UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURIA Y ADMINISTRACION

MAPEO OBJETO RELACIONAL: UNA SOLUCIÓN AL PROBLEMA DE PERSISTENCIA TRANSPARENTE

TESIS PROFESIONAL

PAULINA SÁNCHEZ AGUILAR

MÉXICO, D. F. 2005

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURIA Y ADMINISTRACION

MAPEO OBJETO RELACIONAL: UNA SOLUCIÓN AL PROBLEMA DE PERSISTENCIA TRANSPARENTE

TESIS PROFESIONAL
QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN INFORMÁTICA

PRESENTA:
PAULINA SÁNCHEZ AGUILAR

ASESOR:
L. I. CARLOS FRANCISCO MÉNDEZ CRUZ

MÉXICO, D. F. 2005

Gracias a Dios por todo lo que me ha dado.

Gracias a ti mamá por tu amor y apoyo incondicional que desde siempre me has dado, por ser la persona que siempre ha confiado en mi y a quien le debo todo. Gracias por demostrarme que siempre en la vida hay más por qué luchar. Gracias por ser parte y dejarme vivir este sueño, por tus consejos y por ser la persona que más me quiere en este mundo.

Gracias a mi asesor y amigo, Carlos Méndez, por tus enseñanzas como profesor, por tu apoyo y confianza durante el desarrollo de este trabajo, por tus charlas amenas. Gracias por tus consejos y por ayudarme a crecer como persona.

Gracias a todos mis amigos que siempre me apoyaron, me ayudaron y a quienes quiero mucho. A Mivis por tu cariño y amistad que me demostraste desde que inició la carrera, por los momentos que vivimos juntas y por ser cómplice de muchas locuras. A César, por ser mi amigo incondicional, en las buenas y en las malas y a pesar de todo, gracias por tu cariño. A Clau por tu amistad y por ser testigo de este reto.

Índice

ÍNDICE **1**

1. INTRODUCCIÓN **4**

2. ANTECEDENTES **7**

2.1 TEORÍA GENERAL DE SISTEMAS 8
2.2 BREVE HISTORIA DE LAS BASES DE DATOS 9
2.3 BASES DE DATOS 9
2.4 MODELO RELACIONAL 10
2.4.1 Bases de Datos Relacionales *11*
2.4.2 Dependencias Funcionales *13*
2.4.3 Formas Normales *14*
2.4.4 SQL *15*
2.5 ORIENTACIÓN A OBJETOS 16
2.5.1 Introducción *16*
2.5.2 Modelado Orientado a Objetos *17*
2.5.3 Modelado de Objetos con Unified Modeling Language (UML) *18*
2.5.3 Programación Orientada a Objetos *21*
2.5.3.1 Características de la Programación Orientada a Objetos *21*
2.5.4 Java *22*
2.6 XML 23
2.6.1 DTD *25*

3. BASES DE DATOS ORIENTADAS A OBJETOS **27**

3.1 INTRODUCCIÓN 28
3.2 CARACTERÍSTICAS 28
3.3 ODMG 29
3.4 JDO 2.0 31
3.5 BASES DE DATOS OBJETO - RELACIONALES 31
3.5.1 SQL 3 ..... *32*

4. PROBLEMÁTICA **34**

4.1 PLANTEAMIENTO DEL PROBLEMA 35
4.2 JUSTIFICACIÓN DEL TEMA 36
4.3 OBJETIVO GENERAL 36
4.4 OBJETIVOS ESPECÍFICOS 37
4.5 HIPÓTESIS DE TRABAJO 37
4.6 DELIMITACIONES 37

5. MAPEO OBJETO RELACIONAL **38**

5.1 PERSISTENCIA 39
5.2 PERSISTENCIA TRANSPARENTE 39
5.3 ARQUITECTURA LÓGICA CON PATRONES 39

5.4 Servicio de Persistencia 40
5.5 Opciones de Persistencia de Objetos 41
5.6 Mapeo Objeto Relacional 42
5.7 Mapeo en un RDBMS y estructuras XML 43
5.7.1 Modelos 43
5.7.2 Elementos 45
5.8 Productos para el Mapeo Objeto Relacional 46
6. MAPEO OBJETO RELACIONAL CON HIBERNATE 48
6.1 ¿Qué es Hibernate? 49
6.1.1 Características 49
6.2 Arquitectura 49
6.3 Configuración 52
6.3.1 Conexión JDBC 53
6.3.2 Dialecto SQL 55
6.3.3 Archivo de Configuración XML 55
6.4 Clases Persistentes y Declaraciones para el Mapeo 56
6.4.1 Clases persistentes 56
6.4.2 Declaraciones del mapeo 58
6.4.3 Tipos de valores básicos 63
6.5 Colecciones y Componentes para el mapeo. 63
6.5.1 Colecciones 63
6.5.2 Mapeo de componentes 65
6.6 Herencia en el mapeo 66
6.7 Manipulación de datos persistentes 68
6.7.1 Crear un objeto persistente 68
6.7.2 Cargar un objeto persistente 68
6.7.3 Consulta 69
6.7.4 Actualizar objetos 69
6.7.5 Eliminar objetos persistentes 69
6.7.6 Flush (vaciar) 70
6.7.7 Finalizando una sesión 70
6.8 Lenguaje de Consulta Hibernate 70
6.8.1 La cláusula FROM 71
6.8.2 Asociaciones y Juntas (joins) 71
6.8.3 La cláusula SELECT 71
6.8.4 Funciones agregadas 71
6.8.5 Queries polimórficos 72
6.8.6 La cláusula WHERE 72
6.8.7 Expresiones 72
6.8.8 La cláusula order by 72
6.8.9 La cláusula group by 73
6.8.10 Subqueries 73
7. CONCLUSIONES 74

GLOSARIO **76**

BIBLIOGRAFÍA **77**

LIBROS 77
PÁGINAS WEB 78
MANUALES 79
TESIS 79

CAPÍTULO 1

INTRODUCCIÓN

El manejo de datos es una parte esencial para las empresas, ya que para la toma de decisiones necesitamos tener información fiable, concisa, veraz y oportuna. Por lo anterior es importante tener una capa de persistencia bien construida para el desarrollo de sistemas.

Debido a tal motivo, ha sido tarea para los investigadores el proponer nuevas y mejores tecnologías de persistencia de datos que aporten cada vez más beneficios a los informáticos. De la misma manera que se acoplen a los lenguajes de programación y herramientas para la construcción de los sistemas.

Las empresas siguen prefiriendo las bases de datos relacionales como almacén de sus datos debido al éxito de ellas.

De unos años a la fecha, el *boom* de los lenguajes de programación orientado a objetos han obligado a los manejadores de bases de datos relacionales a incorporar características que tienen estos lenguajes para lograr un acercamiento entre ellos a nivel de paradigma. Ya que ellos no son compatibles en muchos aspectos.

Por ejemplo, en la fase de diseño, un diagrama de clases no coincide con el diagrama entidad relación, a pesar de que en el último ya se puede representar la herencia en tablas.

Una desventaja al desarrollar orientado a objetos con bases de datos relacionales es que se lleva más tiempo en realizar el proyecto debido a que se tiene que programar con el lenguaje orientado a objetos y programar las instrucciones SQL.

Estas incompatibilidades no han sido obstáculo para reunir los beneficios de la orientación a objetos y las bases de datos relacionales. Existen patrones de diseño que nos permiten tener mejores prácticas para la capa de persistencia en el desarrollo sistemas.

Uno de los lenguajes de programación orientado a objetos que ha tenido éxito y aceptación entre los programadores ha sido Java, éste cuenta con una API, conocida por sus siglas como JDBC, que es un conector para bases de datos relaciones permitiendo la comunicación entre éstos.

Conforme se avanza en el desarrollo de otras maneras de realizar una comunicación más cercana entre los lenguajes orientados a objetos y los manejadores relacionales, los sistemas manejadores de bases de datos orientadas a objetos son los que tienen más beneficios sobre otras opciones, ya que existe una manipulación directa de los objetos y estos sistemas prácticamente se convierten en una extensión del lenguaje. Pero de igual tamaño son sus

desventajas, entre ellas la ausencia de un estándar maduro respalde a estos sistemas.
Existe una opción que promete ser una de las mejores opciones para solucionar el problema de persistencia, el mapeo objeto relacional, que integra las capacidades de los lenguajes orientados a objetos y las bases de datos relacionales. Esto lo logra transformando objetos a datos y viceversa.

Una herramienta de mapeo objeto relacional que nos permite tener estas ventajas es Hibernate, ya que podemos tener persistencia transparente del lado del lenguaje de programación, sin perder la madurez y fiabilidad de los sistemas manejadores de bases de datos relacionales.

En los siguientes capítulos se explican las bases de datos, las bases de datos relacionales, su estándar SQL. Las características de las bases de datos orientadas a objetos y los estándares que han nacido para darles mayor soporte como son el ODMG y JDO. Además, la orientación a objetos y el lenguaje de programación Java.

También se define el mapeo objeto relacional y para qué sirve, al igual que se estudia la herramienta Hibernate. Igualmente se mencionan otras opciones para realizar la persistencia de objetos. Todo esto para comprender el problema de incompatibilidad de paradigmas y su solución.

CAPÍTULO 2

ANTECEDENTES

2.1 Teoría General de Sistemas

"La teoría general de sistemas recalca la importancia de examinar todas las partes de un sistema" (Lucas, 1984: 17). Esta teoría nos dice que en un sistema todas las partes están involucradas estableciendo una comunicación entre especialistas de diferentes campos. La teoría está relacionada con la cibernética, que es una combinación de campos como la física, biología, electricidad, ingeniería y más. Precisamente es durante la etapa de análisis y diseño donde se aplican conocimientos de los diferentes campos

Para la teoría de sistemas "un sistema es un conjunto de componentes y variables organizado, interaccionado, interdependiente e integrado" (Lucas, 1984: 18), con objetivos y metas, aunque a veces éstas son difíciles de identificar.

Un sistema se compone de las tareas, actividades, misiones o partes del sistemas que se realizan para lograr objetivos.

Varios teóricos sugieren la siguiente lista de aspectos de la teoría general de sistemas:

1. Los componentes de un sistema están interrelacionados y son interdependientes; los componentes independientes y sin relación entre ellos no constituyen un sistema.

2. Se ve el sistema como un todo conjunto. A veces pueden observarse subsistemas.

3. Los sistemas tratan de hallar metas.

4. Los sistemas tienen entradas y salidas; dependen del procesado de un conjunto de señales de entrada para obtener las metas del sistema. Todos los sistemas producen algunas señales de salida que son necesarias para otros sistemas.

5. Todos los sistemas traducen señales de entrada en señales de salida, por lo regular éstos son diferentes.

6. Los sistemas ofrecen manifestaciones de entropía. Entropía es el estado de un ciclo cerrado (sin entrada de señales del exterior del sistema) donde todos los elementos se mueven hacia una desorganización, incapaz el sistema de producir señales de salida.

7. El sistema debe tener interacción de sus componentes para alcanzar sus objetivos.

8. Sistema compuesto por subsistemas. Forman una jerarquía característica de la teoría de sistemas.

9. Tiene unidades especializadas.

10. Los sistemas suelen presentar equifinalidad, esto es, varias maneras de lograr los fines del sistema.

2.2 Breve Historia de las Bases de Datos

Las bases de datos surgen de la necesidad de almacenar y recuperar datos. Los primeros modelos que surgieron fueron el modelo jerárquico y el de redes, en la década de los 60 en respuesta a que las corporaciones requerían nuevas aplicaciones que les permitieran controlar los datos que se producían cada vez con mayor rapidez y además que esas aplicaciones fueran capaces de relacionarse con otras del mismo sistema.

Pero aún existía el problema de la vulnerabilidad, si la base de datos fallaba todos los archivos o aplicaciones dependientes también se perderían, por lo que se buscaron otras formas de controlarlas, protegerlas y respaldarlas.

En 1970, E. F. Codd publica un artículo para la solución del almacenamiento de grandes cantidades de datos. La publicación del artículo pronto tendría gran atención y aceptación debido a su simplicidad y sus conceptos matemáticos y de álgebra relacional. Codd basó este artículo en la teoría de conjuntos y la lógica de predicados, lo que llevó al surgimiento del modelo de bases de datos relacionales.

A mediados de los 80, surge la necesidad de conectar a varios usuarios en una Red de Área Local de manera que pudieran compartir aplicaciones. El desarrollo de sistemas multiusuario tiene mayor funcionalidad, pero también el problema de coordinar las acciones independientes de cada usuario. Esto conduce a bases de datos capaces de trabajar en una arquitectura cliente servidor.

A finales de los 80 surge la programación orientada a objetos, y con ello la necesidad almacenar estructuras de datos compatibles con ella, de esta manera surgen los Sistemas Manejadores de Bases de Datos Orientados a Objetos.

2.3 Bases de Datos

Una *base de datos* es una colección de datos relacionados en un sistema computarizado que permite a los usuarios agregar, recuperar, modificar o eliminar datos. Se toman aspectos del mundo real en una colección coherente de datos

con significados inherentes, se diseña, construye y puebla para un propósito específico.

Un *sistema manejador de bases de datos* (data base management system) es un producto de software que permite a los usuarios crear y mantener una base de datos. Éste facilita los procesos de definición, construcción y manipulación de la base, además de ocuparse del acceso simultáneo, seguridad, respaldo y recuperación de datos.

2.4 Modelo Relacional

El modelo relacional fue introducido por Edgar F. Codd de IBM Research en 1970, como se mencionó antes, gracias a su artículo llamado "A Relational Model of Data for Large Shared Data Banks". El modelo utiliza el concepto de relación matemática como bloque de construcción y se basa en la teoría de conjuntos y la lógica de predicados. De este artículo se obtienen lo que ahora son llamadas las Doce Reglas de Codd.

Doce Reglas de Codd

Regla 1. Los datos son presentados en tablas.

Regla 2. Los datos son lógicamente accesibles por referencia a: 1) una tabla, 2) una llave primaria o un valor unique key, y 3) una columna.

Regla 3. Los nulos son tratados uniformemente como valores desconocidos.

Regla 4. La base de datos se describe a sí misma, existen dos tipos de tablas en un Manejador de Bases de Datos Relacionales: el catalogo del sistema o diccionario de datos y las tablas de usuario.

Regla 5. Se utiliza un lenguaje simple para comunicarse con el Manejador de Bases de Datos que incluye el lenguaje de manipulación de datos (DML), el lenguaje de definición de datos (DDL).

Regla 6. Provee alternativas para la vista de datos: una vista es una tabla virtual que se usa como alternativa para ver una o más tablas. Una vista no duplica datos y puede ser manipulada igual que una tabla.

Regla 7. Soporta operaciones básicas del álgebra relacional.

Regla 8. Independencia física de los datos. Los datos son físicamente independientes, no importa cómo se accede a ellos ni en donde están. Los cambios al equipo físico no afectan a los datos.

Regla 9. Independencia lógica de los datos. Los datos son lógicamente independientes: si hay una modificación en el esquema conceptual no afecta los datos que ya están almacenados.

Regla 10. La integridad de los datos es una función del Manejador de Bases de Datos Relacional: no es necesario un lenguaje externo al manejador para forzar la integridad de los datos.

Regla 11. Un Manejador de Bases de Datos Relacional soporta operaciones distribuidas: los datos pueden ser almacenados de manera centralizada o distribuida, sin afectar su contenido.

Regla 12. La seguridad de los datos no puede ser cancelada por un lenguaje externo o diferente al del manejador.

2.4.1 Bases de Datos Relacionales

Una *base de datos relacional* es un conjunto finito de tablas que tiene tres partes fundamentales: la estructura, la integridad y la manipulación de datos. El contenido de la información está representado en valores dentro de columnas y filas, y estas a su vez dentro de tablas.

Estructura de Datos

El usuario percibe la información de la Base de Datos en tablas como estructura lógica. Los términos estructurales utilizados son los siguientes: relación, tupla, cardinalidad, atributo, grado, dominio y clave primaria

a) Relación: conjunto de tuplas y atributos (tabla).
b) Tupla: conjunto de asociaciones de atributos (fila de una tabla).
c) Cardinalidad: número de tuplas que hay en el cuerpo de la relación.
d) Atributo: Nombre asignado a una columna, con un dominio determinado.
e) Grado: número de atributos.
f) Dominio: tipo de dato. Conjunto de valores atómicos y válidos para un atributo.
g) Clave Primaria: identificador único de cada tupla.

Las relaciones tienen ciertas propiedades que son muy importantes y que las distinguen de un archivo, estas características nos dicen que dentro de cualquier relación dada:

- No existen tuplas duplicadas.
- No existe un orden de las tuplas.

- Los atributos están en desorden.
- Cada tuplas tiene exactamente un valor para cada atributo, no se admiten atributos multivaluados ni compuestos (atomicidad).

Manipulación de datos

La manipulación de datos hace referencia a la manera en que el usuario, por medio de operadores, puede manipular los datos de la base, tomando tablas y derivando otras como resultado.

"El álgebra relacional es un sistema de operadores, cada uno de los cuales acepta una o dos relaciones como operandos y, como resultado, devuelve otra relación" (Johnson, 2000: 86).

Esta álgebra consta de 8 operadores, divididos en dos grupos, y son:

1. Conjunto tradicional: unión, intersección, diferencia y producto cartesiano
2. Operadores relacionales especiales: restringir, proyectar, juntar y dividir.

La operación *restringir* o *seleccionar* obtiene una relación que contiene todas las tuplas que satisfacen una condición determinada.

La operación *proyectar* selecciona algunos atributos especificados.

La operación *producto* devuelve una relación que contiene todas las combinaciones de tuplas posibles de dos relaciones.

La *unión* regresa una relación que contiene todas las tuplas que aparecen en las dos relaciones especificadas sin duplicados.

La *intersección* regresa una relación que contiene todas las tuplas que aparecen en las dos relaciones especificadas.

La *diferencia* regresa una relación que incluye todas las tuplas que aparecen en la primera, pero no en la segunda relación.

Juntar es la operación que regresa una relación que contiene todas las tuplas que aparecen en dos relaciones específicas gracias a un valor común de los atributos de las dos relaciones.

Dividir es el resultado de dos relaciones unarias y una binaria, donde a partir del producto de las dos primeras, regresa todas las tuplas de la primera relación cuya combinatoria con todos los de la segunda aparecen en la tercer relación que es binaria.

El *cálculo relacional* es el complemento del álgebra, ya que nos va a dar la información que deseamos obtener, más no cómo vamos a obtenerla.

Integridad de datos

Satisface restricciones de integridad para la exactitud o corrección de los datos, dada por:

- Restricciones de tipo: los valores deben ser sólo de un tipo.
- Restricciones de atributo: un atributo especifico debe ser de un tipo en particular.
- Restricciones de relación: se impone a una relación una restricción que es verificada al actualizar.
- Restricciones de bases de datos: reglas de negocio que aplican a dos o más tablas, éstas se construyen en un stored procedure y se verifica que la operación esté correcto para hacer un COMMIT TRANSACTION, de lo contrario realizará un ROLLBACK (trigger).
- Integridad referencia. Claves: llaves primarias y llaves foráneas.

2.4.2 Dependencias Funcionales

Una *dependencia funcional* es una restricción entre dos conjuntos de atributos con el fin de hacer consistente la Base de Datos. Son un vínculo muchos a uno que va de un conjunto de atributos a otro en una misma tabla.

Las dependencias funcionales no son muy fundamentales, sin embargo sí son de gran utilidad al momento de realizar nuestro diseño porque representan restricciones de integridad. Se representan de la siguiente manera:

$$X \rightarrow Y, \text{ se lee Y es dependiente funcionalmente de X}$$

Si r es una relación, X y Y son subconjuntos arbitrarios de un conjunto de atributos de r, X determina funcionalmente a Y sí y sólo sí cada valor de X en r está asociado precisamente con un valor de Y en r. Siempre que dos tuplas de r coincidan en su valor X, también coincidirán en su valor Y.

Ejemplo:

no_cuenta	nombre	edad	sexo
1	Mario	23	m
2	Juan	22	m
3	Martha	20	f
4	Alejandro	22	m
5	Mónica	21	f

Conjunto de Atributos
no_cuenta → nombre
(determinante) → (dependiente)

Dependencias triviales y no triviales

Una forma de reducir el tamaño del conjunto de dependencias funcionales (DF's) es eliminando las dependencias triviales. Una DF es trivial sí y sólo sí la parte de la derecha es subconjunto de la parte izquierda.

$$\{ \text{no\_cuenta, nombre} \} \rightarrow \text{nombre}$$

Dependencias Transitivas

Una tabla con 3 atributos A, B y C tales que las DF A $\rightarrow$ B y B $\rightarrow$ C, entonces A $\rightarrow$ C, ésto es una DF transitiva, a través de B.

Dependencias Reducibles e Irreducibles

Una DF es irreducible sí y sólo si:

- La parte derecha involucra sólo un atributo.
- La parte izquierda (determinante) de toda DF es a su vez irreducible.
- La parte derecha depende completamente de la llave primaria.

2.4.3 Formas Normales

Inicialmente Codd en 1972 propone sólo tres formas normales, después Boyce y Codd proponen una más estricta de la tercera, a ésta se le llamó Forma de Boyce-Codd. El proceso de normalización de datos puede considerarse como un proceso de análisis, que nos permite minimizar la redundancia. Se va haciendo una descomposición sin pérdida, por lo que hace a este procedimiento reversible y de hecho debe serlo para conservar la información total como se concibió en un principio.

Se dice que se está en *primera forma normal*, sí y sólo sí, una relación tiene exactamente un valor para cada atributo. Lo que restringe a un valor de atributo como una unidad atómica.

La *segunda forma normal* nos dice que una relación está en segunda forma normal sí y sólo sí se está en primera forma normal y todo atributo que no sea clave es dependiente irreducible de la clave primaria.

Se están en *tercera forma normal* sí y sólo sí se está en segunda forma normal y todos los atributos que no son clave son dependientes en forma no transitiva de la llave primaria.

La *forma normal de Boyce-Codd* nos dice que se esta en forma normal Boyce Codd sí y sólo sí toda dependencia funcional no trivial, irreducible a la izquierda, tiene una llave primaria como su determinante.

2.4.4 SQL

El *structured query language* (SQL) es un lenguaje estándar internacional para las bases de datos relacionales, y lo soportan casi todas las bases que existen en el mercado. Utiliza el término "tabla" para referirse a una relación.

Fue desarrollado en IBM Research a principios de los años 70 implementándolo en un prototipo inicialmente y después en otros productos comerciales de IBM. Es una estandarización de ANSI y de ISO.

En 1986 ANSI estandariza SQL usándolo como norma de acceso a bases de datos relacionales. ISO lo hace en 1987. Unidos, ANSI e ISO, lo adoptan como estándar en el año de 1989 surgiendo de esa manera el SQL89 (SQL1). Para consolidar el estándar lo revisan y lo ratifican en el año de 1992, éste era aproximadamente 6 veces más grande que su precursor, por lo que los autores decidieron definirlo en tres niveles de conformidad SQL92 (SQL2): conformidad del nivel de entrada, conformidad del nivel intermediario y conformidad completa.

En 1999 ANSI/ISO lanzó el estándar SQL99 (SQL3), que trae las características del anterior e incluye áreas de sistemas modernos tales como conceptos de los sistemas manejadores de bases de datos de objetos

El SQL básico nos permite crear una base de datos, definir las tablas y utilizar operaciones de definición y de manipulación de datos, como select, insert, update y delete.

Algunas características del SQL son: el catálogo de SQL describe una base de datos individual en un esquema de SQL. Las vistas muestran consultas programas, éstas son tratadas como tablas por tanto se pueden hacer consultas a éstas. Las transacciones nos permiten verificar las acciones llevadas a cabo, de ser correctas terminan con un COMMIT, de lo contrario ejecutan un ROLLBACK. El SQL incrustado nos permite tener código SQL con las instrucciones del lenguaje de programación anfitrión

2.5 Orientación a Objetos

2.5.1 Introducción

La Orientación a Objetos es un enfoque de desarrollo de software, sus técnicas pueden aplicarse durante el análisis, diseño e implementación de los programas, y nos permite resolver el problema de una manera metódica y ordenada, otra funcionalidad es que el código se puede reutilizar para otras aplicaciones.

Es un enfoque que organiza tanto el problema como su solución en una colección de objetos discretos, tanto estructura como comportamiento están incluidos en la representación.

Un *objeto* es cualquier cosa tangible o intangible, que tiene atributos y comportamiento, una *clase* es un conjunto de objetos que comparten estructura y comportamiento en común, los *atributos* ayudan a distinguir cada objeto particular de otro. Un *método* es aquél que va a definir el comportamiento del objeto y responde a un mensaje, los *mensajes* son funciones propias de una clase.

Según Lawrence (2002: 298), una representación orientada a objetos tiene 7 características: Identidad, Abstracción, Clasificación, Encapsulamiento, Herencia, Polimorfismo y Persistencia.

Identidad. Los datos son organizados en entidades discretas y distinguibles llamadas objetos. Tienen nombre, atributos y operaciones.

Abstracción. Ayuda a tener puntos de vista del sistema a desarrollar mediante abstracciones que forman una jerarquía, y muestra su relación. Se refiere a la concepción de las características esenciales de algún objeto de acuerdo a la perspectiva de cada observador.

Clasificación: Se utiliza para agrupar objetos que tienen atributos y comportamientos en común.

Encapsulamiento. Una clase encapsula atributos y comportamientos de un objeto, que son ocultos para las demás clases, y solo permite ver lo que es necesario que sea visto.

Herencia. Las clases pueden ser jerarquizadas de acuerdo a sus semejanzas y diferencias entre ellas. Comienza por definir una clase y luego especializarla en subclases, así la subclase hereda el comportamiento, estructura y atributos de la superclase

Polimorfismo. Un mismo comportamiento se manifiesta de manera diferente en distintas clases o subclases.

Persistencia. La capacidad que tiene el nombre, estado y comportamiento de un objeto de que se conserven permanentemente una vez que el objeto es transformado.

Algunas representaciones sólo utilizan un subconjunto compuesto de éstas características, aún así son consideradas como orientadas a objetos.

Para Booch (1995: 31) existen 4 características fundamentales: la Abstracción, Encapsulamiento, Modularidad y Herencia, si faltara alguna de estas características no se puede hablar de orientación a objetos. Y de forma secundaria: tipos, concurrencia y persistencia.

Modularidad. Mediante una fragmentación se obtienen fronteras bien definidas para tener una mejor comprensión del programa. Consigue la arquitectura física del sistema. Las abstracciones se sitúan en módulos.

Tipos. Propiedades o comportamiento que comparten los objetos, en éstos hay comunicación durante los procesos. En objetos de distintos tipos no pueden intercambiarse, o lo hacen de forma restringida.

Concurrencia. La capacidad de que existan distintos tipos de objetos actuando al mismo tiempo.

2.5.2 Modelado Orientado a Objetos

En el proceso de desarrollo de Sistemas Orientados a Objetos es importante definir los requerimientos, diseñar el sistema, codificar y realizar pruebas.

En el análisis se investiga el problema y los requerimientos a desarrollar. El diseño del sistema muestra lo que hará el programa con respecto a lo que se obtuvo en la investigación y nos permite describir los modelos lógicos y físicos, como estáticos y dinámicos del sistema que se diseña.

En estas dos fases encontramos diagramas que nos permiten expresar las ideas que deseamos compartir con otros analistas, diseñadores, programadores y con los mismos clientes, así como con las personas que realizaran las pruebas al sistema. Por lo que resulta importante tener un enfoque notacional que describa las soluciones del sistema.

"Un modelo es una descripción de las características de un área de estudio descritas mediante una serie de vistas" (Larman, 2003: 579).

De esta manera para el modelado orientado a objetos existen muchos métodos o lenguajes que nos permiten realizarlo, entre estos están: el método OMT, el método Coad - Yourdon, el de Schaeler – Mellor, entre otros. El lenguaje de modelado más conocido y utilizado es UML, sus diagramas se utilizan durante la especificación de requerimientos, diseño y codificación.

2.5.3 Modelado de Objetos con Unified Modeling Language (UML)

UML es un enfoque notacional gráfico que se ha generalizado para la descripción de soluciones Orientadas a Objetos. OMG[1] adoptó UML como estándar de la notación orientada a objetos en 1997. "Lenguaje de Modelado Unificado" es un lenguaje para **especificar, visualizar, construir** y **documentar** los componentes de los sistemas software, así como para el modelado del negocio y otros sistemas no software.

Utiliza tres vistas:

- Vista dinámica: casos de uso, lista de actividades, diagramas de interacción y máquinas de estado.
- Vista estática: diagramas de clases, diagrama de paquetes.
- Restricciones: se expresan con OCL[2].

Los diagramas estándar más usados de UML son: Diagramas de caso de uso, diagrama de clases, diagrama de paquetes, diagrama de secuencia, diagrama de colaboración, diagrama de estados, diagrama de actividades, diagrama de componentes y diagrama de despliegue.

Diagrama de caso de uso

Ilustra una unidad de funcionamiento dada por el sistema. Su propósito es ayudar a visualizar los requerimientos funcionales del sistema, incluyendo la relación de actores, en los procesos esenciales, tal como la relación entre diferentes casos de uso. Generalmente muestra casos de uso[3], ya sea para el sistema completo o para un grupo particular de casos de uso. Comunica las funciones de alto nivel y de los alcances.

Diagrama de clases

[1] Object Management Group es una Organización que promueve estándares para la industria.
[2] Es un lenguaje formal asociado con UML denominado Lenguaje de Restricciones de Objetos (Object Constraint Language), que se utiliza para expresar las restricciones en los modelos. Puede utilizarse en lugar del Lenguaje natural informal. Define un formato oficial para la especificación de las pre y post condiciones de las operaciones.
[3] Debemos recordar que un *caso de uso* es un documento de texto que nos sirve para descubrir y registrar requisitos, indica qué hará el sistema.

Los diagramas de clases muestran las diferentes entidades que existen el sistema y la relación con cada una de ellas, es decir, describe objetos, sus relaciones, los atributos de cada objeto, sus comportamientos individuales y las restricciones sobre cada clase u objeto. Este diagrama muestra la estructura estática del sistema. Se utiliza para la implementación de clases.

Una clase es representada mediante una caja dividida en tres secciones. En la sección superior se escribe el nombre de la clase, en la parte intermedia los atributos y el inferior los métodos.

Entre sus tipo de relación se encuentran principalmente la de herencia y la de asociación. La relación de agregación fue ya eliminada de UML 2.0 (Martin, 2004: 33).

Una relación de *herencia* usa un línea con una flecha apuntado a la superclase y la punta deber ser un triangulo completo, se lee "es un". La relación es de manera vertical, es decir, la subclase se coloca debajo de la superclase.

Una relación de *asociación* sucede cuando dos clases mantienen comunicación por algún periodo de tiempo. Se representa con una línea sólida si ambas clases están informadas cada una de la otra, si la asociación es conocida por sólo una clase la flecha debe ser con la punta abierta. La relación debe representarse de manera horizontal.

Diagrama de paquetes

Se utiliza para representar una colección de paquetes, un paquete contiene un conjunto de clases. Este diagrama diseña una arquitectura lógica del sistema y muestran información estática y global de alto nivel del sistema. Utiliza un icono que es un tipo de carpeta etiquetada y muestra agrupaciones lógicas de objetos. Las flechas de trazo cortado muestran las dependencias de los paquetes

Diagrama de secuencia

Muestran el flujo detallado de cómo ocurren las actividades en un caso de uso específico o de sólo una parte. Muestran las llamadas entre los diferentes objetos en su secuencia. Un objeto es representado por una caja en el extremo superior de una línea vertical, esta línea indica las líneas de vida del objeto, una caja angosta sobre la misma indica el comienzo o fin del mensaje. Una flecha entre dos líneas de vida indica un mensaje, ésta puede ser enviada a uno o varios objetos receptores, incluso puede ser enviada a sí misma.

Tiene dos dimensiones: la dimensión vertical muestra la secuencia de los mensajes / llamadas, en el orden de tiempo en que ocurren, la dimensión horizontal muestra las instancias de los objetos que son enviados los mensajes.

Se lee por la parte superior izquierda con el "conductor" instancia que inicia la secuencia.

Diagrama de Colaboración

Los diagramas de colaboración parten de un caso de uso, ilustran las interacciones entre objetos en un formato de grafo o red. Comprende enlaces, mensajes, creación de instancias u objetos. Los objetos son representados mediante una caja y las flechas representan mensajes, la secuencia de los mensajes se indican con un esquema de numeración y debe iniciarse de izquierda a derecha.

Diagrama de estados

El diagrama de estados modela los diferentes estados que una clase puede tomar y cómo éstas transitan de un estado a otro. Cada clase tiene un estado, pero no es necesario diagramar todas las clases, sólo aquellas con tres o más estados interesantes durante la actividad del sistema.

Un nodo inicial se representa con un círculo sólido. Para una transición entre estados se utiliza una flecha con la punta abierta; un estado se utiliza dibujando un rectángulo con las esquinas redondeadas, un punto de decisión con un círculo abierto y uno o más puntos de terminación se representan usando un círculo blanco con otro negro dentro de él. Las decisiones se expresan entre corchetes próximas a una flecha.

Diagrama de actividades

Muestra el flujo procedural de control entre dos o más objetos de clases mientras procesan una actividad. Este diagrama puede modelar el proceso del negocio a alto nivel, a un nivel unitario del negocio o a bajo nivel en las acciones internas de la clase.

Comienza con un círculo sólido conectado a la actividad inicial, la actividad se dibuja mediante un rectángulo con la orilla redondeada. Las actividades pueden ser agrupadas dentro de columnas para indicar el objeto que realiza la actividad. Las decisiones son representadas por rombos.

Diagrama de componentes

Proporciona una vista física del sistema reflejando los módulos reales del sistema. Su propósito es mostrar las dependencias que el software tiene sobre otros componentes software en el sistema. Puede mostrarse a muy alto nivel, con sólo los más grandes rasgos, o a nivel de paquetes.

"Un componente representa una parte de un sistema modular, desplegable y reemplazable, que encapsula la implementación y expone un conjunto de interfaces" (Larman, 2003: 566).

Diagrama de despliegue

Muestra como un sistema será físicamente desplegado en el ambiente hardware. Su propósito es mostrar donde los diferentes componentes del sistema correrán físicamente y cómo se comunican entre ellos.

Incluye elementos de notación usados en el diagrama de componentes. Incluye el concepto nodo, que representa cualquier máquina física o nodo de máquina virtual, se dibuja con un cubo.

2.5.3 Programación Orientada a Objetos

Un lenguaje orientado a objetos trata de amoldarse a nuestra manera de pensar. Utiliza objetos con características particulares para la representación en un lenguaje de programación de acuerdo con los conceptos que el ser humano es capaz de abstraer y generalizar. El uso de este tipo de lenguajes nos permite abordar la resolución del problema de manera metódica y ordenada y facilita su división y la cooperación de varias personas en el desarrollo de la aplicación así como la reutilización de código para otras aplicaciones.

Los conceptos básicos de la programación orientada a objetos, se definieron al principio de esta sección y son:

- objetos
- clases
- métodos
- atributos
- mensajes

2.5.3.1 Características de la Programación Orientada a Objetos

Existen 4 características fundamentales de la Programación orientada a objetos, éstas se describieron al principio del apartado 2.5, ellas son:

- Abstracción
- Herencia.
- Polimorfismo.
- Encapsulamiento.

2.5.4 Java

Java es un lenguaje de programación de propósito general, orientado a objetos y que cumple con las características mencionadas en el apartado anterior.

Este lenguaje tiene sus orígenes en 1990 en los laboratorios de Sun Microsystems. Un grupo de investigadores al mando de James Gosling diseñan y elaboran el software, en 18 meses desarrollan el lenguaje, por lo que en 1991 se cuenta con la primera versión de Java llamada Oak.

Se pensó en este lenguaje para crear programas que pudieran ser ejecutados en dispositivos electrónicos pequeños, como hornos de microondas, controles remotos, etc., y que pudieran correr en cualquier arquitectura hardware, ya que los programas están hechos de acuerdo a las características de los equipos y así compilados. Los chips de los aparatos eran prontamente reemplazados por otros más potentes y más baratos, por lo que se tenía que estar modificando código y volver a compilarlo.

Java nace de la necesidad de tener un lenguaje de programación que fuera multiplataforma, pequeño y eficiente.

A principios de la década de los 90 comienza a tener mucha aceptación Internet, pero sus herramientas como FTP, Telnet, e-mail, etc., no eran muy amigables con el usuario aunque ya se contaba con herramientas gráficas.

El equipo de James Gosling, al ver el éxito de Java en sus prototipos y que los ordenadores en red eran de distintos tipos conectados entre sí, decide utilizar Java como lenguaje para escribir aplicaciones que pudiesen funcionar a través de Internet.

En 1995 Oak es renombrado a Java debido a que el nombre ya había sido registrado. Se le da ese nombre porque así es llamado al buen café con estilo europeo.

Sus características son:

- Portable. Muchos ordenadores y sistemas operativos están conectados a Internet, por lo que se desea tener aplicaciones que puedan ser ejecutados en todos ellos.

- Fiable. Minimiza errores en código

- Sencillo. Es fácil de aprender debido a que se piensa en objetos y se trabaja con objetos, el código es modular y para personas con nociones de

programación, en especial C, les es más fácil utilizar y aprender este lenguaje.

- Seguro. Aplicaciones como Applets no pueden ejecutar código malicioso, debido a que los navegadores comprueban la integridad y consistencia del código, por lo que no pueden acceder a los recursos de la máquina que lo ejecuta.

- Multiproceso. Multithread o Multhilo crean hilos de ejecución que pueden trabajar en paralelo, simplificando la creación de gran cantidad de aplicaciones.

- Recolector de basura. Como en Java no existen punteros, se asignan recursos al momento de crear una nueva instancia de un objeto, cuando el programa ya no lo necesita libera esos recursos de manera que no se sobrecargue la memoria ram.

- Máquina virtual de Java. Cuando se compila un programa en Java da como resultado una representación binaria genérica (bytecode). Con este fin se crea la máquina virtual de Java que permita ejecutar aplicaciones en cualquier plataforma, ésta se encuentra en casi todos los navegadores para leer el bytecode de la aplicación.

2.6 XML

XML es un lenguaje de marcado que definen etiquetas. Está basado en el lenguaje de marcado generalizado estándar (SGML, Standard Generalized Markup Language), que proporciona un método de identificación de las partes y contenido de un documento mediante la información de su interior.

Ambos, SGML y XML, son:

- Conjuntos de reglas para el control de la creación de lenguajes de marcado que identifican el contenido de documentos.
- Adecuados para documentos de gran cantidad de información organizada
- Magníficos para estructurar información

HTML sólo controla la presentación de la información en páginas Web, mientras que XML nos da más enriquecimiento, ya que nos da una estructura personalizada, ocupándose del contenido del documento mediante etiquetas que definen un elemento y sus propiedades.

XML sólo describe el contenido del documento, para darle formato existen dos principales lenguajes de hojas de estilo: CSS y XSLT.

CSS (Cascading Style Sheets) fue el primer paso para separar contenido de formato específico para HTML y XML. Se convierten en estándar en 1996 y en 1998 aparece la recomendación CSS2.

Las transformaciones de hojas de estilo extensible XSLT (Extensible Style Sheet Transformations) son un lenguaje para la creación de, como su nombre lo dice, hojas de estilo de documentos XML, proporcionan un vocabulario basado en XML para dar presentación a la información.

Para tener un documento XML válido se necesita tener identificado los tipos de información y su contenido, así como la definición de la estructura (DTD).

La descripción de la estructura de un documento XML está compuesta por etiquetas que identifican su contenido y que describa su estructura. Al igual que HTML, XML usa paréntesis angulares, que en su interior especifican el nombre identificador del contenido "<" y ">", después de ellas se pone el contenido real marcado por XML. Y para la finalización se colocan de nuevo los paréntesis con el nombre, anteponiéndole a éste una diagonal.

Su estructura es similar a la de un árbol invertido.

```
<NOMBRE DE LA ETIQUETA> contenido </NOMBRE DE LA ETIQUETA>
```

XML es un metalenguaje, se puede utilizar para describir otros lenguajes, ésto permite proporcionar información acerca del contenido, y también identificar elementos y atributos de algún elemento. Esto se refiere a que por ejemplo, un elemento estudiante contiene atributos como nombre, dirección, edad que nos permiten describir al dato, o estudiante, en particular.

XML, como se dijo antes, es un conjunto de reglas, basado en un estándar definido por World Wide Web Consortion4.

En su primer estándar, XML1.0 lanzado en febrero de 1998 describe los requisitos de un documento bien formado, el origen y objetivo de XML:

- XML deberá ser sencillo de implementar en Internet.
- XML deberá ser utilizable en una gran variedad de aplicaciones.
- XML deberá ser compatible con SGML.
- Los documentos XML debe ser legibles por los humanos.
- El diseño de un documento XML deberá ser formal y conciso.
- Los documentos XML deberán ser sencillos de crear.
- El marcador de XML no tiene por que ser breve.

[4] Liga www.w3.org/XML

Una definición de un tipo de documento (DTD, Document Type Definition):

- Contiene las reglas para validar la información de un documento XML.
- Proporciona controles necesarios para identificar los elementos válidos para un documento y atributos válidos.
- Estos controles utilizan la gramática BNF (Backus-Naur Form).
- Es un documento que lista los elementos, atributos, notaciones y entidades que se pueden utilizar en un documento XML.
- Identifica el elemento raíz y sus elementos.

XML no sólo se conforma con la estructura de un documento que contenga información, tiene más elementos como son los Esquemas XML (XML Schemas), Lenguaje de Consulta de XML (XML Query Language), Lenguaje de Identificadores de XML (XPointer), XPath, Firmas Dígitales de XML (XML Digital Signatures), etc.

2.6.1 DTD

Un DTD es un documento en el cuál se declaran las entidades, notaciones y elementos permitidos en documentos XML que las utilizarán. Por lo que una DTD es un mecanismo para describir objetos que aparecerán en un documento XML.

La declaración física del documento es:

- Declaración de entidad `<!ENTITY>`
- Declaración de notación `<!NOTATION>`

La declaración en una estructura lógica del documento es:

- Declaración del elemento `<!ELEMENT>`
- Declaración de lista de atributos `<!ATTLIST>`

Para la declaración de elementos debe poner su nombre y contenido, el siguiente ejemplo lo muestra

```
<!ELEMENT contenedor (contenido)>
```

- El contenido puede definirse con las palabras reservadas `#PCDATA, EMPTY ANY.`
- El contenido puede tener otros elementos `<!ELEMENT contenedor (elemento1, elemento2, elemento3)>`

- El contenido puede ser de datos textuales `<!ELEMENT contenedor (#PCDATA)>`
- El contenido puede ser mixto `<!ELEMENT contenedor (#PCDATA | elemento1 | elemento2)>`
- El contenido pues tener cualquier tipo `<!ELEMENT contenedor ANY>`
- El contenido puede estar vacio `<!ELEMENT contenedor EMPTY>`

CAPÍTULO 3

BASES DE DATOS ORIENTADAS A OBJETOS

3.1 Introducción

Las Bases de Datos Orientadas a Objetos nacen debido al aumento del uso de lenguajes de programación Orientada a Objetos para el desarrollo de aplicaciones de software. Ofrece un almacenamiento persistente de Objetos creados en lenguajes de Programación Orientados a Objetos. Los ODBMS (Object Data Base Management System) se desarrollan a principios de los 80.

En una Base de Datos Orientada a Objetos hay clases, objetos, relaciones, un sistema de señales y métodos para procesarlas y contiene una interfaz uniforme para la base de datos. Un objeto tiene un identificador, atributos, tiempo de vida, estructura y métodos.

No hay un modelo como punto de referencia, por lo que en ese sentido es más flexible qué el modelo relacional, además de que tienen la oportunidad de manejar objetos y operaciones que ya están estructurados en un diagrama de clases.

Sus valores pueden ser objetos complejos, y ya no sólo cadenas y números, un atributo puede ser un conjunto de objetos, estos se ligan mediante identificadores de objeto únicos.

El objetivo de un ODBMS es almacenar objetos persistentes con todo y sus métodos, éstos necesitan almacenarse sólo una vez. Estos productos incluyen un compilador que procesa el código fuente y automáticamente crea estructuras de datos en la base de datos para almacenarlos.

3.2 Características

Existe un Primer Manifiesto para los Sistemas Manejadores de Bases de Datos Orientadas a Objetos[5] que fue escrito debido a la falta de un modelo de datos común entre los diferentes sistemas, la carencia de fundamentos formales y a que existía una actividad de investigación fuerte. Este documento separa en tres grupos las características:

- Obligatorias, que son las necesarias para que un sistema de bases de datos pueda ser orientado a objetos.
- Opcionales, que pueden incluirse para hacer mejor el sistema.
- Abiertas.

[5] *Object-Oriented Database System Manifesto.*
En http://www-2.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html

Dentro de las obligatorias se describen 12 características que se deben cumplir, contenidas bajo dos criterios:

- Debe proporcionar los servicios de un sistema administrador de bases de datos:
  1. Persistencia.
  2. Administrador de almacenamiento secundario.
  3. Concurrencia.
  4. Recuperación.
  5. Facilidad de consulta.

- Debe cumplir características de un sistema orientado a objetos:
  6. Objetos complejos.
  7. Identidad del objeto.
  8. Encapsulamiento.
  9. Tipos o clases.
  10. Sobre carga con combinación retrasada (overloading with late binding).
  11. Extensibilidad.
  12. Completez computacional (Computational completeness).

Las opcionales son:

- Herencia múltiple.
- Verificación de tipos e inferencia.
- Distribución.
- Diseño de transacciones y versiones.

Las abiertas se refieren a la especialización del software, y son:

- Paradigma de la programación.
- Representación del sistema o el tipo de sistema.
- Uniformidad del sistema.

3.3 ODMG

El Object Data Management Group fue concebido por un consorcio de vendedores de Sistemas Manejadores de Bases de Datos de Objetos con el fin de producir un estándar abierto para los DBMS orientados a objetos, su primer acercamiento es en el otoño de 1991. En 1993 se produce el primer reporte basado en el objeto como construcción fundamental llamado ODMG-93 (ODMG 1.0), además es la primera iniciativa sólida de acceso a datos vía objetos. En 1997 el estándar es actualizado por el Dr. Rick Cattel dando por resultado el ODMG 2.0.

En enero del 2000 es publicado el estándar ODMG 3.0 con modificaciones y mejoras con respecto al lenguaje Java y amplía el estándar para ser usado en sistemas de mapeo objeto relacional.

La meta de ODMG es permitir que los clientes de los ODBMS puedan escribir aplicaciones portables y que exista interoperabilidad entre los productos ODBMS.

La transparencia de un ODBMS integra la capacidad de una base de datos con la del lenguaje de programación. Esto hace innecesaria la utilización de un lenguaje por separado, como es el caso de SQL.

El ODMG es una definición de interfaces para productos de administración de datos de objetos (Kroenke, 2003: 578), se basa en: modelo de objetos, el lenguaje de definición de objetos (ODL), el lenguaje de consulta de objetos (OQL), y las ligaduras a los lenguajes de programación orientados a objetos.

El modelo de objetos es el que especifica el tipo de semántica que determina las características de los objetos y su relación. El Lenguaje de Definición de Objetos (ODL), modela dos tipos primitivos básicos: objetos y literales. Un objeto tiene un identificador de objeto y un estado, se describe con cuatro características: identificador, nombre, tiempo de vida y estructura.

Una literal sólo tiene un valor pero no identificador de objeto, puede tener una estructura simple o compleja, existen tres tipos de literales: atómicos, de colección y estructuradas. Un literal es un valor constante.

El Lenguaje de Consulta de Objetos (OQL) es un lenguaje muy parecido a SQL 2. Las ligaduras a lenguajes de programación orientada a objetos incluyen C++, SmallTalk y Java.

Las ideas fundamentales del ODMG se describen en 5 conceptos principales según Loomis (citado en Kroenke, 2003):

- Los objetos son fundamentales debido a que estos son la entidad que se almacena y se maneja.
- Cada objeto tiene un identificador único que es válido durante el tiempo de vida del objeto.
- Los objetos se organizan en grupos y por tipos, donde se tienen las mismas características y comportamiento. Puede haber herencia.
- El estado del objeto se representa por sus propiedades (atributo o relaciones).
- El comportamiento de un tipo de objetos se determina a través de sus métodos.

3.4 JDO 2.0

Actualmente existen dos estándares Java para el almacenamiento de datos, la Serialización y JDBC. La Serialización salva el estado del programa en un almacenamiento persistente, para preservar los objetos y las relaciones que existen entre éstos formando un grafo dirigido, pero no soporta a múltiples usuarios. JDBC es una serie de definiciones de interfaces para permitir los accesos a RDBMS, en particular con un sublenguaje incrustado SQL.

Las técnicas de acceso a datos son diferentes y requieren de un gran esfuerzo para los desarrolladores de aplicaciones por lo que Java Data Object provee de una API para dar solución al almacenamiento de datos. De esa manera el programador no necesita aprender dos lenguajes distintos, el lenguaje de programación Java y otro para la base de datos.

La arquitectura JDO tiene dos objetivos, proveer a los programadores de aplicaciones para tener una vista transparente de información persistente, y permitir implementaciones que se comuniquen con las bases de datos dentro de los servidores de aplicación. JDO tiene una API estándar para definir un sistema portable, escalable, seguro y mecanismos de transacción para la integración de sistemas de información empresariales.

JDO considera los siguientes puntos para el mapeo:

- Una clase es mapeada a una o más tablas y un atributo a una o mas columnas.
- Los modelos de objetos necesitan ser mapeados a un esquema relacional.
- Un modelo de objetos se mapea a una tabla primaria.
- Las tablas secundarias representan tablas no normalizadas que tienen cero o una fila correspondiente cada fila de la tabla primaria.
- Las relaciones también deben ser mapeadas.

3.5 Bases de Datos Objeto - Relacionales

Las Bases de Datos Objeto Relacionales surgen como una extensión de las Bases de Datos Relacionales para añadir características de las Bases de Datos Orientadas a Objetos.

Las Bases de Datos Relacionales se basan en el estándar SQL2 que incluye todas las características del modelo relacional, las bases de datos orientadas a objetos, como vimos, deben cumplir con el estándar ODMG.

Las nuevas tecnologías hicieron necesario modificar el estándar SQL para que los manejadores de bases de datos relacionales fueran mejorados y además tuvieran características de los manejadores orientados a objetos, de esa manera surge el estándar SQL3

3.5.1 SQL 3

Este estándar incorpora 3 grupos de nuevas ideas: soporte a los datos abstractos, mejoras a las definiciones de las tablas y la extensión de la construcción del lenguaje.

Los tipos de datos abstractos son una estructura equivalente a un objeto en un lenguaje de programación orientada a objetos, estos datos abstractos tienen métodos, atributos e identificadores y herencia, estos datos puede ser transitorios o persistentes.

SQL3 define dos tipos de datos abstractos: los de objeto y los de valores. Un datos abstracto de objeto es una estructura de datos identificable e independiente, al cual se le asigna un identificador denominado identificador único de objeto (OID), los OID son apuntadores a objetos. Los tipos de datos abstractos de valores no se les asigna OID y no pueden existir, excepto en el contexto en el cual se crean.

Las extensiones de las tablas SQL3, tienen un identificador de renglones, y el concepto de tabla es la definición de tres tipos: conjunto, multiconjunto y lista:

- Una tabla conjunto no tiene renglones duplicados.
- Una tabla multiconjunto puede tener renglones duplicados .
- La lista tiene un orden definido por una o más columnas.

El SQL3 incluye las siguientes partes:

- SQL/Framework .
- SQL/Foundation, que trata sobre nuevos tipos de datos, nuevos predicados, operaciones Relacionales y rutinas almacenadas.
- SQL/Binding (ligaduras), SQL insertado y la invocación directa como en SQL 92.
- SQL/Objects (objetos), permite los tipos de datos definidos por el usuario, constructores de tipos, funciones y procedimientos.
- Partes dirigidas a aspectos temporales y de transacciones de SQL.

- SQL/CLI (Call Level Interface), proporciona reglas que permiten la ejecución del código de aplicación sin proporcionar código fuente y evita la necesidad de procesamiento.

- SQL/PSM (Persistent Stored Modules).

Tipos abstractos de datos (TAD), crea un tipo con nombre definido por el usuario con comportamiento y estructura.

Soporta Herencia, una subtabla hereda todas las columnas de la supertabla, y contando con sus propias columnas.

CAPÍTULO 4

PROBLEMÁTICA

4.1 Planteamiento del Problema

En la actualidad, los lenguajes de programación orientados a objetos son la elección más recurrente para el desarrollo de software, ya que este enfoque se apega a la percepción de la realidad y ofrece características como herencia, modularidad, abstracción, polimorfismo, etc., que lo hacen tener mayor ventaja sobre otros paradigmas de programación.

Los sistemas necesitan que los datos permanezcan después de que los programas son ejecutados para ser consultados, actualizados o reemplazados, en nuestro caso necesitamos que los objetos sobrevivan, para ello deben guardarse en un almacenamiento persistente, por lo que requerimos una Base de Datos que nos permita guardarlos. Tenemos dos opciones principalmente para ello, las Bases de Datos Orientadas a Objetos, y las Bases de Datos Relacionales[6]. Las primeras guardan objetos y métodos, las segundas guardan registros en tablas.

Una opción lógica en un sistema orientado a objetos sería utilizar bases de datos orientadas a objetos, pero éstas aún no son fáciles de usar, son costosas y la mayoría de las organizaciones tienen sus datos almacenados en un manejador de bases de datos relacional, además de que no se puede migrar de un sistema de base de datos orientada a objetos a otro, ya que aún hay incompatibilidad entre ellos (Palasí, 2003; Kroenke, 2003).

Las estructuras de datos procesadas con Programación Orientada a Objetos son más complicadas para almacenarse en una base de datos relacional que en una base orientada a objetos, debido a que un ODBMS es capaz de guardar tanto los objetos como sus atributos, métodos y relaciones de herencia, así como permitir la persistencia transparente. Se han integrado características a las bases relacionales que soporten objetos, pero no dejan de ser relacionales.

Sin embargo no dejan de ser éstas las más óptimas para el desarrollo de sistemas, sea cual sea el enfoque, pero en el orientado a objetos a pesar de ser la mejor opción existe una incompatibilidad de paradigmas (paradigms mismatch). Por ejemplo la representación de los modelos, para las bases de datos relacionales el modelo entidad relacional y para la orientación a objetos el diagrama de clases.

De hecho la incompatibilidad entre éstos es un tema de debate entre los desarrolladores Java, quienes consideran la persistencia como un problema, ya que algunos opinan que puede solucionarse con un modelo especial de Java, como los EJB, o manejar código primitivamente con SQL y JDBC, pero no podría

[6] Hoy en día también existen las Bases de Datos Objeto-Relaciones, que son Bases de Datos Relacionales con características de las Bases de Datos Orientadas a Objetos. Vease página 16

haber portabilidad entre las bases de datos, ya que cada manejador tiene su propia versión del dialecto.

Los motivos por los que se siguen utilizando los manejadores de bases de datos relacionales en un desarrollo de sistemas orientados a objetos son los siguientes:

- Existen muchas más instalaciones de sistemas RDBMS, y por lo mismo una gran cantidad de datos almacenados bajo su entorno, que de un ODBMS.
- La mayoría de los profesionales están capacitados para escribir y mantener código SQL.
- El modelo de datos SQL está bien definido y es similar entre los productos RDBMS.
- "Hay más usuarios y empresas trabajando en la integración de SQL con Java suministrando herramientas que mejoren la sinergia entre los dos lenguajes" (Melton y Eisenberg, 2002:18).

Una solución, entre muchas, para seguir utilizando los RDBMS y lenguajes de programación orientados a objetos como Java, con las ventajas de la persistencia transparente[7], es el Mapeo Objeto Relacional (Object Relational Mapping).

4.2 Justificación del Tema

Las bases de datos relacionales siguen siendo las más utilizadas, y más aún tomando en cuenta que éstas han ido aumentando sus capacidades para soportar nuevas tecnologías. El desarrollo Orientado a Objetos cada vez toma más fuerza, ya que los lenguajes de programación, metodologías, estándares en los que se está trabajando, tienen esta tendencia. Por lo anterior, considero que es importante para el Informático conocer cuáles son las tecnologías apropiadas para resolver la problemática planteada y poder desarrollar aplicaciones orientadas a objetos y almacenar en bases de datos relacionales sin cambiar el paradigma de desarrollo y programación.

4.3 Objetivo General

Realizar una investigación sobre el Mapeo Objeto Relacional y ejemplificar uno de los mecanismos o tecnologías para resolver el problema de Persistencia Transparente de objetos.

---

[7] Se explica en el siguiente capítulo

4.4 Objetivos Específicos

Dar a conocer qué es y cómo funciona el Mapeo Objeto Relacional como opción para el desarrollo de Sistemas Orientados a Objetos con Bases de Datos Relacionales.

Utilizar una Base de Datos Relacional sin desligarse del lenguaje de programación Orientado a Objetos, mediante una herramienta de Software Libre como Hibernate, para permitir una Persistencia Transparente entre Java y la Base de Datos.

4.5 Hipótesis de trabajo

Si utilizo una herramienta de mapeo objeto relacional entonces puedo tener persistencia de manera más directa para el lenguaje de programación orientado utilizado para el desarrollo y seguir utilizando una base de datos relacional.

4.6 Delimitaciones

Existen muchas opciones de persistencia que nos permiten trabajar en un sistema orientado a objetos teniendo un manejador base de datos relacional, en este trabajo de investigación sólo expondré como una solución a la Persistencia de objetos el mapeo objeto relacional.

Las bases de datos orientadas a objetos no son tema de este trabajo, ni el paradigma estructurado de las bases de datos relacionales con cliente que ejecuta peticiones SQL.

Se eligió trabajar con una herramienta de Mapeo Objeto Relacional que cumpliera con la especificación JDO y ODMG.

En cuanto a tecnologías de desarrollo se escogió el lenguaje de programación Java, el sistema manejador de bases de datos PostgreSQL, e Hibernate como herramienta de mapeo, todas éstas son Open Source.

CAPÍTULO 5

MAPEO OBJETO RELACIONAL

5.1 Persistencia

En el desarrollo de sistemas encontramos que es forzosa la persistencia. Ésta es la capacidad de un objeto de almacenarse de manera permanente para poder ser instanciado de nuevo para su manipulación.

El estado de un objeto puede ser almacenado en un disco y un objeto con el mismo estado recreado en algún punto en el futuro.

Las aplicaciones orientadas a objetos no sólo se limitan a objetos, sino a gráficos enteros de objetos interconectados que pueden ser hechos persistentes y después recreados en un nuevo proceso.

Existen otros objetos que no son persistentes, sino transitorios, éstos tienen un tiempo de vida que es limitado por el proceso que los instancia.

5.2 Persistencia Transparente

La *persistencia transparente* se refiere al almacenamiento y recuperación de datos persistentes de manera directa usando un lenguaje de programación orientado a objetos, que será el mismo con que se desarrolle la aplicación.

Esto facilita el trabajo del programador, ya que no necesita utilizar interfaces como ODBC[8] o JDBC[9] y seguir utilizando el SQL de la base. Además de que se logra escribir menos líneas de código que si se estuviera utilizando el JDBC de Java, por ejemplo.

5.3 Arquitectura Lógica con Patrones

Un sistema es compuesto de muchos paquetes lógicos, como podrían ser el paquete de Interfaz, de acceso a la Base de Datos, interfaces de conexión a otros sistemas, etc., cada paquete agrupa un conjunto de clases que coinciden en las tareas que ellas realizan, es decir, en sus responsabilidades.

"La arquitectura del software es el conjunto de decisiones significativas sobre la organización del sistema software, la selección de los elementos estructurales y sus interfaces, junto con su comportamiento en subsistemas" (Larman, 2003:418).

[8] Open database connectivity. Es una API para acceder a bases de datos utilizando para ello SQL. Esta interfaz es desarrollada con lenguaje C, por lo que no es compatible con Java.
[9] Java database connectivity. Es una API de Java ejecutar comandos SQ y establecer una conexión con una base de datos relacional.

Una arquitectura con Patrones puede dividirse en dos:

- La arquitectura lógica del sistema que comprende su organización conceptual en capas, paquetes, frameworks importantes, clases, interfaces y subsistemas
- El despliegue de la arquitectura, como son los procesos y configuración de la red

5.4 Servicio de Persistencia

Dentro del diseño de un sistema necesitamos un framework de persistencia que nos permita modelar a detalle el servicio de persistencia. Tener esta separación nos permite tener el código de las clases que realizan la conexión con la base de datos.

Un framework de persistencia es un conjunto cohesivo de interfaces y clases concretas y abstractas de propósito general, reutilizable y extensible, que proporciona funcionalidad para dar soporte a los objetos persistentes.

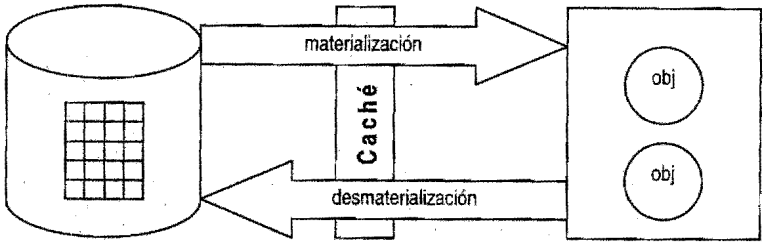
Un servicio de persistencia se escribe normalmente para que trabaje con una base de datos relacional, conocido también como servicio de correspondencia O-R (Objecto - Relacional). Este servicio tiene que traducir los objetos a registros o a alguna otra forma de datos estructurada como XML, y guardarlos en una base de datos, y traducir los registros a objetos cuando los recuperamos de la base de datos.

Las funciones para que el framework pueda dar el servicio son:

- Almacenar y recuperar los objetos mediante un mecanismo de persistencia.
- Se debe establecer alguna correspondencia de esquemas para el mapeo, entre una clase y su almacenamiento persistente y entre los atributos de los objetos y las columnas en un registro (tupla).
- Los objetos deben tener identidad, ésta se localiza por medio de un identificador único.
- Se necesita un conversor de base de datos[10] para lograr la materialización y desmaterialización.
- La materialización es el acto de transformar datos provenientes de una base de datos relacional en objetos. La desmaterialización es la actividad opuesta.

[10] Producto de mapeo objeto relacional

- Los servicios persistentes almacenan en un caché los objetos materializados por razones de rendimiento.



- Conocer el estado de los objetos en función de sus relaciones con la transacción actual para determinar si es necesario que se guarden de nuevo.

- Operaciones de transacción como son las instrucciones commit y rollback.

- Materialización perezosa que es cuando no todos los objetos se materializan a la vez, sino bajo demanda, sólo cuando se necesita.

Como podemos ver, para desarrollar un sistema orientado a objetos es necesario tener un producto que nos permita almacenar los datos, por lo regular es en una base de datos relacional, los datos se traen a la memoria local durante el uso de la aplicación que requiere que éstos sean persistente para los fines que fue creado el sistema.

5.5 Opciones de Persistencia de Objetos

SQL / JDBC es la forma más utilizada por los programadores de aplicaciones, que trabajan directamente con SQL y JDBC debido a que los RDBMS son familiares para ellos y éste se incrusta en el lenguaje anfitrión. Sin embargo el desarrollo requiere de incluir código de persistencia manualmente para las clases, lo que terminan consumiendo una gran porción del trabajo de desarrollo. Además, cuando los requerimientos cambian se requiere de más atención y esfuerzo.

La *Serialización* provee la habilidad de escribir un grafo de objetos a través de un flujo de bytes, ésta puede persistir datos a un archivo o a una base de datos. Otro uso de la serialización es replicar el estado de la aplicación a través de nodos en un clúster de la máquina. Pero a pesar de sus ventajas muestra dificultades al

acceder a los datos, ya que es necesario recorrer todo el grafo para poder recuperar un objeto (instancia de una clase). No es adecuado para sistemas Web con alto nivel de concurrencia o en aplicaciones empresariales, más bien es útil para programas de escritorio.

Los *entity beans* fueron creados especialmente para persistir datos, sin embargo no han sido del todo exitosos en la práctica. Los entity beans soportan la implementación de herencia, pero no soportan asociaciones de polimorfismo y sus queries. A veces necesitan de un producto de mapeo objeto relacional para poder ser portables entre aplicaciones.

Las *bases de datos orientadas a objetos*, como ya se ha mencionado, son el modelo ideal ya que no hay que modificar el modelo de objetos para almacenar los objetos, además de que un Sistema Manejador de Bases de Datos Orientadas a Objetos (OODBMS) es más parecido a una extensión de la aplicación, su implementación es multicapas y el almacenamiento ocurre en backend, hay un caché de objetos y la aplicación cliente engancha fuertemente interactuando con un protocolo de red. Pero a pesar de los esfuerzos que hizo ODMG (análogo a ANSI SQL) aún le falta madurar a este estándar y fortalecer sus características. JDO promete nuevas posibilidades y ya está siendo adoptado por varios productos de OODBMS.

Otra alternativa podría ser *XML*, que es un poco parecida a la serialización, sólo que ésta permite a herramientas acceder a la estructura fácilmente, pero no hay beneficios adicionales de XML, porque éste sólo es un archivo con formato de texto. Se podrían programar stored procedures y mover el problema a la base de datos.

Y finalmente, una opción que promete ser la mejor solución, es el Mapeo Objeto Relacional, que estudiaremos enseguida.

5.6 Mapeo Objeto Relacional

El Mapeo Objeto Relacional (Object Relational Mapping), nos permite combinar las características de la Programación Orientada a Objetos y los Sistemas Manejadores de Bases de Datos Relacionales para el desarrollo de Sistemas, permitiendo traducir objetos en tablas y tablas en objetos (materialización y desmaterialización).

El mapeo objeto relacional especifica la configuración de cómo mapear clases a tablas relacionales, transformando datos. Éste se implementa en una capa intermedia de la aplicación.

"El Mapeo Objeto Relacional es la persistencia automatizada (transparente) de objetos de una aplicación Java a tablas en una base de datos relacional, usando metadatos que describen el mapeo entre objetos y la base de datos" (Bauer y King, 2005: 23).

Una solución de mapeo objeto relacional consiste en:

- Una API para realizar las operaciones básicas para objetos de clases persistentes (create, read, update, delete).
- Un lenguaje o una API para la especificación de consultas a clases y sus propiedades.
- Facilidad para la especificación de metadatos en el mapeo.
- Una implementación para interactuar con las transacciones de los objetos.

Para conseguir un mapeo objeto relacional existen varios productos que ofrecen tener una persistencia transparente, dotándonos de APIs capaces de ser usadas por las aplicaciones para comunicarnos con un RDBMS.

Estos productos deben cumplir con el estándar ODMG que es un estándar dirigido a los manejadores de bases de datos orientados a objetos, como se vio en el capítulo 3, y que en su versión 3.0 ya incluye una especificación para el mapeo. Existe un segundo estándar para la persistencia de datos, Java Data Objects que como su nombre lo dice está enfocado al lenguaje Java, también ya mencionado.

5.7 Mapeo en un RDBMS y estructuras XML

5.7.1 Modelos

Se usan DTD para definir la estructura de un vocabulario XML, hay varios modelos para el contenido de elementos. Éstos han sido explicados previamente en la sección de XML.

- Contenido de sólo elementos.
- Contenido mezclado.
- Contenido de sólo texto.
- Contenido vacío.
- Cualquier tipo de contenido.
- Usando atributos.

Para tener nuestros datos en una base de datos relacional existen dos maneras por las cuales podemos almacenar esos mismos datos en un documento XML: contenido de sólo elementos y usando atributos.

Existen grupos de Puntos de Datos (Data Points) en XML que junto con las bases de datos son representados por tablas y columnas. Estos puntos de datos pueden ser agrupados juntos para describir un concepto representado por un elemento.

Cuando se mapean datos entre documentos XML y bases de datos relacionales una tabla podrá siempre llegar a ser un elemento con contenido de sólo elementos, y un elemento con sólo elementos deberá siempre ser una tabla a menos de que se utilizan las reglas de normalización.

Hay dos estrategias de diseño que pueden ser usadas para representar columnas como estructuras XML:

- Elementos que son anidados como hijos de un elemento que representa el agrupamiento de información.
- Atributos que son agregados a los elementos que representan agrupamiento de información.

Usando elementos

Los puntos de datos son representados por elementos en documentos XML. Para un mejor entendimiento se usará como ejemplo la factura que se expide a un cliente. El DTD sería:

```
<!ELEMENT Cliente (nombre, apellidos, direccion, ciudad, estado,
codigoPostal)>
   <!ELEMENT nombre (#PCDATA)>
   <!ELEMENT apellidos (#PCDATA)>
   <!ELEMENT direccion (#PCDATA)>
   <!ELEMENT ciudad (#PCDATA)>
   <!ELEMENT estado(#PCDATA)>
   <!ELEMENT codigoPostal (#PCDATA)>
```

Los elementos pueden resultar anidados por elementos separados bajo el elemento <Cliente>

```
<?xml versión="1.0" ?>
<!DOCTYPE Cliente SYSTEM http://servidor/xmldb/cliente.dtd>
<Cliente>
   <nombre> Sebastian </nombre>
   <apellidos> López </apellidos>
   <direccion> Av. Del Imán </direccion>
    . . . .
</Cliente>
```

Cuando se representan datos en un documentos XML, cualquier elemento que es definido teniendo como contenido de sólo texto usa las palabras #PCDATA y corresponderá siempre a una columna en una base de datos relacional.

Usando atributos

La otra forma de representar los puntos de datos es con atributos. En este enfoque los elementos que representan tablas tienen atributos asociados a ellos, que a su vez representan columnas. Su DTD con el mismo ejemplo se representa de la siguiente manera:

```
<!ELEMENT Cliente EMPTY>
      <!ATTLIST Cliente
            nombre CDATA #REQUIRED
            apellidos CDATA #REQUIRED

            direccion CDATA #REQUIRED
                  ciudad CDATA #REQUIRED
              estado CDATA #REQUIRED
              codigoPostal CDATA #REQUIRED
              . . . .

<?xml versión="1.0" ?>
<!DOCTYPE Cliente SYSTEM http://servidor/xmldb/cliente.dtd>
```

Aquí se almacenan los detalles del cliente como atributos del elementos

```
<Cliente
      nombre="Sebastián"
      apellidos="López"
      direccion= Av. Del Imán
      . . . .
</Cliente>
```

5.7.2 Elementos

Los elementos para el mapeo son básicamente tres:

1. Un manejador de bases de datos (RDBMS).
2. Clases Java.
3. Documentos XML.

Como hemos visto el RDBMS nos servirá como almacén, los documentos XML para definir los datos tal como estarán en nuestra base. Las clases Java nos permitirán tener una conexión con nuestra base de datos y los documentos XML, y

a su vez conectar los documentos XML con otras clases Java que pertenecerán a nuestra aplicación.

Para fines de nuestro estudio se explicó como se realiza el mapeo con estructuras XML para entender cómo trabaja nuestra herramienta de mapeo. Existen más elementos para el mapeo entre RDBMS y XML como la comprensión de árboles (nodos) que servirán de guía para agrupar elementos raíz e hijos, stored procedures (procedimientos almacenados) para simplificar las tareas de manipulación de datos y una librería que nos permita conectar el lenguaje Java con los documentos XML como es Xerces[11].

5.8 Productos para el Mapeo Objeto Relacional

Entre los productos que existen para el mapeo objeto relacional hay dos grupos, los productos comerciales y los libres. En los primeros están Cocobase, Toplink, Fast Objects, en los segundos se encuentran Object Relational Bridge, Hibernate, Castor JDO, Torque, Cayene, etc. A continuación se describen brevemente algunos de ellos.

Java Hibernate es una herramienta de mapeo muy popular para java, de su misma descripción: "Hibernate es poderoso, de gran alcance para el funcionamiento de persistencia objeto relacional y servicio de consulta para Java. Hibernate le permite desarrollar objetos persistentes, siguiendo el lenguaje común Java – incluyendo asociación, herencia, polimorfismo, composición y la colección de framework de Java"[12].

Java ObJectRelacionalBridge (OJB) es otro proyecto open source para mapeo objeto relacional, es una nueva parte del proyecto Jakarta. OJB es una herramienta de mapeo objeto relacional que permite persistencia transparente para Objetos Java para bases de datos relacionales, provee una API ODMG 3.0 y también una API JDO, OJB es publicado dentro de la licencia GNU LGPL[13].

Cocobase maneja la persistencia y recuperación de datos de la aplicación a una base de datos relacional, toma el código específico de la base de datos y prescribe SQL fuera del objeto y guarda su información en una capa de mapeo. El código de especificación de la base de datos y SQL son creados dinámicamente en tiempo de ejecución. Esto permite reutilizar fácilmente los objetos una y otra vez. Compatible con Enterprise Java Beans, Java Server Pages, Servlets, Applets, JDBC, SQL, plataformas J2EE, J2SE y J2ME. [14]

[11] Liga http://xml.apache.org/dist/xerxes-j/
[12] liga http://www.hibernate.org
[13]liga http://db.apache.org/ojb/
[14] liga http://www.thoughtinc.com/cber_index.html

Castor es un framework de datos open source para Java, logra un puente entre bases de datos relacionales y documentos XML. Ofrece una conexión Java a XML, persistencia Java a SQL. Se encuentra bajo la licencia BSD-like.[15]

Con Toplink los desarrolladores pueden mapear objetos Java y entity beans a esquemas de bases de datos relacionales. Trabaja con cualquier base de datos. Toplink recientemente fue comprado por Oracle[16].

Todos utilizan archivos XML como formato para guardar las especificaciones de las tablas y de las clases. Se encuentran en una capa intermedia entre la aplicación y la base de datos, sobre el conector JDBC.

[15] http://www.castor.org/
[16] http://www.oracle.com/technology/products/ias/toplink/index.html

CAPÍTULO 6

MAPEO OBJETO RELACIONAL CON HIBERNATE

6.1 ¿Qué es Hibernate?

En los sistemas automatizados, una gran porción de tiempo se dedica al desarrollo, la creación y mantenimiento de la capa de persistencia, usada para almacenar y recuperar objetos de la base de datos elegida.

Hibernate nace como una solución al problema del manejo de datos persistentes en Java, permitiendo al desarrollador libertad de concentrarse en el problema del negocio. Ofrece un framework de persistencia objeto relacional, provee soporte a colecciones y relaciones con objetos, tipos de datos compuestos, tiene un lenguaje de consulta rico para recuperar objetos de la base de datos, los tipos de datos definidos por el usuario y llaves primarias compuestas. También ofrece flexibilidad adicional para soportar herencia.

Hibernate fue lanzado bajo la Licencia Pública GNU. Soporta numerosas bases de datos como Oracle, Sybase, PostgreSQL, Microsoft SQL Server, MySQL, DB2, etc.

Hibernate es una capa de acceso a datos. Se encarga de las propiedades persistentes de una clase, para ello los objetos son hechos persistentes definiéndolos en un documento de mapeo.

6.1.1 Características

- Persistencia transparente sin procesar bytecode.
- Lenguaje de consulta orientada a objetos.
- Flexible mapeo objeto relacional.
- API simple.
- Generación de llaves primarias automáticamente.
- Definición de mapeo objeto relacional.
- HDJCA (arquitectura Hibernate de doble capa caché).
- Integración J2EE.

6.2 Arquitectura

El siguiente diagrama (Figura 6.1) muestra a muy alto nivel cómo es la arquitectura de Hibernate, donde puede verse la base de datos y la configuración de datos para proveer el servicio de persistencia a la aplicación.

De la aplicación dependen los objetos persistentes y colecciones, estas llegan hasta la capa de persistencia, soportada por Hibernate. El archivo

hibernate.properties es el que contiene, como su nombre lo dice, las propiedades de conexión con la base de datos, éstas se explican más adelante.

XML Mapping se refiere a los archivos que contienen los metadatos requeridos para el mapeo, el metadato incluye declaraciones de clases persistentes y de las propiedades del mapeo a las tablas de la base de datos. Por lo regular estos archivos tienen extensión .hbm.xml.

Y obviamente la base de datos es el contenedor de los atributos de los objetos.

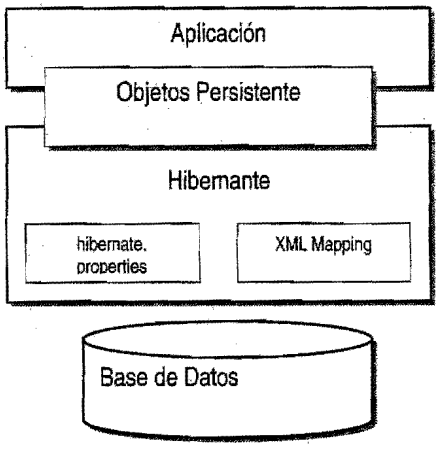


Figura 6.1 Arquitectura de Hibernate

Una vista más detallada de la arquitectura en tiempo de ejecución es como se muestra a continuación (Figura 6.2), como Hibernate es muy flexible y soporta muchos accesos sólo se explicará la arquitectura ligera. De igual forma, cuenta con muchas interfaces, que puede clasificarse de la siguiente manera:

- Interfaces para realizar operaciones básicas de crear, leer, actualizar y eliminar objetos, así como operaciones de consulta, en estas se incluyen Session, Transaction y Query.
- Interfaces llamadas por el mismo código de la aplicación para configurar Hibernate, la más importante es Configuration.
- Interfaces Callback que permiten a la aplicación reactivar eventos ocurridos en Hibernate, entre las principales están Interceptor, Lifecycle y Validate.
- Interfaces que permiten extender la funcionalidad de mapeo, tales como UserType, CompositeUserType e IdentifierGenerator.

Hibernate hace uso de APIs de Java, incluyendo JDBC, Java Transaction API (JTA) y JNDI.

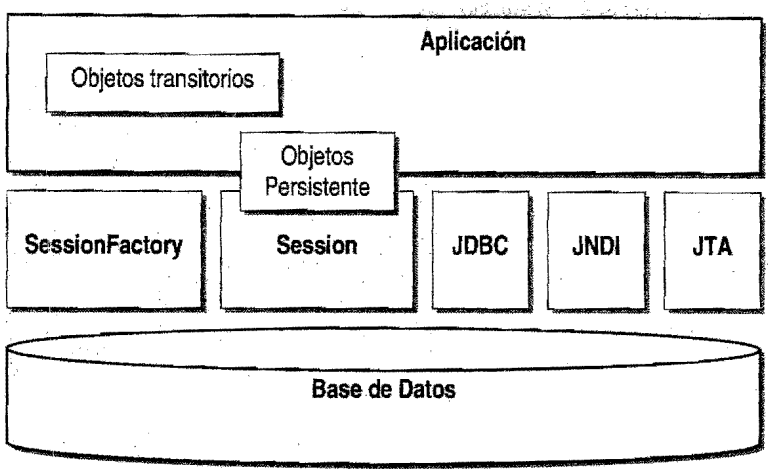


Figura 6.2 Arquitectura ligera de Hibernate

Existen 5 interfaces que son el núcleo de Hibernate, y volviendo a la imagen 6.2, a continuación se explican.

Session. Es la interfaz primaria usada en las aplicaciones Hibernate. Representa una conversación entre la aplicación y la base de datos. Envuelve una conexión JDBC.

SessionFactory. Está pensada para ser compartida entre varios hilos de la aplicación. Es necesario para contar con una instancia de Session y un cliente que es un ConnectionProvider. Soporta un caché de datos que es reusable entre las transacciones, a un nivel de proceso o cluster.

Configuration. La aplicación utiliza una instancia para especificar la localización de los documentos de mapeo y especificaciones de las propiedades de Hibernate, para entonces crear un SessionFactory.

Query y Criteria. La interfaz Query permite realizar las consultas a la base de datos, ya sea en lenguaje HQL[17] o SQL. La interfaz Criteria es muy similar, permite crear y ejecutar consultas orientadas a objetos.

Callback. Permite a la aplicación recibir una notificación cuando algo interesante le ocurre a un objeto, por ejemplo cuando ha sido cargado, salvado o eliminado, no son necesarias pero son útiles para implementar funcionalidad genérica como son los registros de auditoria.

[17] Hibernate Query Language.

Lifecycle y Validate. Permite a un objeto persistente reactivar eventos relacionados a su propio ciclo de vida persistente.

Types. Mapea objeto tipo Java a tipo columna de la base de datos. Hay un amplio rango de tipos construidos, e incluso Hibernate soporta tipos definidos por el usuario. Esta característica la proporcionan las interfaces *UserType* y *Composite.*

Objetos y Colecciones Persistentes. Objetos con estado persistente, están asociados con sólo una sesión, tan pronto es cerrada la sesión éstos son liberados y libres para usarse en otra capa de la aplicación.

Objetos y Colecciones Transitorios. Instancias de clases persistentes que no están actualmente asociadas con una sesión. Pueden ser instanciadas por la aplicación, pero no hechas persistentes, o haber sido instanciadas por una sesión cerrada.

6.3 Configuración

Hibernate generalmente es usado en aplicaciones cliente servidor de dos o tres capas, las aplicaciones cliente generalmente son en un navegador Web, pero también pueden ser usadas en aplicaciones de escritorio, ésto es importante para entender la configuración de Hibernate en ambientes manejadores y ambientes no manejadores.

Ambientes manejadores. Une recursos tales como la conexión a bases de datos y permite límites de transacción y seguridad, por ejemplo en aplicaciones servidor J2EE esta JBoss, BEA WebLogic o WebSphere de IBM que implementan el estándar manejador de ambientes Java. Hibernate se integra con los manejadores de transacciones y fuentes de datos .

Ambiente no manejador. Provee un manejo de concurrencia básico vía hilos pooling. Un contenedor de servelts como Jetty o Tomcat, son ejemplo de este ambiente. Las aplicaciones de escritorio o aplicaciones de línea de comando también son consideradas como ambiente no manejador. La aplicación debe manejar por ella misma las conexiones a la base de datos y demarcar límites de transacción. En este caso, Hibernate maneja las transacciones y las conexiones JDBC.

Hibernate puede ser configurado para ambos ambientes.

Existen dos opciones para configurar Hibernate, una de ellas es por medio de la clase Configuration, la cuál sólo necesita ser instanciada una vez para lograr tener en conjunto nuestros archivos XML de mapeo, y comunicar tipos Java a tipos SQL.

Para poder comunicar nuestra aplicación con la base de datos necesitamos instanciar una Session. La instancia de configuración puede conseguirse directamente:

```
Configuration cfg = new Configuration();
cfg.addFile("archivo.hbm.xml");
SessionFactory sessions = cfg.buildSessionFactory();
```

De esta manera estamos llamando a los archivos xml de mapeo, estos deben estar al mismo nivel con respecto al classpath de donde se encuentran nuestras clases.

La clase Configuration construye un SessionFactory, que como vimos anteriormente sirve para establecer una comunicación entre la base de datos y la aplicación.

Una vez que tenemos Configuration, éste analiza los archivos de mapeo, después se puede obtener una instancia de SessionFactory, esta instancia puede ser compartida por todos los hilos (threads) de la aplicación.

Existe otro método utilizado por Configuration:

```
SessionFactory sessions = new Configuration()
     .addClass(directorio.Clase.class)
     .buildSessionFactory();
```

Ésta es otra manera de escribir el código, ahora utilizando el método `addClass()`, en el que se asume que los archivos .hbm.xml están ya definidos.

Para especificar las opciones de configuración, se puede utilizar una de las siguientes maneras:

1. Pasar una instancia de java.util.Properties a Configuration.setProperties().
2. Poner hibernate.properties en el directorio raíz de nuestro classpath.
3. Configurar las propiedades en System, usando –Dproperty=valor.
4. Incluir los elementos <property> en el archivo hibernate.cfg.xml en el classpath.

6.3.1 Conexión JDBC

Una instancia SessionFactory puede abrir una Session para una conexión JDBC. Como se vio antes, ésto ocurre en un ambiente no manejador.

```
java.sql.Connection con    = datasource.getConnection();
Session session            = sessions.openSession(con) ;
```

Generalmente no es aconsejable crear una conexión casa vez que se quiera interactuar con la base de datos, de otra manera las aplicaciones Java deberían usar un pool de conexiones JDBC, algunas de las razones es que:

- Adquirir una nueva conexión resulta costoso para la aplicación.
- Mantener muchas conexiones perezosas también es costoso.
- Crear declaraciones preparadas es otro tanto costoso para los manejadores.

Como el ambiente no manejador no implementa un pool de conexiones, las aplicaciones podrían implementar su propio algoritmo pool o usar una tercera librería tal como el pool de conexión C3PO.

Para que Hibernate obtenga una conexión usando java.sql.DriverManager, se configuran las siguientes propiedades, todo esto en el archivo hibernate.properties:

`hibernate.connection.driver_class`	`clase del driver JDBC`
`hibernate.connection.url`	`url del JDBC`
`hibernate.connection.username`	`nombre del usuario de la base de datos`
`hibernate.connection.password`	`password del usuario de la base de datos`
`hibernate.connection.pool_size`	`número máximo de conexiones pool`

C3P0 es una conexión open source distribuido junto con Hibernate en el directorio lib. Hibernate usara el constructor C3P0 para un pool de conexiones, si se configura el archivo hibernate.x3p0.*. También hay un constructor soportado por Apache DBCP y por Proxool. Se configuran las propiedades en el archivo hibernate.dbcp.*

Para usar dentro de un servidor una aplicación, Hibernate puede conseguir una conexión de un javax.sql.Datasource registrado en JNDI (como Tomcat), siguiendo las propiedades:

`hibernate.connection.datasource`	`nombre JNDI de la fuente de datos`
`hibernate.jndi.url`	`url del proveedor del JNDI`
`hibernate.jndi.class`	`clase del JNDI`
`hibernate.connection.username`	`usuario de la base de datos`
`hibernate.conecction.password`	`password del usuario de la base de datos`

6.3.2 Dialecto SQL

Hibernate usa una propiedad "dialect" para la correcta subclase de net.sf.hibernate.dialect.Dialect para comunicarse con la base de datos, esto porque como hemos visto, hibernate comunica nuestras clases con nuestras tablas, y necesitamos convertir del lenguaje de programación orientado a objetos a SQL. Hibernate proporciona distintos dialectos o subclases para las distintas bases de datos. Para efectos de nuestra aplicación, utilizaremos:

```
PostgreSQL  net.sf.hibernate.dialect.PostgresSQLDialect
```

6.3.3 Archivo de Configuración XML

De forma alternativa, o como segunda opción de configuración, se tiene al archivo llamado hibernate.cfg.xml. Este archivo nos permite un reemplazo del archivo hibernate.properties, o si existen ambos, sobrescribirlo.

Por lo regular este archivo se encuentra en nuestro directorio raíz (de la aplicación), o en el classpath.

A continuación hay un ejemplo del archivo, en éste se consigue una conexión mediante DataSource (JTA):

```
<?xml versión='1.0' encoding='utf-8' ?>
<DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 2.0//EN"
http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd>

<hibernate-configuration>
      <session-factory
            name="java:comp/env/hibernate/SessionFactory">
          <property name="connection.datasource">my/first/datasource
          </property>
          <property name="dialect">
            net.sf.hibernate.dialect.PostgreSQLDialect
          </property>
          <property name="show_sql">false</property>
          <property name="use_outer_join">true</property>
          <property name="transaction.factory_class">
            net.sf.hibernate.transaction.JTATransactionFactory
          </property>
          <property name="jta.UserTransaction">
            java:comp/UserTransaction
          </property>

          <! - - archivos de mapeo - - >
```

```
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

En este caso, configurar hibernate podría resultar muy simple, ya que sólo se usa el método `.configure()`, y ya no `.addClass()`, o `.addFile()`, como vimos anteriormete.

```
SessionFactory sf = new
Configuration().configure().buildSessionFactory();
```

Haciendo una recapitulación de los archivos de configuración, si utilizamos el archivo hibernate.properties, estaríamos utilizando los métodos `.addFile()` o `.addClase()` de la clase Configuration, si utilizamos el archivo hibernate.cfg.xml sólo llamamos un método como acabamos de ver.

6.4 Clases Persistentes y Declaraciones para el Mapeo

6.4.1 Clases persistentes

Las clases persistentes son clases en una aplicación que implementan las entidades del problema del negocio. Estas clases tienen trascendencia e instancias persistentes almacenadas en una base de datos.

Para un mejor aprovechamiento de la aplicación, se sugiere trabajar con las siguientes reglas, también conocidas como modelo de programación Plain Old Java Objects (POJO).

El siguiente ejemplo muestra un POJO, éste fue extraído de la documentación de Hibernate:

```
package eg;
import java.util.Set;;
import java.util.Date;

public class Cat {
    private Long id;      //Identificador
    private String name;
    private Date birthdate;
    private Cat mate;
    private Set kittens;
    private Color color;
    private char sex;
    private flota weigth;
```

```
        private void setId(Long id) {
              this.id = id;
    }

        public Long getId() {
              return id;
        }

        void setName(String name) {
              this.name = name;
        }

        public String getName() {
              return name;
        }

        void setMate(Cat mate) {
              this.mate = mate;
        }

        public Cat getMate() {
              return mate;
        }

        void setBirthdate(Date date) {
              birthdate = date;
        }

        public Date getBirthdate() {
              return birthdate;
        }

        void setWeight(float weight) {
              this.weight = weight;
        }

        public float getWeight() {
              return weight;
        }
    }
```

Las reglas principales de trabajo son cuatro:

1. Declarar métodos setters y getters.
2. Implementar un constructor default.
3. Proporcionar una propiedad identificadora. Atributo id que soporte la llave primaria de la tabla de la base de datos.

4. Preferentemente, no usar clases finales[18].

Una subclase también sigue la primera y segunda regla, y es posible identificar las instancias por la propiedad identificadora ó id.

6.4.2 Declaraciones del mapeo

El mapeo objeto relacional es definido en un documento XML, con el fin de ser leído, editado y manipulado. El lenguaje del mapeo se centra en Java, por lo que se construyen de acuerdo a declaraciones de clases, no de tablas.

El mapeo XML puede definirse de forma manual, aunque también existen herramientas como XDoclet, Middlegen y AndroMDA que generan el documento de mapeo.

La estructura de un documento de mapeo XML puede verse en la figura 6.3. Para definir nuestro mapeo sólo se utilizan algunos elementos de acuerdo a las necesidades de nuestras clases. A continuación se explican estos elementos brevemente.

1) Doctype. El DTD para el mapeo se encuentra en el URL, en el directorio hibernate-x.x.x/src/net/sf/hibernate o en hibernate.jar, Hibernate buscará el DTD primero en el classpath.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd>

<hibernate-mapping
      schema="nombredelEsquema"
      default-cascade="none|save-update"
      auto-import="true|false"
      package="package.name">

      <class
            name="NombreClase"
            table="nombreTabla"
            discriminator-value="valorDiscriminador"
            mutable="true|false"
            schema="owner"
            proxy="ProxyInterface"
            dynamic-update="true|false"
            dynamic-insert="true|false"
            select-before-update="true|false"
```

[18] Una clase final es una clase con modificador final, no puede tener descendientes

```
polymorphism="implicit|explicit"
where="arbitrary sql where condition"
persister="ClasePersistente"
batch-size="N"
optimistic-lock="nome|version|dirty|all"
lazy="true|false">

<id
      name="id"
      type="string"
      unsaved-value="any|none|null|id_value"
                        column=nombre_columna
      access="field|property|ClassName"

      <generator class="claseGeneradora"/>
            <param name="tabla">uid_table</param>
            <param name="column">next_hi_value_column
            </param>
            <param name="max_lo">100</param>
      </generator>
</id>

<composite-id
      name="propertyName"
      class="ClassName"
      unsaved-value="any|none"
      access="field|property|ClassName">
      <key-property name="nombrePropiedad"
      type="nombretipo"
      column_name="nombre_columna"/>
      <key-many-to-one name="nombrePropiedad"
      class="NombreClase" column="nombre_columna" />

</composite-id>

<discriminator
      column="discriminator_columna"
      type="discriminator_tipo"
      force="true|false"
      insert="true|false"
/>

<version
      column="columna_version"
      name="propertyName"
      type="typename"
      access="field|property|ClassName"
      unsaved-value="null|negative|undefined"
/>
```

```
<timestamp
      column="timestamp_column"
      name="propertyName"
      access="field|property|ClassName"
      unsaved-value="null|undefined"
/>

<property
      name="propertyName"
      column="nombre_columna"
      type="typename"
      update="true|false"
      insert="true|false"
      formula="expresión arbitraria SQL"
      access="field|property|ClassName"
/>

<many-to-one
      name="propertyName"
      column="nombre_columna"
      class="NombreClase"
      cascade="all|none|save-update|delete"
      outer-join="true|false|auto"
      update="true|false"
      insert="true|false"
      property-ref="propertyNameFromAssociatedClass"
      access="field|property|ClassName"
      unique="true|false"
/>

<one-to-one
      name="propertyName"
      class="NombreClase"
      cascade="all|none|save-update|delete"
      constrained="true|false"
      outer-join="true|false|auto"
      property-ref="propertyNameFromAssociatedClass"
      access="field|property|ClassName"
/>

<component
      name="propertyName"
      class="className"
      insert="true|false"
      update="true|false"
      access="field|property|ClassName">

      <property . . . . / >
      <many-to-one . . . . />
      . . . . . .
```

```
        </component>

        <subclass
            name="ClassName"
            discriminator-value="discriminator_value"
            proxy="ProxyInterface"
            lazy="true|false"
            dynamic-update="true|false"
            dynamic-insert="true|false">

            <property . . . . />
        </subclass>

        <joined-subclass
            name="ClassName"
            proxy="ProxyInterface"
            lazy="true|false"
            dynamic-update="true|false"
            dynamic-insert="true|false">

            <key . . . . >
            <property . . . . . />
            . . . . . .
        </joined-subclass>

        <import
            class="ClassName"
            rename="ShortName"
        />

    </class>
</hibernate-mapping>
```

2) class. Se declara una clase persistente, con los elementos que hay dentro de ella, como son: nombre de la clase, nombre de la tabla, valor de discriminador, etc. Se acepta para implementar clases o subclases, o clases internas usando el debido estándar Clase$Subclase.

3) id. Define el mapeo de esa propiedad a la columna de la llave primaria de la base de datos.

4) composite-id. Para una tabla que tiene llaves compuestas se deben declarar aquí las propiedades. En lugar de la propiedad "id".

5) discriminator. Este elemento es usado para la persistencia polimórfica, se tiene una columna discriminadora en la tabla para indicar qué subclase instanciar para una fila en partícular. Sólo pueden usarse unos pocos tipos como: string, character, integer, byte, short, boolean, yes_no, true_false.

6) version. Opcional. Indica la versión de los datos que contiene la tabla, es usado para transacciones largas.

7) timestamp. Opcional. Es equivalente a <version> e indica que tabla contiene los datos con timestamp.

8) property. Indica una propiedad persistente de la clase.

9) many-to-one. Asociación a otra clase persistente, como en el modelo relacional es una asociación muchos a uno.

10) one-to-one. Asociación uno a uno a otra clase persistente. Dos variedades de asociaciones:
    a. asociación de llave primaria.
    b. asociación de llave foránea única.

11) component. Mapea propiedades de un objeto hijo a una tabla de la clase padre. Dentro de component esta la etiqueta <property> que mapea la propiedad de la clase hija a la columna de la tabla.

12) subclass. Es una persistencia polimórfica requiere que se declaren las subclases de la clase padre.

13) joined-subclass. Se utiliza está etiqueta para que una subclase sea hecha persistente a su propia tabla. No es necesario utilizar una columna discriminador.

Siguiendo con el ejemplo de la clase Cat vista anteriormente, el documento de mapeo sería de la siguiente manera:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
      PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
      http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd>
<hibernate-mapping package="eg">
      <class name="Cat" table="CATS" discriminator-value="C">
            <id name="id" column="uid" type="long">
                  <generator class="hilo"/>
            </id>
            <discriminator column="subclass"  type="character"/>
            <property name="birthdate" type="date"/>
            <property name="color" not-null="true"/>
            <property name="sex" not-null="true" update="false"/>
            <property name="weight"/>
            <many-to-one name="mate" column="mate_id"/>
            <set name="kittens">
```

```
            <key column="mother_id"/>
            <one-to-many class="Cat"/>
        </set>
        <subclass name="DomesticCat" discriminator-value="D">
            <property name="name" type="string"/>
        </subclass>
        </class>
        <class name="Dog">
            <.....>
        </class>
</hibernate-mapping>
```

6.4.3 Tipos de valores básicos

En la siguiente tabla se muestra el equivalente de los tipos de datos entre Java y SQL

JAVA	SQL
Integer, long, short, float, double, byte, char, bolean, Character, Integer, Long, Short, Float, Double, Boolean	INTEGER, LONG, SHORT, DOUBLE, TEXT, BOOLEAN
String	VARCHAR
Date	DATE, TIME, TIMESTAMP
Calendar	TIMESTAMP, DATE
BigDecimal	NUMERIC
Locale, TimeZone, Currency	VARCHAR
Class	VARCHAR
Binary	BINARY
String	CLOB, TEXT

6.5 Colecciones y Componentes para el mapeo.

6.5.1 Colecciones

Una colección en Java es un grupo de objetos.

Hibernate puede persistir instancias de las clases java.util.Map, java.util.Set, java.util.SorteedMap, java.util.Collection o java.util.List, y cualquier arreglo (array) de valores o entidades persistentes. además puede soportar una colección de tipo interfaz como List, Set y Map.

Las colecciones son declaradas por los elementos <set>, <list>, <map>, <bag>, <array> y <primitive-array>. No soportan valores nulos, ya que Hibernate no distingue una referencia nula ni una colección vacia.

Algunas de las características para el mapeo de colecciones son las siguientes:

- No comparten referencias.
- Creadas y eliminadas junto con el contenido de la entidad.
- Son persistidas automáticamente cuando son referenciadas por un objeto y eliminadas cuando no son referenciadas.
- Si una colección es pasada de un objeto persistente a otro, sus elementos pueden ser movidos de una tabla a otra.
- Se utilizan las colecciones Hibernate de la misma manera que en Java.
- Las colecciones instanciadas son diferenciadas en la base de datos por una llave foránea de su propia entidad, esta llave es referida como collection key, coletcion key es mapeada por el elemento <key>.
- Las colecciones pueden contener casi cualquier tipo Hibernate (visto en la sección anterior).

Los elementos de la colección pueden ser mapeados por <element>, <composite-element>, <one-to-many>, <many-to-many> o <one-to-any>. Los primeros dos son valores semánticos, lo otros son usados para mapear asociaciones de entidades.

Casi todos los tipos de colecciones, menos Set y bag, tienen una columna index que mapea a un índice o llave de mapa de un arreglo o lista. El índice de un arreglo o lista siempre es de tipo integer.

Los indices son mapeados usando <index>, <index-many-to-many>, <composite-index> o <index-many-to-any>.

El mapeo de una lista o un arreglo requiere de una columna para soportar el índice del arreglo o lista. Si el modelo relacional no tiene una columna índice entonces a cambio se puede usar un Set desordenado.

En Hibernate no es posible crear, eliminar o actualizar campos individualmente, debido a que no hay una llave que pueda ser usada como identificar único.

Un ejemplo de mapeo de colecciones de una lista de componentes, en la siguiente sección se explican los componentes, es de la siguiente manera:

```
<list name="carComponents" table="car_components">
      <key column="id"/>
      <index column="posn"/>
      <composite-element class="org.hibernate.car.CarComponent">
         <property name="price" type="float"/>
         <property name="type" type="org.hibernate.car.ComponentType"/>
```

```
        <property name="serialNumber" column="serial_no" type="string"/>
    </composite-element>
</list>
```

En una asociación muchos a muchos puede guardar una asociación una columna llave, columnas de elementos y posiblemente columnas índices. Para colecciones indexadas como mapas y listas se requiere de un elemento <index>. Para las listas esta columna debe contener una secuencia de entero que debe ser numerada desde cero.

6.5.2 Mapeo de componentes

"Un componente es un objeto contenido que es hecho persistente como un tipo de valor, no una entidad" (Documentación Hibernate, 2004). El concepto de componente se refiere a la noción de composición orientada a objetos.

Por ejemplo, se tiene una Persona, esta persona tiene un Nombre, éste a su vez tiene nombre apellido paterno y materno, estos elementos pueden ser considerados como componentes de Nombre Persona.

Los componentes no pueden ser referencias compartidas.

```
<class name="eg.Person"  table="person">
      <id name="Key" column="pid" type="string">
            <generator class="uuid.hex"/>
      </id>
      <property name="birthday" type="date"/>
      <component name="Name">
            <property name="initial"/>
            <property name="first"/>
            <property name="last"/>
      </component>
</class>
```

Se pueden declarar colecciones como componentes, pero no se pueden crear colecciones de componentes. Si el elemento compuesto contiene a la vez elementos compuestos, se puede usar la etiqueta <nested-composite-element>, aunque pueden utilizarse otras alternativas.

También puede utilizar un componente como identificador de una clase, esto se logra:

- Implementando java.io.Serializable.
- Re-implementando equals() y hashCode(), consistentemente con la base de datos de igualdad de llave compuesta.

En el caso de utilizar los anteriores puntos no se puede utilizar un generador automático de identificador. Para utilizar un identificador compuesto se utiliza la etiqueta <composite-id> en lugar de <id>.

6.6 Herencia en el mapeo

Existen tres tipos de estrategias básicas para mapear herencia, éstas son:

- Tabla por jerarquía de clase.
- Tabla por subclase.
- Tabla por clase concreta.

La primera estrategia no soporta que se mezclen mapeos <subclass> y <joined-subclass> dentro del mismo elemento <class>. De acuerdo con el ejemplo de la documentación de Hibernate, se tiene una interfaz Payment, con tres clases que la implementan: CreditCardPayment, CashPayment. ChequePayment. La estretagia tabla por jerarquía de clase (table per class hierarchy), se vería así:

```
<class name="Payment" table="PAYMENT">
      <id name="id" type="long" column="PAYMENT_ID">
            <generator class="native"/>
      </id>
      <discriminator column="PAYMENT_TYPE" type="string"/>
      <property name="amount" column="AMOUNT"/>
      . . .
      <subclass name="CreditCardPayment" discriminator-value="CREDIT">
      . . .
      </subclass>
      <subclass name="CashPayment" discriminator-value="CREDIT">
      . . .
      </subclass>
      <subclass name="ChequePayment" discriminator-value="CREDIT">
      . . .
      </subclass>
</class>
```

Sólo una tabla es requerida para este tipo de estrategia donde se indica con un discriminador de que tipo de subclase es en el mapeo indicándolo con la etiqueta <subclase>.

Para el mapeo tabla por subclase, el mismo ejemplo se vería de la siguiente manera:

```
<class name="Payment" table="PAYMENT">
      <id name="id" type="long" column="PAYMENT_ID">
            <generator class="native"/>
      </id>
      <property name="amount" column="AMOUNT"/>
```

```
. . .
<joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
      <key column="PAYMENT_ID"/>
. . .
</ joined-subclass >
< joined-subclass name="CashPayment" table="CASH_PAYMENT">
      <key column="PAYMENT_ID"/>
. . .
</ joined-subclass >
< joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
      <key column="PAYMENT_ID"/>
. . .
</ joined-subclass >
</class>
```

Aquí cuatro tablas son requeridas. Las tres tablas de las subclases tienen una llave primaria para asociarlas a la tabla de la superclase. En estos dos casos de mapeo de herencia puede utilizarse el polimorfismo para la interfaz de la siguiente manera:

```
<many-to-one name="payment"
column="PAYMENT
class="Payment""/>
```

La estrategia tabla por clase concreta es de diferente manera:

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
      <id name="id" type="long" column=" CREDIT _PAYMENT_ID"
            <generator class="native"/>
      </id>
      <property name="amount" column="CREDIT_AMOUNT"/>
      . . .
</ class >

<class name="ChashPayment" table="CASH_PAYMENT">
      <id name="id" type="long" column="CASH_PAYMENT_ID"
            <generator class="native"/>
      </id>
      <property name="amount" column=" CASH_AMOUNT"/>
      . . .
</ class >

<class name="ChequePayment" table="CHEQUE_PAYMENT">
      <id name="id" type="long" column=" CHEQUE_PAYMENT_ID"
            <generator class="native"/>
      </id>
      <property name="amount" column=" CHEQUE_AMOUNT"/>
      . . .
</ class >
```

Aquí sólo se utilizaron tres trablas, no fue necesario especificar la interfaz Payment, debido a que Hibernate utiliza el polimorfismo de manera implícita. Las

propiedades de Payment son mapeadas en cada una de las subclases, este polimorfismo puede mapearse utilizando <any>:

```
<any name="payment"
            meta-type="class"
            id-type="long">
      <column name="PAYMENT_CLASS"/>
      <column name="PAYMENT_ID"/>
</any>
```

6.7 Manipulación de datos persistentes

6.7.1 Crear un objeto persistente

Un objeto puede ser una instancia transitoria o persistente con respecto a una sesión (Session), ésta ofrece servicios para salvar instancias transitorias. Ésto se puede lograr de manera simple, únicamente es necesario llamar el método save().

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('F');
fritz.setName("PK);
Long generatedId = (Long) sess.save(fritz);

DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK);
pk.setKittens(new HashSet());
pk.addKittens(fritz);
sess.save(pk, new Long(1234));
```

Sólo hay que tener cuidado al momento de utilizar constraints NOT NULL, ya que ésto puede provocar excepciones si existen valores nulos.

6.7.2 Cargar un objeto persistente

Para cargar un objeto existe el método load() para recuperar instancias persistentes

```
Cat cat = new DomesticCat();
sess.load(cat, new Long(pkId));
Set Kittens = cat.getKittens();
```

También se puede utilizar el método get(). Se utiliza cuando se desea buscar una instancia que no es seguro que exista en la base de datos.

```
Cat cat = (Cat) sess.get(Cat.class, id);
If (cat == null) {
      Cat = new Cat();
      Sess.save(cat, id);
}
return cat;
```

6.7.3 Consulta

Para buscar objetos de los cuales se desconoce su Identificador (id), es posible utilizar el método find():

```
List Cats = sess.find(
      "from Cat as cat where cat.birthdate=?",
      date,
      Hibernate.Date
);
```

Si el número de objetos es muy grande como respuesta a una consulta, entonces se puede utilizar el método iterate() de java.util.Iterator.

En las consultas también puede utilizarse el lenguaje SQL dentro de las comillas. Si es necesario ponerle parámetros a nuestra búsqueda, la interfaz Query puede soportarlos.

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.serMaxResults(10);
List cats = q.list();
```

6.7.4 Actualizar objetos

A veces es necesario recuperar un objeto y enviarlo al cliente para luego ser modificado. Hibernate provee el método Session.update(). También existe el método saveOrUpdate().

6.7.5 Eliminar objetos persistentes

Session.delete() remueve el estado de los objetos, obviamente éstos deben ser previamente referenciados.

Sess.delete(cat);

6.7.6 Flush (vaciar)[19]

Estos procesos de flush ocurren de manera automática en los siguientes puntos:

- Para invocar find() o iterate().
- Al momento de utilizar net.sf.hibemte.Transaction.commite().
- Utilizando el método Session.flush().

6.7.7 Finalizando una sesión

Para finalizar una sesión se involucran cuatro criterios:

- Se vacía la sesión (flush). Ocurre cuando se usa la API Transaction, se realiza implícitamente cuando la transacción es ingresada, de otra manera puede ser llamada por Session.flush() para asegurar que los cambios son sincronizados con la base de datos.
- Se ingresa la transacción. Si se utiliza la API Transaction, lo hace automáticamente al momento de un commit, si se está utilizando como conexión a la base JDBC hay que hacerlo manualmente. O si no se desea ingresar los cambios (rollback).
- Se cierra la sesión. Implica el uso del método close().
- Manejo de excepciones. Es recomendable utilizar excepciones en una sesión para que inmediatamente ocurra un rollback, se llama al método close() para dejar el estado consistente de la base de datos.

6.8 Lenguaje de Consulta Hibernate

Hibernate tiene un lenguaje de consulta que es semejante a SQL, Hibernate Query Language (HQL), que es completamente orientado a objetos, entiende nociones de herencia, polimorfismo y de asociación.

Las consultas son insensibles a mayúsculas y minúsculas, lo más recomendable es definir un estándar para su uso, como por ejemplo, poner las cláusulas de FROM, SELECT, etc., en mayúsculas.

[19] Método para asegurar que los búferes de datos sean escritos realmente al dispositivo de salida físico.

6.8.1 La cláusula FROM

Es la más simple forma de consulta de Hibernate

```
FROM eg.Cat
```

Devuelve todas las instancias de la clase eg.Cat. También es posible utilizar alias para cuando se quiera referir a la clase en otras partes del query

```
FROM Formula as form, Parameter as param
```

6.8.2 Asociaciones y Juntas (joins)

Cuando se desea asociar entidades o valores de un elemento de una colección, se puede usar un join. Los tipos de join que tomados de SQL son:

- Inner join.
- Left outer join.
- Right outer join.
- Full join .

6.8.3 La cláusula SELECT

La cláusula SELECT elige qué objetos y propiedades regresará como resultado un query. Puede seleccionar elementos de colecciones. También puede regresar múltiples objetos y o propiedades como un arreglo de tipo Object[ ].

```
SELECT cat.mate FROM eg.Cat cat
```

6.8.4 Funciones agregadas

En los queries de HQL se puede regresar resultados de funciones agregadas sobre las propiedades. Éstas son:

- avg(. . .), sum(. . .), min (. . .), max(. . .)
- count(*)
- count(. . .), count(distinct . . .), count (all . . .)

Las palabras distinct y all son usadas de la misma manera que en SQL.

6.8.5 Queries polimórficos

Una consulta como

```
mate FROM eg.Cat as cat
```

regresa las instancias de no sólo Cat, sino también de sus subclases. Como se vio anteriormente, si se define en el documento de mapeo la propiedad any, ésto es posible.

6.8.6 La cláusula WHERE

La cláusula WHERE permite reducir la lista de instancias recuperadas de la consulta.

```
select foo
from eg.Foo foo, eg.Bar bar
where foo.startDate = bar.date
```

El operador = puede ser usado para comparar propiedades e instancias.

6.8.7 Expresiones

Las expresiones permitidas en las cláusulas WHERE incluyen muchos tipos de cosas que también se escriben en SQL:

- Operadores matemáticos +, -, *, /.
- Operadores de comparación binarios =, >=, <=.
- Operadores lógicos and, or, not.
- Concatenación de cadenas ||.
- Funciones escalares de SQL como upper() y lower().
- Paréntesis ( ) indicando agrupación.
- in, between, is null.
- Parámetros nombrados :name, :Stara, :X1.
- Literales SQK 'foo', 69, '1970-01-01 10:00:01:0'.
- Constantes Java public static final.

6.8.8 La cláusula order by

Una lista regresada por una consulta puede ser ordenada con las opciones asc o desc que indican ascenso o descenso respectivamente.

```
from eg.DomesticCat cat
```

```
order by cat.name asc, cat.weight desc, cat.birthdate
```

6.8.9 La cláusula group by

Una consulta que regresa valores agregados puede ser agrupada por cualquier propiedad o de una clase o componente regresado.

```
Select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color
```

Las cláusulas having y order by también son permitidas.

```
Select cat
from eg.Cat as cat
      join cat.kittens as kitten
group by cat
having avg(kitten.weight) >100
order by count(kitten) asc, sum (kitten.weight) desc
```

Ninguna de las cláusulas order by o group by tienen expresiones aritméticas

6.8.10 Subqueries

Para las bases que soportan subselect, Hibernate soporta queries dentro de queries. Generalmente éstos deben encerrarse entre paréntesis.

```
from eg.Cat as fatcat
where fatcat.weight > (
select avg(cat.weight) from eg.DomesticCat as cat
)
```

CAPÍTULO 7

CONCLUSIONES

El mapeo objeto relacional resulta una manera sencilla de lograr ese puente entre los lenguajes de programación orientados a objetos y los sistemas manejadores de bases de datos relacionales.

La herramienta Hibernate trabaja de una forma muy sencilla, su API hace el trabajo por nosotros, al traducir el lenguaje orientado a objetos a lenguaje SQL y viceversa de manera más simple, es decir, hace el trabajo sucio de manera transparente, permitiendo ver por parte del programador una simulación de una base de datos orientada a objetos.

El trabajo fuerte sólo se realiza una vez al realizar la especificación de las tablas y los objetos en los documentos de mapeo XML, y de ahí en adelante la programación es más fácil, incluso no sólo da solución al problema de persistencia, sino que también le ofrece beneficios a los programadores que se ven traducidos en menos líneas de código; al igual que nos facilita el mantenimiento de nuestra capa de persistencia.

Cabe aclarar que todos los beneficios que nos ofrece la herramienta de mapeo se dan cuando existe una compresión y manejo del lenguaje Java, debido a que para realizar todas estas maravillas es necesario utilizar la API de la herramienta, escrita en el lenguaje Java. La comprensión de los documentos XML es necesaria para poder especificar el mapeo de acuerdo a nuestras clases y tablas. Sin olvidar que es necesario el conocimiento de las bases de datos relacionales.

Glosario

ANSI. American Nacional Standards Institute, es administrador y coordinador para la estandarización del sector privado de los Estados Unidos de América.

API. Aplication Programming Interface. Biblioteca estándar que dota un gran número de clases.

Base de Datos. Es una colección de datos relacionados en un sistema computarizado que permite a los usuarios agregar, recuperar, modificar o eliminar datos. Nos permite representar algunos aspectos de mundo real en una colección coherente de datos con significados inherentes, y ésta se diseña, construye y puebla para un propósito específico.

DBMS. Sistema Manejador de Bases de Datos (Data Base Management System). Es un Producto Software que permiten a los usuarios crear y mantener una base de datos, facilita los procesos de definición, construcción y manipulación de bases de datos, además de ocuparse del acceso simultáneo, seguridad, respaldo y recuperación de datos.

HTML. HiperText Markup Language.

ISO. Internartional Organization for Standardization

Paradigma. Conjunto cuyos elementos pueden aparecer alternativamente en algún contexto especificado.

Pool de conexiones. Conexión a una base de datos utilizada por una aplicación Web que soporta un gran número de accesos, elevado por el número concurrente de usuarios.

Bibliografía

Libros

- Bauer, Christian; King, Gavin, "*Hibernate in Action*", Manning, E. U. A. 2005.
- Bertino, Elisa; Martino, Lorenzo, "*Sistemas de Bases de Datos Orientadas a Objetos. Conceptos y arquitecturas*", Addison-Wesley – Díaz de Santos, E.U.A., 1995.
- Booch, Grady, "*Análisis y Diseño Orientado a Objetos*", Addison Wesley – Díaz de Santos, E.U.A. 1994, 2ª edición.
- Chaudhri, Akmal B.; Rashid, Awais; Zicari, Roberto, "*XML Data Management. Native XML y XML-enabled Database - Systems*", Addison – Wesley, 2003.
- Chopra, Vivek; Galbraith, Ben; Li, Sing; et al, "*Profesional Apache Tomcat*", Wrox, E. U. A. 2002.
- Cuenca Jiménez, Pedro Manuel, "*Programación en Java para Internet*", Anaya Multimedia, Madrid 1996.
- Date, C. J., "*Introducción a los Sistemas de Bases de Datos*", Prentice – Hall, México 2001, 7ª edición.
- Elmasri, R; Navathe, S. B., "Fundamentos de Sistemas de Bases de Datos", Prentice – Hall, Madrid 2002, 3ª edición.
- Goldfarb, Charles F.; Prescos, Paul, "*Manual de XML*", Prentice – Hall, España 1999.
- Herbert Schldt, "Manual de Referencia *Java 2*", McGraw – Hill, España 2001, 4ª edición.
- Johnson, James Lee, "*Bases de Datos. Modelos, Lenguajes, Diseño*", Oxford, México 2000.
- Kroenke, David M., "*Procesamiento de Bases de Datos: fundamentos, diseño e implementación*", Addison Wesley, México 2003, 8ª edición.
- Larman, Craig, "*UML y Patrones*", Prentice – Hall, España 2003 2ª edición.
- Lucas Jr, Henry C, "*Sistemas de información. Análisis, Diseño y Puesta a Punto*", Paraninfo, Madrid 1984.

- Martin, Robert C., "*UML para programadores Java*", Prentice – Hall, Madrid 2004.
- Melton, Jim; Eisenberg, Andrew, "*SQL y Java. Guía para SQL, JDBC y tecnologías relacionadas*", Alfaomega Ra – Ma, España 2002.
- Mercado H. Salvador, "*Cómo hacer una tesis*", Limusa, México 1991.
- Sommerville, Ian, "*Ingeniería de Software*", Addison Wesley, México 2002, 6ª edición.
- Lawrence Pfleeger, Shari, "*Ingeniería del Software. Teoría y Práctica*", Prentice – Hall, Brazil 2002.
- Williamson, Heather, "*Manual de Referencia XML*", McGraw – Hill, España 2001.
- Williams, Kevin, et al, "*Professional XML Databases*", Wrox Press, 2001.

Páginas Web

- "*Cocobase*", http://www.thoughtinc.com/cber_index.html
  Fecha de consulta: Octubre 2004.
- *"Hibernate"*, http://www.hibernate.org/
  Fecha de consulta: Julio 2004.
- *"OJB, Object Relational Bridge"*, http://db.apache.org/ojb/
  Fecha de consulta: Julio 2004.
- "*Patterns for Object / Relational Mapping and Access Layers*"
  http://www.objectarchitects.de/ObjectArchitects/orpatterns/
  Fecha de consulta: Julio 2004.
- Atkinson, Malcolm; Bancilhon, François; DeWitt, David; Dittrich, Klaus; Maier, David; Zdonik , Stanley, "*The Object-Oriented Database System Manifesto*",
  http://www-2.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html.
- Barry & associates, "*Object-relational mapping articles and products*",
  http://www.service-architecture.com/object-relational-mapping/index.html
  Fecha de consulta: agosto 2004.
- Bell, Donald, "*UML basics: An introduction to the Unified Modeling Language*",

http://www-128.ibm.com/developerworks/rational/library/769.html
Fecha de consulta: Mayo 2004.

- Cattell, R. G. G., "*Standard Overview ODMG 2.0 Book Extract*", http://web.archive.org/web/20020602014318/http://www.odmg.org/standard/odmgbookextract.htm
  Fecha de consulta: Febrero 2005.

- Hanson, Jeff, Traductor: Palos, Juan Antonio (Ozito), "*Persistencia de Objetos Java utilizando Hibernate*", http://www.programacion.com/java/articulo/jap_persis_hib/
  Fecha de consulta: noviembre del 2004.

- López Tallón , Alberto, "*Diseño de Arquitecturas con Modelado O/R e Hibernate*", http://www.aqs.es/web/files/
  Fecha de consulta: Octubre 2004.

- Mark L. Fussell, "*Foundations of Object-Relational Mapping*", http://www.chimu.com/publications/objectRelational/
  Fecha de consulta: Julio 2004.

- Palasí Lallana, Vicent-Ramon, "*Motores de Persistencia*", http://www.programacion.net/bbdd/articulo/joa_persistencia/
  Fecha de consulta: Agosto 2004.

- Worsley, John ; Drake ,Joshua, "*Understanding SQL*"
  http://www.faqs.org/docs/ppbook/book1.htm
  Fecha de consulta: Marzo 2005.

Manuales

- Hibernate, "*Hibernate Referente Documentation. Version: 2.1.6*".

- Rosés Albiol, Francesc, Traductor: Abad Londoño, Jorge Hernán, "*Introducción a Hibernate*", Abril 2004.

Tesis

- Sarabia Delgado, Carina Guadalupe, "*Fortalezas de las bases de datos objeto relacionales*", para obtener el título de Licenciado en Informática, FCA, UNAM.