datos orientadas a objetos es el encapsulamiento, trabaja con objetos activos (es decir las solicitudes hacen que los

[1] Datos que permanecen después de determinado proceso.
[2] Diseño apoyado por computadora.
[3] Ingeniería apoyada por computadora.
[4] Se conocen como colocar inteligencia en la base de datos.
[5] Relacionadas con inteligencia artificial, conocimiento formado por hechos y reglas que pudiera utilizar una computadora.

objetos ejecuten sus métodos) y las estructuras de datos con las que trabaje pueden ser complejas.

Las bases de datos orientadas a objetos soportan tipos de datos más variados que las simples tablas, columnas y filas de las bases de datos por relación.

Con el modelo orientado a objetos, los usuarios no tienen que entenderse con construcciones orientadas al computador como registros o campos (ni tablas, etc.), sino que más bien manejan objetos que se parecen a sus equivalentes en el mundo real; la idea de la orientación a objetos es elevar el nivel de abstracción (en los anteriores modelos). Su objetivo principal es visualizar todo como objetos.

El modelo orientado a objetos es una adaptación a los sistemas de base de datos del paradigma de programación orientado a objetos. Se basa en el concepto de encapsulamiento de datos y código que opera sobre esos datos en un objeto. Es evidente que el ocultar los detalles irrelevantes a la vista del usuario y el ofrecer un acceso disciplinado a objetos sólo a través de una interfaz pública es apropiado para las aplicaciones y no digamos para los usuarios.

1.5.1 Modificación de esquemas

Las modificaciones que pueden hacerse en el modelo relacional son bastante fáciles de llevar a cabo y básicamente son:

- Crear o quitar una relación.
- Añadir o eliminar atributos de un esquema de relaciones.

Esta simplicidad no se cumple en los esquemas de bases de datos orientadas a objetos. Generalmente se complica por los siguientes factores:

- Cambios complejos: por el mismo nivel de abstracción existente es lógico que los cambios sean complejos y no tan simples como en los otros modelos.
- Cambios frecuentes: las aplicaciones que motivan el uso del modelo orientado a objetos requieren cambios de esquema frecuentes.

Por lo regular las modificaciones se resumen en:

- Añadir una nueva clase: la adición de una nueva clase en una base de datos orientada a objetos implica más de lo que implicaba la adición de un esquema de relaciones en una base de datos relacional. La nueva clase debe colocarse en la jerarquía de clase/subclase, y deben resolverse los problemas de herencia. Si la nueva clase no es un nodo hoja en la jerarquía, puede que las subclases de la nueva clase necesiten heredar variables o métodos de la nueva clase. Lo mismo se aplica a las subsubclases, y así sucesivamente.
- Eliminación de una clase: la eliminación de una clase en bases de datos orientadas a objetos requiere varias operaciones. Las variables y los métodos

que heredan las subclases se deben volver a examinar. Cualquier cambio en una subclase puede que necesite propagarse a las subclases, y así sucesivamente. Las instancias de las clases eliminadas deben hacerse instancias de otra clase, generalmente un padre de la clase eliminada.

- Modificación de una definición de clase: puede definirse una nueva variable o método o puede eliminarse una definición de una variable o un método. Como en los casos anteriores, la definición de las subclases puede verse afectada.
- Reposición de clases en la jerarquía : toda reestructuración de la jerarquía de clases tiene consecuencias en la herencia de las antiguas y nuevas subclases de la clase reposicionada.

Una modificación de esquema orientado a objetos que se complica más es el hecho de que las instancias de las clases modificadas deben modificarse para ajustarse a la nueva definición.

1.5.2 Un modelo conceptual unificado:

El mundo no es una colección de tablas, una base de datos orientada a objetos soporta tanto la capacidad de los objetos de referirse en forma directa a los demás como la habilidad computacional de los lenguajes orientados a objetos para el procesamiento de objetos.

Las técnicas OO[6] utilizan los mismos modelos conceptuales para el análisis, diseño y construcción. El modelo conceptual de la base de datos OO es igual al del resto del mundo OO. El uso del mismo modelo conceptual para todos los aspectos del desarrollo, además de simplificar al mismo, mejora la comunicación entre usuarios, analistas y programadores, y también reduce errores. El analista debe determinar los objetos y comportamientos de alto nivel. El diseñador lleva esto a los objetos de bajo nivel que heredan el comportamiento y las propiedades de los objetos de mayor nivel. El programador trabaja con objetos temporales y persistentes de manera uniforme. Los objetos persistentes están en la base de datos orientada a objetos. Además los datos pueden estar ligados entre sí, de modo que los métodos de la clase logren un mejor rendimiento a diferencia del modelo relacional en la que cada relación (tabla) es independiente. Los comandos Join (reunión) relacionan los datos en las diversas tablas.

El uso de un modelo conceptual unificado para el análisis, diseño, programación y la base de datos trae como consecuencia:

- Mayor productividad: se evita el trabajo de traducción entre paradigmas.
- Menos errores: los errores aparecen durante la traducción entre los paradigmas.
- Mejor comunicación entre usuarios, analistas e implantadores.
- Mejor calidad.

[6]Utilizaremos este término para representar Orientadas a Objetos.

- Mayor flexibilidad.
- Mayor inventiva.

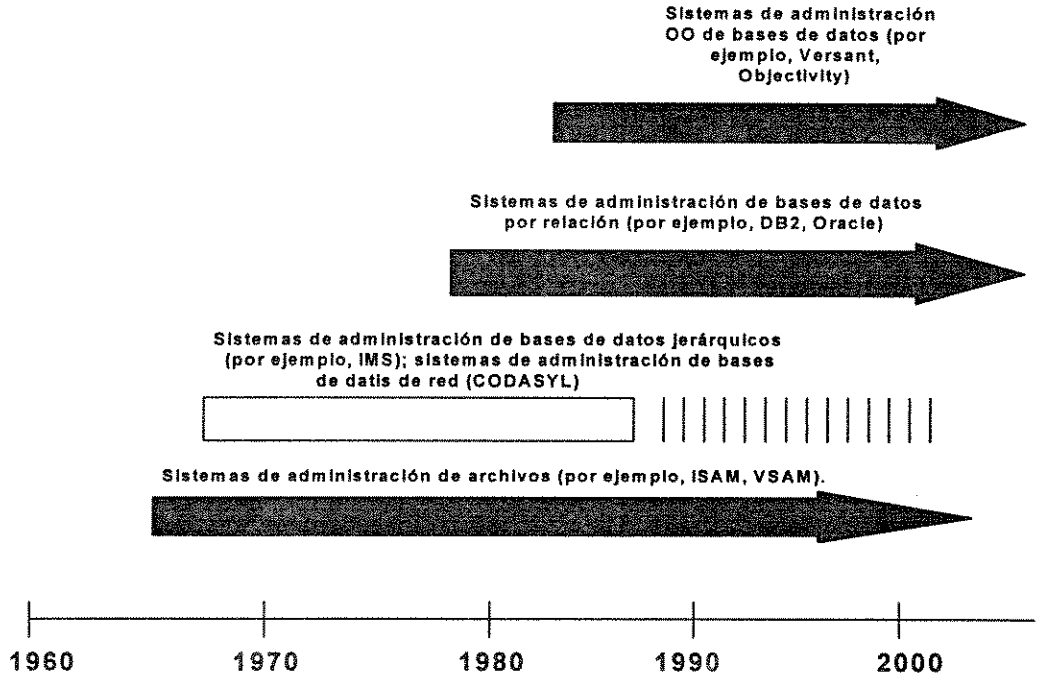


FIGURA 1.1 Cuatro generaciones del manejo de datos.

2. CONCEPTOS GENERALES DE LOS SITEMAS DE GESTION DE BASES DE DATOS ORIENTADAS A OBJETOS

2.1 Introducción

En este capítulo se detallan los términos y conceptos básicos utilizados en los sistemas de bases de datos orientadas a objetos, como los objetos mismos, las clases, métodos, mensajes, jerarquías de clases, etc., con los cuales sería indispensable que el lector se familiarizara. Se ha intentado relacionar todos estos con ideas familiares ligadas a objetos en la vida real.

2.2 ¿Qué son los objetos ?

La idea de un objeto combina estructura de datos estáticos y ajustes dinámicos de procesamiento de comportamiento con la manera en que nosotros inspeccionamos objetos en el mundo real -por lo menos objetos que mueven, crean o manipulan información (Figura 2.1).

Aunque los expertos puedan discernir sobre como debería usarse el término objeto, para nuestros propósitos un objeto es un concepto o cosa única que se distingue de otras cosas o conceptos.

Comúnmente se acostumbra a percibir un objeto como algo tangible, alguna cosa como una silla, una mesa o un libro; generalizando un poco más podemos percibir un objeto de una manera conceptual también. Mas específicamente un objeto es **una** persona, **un** lugar o algo **único** que existe. Un objeto puede ser físico o conceptual; por ejemplo la persona Carol, la ciudad de Guatemala, la computadora personal con número serial 1234567 son objetos físicos. Cada uno de estos objetos son una entidad única, no una categoría general de entidades. Citamos a Carol no a personas, a la ciudad de Guatemala no a ciudades, a la computadora personal con número serial 1234567 no a las computadoras personales en general.

La idea es que un objeto es una única entidad o noción. Cada objeto es único e individual. Un objeto puede ser relacionado con otros objetos o bien puede generar otros objetos.

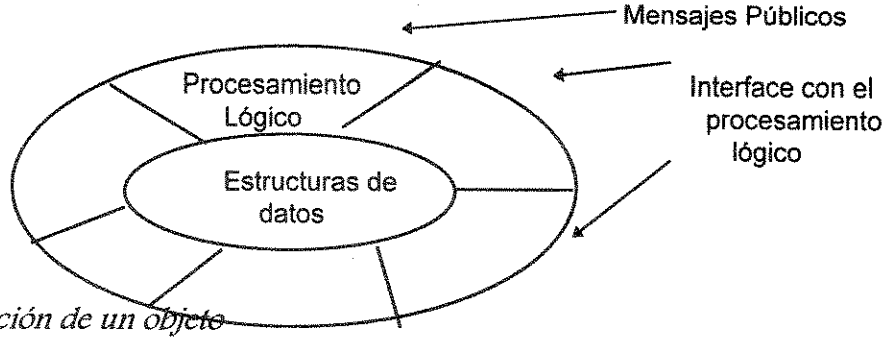


Figura 2.1 Representación de un objeto

Los objetos no son necesariamente físicos; como ejemplo de objetos conceptuales tenemos una estatura de 1.64 mts., una temperatura de 24 grados o la cantidad de Q.1,000,000.00.

Generalmente, tenemos la idea de que el tipo de un objeto es fijo y que su valor puede cambiar, pero la realidad es que un objeto puede ser de varios tipos al mismo tiempo; ver ejemplo Ileana y Roberto en la siguiente sección.

2.3 Clases

Objetos y clases son palabras con significados establecidos hace cientos de años.

2.3.1 Definición

Orden en que se colocan las personas o cosas según su naturaleza, su condición, importancia, etc.

Entonces, una clase es un conjunto de objetos, los cuales tienen exactamente la misma estructura interna y, por tanto, los mismos atributos y los mismos métodos.

Una clase describe un conjunto de objetos que tienen:

- características similares de datos,
- igual comportamiento,
- las relaciones con otros objetos y
- significado en el mundo real

Identificar y estudiar objetos individuales (objeto por objeto) y sus relaciones entre si es útil, pero muy tedioso, y no resulta muy conveniente en la mayoría de los sistemas. Los analistas necesitan de técnicas que los ayuden a organizar y administrar un número grande de objetos y sus relaciones y así poder usarlos en sistemas complejos. Para poder resolver esta complejidad necesitamos de algún método de abstracción que agrupe un cuerpo grande en hechos menores, en unidades mas pequeñas.

La identificación de conjuntos de objetos que pertenecen por alguna razón lógica a un grupo se llama clasificación. En OSA[7] un grupo de objetos que pertenecen juntos a un conjunto por alguna razón lógica se llaman clase de un objeto. El ORM[8] induce a los analistas a organizar los objetos en clases. Cada clase tiene un nombre que es genérico y denota a cualquier miembro de la clase. Así, en un ORM, una clase cuyo nombre es X designa una clasificación de objetos; y

[7] Análisis de sistemas orientado a objetos (siglas en inglés)
[8] Modelo Objeto-Relacional (siglas en inglés)

cada objeto de la clase es un X; lo que deja ver que los objetos de una clase son semejantes.

Por ejemplo, la clase personas de una compañía cualquiera pueden ser todas las personas que sean de interés a la compañía. Los Objetos Carol, Ileana y Roberto pueden ser los miembros de la clase, ya que cada uno es una persona.

Los objetos que se agrupan juntos en una clase generalmente tienen otras características comunes, y no son simplemente miembros de la clase. El concepto de clase es similar a la técnica de clasificación biológica, donde los biólogos agrupan cosas que viven juntas y que comparten características comunes. Un analista puede agrupar cualquier conjunto de objetos en una clase por cualquier razón, pero dicha clasificación debería tener un buen sentido. De otra manera, el analista tendría dificultad para que los objetos pudieran comunicarse en la clasificación, así como un biólogo tendría el mismo problema si no hace la agrupación adecuada.

Para ejemplificar este término se podría pensar que Ileana y Roberto son empleados de una compañía, entonces ellos formarían parte de la clase Empleado (ya que interesan a la compañía como empleados); pero si estas mismas personas fueran en un momento dado clientes, entonces formarían parte de la clase Cliente. La misma analogía podemos hacer con cualquier otro objeto.

El nombre para una clase debería elegirse tal que pueda aplicarse al objeto único (individual). Por ejemplo, Ileana es una Persona, no unas Personas. El bus No. 25 de (de una línea cualquiera) es un vehículo no unos Vehículos. Los nombres de las clases deberían ser bien específicos. A veces, podemos necesitar más de un nombre para una clase.

Como los objetos cambian a través del tiempo, las clases a las que pertenecen también pueden cambiar.

En resumen, en términos formales, una clase describe el comportamiento de un objeto. Las clases pueden ser abstractas o concretas. Las clases abstractas se usan para describir datos y el comportamiento de una familia de clases. Estas no son destinadas para ser usadas como una instancia de los objetos. Las clases abstractas a veces se llaman clases formales. Las clases concretas pueden usarse como instancias de objetos.

2.4 Clases y tipos

Los sistemas orientados a objetos pueden clasificarse en dos categorías principales: los que manejan la noción de clase y los que manejan la noción de tipo; existe tendencia a utilizar indistintamente el concepto de clase y de tipo, pero son dos conceptos diferentes.

Se ha definido anteriormente lo que es una clase; ahora un tipo modela los rasgos comunes a un conjunto de objetos que tienen las mismas características. En los lenguajes de programación, los tipos aseguran la corrección de los programas mediante la compilación, mediante el sistema de verificación de tipos estos pueden ser controlados eficientemente. En general, en un sistema basado en tipos, los tipos no son objetos propiamente dichos y no pueden ser modificados dinámicamente.

La diferencia radica principalmente en que la clase define la implementación de un conjunto de objetos, mientras que un tipo describe como pueden utilizarse tales objetos. Un tipo puede estar implementado por varias clases. Inversamente, una clase puede implementar varios tipos.

Estrictamente hablando, se debería referirse al tipo de un objeto (tal persona) como un tipo de objeto y una ocurrencia especifica de que tipo de objeto (tal como Roberto Rojas) como una instancia de objeto. En adelante, un tipo de objeto es como una clase y una ocurrencia de objeto es una instancia de objeto.

2.5 Instancia

Una instancia es una de varias copias idénticas de un objeto. Más específicamente es una ocurrencia de un objeto. Todas las instancias de una misma clase tienen la misma estructura y comportamiento del objeto al que representan, en otras palabras es un mecanismo por el cual se pueden re-usar las definiciones. Los modelos de datos orientados a objetos usan el concepto de clase como la base para la instanciación. En este sentido una clase es un objeto que actúa como una plantilla. En particular especifica:

- Una estructura, es decir, el conjunto de atributos de las instancias.
- Un conjunto de operaciones.
- Un conjunto de métodos que implementan a las operaciones.

2.6 Atributos

2.6.1 Definición

Cualquier propiedad, calidad, o característica que puede ser asociada a una persona o cosa.

Más formalmente un atributo es algún dato (o datos) por medio del cual cada objeto en una clase tiene su propio valor. Los atributos son los "datos estructurados". Y como en la vida real, representa características de un objeto.

Una persona puede tener un nombre, dirección, número de teléfono, edad y muchos otros atributos. El o ella pueden desempeñar ciertas tareas como parte de su trabajo.

Así una persona específica es una instancia de la clase persona, (Figura 2.2). Los objetos pueden identificarse en una definición de un sistema, (en este caso, un negocio) por sustantivos en oraciones. Las tareas desempeñadas por objetos (las operaciones o comportamiento de objeto) pueden identificarse como verbos en estas mismas oraciones.

CLASE PERSONA **INSTANCIAS DE LA CLASE PERSONA**

Figura 2.2 Clases y objetos

Los objetos que son los miembros de una clase comparten algún comportamiento y tipos de atributos. Las instancias individuales de objeto pueden distinguirse desde otras instancias por diferencias en los valores de los atributos y por asociaciones con otras clases e instancias de objeto. Las instancias de objeto que son los miembros de la misma clase comparten un significado común del mundo real, además de sus atributos compartidos y relaciones.

2.7 Ocultamiento y/o encapsulamiento

Encapsulamiento es la capacidad de un objeto para combinar comportamiento y atributos. La ocultación de la información es sin duda una buena idea en muchos casos: es evidente que los conceptos gemelos de ocultar los detalles irrelevantes a la vista del usuario (con lo cual es posible alterar esos detalles), cuando sea necesario, en una forma controlada, y un acceso disciplinado a objetos solo a través de una interfaz pública, son claramente apropiados para muchos usuarios y muchas aplicaciones.

La idea de encapsulamiento no es utilizada por primera vez en los sistemas orientados a objetos, ni en los lenguajes orientados a objetos, sino que obedece a un proceso evolutivo que comenzó con los lenguajes imperativos; existen por lo menos dos razones válidas por las cuales es necesario encapsular la información: la primera es la necesidad de hacer una distinción clara entre la especificación y la implementación de una operación, es decir diferenciar entre la interfaz y la implementación; la interfaz es la descripción de un conjunto de operaciones, las cuales se pueden invocar para el objeto y constituyen su parte visible; la implementación contiene los datos, esto es, la representación del estado del objeto y los métodos que nos proporcionan. La segunda es la necesidad de contar con modularidad, ya que esta es útil como herramienta para manejar la autorización y la protección de un objeto.

El encapsulamiento proporciona una forma lógica de independencia de los datos y significa que la implementación de los objetos se puede modificar sin que haya que cambiar las aplicaciones que los utilizan (Figura 2.3). Un encapsulamiento real se obtiene sólo cuando las operaciones son visibles y el resto del objeto está oculto.

El uso del sistema se puede simplificar significativamente si no se obliga a un estricto encapsulamiento. El manejo de las consultas es una de las situaciones en las que no usar encapsulamiento es conveniente ya que por lo regular se expresan en términos de predicados sobre los valores de los atributos; por tanto, algunos de los sistemas de bases de datos orientados a objetos permiten el acceso directo a los atributos, las cuales leen y modifican estos atributos. Estas operaciones se ofrecen como parte del sistema y son implementadas a bajo nivel de manera muy eficiente; esto evita que el usuario tenga que implementar una cantidad considerable de métodos cuyo único propósito sería el de leer y escribir los atributos de un objeto; y por otra parte aumenta la eficiencia de las aplicaciones, debido a que el acceso directo a los atributos ya está implementado por el sistema.

Pero, si se considera que permitiendo lo anterior se podría llegar a modificar la definición de los atributos de un objeto, esto de hecho constituye un problema. Los sistemas de bases de datos orientados a objetos ofrecen distintas soluciones: Algunos sistemas como Vbase incluyen métodos definidos por el sistema para la lectura y escritura de los atributos de un objeto. Otros como O_2, permiten al usuario especificar qué atributos y métodos son visibles en la interfaz del objeto y puede invocarse desde afuera. Dichos atributos y métodos se denominan públicos. Inversamente, los atributos y métodos no visibles desde afuera se denominan privados. Por último, en otros sistemas como Orion, se puede acceder directamente a todos los atributos lo mismo para leer que para escribir y todos los métodos pueden ser invocados. En Orion lo que si existe son mecanismos de autorización que pueden utilizarse para proteger el acceso a ciertos atributos y evitar la ejecución de ciertos métodos.

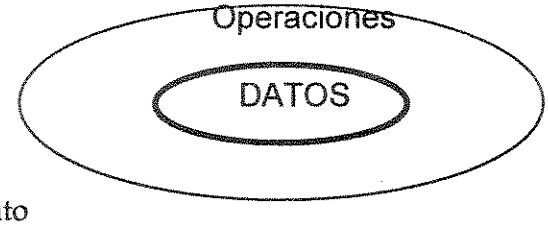


Figura 2.3 Encapsulamiento

El encapsulamiento ayuda a administrar modelos complejos del sistema ocultando detalles en niveles más inferiores de abstracción. Los objetos pueden ser simplificados para definir una jerarquía de estructuras de datos. Dos tipos comunes de jerarquía de datos son la generalización ("tipo de" o estructura de clases) y agregación ("parte de" o estructura de objeto) que se estudian en la siguiente sección.

2.8 Jerarquía y herencia

Los objetos y su organización pueden proveer el beneficio extra de reusabilidad de datos y código. La programación de procedimiento implementado en un objeto puede usarse en otro objeto mediante un sistema de clases, jerarquía y herencia. Necesitamos que combine los procesos y estructuras de datos para formar un solo conjunto unificado de estructuras y asociación de procesos (ver Figura 2.4). Un negocio podría tener un consumidor y un cliente comercial. El consumidor heredaría los comportamientos de un objeto de cliente: la capacidad para mantener un nombre, numero de cuenta y dirección. El objeto de consumidor también tendría características especificas de un cliente consumidor a diferencia de los de un cliente comercial.

El objeto consumidor no necesita que tenga reescritas todas las características para todos, simplemente se hereda el código y estructuras de datos para todos clientes.

Si otra clase de clientes se necesita, se puede construir una nueva subclase para heredar todos los datos y los procedimientos de la clase común de cliente. Los datos y procedimientos que se heredan ya están diseñados, implementados y probados. Esto hace mucho mas fáciles los cambios en las aplicaciones orientadas a objetos. Un sistema suele ser mas productivo cuando no se necesita volver a escribir los objetos y las funciones asociadas en ellos.

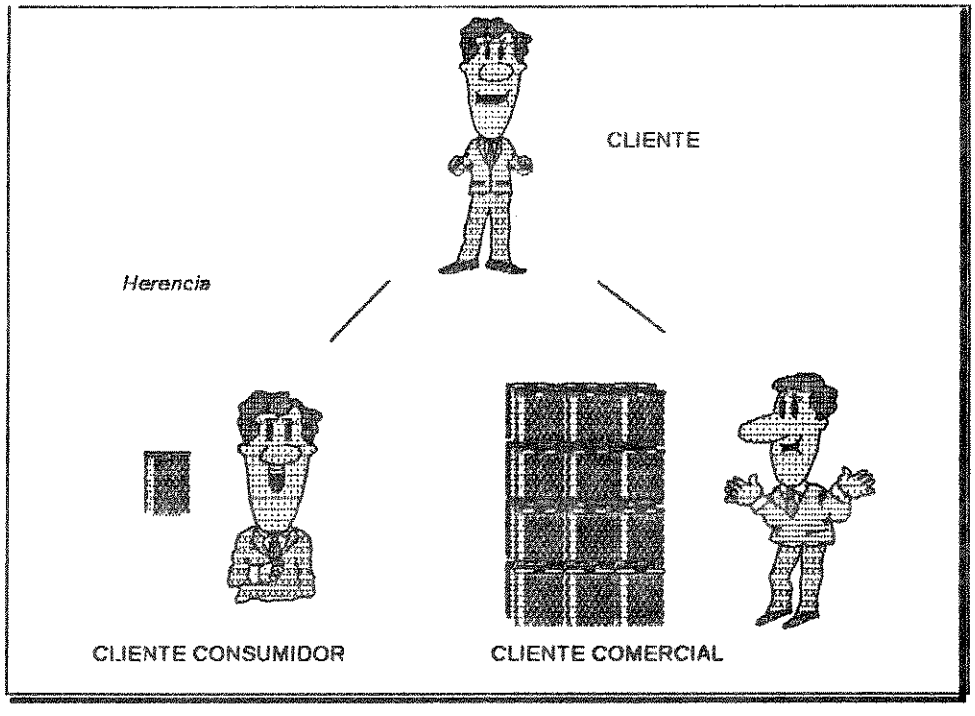


Figura 2.4 Una jerarquía de objetos unificada.

2.8.1 La generalización

Este tipo de jerarquía define una relación entre clases (ver Figura 2.5). Una clase comparte las estructuras y comportamientos definidos en una clase (herencia única) o en más de una clase (herencia múltiple). Unas subclases heredan las características de una o más superclases y pueden refinar la definición de las superclases. Una definición general de una superclase vehículo de motor puede ser refinada por definiciones más especificas de las subclases: automóvil, camión y autobús.

2.8.2 Jerarquía de clases

La jerarquía de clases más o menos funciona así: Si la clase B es una subclase de A, todo caso de B será en forma automática un caso de A, pero lo opuesto no se cumple.

Sea B una subclase de A. En ese caso B heredará de manera automática ciertas características de A. Existen dos tipos de herencia, estructural y de comportamiento. Las que explicamos de la siguiente manera:

2.8.3 Herencia estructural

B hereda en forma automática todos los atributos de A. Por ejemplo, todo PROGRAMADOR tiene un SALARIO, porque todo EMPLEADO tiene un SALARIO. Sin embargo, B podría tener sus propios atributos adicionales, no compartidas por A; así, los PROGRAMADOR(es) podrían tener LENGUAJE(s) (de programación), cosa que no tienen los EMPLEADOs en general.

2.8.4 Herencia de Comportamiento

B hereda en forma automática todos los métodos aplicables a A. Por ejemplo, los PROGRAMADORes pueden ser promovidos, porque todo EMPLEADO puede ser promovido. Sin embargo, B podría tener sus propios métodos adicionales no aplicables a A: los PROGRAMADORes podrían tener un método AGREGAR_LENGUAJE (correspondiente a agregar un nuevo lenguaje de programación al conjunto de lenguajes que consideramos domina el programador en cuestión), en tanto que los EMPLEADOs en general no poseen tal método.

También podría ser posible ampliar (o aun modificar por completo) la definición de un método aplicable a la clase A de modo que realice operaciones adicionales (o totalmente distintas) cuando se aplique a un objeto de la clase B. Esta posibilidad se conoce como sobrecarga.

2.8.5 Herencia múltiple

En la mayoría de los casos, una organización jerárquica de clases es adecuada para describir aplicaciones. En tales casos, todas las superclases de una clase son antepasados de otra en la jerarquía.

El concepto herencia múltiple se refiere a la capacidad de las clases para heredar variables y métodos de múltiples superclases. Cuando se emplea la herencia múltiple es posible que se dé ambigüedad en el caso en que pueda heredarse la misma variable o método de más de una superclase; aunque es conveniente decir que no todos los casos de herencia múltiple conducen a ambigüedad.

2.8.6 Jerarquía de agregación

Las relaciones de agregación tratan la jerarquía "parte de" (ver Figura 2.6). Un automóvil se construye de estos subobjetos: el motor, cuerpo y chasis. El motor a la vez se compone de combustible, sistemas de ignición y el enfriamiento. Cada uno de estos sistemas se compone de subsistemas y/o partes discretas. Las abstracciones de alto nivel se generalizan mientras los niveles más inferiores se especializan. Un automóvil se modela en un nivel más alto de abstracción que cualquier componente de su clase.

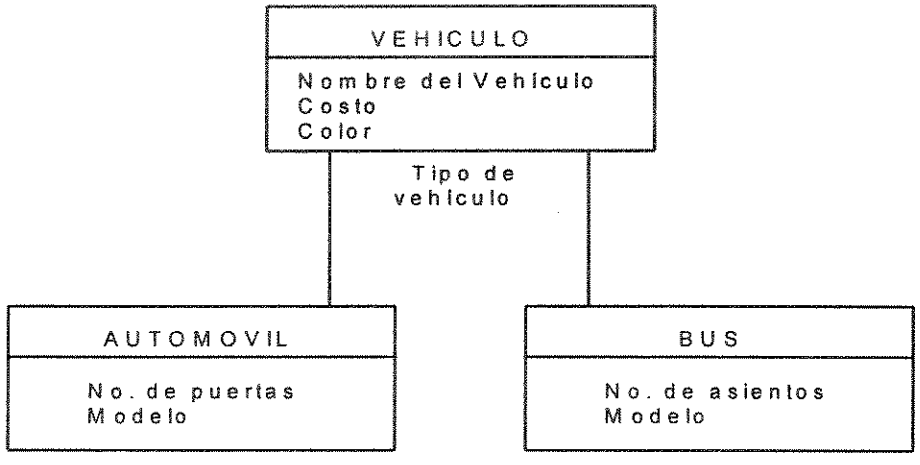


Figura 2.5 Una Jerarquía de generalización.

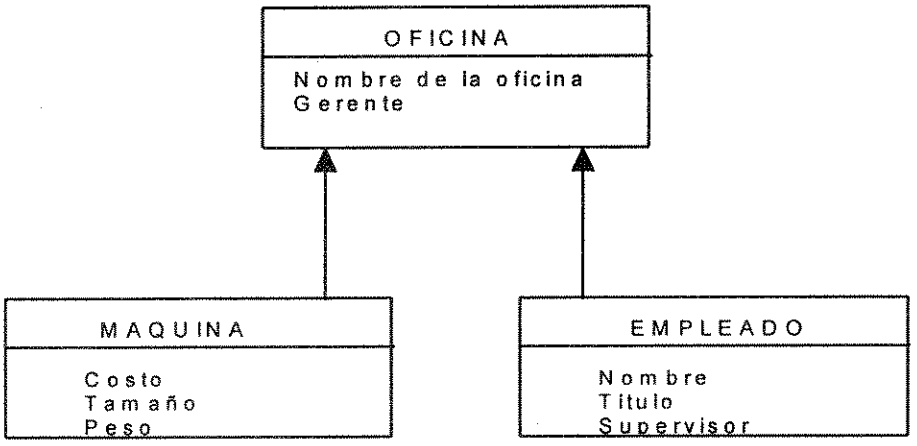


Figura 2.6 Una jerarquía de agregación.

Para las aplicaciones de manejo de datos que requieren una manipulación eficiente de grandes cantidades de datos y que por lo tanto necesitan asignar espacio a las estructuras auxiliares apropiadas, el sistema debe conocer el tipo de los posibles valores de un atributo; en este punto es de vital importancia tener presente que un atributo está asociado con un dominio que especifica las clases de los posibles objetos que pueden ser asignados como valores al atributo.

El hecho de que un atributo de una instancia C tenga una clase C1 como dominio implica que cada instancia C asume como valor del atributo una instancia C1, o de otra subclase de ésta. Se establece una relación de agregación entre las dos clases. Una relación de agregación de la clase C con la clase C1 especifica que C se define en términos de C1. Puesto que C1 a su vez se define en términos de otras clases, el conjunto de las clases del sistema se organiza en una jerarquía de agregación. Sin embargo, es importante también hacer notar que esta no es una jerarquía en el sentido estricto de la palabra, ya que las clases pueden estar definidas recursivamente

2.9 Mensajes

Los objetos se comunican entre ellos pasándose mensajes unos a otros. Es decir los objetos se relacionan mutuamente, se podría pensar en muchas formas acerca de las relaciones entre objetos y las clases, el tipo más común es precisamente el de relaciones.

2.10 Métodos

Un método, es un trozo de código para implementar cada mensaje. Un método devuelve un valor como respuesta al mensaje. En otras palabras, los métodos son "el algoritmo" es decir comúnmente son funciones relacionadas con el objeto. Definen como se comporta un objeto mediante el paso de mensajes. Podríamos encontrar aquí cierta similitud con el concepto de funciones o

procedimientos a nivel de programación, la diferencia principal entre un método y una función tradicional es que los métodos operan sobre datos de un objeto.

La única forma de operar sobre un objeto es mediante los operadores (métodos) definidos para la clase de ese objeto. En un sistema de gestión de bases de datos orientadas a objetos, los objetos se manipulan con métodos. En general un método se compone de encabezado y cuerpo. El encabezado especifica el nombre del método, los nombres y clases de sus argumentos y la clase del resultado (si se retorna uno); por lo tanto el encabezado es la especificación de la operación implementada por el método. El cuerpo representa la implementación del método y consiste en un conjunto de instrucciones expresadas en algún lenguaje de programación.

2.11 Relaciones

Una relación establece una conexión lógica entre objetos. Las relaciones asocian un objeto con otro, si hacemos una analogía con gramática viene siendo parecido al verbo y sustantivo en una oración. Por ejemplo, en la oración "Ileana es el administrador de Carol" la frase "es el administrador de" especifica una relación entre los objetos Ileana y Carol.

Es importante resaltar que aunque las relaciones sean únicas, los nombres de las mismas no necesariamente lo son; es decir una relación puede tener varios nombres.

En resumen; para aplicar un método dado a un objeto determinado es necesario enviar un mensaje a ese objeto. Al recibir el mensaje, el objeto ejecuta la función (o sea, el método) solicitado por el mensaje, y en seguida devuelve un resultado al remitente; dicho de otra forma, los métodos se comunican con los objetos enviándoles mensajes a estos.

2.12 Polimorfismo

La palabra polimorfismo se compone de dos términos griegos; "poly", significa "mucho" y el sufijo "morf" significa "forma". Polimorfismo es la habilidad de dos o más clases para responder al mismo mensaje. Por ejemplo, un soporte de impresora en un ambiente orientado a objetos puede definir una interface general de impresión, y determinar los caminos de comunicación con la interface (Ver figura 2.7). Un mensaje de impresión puede ser respondido por un archivo de texto plano, por un archivo bit-map un archivo gráfico, o un archivo reporte formateado. Cada tipo es implementado en un objeto con un método de impresión que responda a un mensaje de impresión, con un conjunto de parámetros opcionales únicos para cada tipo de archivo o con un conjunto de herramientas asociadas (spooler, cola de impresión, formateador de archivo, impresora, plotter, etc.) usados para la operación de impresión.

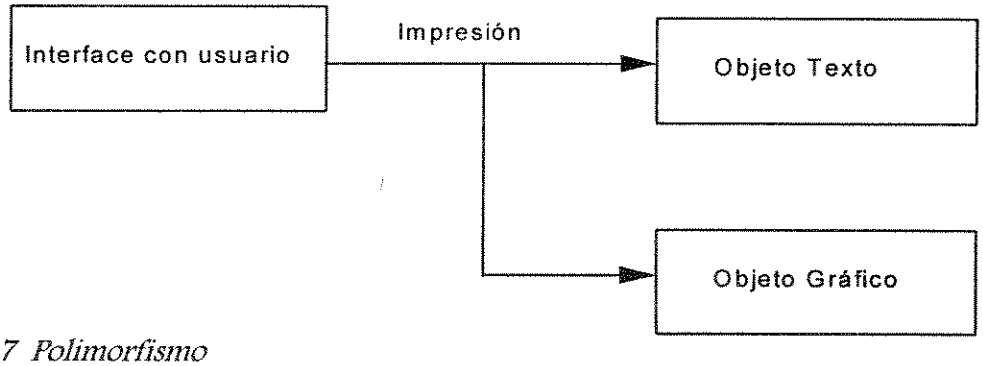


Figura 2.7 Polimorfismo

En la programación orientada a objetos se refiere a métodos y funciones que tienen el mismo nombre, pero tienen diferente comportamiento. El concepto de polimorfismo es ortogonal[9] con respecto al concepto de herencia.

2.13 Identidad de objetos

Los objetos en una base de datos orientada a objetos, normalmente corresponden a una entidad en la empresa que está modelando la base de datos (a una entidad del mundo real). Una entidad conserva su identidad aún cuando algunas de sus propiedades cambien con el tiempo. Igualmente, un objeto conserva su identidad aún cuando algunos o todos los valores de las variables o las definiciones de los métodos cambien con el tiempo.

Cada uno de estos objetos, tiene asociado un estado y un comportamiento. El estado se representa por los valores de los atributos del objeto. El comportamiento se representa por los métodos que actúan sobre el estado del objeto cuando se invocan las operaciones correspondientes. Cada objeto se identifica por medio de un IDO (identificador de objeto); utilizando los IDOs, los objetos pueden compartir otros objetos y formar jerarquías de objetos.

La identidad de objeto es una noción más fuerte que la que se encuentra normalmente en los lenguajes de programación o en los modelos de datos que no están basados en la orientación a objetos. A continuación se ilustran algunas formas de identidad:

- Valor: se utiliza un valor de dato por identidad.
- Nombre: se utiliza un nombre facilitado por el usuario por identidad. Esta es la forma de identidad que normalmente se usa para las variables en los procedimientos. A cada variable se le da un nombre que identifica de manera única a la variable sin importar el valor que contenga.

[9] Generalmente se entiende por ortogonal lo que es perpendicular; de aquí en adelante, se entiende como: dos conceptos distintos.

- Incorporación: una noción de identidad es incorporar en el modelo de datos el lenguaje de programación, y no se requiere que el usuario proporcione ningún identificador. Esta es la forma de identidad que se usa en los sistemas orientados a objetos.

Vale la pena hacer notar la relación de igualdad e identidad entre objetos; la identidad de un objeto introduce al menos dos nociones diferentes de igualdad entre objetos:

- Igualdad por identidad: comúnmente denotada por "=", dice que dos objetos son idénticos si son el mismo objeto, es decir si tienen el mismo identificador.
- Igualdad por valor: comúnmente denotada por "==", dice que dos objetos son iguales si los valores de sus atributos son recursivamente iguales.
  Por tanto, dos objetos idénticos son a su vez iguales, pero dos objetos iguales no siempre son idénticos. Por ejemplo: considérese el objeto archivo, la igualdad por valor se da a nivel de instancias es decir dos archivos pertenecientes a la clase objeto archivo, pero que son dos archivos diferentes.
- Igualdad superficial: Dos objetos son superficialmente iguales, aunque no sean idénticos, si todos sus atributos comparten los mismos valores y las mismas referencias.

Un tema relacionado con el tipo de identidad es la permanencia de identidad. A continuación se describen tres formas de permanencia:

- Intraprograma: la identidad permanece solamente durante la ejecución de un único programa o consulta.
- Interprograma: la identidad permanece de una ejecución de programa a otra.
- Persistente: la identidad permanece no sólo entre las ejecuciones del programa sino también entre las reorganizaciones estructurales de los datos.

2.14 Persistencia

La persistencia de las instancias de las clases se refiere a las modalidades bajo las cuales los objetos se hacen persistentes (se insertan en la base de datos) y bajo las que eventualmente pueden eliminarse (quitarse de la base datos). Existen dos enfoques básicos:

- La persistencia es una característica implícita de todas las instancias de las clases. La creación del objeto automáticamente implica su persistencia porque la creación de una instancia tiene el efecto de insertar el objeto en la base de datos.
- La creación de la instancia no tiene el efecto de insertar la instancia en la base de datos. Una instancia se crea durante la ejecución de un programa y se elimina al terminar éste (como se había definido en la forma de permanencia intraprograma), a menos que se haya hecho persistente. Un mecanismo para

hacer a una instancia persistente es asociar un nombre dado por el usuario a la instancia, o insertar la instancia en una colección de objetos persistentes.

Sin embargo puede adoptarse un enfoque intermedio entre ambos extremos; podemos dividir las clases en persistentes y temporales. Todas las instancias de las clases persistentes se crean automáticamente como persistentes, mientras que no sucede lo mismo con las clases temporales[10].

2.15 Objetos complejos

Llamaremos objetos complejos a aquellos en los cuales los valores de sus atributos pueden ser otros objetos, primitivos o no primitivos; la desventaja principal de utilizar valores complejos[11] es que implica que el modelo de datos se haga conceptualmente más complicado.

Los objetos complejos se forman aplicando constructores a objetos más simples; estos constructores deben ser ortogonales, es decir que puedan aplicarse a cualquier objeto.

[10] Variante adoptada por el lenguaje E (Carey et al., 1988).

[11] Cuando los atributos están formados por otros objectos.

3. BASES DE DATOS ORIENTADAS A OBJETOS

3.1 Introducción

Tal vez alguna vez nos hemos hecho la pregunta: ¿Cómo es que se puede almacenar y hacer grandes recuperaciones de objetos complejos que han complicado interelaciones, pero no pueden representarse en forma de una simple fila-columna?

Básicamente, en la actualidad hay tres tecnologías principales de DBMS disponibles que son la relacional (RDBMS en inglés), la orientada a objetos (ODBMS en inglés), y la Objeto-Relacional (ORDBMS en inglés). La decisión de qué tecnología usar depende de factores tal como si una organización necesita que tal aplicación sea muy rápida o bien pueda esperar, de la similitud de las funciones y los aspectos de un DBMS particular con los requerimientos de una aplicación, de la complejidad y el tamaño real de los datos, y del nivel de confort de una organización con la nueva y prometedora tecnología.

3.2 Manejadores de bases de datos orientados a objetos

Los ODBMS soportan datos complejos y relaciones complicadas usando conceptos de la orientación a objetos tal como encapsulamiento, herencia, y tipos de datos definidos. Se pueden almacenar objetos reusables en librerías de objetos. Los lenguajes tales como C++ o Smalltalk están disponibles para el desarrollo de aplicaciones.

Esta tecnología es relativamente nueva y por lo tanto menos madura que la tecnología RDBMS. Carece de características para un fuerte manejo de datos, y requiere que uno aprenda el paradigma de objetos. Además no hay ningún lenguaje de consultas como SQL.

En ausencia de un modelo de datos común y de fundamentos formales similares a los del modelo relacional, conceptos de la orientación a objetos se han agrupado para formar el modelo central o nuclear para los modelos orientados a objetos, dado que las bases de datos necesitan tener un modelo propio de datos; así que el concepto de bases de datos orientadas a objetos está inmerso en la terminología orientada a objetos y en los propios sistemas de gestión de bases de datos[12].

A continuación, se describen varias características distintivas de los sistemas orientados a objetos. Es importante aclarar que no hay un modelo común a utilizar

[12]Conjunto de datos persistentes y aplicaciones utilizadas para accederlos y actualizarlos.

como punto de referencia, ningún fundamento formal para los conceptos y tampoco, ningún estándar para los modelos orientados a objetos como si lo hay en el modelo relacional.

3.3 Características de las bases de datos orientadas a objetos

Aunque no hay una norma claramente aceptada con respecto a la funcionalidad que ofrezcan los sistemas de bases de datos orientados a objetos, los aspectos generales que ofrecen la mayoría son:

- Las características de los objetos
- Las relaciones.
- Los datos activos.
- La herencia.
- La modificación de esquema.
- Interface de consultas y programación.
- Concurrencia, versiones, y recuperación.

3.3.1 Características de los objetos

En un sistema de base de datos orientado a objetos, los objetos son los elementos de almacenamiento, recuperación, y modificación. Los objetos se usan para agrupar datos que representan entidades del mundo real. Cada objeto tiene un único identificador para referenciarlo. Los objetos son de tipos diferentes, y estos tipos son determinados por la clase a la que ellos pertenecen. Los atributos de los objetos pueden consistir de:

1. Tipo simple (entero, real, etc.)
2. Referencia.
3. Colecciones.
4. Los datos activos (procedimientos).
5. Tipos definidos por el usuario.
6. Los grandes objetos binarios (grandes archivos binarios o archivos de texto).

Para ilustrar como definir una base de datos orientada a objetos para los tipos de atributos que se enumeraron arriba, considérense los siguiente ejemplos:

```
make-class  capítulo
{
        Título      :   string;
        Número      :   integer;
        doc         :   document;
```

```
        Texto       :    Binary;
}
make-class  document
{
        Título      :   string;
        Cap         :   list[capitulo];
}
```

En este ejemplo, el objeto de tipo capítulo consiste de un título para el capítulo, y del número de capítulo. Estos son ambos atributos simples. El atributo doc es un atributo de referencia al documento al que el capítulo pertenece. El atributo texto es un objeto binario grande que es el texto real del capítulo. Un objeto de tipo document consiste de un título para el documento y un atributo de colección. El atributo de colección es un conjunto de referencias a capítulos que constituyen el documento.

3.3.1.1 Atributos simples

Los atributos simples consisten de enteros, números reales, cadenas de caracteres (strings), y otros valores simples que son predefinidos por el sistema.

3.3.1.2 Atributos de referencia

Los atributos de referencia se usan para representar relaciones entre objetos. Estos son implementados típicamente como identificadores únicos de objetos en la base de datos. Estas referencias son parecidas a los apuntadores en lenguajes de programación tales como C; sin embargo, desde que ellos son mantenidos en el sistema de base de datos los mismos no pueden corromperse. Por ejemplo, cuando un objeto se borra, todas las referencias a él se invalidan.

3.3.1.3 Atributos de colección

Los atributos de colección se usan para representar conjuntos, o arreglos de valores. Estos pueden ser de longitud fija o variable. Los atributos de colección pueden consistir de atributos simples o de atributos de referencia. El uso de conjuntos o arreglos es determinado por la aplicación; los arreglos utilizan una ordenación explícita de los valores, y los elementos de un conjunto pueden estar en cualquier orden. Por ejemplo, el objeto document podría contener un arreglo de atributos capítulo en vez de un conjunto de atributos de capítulo, así tendría más sentido que cada elemento del arreglo capítulo correspondiera al orden de números de los capítulos.

3.3.1.4 Atributos activos

Las bases de datos orientadas a objetos permiten que los atributos se asocien con procedimientos. El usuario especifica si un campo es activo o no en la definición del esquema de la base de datos. Si el campo se define para ser activo, el usuario debe definir el procedimiento para el campo. Como resultado, cuando el campo de atributo se recupera (o es llamado), un procedimiento que corresponde al campo es ejecutado por el sistema. La sintaxis para acceder atributos activos es idéntica a la de los campos atributos no activos. De hecho, un usuario ni siquiera puede saber si el campo se derivó o se almacenó explícitamente.

3.3.1.5 Atributos definidos por el usuario

Algunas bases de datos orientadas a objetos permiten a los usuarios definir sus propios tipos, y las funciones comunes de entrada y salida asociadas con esos tipos. Estos sistemas permiten al usuario imponer su propia semántica sobre los datos mediante la definición de las funciones de entrada y salida. Por ejemplo, el usuario podría definir un tipo circulo con funciones asociadas de entrada y salida como se indica a continuación:

```
define type circulo
{
        x           : real;
        y           : real;
        radio       : real;
        circle_in   : function;
        circle_out  : function;
}


define function circle_in
{
        lenguaje    : c;
        return      : circulo;
        arg         : string;
        as          : /usr/circulo.o
}


define function circle_out
{
```

```
        lenguaje    : c;
        return      : string;
        arg         : círculo;
        as          : /usr/circulo.o
}
```

En este ejemplo, círculo es un tipo definido por el usuario que consta de tres números reales, x & y: posición del centro, y radio: el radio del círculo. El tipo círculo también contiene dos funciones definidas por el usuario, una para entrada y otra para salida. La función de entrada "circle_in", que se implementa en C, devuelve un tipo círculo, tiene un parámetro de entrada que es un string, y usa el objeto archivo círculo.o. Similarmente, la función de salida "circle_out" es una función en C, devolviendo un tipo string, que tiene un parámetro de entrada de tipo círculo, y usa el objeto archivo círculo.o. La función "circle_in" puede ser usada por el sistema de base de datos cuando un objeto círculo es salvado (o grabado). Similarmente, "circle_out" se usaría cuando un objeto círculo se recobra desde la base de datos. Típicamente, los sistemas de base de datos orientados a objetos requieren que el usuario defina tipos y funciones a fin de proveer el almacenamiento de tipos no estándar. Además de los tipos definidos por el usuario, el sistema también permite la definición de operadores tales como: menor que, mayor que, e igual, para operar sobre los tipos de datos de los usuarios. Por ejemplo, para determinar cuando dos círculos son iguales puede definirse un operador como se indica a continuación:

```
define function es_un_círculo_igual
{
        lenguaje    : c;
        return      : boolean;
        arg         : círculo, círculo;
        as          : /usr/círculo.o
}


define operador =
{
        arg         : círculo, círculo;
        procedure   : es_un_círculo_igual;
}
```

En este ejemplo, la primera definición especifica que la función de usuario es_un_círculo_igual estará implementada en lenguaje C, la función devuelve un tipo boolean, los parámetros por argumento son los tipos círculo a ser comparados, y el código del objeto se ubica en el archivo /usr/círculo.o. La segunda definición informa al sistema de base de datos para asociar el operador "=" con la función definida por el usuario "es_un_círculo igual", y lo registra en la base de datos.

3.3.1.6 Grandes objetos binarios

Los sistemas de base de datos orientados a objetos permiten un almacenamiento uniforme de objetos binarios grandes (BLOBs[13]) como objetos regulares. Ejemplos de datos que consideran a los BLOBs son: imágenes scaneadas, grandes archivos ASCII o archivos binarios, y cualquier tipo de datos que es relativamente grande comparado a los atributos definidos en el sistema. Hay varias formas en que los BLOBs pueden traerse desde la base de datos. Algunos sistemas, tal como ObjectStore, presentarán los BLOBs enteros para obtener un apuntador, o el nombre de archivo. Otros sistemas, tal como EXODUS, provee una interface de mecanismo de apoyo para que los BLOBs puedan ser recobrados en una o más operaciones. El mecanismo de apoyo tiene la forma:

- SetPos (índice): se coloca en la posición de lectura. La posición default estará al principio del BLOB.
- Read (size, data): usando la posición por el llamado a Setpos lee los bytes de tamaño size dentro del bloque de datos.
- Write (size, data): desde la posición actual indicada por el llamado a Setpos escribe los bytes size desde el bloque de datos.
- Delete (size): desde la posición actual indicada por el llamado a Setpos borra los bytes de tamaño size.

3.3.2 Relaciones

Hay dos tipos de relaciones en los sistemas de bases de datos orientados a objetos. Estas son relaciones binarias y no binarias. Ambos tipos de relaciones se representan usando atributos de referencia y arreglos de atributos de referencia. En una relación binaria la representación es directa. Por ejemplo, considérese un esquema de base de datos para un sistema de localización de documentos. La representación de una relación entre los objetos capítulo y document se especificarían como se indica a continuación:

```
document
{
        Título      : string;
        Revisión    : string;
        Caps        : array[capítulos];
```

[13]binary large objects (BLOBs) en inglés, y como se referenciarán de aquí en adelante.

```
}

capítulo
{
        Título      : string;
        Número      : integer;
        Doc         : document;
        Text        : binary;
}
```

En este ejemplo, cada instancia del objeto de tipo document tiene un título, una fecha de revisión, y una lista de capítulos a los cuales hace referencia. Las instancias del objeto de tipo capítulo contienen un título de capítulo, el número de capítulo, y una referencia al objeto document. Cuando un objeto capítulo se crea, con una referencia particular a document, el sistema automáticamente actualizará el arreglo Caps en el objeto document para el capítulo correspondiente. De manera similar, cuando un capítulo se borra, la referencia desde document a Caps es removida. Las relaciones no binarias son representadas por un tercer objeto que representa la relación. La estrategia involucra la descomposición de la relación en un conjunto de relaciones binarias entre los tipos de objetos participantes y el nuevo tipo de objeto intermedio. Estas relaciones son representadas por relaciones binarias. Varios enfoques se usan para representar relaciones; estos son: colocación e indexamiento. En el caso de colocación, los objetos que comparten las relaciones, se almacenan físicamente uno junto al otro. En el caso de indexamiento, los objetos que comparten la relación se almacenan separadamente. De hecho, los objetos relacionados pueden almacenarse en segmentos diferentes de la base de datos. En este tipo de representación, el sistema podría utilizar un árbol B o una técnica de indexamiento Hash para proporcionar un mapeo entre los dos objetos relacionados.

3.3.3 Procedimientos

En las bases de datos orientadas a objetos, como en las demás, se utilizan los datos activos para permitir la asociación de objetos o sus atributos con procedimientos. En los sistemas de bases de datos orientados a objetos se utilizan tres tipos de datos activos:

- Datos derivados: estos son atributos de objeto que se definen proceduralmente. Con este tipo de atributo, cuando se haga un intento para recobrar o recuperar el objeto, un procedimiento es ejecutado.

- Reglas o triggers: las reglas definidas para un objeto están basadas en un predicado, al cumplirlas ocasionan en el sistema de base de datos ejecutar uno o más conjuntos de procedimientos. Estos procedimientos, típicamente se usan

para realizar actualizaciones en la base de datos. Por ejemplo, una regla podría definirse para una base de datos de localización de documentos tal que cuando un documento se borre desde la base de datos, todos los capítulos asociados con el documento también se borren. Las reglas son importantes en los sistemas de bases de datos orientados a objetos porque ellas proveen de un mecanismo por medio del cual el conocimiento de éstas puede ser codificado en la base de datos.

- Agentes: los agentes son parecidos a las reglas porque ellos se ejecutan con base en un predicado verdadero. Sin embargo, la ejecución de la acción es un proceso separado. Cuando un predicado llega a ser cierto, el proceso puede desempeñar otras acciones que son independientes de las actualizaciones de la base de datos. Por ejemplo, un proceso puede verificar el número de usuarios que están registrados en la red.

3.3.4 Herencia

El concepto de herencia o "subtipos" en lenguajes de programación orientados a objetos es conservado en los sistemas de base de datos orientados a objetos. El concepto es tan importante que la mayoría de los sistemas de bases de datos orientados a objetos (p. ej. ObjectStore, O2, POSTGRES) permiten a los objetos heredar propiedades tales como atributos, y procedimientos desde otros objetos. Este mecanismo de subtipos permitido por los sistemas de base de datos orientados a objetos tienen algunas implicaciones útiles para los objetos:

- Especificación: los subtipos y los atributos que ellos heredan pueden definirse durante el tiempo de ejecución en base a un predicado. Por ejemplo, un subtipo del objeto documento puede definirse como un objeto de "historia corta" o un objeto en base al número de páginas que contiene el documento.

- Clasificación: los subtipos pueden usarse para clasificar objetos. Por ejemplo, un documento podría clasificarse como una historia corta, un libro, o una novela, aún cuando los atributos y los métodos dentro de los objetos sean idénticos.

- Especialización: los subtipos pueden incorporar métodos o atributos adicionales aparte de los que heredaron desde el supertipo. Por ejemplo, una clase, SourceProgram podría ser un subtipo de document, sin embargo el subtipo SourceProgram puede contener un atributo lenguaje de programación que document no contiene. El atributo lenguaje de programación especifica el tipo de lenguaje usado para codificar el programa fuente. Esta distinción es necesaria a fin de representar el tipo SourceProgram. Una implicación importante de la especialización, con el respecto a agregar métodos adicionales, es que esa herencia puede usarse para reducir la cantidad de implementación y promover la reusabilidad por la especialización. Por ejemplo, los objetos pueden agregar métodos o atributos adicionales para aumentar los atributos existentes adquiridos

a través de la herencia. Como resultado sólo lo nuevo necesita ser codificado, y el resto se hereda del padre de la clase.

- Implementación: los subtipos pueden tener diferentes implementaciones para cada uno de sus métodos, aún cuando el nombre del atributo heredado es el mismo. En este caso, el método que se redefine se ejecuta en vez del que se heredó. Como un ejemplo, los subtipos de SourceProgram tal como programas en C o programas en Ada, pueden tener diferentes conjuntos de instrucciones para compilar su código original.

3.3.5 Modificaciones al esquema

La mayoría de los sistemas de bases de datos orientadas a objetos incorporan mecanismos para la modificación de tipos de objetos o del esquema. El grado de las modificaciones de esquema permitidas por el sistema de bases de datos orientadas a objetos varían de sistema a sistema dependiendo de que tan complejo sea el sistema. Esto es porque la complejidad y la cantidad de implementaciones requeridas aumentan tanto como se permitan opciones adicionales. Un sistema razonablemente completo puede permitir las modificaciones de esquema siguientes:

1. Cambio de atributos y métodos:
   - (a) Cambios a los atributos
     - i. Agregar un atributo.
     - ii. Eliminar un atributo.
     - iii. Cambiar el nombre de un atributo.
     - iv. Cambiar el tipo de un atributo.
     - v. Heredar diferente definición de atributos mediante un cambio de supertipos o subtipos.
   - (b) Cambios a los métodos
     - i. Agregar un método.
     - ii. Eliminar un método.
     - iii. Cambiar el nombre de un método.
     - iv. Cambiar el método.
     - v. Heredar métodos diferentes mediante un cambio de supertipo y subtipo.

2. Cambios en la estructura jerárquica adquirida mediante la herencia
   - (a) Agregar un nuevo supertipo o subtipo.
   - (b) Eliminar un supertipo o subtipo.
   - (c) Cambiar la ordenación entre tipos en sistemas que permiten herencia múltiple.

3. Cambios al tipo de objeto
   (a) Agregar un tipo.
   (b) Eliminar un tipo.
   (c) Cambiar el nombre de un tipo.

La mayoría de los sistemas de base de datos orientados a objetos (O2, ONTOS, ORION, etc.) permiten cambios a los nombres de atributos y métodos, añadir y quitar atributos y métodos, y cambios a los tipos. El grado de dificultad en la implementación varía con el tipo de cambios permitidos. Los cambios que involucran modificaciones de los nombres de atributos o métodos son directos. Los cambios que involucran agregar y quitar atributos y relaciones son más difíciles de implementar. La complicación adicional viene cuando la base de datos se puebla con instancias. Por ejemplo, agregando un atributo a un objeto que ya contiene instancias, involucra destinar espacio adicional para el atributo. Esto puede forzar a las otras instancias a modificar su estructura.

Los cambios en las propiedades de los objetos pueden iniciarse inmediatamente en el tiempo del pedido del cambio (GemStone) o luego cuando el objeto es manipulado. El primer enfoque requiere actualizar el esquema, y todas las instancias que puedan existir. Esto involucra copiar todas las instancias viejas dentro de la nueva instancia e inicializar los campos que no tienen valores a NULL. Al segundo enfoque, podríamos llamarle un enfoque perezoso, el cual actualiza las instancias si ellas se usan. Esto involucra guardar la versión actual y compararla con el objeto que se está recuperando. Si un cambio se detecta, el objeto se actualiza a la representación correcta antes de que se escriba a disco. El primer enfoque es más eficiente, a largo plazo (o sea a lo largo de la ejecución), desde el momento de que no requiere la comprobación continua de versiones de objeto, y las instancias dentro de la base de datos están siempre en un estado consistente.

3.3.6 Consultas e interface de programación

Los sistemas de base de datos orientados a objetos, típicamente proveen un lenguaje de programación y un lenguaje de consultas para permitir acceso al contenido de la base de datos. La interface de lenguaje de programación comienza con un lenguaje de programación orientado a objetos e incorpora funcionalidad adicional para proveer a la base de datos características tal como almacenamiento de objetos a largo plazo, control de concurrencia, etc. Con la tradicional y extendida base de datos relacional, el lenguaje de consultas se incorpora dentro de el programa de aplicación, el ambiente de ejecución para el lenguaje de programación y el lenguaje de consultas son diferentes. Como resultado, el programador es forzado a usar dos lenguajes diferentes para pasar datos entre el programa de aplicación y el sistema de base de datos.

Algunos sistemas de base de datos orientados a objetos no proveen lenguaje de consultas, otros proveen una versión más débil que el lenguaje de consultas relacional, y algunos proveen un lenguaje de consultas que es más poderoso que el lenguaje de consultas relacional. Los enfoques tomados por los sistemas de base de datos orientados a objetos para los lenguajes de consultas pueden ser resumidos por el tipo de resultados que ellos producen:

- Ningunos resultados: esto se da en sistemas que no tienen capacidades de lenguaje de consultas. El acceso al contenido de la base de datos debe hacerse a través de la interface del lenguaje de programación.

- Resultados de Colección: esto se da en sistemas que proveen un limitado acceso a los datos. El acceso al contenido de la base de datos es permitido por un subconjunto de la base de datos en forma de grupos o colección de objetos.

- Resultados de Relación: esto se da en sistemas que proveen un acceso completo a los datos. El acceso al contenido de la base de datos es permitido para cualquier dato dentro de la base de datos en forma de tablas como en los sistemas tradicionales de base de datos relacionales.

- Resultados libres: esto se da en sistemas que proveen acceso completo a los datos, sin embargo los datos pueden estar en cualquier forma. Por ejemplo, un atributo, un objeto, o una colección de objetos.

Aunque el soporte del lenguaje de programación provisto por los sistemas de base de datos orientados a objetos en su mayoría es completo, la interface de lenguaje de consultas es todavía un proceso en desarrollo. Hasta ahora, ningún consenso claro se ha alcanzado para establecer un lenguaje estándar de consultas para los sistemas de base de datos orientados a objetos.

Esta es la principal incompatibilidad entre las estructuras usadas por las bases de datos orientadas a objetos, las bases de datos relacionales, y el lenguaje de consultas. En los sistemas orientados a objetos, las consultas involucran navegar a través de apuntadores para encontrar los objetos deseados. Con sistemas relaciones de base de datos, las consultas se desarrollan sobre tablas. Varios enfoques se han tomado para resolver estos conflictos. Un enfoque se refiere a extender el lenguaje de consultas para permitir consultar objetos así como también tablas. El segundo enfoque consiste en combinar consultas relacionales y consultas orientadas a objetos para desarrollar las consultas sobre objetos y relaciones en paralelo, y para combinar el resultado. Esta incompatibilidad también ha presentado problemas con respecto a la optimización de consultas. Con el sistema relacional de base de datos, la optimización es posible explotando las propiedades algebraicas de las operaciones relacionales y las estructuras físicas de las tablas. Esto hace posible optimizar las consultas antes de ejecutarlas. Sin embargo con los sistemas de base de datos orientados a objetos no hay un conjunto

estándar de operaciones definido sobre todo el conjunto de datos porque cada objeto puede tener su propio conjunto separado de métodos.

Como resultado, las técnicas de optimización desarrolladas para los sistemas relaciones de base de datos no pueden usarse para perfeccionar consultas sobre un objeto. Sugerencias que se han hecho para resolver el problema de optimización permiten al optimizador (programador de aplicaciones) verificar los contenidos de la definición de clases para la información necesaria. La implicación de esto es que el paradigma de la orientación a objetos de encapsulamiento es violado, desde que el acceso a datos privados se permite.

3.3.7 Concurrencia, recuperación y versiones

Los requerimientos de concurrencia para los sistemas de base de datos orientados a objetos son satisfechos parcialmente por transacciones a corto plazo tradicionales. En situaciones donde las necesidades de acceso son cortas (p. ej. aplicaciones de negocios), las transacciones cortas tradicionales solas, pueden ser adecuadas. Sin embargo, para aplicaciones más complejas, en donde la duración de las transacciones puede ser de horas o días, se requiere de nuevas técnicas que resuelvan el problema de concurrencia. Estas técnicas nuevas son importantes porque también se usan para recuperación. Las soluciones propuestas para el problema de concurrencia son:

1. Conversational transaction: Este mecanismo permite a los objetos y sus datos conexos estar desocupados sobre un período largo de tiempo.
2. Versiones: Este mecanismo permite que múltiples versiones del mismo objeto puedan ser creadas para que varios individuos puedan tener acceso a los datos.

3.3.7.1 Conversational transactions

Las conversational transactions permiten a los usuarios desocupar el conjunto de datos sobre los que quieren trabajar. Este tipo de transacciones requiere que los datos sean bloqueados (cerradura o comúnmente llamado candado "lock") por largos períodos de tiempo (horas o días). A continuación se discuten varios enfoques que han sido tomados para soportar transacciones más largas:

- Soft Locks. Los soft locks pueden usarse en conversational transactions para permitir a otros usuarios desocupar datos. Cuando los datos se desocupan por medio de una conversational transaction, el sistema permite otro proceso para "romper" la cerradura y entonces notifica a la aplicación que la cerradura se rompe.

- Optimistic Locking. Las técnicas tradicionales para bloquear datos son en base a la suposición de que las transacciones son cortas, y la potencialidad para conflictos de bloqueo es alta. Por lo tanto, la primera vez que un bloque de datos es accedido, el sistema trata de adquirir una cerradura para ellos. Algunos sistemas de base de datos orientados a objetos tal como GemStone y Objetivity/DB tratan de resolver este problema permitiendo al usuario especificar si desea usar la cerradura optimista o pesimista. Con la cerradura optimista, el sistema no trata de adquirir las cerraduras que necesita hasta el commit transaction. Con la cerradura pesimista, el enfoque tradicional de adquirir una cerradura sobre el primer acceso es usado. Esta opción permite al usuario decidir sobre el riesgo de un conflicto de cerradura que puede resultar al abortar la transacción.
- Archivos Bitácora o de registro. Desde que los sistemas de base de datos orientados a objetos tienen necesidades de transacciones más largas, las tradicionales técnicas de implementación de registros bitácora o log de datos también ha sido necesario revisarlas. Se da el caso de que en varios días de transacciones es posible que no haya habido commit, pero se pueden haber escrito al archivo de registro. Como resultado, el archivo bitácora de registro puede llegar a ser muy grande y desbordarse. Los diversos puntos tomados para resolver este problema son:

  - Archivo de registro múltiple: en este enfoque, el sistema mantiene dos archivos de registro. Cuando el archivo principal de registro es demasiado grande, las entradas viejas que están "uncommitted" (que no se han cometido) se escriben de segundas, condensadas en el archivo de registro. Los contenidos del archivo viejo de registro son removidos y se vuelve a usar como el archivo actual de registro.
  - "Shadow paging": este enfoque usado por GemStone es una variación sobre la técnica convencional de sombra de pagina. En este enfoque, cuando una página de base de datos se modifique, el sistema destina una página nueva de base de datos. Las páginas actualizadas no son escritas al disco hasta que la transacción haya sido cometida "commit" o cuando la memoria principal cache está llena. En el "commit transaction", todas las páginas que se actualizaron se escriben al disco y la tabla de segmento se actualiza para indicar las nuevas páginas de la base de datos. La página de cabecera de la tabla de segmento se escribe como una actualización atómica única de disco, ocasionando que todas las páginas nuevas sean visibles.
  - Time Stamping: un enfoque usado en POSTGRES, este crea una nueva versión del objeto con el tiempo sellado (time stamp), cuando un objeto existente se actualice. El objeto nuevo puede actualizarse varias veces o hasta que la transacción sea cometida "commit transaction". Una vez se comete, el objeto nuevo se escribe al disco. Esta solución elimina la necesidad de mantener un archivo de registro.

3.3.7.2 Versiones

La capacidad de manejo de versiones es importante en los sistemas de base de datos orientados a objetos porque proporcionan una alternativa a las transacciones a largo plazo para el control de concurrencia. Orion, ObjectStore y Objetivity/DB proveen la funcionalidad básica para versiones por medio de implementación de mecanismos para crear y destruir versiones de objeto. Los objetos pueden hacerse versionables en una de dos maneras. En la primera manera, el objeto puede hacerse versionable (es decir que pueda tener muchas versiones) cuando se declara, para que cuando un tipo de objeto se modifique, el sistema mantenga múltiples versiones de él. En la segunda manera, el objeto puede hacerse versionable cuando se crea. Típicamente los sistemas de base de datos orientados a objetos permitirán una u otra forma, o ambos tipos de capacidad de versiones. Por ejemplo, Objetivity/DB permite ambos tipos de versionismo.

El uso de versionismo para resolver el problema de concurrencia introduce un problema para objetos que contienen referencias a otros objetos. Por ejemplo, cuando un objeto versionado se crea o destruye, el sistema debe determinar maneras para manejar las referencias dentro del objeto. Los sistemas de base de datos orientados a objetos han tomado dos enfoques a este problema. El siguiente párrafo describe el concepto de configuración que es usado por los sistemas de base de datos orientados a objetos para resolver este dilema.

Una configuración es un conjunto de versiones de los objetos en una base de datos que son mutuamente consistentes. Por ejemplo, si las actualizaciones son hechas a 20 módulos de software para crear una versión nueva de un sistema operativo, entonces estas 20 nuevas versiones de objeto, más las versiones existentes de todos los otros módulos en el sistema operativo, representan una configuración.

Los dos enfoques de configuraciones en los sistemas de base de datos orientados a objetos son:

1. El sistema puede proveer un mecanismo para manejar la gestión de configuraciones que es explícito. En este enfoque, el usuario se responsabiliza especificando el tratamiento de referencias. Por ejemplo, en Objetivity/DB, el sistema permite al usuario especificar una de las tres opciones para referencias a versiones de objetos:

   - Movimiento: cuando una nueva versión del objeto se crea, todas las referencias dentro del objeto viejo se mueven (o el conjunto indicando) al objeto nuevo, y las referencias dentro de el objeto viejo son un conjunto a NULL.

   - Reducción: cuando una nueva versión del objeto se crea, todas las referencias dentro de el son un conjunto a NULL.

- Copia: cuando una nueva versión del objeto se crea, todas las referencias dentro de él se duplican, para que el objeto nuevo y el objeto viejo tengan las mismas referencias en ellos.

2. El sistema puede proveer un mecanismo para manejar la gestión de configuraciones que es automático, con alguna intervención del usuario. En este enfoque, el sistema maneja configuraciones automáticamente con alguna política de opciones especificada por el usuario. Por ejemplo, en ObjectStore, el sistema define una configuración actual. Los usuarios pueden cambiar la configuración actual o especificar una nueva. Los cambios que se hacen al objeto dentro de la configuración, afectan a la configuración actual.

3.4 EL MODELO DE OBJETOS

3.4.1 Descripción

En esta sección se define el Modelo de Objetos apoyado por ODMG[14]. El Modelo de Objetos es importante porque especifica los tipos de semántica que pueden definirse explícitamente en un ODBMS. Por otro lado, la semántica del Modelo de Objetos determina las características de los objetos, cómo los objetos pueden relacionarse uno con el otro, y cómo los objetos pueden nombrarse e identificarse.

La sintaxis del lenguaje de definición de objetos (ODL) que es usado para aplicaciones específicas de modelos de objetos, está presente para todas las construcciones explicadas en éste capítulo. También se ha usado en este capítulo para definir las operaciones sobre varios objetos.

El Modelo de Objetos especifica las construcciones que son soportadas por un ODBMS:

- Las primitivas del modelo básico son el objeto y la literal. Cada objeto tiene un único identificador. Una literal no tiene identificador.
- El estado de un objeto es definido por los valores que lleva para un conjunto de propiedades. Estas propiedades pueden ser atributos del objeto mismo o

[14] Dócil gestión de sistemas de base de datos orientados a objetos.

relaciones entre un objeto y uno o más objetos. Típicamente, los valores de las propiedades de un objeto pueden cambiar a través del tiempo.

- El comportamiento de un objeto es definido por el conjunto de operaciones que pueden ejecutarse sobre o por el objeto. Por ejemplo, un objeto Documento incluye una operación de formato.
- Los objetos y las literales pueden ser categorizados por sus tipos. Todos los elementos de un tipo determinado tienen una gama común de estados (p. ej., el mismo conjunto de propiedades) y un comportamiento común (p. ej., el mismo conjunto de operaciones definidas). Un objeto a veces se refiere como a una instancia de su tipo.
- Una base de datos almacena objetos, permitiendo ser compartida por múltiples usuarios y aplicaciones. Una base de datos está basada en un esquema que se define en el ODL y contiene las instancias de los tipos definidos por su esquema.

El Modelo de Objetos ODMG especifica qué se entiende por objetos, literales, tipos, operaciones, propiedades, atributos, relaciones, y más. Un desarrollador de aplicaciones usa la construcción del Modelo de Objetos ODMG para construir el modelo de objetos para su aplicación. El modelo de objetos de la aplicación, especifica tipos particulares, tal como Documento, Autor, Editor, y Capítulo, y las operaciones y propiedades de cada uno de estos tipos. El modelo de objetos de la aplicación es el esquema (lógico) de la base de datos.

Análogo al Modelo de Objetos ODMG para las bases de datos de objetos (u orientadas a objetos) está el modelo relacional para bases de datos relaciones, personificadas en SQL. El modelo relacional es la definición fundamental de la funcionalidad de los sistemas de gestión de bases de datos relacionales. El Modelo de Objetos ODMG es la definición fundamental de la funcionalidad de los ODBMSs. El modelo de objetos ODMG incluye semántica significativamente más rica que la del modelo relacional, para declarar relaciones y operaciones explícitamente.

3.4.2 Tipos y clases; interfaces e implementación

Hay dos aspectos en la definición de un tipo. Un tipo tiene una especificación de interface y una o más especificaciones de implementaciones. La interface define las características externas del tipo de los objetos. Estas características externas son los aspectos de los objetos que son visibles a los usuarios de los objetos. Estas son las operaciones que pueden invocarse sobre los objetos y las variables de estado cuyos valores pueden ser accedidos. En contraste, la implementación de un tipo define los aspectos internos del tipo de los objetos.

Una implementación de un tipo consiste de una *representación* y un conjunto de *métodos*. La representación es una estructura de datos. Los métodos son el cuerpo de procedimientos. Hay un método para cada una de las operaciones definidas en la especificación de la interface del tipo. Un método cumple el comportamiento externamente visible de su operación asociada. Un método podría

leer o modificar la representación del estado de un objeto o invocar operaciones definidas sobre otros objetos. También podrían ser estructuras de datos y métodos en una implementación que no tiene operaciones directas o variables de estado en la interface de tipo. Lo interno de una implementación no es visible por los usuarios de los objetos.

La diferencia entre interface e implementación es importante. La separación entre estas dos es la manera en que el Modelo de Objetos refleja la encapsulación. El ODL se usa para especificar las interfaces de tipos en los modelos de objeto de la aplicación. C++ y Smalltalk, por ejemplo, construyen las implementaciones de estos tipos.

Un tipo puede tener más de una especificación de implementación, aunque una única implementación puede usarse en cualquier programa particular. Por ejemplo, un tipo podría tener una implementación en C++ y otra implementación en Smalltalk. O un tipo podría tener una implementación en C++ para una arquitectura de máquina, y otra implementación en C++ para una diferente arquitectura de máquina.

Separar las interfaces de las implementaciones es un paso positivo hacia el acceso multi-lingual a objetos de un único tipo y compartimiento a través de ambientes de computación heterogéneos.

A veces libremente nos referimos a una interface por sí misma como a un tipo, y a unas implementaciones de un tipo como una clase. Un objeto es una instancia de una clase. Una especificación de clase, entonces, se usa para implementar todas las instancias del tipo. Por ejemplo, una especificación de clase en C++ es usada por ambos, el compilador de C++ y un ODBMS para crear instancias (objetos) del tipo.

3.4.2.1 Subtipos y herencia

Como muchos modelos de objetos, el Modelo de Objetos ODMG incluye herencia basada en relaciones de tipo-subtipo. Estas relaciones usualmente se representan en diagramas; cada nodo es un tipo y cada arco conecta un tipo, llamado supertipo, y otro tipo, llamado subtipo. La relación tipo/subtipo a veces también es llamada una relación de generalización-especialización. El supertipo es el tipo más general; el subtipo es el más especializado.

```
interface Employee {...};
interface Professor : Employee {...};
interface Associate_Professor : Professor {...};
```

Por ejemplo, Associate_Professor es un subtipo de Professor; Professor es un subtipo de Employee. Una instancia de un subtipo es también lógicamente una instancia del supertipo. Así la instancia Associate_Professor también es una instancia de Professor. Esto es, Associate_Professor es un caso especial de Professor.

Un tipo de objeto más específico es el tipo que describe el comportamiento y propiedades de la instancia. Por ejemplo, el tipo más específico para un objeto Associate_Professor es la interface Associate_Professor; este objeto también lleva información desde las interfaces Professor y Employee; básicamente el comportamiento de subtipos en el modelo de objetos ODMG es como se explicó en la sección de herencia.

3.4.3 Objetos

3.4.3.1 Nombres de objetos

Además de ser asignado un identificador de objeto por el ODBMS, a un objeto puede dársele uno o más nombres que tienen significado para el programador o para el usuario final. El ODBMS provee una función que se usa para mapear desde un nombre de objeto a un identificador de objeto. La aplicación puede referirse a conveniencia, a un objeto por su nombre; el ODBMS aplica la función de mapeo para determinar el identificador de objeto que ubica al objeto deseado. ODMG espera los nombres que comúnmente son usados por las aplicaciones para referirse a la "raíz" de los objetos, que dan puntos de entrada en las bases de datos.

Los nombres de objeto son como los nombres de variables globales en lenguajes de programación. Pero no son iguales que las llaves. Una llave está compuesta de propiedades especificadas en una interface de tipo de objeto. Un nombre de objeto, por el contrario, no es definido en una interface de tipo y no corresponde a valores de propiedades del objeto.

El Modelo de Objetos no incluye una noción de espacios jerárquicos de nombre dentro de una base de datos.

3.4.3.2 Vidas de los objetos

La vida de un objeto determina como se administran la memoria y el almacenamiento destinados al objeto. La vida de un objeto se especifica en el tiempo en que el objeto es creado.

Dos vidas son soportadas por el ODMG:

- transitoria

- persistente

A un objeto cuya vida es transitoria se le destina la memoria que es administrada por el lenguaje de programación en el tiempo de ejecución del sistema. A veces un objeto transitorio se declara al principio de un procedimiento y se destina la memoria según la pila creada por el lenguaje de programación en el tiempo de corrida del sistema cuando el procedimiento es invocado. Esa memoria se libera cuando se regresa del procedimiento. Los otros objetos transitorios son alcanzados por un proceso más bien que por la activación de un procedimiento típicamente se destinan o la memoria estática o a la pila por el lenguaje de programación del sistema. Cuando el proceso termina, la memoria es liberada. Un objeto cuya vida es persistente se le destina la memoria y el almacenamiento administrado por el ODBMS en el tiempo de corrida del sistema. Estos objetos continúan existiendo después que termina el proceso o procedimiento que los crea. Los objetos persistentes son referenciados algunas veces como objetos de la base de datos. Los lenguajes de programación pueden refinar las nociones de vidas transitorias en formas consistentes con sus conceptos de vida.

Un aspecto importante de las vidas de objetos es que éstas son independientes de los tipos. Un tipo puede tener algunas instancias que son persistentes y otras que son transitorias. Esta independencia de tipo y de vida es bastante diferente que la del modelo relacional, cualquier tipo conocido por el DBMS por definición tiene solo instancias persistentes, y cualquier tipo no conocido por el DBMS (p. ej., cualquier tipo no definido usando SQL) por definición tiene solo instancias transitorias. Ya que el ODMG soporta independencia de tipo y vida, ambos objetos transitorios y persistentes pueden manipularse usando las mismas operaciones. En el modelo relacional, SQL debe usarse para definir y usar datos persistentes, mientras que el lenguaje de programación se usa para definir y usar datos transitorios.

3.4.3.3 Objetos atómicos

Un tipo atómico de objeto es un tipo definido por el usuario. No hay tipos de objetos atómicos built-in (objetos empotrados) incluidos en el modelo de objetos ODMG. Vea la sección Modelando Estado-Propiedades para obtener información sobre las propiedades y comportamiento que pueden definirse para objetos atómicos.

3.4.3.4 Objetos de colección

En el modelo de objetos ODMG, tipos de objeto que no son atómicos son los de colección. Las instancias de este tipo comprenden distintos elementos, cada uno de los cuales pueden ser una instancia de un tipo atómico, otra colección, o un tipo literal. Los tipos literales se discutirán más adelante. Una característica distintiva

importante de una colección es que todos los elementos de la colección deben ser del mismo tipo. Son o todos el mismo tipo atómico, o todos el mismo tipo de colección, o todos el mismo tipo de literal.

Los tipos de colección soportados por el modelo de objetos ODMG son:

- Set<t>
- Bag<t>
- List<t>
- Array<t>

Cada uno de estos es un tipo generador, parametrizado por el tipo mostrado dentro de los paréntesis angulares. Todos los elementos de un objeto Conjunto (Set) son del mismo tipo t. Todos los elementos de un objeto Lista (List) son del mismo tipo t. En las siguientes interfaces, vamos a usar cualquier tipo ODL para representar estos parámetros, reconociendo que esto puede implicar una heterogeneidad que no es el intento de este modelo de objetos.

Todas las colecciones tienen las operaciones siguientes:

```
interface Collection : Object {
        unsigned long cardinality();
        boolean       is_empty();
        void          insert_element(in any element);    //cualquier elemento
        void          remove_element(in any element);
        boolean       contains_element(in any element);
        Iterator      create_iterator();
}

interface Iterator  :  Object {
        exception     Empty();
        exception     NoMoreElements();
        boolean       not_done();
        boolean       next(out any next_obj);
        void          advance()  raises(NoMoreElements);
        any           get_element() raises(Empty);
        void          reset();
}
```

Un elemento puede insertarse en una colección o puede eliminarse de una colección. Un objeto de colección puede probarse para la existencia de un elemento particular. Un Iterator, que es un mecanismo para acceder a los elementos de un objeto de colección, puede crearse. Una copia de una colección devuelve un nuevo objeto de colección cuyos elementos son iguales que los elementos del objeto original de colección. Esta es una somera operación de copia.

3.4.3.4.1 Set Objects. Un objeto de conjunto (set) es una colección no ordenada de elementos, con ningunos duplicados permitidos. Set refina las operaciones siguientes heredadas desde su supertipo Collection:

```
interface Set : Collection {
        Set         union_with(in Set other);
        Set         intersection_with(in Set other);
        Set         difference_with(in Set other);
        boolean     is_subset_of(in Set other_set);
        boolean     is_proper_subset_of(in Set other_set);
        boolean     is_superset_of(in Set other_set);
        boolean     is_proper_superset_of(in Set other_set);
}
```

La operación heredada insert_element inserta el objeto pasado como argumento en un objeto existente Set. Si el objeto pasado como argumento es ya un miembro del objeto Set, la operación levanta una excepción. La igualdad es determinada por el operador same_as.

Además de las operaciones heredadas desde su supertipo, la interface Set tiene las operaciones convencionales de conjuntos matemáticos, así como también pruebas lógicas (falso y verdadero) de subconjuntos y superconjunto. Las operaciones union_with, intersection_with, y difference_with cada una retorna como resultado un objeto Set.

3.4.3.4.2 Bag Objects. Un objeto bag es una colección de elementos no ordenados que pueden contener duplicados. La igualdad de elementos es determinada por el elemento operador same_as. Bag refina las siguientes operaciones heredadas de su supertipo colección: insert_element, remove_element.

La operación insert_element inserta dentro del objeto bag el elemento pasado como argumento. Si el elemento es ya un miembro de bag, es insertado otra vez, aumentando su multiplicidad en bag.

Además de las operaciones heredadas de su supertipo Collection, el tipo bag tiene las operaciones siguientes definidas en su interface:

```
interface Bag : Collection {
        Bag         union_with(in Bag other);
        Bag         intersection_with(in Bag other);
        Bag         difference_with(in Bag other);
};
```

La operación union_with es equivalente a crear un nuevo objeto Bag que es una copia del Bag receptor.

3.4.3.4.3 List Objects. El objeto lista es una colección ordenada de elementos. Además de las operaciones que hereda de su supertipo collection, el tipo List tiene estas operaciones específicas:

```
interface List : Collection {
        void    replace_element_at(in unsigned long index, in any element);
        any     remove_element_at(in unsigned long index);
        any     retrieve_element_at(in unsigned long index);
        void    insert_element_after(in any obj, in unsigned long index);
        void    insert_element_before(in any obj, in unsigned long index);
        void    insert_element_first(in any obj);
        void    insert_element_last(in any obj);
        any     remove_first_element();
        any     remove_last_element();
        any     retrieve_first_element();
        any     retrieve_last_element();
        List    concat(in List other);
        void    append(in List other);
};
```

Estas operaciones son posicionales por naturaleza, en referencia al índice determinado o al principio o fin de un objeto de Lista. El indexamiento de un objeto lista comienza en 0 (cero).

3.4.3.4.4 Array Objects. Un objeto Array es una colección con un número fijo de elementos que pueden ser ubicados por la posición. El tipo Array define las operaciones siguientes además de las que hereda de su supertipo Collection:

```
interface Array : Collection {
        void    replace_element_at(in unsigned long index, in any element);
        any     remove_element_at(in unsigned long index);
        any     retrieve_element_at(in unsigned long index);
        void    resize(in unsigned long new_size);
};
```

La operación remove_element_at reemplaza cualquier elemento actual contenido en la celda del objeto Array identificada por la posición, con un valor nulo. No elimina la celda ni cambia el tamaño del Array. Este es lo contrario a la operación remove_element_at definida en el tipo List, el cual cambia el número de elementos en un objeto List. La operación resize permite a un objeto Array cambiar el número máximo de elementos que puede contener.

3.4.4 Literales

Las literales no tienen identificadores de objeto. El modelo de objetos ODMG soporta tres tipos literales:

- Literal atómica
- Literal de colección
- Literal estructurada

3.4.4.1 Literales Atómicas

Números y caracteres son ejemplos de tipos de literales atómicas. Las instancias de estos tipos no son creadas explícitamente por las aplicaciones, pero si existen implícitamente. El modelo de objetos ODMG soporta los siguientes tipos de literales atómicas:

- Long
- Short
- Unsigned long
- Unsigned short
- Float
- Double
- Boolean
- Octet
- Char (caracter)
- String
- Enum (enumerado)

Todos estos tipos también son apoyados por la OMG Interface Definition Language (IDL). Si el lenguaje de programación no contiene un análogo para uno de los tipos del Modelo de Objetos, entonces se debería implementar una librería de clase del tipo que hay que suplir.

Enum es un tipo generador. La declaración de un Enum define un tipo de literal que puede tomar únicamente los valores enumerados en la declaración. Por ejemplo, un atributo género podría ser definido por:

attribute Enum género{masculino, femenino};

Un atributo state_code podría ser definido por:

attribute Enum state_code{AK,AL,AR,AZ,CA, ... WY};

3.4.4.2 Literales de colección

El modelo de objetos ODMG apoya los tipos siguientes de literales de colección:

- Set<t>
- Bag<t>
- List<t>
- Array<t>

Estos tipos generadores son análogos a los de los objetos de colección, pero estas colecciones no tienen identificadores de objetos. Sus elementos, sin embargo, pueden ser de tipos literales o de tipos de objeto.

3.4.4.3 Literales estructuradas

Una literal estructurada (o simplemente estructura), tiene un número fijo de elementos, cada uno de los cuales tiene un nombre de variable y puede contener o un valor literal o un objeto. Un elemento de una estructura es típicamente referido por un nombre de variable, p. ej., address.zip_code = 12345; address.city = "San Francisco". Los tipos de estructura apoyados por el modelo de objetos ODMG incluyen:

- Date
- Interval
- Time
- Timestamp

Estos tipos se definen como en el ANSI SQL por las siguientes interfaces:

3.4.4.3.1 Date (Fecha): La siguiente interface define las operaciones sobre objetos de tipo Date.

```
interface Date : Object {
        typedef     unsigned short      ushor;
        enum        DiasDeLaSemana{Domingo, Lunes, Martes, Miércoles,
                    Jueves, Viernes, Sábado};
        enum        Mes{Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio, Agosto,
                    Septiembre, Octubre, Noviembre, Diciembre};
        // Usado para representar un objeto de tipo Date, por un tipo de valor
        estructura como Valor{ushort, mes, día, año};

        ushort              año();
        ushort              mes();
        ushort              día();
        ushort              día_del_año();
        Mes                 mes_del_año();
        DiasDeLaSemana      día_de_la_semana();
        boolean             es_año_bisiesto();
        boolean             es_igual(in Date a_date);
```

```
        boolean         es_mayor(in Date a_date);
        boolean         es_mayor_o_igual(in Date a_date);
        boolean         es_menor(in Date a_date);
        boolean         es_menor_o_igual(in Date a_date);
        boolean         esta_entre(in Date a_date, in Date b_date);

        Date        siguiente(en DiasDeLaSemana);
        Date        anterior(en DiasDeLaSemana);
        Date        añadir_días(in ushort days);
        Date        restar_días(in ushort days);
        long        restar_fecha(in Date a_date);
};
```

3.4.4.3.2 Interval (Intervalos): Los intervalos representan una duración de tiempo y se usan para desempeñar algunas operaciones de objetos Time y TimeStamp.

```
interface Interval : object {
        typedef     unsigned short     ushort;
        ushort      día();
        ushort      hora();
        ushort      minuto();
        float       segundo();

        //usado para representar un objeto intervalo como un valor tipo estructura
        asValue{ushort día, hora, minuto; float segundo;};

        boolean     es_cero();

        interval    mas(in Interval an_interval);
        interval    menos(in Interval an_interval);
        interval    producto(in short val);
        interval    cociente(in short val);

        boolean     es_igual(in Interval an_interval);
        boolean     es_mayor(in Interval an_interval);
        boolean     es_mayor_o_igual(in Interval an_interval);
        boolean     es_menor(in Interval an_interval);
        boolean     es_menor_o_igual(in Interval an_interval);
};
```

3.4.4.3.3 Time: los times denotan tiempos específicos mundiales.

```
interface Time : Object {
        typedef short       Time_Zone;
```

```
        const       Time_Zone  GMT = 0;
        const       Time_Zone  GMT1 = 1;
        const       Time_Zone  GMT2 = 2;
        const       Time_Zone  GMT3 = 3;
        const       Time_Zone  GMT4 = 4;
        const       Time_Zone  GMT5 = 5;
        const       Time_Zone  GMT6 = 6;
        const       Time_Zone  GMT7 = 7;
        const       Time_Zone  GMT_1= -1;
        const       Time_Zone  GMT_2 = -2;
        const       Time_Zone  GMT_3 = -3;
        const       Time_Zone  GMT_4 = -4;
        const       Time_Zone  GMT_5= -5;
        const       Time_Zone  GMT_6 = -6;
        const       Time_Zone  GMT_7 = -7;
        const       Time_Zone Useste = -5;
        const       Time_Zone UScentral = -6;
        const       Time_Zone USmontaña = -7;
        const       Time_Zone USPacifico = -8;

        ushort      hora();
        ushort      minuto();
        float       segundo();
        short       tz_hora();
        short       tz_minuto();

        boolean     es_igual(in Time a_Time);
        boolean     es_mayor(in Time a_Time);
        boolean     es_mayor_o_igual(in Time a_Time);
        boolean     es_menor(in Time a_Time);
        boolean     es_menor_o_igual(in Time a_Time);
        boolean     esta_entre(in Time a_Time, in Time b_Time);

        Time        añadir_intervalo(in Interval an_interval);
        Time        restar_intervalo(in Interval an_interval);
        Interval    restar_tiempo(in Time a_Time);
};
```

3.4.4.3.4 TimeStamp: los TimeStamps consisten de una fecha y tiempo encapsulados.

```
interface TimeStamp : Object {
        typedef     unsigned short  ushort;
        Date        the_date();
```

```
        Time        the_time();

        ushort      año();
        ushort      mes();
        ushort      día();
        ushort      hora();
        ushort      minuto();
        float       segundo();
        short       tz_hora();
        short       tz_minuto();

        TimeStamp   mas(in Interval an_interval);
        TimeStamp   menos(in Interval an_interval);

        boolean     es_igual(in TimeStamp a_Stamp);
        boolean     es_mayor(in TimeStamp a_Stamp);
        boolean     es_mayor_o_igual(in TimeStamp a_Stamp);
        boolean     es_menor(in TimeStamp a_Stamp);
        boolean     es_menor_o_igual(in TimeStamp a_Stamp);
        boolean     esta_entre(in TimeStamp a_Stamp, in TimeStamp b_Stamp);
};
```

3.4.5 Estructuras definidas por el usuario

Ya que el modelo de objetos es extensible, los desarrolladores pueden definir otros tipos de estructuras según sus necesidades. El Modelo de Objetos incluye un tipo generador "Struct", que se usa para definir estructuras en las aplicaciones. Por ejemplo:

```
attribute Struct Address { String dorm_name, String room_no} dorm_address;
```

Las operaciones definidas por el generador de tipos de estructura incluyen las siguientes:

```
interface Struct {
        unsigned long size();
        void        set_element(in any field, in any value);
        any         get_element(in any field);
        Struct      copy();
        void        delete();
};
```

Las estructuras pueden componerse libremente. El modelo de objetos apoya conjuntos de estructuras, estructuras de conjuntos, arreglos de estructuras, y más.

Esto permite la definición de tipos como Degrees (grados), como una lista cuyos elementos son estructuras que contienen tres campos:

```
struct Degree {
        string      school_name;
        string      degree_type;
        string      degree_year;
};
typedef list<Degree>Degrees;
```

Cada instancia de Degrees podría tener sus elementos clasificados por el valor de degree_year.

Por lo regular en una implementación, el lenguaje de programación provee las estructuras y colecciones. Por ejemplo, Smalltalk incluye sus propias clases de colección, Fecha, Tiempo, y Timestamp.

3.4.6 Modelado de estado - propiedades

Un tipo define un conjunto de propiedades a través de las cuales los usuarios pueden accesar, y en algunos casos directamente manipular, el estado de las instancias del tipo. Dos tipos de propiedades se definen en el modelo de objetos ODMG: *atributo y relación.* Un atributo es de un tipo. Una relación es definida entre dos tipos, cada uno de los cuales debe tener instancias que sean referenciables por identificadores de objetos. Los literales, no pueden participar en las relaciones porque no tienen identificadores de objeto

3.4.6.1 Atributos

Las declaraciones de atributo en una interface definen el estado de abstracto de un tipo. Por ejemplo, el tipo persona podría contener las siguientes declaraciones de atributo:

```
interface Persona {
        attribute short             Edad;
        attribute string            Nombre;
        attribute enum              Genero{masculino, femenino};
        attribute Address           Dirección;
        attribute set<Phone_no>     Teléfonos;
        attribute Department        Dept;
};
```

Una instancia particular de Persona podría tener un valor específico para cada uno de los atributos definidos. El valor para el atributo Dept es el identificador

de objeto de una instancia de Department. El valor de un atributo es siempre o una literal o un identificador de objeto.

Es importante notar que un atributo no es igual que una estructura de datos. Un atributo es abstracto, mientras que una estructura de datos es una representación física. Mientras que comúnmente los atributos son implementados como estructuras de datos, a veces un atributo es implementado como un método. Por ejemplo, el atributo Edad podría muy bien ser implementado como un método que calcula la edad desde un valor almacenado de la persona en date_of_birth y la fecha actual.

En el modelo de objetos ODMG, los atributos no son "primera clase". Esto significa que un atributo por sí mismo no es un objeto y por lo tanto no tiene un identificador de objeto. No es posible definir atributos de atributos o relaciones entre atributos u operaciones de subtipos específicas para atributos.

3.4.6.2 Relaciones

Las relaciones se definen entre tipos. El modelo de objetos ODMG soporta relaciones binarias únicas, p. ej., relaciones entre dos tipos. El modelo no soporta n-ary relaciones, que involucran más de dos tipos. Una relación binaria puede ser uno - a - uno, uno - a - muchos, o muchos - a - muchos, dependiendo de cuántas instancias de cada tipo participan en la relación. Por ejemplo, el matrimonio es una relación uno - a - uno entre dos instancias de tipo Persona. Una mujer puede tener una relación uno - a - muchos (*madre de)* con muchos niños. Profesores y estudiantes típicamente participan en relaciones muchos - a - muchos. Las relaciones en el Modelo de Objetos son parecidas a las relaciones del modelo entidad-relación.

Las relaciones en el modelo de objetos no tienen nombres y no son "primera clase." Una relación no es en sí misma un objeto y no tiene un identificador de objeto. Una relación es definida implícitamente por la declaración de *trayectorias navegables* que permiten a las aplicaciones usar conexiones lógicas entre los objetos que participan de la relación. Las *trayectorias navegables* se declaran en pares, una en cada dirección de la relación binaria. Por ejemplo, un profesor enseña cursos y un curso es enseñado por un profesor. La trayectoria enseña se definiría en la declaración de la interface para el tipo profesor. La trayectoria es_enseñado_por se definiría en la interface de la declaración para el tipo curso. El hecho que ambas trayectorias navegables apliquen a la misma relación es indicado por una cláusula inversa en ambas declaraciones de las trayectorias navegables. Por ejemplo:

```
interface Profesor {
        ...
        relationship Set<Curso> enseña
```

```
                inverse Curso::es_enseñado_por;
        ...
    }
and
    interface Curso {
        ...
        relationship Profesor es_enseñado_por
            inverse Profesor::enseña;
        ...
    }
```

La relación definida por las trayectorias navegables enseña y es_enseñado_por es una relación uno - a - muchos entre los objetos Profesor y Curso. Esta cardinalidad se muestra en las declaraciones de las trayectorias navegables. Una instancia de Profesor se asocia con un conjunto de instancias de Curso por medio de la trayectoria enseña. Una instancia de Curso se asocia con una única instancia de Profesor por medio de la trayectoria es_enseñada_por.

Las trayectorias navegables de muchos objetos pueden ser ordenadas o desordenadas. Si se usa Set, como en Set<Curso>, los objetos al final de la trayectoria están desordenados. Si se usa List, como en la siguiente definición, los objetos al final de la trayectoria se ordenan. La cláusula order_by es aplicable solamente cuando List es usado en la declaración de la trayectoria:

```
interface Profesor {
    ...
    relationship List<Curso> enseña
        inverse Curso::es_enseñado_por
        {order_by Curso::curso_no};
}
```

El ODBMS es responsable de mantener la integridad referencial de relaciones. Esto significa que si un objeto que participa en una relación se borra, entonces cualquier trayectoria navegable al objeto también debe ser borrada. Por ejemplo si una instancia particular de curso se borra, entonces no solamente la referencia de ese objeto a la instancia de profesor por medio de la trayectoria es_enseñado_por es borrada, si no también cualquiera referencias en objetos profesor a la instancia de curso por medio de la trayectoria enseña es borrada. Mantener la integridad referencial asegura que las aplicaciones no pueden referenciar trayectorias que conducen a objetos inexistentes.

```
    attribute Estudiante top_of_class;
```

Un atributo puede ser un objeto valuado. Este tipo de atributo permite a un objeto referenciar otro, sin la expectativa de una trayectoria navegable inversa o integridad referencial.

Es importante notar que una relación de trayectoria navegable no es equivalente a un apuntador. Un apuntador en C++ o Smalltalk no tiene la connotación correspondiente de una trayectoria navegable inversa, que formaría una relación.

Las operaciones definidas en una relación y sus trayectorias navegables varían según la cardinalidad de la trayectoria. Cuando la trayectoria tiene cardinalidad "uno", las operaciones se definen para crear una relación, para destruir una relación, y para atravesar la relación. Cuando la trayectoria tiene cardinalidad "muchos", la trayectoria además soporta todos los comportamientos definidos arriba sobre la clase Collection usada para definir el comportamiento de la relación. Las trayectorias garantizarán la integridad referencial en todos casos.

3.4.7 Modelado de comportamiento - operaciones

Aparte de las propiedades de los atributos y las relaciones, la otra característica de un tipo es su comportamiento, que se especifica como un conjunto de *signaturas de operación*. Cada signatura define el nombre de una operación, el nombre y tipo de cada uno de sus argumentos, los tipos de valor (es) que retornan, y los nombres de cualquiera *excepciones* (condiciones de error) que la operación puede causar. Nuestra especificación en el Modelo de objetos para operaciones es idéntica a la especificación de operaciones del OMG CORBA.

Una operación se define solamente sobre un único tipo. No hay noción en el Modelo de Objetos de una operación que exista independiente de un tipo, o de una operación definida sobre dos o más tipos. Un nombre de una operación necesita ser único dentro de una única definición de tipo. Así diferentes tipos podrían tener operaciones definidas con el mismo nombre. Los nombres de estas operaciones se ha dicho que son "overloaded". Cuando una operación es invocada usando un nombre overloaded, una operación específica debe seleccionarse para ser ejecutada. Esta selección, algunas veces llamada "operation name resolution" o "operation dispatching", está basada en el tipo más específico del objeto proveído como el primer argumento de la llamada actual.

Una operación puede tener efectos secundarios. Algunas operaciones no retornan valores. El modelo de objeto ODMG no incluye una especificación formal de la semántica de las operaciones. Es una buena práctica, sin embargo, incluir comentarios en las especificaciones de interface, por ejemplo comentando sobre el propósito de una operación, cualquier efecto secundario que podría tener, pre y post condiciones, etc.

El Modelo de Objetos asume ejecución secuencial de operaciones. No requiere apoyo para operaciones paralelas o concurrentes, pero no excluye un ODBMS desde el que se pueda aprovechar el apoyo del multiprocesador.

3.4.8 Modelo de excepción

El modelo de objetos ODMG soporta dinámicamente la necesidad de manejadores de excepción, usando un modelo de terminación de manejadores de excepción. Las operaciones pueden ocasionar excepciones, y las excepciones pueden comunicarse a los resultados de excepción. Las excepciones en el Modelo de Objetos son en sí mismas objetos y tienen una interface la cual permite que se relacionen con otras excepciones en una jerarquía de generalización-especialización.

Una excepción raíz es proveída por el ODBMS. Este tipo incluye una operación para emitir un mensaje que informa que una excepción de tipo Exception_type ha ocurrido al terminar el proceso. La Información sobre la causa de una excepción o el contexto en que ha ocurrido se pasa al manejador de excepciones como propiedades del objeto Excepción.

El control es como se indica a continuación:

1. El programador declara un manejador de excepción dentro del alcance *s* capaz de manejar excepciones de tipo *t*.

2. Una operación de alcance *s* puede "levantar" una excepción de tipo *t*.

3. Cuando una excepción es "atrapada". La pila de llamadas es automáticamente levantada en tiempo de corrida del sistema fuera del nivel del manejador. Los destructores de los objetos son llamados desde la pila. Cualquiera transacciones que se encuentren activas, en el tiempo de ejecución en que el manejador de excepciones levante la pila, serán abortadas.

4. El manejador puede decidir manejar la excepción o bien pasarla.

Un manejador de excepciones que se declara a sí mismo capaz de manejar una excepción de tipo t, también puede manejar excepciones de cualquier subtipo de t. Un programador que requiere un control más específico sobre excepciones de un subtipo específico de t puede declarar un manejador para este tipo más específico dentro del mismo alcance del manejador.

La signatura de una operación incluye la declaración de las excepciones que la operación puede ocasionar.

3.4.9 Metadata

Metadata es información descriptiva acerca de los objetos en la base de datos. La metadata define los esquemas de las bases de datos; es usada por el ODBMS para estructurar bases de datos y en tiempo de ejecución para orientar el acceso a las bases de datos. La metadata es accesible a las herramientas y a las aplicaciones que usan las mismas operaciones que se aplican a los tipos definidos por el usuario. En el ambiente de OMG CORBA, la metadata se almacena en un interface repository IDL.

3.4.10 La jerarquía completa del tipo empotrado (built-in)

La figura 3.1 muestra el conjunto completo de tipos built-in (objetos empotrados) de la jerarquía de tipos del Modelo de objetos. Los tipos concretos son mostrados en letras no-cursivas y son directamente instanciables. Los tipos abstractos son mostrados en letras cursivas. En aras de simplificar, generador de tipo y tipos se incluyen en la misma jerarquía. Los generadores de tipo son representados con paréntesis angulares (p. ej., Seg < >).

3.4.11 Reglas para la compatibilidad de tipos

Todos los objetos o literales tienen un tipo, y cada operación requiere operandos. Las reglas para la identidad de tipos y para la compatibilidad de tipos se definen en esta sección.

Dos objetos o literales tienen el mismo tipo si y sólo si se han declarado para ser instancias del mismo tipo. Los objetos o literales que se han declarado para ser instancias de dos tipos diferentes no son del mismo tipo, aun cuando los tipos en cuestión definen el mismo conjunto de propiedades y operaciones. La compatibilidad de tipos sigue las relaciones de subtipos definida por la jerarquía de tipo. Si *TS* es un subtipo de *T*, entonces un objeto de tipo *TS* puede asignarse a la variable de tipo *T*, pero al revés no es posible. No se proveen conversiones implícitas entre tipos por el Modelo de Objetos.

Dos literales atómicas tienen el mismo tipo si pertenecen al mismo conjunto de literales. Dependiendo del lenguaje de programación, conversiones implícitas pueden ser proveídas entre tipos de literales escalares, p. ej., Long, Short, Unsigned long, Unsigned short, Float, Double, Boolean, Octet, Char. No se proveen conversiones implícitas para literales estructuradas.

Literal_type
    Atomic_literal

Long
Short
Unsigned long
Unsigned short
Float
Double
Boolean
Octet
Char
String
Enum<>
Collection_literal
Set<>
Bag<>
List<>
Array<>
Structured_literal
Date
Time
Timestamp
Interval
Structure<>
Object_type
Atomic_object
Set<>
Bag<>
List<>
Array<>

Figura 3.1. Conjunto completo de tipos Built-in

3.4.12 Valor nulo

Para cada tipo de literal (p. ej., float o Set< >) existe otro tipo de literal que soporta un valor nulo (p. ej., nullable_float o nullable_Set < >). Este tipo nullable es igual al valor nulo "nil". La semántica del valor nulo es igual a la definida en SQL.

3.4.13 Tipo tabla (table)

El tipo generador Table < > se define en el modelo de datos ODMG como un sinónimo del tipo generador Bag< Struct < >> tal que:

Table(a1:t1, a2:t2, ..., an:tn)

es equivalente a la definición

Bag<Struct<a1:t1, a2:t2, ..., an:tn>>

3.4.14 Modelo de transacciones

Los programas que usan objetos persistentes se organizan en transacciones. La gestión de transacciones es una importante funcionalidad del ODBMS, fundamental a la integridad de la base de datos, compartimiento, concurrencia, y recuperación. Cualquier acceso, creación, modificación, y eliminación de objetos persistentes debe hacerse dentro de una transacción.

Una transacción es una unidad lógica para que un ODBMS pueda ofrecer las garantías de atomicidad, consistencia, aislamiento, y durabilidad. *Atomicidad* significa que una transacción o termina o no tiene efecto en todo. La *consistencia* significa que una transacción lleva a la base de datos desde un estado internamente consistente a otro estado internamente consistente. Pueden haber veces durante la transacción en las que la base de datos es inconsistente. Sin embargo, ningún otro usuario de la base de datos ve cambios hechos por una transacción hasta que esa transacción es cometida. Los usuarios concurrentes siempre ven una base de datos internamente consistente. *Aislamiento* significa que la ejecución de transacciones concurrentes debe rendir resultados los cuales son indistinguibles de los resultados que se habrían obtenido si las transacciones se hubieran ejecutado en serie. Esta propiedad a veces se llama *serializabilidad.* La *durabilidad* significa que los efectos de transacciones cometidas se conservan, aun cuando hayan fallas de los medios de almacenamiento, pérdida de memoria, o pérdida de sistema. Una vez una transacción se ha cometido, las garantías del ODBMS de los cambios hechos por la transacción nunca se pierden. Cuando una transacción es cometida, todos los cambios hechos por esa transacción quedan permanentemente 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 permanecen en la base de datos, incluyendo cualquiera cambios hechos con anterioridad al tiempo de abortar.

En el modelo de objetos, los objetos transitorios no son sujetos a la semántica de la transacción. Esto significa que al abortar una transacción no se restaura el estado de los objetos transitorios modificados.

3.4.15 Control de bloqueo y concurrencia

El modelo de objetos ODMG usa bloqueo (lock) convencional basado en un enfoque del control de concurrencia. Las cerraduras pueden adquirirse sobre objetos particulares. Algunas implementaciones dóciles pueden o forzar o permitir cerraduras para escalar a algún otro nivel de granularidad.

El modelo de objetos ODMG apoya el tradicional control pesimista de concurrencia como su política default, pero no excluye DBMSs que apoyan una amplia gama de políticas de control de concurrencia. El modelo default requiere de la adquisición de un candado de lectura sobre un objeto antes de ser leído, y la adquisición de un candado de escritura sobre un objeto antes de que el objeto pueda modificarse. Los lectores de un objeto no ocasionan ningún conflicto con otros lectores, pero los escritores si ocasionan conflictos con lectores y escritores. Si hay un conflicto, el DBMS permite al que posee el candado proceder y la transacción que pidió acceso y se encuentra en conflicto con el candado debe esperar hasta que el que tiene el candado complete su uso. El que posee el candado puede completar por un "commit" o bien abortar, en este punto todos sus candados o cerraduras se liberan. Las transacciones sujetas a este protocolo se serializan en el orden del "commit".

3.4.15.1 Operaciones de transacción

Un ODBMS provee un tipo Transaction con las siguientes operaciones:

```
interface Transaction {
        void        begin();
        void        commit();
        void        abort();
        void        checkpoint();
};
```

La operación "*begin*" comienza una Transacción. Las transacciones deben explícitamente crearse y comenzarce; no son creadas automáticamente por el ODBMS cuando una base de datos se abre o sigue de un commit o un abort de una transacción.

La operación "*commit*" ocasiona que todos los objetos persistentes creados o modificados durante esta transacción lleguen a ser accesibles a otras transacciones en otros procesos. La instancia de Transaction se borra y todos los candados retenidos por esa instancia de Transaction se liberan.

La operación "*abort*" ocasiona que el objeto "Transaction" sea borrado y la base de datos pueda volver al estado en que estuvo con anterioridad al principio de la transacción. Todos los candados (cerraduras) retenidos por la transacción se liberan.

La operación "*checkpoint*" escribe todos los objetos modificados a la base de datos y retiene todos los candados retenidos por la transacción. No borra el objeto Transaction.

3.4.16 Operaciones de la base de datos

Un ODBMS puede administrar una o más bases de datos lógicas, cada una de las cuales pueden estar en una o más bases de datos físicas. Cada base de datos lógica es instancia del tipo "Database", el cual es proveído por el ODBMS. El tipo Database soporta las siguientes operaciones:

```
interface Database {
        void        open(instring database_name);
        void        close();
        void        bind(in any an_object, in string name);
        any         lookup(in string object_name);
};
```

La operación "*open*" debe invocarse, con un nombre de base de datos como parámetro, antes que cualquier acceso pueda ser hecho a los objetos persistentes en la base de datos. El modelo de objetos requiere que solamente una base de datos este abierta a la vez. Las implementaciones pueden extender esta limitación, incluyendo transacciones que soporten múltiples bases de datos. La operación *close* debe invocarse cuando un programa ha completado todo acceso a la base de datos. Cuando el ODBMS cierra una base de datos, desempeña operaciones necesarias de limpieza.

La operación "*lookup*" se usa para encontrar un identificador de objeto con el nombre pasado como argumento. Esta operación es definida en el tipo "Database", porque la gama de nombres de objetos es la base de datos. Los nombres de objetos en la base de datos, los nombres de tipos en el esquema de la base de datos, y la gama de instancias en la base de datos son globales a la base de datos. Son accesibles al programa una vez la base de datos se ha abierto. Los nombres de los objetos son un conveniente punto de entrada a la base de datos. Un nombre es destinado a un objeto usando la operación "*bind*".

El tipo Database puede soportar también operaciones diseñadas para la administración de bases de datos, p.ej., move, copy, reorganize, verify, backup, restore. Esta clase de operaciones no son especificadas aquí porque son consideradas como consideraciones de implementación fuera del alcance del modelo de datos.

4. PRINCIPIOS DE ARQUITECTURA EN LOS SISTEMAS DE BASES DE DATOS ORIENTADOS A OBJETOS

4.1 Introducción

El rendimiento es un requerimiento básico en cualquier sistema de administración de bases de datos. Gran número de factores son consecuencia de la complejidad de éste en los sistemas orientados a objetos (p.ej., la herencia, objetos complejos). Por lo que se necesitan técnicas adecuadas de almacenamiento para los objetos y de idexación (indización) para proporcionar un buen nivel de rendimiento.

4.2 Técnicas de almacenamiento para los sistemas de gestión de base de datos relacionales

Un disco está dividido en un conjunto de particiones o pistas, consistentes cada una de un número de segmentos. Cada segmento consta de un número de páginas o bloques. Una porción específica del disco, llamada cabecera, contiene información que incluye el número de particiones, la dirección y el tamaño de cada partición y un diario (log) para la recuperación en caso de caída del sistema. Los segmentos se describen por tablas en las cuales se almacenan las direcciones de las páginas de cada segmento.

La figura 4.1 muestra la organización de una página que contiene objetos. Cada página contiene una cabecera para mantener la información de la página, un array que contiene los desplazamientos de los objetos dentro de la página y los objetos propiamente. El desplazamiento de un objeto puede cambiar cuando el objeto crece o cuando la página se compacta para dejar espacio para otros objetos.

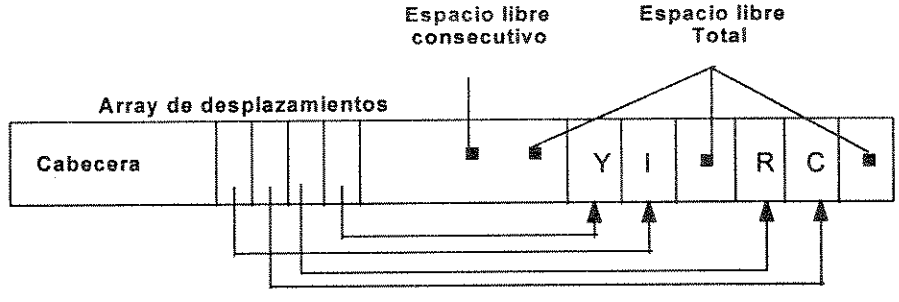


Figura 4.1. Organización de una página de almacenamiento.

En los sistemas de gestión de base de datos relacionales, los registros que representan las tuplas de una relación por lo general se almacenan uno a

continuación del otro, es decir consecutivamente en el disco. Si los atributos de una relación son de longitud fija, éstos se almacenan dentro del registro correspondiente como campos adyacentes. En estos casos, los registros de una relación se pueden almacenar en un solo archivo.

Para manejar registros de longitud variable, la mayoría de los sistemas de bases de datos almacenan los registros directamente sobre páginas de disco y simultáneamente le asignan un identificador (ID) a cada registro. La estructura que se escoge para los IDs desempeña un papel muy importante en la velocidad con la que se recuperan los registros. Por ejemplo, en un sistema los IDs pueden consistir en dos partes:

1. Los bits más altos o más significativos de un ID identifican el segmento y la página dentro del archivo donde está almacenado el registro. Los segmentos pueden estar asignados a diferentes porciones de uno o más archivos por medio de una tabla de segmentos. Estos bits indican una dirección física, y de este modo la página que contiene un registro puede ser recuperada con un solo acceso a disco.

2. Los bits más bajos o menos significativos identifican un registro dentro de una página. Estos bits son el índice a los campos de un vector, almacenado al comienzo de la página (o en alguna otra posición fija), que contiene las direcciones de los registros dentro de las páginas.

La figura 4.2 muestra la organización de una página donde las direcciones de los registros en la página están contenidas en un vector. Esta técnica para identificar los registros tiene las ventajas siguientes:

- En términos de acceso a disco, esto es tan rápido como un sistema que utilice las direcciones completas de los registros (página y byte dentro de la página), pero tiene como ventaja que permite cambiar la longitud de los registros y relocalizar éstos en la misma página o en una diferente, puesto que es fácil de cambiar la componente dentro del vector correspondiente al registro, insertando en la componente la nueva dirección del registro.
- Es a menudo más rápida que un sistema que utilice un ID puramente lógico (llamado ID sustituto). En efecto, el uso de un ID sustituto requiere de otro nivel de direccionamiento, puesto que se tiene que usar una tabla hash (la tabla que asocia las direciones físicas de los registros con los ID subrogados). Por lo tanto, para recorrer la tabla hash se requieren uno o más accesos a disco.

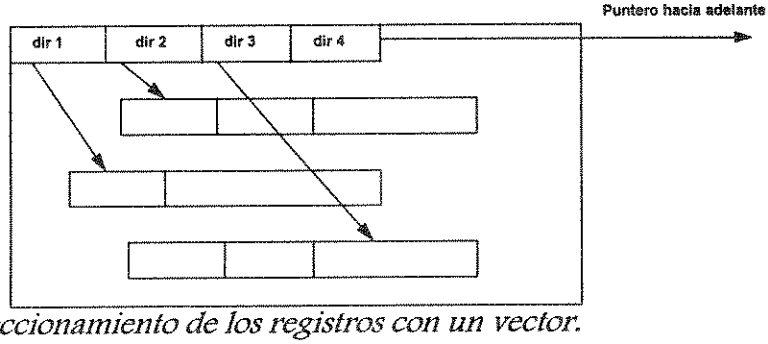


Figura 4.2. Direccionamiento de los registros con un vector.

4.3 La gestión de almacenamiento para los objetos

Dado que un modelo de base de datos orientado a objetos es más complejo que un modelo relacional, la organización de memoria en ellos debe ser capaz de manejar eficientemente:

- Objetos con atributos lo mismo atómicos que complejos.
- Objetos con atributos que pueden ser a su vez atómicos o complejos, (multievaludados).
- Objetos con atributos variantes.
- Objetos con atributos de campos muy largos. (BLOBs)

La eficiencia de la organización del almacenamiento depende no sólo de la estructura de los objetos y sus relaciones, sino también de la forma en que los programas de aplicación acceden a los objetos, que de ahora en adelante será denominado patrón de acceso. Los patrones de acceso se ubican en dos categorías principales:

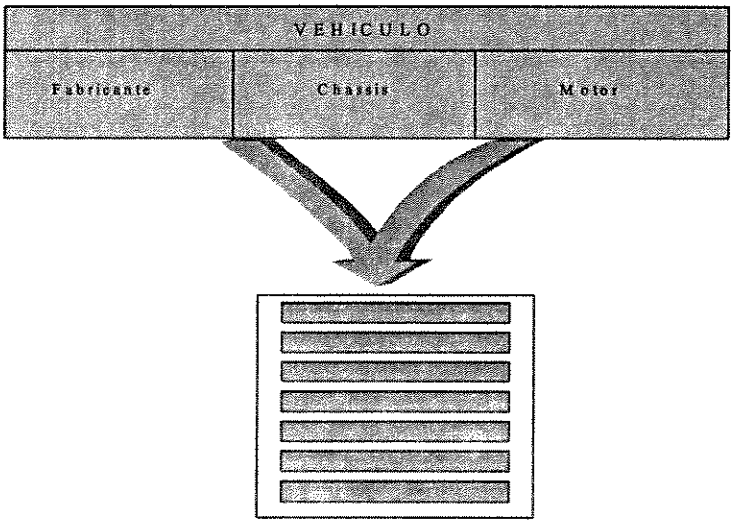
- *Acceso basado en el objeto completo.* Este patrón de acceso es apropiado en aplicaciones que llevan a cabo manipulaciones complejas de los objetos mediante programas especializados. En estos casos, el objeto completo se copia en la memoria virtual de la aplicación.
- *Acceso basado en los atributos del objeto.* Este patrón de acceso se usa para recuperar atributos de objetos ubicados en un nivel determinado a lo largo de la jerarquía de agregación, y es apropiado cuando es necesario acceder a objetos muy grandes.

Las técnicas de almacenamiento propuestas hasta la fecha, para los sistemas de bases de datos orientadas a objetos se encuentran dentro de dos enfoques:

- *El modelo directo.* En este modelo, los objetos se almacenan de la misma forma en que están definidos en el esquema conceptual: esto es, la unidad de almacenamiento es la misma que la unidad semántica. Más específicamente, los objetos que pertenecen a una misma clase se almacenan en un mismo archivo, y cada registro de un archivo es un objeto instancia de una clase (ver figura 4.3a). La mayor ventaja de este enfoque es que la transferencia de un objeto completo es un proceso muy eficiente, puesto que no se requieren operaciones de reunión (join) para reconstruir objetos que han sido previamente descompuestos. La desventaja consiste en el hecho de que el acceso a un conjunto de atributos puede ser muy costoso, especialmente si los objetos tiene campos de grandes dimensiones.

- *El modelo normalizado.* En este modelo, los objetos se descomponen en componentes atómicas que se almacenan en archivos diferentes (ver figura 4.3b). La relación entre las diferentes componentes se mantiene por medio de los IDOs[15].

a) El modelo directo.



b) El modelo normalizado.

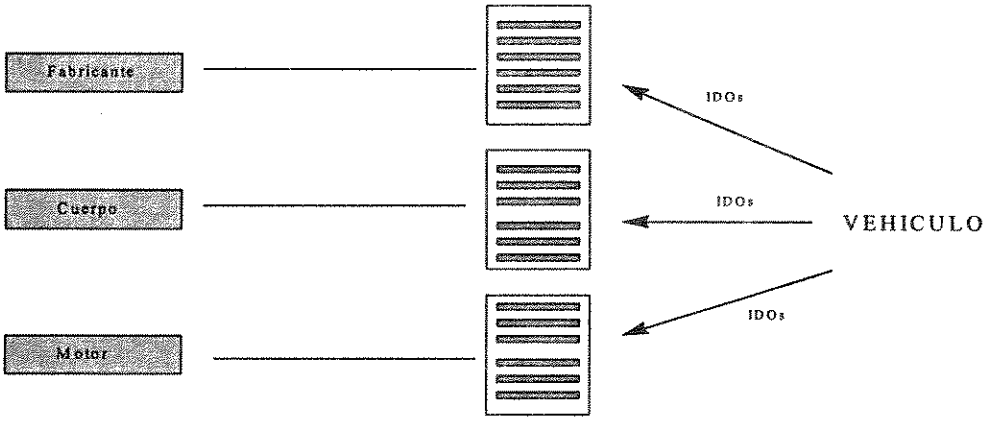


Figura 4.3. Enfoques de las técnicas de almacenamiento de las bases de datos orientadas a objetos.

Se puede adoptar un enfoque intermedio entre estos dos extremos; los objetos complejos se descomponen, pero las componentes se agrupan entre sí de acuerdo con patrones de acceso, de modo que las componentes a las que se accede contemporáneamente estén almacenadas en el mismo archivo. El problema

[15] IDOs: Identificadores de objetos.

de este enfoque es que la eficiencia depende de tener un conocimiento previo de los patrones de acceso exactos de las aplicaciones.

El modelo directo de almacenamiento que se ha presentado anteriormente es el método más simple de almacenamiento de los atributos de los objetos y es el mismo que el que se utiliza en los sistemas de bases de datos relacionales. Este es un método eficiente, pero sufre de un cierto número de inconvenientes en situaciones como las siguientes:

- Cuando se tiene que manejar atributos de longitud variable, por ejemplo, cadenas de caracteres de longitud variable, listas, conjuntos y colecciones. En estos casos es mejor utilizar el enfoque normalizado, almacenar el atributo como un objeto separado en un área separada, de modo que el objeto se pueda encontrar por medio de un IDO.
- Cuando se crean nuevos atributos, por ejemplo, si se modifica el esquema. Si se reserva un espacio fijo para los atributos de un objeto, entonces no se pueden añadir nuevos atributos, a menos que se reserve algún área al final del objeto para este propósito.
- Si la mayoría de los atributos tiene el valor nulo. Asignar espacio de tamaño fijo para estos atributos puede significar malgastar el espacio.

En los últimos dos casos, en la creación de nuevos atributos y los atributos nulos (llamados también esparcidos), se usa una lista de propiedades. Una lista de propiedades consta de una secuencia de tripletas < *identificador, tamaño, valor* > para cada atributo del objeto. Los *identificadores*, que no deben ser confundidos con los IDOs, indican qué atributos del objeto se almacenan. El *tamaño* contiene el número de bytes almacenados (que puede ser omitido para atributos de tamaño fijo), y el *valor* es el valor (de tamaño variable) del atributo.

Las listas de propiedades son particularmente flexibles. En efecto, el mismo tipo de atributo puede tener valores de diferentes longitudes en objetos diferentes, o los atributos pueden estar almacenados en localizaciones físicas diferentes, o no todos los objetos en la misma clase tienen necesariamente que tener el mismo conjunto de atributos. Las listas de propiedades son útiles también si hay atributos esparcidos, puesto que a los atributos que tienen valores nulos no se les necesita almacenar. Por tanto, se puede ganar en espacio y en tiempo.

La desventaja principal de la lista de propiedades es que, para recuperar o almacenar el valor de un atributo determinado, se necesita recorrer toda la lista de propiedades para encontrar el atributo deseado (ver figura 4.4). Otra desventaja es que el formato de representación de la lista de propiedades se debe transformar para adecuarse al formato de representación utilizado por el lenguaje de programación usado por el sistema de bases de datos.

Figura 4.4. Lista de propiedades.

La jerarquía de herencia es otro factor que influye en la forma en que se almacenan los atributos. En efecto, si hay herencia (simple o múltiple) no sólo se debe almacenar los atributos de una clase, sino también aquellos de su superclase (o de sus superclases si la herencia es múltiple).

En el caso de herencia simple, el objeto se puede almacenar primero almacenando los atributos de su clase padre y luego aquéllos de sus subclases, respectivamente. Este método de almacenamiento de los atributos, que es utilizado por la lista de propiedades, es conveniente incluso para almacenar atributos de tamaño variable. Debe notarse que, para atributos de tamaño fijo, el desplazamiento de un campo en esta forma de representación es siempre el mismo, incluso para el campo que represente un atributo de la subclase. Esto se debe a que los atributos de la subclase se concatenan con aquéllos de la superclase.

Para los sistemas de bases de datos orientados a objetos que permiten herencia múltiple, se utilizan diferentes técnicas de almacenamiento, a las descritas anteriormente. En este caso se puede utilizar la lista de propiedades; o los objetos se pueden almacenar separadamente, cada uno de ellos conteniendo los campos asociados con una superclase, y enlazado con otro. En lo que respecta al almacenamiento de atributos de tamaño variable y de grandes dimensiones (p. ej., imágenes o textos), éstos se pueden manejar como se ilustró anteriormente. Este método de almacenamiento se combina con un mecanismo de flujo o demanda de página de tal modo que sólo se transfieran a memoria principal porciones del objeto, en lugar del objeto completo.

4.4 Técnicas de agrupamiento

El problema de agrupamiento (clustering) en los sistemas de administración de bases de datos se centra en *particionar* los objetos en una base de datos y ubicar estas particiones en el disco. El objetivo es reducir el número de operaciones de E/S que es necesario hacer sobre disco para la recuperación de los objetos.

Con la finalidad de definir estas agrupaciones, lo mismo en los sistemas de bases de datos relacionales que en los sistemas de bases de datos orientados a objetos, se debe tener en cuenta dos factores principales: *la estructura de los objetos y el patrón de acceso.* Este último se puede definir previendo que se conocen las características de las aplicaciones involucradas, pero también sería útil tener la capacidad de analizar su evolución en el tiempo. Para lograr esto se podrían coleccionar estadísticas no sólo con respecto a la frecuencia de acceso a

objetos individuales, sino también con respecto a la frecuencia con la cual cuando se accede a un objeto *x* también hay que acceder a un objeto *y*. Sin embargo, acumular una estadística tal podría implicar elevados costos adicionales y sería necesario almacenar grandes matrices de correlación.

Los sistemas de bases de datos relacionales, utilizan dos técnicas de agrupamiento. La primera involucra el almacenamiento de las tuplas en un mismo segmento de página del disco, sobre la base del valor de un atributo o de una combinación de atributos de una relación. La segunda consiste en almacenar las tuplas de más de una relación en el mismo segmento, si las relaciones tienen uno o más atributos en común y éstos tienen valores iguales. El rendimiento del sistema se mejora cuando se hacen consultas que requieren la ejecución de una reunión.

Las técnicas de agrupación para los sistemas de bases de datos orientados a objetos, comparadas con las de los sistemas relacionales, deben tener en cuenta la existencia de objetos complejos, al igual que la herencia (simple o múltiple) y la presencia de métodos. La estrutura de un objeto complejo se puede representar como una jerarquía o como un grafo acíclico dirigido (DAG). Las operaciones sobre una estructura tal son navegacionales a través de los IDOs y de la recuperación de los ancestros y descendientes de un nodo.

Una buena estrategia de agrupamiento implica poner los nodos de una jerarquía de agregación y/o de un DAG en una *secuencia de agrupamiento lineal.* Los nodos en la jerarquía de agregación se pueden almacenar en un orden primero en profundidad, de modo que todos los nodos descendentes de un nodo *p* en la jerarquía se almacenan inmediatamente después de *p*. Esta estrategia es eficiente cuando se tiene que recuperar un objeto y todos sus descendientes. Se han dado cinco opciones básicas de agrupación.

Las dos primeras opciones son las mismas que las presentadas anteriormente para las bases de datos relacionales. La tercera alternativa involucra la agrupación de todos los objetos instancias de las clases que pertenezcan a una jerarquía de agregación de clases; esta estrategia se puede ver como una variación de la segunda opción, que implica el almacenamiento de objetos de diferentes clases en posiciones adyacentes de memoria. La cuarta opción de agrupación consiste en almacenar contiguamente todos los objetos instancias de las clases pertenecientes a la jerarquía de herencia de una clase dada, con análogas consideraciones que la jerarquía de agregación. La última de las estrategias es el resultado de una combinación de las dos estrategias precedentes; ésta es un intento por encontrar una organización conveniente para la recuperación de las instancias de cualquier subgrafo conectado del grafo del esquema con raíz en una clase particular. En efecto, una clase es la raíz de una jerarquía de herencia y de una jerarquía de agregación.

Las estrategias de agrupamiento anteriores son estáticas, debido a que una vez que un agrupamiento ha sido definido, no puede cambiar en tiempo de ejecución. Esto crea dos problemas potenciales:

1. Un esquema de agrupamiento estático no toma en cuenta la evolución dinámica de los objetos (creación y eliminación de objetos). Por ejemplo, en aplicaciones como las de bases de datos de diseño, los objetos son actualizados frecuentemente durante las primeras etapas del ciclo de diseño. Estas actualizaciones pueden destruir la estructura del agrupamiento inicial. Esto puede significar que la estructura de agrupamiento tiene que ser reorganizada para mantener el nivel de rendimiento.
2. Los objetos pueden estar conectados por medio de varias relaciones, que pueden generar jerarquías de agregación o DAGs independientes. Por ejemplo, en una base de datos de diseño, un diseño evoluciona a través de varios estados, tales como creación inicial, verificación de reglas de diseño, extracción de partes de diseño y simulación. Las herramientas de diseño utilizadas durante las diferentes etapas pueden tener patrones de acceso que no son los mismos que aquellos escogidos para la definición inicial del agrupamiento. Por tanto, es difícil que un único esquema de agrupamiento pueda adecuarse a todos los posibles patrones de acceso de las aplicaciones, en cuyo caso sería preferible usar esquemas diferentes de agrupamiento para cada etapa. Es más, varios usuarios pueden tener acceso al mismo tiempo, con patrones de acceso diferentes, a los mismos objetos; de este modo, el uso de un esquema de agrupamiento basado en los requerimientos de una aplicación individual (o de una clase individual de usuarios) puede penalizar el rendimiento de otras aplicaciones y de otras clases de usuarios.

También muchas técnicas de agrupamiento usan la página de disco como unidad de agrupación: el objetivo es reducir el número de páginas accedidas que se requieren para recuperar un objeto complejo. Estas técnicas dan por hecho que cada acceso a una página requiere una operación de E/S sobre disco y que el tiempo total de acceso se obtiene multiplicando el número de páginas accedidas por el tiempo medio de aceso a la página; pero ellas no toman en cuenta el efecto de que las páginas pueden no estar adyacentes unas con otras, lo que puede resultar en un tiempo de acceso no uniforme para las diferentes páginas. Es más, estas técnicas agrupan los objetos entre sí sobre la base de un patrón de acceso único.

4.4.1 Agrupamiento dinámico

En lo que se refiere a los efectos de la evolución dinámica de los objetos en un esquema de agrupamiento, entre las operaciones signficativas están las de crear y eliminar objetos. Mientras que eliminar puede no ser un problema, puesto que los objetos eliminados se pueden marcar para que el espacio que ellos dejan disponible

pueda ser reutilizado más tarde, cuando se crea un nuevo objeto se le debe asignar inmediatamente espacio en disco.

Con independencia de la estrategia de agrupamiento utilizada, es improbable que la secuencia de creación de los objetos de la jerarquía de agregación sea la misma que la secuencia deseada de agrupamiento. Por ejemplo, los objetos podrían crearse en una secuencia primero en amplitud; en este caso, las páginas en las que se almacenan los objetos tienen que estar conectadas, con el fin de mantener la validez de la secuencia deseada de agrupamiento. Sin embargo, las páginas, en general, estarán esparcidas, esto es, no adyacentes unas con otras. Esto puede tener un impacto considerable, puesto que recuperar desde disco un conjunto de páginas esparcidas requiere más tiempo del que se requeriría si las páginas estuvieran adyacentes, debido a las características mecánicas de los controladores de disco. En efecto, leer un bloque de datos desde el disco implica un tiempo de posicionamiento, un tiempo de latencia y un tiempo de transferencia.

Para un conjunto de bloques almacenados en secuencia, sólo se necesitan un posicionamiento y una latencia, mientras que para tener acceso a *n* bloques almacenados aleatoriamente, se requiere un tiempo igual a la suma de los tiempos de posicionamiento y de latencia para cada bloque.

La reorganización y la recompactación de páginas en un agrupamiento después de las operaciones de modificación es similar al problema de reorganización de archivos. Hay dos tipos de técnicas de reorganización de archivos: en línea (on-line) y fuera de línea (off-line). El problema de la reorganización fuera de línea implica determinar el mejor punto de reoganización, esto es, la frecuencia con la cual esto se debe hacer. Existe una técnica de organización en línea que se basa en la utilización de *chunks* (un conjunto de páginas almacenadas adyadentemente en disco) como unidad de almacenamiento de las agrupaciones. En general, se justifica la reorganización dinámica en casos donde la razón entre las operaciones de lectura y escritura es alta, mientras que, si lo contrario es cierto, es más conveniente la reorganización fuera de línea.

4.4.2 Agrupaciones para relaciones múltiples

Se ha visto que pueden existir diferentes relaciones entre los objetos que resultan en diferentes jerarquías de agregación y/o DAGs. Por ejemplo, en una base de datos para CAD se piensa que un conjunto de objetos constituye una configuración, o una versión de diseño. Debido a las características de los algoritmos utilizados en programas de aplicación tales como herramientas CAD, algunas relaciones se usan con mayor frecuencia que otras. Esta situación se puede representar en un grafo dirigido, en el cual los nodos representan los objetos y los archivos representan las relaciones entre los objetos. Los diferentes patrones de acceso para las distintas relaciones se pueden representar asociándole un peso

a cada arco, dándole un peso más alto a aquellos que corresponden a relaciones que se usan con más frecuencia. En otras palabras, un arco con un peso *p* que va de un nodo *x* hacia un nodo *y* significa que la probabilidad condicional de acceder a *y*, una vez que se ha accedido a *x*, es *p*.

4.5 Técnicas de indexación para los sistemas de bases de datos orientadas a objetos

Las técnicas de indexación para las bases de datos orientadas a objetos deben considerar los siguientes factores:

- *Predicados anidados:* debido a las estructuras anidadas de los objetos, la mayoría de los lenguajes de consulta orientados a objetos permiten predicados (condiciones) sobre atributos anidados y no anidados de los objetos. Los atributos anidados se expresan a menudo por medio de las *expresiones de camino*[16].
- *Herencia:* una consulta se puede aplicar a una clase, o a una clase y a todas sus subclases.
- *Métodos:* los métodos se pueden usar en consultas como métodos atributos derivados o como métodos predicados. Un método atributo derivado tiene una función comparable a la de un atributo, en el hecho de que retorna un objeto (o un valor) al cual se le pueden aplicar comparaciones. Un método predicado retorna la constante lógica True o False. El valor retornado por un método predicado puede, por consiguiente, ser utilizado en la evaluación de una expresión booleana que determina si el objeto satisface o no la consulta.

Las técnicas de idexamiento descritas en esta sección están basadas en las que se han implementado en ORION.

ORION, como la mayoría de los sistemas convencionales de base de datos, soporta índices secundarios en atributos especificados por el usuario (columnas) de clases especificadas (relaciones). Tales índices impulsan búsquedas asociativas de la base de datos para consultas con criterios de búsqueda. Mientras el alcance de acceso de una consulta en una única relación R en una base de datos relacional es simplemente R, el de una consulta contra una clase C en una base de datos orientada a objetos es en general la clase C y todas las subclases de C, y sus subclases, etc. Desde que un atributo de una clase C es heredado dentro de todas sus subclases descendientes, puede tener sentido mantener un índice sobre un atributo para todas las clases sobre una jerarquía de clases derivada en la clase C, más bien que mantener un índice separado sobre el atributo para cada una de las clases en la jerarquía de clases.

[16] Camino: Una rama en una jerarquía de agregación que comienza con una clase C y termina con un atributo anidado de C.

Nos referiremos al índice que se mantiene sobre un atributo de una única clase como un índice de una clase (*single-class index)*, y un índice sobre un atributo de todas las clases sobre una jerarquía de clases derivada de una clase particular como una índice de una jerarquía de clases (*class-hierarchy index).* ORION actualmente soporta indexamiento para una clase (single-class).

4.6 Evaluación de una consulta orientada a objetos

El modelo de datos orientado a objetos, en su forma convencional, es suficientemente poderoso para representar un objeto complejo como un objeto anidado recursivamente. Un objeto puede definirse con un conjunto de variables instancia. Una clase puede especificarse como el dominio (tipo) de una variable instancia, y la clase dominio, a menos que sea una clase primitiva (tal como una clase string, integer, o boolean), a la vez consiste de un conjunto de variables de instancia, y así sucesivamente. El estado interno de un objeto consta de los valores de todas sus variables de instancia.

El valor de una variable de instancia es una instancia de su dominio, si el dominio es una clase primitiva; o de otra manera, referencia al (*identificador de objeto)* de una instancia del dominio. Por ejemplo, en la figura 4.5, se muestra el esquema de la clase Vehículo desde el punto de vista de las variables de instancia Fabricante, Cuerpo, TrenDeManejo, y Color. El dominio de la variable de instancia Color es la clase primitiva String. El dominio de la variable de instancia Fabricante es la clase Compañía, la variable de instancia Cuerpo tiene CuerpoDelAuto como su dominio, y el dominio de TrenDeManejo es la clase TrenDeManejoDelAuto. Cada una de las clases Compañía, CuerpoDelAuto, y TrenDeManejoDelAuto consiste de su conjunto propio de variables de instancia, las cuales a la vez tienen asociados dominios (los cuales por simplicidad no se muestran).

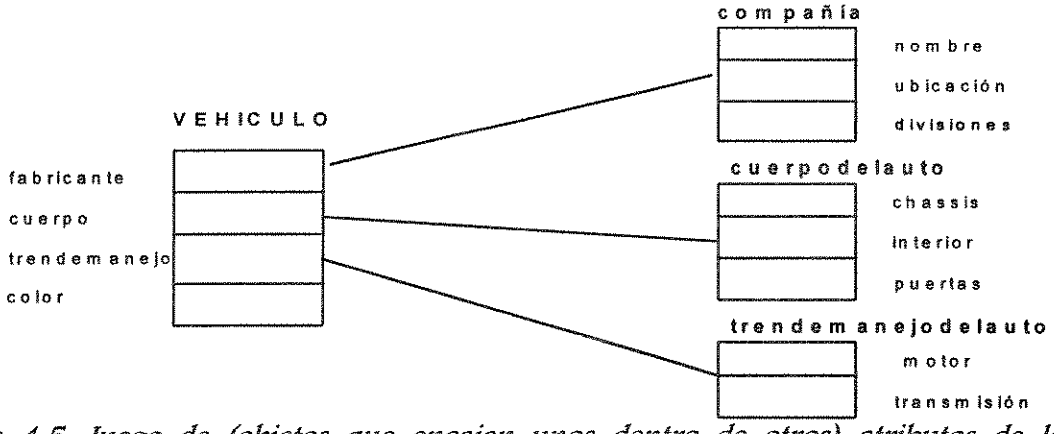


Figura 4.5. Juego de (objetos que encajan unos dentro de otros) atributos de la clase Vehículo.

El almacenamiento de un objeto a través de los dominios de sus atributos inmediatamente sugiere totalmente traer una instancia, la instancia y todas las

instancias que la instancia referencie mediante sus atributos deben ser traídos recursivamente. Esto significa que para traer una o más instancias de una clase, la clase y todas las clases especificadas como dominios no primitivos de los atributos de la clase deben ser recorridos recursivamente. Por ejemplo, para traer instancias de la clase vehículo de la figura 4.5, las clases que necesitan ser recorridas incluyen no solamente Vehículo sino también los dominios no primitivos de Vehículo, específicamente, Compañía, CuerpoDelAuto, TrenDeManejoDelAuto, así como también los dominios no primitivos de estas clases.

En general, una consulta puede formularse en un esquema orientado a objetos, el cual traerá instancias de una clase que satisfaga verdaderamente el criterio de búsqueda. Una consulta puede restringir las instancias de una clase a ser traídas especificando predicados contra cualquiera variables de instancia de la clase. Un ejemplo de una consulta en el esquema de la figura 4.5 es el siguiente:

Q1. Encontrar todos los vehículos azules fabricados por Ford Motor Compañía.

En una base de datos orientada a objetos, un atributo puede ser de dos tipos: simple o complejo. Un atributo simple es uno cuyo dominio es una clase primitiva. Un atributo complejo es uno cuyo dominio es una clase con uno o más atributos, incluyendo atributos complejos. Un predicado sobre un atributo simple es llamado un predicado simple, mientras que uno sobre un atributo complejo es llamado un predicado complejo. Adicionalmente, una consulta que involucra solamente predicados simples se llamará una consulta simple, y una que involucra uno o más predicados complejos se llamará una consulta compleja.

Podemos representar una clase y los dominios de todos sus atributos complejos en forma de un grafo dirigido, que llamaremos el grafo de consulta. Cada nodo sobre un grafo de consultas representa una clase, y un borde desde un nodo A hacia un nodo B significa que esa clase B es el dominio de un atributo complejo de la clase A. Un grafo de consultas tiene una única raíz, la clase cuyas instancias van a ser traídas. Cada hoja nodo de un grafo de consultas tiene únicamente atributos simples. Un grafo de consultas puede contener ciclos.

El proceso de traer objetos anidados, que llamaremos instanciación de objetos (object instantiation), es parecido a la evaluación relacional de consultas. Podemos ver una clase como una columna de una relación. Una relación se aumenta con un sistema-definido de identificadores únicos (UID[17]) de columnas para el identificador de tuplas. La recuperación de una instancia del dominio de la clase D de un atributo A de una clase C es parecido al Join relacional de una tupla de una relación C con una tupla de una relación D, donde el Join de las columnas son la columna A de la relación C y la columna UID de la relación D. Podemos

[17]Los UIDs (Identificadores únicos definidos) son términos que se utilizan en los sistemas relacionales para identificar las tuplas cuando estas se agregan a una relación; analogamente, se utiliza el término IDOs (Identificadores de objetos) en los sistemas orientados a objetos para representar referencias entre los objetos.

apresurarnos a comentar que, a pesar de estas similitudes, hay unas pocas diferencias importantes entre evaluación de consultas relaciones y la instanciación de objetos. Se presentará una discusión detallada de estos puntos más adelante.

En general hay más de una manera (frecuentemente llamada un plano de evaluación de consultas) para evaluar que una consulta rinda el resultado correcto. Sin embargo, cada plan incurre en un costo diferente. Hay dos opciones fundamentales en planes para recorrer las clases anidadas para la instanciación de objetos: recorrido hacia adelante y al reverso. El optimizador de consultas de un sistema de base de datos considerara un número de planes razonables con base en estas opciones (y sus combinaciones) para evaluar cualquier consulta determinada, y seleccionar uno con el mínimo costo esperado.

En el recorrido hacia adelante, las clases sobre un grafo de consultas se recorren en orden primero-profundidad que comienza desde la raíz del grafo de consultas, y sigue a través de los dominios de cada variable de instancia compleja. Como un ejemplo, consideremos la consulta Q1. Un recorrido hacia adelante de el grafo de consultas comienza con el conjunto de todas las instancias de la clase Vehículo en las que el atributo color tiene un valor "azúl". Para cada una de estas instancias, el valor de su atributo fabricante es extraído; ese valor es una instancia de la clase compañía. El valor del atributo Nombre en la instancia compañía se examina entonces. Si el valor es el string "Ford", la instancia de compañía califica, y a la vez, la instancia de Vehículo que tiene esa instancia de compañía como su fabricante (fabricante) satisface la consulta.

Otra manera para desempeñar la instanciación de objetos es el recorrido al revés, en el cual las hojas clases de un grafo de consulta se visitan primero, y luego sus padres, trabajando hacia la clase raíz. Como un ejemplo, considere una vez más la consulta Q1. En vez de comenzar con el conjunto de todas las instancias de Vehículo, la evaluación de consultas comienza con la clase compañía. De todas las instancias de Compañía son identificadas las que tienen el string "Ford" en el atributo Nombre. Los IDOs de estas instancias se buscan en el atributo Fabricante de la clase Vehículo. El resultado de la consulta es el conjunto de las instancias de Vehículo que tienen el string "azúl" en el atributo Color y que contienen en el atributo Fabricante un IDO que está en la lista de IDOs para las instancias de Ford Motor Compañía.

Para apoyar la eficiente recuperación de tuplas que satisfagan predicados de búsquedas, el subsistema de almacenamiento de un sistema relacional de base de datos comúnmente soporta índices secundarios sobre columnas de relaciones especificadas por el usuario. Similarmente, los sistemas de bases de datos orientados a objetos pueden mantener un índice sobre un atributo de una clase. Por ejemplo, si un índice se mantiene sobre el atributo primitivo Nombre de la clase compañía, puede usarse la ventaja del recorrido en reverso del grafo de consultas para nuestro ejemplo de la consulta Q1. Por otra parte, si hay un índice sobre el

atributo color de la clase Vehículo, puede usarse el recorrido hacia adelante del grafo de consultas. En ambos casos, el uso de un índice puede reducir significativamente el costo de E/S de recorrer el grafo de consultas para la instanciación de objetos.

Una de las mayores e importantes diferencias entre una base de datos relacional y una base de datos orientada a objetos es que en una base de datos orientada a objetos una clase puede especializarse en un número de subclases. Por ejemplo, en la figura 4.6 se muestra una jerarquía de clases de base de datos que incluye la clase Vehículo y el dominio de las clases de los atributos de la clase Vehículo. La clase vehículo se muestra que ha sido especializada en la clase Automóvil y la clase camión. Similarmente, la clase compañía tiene las subclases CompañíaVehículo y CompañíaComputadora. En general, una clase puede tener cualquier número de subclases y/o superclases. La raíz de una jerarquía de clases es un sistema-definido clase OBJETO, y cualquier clase que el usuario define sin una superclase es por default una subclase de la clase OBJETO.

El hecho de que un esquema de bases de datos orientadas a objetos explícitamente capture la relación IS-A entre un par de clases tiene dos impactos importantes sobre la semántica de la instanciación de objetos. Uno es que el alcance de acceso de una consulta en una clase puede ser solamente a las instancias de la clase, o puede incluir también las instancias de todas las subclases de la clase. Por ejemplo, el usuario puede emitir una consulta contra la clase vehículo para traer sólo las instancias de la clase el vehículo, o puede emitir una única consulta contra la clase Vehículo para traer todas las instancias que califiquen de la clase vehículo y las subclases de vehículo.

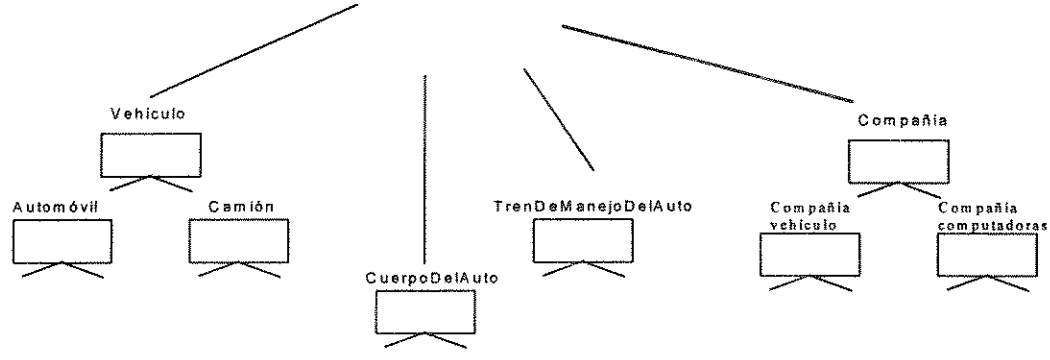


Figura 4.6. Una Jerarquía de Clases.

Otro impacto importante es que el dominio D de un atributo de una Clase C es realmente la clase D y todas las subclases de D. Por ejemplo, el atributo fabricante de la clase vehículo puede tomar como su valor una instancia de la clase Compañía o una instancia de cualquier subclase de Compañía. Esto significa que en el recorrido al reverso del grafo de consulta para Q1, la clase Compañía y todas sus subclases deben ser recorridas.

Esta semántica de la instanciación de objetos fuerza cambios importantes en la manera en que un sistema de base de datos puede usar un índice. Tradicionalmente, un índice se ha mantenido sobre un atributo de una clase única (o una relación). Esto significa que para apoyar la evaluación de una consulta cuyo alcance de acceso es una jerarquía de clase, el sistema debe mantener un índice sobre el atributo para cada una de las clases en la jerarquía de clase. Sin embargo, es claro que frecuentemente puede tener sentido mantener un índice sobre un atributo para una jerarquía de clases, y usar la evaluación de consultas en cualquier clase única en la jerarquía de clase o en cualquier sub-jerarquía de la jerarquía de clases. Llamaremos al enfoque tradicional de mantener un índice como la clase indexamiento de una clase (*single-class indexing)*, y nos referiremos al enfoque alternativo de mantener un índice sobre un atributo para una jerarquía de clases *class-hierarchy indexing.*

Intuitivamente, ese índice de jerarquía de clases (class-hierarchy) puede en general ser más efectivo en la evaluación de una consulta cuyo alcance de acceso mide un subconjunto importante (o mayor) de las clases en la jerarquía de clases indexada, mientras que un índice de una clase (single-class) debería ser más apropiado para una consulta sobre una única clase.

4.7 Estructura de índices

En esta sección se describen los formatos de los nodos de los índices B-tree. Estos formatos son en base al índice de una única clase (single-class) B-tree que se ha implementado en ORION. También es parecido al que se usó en en el sistema de base de datos relacional de IBM SQL/DS. En una base de datos relacional, las columnas tienen tipos primitivos de datos; así, los valores claves en un índice son los datos primitivos tales como enteros o strings. En una base de datos orientada a objetos, el dominio de un atributo puede ser o una clase primitiva o alguna clase definida por el usuario. Por lo tanto, los valores claves en un índice pueden ser o los IDOs de las instancias del dominio de la clase o algunos valores primitivos.

En la figura 4.5 se muestra el formato de un nodo no-hoja (un nodo que no constituye hoja). El nodo consta de *f* registros, donde cada registro es un par (llave, apuntador), y la llave a la vez es un par (llave-longitud, llave-valor), donde llave-longitud es la longitud en bytes de la llave-valor. El rango de *f*, está entre d y 2d, donde d es el orden de un B-tree. El nodo raíz puede abarcar entre 2 y 2d registros. El apuntador en cada registro contiene la dirección física del nodo índice del siguiente. Si un registro necesita ser insertado en un nodo que contiene 2d registros, el nodo divide y los 2d + 1 registros se distribuyen en dos nodos.

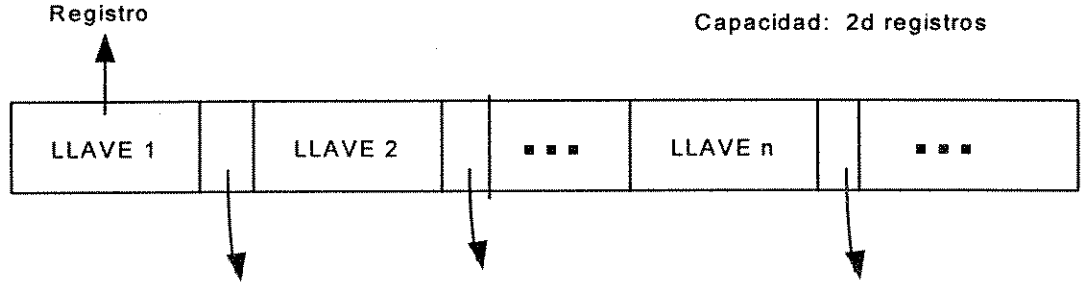
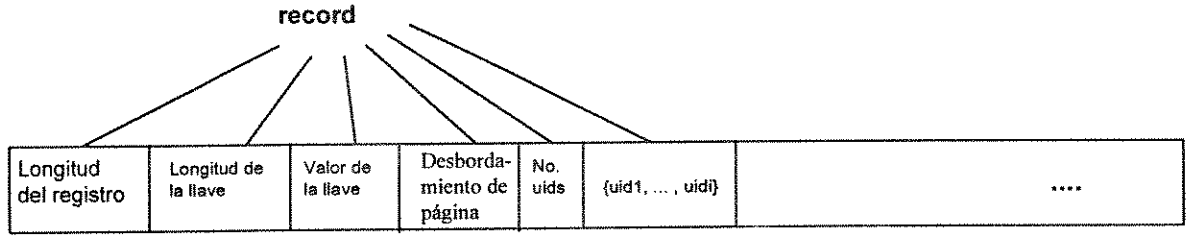


Figura 4.5. Un nodo no-hoja.

Un nodo hoja de un índice tiene un formato diferente que el de un nodo no-hoja. Adicionalmente, el formato de un nodo hoja de un índice de una única clase (single-class index) es diferente al de un índice de jerarquía de clases (class-hierarchy index), como se muestra en la figura 4.6a y 4.6b. Un registro índice en un nodo hoja de un índice de una única clase (single-class index) consta de longitud de registro, longitud de la llave, valor de la llave, un apuntador al desbordamiento de página, el número de elementos en la lista de IDOs[18] de los objetos que tienen el valor de la llave en el atributo indexado, y la lista de IDOs.

a)



b)

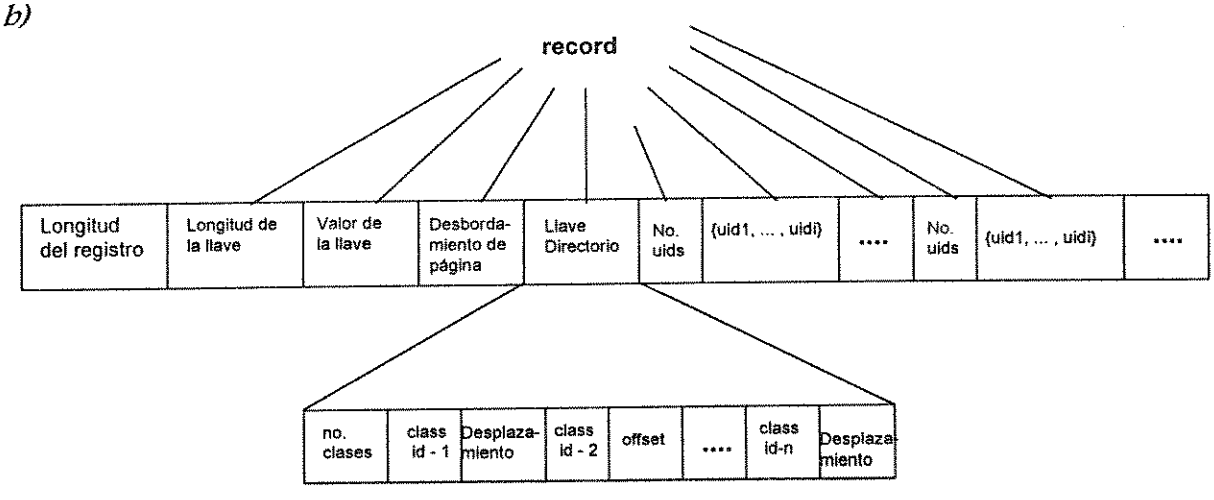


Figura 4.6. (a) Un nodo hoja de un single-class index. (b) Un nodo hoja de un class-hierarchy index.

[18]Referenciados como UIDs (Identificadores únicos) en la figura.

Un registro índice en un nodo hoja de un índice de jerarquía de clases (class-hierarchy index) consta de: longitud del registro, longitud de la llave, valor de la llave, un apuntador de desbordamiento de página, llave-directorio, y, para cada clase en la jerarquía de clases, el número de elementos en la lista de IDOs para los objetos que tienen el valor de la llave en el atributo indexado, y la lista de IDOs. La llave-directorio consta del número de clases que contienen objetos con el valor de la llave (llave-valor) en el atributo indexado, y, para cada clase un identificador de clase y el offset (desplazamiento) en el registro índice en el cual se encuentra la lista de IDOs de los objetos. El nodo hoja de un índice de jerarquía de clases (class-hierarchy index) agrupa la lista de IDOs para un valor de la llave desde el punto de vista de las clases a la que ellos pertenecen.

El razonamiento que se sigue para esta organización es que un índice de jerarquía de clases (class-hierarchy index) se mantiene sobre un atributo para una jerarquía de clases que consta de n clases arraigadas a una clase C, y que el índice se puede necesitar frecuentemente para ser usado por una consulta que se dirige a una subclase de la clase C. Si el nodo hoja se organiza como en un índice de una única clase (single-class index), y revisa exhaustivamente la lista entera de IDOs para un valor de la llave es necesario resguardar los IDOs que no pertenecen a las clases pertinentes a una consulta. Adicionalmente, si una clase en la jerarquía de clases se baja, los IDOs de las instancias para la clase deben ser eliminadas desde el índice de jerarquía de clases (class-hierarchy index); la organización mostrada en la figura 4.6.b facilita la eliminación de una lista de IDOs para cualquier clase sobre una jerarquía de clases.

Una registro índice nodo-hoja puede ser pequeño (no más grande que el tamaño de un índice página) o grande (más grande que el tamaño del índice de página). Un registro índice pequeño puede crecer a un registro grande índice o simplemente crecer fuera de los límites de su página índice actual. Hay maneras a seguir para estas situaciones de desborde con un nodo hoja. El enfoque que generalmente se adopta es el que se indica a continuación. Por un lado, si un registro de índice pequeño crece fuera de los límites de su página índice, pero permanece siendo un registro pequeño, la página índice es como una fisura. Por otra parte, si un registro índice llega a ser un registro grande, le es asignado un nodo hoja entero, y la parte del registro que aún no se adapta en el nodo se almacena en el desborde de página(s). Este es el uso del campo apuntador de desborde (overflow) de página en un registro índice nodo-hoja; si el valor de este campo es cero, puede presumirse que el registro índice está totalmente contenido en el actual índice de página.

4.8 Tamaños de los índices

El tamaño de un índice es el número total de nodos del índice, donde cada nodo ocupa una página física en el almacenamiento secundario. Existen límites para la altura de un B-tree y el número de nodos en cada nivel del B-tree, dado el orden y el número de llaves; con estos límites se formulan modelos de costos para un índice de jerarquía de clases y su correspondiente conjunto de índices de clases únicas (single-class indexes), en base a estos modelos de costos muchos autores derivan sus propias fórmulas y realizan sus respectivos experimentos de simulación las cuales no se presentan en este capítulo ya que se considera únicamente el estudio de la arquitectura interna.

4.9 Control de concurrencia y bases de datos orientadas a objetos

El procesamiento de transacciones es el encargado de controlar la forma en que los programas comparten una base de datos común. Una transacción normalmente se conoce como la unidad de concurrencia y como la unidad de recuperación para las base de datos. Como una unidad de concurrencia, los pasos de varias transacciones pueden intercalarse y ellos no se inmiscuan el uno con el otro. Como una unidad de recuperación, una transacción o sucede totalmente o no tiene efecto sobre la base de datos. El sistema puede recuperarse siempre de una transacción no completa, de tal forma que los resultados de una transacción parcialmente completa no son visibles. Compartir también es importante para las bases de datos orientadas a objetos. En esta sección se describen varias maneras de cómo es visto el compartimiento en una base de datos orientada a objetos las cuales pueden ser diferentes del punto de vista tradicional. Algunos de estos conceptos son muy recientes y hasta ahora no están al mismo nivel de la teoría que existe para el tradicional procesamiento de transacciones. No obstante, es una nueva dirección importante en la investigación de los sistemas de base de datos. No se quiere dar a entender que esto sea aceptado por completo, sino más bien se intenta que sea algo representativo del área.

La siguiente sub-sección introduce los conceptos tradicionales del procesamiento de transacciones, y a continuación se discute como estos podrían ser diferentes para las bases de datos orientadas a objetos. Luego se detalla la noción de tipo - específico en control de concurrencia, y por último se describe un nuevo modelo de control de concurrencia para ambientes de trabajo en grupo (cooperadores).

4.10 El Procesamiento de transacciones

Las transacciones son programas que típicamente se desarrollan independientemente uno del otro. Por razones de eficiencia, es deseable permitir al sistema de base de datos intercalar los pasos de transacciones concurrentemente ejecutables. Sin embargo, ya que estos pasos se escriben sin el conocimiento el uno del otro, es posible que se den intercalamientos que produzcan resultados insospechados. El intercalamiento no puede, por lo tanto, ser hecho indiscriminadamente. Debe haber algún criterio que nos permita rechazar algunos intercalamientos mientras que otros sean aceptados. En base a este criterio, un sistema de procesamiento de transacciones podrá sincronizar ejecuciones de transacciones tal que no puedan ocurrir intercalamientos incorrectos.

El concepto de transacción, como aparece en muchos sistemas de base de datos, frecuentemente combina varias nociones importantes incluyendo:

- Visibilidad
- Recuperación
- Consistencia

Visibilidad se refiere a la capacidad de una transacción para ver los resultados de otra transacción mientras se está ejecutando. Recuperación es la capacidad del sistema, cuando hay un fracaso, para recobrar el estado que se considera correcto. Consistencia se refiere a la corrección del estado de la base de datos que una transacción comprometida o completa (commit) produce.

Atomicidad también se discute como una característica de las transacciones. Por atomicidad entendemos que, como se ha visto desde la perspectiva de otras transacciones en el sistema, cualquier transacción determinada, o sucede en su totalidad, o no sucede nada . Reexaminando el concepto de transacción para aplicaciones avanzadas, tales como ambientes de diseño, se necesita considerar cada de los tres conceptos arriba mencionados y así poder decidir si éstos juntos deberían dirigirse uniformemente por un mecanismo único.

Si se escoge imponer atomicidad para toda ejecución de transacciones permitidas por el sistema, entonces limitamos las maneras en que podemos enfocar la visibilidad, la recuperación, y la consistencia. Si esperamos solo transacciones atómicas, entonces los resultados intermedios de una transacción no pueden hacerse visibles a otra transacción. También, atomicidad requiere que, cuando ocurre algún fracaso (en la ejecución de una transacción), debe ser posible dar marcha atrás (roll-back) a cualquier transacción parcialmente completa. La consistencia requiere que cualquier regla (constraint) de integridad C sobre una base de datos d sea satisfecha al término de la transacción. C(d) debe satisfacerse antes de que una transacción comience a ejecutarse así como también después de completarse. Dentro de la transacción C(d) puede infringirse, pero al término debe ser restablecida cuando la transacción queda completa.

Una extensión de la teoría tradicional de transacciones la podemos representar como un modelo que permita incluir transacciones dentro de transacciones. A este tipo de el modelo de transacciones se le conoce como transacciones anidadas. Una estructura anidada de transacción consiste en un conjunto de pequeñas transacciones que se ejecutan atomicamente con respecto a su padre y a sus hermanos. Una transacción padre no ve ninguna estructura interna de sus hijos. Si una transacción hijo fracasa o aborta, sus resultados se eliminan así como en las transacciones no anidadas.

El modelo de transacciones anidado permite explotar el paralelismo de transacciones que puede ocurrir naturalmente dentro de sí mismos. Uno de los principales beneficios del paralelismo de transacciones son los niveles de abstracción que se crean.

4.11 Sistemas orientados a objetos

Superficialmente, control de concurrencia en las bases de datos orientadas a objetos se parece al de las bases de datos tradicionales. Las transacciones son programas que hacen requerimientos a la base de datos, y por lo menos para algunas aplicaciones, serializabilidad[19] parece apropiado como un criterio de corrección. Podemos imaginar que en una base de datos orientada a objetos, un objeto es la unidad de interés para el esquema de bloqueo (lock). Las transacciones bloquean objetos de una manera que consta de dos fases, y que produce resultados de serializabilidad. Todavía, hay por lo menos dos áreas en las que la naturaleza de las bases de datos orientadas a objetos ó las formas en que pueden ser utilizadas podrían influenciar el enfoque de control de concurrencia:

- usar la semántica tipo-nivel para lograr mayor concurrencia
- usar la semántica tipo-nivel para permitir un comportamiento no serializable

En el resto de esta subsección se discuten cada una de estas posibilidades. Las bases de datos orientadas a objetos pueden distinguirse de sus predecesoras en que incorporan un extenso sistema de tipos. Los usuarios pueden extender los tipos del sistema para incluir nuevos tipos que son indistinguinbles los tipos empotrados[20]. Esta extensión es posible porque el sistema implementa la noción de abstracción de datos. Una abstracción de tipos de datos se caracteriza por un conjunto de operaciones que pueden ser usadas para acesar y manipular sus

[19]Serializabilidad define la corrección en términos de los resultados que se obtendrían si no hubiera intercalamiento. Por ejemplo: Una ejecución de un conjunto de transacciones, T={T1,...,Tn}, lo podemos expresar como H(T). H(T) es serial cuando, para cualquieras dos transacciones Ti y Tj en T, todas las operaciones de Ti aparecen antes de las de Tj o todas las operaciones de Tj aparecen antes de las de Ti. H(T) se dice que es serial si el resultado del intercalamiento de las transacciones es equivalente al resultado de alguna ejecución serial de T. Esto es concurrencia.

[20]Los tipos empotrados corresponden corresponden a las clases (o tipos) primitivas p. ej. string, integer, etc.

instancias. Las operaciones son el único camino por el cual otros programas pueden interactuar con las instancias del tipo.

La estricta interface operacional definida por los tipos de datos abstractos es importante, desde los sistemas anteriores de base de datos era posible interactuar con instancias de un tipo en términos o combinaciones arbitrarias de operaciones sobre la representación de ese tipo. Las representaciones que típicamente estaban disponibles incluían tipos básicos estructurados como registros, conjuntos, y archivos. Con este nivel de abstracción, lo más que podíamos hablar de era de lectura o escritura de objetos o de componentes de objetos.

Ilustraremos esta distinción con un ejemplo. Suponga que tenemos una base de datos de empleados. Podríamos representar un conjunto de empleados por un tipo de relación llamada Empleados que contiene registros de empleado con los campos emp#, nombre, y sueldo. Los programas que interactúan con Empleados pueden leer o escribir los campos de estas relaciones de manera arbitraria. Esto significa que, en cuanto al sistema de base de datos, las operaciones de leer y escribir son el nivel de abstracción en que las aplicaciones externas interactúan. Por lo tanto, la teoría de serializabilidad ha producido algoritmos que son la tendencia desde el punto de vista de la semántica de lectura y escritura. Esto es, bloqueando protocolos involucrando bloqueos especiales para leer y escribir con base en la manera en que estas dos operaciones interfieren una con la otra.

Por otra parte, los sistemas orientados a objetos proveen nuevos tipos con operaciones que son típicamente de un nivel más alto que las de lectura y escritura. Las operaciones podrían actuar de maneras que son bastantes diferentes del estándar de lectura y escritura. De hecho, usando la semántica de las operaciones que son provistas por un tipo, podemos lograr más concurrencia que por un simple bloqueo en las operaciones de lectura y escritura. Esto nos conduce a una diferencia crucial en donde son posibles técnicas de control de concurrencia para bases de datos orientadas a objetos.

Una segunda forma en la que puede diferir el control de concurrencia para las bases de datos orientadas a objetos deriva de los tipos de aplicaciones a las que estas conducen su desarrollo. En la actualidad una de las áreas más prometedoras de aplicaciones para las bases de datos orientadas a objetos es en ambientes de diseño, incluyendo ambientes de programación, herramientas CAD eléctricas y mecánicas, y office information systems. En todas estas áreas, encontramos requerimientos para trabajo en grupos (cooperativo). Esto es, debemos ser capaces de apoyar programas que ayuden o asistan a múltiples gentes a lograr una meta común. Ellos trabajan juntos para alcanzar este fin, y la computadora debe hacer la acción de información a lo largo de este proceso tan naturalmente como sea posible.

El diseño en grupo (colaborador) argumenta un mecanismo que cede algunas de las limitaciones de un mundo estrictamente consecutivo. Podemos ceder

serializabilidad para permitir comunicación de manera arbitraria entre transacciones; sin embargo, perdemos nuestra capacidad previa muy conveniente que argumentábamos sobre la corrección de transacciones. Si las transacciones pueden interactuar de maneras caprichosas, llega a ser difícil comprender cómo una mezcla concurrente de transacciones podría afectar a la base de datos. Una pregunta importante entonces es: cómo podemos permitir la clase de compartimiento que necesitamos y todavía conservar alguna noción de corrección? Un enfoque que parece prometedor involucra la extensión del trabajo de control de concurrencia tipo-específico que permita para la aplicación de criterios específicos de corrección que son la tendencia desde el punto de vista de sucesiones permisibles de operaciones.

5. DESARROLLO E IMPLEMENTACION DE UN DBMS ORIENTADO A OBJETOS

5.1 Introducción

El sistema de base de datos GemStone es el resultado del desarrollo de un proyecto que se hizo hace algunos años en Servio. GemStone combina conceptos de un lenguaje orientado a objetos con los de los sistemas de base de datos, y proporciona un lenguaje de base de datos orientado a objetos llamado OPAL el cual es usado para definición de datos, manipulación de datos y computación en general.

Los sistemas de bases de datos convencionales frecuentemente reducen el tiempo de desarrollo de aplicaciones y mejoran el compartimiento de datos entre aplicaciones. Sin embargo, estos DBMSs están sujetos a las limitaciones de un conjunto finito de tipos de datos y a la necesidad de normalizar los datos. En contraste, los lenguajes orientados a objetos ofrecen facilidades de una flexible escritura de datos abstractos y la capacidad de encapsular datos y operaciones por medio del uso de mensajes.

La premisa de este proyecto era que al combinar las capacidades del lenguaje orientado a objetos con las funciones de la administración del almacenamiento de un sistema tradicional de gestión de datos resultaría un sistema que reduciría los esfuerzos en el desarrollo de una aplicación e incrementaría el poder del modelo. Además, se cree que un lenguaje orientado a objetos es suficiente para manejar el diseño de la base de datos, el acceso a la base de datos, y las aplicaciones. Los objetos como modelos han sido populares por un largo tiempo en CAD y han dado buenos resultados en ambientes de programación, sistemas hypermedia, bases de conocimiento, y sistemas de información de oficina. Los otros grupos están en el proceso de implementar el modelo de sistemas de base de datos orientadas a objetos.

En la siguiente sección, se discute la elección de Smalltalk como un lenguaje de administración de una base de datos. Seguidamente, brevemente se introduce el modelo GemStone. Luego, se discuten las extensiones que Smalltalk necesita en un DBMS. Seguido de ello se muestra un enfoque para proveer estas extensiones. Y por último, se examina el indexamiento en GemStone.

5.2 La elección de smalltalk

Inicialmente, se desarrollo un lenguaje de consultas.. Sin embargo, se sintió deficiencia en capacidades procedurales necesarias en el modelo para el

comportamiento de entidades del mundo real. Debido a los problemas vinculados en proveer extensiones procedimentales y el adiestramiento que había que dar a los usuarios para que conocieran un lenguaje completamente nuevo, se decidió utilizar un lenguaje orientado a objetos ya existente, Smalltalk-80, como la base del desarrollo del producto. Más adelante, se discuten esas características de Smalltalk que condujeron a esta elección.

Otro lenguaje orientado a objetos y sistemas de programación soportan muchas de las mismas características de Smalltalk. Algunas de las principales razones por las que se eligió Smalltalk sobre otras alternativas son:

1. La literatura disponible: el cuerpo de textos y documentos, particularmente artículos introductorios, es más grande para Smalltalk que para las otras alternativas.
2. El modelo de máquina virtual: las implementaciones previas sobre Smalltalk han producido una abstracción conceptual llamada máquina virtual que provee una buena estructura para la implementación. En particular, el concepto de máquina virtual hace una diferencia entre la memoria de objetos, que almacena objetos y provee acceso estructural, y el intérprete, el cual comprende la semántica del modelo de datos y el procedimiento de evaluación.
3. Esquema/distinción de instancias: algunos de los sistemas orientados a objetos no distinguen estrictamente entre el "subtipo de" y la "instancia de". Smalltalk lo hace. Se consideró que la distinción era importante en el mundo de las bases de datos, donde programadores necesitan una distinción clara entre el esquema y los datos.

5.3 Identidad de los objetos

Smalltalk soporta la identidad de objetos. Un objeto mantiene su identidad aún cuando hayan cambios arbitrarios en su estado propio. Las entidades con información en común pueden modelarse como dos objetos con un subobjeto compartido conteniendo la información común. Tal compartimiento reduce las "anomalías de actualización" que existen en el modelo de datos relacional. En el modelo relacional, las propiedades de una entidad deben ser suficientes para distinguirla de otras entidades. En el orden en que una entidad haga referencia a otras, debe haber algunos campos que unicamente identifican a la otra entidad. Usar nombres de departamento para identificar tuplas de departamento está bien hasta que departamento cambie de nombre. Constituyendo números únicos de departamento agrega carga al desarrollador de la aplicación e introduce algunas cosas dentro del esquema de la base de datos que no estaban presentes en el modelo. Algunas extensiones al modelo relacional incorporan formas de identidad.

5.4 Identificando Fortalezas

Smalltalk soporta directamente el modelo de objetos complejos y relaciones, y organiza clases de objetos en una jerarquía de herencia. Una entidad única es modelada como un objeto único, no como múltiples tuplas en múltiples relaciones. Las propiedades de las entidades no necesitan ser los valores simples de los datos, pero pueden ser otras entidades de complejidad arbitraria. El componente dirección de domicilio de un objeto empleado no necesita ser simplemente un texto string: en Smalltalk puede ser un objeto estructurado, por sí mismo puede tener componentes como el número de calle, calle y ciudad, y su comportamiento propio definido (Figura 5.1). Smalltalk directamente soporta un conjunto valuado de entidades sin el código requerido en el modelo relacional. Además, en los conjuntos pueden haber objetos arbitrarios como elementos y no necesitan ser homogéneos. Smalltalk nos provee de la independencia física de datos de bases de datos relacionales sin sus limitaciones en el modelo.

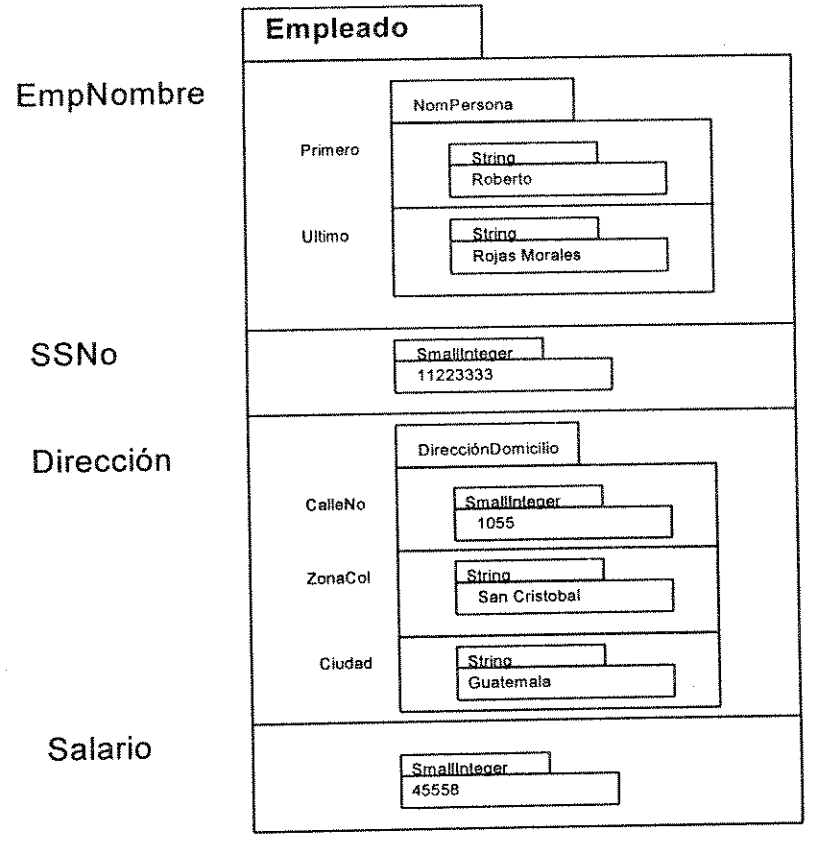


Figura 5.1. Un Objeto empleado

5.5 El comportamiento de los objetos

Smalltalk soporta el modelo del *"comportamiento de entidades del mundo real"*, no solamente su estructura. Los comandos de manipulación de datos en los sistemas convencionales se orientan hacia las representaciones de máquina: "modifique campo", "inserte tupla", "consiga el próximo". Para un sistema de gestión de oficina, varias aplicaciones podrían reservar una sala. En un sistema convencional de base de datos, cada aplicación podría contener instrucciones para probar la disponibilidad de sala, la inserción o cambio de un registro indicaría la reservación, y quizás crearían otro registro para el calendario de citas de la reservación. Los cambios en la estructura de la base de datos o la política para reservar salas puede requerir de ubicación y modificación en todas las aplicaciones que hacen uso de la base de datos. En Smalltalk, por otra parte, uno puede definir un mensaje ReserveSala que toma una fecha y tiempo como parámetros y desempeña todas las actualizaciones y chequeos necesarios a la base de datos para reservar una sala.

Por supuesto, en un sistema convencional se podría factorizar esta funcionalidad en un procedimiento separado que múltiples programas de la aplicación podrían usar. Sin embargo, ninguna de esas aplicaciones se fuerza a usar el procedimiento para acceder los datos de reserva de sala. En Smalltalk, el estado de un objeto es accesible sólo a través de su interface de mensajes. Así, el diseñador de una clase controla el acceso y uso de los datos en los objetos de esa clase. Smalltalk tiene la ventaja adicional que organiza los mensajes conjuntamente con la descripción estructural de los objetos. En sistemas convencionales, un procedimiento común podría típicamente residir en un sistema de archivos externo a la base de datos, más que ser una parte de la base de datos.

El procedimiento o método de Smalltalk que implementa un mensaje puede ejecutar cualquier número de actualizaciones y consultas en la base de datos, con muchas ventajas. Las aplicaciones son más concisas: enviar un mensaje toma el lugar de muchas operaciones de la base de datos. El código es más confiable, como cada aplicación que reserva una sala usa exactamente el mismo procedimiento -el método asociado con el mensaje ReserveSala. Si la estructura de la base de datos o la política de reservar salas cambia, modificando un único método para ReserveSala significa que todas las aplicaciones que hagan reservaciones habrán cambiado el comportamiento. El enlace del mensaje al método ocurre en tiempo de ejecución en Smalltalk, así que las aplicaciones aplicaciones no necesitan recompilarse para tener disponible el comportamiento que ha cambiado. Adicionalmente, los mensajes pueden proteger la integridad de la base de datos por la consistencia que registra en sus métodos. El mensaje ReserveSala puede hacer los chequeos para evitar una doble reservación de salas. Las aplicaciones no pueden modificar los datos de reservación de salas directamente, si se pudiera hacer dicha modificación podría conducir a una planilla inconsistente de reservaciones si ellos no incluyeran los chequeos que ReserveSala hace.

5.6 Clases

La estructura de clases de Smalltalk hace lo posible para producir nuevos tipos de datos y ayuda a organizar esquemas de las base de datos. Smalltalk viene con un complemento grande que son las clases, que implementan los tipos de datos frecuentemente usados, que los diseñadores de bases de datos pueden usarlos para representar el estado interno de los objetos que ellos definen. Las definiciones de clases son análogas a los esquemas de los sistemas de base de datos, pero las clases también operan con la estructura y encapsulan el comportamiento. Smalltalk parcialmente ordena clases en una jerarquía de herencia. Considerando que una clase ayuda a organizar datos, la jerarquía de clases ayuda a organizar clases. El mecanismo de herencia de subclases permite un esquema de base de datos para capturar similitudes entre diversas clases de entidades que no son totalmente idénticas en estructura o en comportamiento. Las subclases también proveen medios para manejar casos especiales sin atropellar la definición (hecha más arriba) del caso normal.

5.7 Asociando tipos con objetos

A diferencia de la mayoría de los lenguajes de programación que soportan tipos abstractos de datos, Smalltalk asocia tipos con valores y no tipos con variables (ver capítulo 2).

En particular, el tipo de un campo puede inicialmente ser sólo un objeto, la clase más general, lo que significa que no hay limite en los valores que ocupan el campo. Conforme el análisis crece, más y más tipos específicos pueden asociarse con el campo, para integridad o eficiencia. Otra característica importante es que el detectar mensajes enviados a objetos donde no hay un método para el mensaje indica que hay partes en donde el modelado de datos está incompleto.

5.8 Un lenguaje unificado

Smalltalk es mucho más poderoso que los lenguajes estándar de manipulación de datos. Es un lenguaje de programación de bases de datos, y no simplemente un lenguaje de manipulación de datos. Es computacionalmente completo y soporta fácilmente casi todo el cómputo requerido en una aplicación.

5.9 El modelo GemStone

Esta sección plantea el modelo de datos GemStone y el lenguaje de programación OPAL.

Los tres conceptos principales del modelo GemStone y del lenguaje son: *objeto, mensaje, y clase.* Que corresponden a registro, llamado a un procedimiento, y el tipo de registro en sistemas convencionales (ver Figura 5.2 para otras correspondencias). Un objeto es un pedazo de memoria privada con una interface pública. Los objetos se comunican con otros objetos pasándose mensajes, que son los pedidos para el objeto receptor para cambiar su estado o devolver un resultado. El conjunto de mensajes a los que un objeto responde se llama: su protocolo (su "interface pública"). Un objeto puede ser inspeccionado o modificado únicamente mediante su protocolo. El medio por el que un objeto responde a un mensaje es un método, un procedimiento de OPAL que es invocado cuando un objeto recibe un mensaje particular. Para que cada objeto no necesite acarriar con sus propios métodos, los objetos con los mismos métodos y estructura interna se agrupan juntos en una clase y se llaman instancias de la clase. Los métodos y la estructura están en un objeto único que describe la clase, la definición de clases del objeto o CDO[21], y todas las instancias de la clase contienen una referencia a CDO. Diferente a algunos otros modelos orientados a objetos, un objeto es una instancia de específicamente una clase, aunque que pueda heredar comportamiento desde las superclases de su clase.

EQUIVALENCIAS APROXIMADAS

GemStone	Convencional
objeto	intancia registro, instancia conjunto
variable instancia	campo, atributo
variable instancia constraint	tipo del campo (field type), dominio
mensaje	llamada a procedimiento (procedure call)
método	cuerpo del procedimiento
definición de clases del objeto (class-definig object)	tipo de registro (record type), esquema de relación (relation scheme)
jerarquía de clases	esquema de la base de datos
instancia de clase (class instance)	instancia de registro (record instance), tupla
clase colección (collection class)	conjunto, relación

Figura 5.2. La correspondencia entre bases de datos orientadas a objetos y bases de datos convencionales.

[21] CDO son las siglas en inglés de class-defining object.

5.9.1 Objetos

Internamente, la mayoría de los objetos se dividen en campos llamados variables de instancia. Cada variable de instancia puede tener un valor, que es otro objeto. La figura 5.1 muestra un objeto empleado con cuatro variables de instancia: empName, ssNo, dirección, y sueldo. La variable de instancia empName tiene un objeto que es una instancia de la clase NomPersona. No todos los objetos se dividen internamente en variables de instancia. Ciertos tipos básicos tal como SmallInteger y el Character no tienen ninguna descomposición adicional. Las variables de instancia en el objeto Empleado son llamadas *variables de instancia nombradas*. Un objeto puede tener *variables de instancia indexadas*, que pueden ser vistas como variables de instancia con números en lugar de nombres. Las variables de instancia indexadas son usadas principalmente para implementación de colecciones ordenadas tales como los arreglos.

5.9.2 Mensajes

La forma básica de todos los mensajes es *<receptor> <mensaje>*. La parte <receptor> es un identificador o una expresión denotada por un objeto que recibe e interpreta el mensaje. La parte <mensaje> da el selector[21] del mensaje y posiblemente argumentos al mensaje. Cada mensaje devuelve un resultado al remitente, que es comúnmente otro objeto. Hay tres tipos de mensajes: unario, binario y de palabra reservada (keyword).

Los mensajes unarios no tienen argumentos y tienen selectores que son un único identificador. Asuma que emp es un variable que contiene un objeto Empleado. Si tenemos un mensaje unario PrimerNombre para recuperar el primer nombre del empleado, entonces

emp FirstName

devuelve un string que es el primer nombre de emp. Las expresiones binarias de mensajes tienen un receptor, un argumento, y un selector de mensaje que es uno o dos caracteres no alfanuméricos. Al multiplicar 8 por 3, nosotros enviamos a 8 el mensaje "Multipliquese usted mismo por 3":

*8 * 3*

aquí * es el selector binario para el mensaje de multiplicación. Las comparaciones también se manejan con mensajes binarios. En

(emp1 sueldo) < = (emp2 sueldo)

[21] Selector desde el punto de vista que escoge o selecciona el mensaje.

el receptor y el argumento del mensaje binario <= son resultados de expresiones unarias. Los mensajes de palabra reservada o Keyword tienen uno o más argumentos y tienen selectores multiparte compuestos de caracteres alfanuméricos y colones (colons). Al asignar 'Rojas' como tercer componente de un arreglo anArray , usamos

anArray at: 3 put: 'Rojas',

que es lo mismo que si en otro lenguaje se hiciera

anArray [3]: ='Rojas',

aquí el selector de mensaje tiene dos partes, at: y put:, como toma dos argumentos, el índice del arreglo y el objeto para que sean almacenados en ese índice. Nos referimos al mensaje por la concatenación de sus partes: at: put.

5.9.3 Métodos

Al construir un método se necesita saber qué objetos son visibles dentro de su alcance. Todas las variables de instancia nombradas del receptor están disponibles por medio de sus nombres. Así, en un método para la clase empleado, como se definió en la Figura 5.1, los variables de instancia empNombre, ssNo, dirección y salario son accesibles. Un método también puede tener variables temporales, que se declaran entre barras verticales al principio del método: |temp1 temp2|. Cada usuario tiene uno o más diccionarios de variables globales, y esas variables globales pueden aparecer en los métodos. Hay otros dos operadores que necesitamos antes de poder escribir métodos: el operador de asignación, :=, y el operador ^ que devuelve el valor de la expresión como el resultado de un método:

Para un mensaje unario nombrecompleto en la clase NomPersona que devuelve el primer y último nombre concatenados con un espacio entre ellos, podemos usar el siguiente método:

```
NombreCompleto
    |temp|
    temp := primero.
    temp := temp + ' ' + último.
    ^temp
```

La primera instrucción, *temp := primero*, asigna el valor de la variable de instancia primero del receptor (un objeto NomPersona) a la variable temporal temp. La instrucción

temp: = temp + ' ' + último

concatena al valor de temp, un blanco, y el contenido de la variable de instancia último del receptor (+ es el mensaje binario para concatenación de strings). El resultado de la concatenación se asigna a temp. Finalmente, la instrucción *^temp* devuelve el valor de temp como el resultado del mensaje NombreCompleto. (Este mensaje en particular podría implementarse con la única expresión de mensaje:

^ primero + ' ' + último

con ninguna necesidad de una variable temporal). Un método también puede cambiar el estado del receptor, asignando nuevos valores a las variables de instancia. Suponga que objetos de la clase departamento tienen una variable de instancia presupuestario. El siguiente método aumenta el presupuesto de un departamento en un cierto porcentaje, de hasta un límite:

```
IncrementePresupuestoEn: unPorcentaje upTo: unLímite
        presupuesto := presupuesto + (presupuesto * (unPorcentaje / 100)).
        (presupuesto > unLímite) ifTrue: [ presupuesto := unLímite].
        ^presupuesto
```

El mensaje *IncrementePresupuestoEn: upTo:* toma dos argumentos representados por las variables unPorcentaje y unLímite. Este método cambia la variable de instancia presupuestario del objeto Departamento que recibe el mensaje *IncrementePresupuestoEn: upTo:*. En la segunda línea del método vemos un mensaje palabra reservada (keyword) ifTrue: que funciones como un constructor de control condicional. El receptor de este mensaje es un valor Boolean. El argumento del mensaje es un bloque[23]. Un bloque es una sucesión de una o más instrucciones de OPAL dentro de paréntesis, y es un objeto de primera - clase en GemStone. El efecto de un mensaje aBoolean ifTrue: aBlock está al desempeñar el código en aBlock si aBoolean es verdadero. La tercera línea del método devuelve el nuevo valor de presupuesto.

OPAL incluye mensajes para estructuras de control iterativas, usando bloques con argumentos. Los argumentos de bloque se declaran al principio de un bloque. Por ejemplo,

*[: n | (2 * n) - 1]*

es un bloque con un argumento, n. Este bloque para ser evaluado, necesita un valor para el argumento. El mensaje valor: provee el argumento y causa su ejecuón. Un bloque devuelve el valor de su última expresión cuando es ejecutado. Así,

[23] Más comunmente se le conoce por su nombre en inglés "block".

*[: n | (2 * n) - 1] valor: 7*

devuelve 13. En general, dado un valor de N, este bloque devuelve el *N*avo. número impar.

Todos los métodos que hemos visto están lejos de haberse definido en términos de otros mensajes. Las definiciones de métodos no son completamente circulares, sin embargo: en el fondo todo es un puñado de métodos primitivos. Cuando el intérprete de OPAL encuentra un mensaje que tiene un método primitivo, ejecuta un pedazo de código de máquina más bien que un método de OPAL. Los métodos primitivos existen para la aritmética, comparaciones, la creación y copiado de un objeto, arreglos, manipulación de strings, y funciones de conjunto (sets). El conjunto de métodos primitivos en GemStone no puede ser aumentado por un programador GemStone, aunque se puede proveer tal extensión.

5.9.4 Clases

Cada clase es representada por una CDO que describe la estructura y comportamiento de las instancias de la clase, así como también la posición de la clase en la jerarquía de clases. Cualquier objeto devuelve la CDO para su clase con respecto al mensaje *class*. Suponga que la variable *asoc* contiene un objeto de la clase Asociación (una Asociación es un par llave-valor usado para construir diccionarios). La asignación

ac := asoc class

causa que a *ac* sea asignada la CDO para la clase Asociación. Las CDOs responden a mensajes así como todos los demás objetos lo hacen. Por ejemplo, las CDOs responden al mensaje *nombre*. El resultado de la expresión

asoc class nombre

es # Asociación. (Los símbolos son indicados por el prefijo #).

Los nombres de las variables de instancia y los métodos de las instancias de una clase se almacenan en la CDO de la clase. Las CDOs también almacenan los métodos que las instancias ejecutan con respecto a diversos mensajes. Cuando un objeto recibe un mensaje, consulta la CDO para averiguar como ejecutar ese mensaje (ver Figura 5.3).

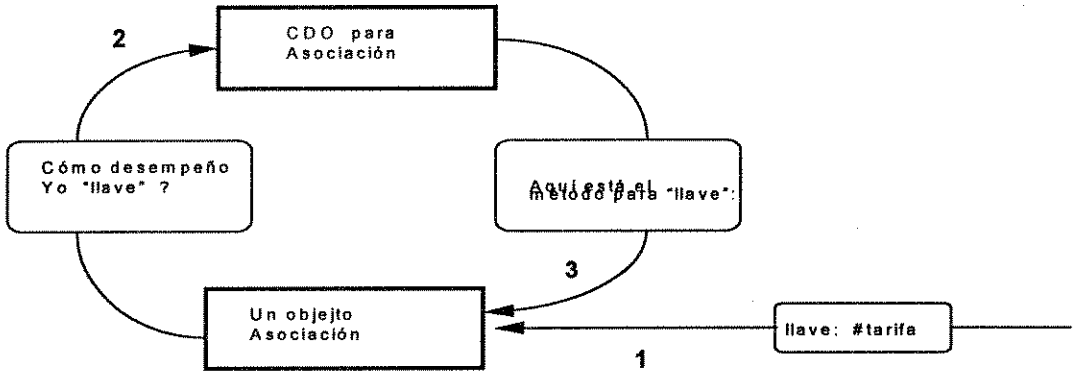


Figura 5.3. Determinando el método para el mensaje.

OPAL provee una jerarquía de clases para explotar similitudes en la estructura y comportamiento de entidades. Una subclase hereda estructura y comportamiento desde su superclase. La estructura se hereda en que todas las variables de instancia nombradas en la superclase están también presentes en cualquier subclase. Suponga que queremos objetos que representen nombres persona con títulos. Podemos crear una subclase nombrecontítulo de nompersona. Las instancias de nombrecontítulo automáticamente tienen variables de instancia *primero* y *último*. Agregamos un variable de instancia, título, a nombrenontítulo para retener el título (ver Figura 5.4).

Una subclase hereda métodos de su superclase. Así, si nombrecompleto es un mensaje para NomPersona, instancias de NombreConTítulo responden a ese mensaje debido al método en NomPersona. El proceso de mirar hacia arriba determina que método corresponde al mensaje en una clase de objeto. Si el mensaje no esta definido en esa clase, se procede a buscar en la superclase de la clase, la superclase de esa clase, y así sucesivamente.

Clases	**Instancias**
Nompersona	'Roberto Rojas'
Nombrecontítulo	'Dr. Roberto Rojas'
Nombrecontítuloyletras	'Dr. Roberto Rojas, OBE'

Clase	**Variables de Instancia**	**Mensajes**
NomPersona	primero último	primero primero: último último: nombrecompleto
NombreConTítulo	(anterior) +	(anterior) +

	título	nombrecontítulo
NombreConTítuloYL etras	(anterior) + letras	(anterior) + nombrecontítulo (nuevo método)

Figura 5.4. Una porción de la jerarquía de clases.

Una subclase puede implementar mensajes propios. Para nombrecontítulo se podría querer un mensaje que devuelva un string que contenga el nombre completo con el título:

```
título:
    NombreConTítulo
        ^ título + ' ' + self NombreCompleto
```

Una nueva característica en este ejemplo es *self* (su mismo, propio), una variable especial cuyo valor es siempre el receptor del mensaje. Una subclase puede reescribir o reimplementar un mensaje heredado. Suponga que definimos una clase TítuloConNombreYLetras como una subclase de NombreConTítulo. Esta subclase agrega la variable de instancia Letras que tiene letras después de un nombre de persona, como en Dr. Roberto Rojas, OBE. Podemos reimplementar el mensaje NombreConTítulo en NombreConTítuloYLetras para letras después del nombre.

5.9.5 Convirtiendo Smalltalk en un DBMS

Smalltalk está implementado en modo de un único usuario, memory-based, y sistema de único procesador. No tiene los requerimientos para un sistema de base de datos. Mientras que Smalltalk provee de una poderosa interface con el usuario y muchas herramientas para el desarrollo de aplicaciones, está orientado a workstation de un único usuario. Para encontrar los requerimientos para un sistema de base de datos, se necesitan las siguientes extensiones.

5.9.6 Soporte para un ambiente multi-usuario, basado en disco

La base de datos debe ser muy inteligente para montar objetos entre el disco y la memoria. Debe tratar de agrupar objetos de tal forma de accesarlos juntos en las mismas páginas de disco, se intenta prever que los objetos en la memoria principal serán probablemente utilizados de nuevo muy pronto, y organizar el procesamiento de consultas para minimizar el tráfico de disco.

Desde que los datos en GemStone son compartidos por múltiples usuarios, el sistema debe proveer acceso concurrente. Cada usuario debería ver una versión

uniforme de la base de datos, a la par de otros usuarios que corren simultáneamente. Un requerimiento también es la gestión de espacios múltiples de nombre. Smalltalk asume un único usuario por imagen, así que provee un espacio único de nombre global. Es irrazonable esperar que los usuarios en GemStone compartan un espacio único de nombre global, o que esos espacios de nombre de usuario sean disjuntos.

Las actuales implementaciones en Smalltalk usan un procesador único para procesamiento de despliegue y administración de objetos. Esperamos que GemStone soporte múltiples aplicaciones interactivas. De aquí en adelante, no parece sabio usar el mismo procesador para la gestión secundaria de almacenamiento en lo que concierne al procesamiento de despliegue en la interface del usuario final.

5.9.7 Integridad de los datos

Los diversos tipos de fallas (programa, procesador, medios) y las violaciones (consistencia, acceso, mecanografiado) pueden comprometer la validez e integridad de una base de datos. Un sistema de base de datos debe ser capaz de arreglar las fallas restaurando la base de datos al estado uniforme, y debería prevenir violaciones a ocurrir.

Por "falla de programa" se entiende que un programa de aplicación puede fallar y no completar su ejecución, es decir, a causa de un error en tiempo de ejecución. Los sistemas de base de datos proveen múltiples modificaciones para ser desempeñadas mediante el uso de transacciones. Una transacción se usa para marcar una sección de procesamiento para que todos sus cambios sean hechos permanentes (commit transaction) o ninguno sea hecho permanente (the transaction aborts). Por "falla de procesador" entendemos que el procesador que maneja la gestión de almacenamiento de GemStone falla o fracasa. Por tales fracasos, la base de datos debe guardarse intacta. Recuperarse del fracaso del procesador y del programa implica que las copias del dueño de objetos sobre el almacenamiento secundario deben actualizarse cuidadosamente. Por "falla de medios" entendemos esos defectos de disco que pueden ocasionar que datos comprometidos se pierdan. Ninguna estrategia puede proveer protección completa contra la falla de medios. Se quiso que GemStone proveyera backups periódicos y replicación dinámica: se quiso guardar múltiples copias en línea de una base de datos, las cuales se actualizan en cada transacción.

Volviendo a las violaciones de la consistencia de la base de datos, ésta puede infringirse si la transacción hace sus actualizaciones intercalando múltiples usuarios. GemStone debe soportar serializabilidad de transacciones: el efecto neto de transacciones concurrentes sobre la base de datos debe ser equivalente a algunas ejecución consecutivo de esas transacciones. La integridad de una base de datos

puede infringirse también si un usuario accesa datos que no debería permitírsele ver. En Smalltalk, todos los objetos están disponibles al usuario. GemStone debe asignar privilegios de acceso y propiedad a cada objeto. Las limitaciones de integridad, tales como llaves únicas e integridad referencial (que es, referenciar objetos existentes), son las afirmaciones que a priori excluyen ciertos estados de la base de datos. Como mínimo, la base de datos debería soportar, limitaciones que requieren subpartes de una entidad o colecciones que pertenezcan a cierta clase. La integridad referencial viene "for free" en GemStone. Un objeto se refiere directamente a otro objeto, no al nombre de ese objeto. La referencia no puede crearse si el otro objeto no existe. GemStone no tiene un mecanismo explícito de eliminación. El espacio para un objeto es reclamado sólo si ningún otro objeto hace referencia a ese objeto. Así una vez existe una referencia, continuar es válido.

5.9.8 Espacio de grandes objetos

GemStone debe almacenar gran número de objetos y objetos de gran tamaño. Las primeras implementaciones en Smalltalk-80 tuvieron un límite de 2^{15} objetos, 2^{15} variables de instancia en cualquier objeto, y un total 2^{20} palabras de memoria de objeto. Los grandes objetos requieren nuevas técnicas almacenamiento. Algunos objetos son demasiado grandes para la memoria principal y deben paginarse. La memoria virtual implementa la paginación de grandes objetos.

En Smalltalk, para hacer "crecer" un objeto tal como un arreglo, se crea un nuevo objeto más grande y el contenido del objeto pequeño se copia dentro del él. Si el objeto a crecer es un Set o subclase de ello, entonces un remapeo de cada elemento del conjunto puede requerirse. Se quiere que el tiempo requerido para modificar o extender un objeto sea proporcional al tamaño de la modificación o extensión, no al tamaño del objeto que es actualizado. Un simple mapeo no es suficiente para manejar conjuntos del tamaño que un DBMS debe estar dispuesto a representar eficientemente. Las operaciones básicas de unión, intersección y diferencia llegan a ser prohibitivamente caras. Adicionalmente, Smalltalk mapea conjuntos de elementos por valor, no por identidad, y no provee de un mecanismo "trigger" para remapear un objeto en el conjunto(s) al cual pertenece cuando el objeto cambia. El que no exista un mecanismo trigger para cambiar la posición de un objeto dentro de una colección ordenada cuando el valor del objeto cambia en una dirección que altera su posición dentro de la colección. Por estas razones la representación del almacenamiento básico de Smalltalk fue inadecuado para soportar grandes colecciones. Esto es, GemStone necesita un tipo de almacenamiento básico para grandes colecciones, tanto en orden como en desorden.

La mayoría de las implementaciones en Smalltalk incluyen un *become*: un mensaje el cual intercambia la identidad de dos objetos. Por ejemplo, después de ejecutar A become: B, todas la referencias de A refieren a B y vice versa. Su

principal uso es el crecimiento de objetos, cambiando la clase de un objeto y modificaciones atómicas. Por ejemplo, para causar una modificación instantánea de un objeto desplegado C, un método en Smalltalk podría hacer una copia D de C, haciendo los cambios en D, al ejecutar C become: D. Los efectos son difíciles de anticipar y controlar. GemStone no soporta el mensaje become: pero usa otro mecanismos de soporte que lo hacen funcional. Instancias de clases colección pueden crecer y disminuir en GemStone como en Smalltalk. GemStone también soporta un protocolo para cambiar la clase de un objeto. Finalmente las transacciones en Smalltalk aseguran que los cambios desde una sesión siguen el principio de todo o nada con respecto a las otras sesiones.

Finalmente, buscar una colección grande por una búsqueda secuencial puede ocasionar un desempeño inaceptable en un objeto basado en disco. La búsqueda de ementos podría ser más logaritmica que lineal. Esto es, GemStone soporta acceso asociativo en elementos de grandes colecciones: esto podría suplir representaciones de almacenamiento y estructuras auxiliares para soportar localizar un elemento por su estado inteno.

Conjuntamente con el nivel de almacenamiento soportado para el acceso asociativo, OPAL debe tener construcciones de lenguaje que permitan acceso asociativo.

5.9.9 Administración del almacenamiento físico.

GemStone debe proveer aspectos para administrar la colocación física de objetos en el disco. Smalltalk es un sistema residente en memoria y no hay mucha necesidad de decir a donde va un objeto. El administrador de la base de datos, o un programador de aplicación con sentido común, debiera ser capaz de saber que en GemStone ciertos objetos frecuentemente se usan juntos y deben agruparse sobre el disco. El administrador debiera ser capaz de tomar objetos fuera de línea, dígase para archivar, y luego traerlos de regreso en línea.

5.9.10 Acceso desde otros sistemas

Mientras OPAL va más allá que los lenguajes convencionales de bases de datos en proveer un único lenguaje para la programación de aplicaciones de bases de datos, concentremos nuestros esfuerzos iniciales en la administración del almacenamiento usado más bien que en la interfaz con el usuario. Así, GemStone provee facilidades de acceso desde otros lenguajes de programación. Se quizo

apoyar un ambiente de desarrollo de aplicaciones para OPAL a lo largo de las líneas del ambiente de programación de Smalltalk, pero se reconoció que el ambiente de desarrollo de aplicaciones no puede ser igual al ambiente en cual la aplicación terminada corre. Se provee de una interface procedural con C y es desarrollada una interface a Smalltalk.

5.9.11 Arquitectura de GemStone

La figura 5.5 muestra la mayoría de las piezas del sistema GemStone. Stone y Gem corresponden a la memoria de objeto y a la máquina virtual del estándar de implementación de Smalltalk. Stone provee la gestión de almacenamiento secundario, control de concurrencia, autorización, transacciones, recuperación, y soporte para acceso asociativo. Stone también administra espacios de trabajo para jornadas activas. Stone usa sustitutos únicos llamados apuntadores orientados a objetos (OOPS) para referir a objetos y una tabla de objetos para mapear un OOP a una ubicación física. Otros sistemas orientados a objetos usan referencias directas entre objetos para acortar tiempo de acceso. Se eligió el enfoque de la tabla de objetos porque da flexibilidad para mover los objetos alrededor del almacenamiento secundario (que es importante cuando los objetos cambian tamaño) y permite que una implementación fácil de nuestro esquema sombra para recuperación. Stone está construido sobre el sistema de archivo subyacente VMS. El modelo de datos que provee Stone es más simple que el modelo completo GemStone, y provee operadores únicos para el acceso y actualización estructural. Un objeto puede almacenarse separadamente de sus subobjetos, pero los OOPS para los valores de las variables de instancia de un objeto están agrupados. Otros consideran representaciones descompuestas de objetos, y más representaciones de agrupaciones de objetos.

Gem asienta a Stone y elabora el modelo de almacenamiento de Stone dentro del modelo completo GemStone. Gem también añade las capacidades de compilación de los métodos de OPAL en códigos bytes y ejecutando ese código, autenticación del usuario y control de sesión. (Los códigos bytes de OPAL son similares, pero o idénticos, a los códigos bytes usados en Smalltalk). Parte de la capa de Gem es la imágen virtual: la colección de clases OPAL, métodos y objetos que son proveídos con el sistema GemStone.

Al ir desde la jerarquía Smalltalk hasta la imagen virtual GemStone, se han quitado las clases para el acceso de archivos, comunicación, manipulación de pantalla y ambiente de programación; son innecesarias, ya que tenemos almacenaje persistente para todos los objetos de GemStone objetos. El cómputo para la manipulación de pantalla necesita suceder cerca del usuario final, mientras GemStone se perfecciona hacia mantener números grandes de objetos persistentes. Las clases del ambiente de programación son reemplazadas por una aplicación que corre en un procesador separado. Se han añadido clases y métodos que hacen las

funciones de administración de datos de control de transacciones, conteo, propiedad, replicación, user profiles y creación de índices sea controlable dentro de OPAL.

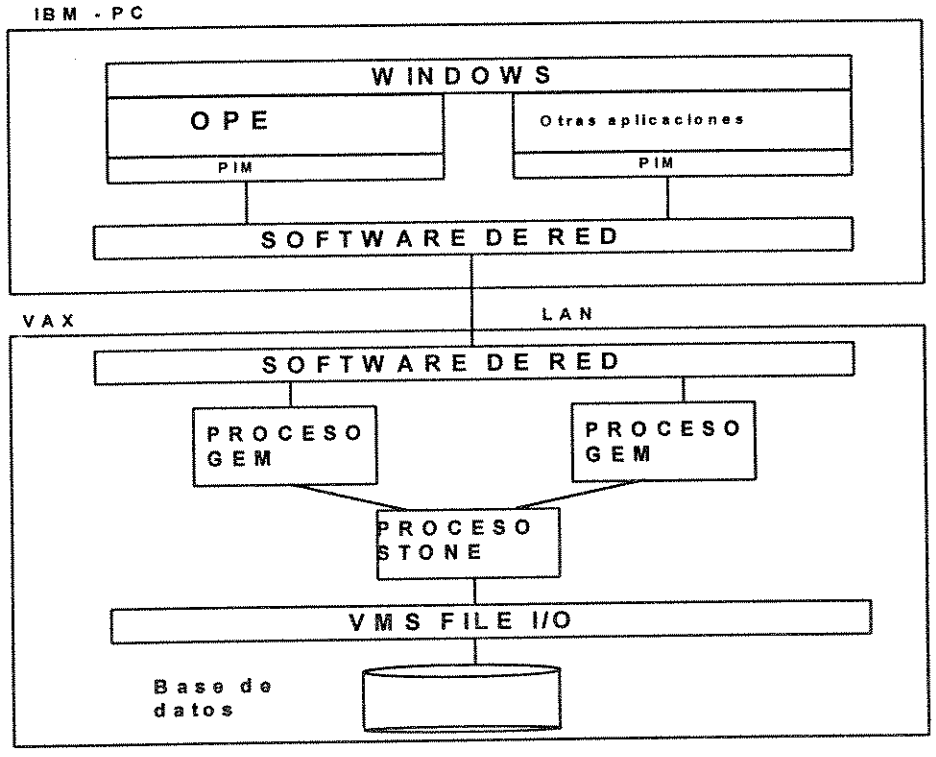


Figura 5.5 La arquitectura de GemStone.

El módulo de interface procedural (PIM) es un conjunto de rutinas para facilitar comunicación desde otros programas en otros lenguajes que corren sobre procesadores (posiblemente) remotos desde Gem. El PIM, actualmente, soporta llamadas desde programas en C que corren sobre una PC IBM para el control de transacción y sesión, enviando mensajes a los objetos de GemStone, ejecutando una secuencia de instrucciones de OPAL que recopilan métodos de OPAL, y explicación de errores. Además, el PIM provee llamadas al "acceso estructural" determinando un tamaño, clase, e implementación de objetos, accediendo un estado de objeto y creando nuevos objetos.

La Información pasa entre el PIM y Gem en forma de octetos (bytes) y apuntadores de objeto GemStone. Ciertos objetos tienen predefinidos apuntadores de objeto, tales como instanias de Boolean, char y SmallInteger. Las instancias de Real y String se pasan como sucesiones de bytes. Las instancias de otras clases deben descomponerse en instancias de las clases mencionadas, a fin de pasar su estructura interna entre el PIM y Gem, sin considerar su complejidad.

Gem, Stone y el PIM están estructurados como procesos separados. El actual mapeo de procesos a procesadores tienen a Gem y Stone corriendo sobre una VAX debajo de un VMS. Mientras un sistema GemStone tiene un único proceso Stone, mantiene un proceso separado de Gem para cada usuario activo, y el PIM maneja la comunicación por - aplicación base.

5.9.12 Múltiples Usuarios

Stone soporta múltiples usuarios concurrentes proveiendo a cada sesión de usuario con un espacio de trabajo que contiene una "copia sombre" de la tabla de objetos deribada desde la más recientemente tabla de objetos cometida (committed), llamada tabla compartida. Siempre que una sesión modifica un objeto, una nueva copia de ese objeto es creada y colocada en una página que es inaccesible a las otras sesiones. La copia sombra de la tabla de objetos es modificada, así que el OOP del objeto referencia a la nueva copia. Una de las ventajas de este método es que varias versiones consistentes de la base de datos pueden ser accesadas simultáneamente.

Conceptualmente, la tabla sombra de objetos de un espacio de trabajo es una copia completa de la versión de la tabla compartida cuando la sesión comienza. Actualmente no podemos hacer una copia de todo a la vez. Las tablas de objetos son representadas mediante árboles B, indexadas con OOPs. Para una tabla sombra de objetos, necesitamos copiar solamente el nodo de arriba de la tabla de objetos comprometida.

5.9.13 Indexamiento

En GemStone, el problema básico es seleccionar eficientemente desde una colección a esos miembros que se encontrarán en el criterio de selección. Por ejemplo, podemos querer encontrar todos los objetos que contengan un objeto dado como el valor de una variable de instancia particular. GemStone no soporta directamente navegación desde un objeto O hacia objetos en los cuales O es el valor de una variable de instancia, como un objeto puede ser el valor de una variable de instancia en varios objetos. Los objetos no asumen que esto hace una única referencia a cualquiera de los valores de las variables de instancia. Todos los objetos en GemStone son independientes; así que GemStone facilita el acceso asociativo mediante expresiones de camino o índices de camino que han sido introducidos en OPAL. Una expresión de camno (o simplemente un camino) es un identificador de variable seguido por una secuencia de cero o más identificadores de variables de instancia llamados enlaces o links. El identificador de variable que aparece que un camino es llamado el camino preffix, la secuencia de enlaces, el camino suffix.

CONCLUSIONES

1. El manejo de los datos e información ha evolucionado desde los sistemas de archivos a los sistemas de administración de bases de datos jerárquicas, luego a las bases de datos relacionales y, por último, a las bases de datos orientadas a objetos.

   Las bases de datos orientadas a objetos soportan tipos de datos más variados que solamente tablas, columnas y filas por relación; se basan en el encapsulamiento de datos y código que opera sobre esos datos en un objeto; y que tiene como características: mayor flexibilidad, mayor inventiva, mejor calidad, mejor comunicación entre usuarios, menos errores, y mayor productividad.

2. Las bases de datos convencionales almacenan **datos** de modo que resulten independientes de los procedimientos, a diferencia de las bases de datos orientadas a objetos que almacenan **objetos** ; es decir los datos junto con los métodos que procesan esa información; en ellas los datos no pueden ser accesados sino es a través de métodos.

3. Para poder conocer a las bases de datos orientadas a objetos es necesaria la comprensión de la conceptualización de la orientación a objetos: Objetos, clases, tipos, instancias, atributos, ocultamiento y/o encapsulamiento, jerarquía y herencia, mensajes, métodos, polimorfismo, identidad de objetos, persistencia, objetos complejos. Si el lector no se familiariza con todos éstos términos, le será difícil la comprensión del tema, además de la gran importancia que tiene el relacionar éstos términos con el mundo real.

4. Para poder conocer la arquitectura interna de las bases de datos orientadas a objetos, es necesario antes conocer a las bases de datos orientadas a objetos como tales, lo primero que debemos hacer es olvidar al modelo relacional y no tratar de encontrar similutedes cuando hacemos un diseño o una implementación, porque son dos enfoques totalmente diferentes; los sistemas de bases de datos orientados a objetos soportan datos complejos y relaciones complicadas usando conceptos de la orientación a objetos; carece de características de fuerte manejo de datos, es decir de alguna lenguaje como SQL que es estándar, aunque los actuales manejadores de bases de datos orientados a objetos (ODBMS) proveen su propio ODL (Object database language) no hay una estandarización (lo único que podríamos considerar como estándar sería "el objeto"), lo que significa que no hay un modelo de datos común que pueda utilizarse como punto de referencia.

5. Otra característica muy importante es que los ODBMS soportan no sólo gran número de objetos sino que objetos de gran tamaño (gráficos, archivos binarios,

etc.), además que provee el manejo de versiones de objetos, factor muy importante para control de concurrencia para las transacciones a largo plazo.

6. El rendimiento es un requerimiento básico en cualquier sistema de administración de bases de datos; dado que un modelo de bases de datos orientadas a objetos es más complejo que un modelo relacional.

   La eficiencia de la organización del almacenamiento depende no sólo de la estructura de los objetos y sus relaciones, sino también de la forma en que los programas de aplicación acceden a los objetos; es decir del patrón de acceso.
   Las técnicas de almacenamiento pueden basarse en dos formas:

   a. Los objetos que son de una misma clase se almacenan en un mismo archivo. De manera que cuando extraemos un objeto, lo extraemos completo, es decir el objeto y todos sus atributos, esta técnica presenta esta ventaja pero, cuando los objetos crecen se hace cada vez más difícil su almacenamiento debido a que existe un tamaño fijo para ellos.
   b. Descomponer los objetos en componentes atómicas y almacenar dichas componentes en un mismo archivo. Es decir en un archivo almacenamos por ejemplo un conjunto de atributos atómicos (indivisibles) que forman parte de un objeto y luego los referenciamos por medio de IDOs para construir los objetos completos, la ventaja es que los objetos pueden crecer indefinidamente, la desventaja está en la construcción de los objetos.
      Lo curioso de esto es que puede adoptarse un enfoque intermedio, que resulta ser mucho más flexible.

7. Los índices siguen siendo manejados con árboles B y varían de acuerdo a si son nodos hoja o no; la diferencia entre el manejo de índices de las bases de datos relacionales y el manejo de índices en las bases de datos orientadas a objetos está en el enfoque que le demos; no es lo mismo accesar datos comunes como números y letras que accesar datos complejos como mapas, gráficos, etc.. P. ej. imaginemos un supermercado, y deseamos investigar la ubicación de pañales; en los sistemas relacionales debemos forzosamente especificar: código 23456, marca "Pamper", talla mediano y la respuesta sería: pasillo 4, estante 2 y nada más; mientras que en el modelo orientado a objetos pediríamos la ubicación de los pañales así: **Pañales**, y nada más, y la respuesta sería un mapa con la ubicación exacta de los pañales encerrada en un círculo o algo así y más aún ¿alguna marca en especial? y solamente indicaríamos la marca y de nuevo el mapa con la ubicación exacta.

8. Las bases de datos orientadas a objetos son competitivas, en ellas se aprovecha toda la información, es decir tipos de datos que no estamos acostumbrados a manejar p. ej. sonido, video, documentos, presentaciones, ¿quién no querría tener todo esto en una base de datos ? y más aún puedo pedir algo así: muéstreme una cara parecida a la mía, en una edad comprendida entre 25 y 30 años, por

supuesto proporcionando la información adecuada (fotografías, datos, etc.) y el sistema respondería con todas las instancias de una cara como la mía. Además provee velocidad en las aplicaciones (rápido desarrollo, rápida administración) y extensibilidad (tipos de datos no comunes o complejos).

RECOMENDACIONES

1. Es indispensable al decidir trabajar con un manejador de bases de datos orientadas a objetos:
    - Contar con un equipo de personas dedicadas al análisis y desarrollo que se encuentre muy bien capacitado en lo que se refiere a la conceptualización del ambiente orientado a objetos; es decir que comprenda los conceptos y pueda familiarizarlos con el mundo real porque así serán las aplicaciones en el futuro.
    - Entrenamiento constante, ya que se trata de una nueva tecnología.
    - Estar dispuestos al cambio ya que es un enfoque totalmente diferente al del modelo relacional; y saber elegir un manejador que más se ajuste a sus necesidades.

2. Para escoger un manejador de bases de datos orientado a objetos es importante tomar en cuenta:
    - Competitividad: que ofrezca comodidades en el análisis, desarrollo e interfaces con usuarios.
    - Velocidad de las aplicaciones: implica velocidad de desarrollo y administración.
    - Extensibilidad: manejo de tipos de datos que no son comunes.
    - Acceso óptimo: un acceso óptimo implica aplicaciones rápidas.
    - Lenguaje de consulta común: significa curva de aprendizaje suave.

3. Los sistemas de bases de datos orientados a objetos aún no están totalmente definidos, como se comentó anteriormente no existe un patrón a seguir, así que aprender PostGress, O2, Iris, GemStone, Orion, Vbase u otro es distinto en cada uno, refiriéndose por supuesto al manejo de la herramienta como tal no a la abstracción que debe realizarse en el análisis que varía según el problema y que será distinto siempre, así que si está interesado en comenzar el cambio existen algunos modelos objeto-relacional (object-relational) que unifican ambos modelos y que constituyen un buen comienzo.

BIBLIOGRAFIA

Bertino, Martino. **Sistemas de bases de datos orientdos a objetos, Conceptos y arquitecturas.** Addison-Wsley/ Diaz de Santos, pp. 200-223.

Brackett. **Data Sharing using a common data architecture.** John Wiley & Sons, Inc. New York. 1994, pp. 30-67.

Catell. **The Object Database Standard.** Estados Unidos: Morgan Kaufmann Publishers, Inc. 1996, pp. 12-17.

Codad Peter, Yourdon Edward. **Object-Oriented Analysis**. Prentice Hall. 1996, pp. 56-90.

Embley David, Kurtz Barry, et.al. **Object-Oriented Systems Analysis**. Prentice Hall. Primera Edición. 1993, pp. 67-80.

Hawryszkiewycz. **Database Analysis and Design**. Addison-Wesley Publishing Company, segunda edición. 1995, pp. 78-95.

Montgomery Stephen. **Object-oriented information engineering**. Academic Press, Inc. New York. 1994, pp. 23-80.

Pinson Lewis, Wiener Richard. **Applications of Object-Orinted Programming**. Addison-Wesley publishing company.

Watson Tuan. **Experimental Object Database System**. Master's Thesis from San Diego State University, 1993, pp. 45-150.

Won Kim, Lochovsky Frederick. **Object-Oriented Concepts, Databses and Applications.** Addison-Wesley Publishing Company.

Zdonik Stanley, Maier David. **Object-Oriented Database Systems.** Estados Unidos: Morgan Kaufmann Publishers, Inc. 1990.