Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

FRAMEWORK'S PARA MAPEO OBJETO RELACIONAL: UN ANÁLISIS COMPARATIVO

Víctor Adolfo González García

Asesorado por el Ing. Ricardo Morales Prado

Guatemala, marzo de 2009

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

FRAMEWORK'S PARA MAPEO OBJETO RELACIONAL: UN ANÁLISIS COMPARATIVO

TRABAJO DE GRADUACIÓN

PRESENTADO A JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR:
VíCTOR ADOLFO GONZÁLEZ GARCÍA

ASESORADO POR EL ING. RICARDO MORALES PRADO

AL CONFERÍRSELE EL TÍTULO DE
INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, MARZO DE 2009

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Inga. Glenda Patricia García Soria
VOCAL II	Inga. Alba Maritza Guerrero de López
VOCAL III	Ing. Miguel Ángel Dávila Calderón
VOCAL IV	Br. José Milton De León Bran
VOCAL V	Br. Isaac Sultán Mejía
SECRETARIA	Inga. Marcia Ivónne Véliz Vargas

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADORA	Inga. Virginia Victoria Tala Ayerdi
EXAMINADOR	Ing. Edgar Estuardo Santos
EXAMINADOR	Ing. Juan Álvaro Díaz Ardavin
SECRETARIA	Inga. Marcia Ivónne Véliz Vargas

HONORABLE TRIBUNAL EXAMINADOR

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

FRAMEWORK'S PARA MAPEO OBJETO RELACIONAL: UN ANÁLISIS COMPARATIVO,

tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, en enero de 2008.

Víctor Adolfo González García

Guatemala, 09 de febrero de 2009

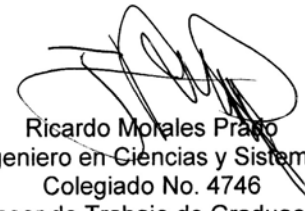
Ingeniero
Carlos Azurdia
Revisor de Trabajo de Graduación
Escuela de Ciencias y Sistemas
Facultad de Ingeniería

Respetable Ing. Azurdia:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante VÍCTOR ADOLFO GONZÁLEZ GARCÍA, titulado: "FRAMEWORK'S PARA MAPEO OBJETO RELACIONAL: UN ANÁLISIS COMPARATIVO", y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Sin otro particular, me suscribo de usted.

Atentamente,



Ricardo Morales Prado
Ingeniero en Ciencias y Sistemas
Colegiado No. 4746
Asesor de Trabajo de Graduación

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 02 de Marzo de 2009

Ingeniero
Marlon Antonio Pérez Turk
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Pérez:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **VÌCTOR ADOLFO GONZÀLEZ GARCÌA,** titulado: **"FRAMEWORKS PARA MAPEO OBJETO RELACIONAL: UN ANÀLISIS COMPARATIVO"**, y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA - INGENIERIA EN CIENCIAS Y SISTEMAS - COORDINACION TESIS

Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

ESCUELA DE CIENCIAS Y SISTEMAS

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÌA
ESCUELA DE CIENCIAS Y SISTEMAS
TEL: 24767644

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor y del Licenciado en Letras, de trabajo de graduación titulado ***"FRAMEWORKS PARA MAPEO OBJETO RELACIONAL: UN ANÁLISIS COMPARATIVO",*** *presentado por el estudiante VICTOR ADOLFO GONZALEZ GARCIA, aprueba el presente trabajo y solicita la autorización del mismo.*

"ID Y ENSEÑAD A TODOS"

Ing. Marlon Antonio Pèrez Turk
Director, Escuela de Ingeniería Ciencias y Sistemas

Guatemala, 20 de marzo 2009

Universidad de San Carlos
de Guatemala

Facultad de Ingeniería
Decanato

Ref. DTG.082.09

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al trabajo de graduación titulado: **FRAMEWORK'S PARA MAPEO OBJETO RELACIONAL: UNA ANÁLISIS COMPARATIVO,** presentado por el estudiante universitario **VICTOR ADOLFO GONZÁLEZ GARCÍA,** procede a la autorización para la impresión del mismo.

IMPRÍMASE.

Ing. Murphy Olympo Paiz Recinos
DECANO

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANO
FACULTAD DE INGENIERIA

Guatemala, marzo de 2009

/cc

ACTO QUE DEDICO A:

Dios, por darme la vida, el amor, la felicidad y ser la fuente de sabiduría en mi vida, solo gracias a su apoyo y misericordia puedo alcanzar mis metas, toda la gloria sea para él.

Mis padres, Alfonso González y Roselia García, quienes son un regalo de Dios para mí, siendo esta meta una bendición de sus esfuerzos.

Mis hermanos, Edgar, Mayra, Alida; cuñados Velveth, Christian, Percy; y sobrinitos con mucho cariño.

Mis amigos, quienes son una muestra del tamaño del amor de Dios hacia mí, siempre a mi lado sin esperar nada cambio, de quienes no quiero mencionar nombres ni apellidos porque ellos lo saben y se dan por aludidos.

Mi asesor Ingeniero Ricardo Morales, por compartir sus conocimientos y brindarme sus consejos que me han servido, tanto en mi vida personal como profesional.

Al pueblo de Guatemala, por darme la oportunidad de estudiar en la gloriosa y tricentenaria Universidad de San Carlos de Guatemala.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

EJEMPLOS

GLOSARIO

API Interfaz de programación de aplicaciones (Application Programming Interface) es un conjunto de rutinas, estructuras de datos, y/o protocolos proveídos por una librería y/o servicio de sistema operativo para ser utilizado por otro software.

Asociación Es la relación que existe entre dos objetos.

Atributo Propiedad del objeto al cual se le puede asignar un valor.

Clase Define cómo será el comportamiento del objeto y como almacenará su información. Es responsabilidad de cada objeto ir recordando el valor de sus atributos.

Columna Identifica un nombre que participa en una relación y especifica el dominio sobre el cual se aplican los valores.

Commit En el contexto de ciencias de la computación y administración de datos, commit se refiere a la idea de hacer de un conjunto de cambios tentativos se conviertan en permanentes. Un popular uso es al final de una transacción de base de datos.

Comportamiento	Es el conjunto de interfaces del objeto
EJB	La especificación EJB es parte del Api de java en la plataforma Enterprise Edition. El objetivo principal de los EJBs es ayudar al desarrollador a enfocarse en la lógica de negocio, evitando desarrollar problemas generales como: transacciones, concurrencia, seguridad, persistencia, etc.
Encapsulación	Es el ocultamiento de los detalles de implementación de las interfaces del objeto respecto al cliente.
Escalabilidad	Es la propiedad deseable de un sistema informático de manejar su tamaño o configuración para adecuarse al ambiente cambiante.
Framework	Un software *Framework*, es un diseño reutilizable para sistemas de software o subsistemas, estos pueden incluir soporte de programas, librerías y un lenguaje scripting entre otros software para ayudar a desarrollar y enlazar los distintos componentes de un proyecto.
Herencia	Especifica que una clase usa la implementación de otra clase, con la posible sobre escritura de la implementación de las interfaces.
Identidad de objeto	Propiedad por la que cada objeto es distinguible de otros, aún si ambos tienen el mismo estado o valores de atributos.

Interface	Operación mediante la cual el cliente accede al objeto.
Jar	JAR (Java Archive por sus siglas en Inglés) en desarrollo de software generalmente se utiliza para distribuir clases java y asociar metadata.
JDBC	Java Database Connectivity, es un API para la programación en Java que define como el cliente puede tener acceso a la base de datos, permitiendo la ejecución de operaciones sobre bases de datos desde Java sin dependencia de la base de datos o del sistema operativo.
JVM	Máquina Virtual Java es un conjunto de programas de software y estructuras de datos que utilizan un modelo de máquina virtual, para la ejecución de programas y scripts, es capaz de interpretar y ejecutar rutinas expresadas en Java bytecode, este se genera mediante el compilador del lenguaje Java.
Method Chaining	Es una forma de programación donde se pueden realizar encadenamiento de métodos, devolviendo un puntero this en lugar de un void.
Modelo Relacional	En administración de bases de datos, este es un modelo de datos basado en la teoría de conjuntos. Actualmente es el modelo que más se utiliza.

OID Identificador utilizado para nombrar un objeto, es una secuencia de números asignados jerárquicamente, permitiendo identificar objetos en la red y ser utilizados con distintos tipos de protocolos.

POJO Acrónimo de Plain Old Java Object utilizada por programadores de Java para enfatizar el uso de clases simples que no dependan de un *Framework* en especial.

Polimorfismo El polimorfismo es una característica nueva que surgió gracias a la POO. Esta característica nos permite especificar varias sentencias con el mismo nombre, siendo solo diferentes por sus parámetros de entrada.

Refactorización Es el proceso de cambiar la estructura interna de un código fuente, cambiando su estructura interna sin repercutir en su comportamiento externo o la funcionalidad existente.

Reusabilidad Es la probabilidad de un segmento de código para ser reutilizado y añadir nuevas funcionalidades con poca o ninguna modificación.

Rollback En el contexto de ciencias de la computación y administración de datos, rollback es una operación que se utiliza para volver a la base de datos a algún estado previo.

Serialización En ciencias de la computación, serialización es el proceso de convertir un Objeto en una secuencia de bits, estos pueden ser almacenados en un medio de almacenamiento (como un archivo, o un buffer de memoria).

SQL Lenguaje de consulta estructurado (Structured Query Language) diseñado para obtener y administrar datos en un sistema de administración de bases de datos Relacional (RDBMS), una de sus características principales es el manejo de cálculo y algebra relacional. Este es un lenguaje de cuarta generación (4GL).

Stand Alone Es utilizado para designar aquellas aplicaciones que pueden ser ejecutadas y controladas por un operador como entidades independientes de cualquier otra.

Tabla Mantiene tuplas relacionadas a lo largo del tiempo y puede actualizar sus valores. Esquematiza como están organizados los atributos para todas las tuplas que contiene.

UML Lenguaje unificado de modelado, es un lenguaje de modelado de propósito general respaldado por el OMG *(Object Management Group)*.

Usabilidad Termino utilizado para denotar la facilidad con que las personas pueden emplear una herramienta de software para lograr un particular objetivo.

XML	Extensible Markup Language, es un metalenguaje de etiquetas desarrollado por el Consortium World Wide Web.

RESUMEN

El mapeo objeto relacional en las aplicaciones empresariales, requiere de mucho esfuerzo, haciendo que el desarrollador desvíe su atención de la lógica de negocio. El manejo de la de la información persistente en una aplicación utilizando medios comunes como JDBC, no permiten el desacoplamiento de la capa de datos de la capa de lógica de negocio, se tiene la tendencia a fallas por el hecho de concatenar sentencias SQL, y el código se vuelve muy complejo, difícil de entender.

Una herramienta ORM trata de resolver los problemas consecuentes de las diferencias entre el paradigma de objetos y el modelo relacional. Haciendo uso de patrones de diseño, logran ahorrar bastante trabajo al momento de hacer persistentes los objetos, de igual forma nos ayudan a desarrollar aplicaciones escalables, optimizadas y con un bajo acoplamiento de sus módulos utilizando principios de diseño.

Este trabajo de graduación trata del estudio de *Framework's ORM* mediante un enfoque comparativo, estableciendo sus ventajas, desventajas y la información necesaria para poder determinar qué herramienta utilizar en determinada situación.

OBJETIVOS

General

Proporcionar una evaluación que compare los *Framework's ORM* más utilizados en el mercado, para guiar en la selección del ORM más conveniente en un proyecto en particular.

Específicos

1. Establecer los conceptos principales en el mapeo objeto relacional.

2. Definir el concepto de ORM, qué significado tiene y su importancia para los negocios modernos como una ventaja competitiva.

3. Obtener una guía, que podría ser utilizada para comprender como funciona un ORM.

4. Comprender cuáles son las diferencias, ventajas y desventajas de un ORM en términos de: reusabilidad, persistencia transitiva, facilidad de uso, curva de aprendizaje, facilidad de pruebas, escalabilidad, herramientas de apoyo, integración, seguridad, facilidad de refactorización, concurrencia, rendimiento, caché, carga retardada, facilidad de debug (errores SQL), soporte para transacciones e intrusión.

5. Documentar la arquitectura de los diferentes ORM's a analizar.

6. Establecer en que situaciones es viable utilizar un ORM.

INTRODUCCIÓN

Hace varios años, nació la programación orientada a objetos, que es un paradigma que nos ayuda a definir los programas en términos de clases y objetos, de igual forma a medida que las computadoras iban evolucionando, las personas necesitaban cada vez más hacer persistente su información, de esta forma nacieron las bases de datos, al principio el modelo que utilizaban eran jerárquicos, de red, etc., pero la que más sobresalió sobre estas fueron las de modelo relacional, pues esta se conceptualiza de una mejor manera. En el modelo relacional, se abstraen las relaciones como si fueran tablas que están compuestas por registros que representan *tuplas*, y campos que representan las columnas de la tabla. En la actualidad estos dos paradigmas son los más utilizados, claro uno para programación y el otro para un modelo de base de datos.

La persistencia de la información es muy importante en las aplicaciones y sí éstas tienen su diseño orientado a objetos, solo se puede lograr de dos formas: serializando los objetos o guardando la información en una base de datos. Como se mencionaba anteriormente, las bases de datos más populares obedecen al modelo relacional. De allí surge la idea de hacer uso de una herramienta que permita de forma fácil el mapeo objeto-relacional.

1. MARCO CONCEPTUAL

1.1. Antecedentes del problema

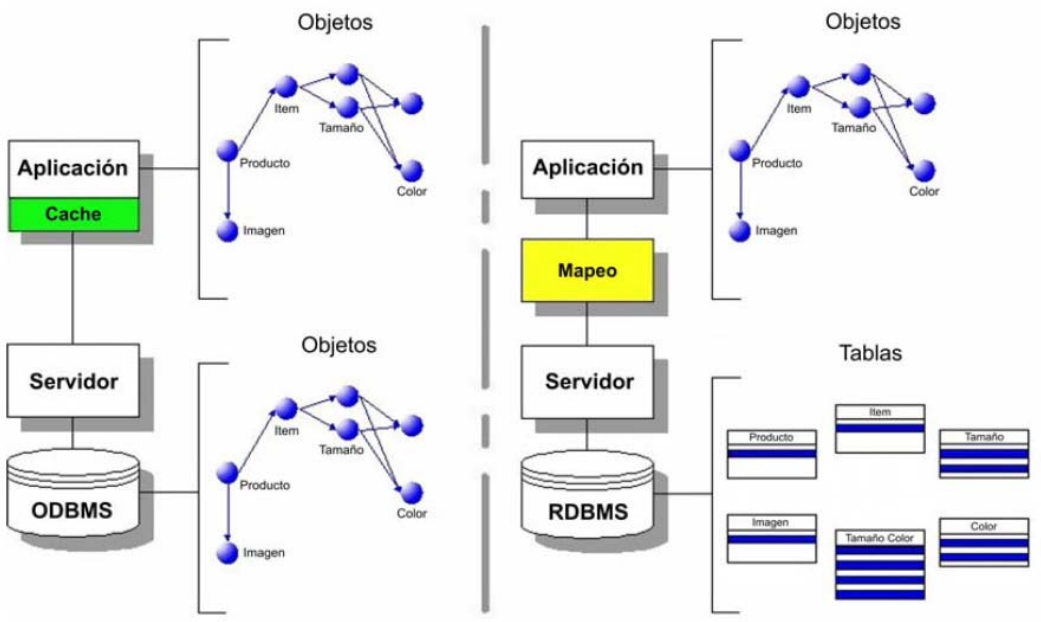
- Por falta de conocimiento, muchas veces se llega a tomar una decisión equivocada sobre cómo manejar la persistencia de la información en nuestra aplicación, esto trae como consecuencia, bajo rendimiento, mayor tiempo en el desarrollo, etc.

- Dada la cantidad de *Framework's ORM* que existen en la actualidad, se hace necesario tener una guía que exprese sus diferencias y ambientes donde podamos utilizarlos.

1.2. Justificación del problema

La persistencia de la información es muy importante en las aplicaciones y si estas están diseñadas orientadas a objetos, la persistencia solo se puede lograr de dos formas: serializando los objetos o guardando la información en una base de datos. Como se mencionaba anteriormente, las bases de datos más populares obedecen al modelo relacional. De allí proviene la idea de utilizar una herramienta que permita de forma fácil el mapeo objeto relacional.

Figura 1 Almacenamiento directo vs. mapeo de objetos



Fuente: (Viscosu)

1.3. Descripción del problema

En la actualidad existen diversidad de herramientas ORM, los cuales pretenden dar solución al problema del manejo de persistencia en las aplicaciones, cada una de estas herramientas tienen características especificas, particularidades, ventajas y desventajas, por lo que el ingeniero de software debe conocerlas; para tener la capacidad de tomar decisiones en cuanto a que herramienta utilizar o si realmente es necesario utilizar alguna.

1.4. Delimitación del problema

Se realizará una descripción y un análisis comparativo entre Hibernate, e Ibatis, tomando en cuenta las siguientes características: reusabilidad, persistencia transitiva, facilidad de uso, curva de aprendizaje, facilidad de pruebas, escalabilidad, herramientas de apoyo, integración, seguridad, facilidad

de refactorización, concurrencia, rendimiento, caché, carga retardada, facilidad de debug (errores SQL), soporte para transacciones e intrusión.

2. MARCO TEÓRICO

2.1. Mapeo de objetos al modelo relacional

2.1.1. Generalidades[1]

La persistencia de la información es la parte más crítica en una aplicación de software. Si la aplicación está diseñada con orientación a objetos, la persistencia se logra por: serialización del objeto o almacenando en una base de datos.

El modelo de objetos difiere en muchos aspectos del modelo relacional. La interface que une esos dos modelos se llama marco de mapeo relacional-objeto (ORM en inglés). Marcos de trabajo como Java o .Net han popularizado el uso de modelos de objetos (UML) en el diseño de aplicaciones dejando de lado el enfoque monolítico de una aplicación.

Las bases de datos relacionales son las más utilizadas hoy en día. Oracle, SQLServer, Mysql, Postgress son los DBMS más usados.

En el momento de persistir un objeto, normalmente, se crea una conexión a la base de datos, se crea una sentencia SQL parametrizada, se asignan los parámetros y recién allí se ejecuta la transacción. Imaginemos que tenemos un objeto con varias propiedades, además de varias relaciones, ¿como las asociamos relacionalmente? ¿Cómo las almacenamos? ¿Automáticamente, manualmente? ¿Qué pasa con las llaves secundarias?

[1] (Pizarro, 2005)

Ahora, si necesitamos recuperar los datos persistidos. ¿Cargamos únicamente el objeto? ¿Cargamos también las asociaciones? ¿Cargamos el árbol completo? Y si los mismos objetos están relacionados con otros, ¿se cargan x veces hasta satisfacerlos?

Está demostrado que un 35% de tiempo de desarrollo de software está dedicado al mapeo entre objeto y su correspondiente relación.[2] Como se ve, la incongruencia entre los dos modelos aumenta a medida que crece el modelo de objetos. Hay varios puntos por considerar:

- Carga perezosa.
- Referencia circular.
- Caché.
- Transacciones.

Un buen ORM permite:

- Resolver la mayor parte de transacciones entre la base de datos y la aplicación.

- Mapear clases a tablas, propiedad a columna, clase a tabla.

- Persistir objetos. A través de un método Orm. *Save(objeto).*

- Generar el SQL correspondiente para persistir objetos

- Recuperar objetos persistidos. A través de un método/objeto = *Orm.Load (objeto.class, clave_primaria).*

[2] (Pizarro, 2005)

- Recuperar una lista de objetos a partir de un lenguaje de consulta especial a través de un método.

2.1.2. Inconsistencias entre el modelo relacional y el de objetos[3]

Sabemos que las tablas tienen atributos simples, o sea, tipos definidos previamente por los arquitectos del software. Por otro lado, un objeto tiene tanto atributos simples como aquellos definidos por el usuario, que en sí es otro objeto más.

La incongruencia entre el modelo relacional y el de objetos es la diferencia en la forma de representar atributos de los dos modelos. Así en uno tenemos una representación tabular, mientras que en otro tenemos una representación jerárquica. La incongruencia entre la tecnología de objetos y la relacional, obliga al programador a mapear el esquema de objetos a un esquema de datos.

Hablamos de que los objetos deberían almacenarse en una base de datos relacional, ahora, una tabla mantiene relacionados los atributos que contiene, mientras un modelo de objetos tiene una jerarquía en árbol. Para realizar dicho mapeo, se utiliza una capa extra muy fina pero suficiente para servir como un puente entre los dos modelos, además es necesario agregar código a los objetos de negocios, código que impacta en la aplicación.

En la gran mayoría de nuestras aplicaciones, hacer persistente la información implica accesos algún tipo de base de datos relacional. Esto representa un problema fundamental pues algunas veces el diseño de datos relacionales y los modelos orientados a objetos suelen compartir estructuras.

[3] (Pizarro, 2005)

Las bases de datos relacionales, tiene una estructura muy distinta a la forma en que están organizados los objetos, la primera tiene una configuración de forma tabular mientras que la segunda se organiza en forma de un árbol jerárquico. Debido a esta diferencia, los desarrolladores han intentado construir un puente entre ambas tecnologías.

2.1.3. Mapeo de objetos[4]

Un objeto está compuesto de propiedades y métodos. Como las propiedades, representan a la parte estática de ese objeto, son las partes que son persistidas.

Cada propiedad puede ser simple o compleja. Por simple, se entiende que tiene algún tipo de datos nativos como por ejemplo entero, de coma flotante, cadena de caracteres. Por complejo, se entiende algún tipo definido por el usuario ya sea objetos o estructuras.

En el modelo relacional, cada fila en la tabla se mapea a un objeto, y cada columna a una propiedad. Normalmente, cada objeto de nuestro modelo, representa una tabla en el modelo relacional. Así que cada propiedad del objeto se mapea a cero, 1 o más de 1 columna en una tabla. Cero columnas, pues se puede dar el caso de propiedades que no necesitan ser persistidas, el ejemplo más sencillo, es el de las propiedades que representan cálculos temporarios. Lo más común que sucede es que cada propiedad del objeto se mapea a una única columna, se debe tener control del tipo de datos.

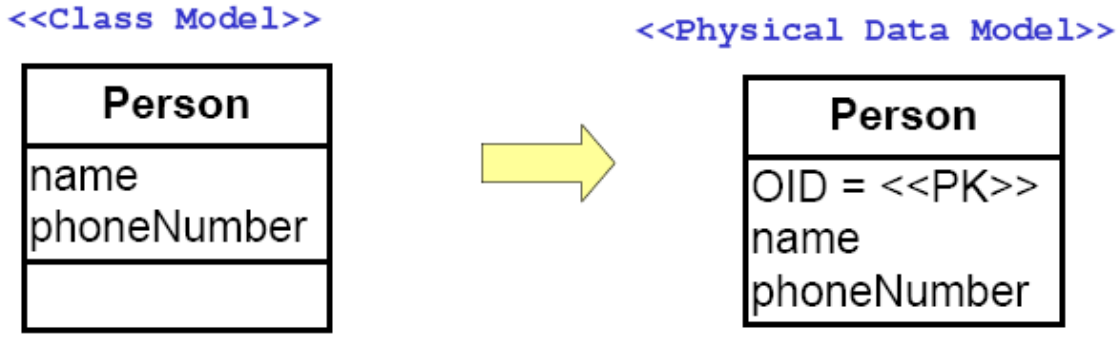
[4] (Pizarro, 2005)

Una propiedad puede ser mapeada en más de una columna. También se da el caso contrario de que en la tabla se tienen otros atributos que no se representan en el objeto real. Estos atributos suelen estar disponibles para el manejo de la concurrencia, auditorías, etcétera.

Cada columna de la tabla debería respetar el tipo de datos con su correspondiente en la propiedad del objeto. Aunque a veces, por optimización, son necesarios algunos ajustes en la base de datos relacional. Una de las razones principales, es que el rendimiento aumenta considerablemente si se trabaja con valores numéricos que con caracteres. En caso de no poder representarse el tipo de datos, debemos tener en cuenta la pérdida de información. Por ejemplo, si almacenamos un valor de punto flotante como cadena de caracteres, al reconstruirlo en propiedad, es posible la pérdida de información.

Para acceder a las propiedades normalmente se usan métodos especiales llamados getters y setters.

Figura 2 Mapeo de clases a objetos



Fuente: (Sevilla)

2.1.3.1. Identidad del objeto

Para distinguir cada fila de las otras, se necesita un identificador de objetos (OID) que es una columna más. Este identificador no es necesario en memoria, porque la unicidad del objeto queda representada por la unidad de la posición de memoria que ocupa. El OID siempre representa la clave primaria en la tabla. Normalmente es numérico por razones de rendimiento.

Cuando se desea mapear objetos, como lo vimos anteriormente se mapea solo la parte estática, así cada objeto se mapea a una fila en la tabla, cada propiedad del objeto se mapea por lo general a una columna. Existe una columna especial que identifica al objeto en la tabla llamada Oid.

2.1.4. Mapeo de relaciones

En esta sección se verá cómo se mapean las relaciones entre los objetos. Por relación se entiende asociación, herencia o agregación. Cualquiera sea el caso, para persistir las relaciones se usan transacciones, ya que los cambios pueden incluir varias tablas.

2.1.4.1. Asociaciones

Una regla general para el mapeo es respetar el tipo de multiplicidad en el modelo de objetos, y en el modelo relacional. Así una relación 1-1 en el modelo de objetos, deberá corresponder a una relación 1-1 en el modelo relacional.

Las asociaciones, a su vez, están divididas según su multiplicidad y su navegabilidad. Según su multiplicidad, pueden existir asociaciones 1-1, 1-n, m-n.

Según su navegabilidad, se tiene unidireccional o bidireccional. Se pueden dar las seis combinaciones posibles. Una aclaración importante es que en las bases de datos relacionales, todas las asociaciones son bidireccionales, también es un factor de la incongruencia de modelos.

2.1.4.1.1. Asociación clave foránea

Para mapear las relaciones, se usan los identificadores de objetos (OID). Estos OID son la llave primaria de la tabla relacionada y se agregan como una columna más en la tabla donde se quiere establecer la relación. Dicha columna es una clave foránea a la tabla con la que se está relacionada. Así, queda asignada la relación, recordar que las relaciones en el modelo relacional son siempre bidireccionales. El patrón se llama Foreign Key Mapping.

2.1.4.1.2. Multiplicidad 1-1

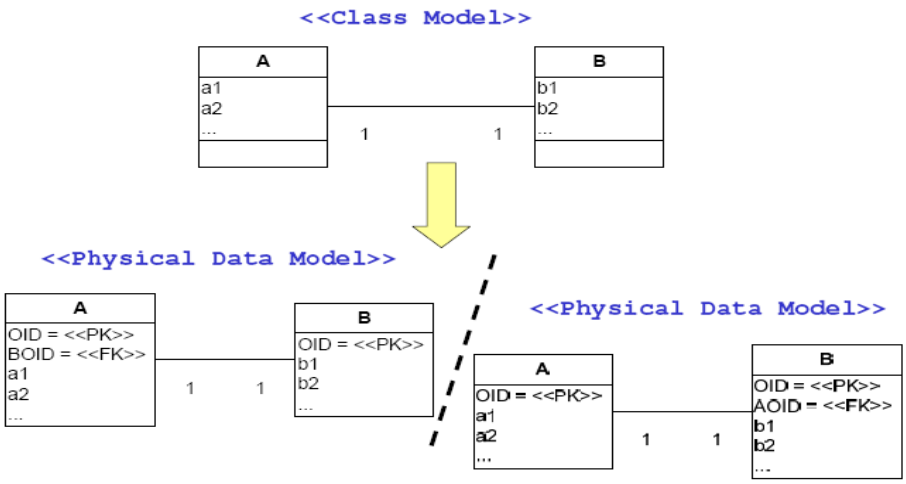
Cada objeto A está asociado con cero o un objeto B y, cada objeto B está asociado con cero o un objeto A.

En el modelo de objetos, este tipo de relación, se representa como una propiedad de tipo de datos de usuario. Así es común accederla vía setObjetoB(), getObjetoB().

En el modelo relacional, cualquiera de las dos tablas relacionadas implementará una columna con el Oid de la otra tabla, y esta columna será la clave foránea para relacionarlas.

No obstante, este tipo de relación al ser un subconjunto de 1-n, y ésta a su vez de n-m, puede mapearse como esta última a través de una tabla asociativa o implementando la clave en la otra tabla relacionada (ver Navegabilidad unidireccional).

Figura 3. Relación Uno a Uno



Fuente: (Sevilla)

2.1.4.1.3. Multiplicidad 1-n[5]

Cada objeto A puede estar asociado con cero o más objetos B, pero cada objeto B está asociado con cero o un objeto A.

[5] (Pizarro, 2005)

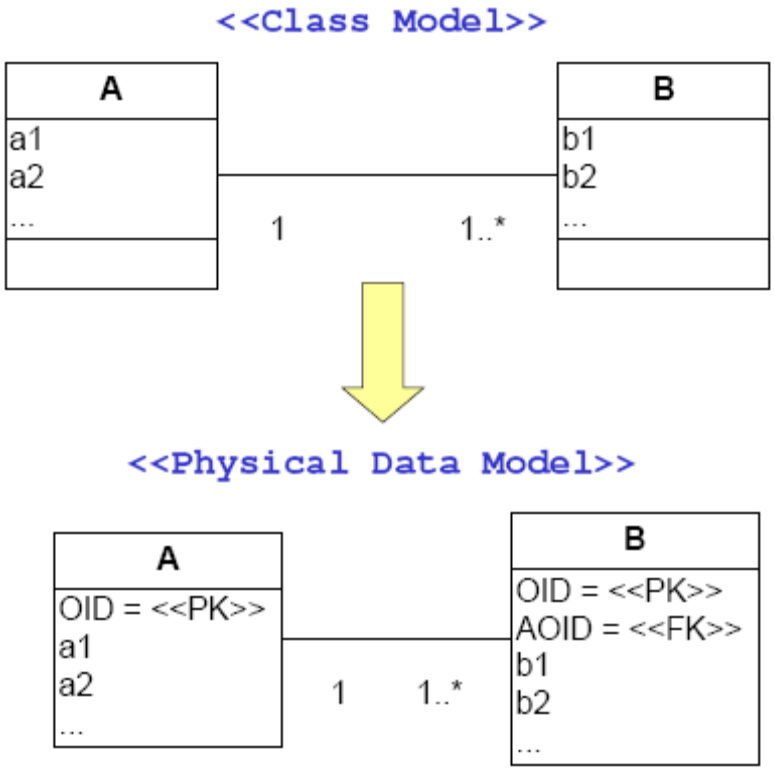
En el modelo de objetos, este tipo de relación se representa como una colección o array de tipo de datos de usuario. Así es común accederla vía addObjectosB(ObjetoB), removeObjetosB(ObjetoB).

En el modelo relacional se pueden seguir dos estrategias para establecer la relación:

- Implementando la clave foránea OID en la tabla “muchos” a la tabla “uno”.

- Implementando una tabla asociativa, convirtiendo la relación en muchos a muchos.

Figura 4. Relación de uno a muchos



Fuente: (Sevilla)

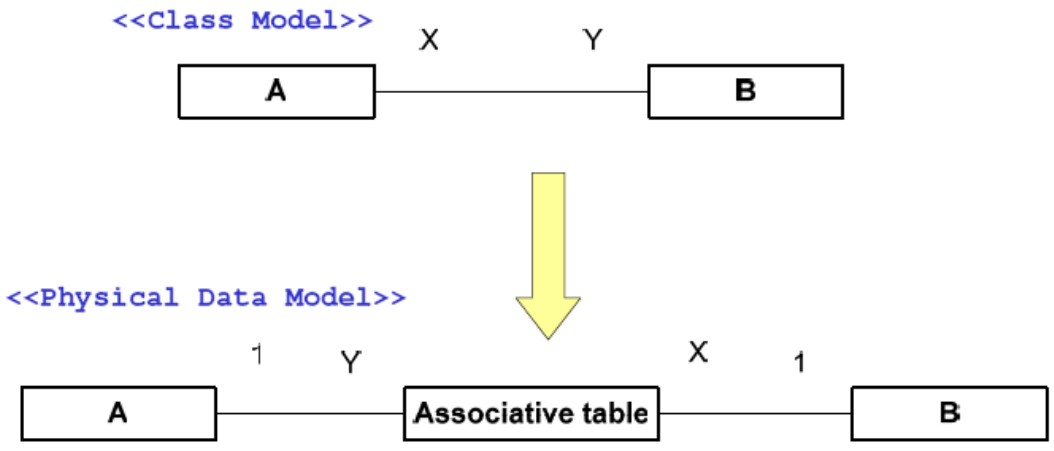
2.1.4.1.4. Multiplicidad n-m

Cada objeto A está asociado con cero o más objetos B, y a su vez, cada objeto B está asociado a cero o más objetos A.

En el modelo de objetos, esta relación será similar a la anterior. Será implementada mediante una colección o array en ambos objetos.

En el modelo relacional, se usa una tabla auxiliar asociativa para representar la relación y cuyo único objetivo es relacionar dos o más tablas. Dicha tabla tendrá al menos dos columnas, cada una representando la clave foránea a las dos tablas que relaciona. Con esto se transforma la relación n-m a dos relaciones (1-n y 1-m).

Figura 5. Relaciones muchos a muchos



Fuente: (Pizarro, 2005)

2.1.4.1.5. Navegabilidad unidireccional[6]

Una relación unidireccional es aquella que sabe con qué objetos está relacionado, pero dichos objetos no conocen al objeto original.

En el modelo de objetos, la navegabilidad unidireccional está representada por una flecha en el extremo de la asociación. Por lo tanto, el objeto original es quien implementa los métodos de acceso, mientras que el secundario no.

En el modelo relacional, la navegabilidad está representada por una llave foránea y depende de la multiplicidad la tabla en la que se implementa. Como se mencionó anteriormente, todas las relaciones en la base de datos son bidireccionales. Por ejemplo, para implementar una relación 1-1 podríamos haber puesto la clave foránea en cualquiera de las dos tablas, pues al hacer la unión (INNER JOIN, LEFT JOIN, RIGHT JOIN) solo indicamos la clave foránea y su unión.

2.1.4.1.6. Navegabilidad bidireccional

Una relación bidireccional existe cuando los objetos en ambos extremos de la relación saben del objeto en el extremo contrario.

En el modelo de objetos se representa por una línea que los une, sin flechas que indiquen direccionalidad. Ambos objetos deben implementar los métodos de acceso hacia el objeto con el cual está relacionado.

[6] [12] (Pizarro, 2005)

2.1.4.1.7. Asociación recursiva

Una asociación recursiva (o reflexiva) es aquella donde ambos extremos de la asociación es una misma entidad (clase, tabla).

En el modelo de objetos se representa por una línea asociativa que empieza y termina en la misma clase. Puede tener cualquier multiplicidad y navegabilidad.

En el modelo relacional, se representa por una clave foránea a sí misma, y depende de la multiplicidad si esa clave se implementa en la misma tabla o en una tabla asociativa.

2.1.4.2. Agregación / Composición

Como ya sabemos, una asociación es una relación débil e independiente entre dos objetos. Una agregación es una relación más fuerte que una asociación pero aún independiente. Una composición es a la vez una relación fuerte y dependiente entre dos objetos. [7]

Con lo anterior asumimos que tanto una asociación, agregación o composición se mapea en el modelo relacional como una relación.

La agregación normalmente se mapea como una relación n-m, o sea que hay una tabla auxiliar para mapear la relación.

La composición puede mapearse como una relación 1-n. Donde los objetos compuestos mantienen una relación con el objeto compositor. A nivel

[7] (Pizarro, 2005)

relacional, indica que la tabla de los objetos compuestos tiene una columna con la clave foránea al objeto que los compuso.

En el modelo objetos, tanto las agregaciones como las composiciones se corresponden con un array o una colección de objetos.

2.1.4.3. Herencia

Como vimos anteriormente, las asociaciones funcionan en ambos modelos, objeto y relacional. Para el caso de la herencia se presenta el problema que las base de datos relacionales no la soportan. Así es que somos nosotros quienes debemos modelar como se verá la herencia en el modelo relacional. Una regla a seguir es que se debe minimizar la cantidad de joins posibles. Existen tres tipos de mapeos principales: modelar la jerarquía a una sola tabla, modelar la jerarquía completa en tablas, mapear cada tabla en tablas concretas. La decisión estará basada en el rendimiento y en la escalabilidad del modelo. [8]

2.1.4.3.1. Mapeo de la jerarquía a una tabla.

Se mapean todos los atributos, de todas las clases del árbol de herencia en una única tabla. En el mapeo se agregan dos columnas de información oculta la llave primaria de la tabla OID y el tipo de clase que es cada registro. El tipo de clase se resuelve con una columna carácter o numérica entera. Para los casos más complejos se necesita de varias columnas booleanas (si/no). [9]

Por ejemplo, la jerarquía persona (que es abstracta), cliente y empleado quedarían en una única tabla llamada persona (es recomendable colocarle el

[8] (Pizarro, 2005)
[9] (Pizarro, 2005)

nombre de la raíz de la estructura). A la tabla se le agrega el tipo de clase que es, así tenemos la columna TipoPersona donde C será cliente, E empleado, D para aquellos que son clientes y empleados al mismo tiempo.

Las ventajas son:

- Aproximación simple.

- Cada nueva clase, simplemente se agregan columnas para datos adicionales.

- Soporta el polimorfismo cambiando el tipo de fila.

- El acceso a los datos es rápido porque los datos están en una sola tabla.

- El reporte es fácil porque los datos están en una tabla.

Las desventajas son:

- Mayor acoplamiento entre las clases. Un cambio en una de ellas puede afectar a las otras clases ya que comparten la misma tabla.

- Se desperdicia espacio en la base de datos (por la tanto disminuye el rendimiento), para aquellas columnas en las que son de las clases derivadas.

- La tabla crece rápidamente a mayor jerarquía.

2.1.4.3.2. Mapeo de cada clase concreta a una tabla.[10]

Cada clase concreta es mapeada a una tabla. Cada tabla incluye los atributos heredados más los implementados en la clase. La clase base abstracta entonces, es mapeada en cada tabla de las derivadas.

Las ventajas son:

- Reportes fáciles de obtener ya que los datos necesarios están en una sola tabla.

- Buen rendimiento para acceder a datos de un objeto.

Las desventajas son:

- Pobre escalabilidad, si se modifica la clase base, se debe modificar en todas las tablas de las clases derivadas para reflejar ese cambio. Por ejemplo, si a la clase Persona le agregamos el atributo de Estado Civil, debemos agregarlo en las tablas Cliente y en Empleado.

- Actualización compleja, si un objeto cambia su rol, se le asigna un nuevo OID y se mueven los datos a la tabla correspondiente.

- Se pierde integridad en los datos, para el caso en que el objeto tenga los dos roles. Por ejemplo, cliente y empleado a la vez.

Cuando usar:

Cuando el campo de tipo no sea común.

[10] (Pizarro, 2005)

2.1.4.3.3. Mapeo de cada clase a su propia tabla

Se crea una tabla por cada clase de la herencia, aún la clase base abstracta. Se agregan también las columnas para el control de la concurrencia o versión a cualquiera de las tablas.

Cuando se necesita leer el objeto heredado se unen (join) las dos tablas de la relación o se leen las dos tablas en forma separada. Las llaves primarias de todas las tablas heredadas, será la misma que la tabla base. A su vez, también serán claves foráneas hacia la tabla base.

Para simplificar las consultas, a veces será necesario agregar una columna en la tabla base indicando los subtipos de ese elemento o agregando varias columnas booleanas. El mismo efecto se logra y más eficiente, a través de vistas.

Las ventajas son:

- Fácil de entender, porque es un mapeo uno a uno.

- Soporta muy bien el polimorfismo, ya que tiene almacenado los registros en la tabla correspondiente.

- Fácil escalabilidad, se pueden modificar atributos en la superclase que afectan a una sola tabla. Agregar subclases es simplemente agregar nuevas tablas.

Las desventajas son:

- Hay muchas tablas en la base de datos.

- Menor rendimiento, pues se necesita leer y escribir sobre varias tablas.

- Reportes rápidos difíciles de armar, a menos que se tengan vistas.

Usar cuando exista mucha relación entre los tipos o cuando el cambio de tipos sea muy común.

2.2. Persistencia[11]

Persistencia es la habilidad que tiene un objeto de sobrevivir al ciclo de vida del proceso en el que reside. Los objetos que mueren al final de un proceso se llaman transitorios.

Como se mencionó anteriormente, se tiene una incongruencia entre el mundo orientado a objetos y las bases de datos relacionales. Para reducir esa incongruencia recurrimos a una capa auxiliar que mapeará entre los dos mundos, adaptándolo según las especificaciones hechas. Esa capa auxiliar se denomina ORM, Object Relational Mapping.

Los ORM nacieron a mediados de la década del 90, se hicieron masivos a partir de la masificación de Java, por esa razón, los *framework's* más populares hoy en día en .Net son adaptaciones del modelo pensado para Java.

Imaginemos que queremos cargar un objeto persistido en memoria, los pasos a seguir serían: abrir la conexión a la DB, construir una sentencia SQL parametrizada, llenar los parámetros (por ejemplo la clave primaria), recién allí ejecutarlo como una transacción y cerrar la conexión a la base de datos.

[11] (Pizarro, 2005)

Con un *framework*, la tarea se reduce a: abrir una sesión con la base de datos, especificar el tipo de objeto que queremos (y su clave primaria correspondiente), cerrar la sesión.

No obstante, el *framework* debería resolver los siguientes puntos:

- Transacciones. ¿Se hacen todos los cambios sin excepción, o, se hace nada?

- Caché. ¿se almacenan en memoria los objetos usados? ¿se almacenan las consultas SQL hechas?

- Carga retardada. cuándo se carga el objeto en memoria, ¿se cargan en memoria todas sus relaciones? ¿se cargan en memoria sus campos menos usados (por ejemplo los BLOBS de gran tamaño)?

- Referencia circular. Si el objeto está relacionado dos veces con el mismo objeto, ¿se carga dos veces el objeto relacionado?

- OID. ¿asignamos manualmente o automáticamente las claves primarias?

Estudios muestran que aproximadamente, el 35% de desarrollo de software está dedicado al mapeo entre el modelo de objetos y su correspondiente modelo relacional.

2.2.1. Patrón CRUD[12]

Acrónimo de Create-Read-Update-Delete. Conocido como el padre de todos los patrones de capa de acceso. Describe que cada objeto debe ser

[12] (Pizarro, 2005)

creado en la base de datos para que sea persistente. Una vez creado, la capa de acceso debe tener una forma de leerlo para poder actualizarlo o simplemente borrarlo.

Teóricamente el borrado de objetos debería quedar a cargo de la misma base de datos. Pero un recolector de objetos "basura" (garbage collector) en una base de datos gigante afecta en gran medida el rendimiento. Por ello es que la tarea de borrado queda delegada al programador.

2.2.2. Caché[13]

En la mayoría de las aplicaciones, se aplica la regla del 80-20 en cuanto al acceso a datos, el 80% de accesos de lectura accede al 20% de los datos de la aplicación. Esto significa que hay un conjunto de datos dinámicos que son relevantes a todos los usuarios del sistema, y por lo tanto accedido con más frecuencia.

Las aplicaciones empresariales de sincronización de caché normalmente necesitan escalarse para manejar grandes cargas transaccionales, así múltiples instancias pueden procesar simultáneamente. Es un problema serio para el acceso a datos desde la aplicación, especialmente cuando los datos involucrados necesitan actualizarse dinámicamente a través de esas instancias. Para asegurar la integridad de datos, la base de datos comúnmente juega el rol de árbitro para todos los datos de la aplicación. Es un rol muy importante dado que los datos representan la proporción de valor más significante de una organización. Desafortunadamente, este rol también no está fácilmente distribuido sin introducir problemas significantes, especialmente en un entorno transaccional.

[13] (Pizarro, 2005)

Es común para la base de datos usar replicación para lograr datos sincronizados, pero comúnmente ofrece una copia offline del estado de los datos más que una instancia secundaria activa. Es posible usar bases de datos que puedan soportar múltiples instancias activas, pero se pueden volver caras en cuanto a rendimiento y escalabilidad, debido a que introducen el bloqueo de objetos y la latencia de distribución. La mayoría de los sistemas usan una única base de datos activa, con múltiples servidores conectada directamente a ella, soportando un número variables de clientes.

En esta arquitectura, la carga en la base de datos incrementará linealmente con el número de instancias de la aplicación en uso, a menos que se emplee algún mecanismo de caché.

Pero implementando un mecanismo de caché en esta arquitectura puede traer muchos problemas, incluso corrupción en los datos, porque la caché en el servidor uno no sabrá sobre los cambios en el servidor dos.

2.2.2.1. Identidad en la caché

En la siguiente porción de código veremos algunos aspectos de la caché.

```
Persona prsnPerezJuan = new Persona(”Perez”, “Juan”)
Persona prsnPerezAlberto = new Persona(”Perez”, “Alberto”)
Persona prsnPerez = ORM.LoadByApellido(“Perez”) //¿?
```

Se han creado dos instancias en memoria de la clase Persona, o sea que para el objeto tenemos dos objetos diferentes usando el mismo apellido. Si el apellido es la clave primaria en la base de datos, tendríamos problemas cuando la aplicación trate de escribir la segunda instancia en la base de datos. Por eso debemos tener cuidado con la identidad del objeto y sus diferentes notaciones en la base de datos y en el programa. La solución para esto es aplicar la identidad en la misma caché.

Normalmente, se usa una tabla de hashing en base a las claves primarias.

Imaginemos un código como el siguiente:

```
Persona prsnJuan = (Persona) ORM.Load(“PersonaOID”,100)
Persona prsnPerez = (Persona) ORM.Load(“PersonaOID”,100)
If (prsnJuan != prsnPerez)
```

En la porción de código anterior tenemos 2 variables que deberían hacer referencia al mismo objeto: Juan Pérez. Si la capa de acceso no es buena, habrá cargado dos instancias del objeto en memoria de los mismos datos en la base de datos. Para evitar que suceda, necesitamos tener un mecanismo que verifique que el objeto ya está cargado en memoria desde la base de datos.

La primera vez que cargamos el objeto, la capa de acceso lo lee desde la base de datos; la segunda vez, el método Load necesita verificar si la persona con OID igual a 100 ya está cargada en memoria. Si es así, solo retorna la referencia al objeto en memoria creado por la primera llamada.

2.2.3. Carga de las relaciones[14]

Uno de los factores que debemos decidir luego del mapeo es si siempre se cargan todos los objetos relacionados a uno principal. La respuesta más probable es no, porque las redes de relaciones tienden a ser más compleja y la cadena de relación tiende a ser más larga en la vida real. Imaginemos que por defecto se carga cada relación de un objeto. En el caso de una gran base de datos se volverá grandísimo y habrá pérdida de rendimiento. La solución para este problema es conocida como “Carga retardada” de las relaciones, y se

[14] (Pizarro, 2005)

implementa por algún tipo de objetos con “Patrón Proxy” que lanzan la carga cuando se acceden.

Para lograrlo, el objeto tiene un método accesor *(“Get”)* cuyo único propósito es proveer el valor de un atributo simple, que verifica a ver si el atributo ha sido inicializado y si no es así lo lee desde la base de datos.

Otro uso común de carga retardada es la generación de reporte y objetos que se dan como resultados de una búsqueda, casos en los cuales se necesita solo un subconjunto de datos del objeto.

Lo mismo sucede para aquellos campos grandes y menos usados. Por ejemplo, si se almacena la foto de una persona ocupará alrededor de 100k mientras que el resto de los atributos no llegan, en total, a 1k; y raramente son accedidas.

Pero también, habrá veces en la que la “Carga Directa” de las relaciones se prefiera. Con ella, cada vez que se cargue un objeto querríamos tener algunas de sus relaciones cargadas obviando la necesidad de realizar otro acceso a la base de datos.

2.2.3.1. Proxy (listado y selección, reporte)[15]

Es normal que la aplicación presente un listado de un objeto particular, mostrando campos como OID, nombre y un resumen breve de otros campos; el objetivo de ese listado es darle al usuario la posibilidad de seleccionarlo para ver información más detallada. Algo similar se da en la generación de reportes.

[15] (Pizarro, 2005)

Si siguiéramos aplicando un ORM, quizás produjéramos una sobrecarga innecesaria en la capa de persistencia con cargas no triviales en la base de datos.

La aproximación más básica es no manejarla con los mecanismos de persistencia.

Algunos consejos serían:

1. Evitar mostrar al usuario todos los objetos al principio. Dejarlo como posibilidad, pero mientras tanto, darle una opción de búsqueda. Mostrar ventanas de selección en vez de cajas de selección (combo box). Da una mayor usabilidad y puede incluir un buen resumen.

2. La capa de persistencia debe proveer el manejo de solo lectura en masa.

3. Retornando los datos en forma de filas o tuplas.

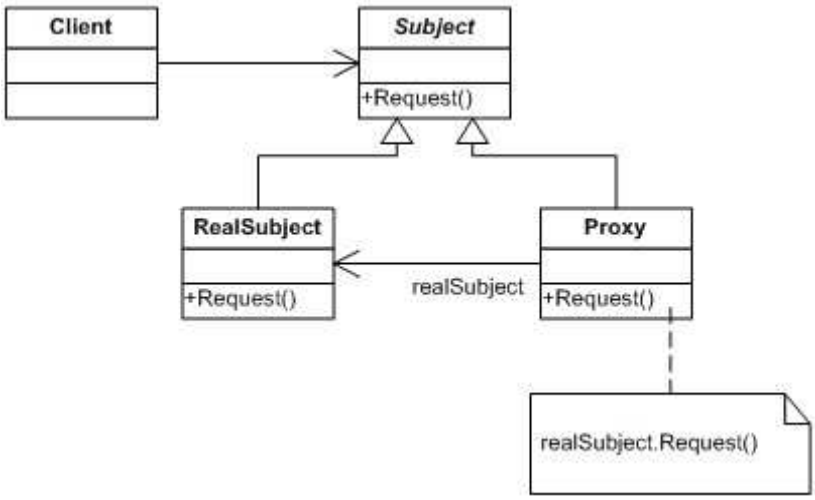
4. La consulta a la base de datos debería usar un cursor. Ir retornando los cursores gradualmente al cliente. También debemos tener en cuenta, que la mayoría de las bases proveen cursores hacia delante; con esto la capa de persistencia debería enviar al buffer internamente así el cliente puede volver atrás.

5. A veces, el reporte y la selección requieren que los datos sean unidos a través de múltiples tablas. Por defecto son las uniones hacia la izquierda las que se usan en las asociaciones incluyendo los roles de las tablas unidas; nombre de columnas no ambiguas, alias únicos cuando una tabla se une a sí misma; típicamente para estructuras de jerarquía o árbol.

2.2.3.2. Proxy

Un proxy es un objeto que representa a otro objeto pero no incurre en la misma sobrecarga que al objeto representado. Un proxy contiene bastante información para que, tanto la aplicación como el usuario puedan identificar el objeto. Por ejemplo el proxy para el objeto Persona debería contener su oid así la aplicación pueda identificarlo y el apellido así el usuario puede reconocer a quien representa el objeto proxy. Los proxies se usan, generalmente cuando se despliegan los resultados de una consulta, de la cual el usuario elegirá solo una o dos. Cuando el usuario elige el objeto proxy de la lista, recién ahí se trae automáticamente el objeto real del *framework ORM*, el cual es más grande que el proxy. Por ejemplo, el objeto real de Persona incluye dirección, foto de la persona. Con el patrón proxy solo se da la información que el usuario necesita.

Figura 6. Diagrama UML del Patrón Proxy.



Fuente: http://www.dofactory.com/Patterns/Diagrams/proxy.gif

2.2.4. Transacción[16]

Es necesario proteger mediante una transacción la información almacenada en una base de datos. Esto permite múltiples inserciones, modificaciones y borrados con la seguridad de que todo o se ejecuta o falla, como si fuera una sola entidad coherente.

Las transacciones también pueden ofrecer protección de concurrencia; el bloqueo pesimista de tuplas mientras los usuarios están trabajando en ellos, evita que otros usuarios comiencen con sus cambios.

Sin embargo, el mecanismo de transacción de la base de datos tiene algunas limitaciones:

- Cada transacción requiere una sesión separada, para permitir que los usuarios abran ventanas relacionadas a su trabajo requeriría de licencias extras o que las ventas estén limitada a acceso de solo lectura.

- Una alta aislación en la transacción y bloqueo basado en páginas puede evitar que otros usuarios accedan a los datos que deberían estar legítimamente permitidos.

Una alternativa es que los datos sean puestos en un buffer en la capa de persistencia, con todas las escrituras que se harán hasta que el usuario lo confirme. Esta transacción de base de datos se necesita solo durante la operación de escritura en masa (bulk operation), permitiendo ser compartida entre múltiples ventanas. Esto requiere un esquema de bloqueo optimista donde las tuplas son chequeadas mientras se escriben.

[16] (Pizarro, 2005)

Obviamente, esta escritura en masa corre protegida por la integridad referencial. Esas restricciones especifican los requerimientos lógicos según el caso: la tupla debe existir antes de que se la relacione y las tuplas relacionadas a otra deben ser borradas antes de que se borre la tupla objetivo. Esto se logra mediante un mecanismo en la base de datos que mantiene correctamente las dependencias entre las tuplas.

2.2.5. Concurrencia

La capa de persistencia debe permitir que múltiples usuarios trabajen en la misma base de datos y proteger los datos de ser escritos equivocadamente. También es importante minimizar las restricciones en su capacidad concurrente para ver y acceder información de las tablas.

La integridad de datos es un riesgo cuando dos sesiones trabajan sobre la misma tupla: la pérdida de alguna actualización está asegurada. También se puede dar el caso, cuando una sesión está leyendo los datos y la otra los está editando de hacer una lectura inconsistente.

Hay dos técnicas principales para el problema: bloqueo pesimista y bloqueo optimista. Con el primero, se bloquea todo acceso desde que el usuario empieza a cambiar los datos hasta que se haga COMMIT a la transacción. Mientras que en el optimista, el bloqueo se aplica cuando los datos son aplicados y se van verificando mientras los datos son escritos.

2.2.5.1. Bloqueo optimista[17]

En este enfoque, cuando se detecta un conflicto entre transacciones concurrentes, se cancela alguna de las transacciones.

Para resolver el problema, valida que los cambios a los cuales se les han dado COMMIT por una sesión no entran en conflicto con los cambios hechos en otra sesión. Una validación exitosa pre-COMMIT es obtener un bloqueo de los registros con una transacción simple.

Asume que la probabilidad de que aparezcan conflictos es baja: se espera que no sea probable que los usuarios trabajen sobre los mismos datos al mismo tiempo.

El bloqueo optimista desacopla la capa de persistencia de la necesidad de mantener una transacción pendiente, chequeando los datos mientras se escribe. Se usan varios esquemas de bloqueo optimista. Difieren en que campos son verificados; a veces se usa un campo de estampa de tiempo (timestamp) o un simple contador (counter). El bloqueo por estampa de tiempo no es confiable, hoy en día el hardware es cada vez más rápido y la cuantificación del tiempo llega al orden de los microsegundos. Los contadores van indicando la última versión guardada en la base de datos.

Generalmente, la aplicación iniciará una transacción, leerá los datos desde la base, cerrará su transacción, seguirá con las reglas de negocio involucradas para volver a iniciar una transacción, esta vez con los datos a ser escritos. En el caso de la estampa de tiempo, la capa de persistencia verificará que sea la misma que existe en la base de datos, escribirá los datos y

[17] (Pizarro, 2005)

actualizará la estampa de tiempo a la hora actual. Muy similar para el contador, difiere en que la actualización consiste en incrementar en 1 el campo.

2.2.5.2. Bloqueo pesimista[18]

Evita que aparezcan conflictos entre transacciones concurrentes permitiendo acceder a los datos a solo una transacción a la vez.

La aproximación más simple, consiste en tener una transacción abierta para todas las reglas de negocio involucradas. Hay que tener precaución con transacciones largas.

Por eso, se recomienda usar múltiples transacciones en las reglas de negocios. Imaginemos que varios usuarios acceden a los mismos datos dentro de una regla de negocios, uno de ellos hace COMMIT todo, mientras que otros usuarios no lo lograrán y fallarán. Dado que el conflicto se detecta cuando termina la transacción, las víctimas aplicarán todas las reglas de negocios, hasta que en el último instante sabrán que todo fallará, con lo cual fue una pérdida de tiempo.

El bloqueo pesimista evita el conflicto anterior por completo. Obliga a las reglas de negocios a adquirir el bloqueo de los datos antes de empezar a usarlo, así, la transacción se usa completamente sin preocuparse por los controles de concurrencia.

El bloqueo pesimista notifica a los usuarios tempranamente de la contención de los datos, pero para los propósitos de negocios la concurrencia es considerada altamente importante.

[18] (Pizarro, 2005)

2.2.6. Otros aspectos

En esta sección veremos otros aspectos a tener en cuenta en la elección de un *framework ORM*.

- **Referencia circular:** Se refiere a si el *framework* es capaz de detectar cual es el objeto que se está solicitando, sin tener que ir de nuevo a la base de datos.

- **Información oculta *(Shadow information)*:** Así como el OID hay muchas columnas de la tabla que no necesitan ser mapeadas a una propiedad del objeto. Estas columnas contienen información oculta para el modelo de objetos pero necesaria para el modelo relacional. En esta categoría entran los mecanismos de concurrencia: estampa de tiempo y versión de objeto. Al leer el objeto, se obtiene esta información que es ocultada al objeto pero mantenida por el *framework*.

- **Lenguaje de consulta (OQL - Object Query Language):** La obtención de varios objetos a través de un lenguaje especial es una de las características más apreciadas. Por ejemplo, obtener todos los "Juan Pérez" de una base de Personas. "SELECT Persona FROM Personas WHERE Nombre = 'Juan'". Hibernate con HQL es uno de los mejores.

- **Actualización en cascada:** La posibilidad de que modificaciones hechas a un objetos repliquen en los objetos relacionados.

- **Operaciones en masa:** Habrán veces que por razones de rendimiento, se querrá hacer una operación en masa. Por ejemplo, actualizar todos los objetos con nombre Juan a J. De la manera tradicional, deberíamos

leer cada objeto, modificarlos en memoria, y recién allí guardarlos de nuevo.

2.2.7. Software para mapeo objeto relacional

Esta sección contiene un listado de *Framework's ORM* para varios lenguaje usados en la actualidad.

Framework's ORM para Java:

- Carbonado
- Cayenne
- Ebean
- EclipseLink
- Enterprise Objects *Framework*
- Hibernate
- iBATIS
- Java Data Objects
- JPOX
- Kodo
- OpenJPA
- TopLink by Oracle
- WebObjects

Framework's ORM para .NET:

- Developer Express, eXpress Persistent Objects (XPO)
- ADO.NET Entity *Framework*
- Business Logic Toolkit for .NET
- Castle ActiveRecord, ActiveRecord
- EntitySpaces

- Gentle.NET
- Habanero
- Persistor.NET
- LightSpeed
- LLBLGen
- MX-Frame
- Neo
- Linq language integrated query
- ObjectMapper .NET
- Sooda
- Vanatec OpenAccess
- NHibernate

Framework's ORM para PHP:

- Horde Rampage Data Object
- ADOdb Active Record
- Doctrine (GNU LGPL)
- Metastorage
- Xyster *Framework*
- Propel
- definition in XML (XMI conversion tool)
- SilverStripe

Framework's ORM para Python:

- Storm (software)
- Django
- SQLAlchemy
- SQLObject

Framework's ORM para Ruby:

- Datamapper
- ActiveRecord, Parte de Ruby on Rails (open source)
- Sequel

Framework's ORM para Perl:

- Class::DBI
- Data::ObjectDriver
- DBIx::Class
- Rose::DB::Object

Framework's ORM para C++:

- LiteSQL, open source

Actualmente, una las herramientas ORM más populares son Hibernate e Ibatis, estas cuentan con una extensa documentación y grandes comunidades de usuarios, estas dos herramientas tienen el respaldo de dos grandes empresas de software, Jboss y Apache respectivamente. Por este motivo en los siguientes capítulos se analizaran dichas herramientas.

3. IBATIS

3.1. Qué es Ibatis?

Es un *framework* de mapeo objeto relacional, el cual facilita el uso de base de datos relacionales con Java, Ibatis relaciona objetos utilizando descriptores XML.

Para utilizar Ibatis, simplemente se necesita: Pojos, XML y SQL. Con Ibatis no es necesario tanto estudio, puesto que con unos simples pasos, se podrá sacar provecho a lo que Ibatis provee, simplicidad es una de las mejores ventajas de Ibatis sobre otras herramientas de mapeo Objeto Relacional.

3.2. Significado de iBATIS.

Hoy en día, la palabra “Ibatis” es sinónimo de SQL Maps y framework DAO (De igual forma “Xerox” es sinónimo de fotocopias). Pero la palabra Ibatis realmente tiene un significado, el cual está compuesto por dos palabras Internet + Ibatis.[19]

3.3. Características

Es posible categorizar la capa de persistencia en tres subcapas:

- La capa de **Abstracción** será el interfaz con la capa de la lógica de negocio, haciendo las veces de “facade” entre la aplicación y la persistencia. Se implementa de forma general mediante el patrón Data

[19] http://ibatis.apache.org/background.html

Access Object (DAO), y particularmente en Ibatis se implementa utilizando su *framework DAO* (ibatis-dao.jar). [20]

- La capa de *framework* de persistencia será la interfaz con el gestor de Base de Datos ocupándose de la gestión de los datos mediante un API. Normalmente en Java se utiliza JDBC; Ibatis utiliza su framework SQL-MAP (ibatis-sqlmap.jar). [32]

- La capa de ***Driver*** se ocupa de la comunicación con la propia Base de Datos utilizando un Driver específico para la misma. [32]

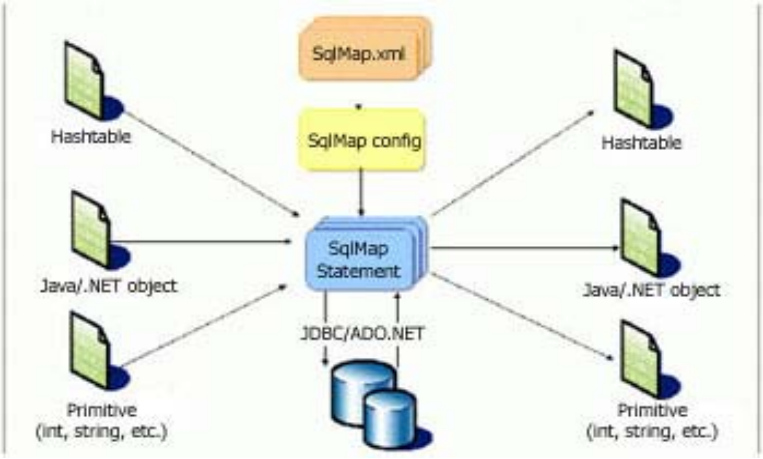
Toda implementación de Ibatis incluye los siguientes componentes:

- ***Data Access Object***: abstracción que oculta la persistencia de objetos en la aplicación y proporciona un API de acceso a datos al resto de la aplicación. [32]

- ***Data Mapper****:* proporciona una forma sencilla de interacción de datos entre los objetos Java y .NET y bases de datos relacionales. [32]

[20] http://es.wikipedia.org/wiki/IBATIS

Figura 7. Arquitectura Ibatis



Fuente: http://ibatis.apache.org/flow.jpg.

3.4. Por qué utilizar Ibatis?

Un *framework* como Ibatis ofrece la oportunidad de inyectar beneficios a nivel de arquitectura de nuestra aplicación. A continuación se presentan varios motivos, por los cuales se podría optar por una solución como Ibatis para el manejo de la persistencia en nuestra aplicación.

3.4.1. Simplicidad

Ibatis es considerado como uno de los *framework's* de persistencia más simple disponibles hoy en día. Simplicidad es el objetivo principal del equipo de diseño de Ibatis y toma prioridad sobre todo. Con Ibatis es fácil crear aplicaciones java puesto que trabaja como un jdbc, pero con mucho menos código. Se puede llegar a pensar que Ibatis es un jdbc descrito en código XML, pero esto es totalmente erróneo ya que Ibatis incluye muchos beneficios de arquitectura que un jdbc no posee, Ibatis también es fácil de entender por los administradores de bases de datos y programadores SQL. La descripción de

los archivos de configuración de Ibatis es fácil de crear para un programador con experiencia en SQL.

3.4.2. Productividad

El propósito principal de un buen *framework* es hacer al desarrollador más productivo. La razón principal de ser de un *framework* es ocuparse de las tareas comunes, reducir el código, y resolver problemas complejos de arquitectura. En un caso de estudio presentado por un grupo de Usuario java en Italia (www.jugsardegna.org/vqwiki/jsp/Wiki?IBatisCaseStudy), Fabrizio Gianneschi logró reducir el código en la capa de persistencia significativamente en un 62%.[21]

3.4.3. Rendimiento:

El tema del rendimiento es un debate entre los autores del *framework*, usuarios y vendedores comerciales. Generalmente si se compara el código jdbc con Ibatis con un loop de 1000 iteraciones, lógicamente el mejor rendimiento estará en el código jdbc, pero este tipo de aplicaciones no se manejan en el desarrollo de aplicaciones modernas, actualmente en el rendimiento de una aplicación, influye como se recuperan la información de la base de datos, la forma en que se obtienen y con qué frecuencia. Por ejemplo, utilizando una lista paginada de datos obtenidos dinámicamente de la base de datos puede incrementar el rendimiento pues no se están recuperando datos innecesarios. De forma similar utilizando características como la carga perezosa se pueden recuperar los datos hasta que sean necesarios, entre otras opciones que pueden ser consultadas en la documentación propia de Ibatis.

[21] (Begin, Goodin, & Larry, 2007)

3.4.4. Modularidad:

En una aplicación utilizando jdbc, se utilizan recursos como conexiones, resultset, etc. colocando estos en cualquier parte de nuestra aplicación. Ibatis nos ayuda a encapsular la capa de acceso a datos, administrando todos nuestros recursos de persistencia en un solo lugar, pudiendo separar de forma adecuada el acceso a datos de la capa de presentación y lógica de negocio. Con Ibatis siempre se trabajaran con objetos, no con resultSets independientes lo que obliga a la realización de las mejores prácticas en la programación.

3.4.5. División del trabajo

Algunos desarrolladores adoran escribir código SQL, pero a otros les gusta más escribir código en java, Ibatis ayuda a escribir el código SQL independiente de la aplicación, esto es debido a que el código SQL está separado del código fuente, de igual forma cuando el DBA desea optimizar la base de datos y solicita el SQL, se puede obtener de forma fácil y rápida.

3.5. Cuándo no utilizar Ibatis

Todos los *framework's* tienen sus ventajas y desventajas, así por ejemplo un jdbc tiene varias extensiones, es muy práctico, etc., pero existen otras aplicación que son mucho más fáciles y nos ahorran mucho trabajo, pero están construidas con muchas más restricciones y menos formas de aplicación.

A continuación se listan cuáles son las situaciones en las cuales no se recomienda utilizar Ibatis.

- Cuando siempre se tiene un control total de la aplicación. Si se posee el control total de la aplicación, entonces se está en la situación donde se puede utilizar una solución de mapeo objeto relacional completa como

Hibernate, se puede tener el beneficio en productividad que esta solución puede brindar.

- Cuando la aplicación requiere plenamente de SQL dinámico. Si la funcionalidad básica de la aplicación es la generación dinámica de SQL, Ibatis es la opción equivocada.

- Cuando no se está usando una base de datos relacional. Ibatis solo soporta mapeo de Objetos al modelo relacional, en cualquier otro caso no se podría utilizar este *framework*.

- Cuando simplemente no funciona. Como es bien sabido, un *framework* se crea con el principal objetivo de satisfacer las necesidades comunes de las aplicaciones, pero van a existir casos donde los requerimientos no se ajustan a los objetivos de Ibatis y el desarrollar una extensión, resulta muy complejo.

3.6. Ibatis Data Acces Objets

Data Access Object es un componente de software que provee una interfaz común entre la aplicación y uno o más medios de almacenamiento de datos, tales como una base de datos o un archivo. El término se aplica frecuentemente al patrón de diseño Object. [22]

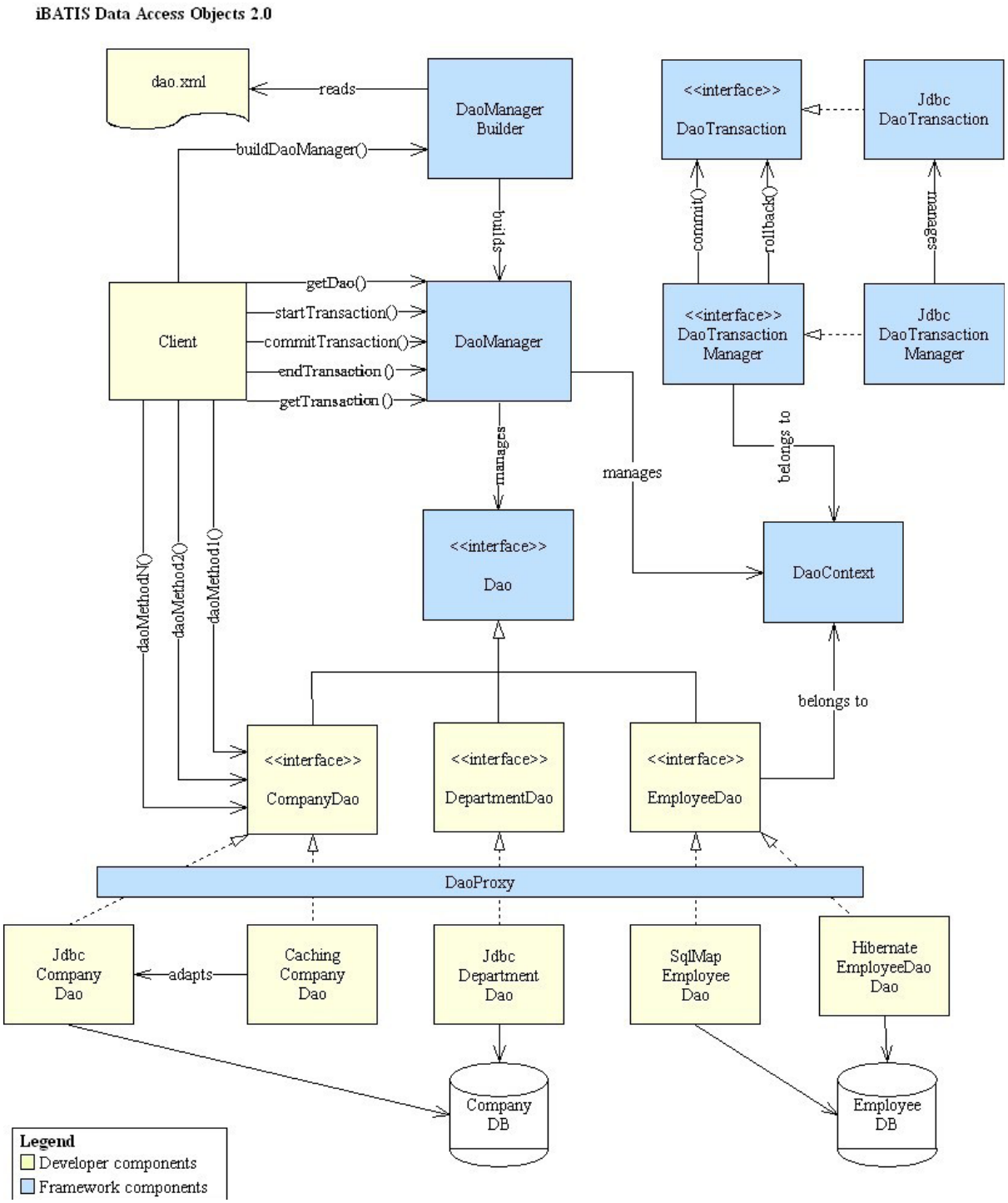
Ibatis data Objects es una capa abstracta que oculta los detalles de la solución de persistencia proveyendo un api común para el resto de la aplicación.

[22] http://es.wikipedia.org/wiki/Data_Access_Object

Cuando desarrollamos aplicaciones robustas en java y .Net, una de las mejores prácticas de programación es, separar el comportamiento común del Api de acceso a datos. Data Acces Object's permite crear componentes simples que proveen acceso a los datos sin revelar la especificación de la implementación del resto de la aplicación. Los DAO's permiten utilizar distintitos mecanismos de persistencia. Si se tiene una aplicación compleja con un numero diferente de bases de datos, los DAO's ayudan a establecer un API consistente para los demás componentes de la aplicación.

Los objetos de acceso a datos son parte de la capa de base de datos de Ibatis, la cual incluye el *framework* SQL Maps. Aunque están conjuntamente empaquetados, el *framework* DAO es completamente independiente y puede ser utilizado sin SQL Maps.

Figura 8 Ibatis Data Acces Object



Fuente: (Clinton, 2006)

Ibatis DAO es válido cuando:

- La capa de acceso a datos puede cambiar de un ambiente de desarrollo a otro.

- Se necesiten realizar cambios considerables en el futuro en la implementación del acceso a datos.

- Se tiene conocimiento que el gestor de base de datos pueda cambiar en el futuro.

3.7. Ibatis SQL Maps:

Ibatis SQL Maps, es un componente el cual facilita la asignación de datos entre el modelo de datos de la base de datos y el modelo de clases de nuestra aplicación. En las siguientes secciones vamos a poder aprender cómo se configura y utiliza esta parte que de las más importantes de Ibatis.

3.7.1. Instalación[23]

La instalación de Ibatis Data Mapper consiste simplemente en colocar los archivos JAR en el classpath, específicamente sobre el classpath de la JVM en tiempo de ejecución (por parámetros), o podría ser sobre el directorio /WEB-INF/lib de la aplicación web.

[23] http://ibatisnet.sourceforge.net/DevGuide.html

IBatis Data Mapper contiene los siguientes archivos:

Tabla I. Archivos JAR utilizados por Ibatis data mapper

Archivo	***Descripción***	***Requerido***
ibatis-common.jar	iBATIS Common Utilities	Si
ibatis-sqlmap.jar	iBATIS Data Mapper Framework	Si
ibatis-dao-1-x-x-b.jar	Legacy Ibatis Data Access Objects for backward compatibility.	No
ibatis-compat		No

Fuente: http://openframework.or.kr/JSPWiki/attach/Hibernate/iBATIS-SqlMaps-2_ko.pdf

3.7.1.1. Archivos Jar y sus dependencias

Cuando un *framework* tiene muchas dependencias, se dificulta la integración con las aplicaciones. Una de las características de Ibatis es que realiza una administración y reducción de dependencias, de esta forma si se está utilizando el JDK 1.4, la única dependencia seria “Jakarta Commons Logging framework”. Los archivos JAR opcionales, están organizados dentro de paquetes que contiene la siguiente estructura /lib/optional dentro del directorio de la distribución, estos están categorizados por funciones. En la siguiente tabla se desglosa cuando utilizar los paquetes opcionales.

Tabla II. Cuando utilizar los paquetes opcionales

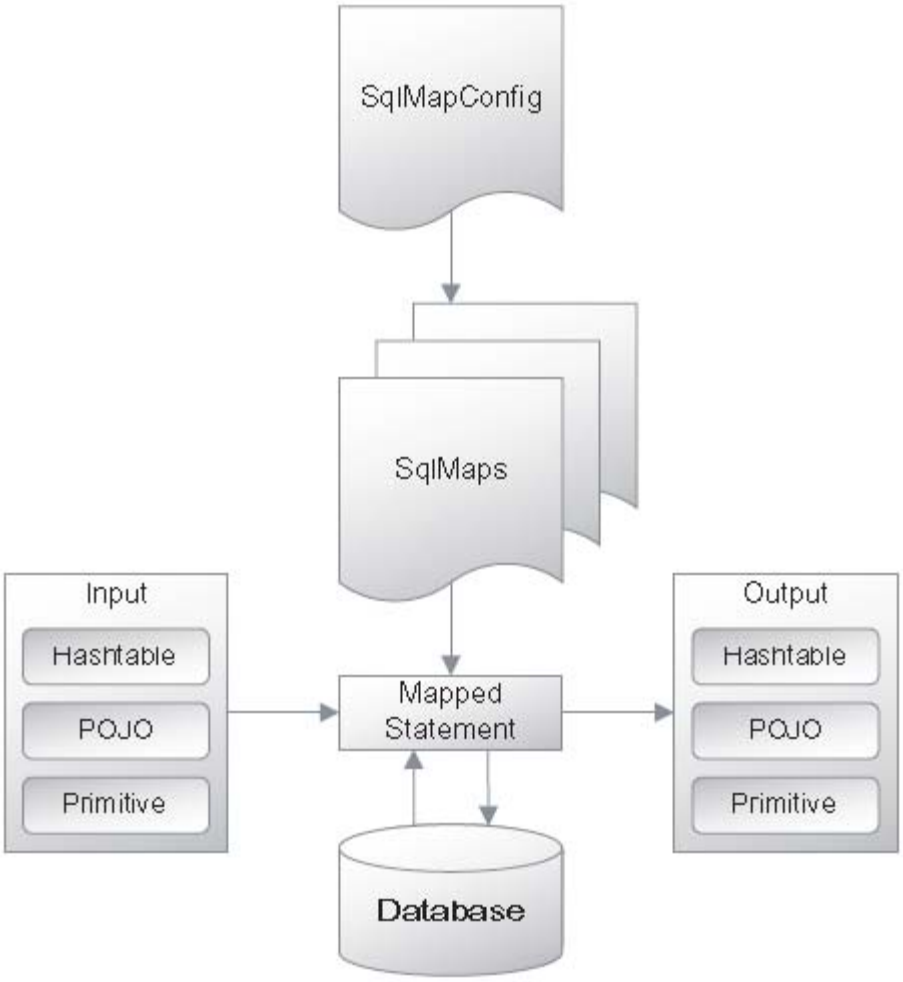
Descripción	***Cuando utilizarlo***	***Directorios***
Soporte JDK	Si se está utilizando una versión más Antigua del JDK 1.4 y el servidor de aplicaciones no provee un jdbc, se pueden utilizar estas opciones.	/lib/optional/xml /lib/optional/jdbc /lib/optional/jta
Soporte DAO	Cuando se utiliza iBATIS (1.x) , para poder utilizar la funcionalidad del framework DAO, se debe incluir estas librerías.	/lib/optional/old_dao
Runtime - Bytecode - Enhancement	Si se desea habilitar CGLIB 2.0 para mejorar la calidad del código generado y el rendimiento de carga perezosa y reflection.	/lib/optional/enhancement
DataSource - Implementation	Si se desea utilizar Jakarta DBCP connection pool.	/lib/optional/dbcp
Distributed - Caching	Si se desea utilizar OSCache para soporte de cache centrada o distribuida.	/lib/optional/caching
Logging Solution	Si se desea utilizar Log4J logging.	/lib/optional/logging

Fuente: http://ibatisnet.sourceforge.net/DevGuide.html

3.8. Configurando Ibatis

La función principal de Ibatis es hacer una relación entre objetos con procedimientos almacenados o sentencias SQL mediante archivos descriptores XML.

Figura 9 Configuración conceptual de Ibatis



Fuente: (Begin, Goodin, & Larry, 2007, pág. 69)

3.8.1. Capa de persistencia

Esta capa está representada por el archivo de configuración principal de SqlMaps, el cual establece la configuración de acceso a la base datos, manejo de las transacciones, manejo de caché, etc. También se incluyen los descriptores XML con las sentencias SQL para cada objeto de modelo.

3.8.2. Archivo de Configuración SQL Map

La configuración de SQL Map se realiza mediante un archivo xml que por lo regular tiene el nombre SqlMapConfig.xml, es el eje central de la configuración de Ibatis (Ver Configuración Conceptual iBatis). Este archivo de configuración le proporciona al framework la información necesaria acerca la conexión y los archivos sqlMap.

Ejemplo 1 Configuración archivo Sql Map

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-config-2.dtd">
<sqlMapConfig>
<properties resource="db.prop" />
<settings
  cacheModelsEnabled="true"
  useStatementNamespaces="false"
  enhancementEnabled="true"
  maxRequests="28"
  lazyLoadingEnabled="false"
  maxSessions="8"
/>
<transactionManager type="JDBC" >
  <dataSource type="SIMPLE">
          <property name="JDBC.Password" value="${pword}"/>
          <property name="JDBC.ConnectionURL" value="${conUrl}"/>
          <property name="JDBC.Driver" value="${driver}"/>
          <property name="JDBC.Username" value="${usuario}"/>
  </dataSource>
</transactionManager>

<sqlMap resource="Tesis/mapeo/SqlRec.xml" />

</sqlMapConfig>
```

Configuraciones Globales

Administración de Transacciones

Referencia a los SqlMap

En el ejemplo anterior se muestran los bloques de los que se compone el archivo de configuración, podemos observar que es en este donde se define parámetros importantes, como lo es el uso de namespaces, si se habilita el modelo de cache, la carga perezosa, la sesiones, etc.

3.8.3. Fichero sqlMap.xml

En este archivo de configuración se colocaran todos los mapeos correspondientes a nuestros objetos (entradas y salidas respectivamente), por ejemplo: insert, update, delete, select, etc.

Ejemplo 2 Configuración sqlMap

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap  namespace="Persona">

<select id="getPersona" resultClass="tesis.ejemplo.Persona">
  SELECT
  PERS_ID as id,
  PERS_HEIGHT_M as heightInMeters,
  PERS_APELLIDO as lastName,
  PERS_PRIMER_NOMBRE as firstName,
  FROM PERSONAS
  WHERE pers_ID = #value#
</select>
</sqlMap>
```

3.8.4. Trabajando con sentencias mapeadas

Luego de tener los archivos de configuración xml, desde la aplicación java, se pueden realizar las llamadas mediante la interfaz sqlMapClient, esta tiene varios métodos, que nos permiten interactuar con la base de datos, obteniendo como resultado objetos que podemos manipular y persistir cuando consideremos conveniente.

3.8.4.1. sqlMap Api

La interfaz SqlMapClient tiene más de 30 métodos en el, por ahora vamos a ver algunas de sus características.

Existen varios tipos de sentencias mapeadas, cada una con un propósito especifico, un conjunto de atributos y atributos hijos. Estas sentencias pudieran parecer obvias pero es importante utilizar la sentencia adecuada para lo que se esté intentando realizar.

3.8.4.1.1. Los métodos queryForObject

Los métodos queryForObjects se utilizan para obtener la información de una sola fila de la base de datos, en un objeto y tiene las siguientes firmas.

- `Object queryForObject(String idMap, Object parameter, Object result) throws SQLException;`
- `Object queryForObject(String idMap, Object parameter) throws SQLException;`

La primera versión es la más utilizada, devuelve un objeto si tiene un constructor por defecto y lanza una excepción en tiempo de ejecución si no tuviera un constructor por defecto.

La segunda opción es muy útil cuando se tiene un objeto que no puede ser fácilmente instanciado por causa de un constructor protegido o si no tiene un constructor por defecto, por lo que Ibatis devuelve el resultado en el objeto enviado como parámetro.

Algo muy importante que hay que recordar al utilizar estos métodos es que si la consulta devuelve más de una fila, este método lanzará una excepción.

3.8.4.1.2. Los métodos queryForList

Los métodos queryForList() son utilizados para obtener una o más filas del repositorio de datos dentro de una lista de objetos java, como queryForObject, este viene en dos versiones:

- `List queryForList(String idMap, Object parameterIn, int skip, int max)throws SQLException;`

- `List queryForList(String idMap, Object parameterIn) throws SQLException;`

El primer método retorna todos los objetos por la sentencia mapeada, la segunda retorna solo un subconjunto de estos, omite hasta el numero indicado por skip y retorna solamente el numero de filas indicado por max, así, si se quieren obtener las primeras 10 filas, se debe pasar por parámetro skip=0 y max=10, para obtener las segundas 10 filas skip=10 y max=10.

3.8.4.1.3. Los Métodos queryForMap

Los métodos queryForMap() retornan un Map (en lugar de una lista) de una o más filas de la base datos en objetos Java. De igual forma que los métodos anteriores, este tiene dos versiones.

- `Map queryForMap(String idMap, Object parameterIn, String key, String valor) throws SQLException;`

- `Map queryForMap(String idMap, Object parameterIn, String key) throws SQLException;`

El primero método ejecutará un query que retornara un Map de Objetos, donde la llave del objeto es identificada por la propiedad nombrada por el parámetro key, y el valor del objeto será completado por la sentencia mapeada. La segunda funciona de forma similar que la primera con la diferencia que el valor será dado por la propiedad del objeto nombrada por el parámetro valor.

3.8.4.2. Tipos de sentencias Mapeadas

Antes de utilizar sqlMap Api, se necesita saber cuáles son las sentencias con las que contamos para trabajar.

Son varios tipos de sentencias mapeadas, cada una con un propósito específico y un conjunto de atributos y elementos hijos. Es importante utilizar la sentencia adecuada para lo que se desea realizar, puesto que cada sentencia cuenta con recursos adicionales propios de cada una. La siguiente tabla contiene una lista de sentencias mapeadas con información adicional.

Tabla III Tipos de sentencias mapeadas y elementos XML relacionado

Tipos de Sentencia	Atributos	Elementos hijos	Uso
<select>	Id resultClass parameterClass parameterMap cacheModel resultMap	Todos los elementos dinámicos	Extracción de datos

<insert>	id parameterMap parameterClass	Todos los elementos dinámicos	Inserción de datos
<update>	Id parameterMap parameterClass	Todos los elementos dinámicos	Modificar Datos
<delete>	id parameterMap parameterClass	Todos los elementos dinámicos	Borrar Datos
<procedure>	id resultClass parameterClass resultMap parameterMap xmlResultName	Todos los elementos dinámicos	Llamar a un procedimiento almacenado
<statement>	id parameterClass resultClass parameterMap resultMap cacheModel xmlResultName	Todos los elementos dinámicos	Esta sentencia se utiliza para realizar cualquier llamada a la base de datos.
<sql>		Todos los elementos dinámicos	Realmente no es una sentencia mapeada, pero puede ser utilizada para crear componentes que pueden ser usados en una sentencia mapeada.
<include>	Refid	none	Realmente no es una sentencia mapeada, pero puede ser utilizada para insertar componentes creados con <sql> dentro de sentencias mapeadas.

Fuente: http://ibatisnet.sourceforge.net/DevGuide.html

3.8.4.3. Carga de la configuración y ejecución de una consulta

Luego de tener configurados todos los archivos de configuración xml, y de importar las dependencias mencionadas, se muestra a continuación un breve ejemplo de cómo cargar la configuración de Ibatis.

Ejemplo 3 Carga de configuración Ibatis

```
import com.ibatis.sqlmap.client.*;
import java.io.Reader;
import java.util.List;

public class Principal
{
  //Método principal
  public static void main(String arg[]) throws Exception
  {
      String recurso = "EjemploSqlConfig.xml";
      Reader reader = Resources.getResourceAsReader(recurso);
      SqlMapClient sqlMap =
      SqlMapClientBuilder.buildSqlMapClient(reader);
      List listado = sqlMap.queryForList("getAllEmp",
"EMPLEADOS");
      System.out.println("Seleccionados " + listado.size() +
"records.");
      for(int i = 0; i < listado.size(); i++)
      {
                  System.out.println(listado.get(i));
      }
  }
}
```

3.9. Modelos de Caché[24]

Algunos valores en la base de datos son conocidos porque cambian con menos frecuencia que otros. Para mejorar el rendimiento, muchos desarrolladores utilizan la caché para almacenar datos utilizados con más frecuencia, para no realizar accesos innecesarios a la base de datos. Ibatis provee su propio sistema de caché, que se configura a través de un elemento <cacheModel>.

[24] http://ibatisnet.sourceforge.net/DevGuide/ar01s03.html#d0e1582

El resultado de una consulta puede ser capturada simplemente especificando en las etiquetas el modelo de cache a utilizar. El modelo de caché se configura dentro del archivo de configuración de mapeo de datos. El modelo de caché está configurado utilizando la siguiente estructura:

Ejemplo 4 Configuración de una caché utilizando elementos de *caché model*.

```
<cacheModel type ="LRU" id="productos-cache" readOnly=”true”>
  <flushInterval hours="20"/>
  <property name=”cache-size” value=”999” />
  <flushOnExecute statement="insertProductos"/>
  <flushOnExecute statement="deleteProductos"/>
  <flushOnExecute statement="updateProductos"/>
</cacheModel>
```

El módulo de caché sobre el cual será creada una instancia llamada “product-cache” utiliza la asignación últimamente utilizada (LRU). Basado en los elementos especificados entre el cacheModel, la caché será volcada cada 20 horas. Estos valores pueden estar dados en minutos, segundos o milisegundos. Además de esto la caché se vaciará cuando se ejecuten las sentencias mapeadas insertProductos, updateProductos. Se pueden especificar cualquier número de sentencias “flush on execute” para la cache. Algunas implementaciones de caché pueden necesitar propiedades adicionales, como por ejemplo ‘cache-size’. En el caso de LRU, size determina el número de entradas para guardar en la caché. Una vez que el modelo de caché es configurado, se puede especificar el modelo de caché a utilizar en una instancia mapeada, por ejemplo:

Ejemplo 5 Especificación de un modelo de caché sobre una sentencia mapeada

```
<statement cacheModel=”usuario-cache” id=”getListadoUsuarios”>
  select * from USUARIOS where ID_USUARIO = #valor#
</statement>
```

3.9.1. Read-Only vs. Read/Write

Ibatis soporta dos tipos de caché read-only y read/write. La caché de tipo Read-only es compartida por todos los usuarios, por lo que representa un mejoramiento en el rendimiento. Sin embargo, un objeto extraído de la caché read-only no puede ser modificado. En cambio, un nuevo objeto puede ser leído de del repositorio de datos o de la caché read/write para ser modificado. Por otro lado, si la intención es utilizar un objeto para su modificación, una cache read/write es recomendable. Para utilizar una cache read/only, establecer sobre el elemento cache model readOnly="true", readOnly="false". La opción por default es read-only (true).

3.9.2. Tipos de Caché

El modelo de caché se maneja como un plugin para soportar diferentes tipos de caches. La implementación en el atributo type del elemento cacheModel (como se vio anteriormente). El nombre de la clase especificada debe ser una implementación de la interface CacheController o uno de los cuatro seudónimos discutidos posteriormente. Los parámetros se pasan mediante la configuración existente en el cuerpo. Actualmente existen 4 implementaciones incluidas en la siguiente distribución, las cuales son las siguientes:

3.9.2.1. "MEMORY"

La implementación "MEMORY" administra el comportamiento de la caché. Es decir, el recolector de basura eficazmente determina que debe estar en memoria o que no. La caché MEMORY es buena opción para aplicaciones que no tienen un patrón identificado para reutilizar objetos, o aplicaciones donde la memoria es escasa.

La implementación MEMORY es configurada de la siguiente Manera:

Ejemplo 6 Configurando tipos de memoria Caché.

```
<cacheModel id="usuario-cache" type="MEMORY">
  <property name=”reference-type” value=”WEAK” />
  <flushOnExecute statement="deleteUsuarios"/>
  <flushInterval hours="20"/>
</cacheModel>
```

La única propiedad reconocida por la implementación Memory, es la propiedad llamada “reference-type”, el valor de esta propiedad, puede ser STRONG, SOFT o WEAK. Estos valores corresponden a los diferentes tipos de referencia de memoria disponibles en la JVM.

La siguiente tabla describe las diferentes “reference-type” que pueden ser utilizadas por una cache MEMORY.

Tabla IV. Tipos de referencias que pueden ser utilizados para la memoria Caché

WEAK (default)	Este tipo de referencia probablemente en la mayoría de casos es la mejor opción y es el valor por defecto si el tipo de referencia no está especificado. Tendrá prioridad para los elementos más populares, pero liberará la memoria para ser utilizada por otros objetos, asumiendo que no están en uso.
SOFT	Este tipo de referencia reducirá la probabilidad de asignar memoria a objetos que no estén actualmente en uso, sin embargo puede darse el caso de asignar memoria a los objetos no tan importantes.
STRONG	Este tipo de referencia garantiza que los resultado estén en memoria hasta que la caché sea volcada explícitamente (Tiempo expirado, o flush). Esto es ideal cuando la información es: 1) muy pequeña, 2) absolutamente estática y 3) Se utiliza muy a menudo. La principal ventaja es el buen rendimiento. La desventaja es que no se liberará memoria de los objetos que pudiera ser necesitada por otros objetos (posiblemente más importantes).

3.9.2.2. "LRU"

La implementación LRU utiliza el algoritmo último recientemente utilizado para determinar cómo los objetos serán removidos automáticamente de la caché. Cuando la caché se llena, el objeto que se a accedido menos recientemente será el que se eliminará de la caché. Cuando un objeto en particular se utiliza frecuentemente, la probabilidad que sea desechado de la caché es muy pequeña. La caché LRU es una buena opción para usuario que utilizar objetos más frecuentemente por un periodo largo de tiempo, como por ejemplo: Navegando hacia atrás y hacia adelante, listas paginadas.

La implementación LRU tiene la siguiente configuración:

Ejemplo 7 Configurando un tipo de caché LRU

```
<cacheModel type="LRU" id="usuario-cache">
  <flushOnExecute statement="insertUsuario"/>
  <flushOnExecute statement="deleteUsuario"/>
  <flushOnExecute statement="updateUsuario"/>
  <flushInterval hours="20"/>
  <property name=”size” value=”900” />
</cacheModel>
```

Solo una sola propiedad es reconocida por la implementación LRU. Esta propiedad llamada ‘size’ contiene un valor entero que representa el número máximo de objetos que pueden estar en la cache a la vez. Algo importante que debemos recordar es que un objeto puede ser desde una instancia de un ArrayList hasta una cadena de caracteres. Por lo tanto, se debe tener el cuidado de no guardar muchos objetos en caché pues se corre el riesgo de quedarse sin memoria.

3.9.2.3. “FIFO”

La implementación de caché FIFO utiliza el algoritmo primero en entrar, primero en salir para determinar que objeto será removido de la cache. Cuando la caché empieza a llenarse el objeto más antiguo será removido de esta. La caché FIFO es buena para patrones donde se utilizan consultas frecuentemente y luego se utilizan hasta un momento posterior.

La implementación FIFO es configurada de la siguiente manera:

Ejemplo 8 Configurando una cache tipo FIFO

```
<cacheModel type="FIFO" id="usuario-cache">
  <property name=”size” value=”900” />
  <flushOnExecute statement="deleteUsuario"/>
  <flushOnExecute statement="insertUsuario"/>
  <flushOnExecute statement="updateUsuario"/>
  <flushInterval hours="19"/>
</cacheModel>
```

Solo una sola propiedad es reconocida por la implementación LRU. Esta propiedad llamada 'size', la cual cumple exactamente con la misma función que en la implementación LRU.

3.10. Transacciones

Por defecto la llamada a un método executeXxxx() sobre una instancia SqlMapClient tendrá auto-commit true. Lo que significa que cada llamada será una única unidad de trabajo. Esta idea es algo simple, pero no es ideal si se desea utilizar una serie de declaraciones que se deben ejecutar como una sola unidad de trabajo (es decir se ejecuta todo o nada) es allí donde las transacciones entran en juego.

Si se está utilizando una transacción global (configurada el archivo de configuración del Data Mapper), se puede utilizar el autocommit encerrando toda la unidad de trabajo que se desee. Sin embargo, es necesario delimitar la transacción por motivos de rendimiento y reducir el tráfico en el pool de conexiones.

La interfaz SqlMapClient tiene métodos que permiten demarcar los límites transaccionales. Una transacción puede iniciarse, ejecutar un commit y/o realizar un rollback usando los siguientes métodos sobre la interfaz SqlMapClient:

```
public void startTransaction () throws SQLException;
public void endTransaction () throws SQLException;
public void commitTransaction () throws SQLException;
```

Para iniciar una transacción se necesita una conexión del pool de conexiones, y abrirla para recibir Querys y modificaciones.

Un ejemplo utilizando transacciones sería el siguiente:

Ejemplo 9 Usando transacciones en Ibatis.

```
//Se obtiene el archive de entrada
private Reader readerIn = new Resources.getResourceAsReader
("tesis/ejemplo/sqlMapconfig.xml");
  //Se crea una instancia de sqlMap con el archive de entrada.
private SqlMapClient sqlMap =
XmlSqlMapBuilder.buildSqlMap(readerIn);

public updateItemDescripcion (String Id, String
Descripcion)throws SQLException {
  try {
    sqlMap.startTransaction ();
    ItemI itemAux = (ItemI)sqlMap.queryForObject ("getItem", Id);
    itemAux.setDescription (Descripcion);
    sqlMap.update ("updateItemUsuario", itemAux);
    sqlMap.commitTransaction ();
          //Se finaliza la transacción.
  } finally
  {
    //Se finaliza la transacción.
    sqlMap.endTransaction ();
  }
}
```

3.10.1. Transacciones automáticas

Aunque las transacciones explicitas son muy recomendadas, existe una simplicidad semántica que puede ser usada para simples requerimientos (generalmente read-only). Si no se especifica explícitamente el inicio y el fin de la transacción mediante los métodos startTransaction(), commitTransaction() y endTransaction(), estos serán llamados automáticamente cuando se ejecute la sentencia, como se especifica en el siguiente ejemplo:

Ejemplo 10 Transacciones automáticas en Ibatis

```
private Reader readerIn = new Resources.getResourceAsReader
("gt/tesis/ejmplo/SampeleSqlMap.xml");

//Se crea una instancia de sqlMap con el archive de entrada.
private SqlMapClient sqlMap =
XmlSqlMapBuilder.buildSqlMap(readerIn);
```

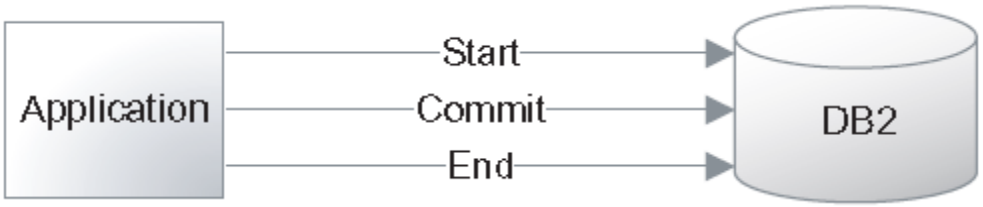
```
//Se crea el método para realizar un update.
public updateItemDescription (String Id, String nDescrip) throws
SQLException {
//Se captura la excepción.
  try {
    ItemI itemAux =(ItemI)sqlMap.queryForObject ("selectItem",
Id);
    itemAux.setDescripcion (“Trans1”);
  // La Transacción se hará de forma automática
    sqlMap.update("updateItemUsuario", itemAux);
    itemAux.setDescripcion (nDescrip);
    itemAux.setDescripcion (“Trans2”);
  // La Transacción se hará de forma automática
    sqlMap.update("updateItemUsuario    ", itemAux);
  }
  catch (SQLException z) {
    throw (SQLException) z.printStackTrace();  }}
```

Se debe tener cuidado al utilizar las transacciones automáticas, pues por muy atractivas que parezcan, pueden poseer problemas cuando se requieran unidades de trabajo que consistan en mas que un simple update a la base datos. En el ejemplo anterior, si la segunda llamada “updateItem” falla, la descripción del término no será modificada y permanecerá con la primera descripción (este no es un comportamiento transaccional).

3.10.2. Transacciones locales

Las transacciones locales son el tipo de transacciones más comunes, y es realmente lo mínimo que se puede utilizar en una base de datos de tipo relacional. Una transaccion local es la forma mínima en la cual se puede realizar una transaccion en una base de datos relacional. Las transacciones locales estan en una misma aplicación y de un solo recurso (una base de datos relacional), esto se explica en la figura siguiente:

Figura 10 Alcance de las transacciones locales



Referencia: (Begin, Goodin, & Larry, 2007, pág. 152)

Las transacciones locales son configuradas en el archivo de configuración xml de IBATIS SQL Map, mediante una administración de de transacciones jdbc, como se muestra en el siguiente Ejemplo.

Ejemplo 11 Configuración de transacciones locales Ibatis

```
<transactionManager type="JDBC">
  <dataSource type="SIMPLE">
  <property …/>
  <property …/>
  <property …/>
  </dataSource>
</transactionManager>
```

El atributo type="JDBC", define que iBatis debe utilizar el API estándar de conexión JDBC para administrar las transacciones. Utilizando el API SqlMapClient es bastante sencillo demarcar las transacciones con startTransaction, commitTransaction y endTransaction.

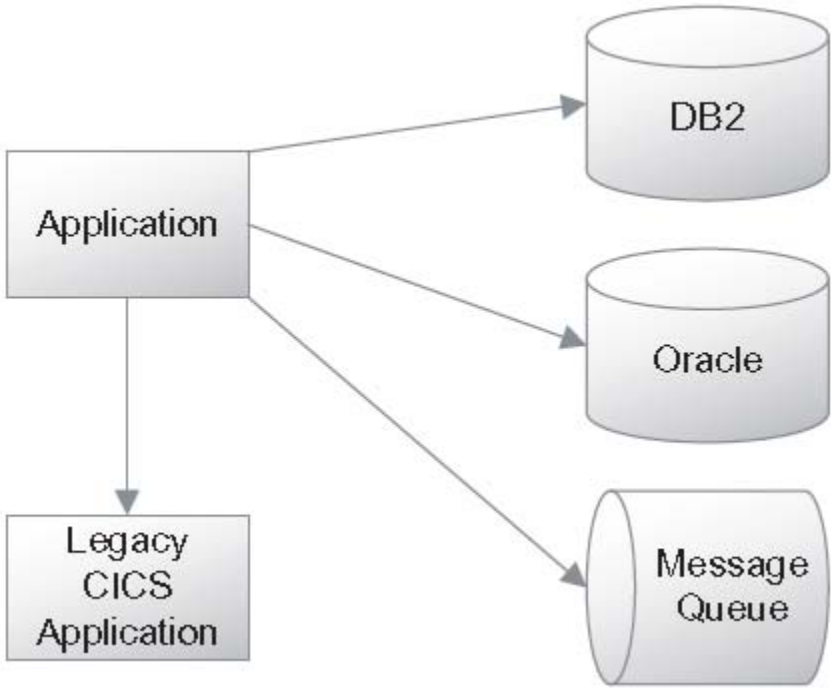
3.10.3. Transacciones Globales (Distribuidas)

El *framework* Ibatis Data Mapper, tiene soporte para transacciones globales. También conocidas como transacciones distribuidas. Permiten realizar modificaciones en múltiples bases de datos (u otros recursos compatibles con JTA) en una misma unidad de trabajo.

Se tiene la opción de administrar las transacciones globales externamente. Ya sea mediante programación (Código escrito a mano) o la aplicación de otro *framework*, como por ejemplo EJB. Utilizando EJB's se pude delimitar la transacción mediante un descriptor de despliegue.

Cuando se utilizan transacciones globales, los métodos de SQL Map para el control de transacciones son redundantes, porque las sentencias begin, commit y rollback son controladas por el administrador externo de transacciones.

Figura 11 Alcance de las transacciones globales Ibatis



Fuente: (Begin, Goodin, & Larry, 2007, pág. 154)

3.10.3.1. Administración de transacciones globales

El *framework* Ibatis data Mapper también puede administrar las transacciones globales. Para habilitar el soporte de administración global de transacciones, se debe establecer el atributo <transactionManager> de tipo "JTA" en el archive de configuración de SQL Map y establecer la propiedad "UserTransaction" con el JDNI donde el cliente SqlMap buscará la instancia de la transacción.

Las transacciones globales programadas no son muy diferentes, simplemente se deben tener algunas consideraciones, por ejemplo:

Ejemplo 12 Transacciones globales con Ibatis

```
try {
  //Se inicia la primera transacción.
  ordenSqlMap.startTransaction();
  //Iniciamos la segunda transacción.
  tiendaSqlMap.startTransaction();
  ordenSqlMap.insertOrden(…);
  tiendaSqlMap.updateCantidad(…);
  //Terminamos la primera transacción.
  tiendaSqlMap.commitTransaction();
  //Terminamos la segunda transacción.
  ordenSqlMap.commitTransaction();
  }
  finally {
    try {
      tiendaSqlMap.endTransaction()
    }
    finally {
      ordenSqlMap.endTransaction()
    }
}
```

En el ejemplo anterior, tenemos dos instancias SqlMapClient, las cuales utilizan bases de datos distintas. La primera instancia SqlMapClient (ordenSqlMap) utilizamos para iniciar una transacción también iniciamos una transacción global. Luego de esto, todas las demás actividades son

consideradas parte de la transacción global hasta que la misma instancia ejecute commitTransaction() y endTransaction(), donde todo el trabajo dentro de la transacción es considerada realizada.

Aunque pareciera sencillo, es muy importante que no se abuse de las transacciones globales (distribuidas). Existen implicaciones en el rendimiento, así como configuraciones complejas para el servidor de aplicaciones y de los drivers de la base de datos.

3.11. ¿Cuándo utilizar Ibatis?[25]

Como toda herramienta ORM, Ibatis no siempre es recomendable, a continuación se listan una serie de características donde es recomendable utilizar Ibatis:

- Cuando se necesite cambiar el SQL para optimizarlo, utilizar SQL propietario, etc. o sea necesario ejecutar procedimientos almacenados.

- Se requiere un rápido aprendizaje de la herramienta no teniendo previos conocimientos, pues no requiere aprender un nuevo lenguaje de consultas como en EJB, CMP o Hibernate.

- Cuando se requiere un alto rendimiento y optimización.

Por el contrario no es recomendable Ibatis cuando:

- Se requiere que la aplicación soporte multi-RDBMS de forma transparente y automático.

[25] http://www.adictosaltrabajo.com/tutoriales/tutoriales.php

- Se requiere una automatización total y transparente de la persistencia.

3.12. Licencia

Ibatis es un producto Open Source modificable, disponible y comercializable libremente, es un proyecto de Apache Software Foundation.

3.13. Quienes utilizan Ibatis:

A continuación se listan algunos sitios públicos que utilizan iBatis.

Tabla V Listado de empresas que utilizan Ibatis.

Nombre	URL
GalMarley.com	http://www.galmarley.com
Fiskars - Consumer products manufacturer	http://www.fiskars.com
BullionVault.com - Online gold trading	http://www.bullionvault.com
1up.com - Gaming community	http://www.1up.com
GAPay	http://www.gapay.com.
The Ohio State University School of Communication	http://www.comm.ohio-state.edu
Ideal Financial Services	http://www.idealfsi.com
Telecom Italia's OpenID Provider	http://openid.alice.it (Italian)
Live Music Addicts	http://www.livemusicaddicts.com/newyork/
OfficeMax Impress	http://www.officemax.com
Wisconsin Department of Natural Resources	http://botw.dnr.state.wi.us/botw/Welcome.do
Newport	http://www.newpoint.com
Wilkinson Sword Garden Collection	http://www.wilkinsonswordgarden.com
TheLadders.com	http://www.theladders.com
Abebooks.com - Worlds largest online marketplace for books	http://www.abebooks.com

MySpace.com - A place for friends – see Feedback and Experiences	http://www.myspace.com
PowerSentry	http://www.powersentry.com
Watkins Printing Company	http://www.watkinsprinting.com/
WomanOwned - Business Networks for Women	http://www.womanowned.com
Workeffort - time tracking system	https://workeffort.dev.java.net/
Plateau Systems - Learning Management and Performance Management Systems	http://www.plateausystems.com
Zinck-Lysbro - Gardening products	http://www.zincklysbro.dk
FNM Group - Ferrovie Nord Milano (Railways - Italy)	http://www.fnmgroup.it (Italian)
Cool Advance - All software projects use IBatis as its persistence layer (both .Net and Java). Example: Shoesathome.com	http://www.cooladvance.com (Portuguese)

Fuente: http://opensource.atlassian.com/confluence/oss/display/IBATIS/Home

3.14. Soporte

El soporte de Ibatis se provee mediante una lista de correo al usuario. Se puede suscribir enviando una nota a user-java-subscribe@ibatis.apache.org, para darse de baja en la suscripción, se debe enviar un correo a: user-java-unsubscribe@ibatis.apache.org.

Una vez suscrito se pueden enviar preguntas o informes de error a: user-java@ibatis.apache.org.

3.15. Herramientas de mapeo.

3.15.1. Abator

Abator es un generador de código para Ibatis. Abator realiza ingeniería inversa sobre una o muchas tablas de la base de datos, genera artefactos Ibatis que pueden ser utilizados para el acceso a las tablas. Esto disminuye la molestia configuración inicial de los archivos xml. Abator busca realizar una automatización sobre las operaciones que se realizan con mayor periodicidad en las bases de datos, como las operaciones CRUD (Create, Retrieve, Update, Delete).

Abator genera los siguientes artefactos:

- Java POJO's que representa la estructura de la tabla. Estos pueden incluir:
    - Una clase que concuerda con la llave primaria de la tabla.
    - Una clase que concuerda con todos los campos que no son llave primaria de la tabla (Exceptuando los campos BLOB).
    - Una clase que incluye los campos BLOB de la tabla (si la tabla tuviera campos BLOB).
    - Una clase que implementa consultas, modificaciones y eliminaciones dinámicas.

- iBATIS Compatible con SQL Map XML Files. Abator genera SQL para las funciones CRUD sobre la configuración de cada tabla en la configuración. Las sentencias generadas incluyen:
    - Insert
    - Update by primary key
    - Update by example(utilizando una clausula “where” dinámica)
    - Delete by primary key
    - Delete by example (utilizando una clausula where dinámica)
    - Select by primary Key
    - Select by example ((utilizando una clausula where dinámica)

Estas sentencias dependen de la configuración de la tabla que se está mapeando, por ejemplo si la tabla no tuviera una llave primaria, Abator no generaría una función update by primary key.

- Interfaces Dao e implementación de clases apropiadas para uso sobre los objetos. La generación de clases DAO es opcional. Abator puede generar DAO’s de los siguientes tipos:
    - DAOs que son parte del *framework* Spring.
    - DAOs que solo pueden utilizar el API de Ibatis SQL mapping. Estos DAOs pueden ser generados regulados mediante el constructor o por medio de inyecciones de spring.

- DAOs que conforman el *framework* DAO de iBATIS (Ahora ya no se utiliza fue remplazado por el framework de spring).

Abator está diseñado para correr en un ambiente de desarrollo iterativo, también puede incluir tareas Ant. Las variables que se deben tomar en cuenta cuando se tiene a Abator corriendo en un ambiente iterativo.

1) Abator realizará una unión (merge) entre los archivos XML si estos existiesen, de otro modo crearía los nuevos archivos. Abator nunca sobrescribirá los cambios realizados a los archivos XML, se puede correr Abator sin miedo a que esto suceda.

2) Abator no unirá o fusionará los archivos java. Puede grabarlos con otro nombre y si se realizan cambios se deben fusionar a manualmente. Solamente que se utilice como un plugin de Eclipse, Abator si puede realizar la fusión de estos archivos.

Ejemplo 13 Configurar Abator mediante una conexión JDNI Oracle.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE abatorConfiguration
  PUBLIC "-//Apache Software Foundation//DTD Abator for iBATIS
Configuration 1.0//EN"
  "http://ibatis.apache.org/dtd/abator-config_1_0.dtd">

<abatorConfiguration>
  <abatorContext id="OracleTables" generatorSet="Java2">
    <jdbcConnection ◄———— 1
driverClass="oracle.jdbc.driver.OracleDriver"

connectionURL="jdbc:oracle:thin:@10.100.62.107:1521:DESA"
        userId="TIGER"
        password="Tiger2007">
               <classPathEntry
location="C:\desarrolloNuevo\Program\Program\iBator\ojdbc14.jar
" />
    </jdbcConnection>
```

```
    <javaTypeResolver > ←————2
      <property name="forceBigDecimals" value="false" />
    </javaTypeResolver>

    <javaModelGenerator ←————3
targetPackage="gt.gob.sat.jfiscalizacion.jprogram.dto"
targetProject="..\Model\src">
      <property name="trimStrings" value="true" />
    </javaModelGenerator>

    <sqlMapGenerator ←———— 4
targetPackage="gt.gob.sat.jfiscalizacion.jprogram.dao.xml"
targetProject="..\Model\src">
    </sqlMapGenerator>

    <daoGenerator type="IBATIS" ←————5
targetPackage="gt.gob.sat.jfiscalizacion.jprogram.dao.obj"
targetProject="..\Model\src">
    </daoGenerator>

        <table tableName="Nombre_tabla" schema="nombre_esquema">
        </table> ←———— 6

  </abatorContext>
</abatorConfiguration>
```

El ejemplo anterior muestra una configuración básica de Abator, que está constituida de los siguientes elementos:

1. jdbcConnection: Esta sección permite indicarle a Abator la forma en la que nos vamos a conectar a la base de datos, mediante esta conexión Abator podrá extraer los DLL de las tablas y generar los archivos de configuración necesarios para el manejo de la persistencia con Ibatis. classPathEntry indica en donde tenemos el archivo jar que contiene el driver de conexión.

2. javaTypeResolver: Con Abator podemos definir los mapeos de los tipos de las columnas de las tablas hacia los atributos de nuestros objetos, por ejemplo mapear todos los varchar2 a String, Number

a Integer, etc. forceBigDecimals indica si todos los tipos numéricos serán mapeados a BigDecimals.

3. javaModelGenerator: En esta sección definimos como Abator nos generará los pojos correspondientes a las tablas mapeadas, indicándole el paquete en java al que pertenecerán y la ubicación de los archivos.

4. sqlMapGenerator: Mediante esta sección le decimos a Abator como generar los archivos XML compatibles con SQL Maps, estos archivos contienen SQL para las funciones CRUD sobre la configuración de cada tabla en la configuración.

5. daoGenerator: Con esta opción le indicamos a Abator que nos genere las interfaces Dao con sus respectivas implementaciones, indicándole que tecnología vamos a utilizar para nuestra persistencia, en este caso es Ibatis.

6. Table: En esta sección colocamos todas las tablas que queremos mapear.

Por último, luego de tener nuestro archivo de configuración procedemos a ejecutar el jar desde consola.

Ejemplo 14 Correr Abator desde Consola

```
java -jar abator.jar -configfile Program.xml
```

Abator tiene más configuraciones, estas están disponibles en la guía de referencia http://ibatis.apache.org/docs/tools/abator/.

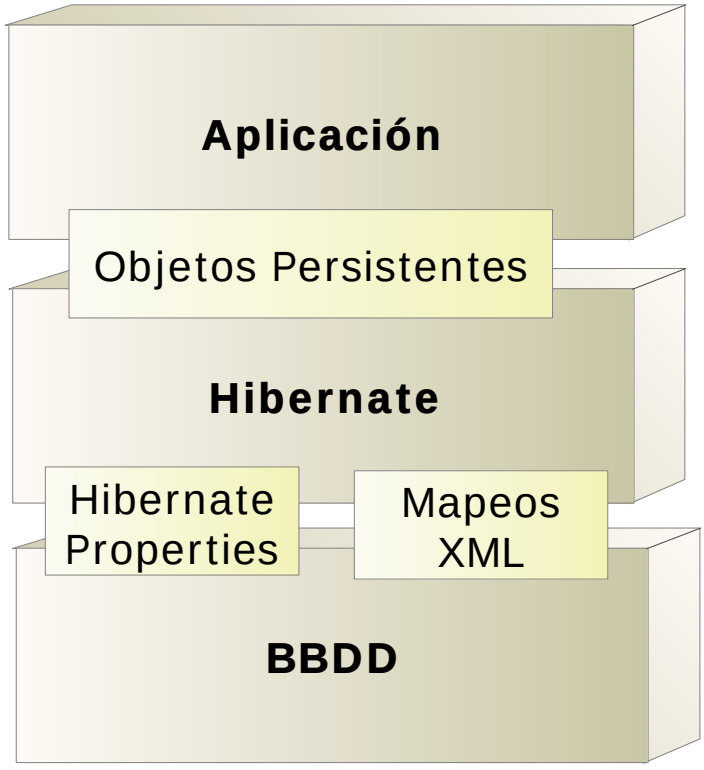
3.15.1.1. Dependencias

Abator no tiene dependencias más que el JRE. Abator requiere JRE 1.4 o mayor. Abator también requiere un driver JDBC que implemente especialmente los métodos getColumns and getPrimaryKeys.

4. HIBERNATE

Es un *framework* de mapeo Objeto Relacional open source. Una de las principales características de Hibernate es que permite tener persistencia con los datos transforman modelos de diseño de datos a formato XML de acuerdo a DTD definidos por Hibernate.[26]

Figura 12. Diagrama de Bloques Hibernate



Fuente: (Chambi Aruquipa & Marquez Granado, 2005)

[26] (Chambi Aruquipa & Marquez Granado, 2005)

Hibernate es un *framework* que se ubica en la capa de persistencia objeto/relacional y es un generador de sentencias SQL, permite manejar objetos persistentes, que podrán incluir características tales como: polimorfismo, colecciones, relaciones, y múltiples tipos de datos. De manera muy optimizada y rápida se pueden generar mapeos Objeto Relacionales en cualquiera de los entornos soportados (tiene soporte para todos los sistemas gestores de bases de datos y se integra de forma elegante y sin restricción alguna con los más conocidos contenedores web y servidores de aplicaciones J2EE, también puede utilizarse en aplicaciones standalone). Hibernate es la solución ORM más conocida en las aplicaciones Java. Permite implementar clases persistentes a partir de pojos no importando si incluyen composición, colecciones de objetos, polimorfismo, herencia y asociación. Hibernate provee una forma muy elegante para accesar a la información de las bases de datos relacionales, esto es gracias a su lenguaje de consultas HQL (Hibernate Query Language), el cual tiene un diseño como una mínima extensión orientada a objetos de SQL. Hibernate también permite realizar consultas basadas en criterios o en SQL puro.

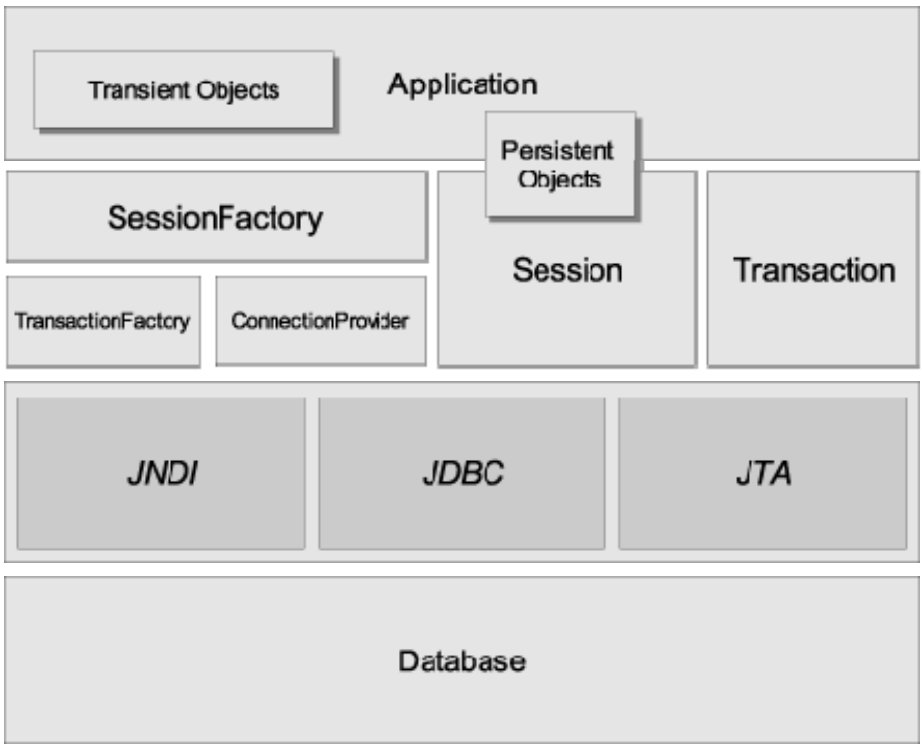
4.1. Arquitectura[27]

El API de Hibernate es una arquitectura de dos capas (Capa de persistencia y Capa de Negocio).

En la siguiente figura se muestran los roles de las interfaces Hibernate más importantes en las capas de persistencia y de negocio de una aplicación J2EE. La capa de negocio está situada sobre la capa de persistencia, debido a que actúa como un cliente de la capa de persistencia.

[27] (Universidad Femenina del Sagrado)

Figura 13 Arquitectura Hibernate



Fuente:http://www.hibernate.org/hib_docs/reference/en/html/architecture.html

Las Interfaces mostradas se clasifican de la siguiente forma:
Interfaces llamadas por la aplicación para realizar operaciones básicas:

- *Session*: interfaz primaria utilizada en cualquier aplicación Hibernate (*SessionFactory*).

- *Transaction*: Esta interfaz se utiliza para el manejo de transacciones, las cuales se listaran más adelante.

- *Query*: permite realizar peticiones a la base de datos y controla cómo se ejecuta dicha petición (query). Las peticiones se escriben en HQL o en

SQL nativo de la base de datos que estamos utilizando. Una instancia Query se utiliza para enlazar los parámetros de la petición, limitar el número de resultados devueltos por la petición y para ejecutar dicha petición.

- Interfaces llamadas por el código de la infraestructura de la aplicación para configurar Hibernate. La más importante es la clase “Configuration”**,** esta clase se utiliza para configurar y "arrancar" Hibernate. La aplicación utiliza una instancia de configuration para especificar la ubicación de los documentos que indican el mapeado de los objetos y propiedades específicas de Hibernate, y a continuación crea la Session Factory.

- Interfaces callback que permiten a la aplicación reaccionar ante determinados eventos que ocurren dentro de la aplicación, tales como Interceptor, Lifecycle, y Validatable.

- Interfaces que permiten extender las funcionalidades de mapeado de Hibernate, como por ejemplo UserType, CompositeUserType, e IdentifierGenerator.

Además, Hibernate hace uso de API’s de Java, tales como JDBC, JNDI (Java Naming Directory Interface) y JTA (Java Transaction Api).

4.2. Características clave

A continuación, se listan una serie de características claves propias de Hibernate.

- Modelo programación orientada a objetos: Hibernate tiene soporte para todas las características del paradigma orientado a objetos, pudiendo

utilizar sin ningún problema, características como: herencia, polimorfismo, composición y el *framework* de colecciones de Java.

- Escalabilidad: Hibernate posee una gran escalabilidad, posee una caché de dos niveles que puede ser usado en un clúster, además de esto también permite inicialización perezosa de objetos y colecciones.

- Persistencia transparente: Una gran ventaja Hibernate es que la persistencia se realiza de forma transparente para el desarrollador, el desarrollador nunca tiene que construir una sentencia SQL.

- Lenguaje de consultas HQL: Hibernate proporciona su propio lenguaje de consultas, el uso de este lenguaje crea independencia del gestor de base de datos que se esté utilizando. Este lenguaje puede ser utilizado tanto para el almacenamiento de objetos como para la extracción de los mismos.

- Manejo automático de llaves primarias: Hibernate permite asignaciones de identificadores automáticos por la aplicación, así como también tiene soporte de los distintos tipos de generación de llaves que proporciona los manejadores de bases de datos, como ejemplo: secuencias, columnas autoincreméntales, entre otras.

- Mejoramiento del código compilado (bytecode): No se necesita realizar un procesamiento del bytecode ni generar código en tiempo de compilación.

- Soporte para múltiples modelos de objetos: Permite múltiples tipos de mapeos entre objetos dependientes y colecciones.

- Soporte para transacciones: Hibernate tiene soporte para transacciones largas y gestiona la política optimistic locking de forma automática.

4.3. Simplicidad y flexibilidad[28]

En lugar de un conjunto de clases y configuraciones requeridas por algunas soluciones de persistencia tal como EJB, Hibernate requiere un único archivo de configuración en tiempo de ejecución y un documento de especificación para cada objeto persistente. La configuración en tiempo de ejecución puede ser seteada por valor o por medio de un archivo XML, alternativamente se puede configurar parte de Hibernate programáticamente en tiempo de ejecución.

El formato XML de los documentos de mapeo puede ser muy corto, dejando al propio *framework* determinar las demás características. Adicionalmente, usted puede proveer al *framework* con más información para especificar propiedades adicionales.

Ejemplo 15 Descriptor xml de una tabla.

```
<?xml version="1.0"?>
<hibernate-mapping package="com.manning.hq.ch01">
    <class name="Event" table="events">
        <id name="id" column="uid" type="long">
            <generator class="native"/>
        </id>
        <property name="name" type="string"/>
        <property name="startDate" column="start_date"
                  type="date"/>
        <property name="duration" type="integer"/>
        <many-to-one name="location" column="location_id"/>
        <set name="speakers">
            <key column="event_id"/>
            <one-to-many class="Speaker"/>
        </set>
        <set name="attendees">
            <key column="event_id"/>
            <one-to-many class="Attendee"/>
        </set>
    </class>
</hibernate-mapping>
```

[28] (Peak & Heudecker, 2006)

Al utilizar algunos *framework* de persistencia como EJB, la aplicación se torna dependiente del *framework*. Hibernate no crea esta dependencia adicional. Los objetos persistentes en la aplicación no contienen información extra de Hibernate en su semántica. Simplemente se crean objetos de java asociados a los documentos de especificación.

A diferencia de EJB's, Hibernate no requiere de un especial contenedor para funcionar, puede ser utilizado en cualquier entorno desde un aplicación standalone hasta un servidor de aplicaciones empresarial.

4.4. Rendimiento[29]

Una idea errónea acerca de los ORM, es que existe un gran impacto en el rendimiento de las aplicaciones, este no es el caso de Hibernate. La forma clave de medir el rendimiento de un ORM, es si utiliza el número mínimo de acceso a la base datos. Esto es válido para inserción, modificación y extracción de datos. Muchos *framework* basados en jdbc realizan la modificación en la base de datos de los objetos aun si estos no han cambiado, Hibernate asegura una actualización si solo si el estado del objeto a cambiado.

La carga perezosa provee otra forma de mejorar el rendimiento, puesto que para un objeto ya cargado en memoria, se poblan las colecciones (hijos) hasta que estén a punto de accederse. Esto asegura que se tendrán cargados solo los objetos necesarios lo que tiene un gran impacto en el rendimiento.

Otra forma de mejorar el rendimiento es la capacidad de deshabilitar selectivamente objetos asociados que se recuperan cuando el objeto principal es obtenido. Esto se logra mediante el establecimiento de las regiones a la cual

[29] (Peak & Heudecker, 2006)

está asociada la propiedad de asociación. Por ejemplo, imaginemos que se tiene una clase *Usuario* con una asociación one-to-one con una clase *Departamento.* Cuando se recupera la clase usuario no siempre se querrá recuperar el Departamento asociado, por lo que se puede declarar que el objeto Departamento sea recuperado solo cuando este sea necesario mediante la configuración outer-join false.

También es posible declarar un proxy de objetos, dando como resultado objetos poblados solo cuando el desarrollador los solicita. Cuando se crea un proxy, Hibernate crea una representación no poblada de clases persistentes. El proxy se reemplazará por la instancia actual de la clase persistente llamando a un método del objeto de acceso.

El proxy está relacionado con la configuración outer-join que se acaba de mencionar. Utilizando el mismo ejemplo: si la clase Departamento es declarada para tener un proxy, no se necesita establecer outer-join false sobre la clase Usuario para el departamento (El servicio de Hibernate solo poblará los Departamentos cuando sean necesarios).

La caché de los objetos también juega un rol importante en el mejoramiento del rendimiento de la aplicación. Hibernate soporta varios productos open source y comerciales para manejo de la caché. La caché puede ser habilitada para clases persistentes o para colecciones de objetos persistentes. Los resultados de las consultas también pueden utilizar caché pero estos solo se benefician las consultas con los mismos parámetros. La caché de consultas no garantiza un aumento en el rendimiento en las aplicaciones pero está disponible para ser utilizada en casos especiales.

4.5. Interfaces básicas

- **Session**: Interfaz primaria, se crean y destruyen muchas instancias en la ejecución, pero sus instancias son objetos ligeros. Una sesión está entre una conexión y una transacción. También hace de cache de objetos cargados.

- **SessionFactory**: Devuelve instancias de sesiones, se necesita una instancia diferente por cada tipo de BD accedida.

- **Configuration**: para indicar la localización de los ficheros de mapeo.

- **Transaction**: (opcional) Abstracción de una transacción concreta (JDBC, JTA, CORBA).

- **Query**: para realizar consultas, y controlar su ejecución, en HQL o SQL.

- **Criteria**: Para crear y ejecutar OO consultas (parecido a Query).

- **Otras**:
    - Interfaces CallBack: para recibir notificaciones de eventos sobre objetos (cargado, guardado, borrado). Ej.: Lifecycle, Validatbale, Interceptor.

    - Type: mapea el tipo de un objeto a el tipo de una tabla(s). Se pueden añadir tipos de usuario con las interfaces UserType y CompositeUserType.

4.6. Configuración básica Hibernate[30]

Para utilizar Hibernate en una aplicación, es necesario conocer cómo configurarlo. Hibernate puede configurarse y ejecutarse en la mayoría de aplicaciones Java y entornos de desarrollo. Generalmente, Hibernate se utiliza en aplicaciones cliente/servidor de dos y tres capas, desplegándose únicamente en el servidor. Las aplicaciones cliente normalmente utilizan un navegador web, pero las aplicaciones swing y AWT también son usuales. Aunque solamente vamos a ver cómo configurar Hibernate en un entorno no gestionado, es importante comprender la diferencia entre la configuración de Hibernate para entornos gestionados y no gestionados:

- Entorno gestionado: los pools de recursos tales como conexiones a la base de datos permiten establecer los límites de las transacciones y la seguridad se debe especificar de forma declarativa, es decir, en sus metadatos. Un servidor de aplicaciones J2EE, tal como JBoss, Bea WebLogic o IBM WebSphere implementan un entorno gestionado para Java.

- Entorno no gestionado: proporciona una gestión básica de la concurrencia a través de un pooling de threads. Un contenedor de servlets, como Tomcat proporciona un entorno de servidor no gestionado para aplicaciones web Java. Una aplicación stand-alone también se considera como no gestionada. Los entornos no gestionados no proporcionan infraestructura para transacciones automáticas, gestión de recursos, o seguridad. La propia aplicación es la que gestiona las conexiones con la base de datos y establece los límites de las transacciones.

[30] (CCIA, 2005-2006)

Tanto en un entorno gestionado como en uno no gestionado, lo primero que debemos hacer es iniciar Hibernate. Para hacer esto debemos crear una SessionFactory desde la clase Configuration.

4.6.1. Especificación de opciones de configuración (Configuration)[31]

Una instancia de org.hibernate.cfg.Configuration representa un conjunto completo de correspondencias entre los tipos Java de una aplicación y los tipos de una base de datos SQL, además de contener un conjunto de propiedades de configuración. Una lista de las posibles propiedades de configuración y su explicación la podemos consultar en el manual de referencia de Hibernate incluido en la distribución (directorio doc\reference\en\pdf). Para especificar las opciones de configuración, se pueden utilizar cualquiera de las siguientes formas:

- Pasar una instancia de java.util.Properties a Configuration.setProperties().
- Establecer las propiedades del sistema mediante java-Dproperty=value.
- Situar un fichero denominado hibernate.properties en el *classpath.*
- Incluir elementos <property> en el fichero hibernate.cfg.xml en el *classpath.*

Las dos primeras opciones no se suelen utilizar, excepto para pruebas rápidas y prototipos. La mayoría de las aplicaciones requieren un fichero de configuración estático. Las dos últimas opciones sirven para lo mismo:

[31] (CCIA, 2005-2006)

configurar Hibernate, elegir entre una u otra depende simplemente de nuestras preferencias sintácticas.

4.6.2. Creación de una SessionFactory[32]

Para crear una SessionFactory, primero debemos crear una única instancia de configuration durante la inicialización de la aplicación y utilizarla para determinar la ubicación de los ficheros mapeados. Una vez configurada la instancia de configuration se utiliza para crear la SessionFactory. Una vez creada la SessionFactory, podemos olvidarnos de la clase Configuration.

Ejemplo 16 Inicialización de Hibernate

```
Configuration config = new Configuration();
config.addResource("test/Mensaje.hbm.xml");
config.setProperties (System.getProperties());
SesionFactory ses = config.buildSessionFactory();
```

La ubicación del fichero de mapeado, Message.hbm.xml, es relativa a la raíz del classpath de la aplicación. Por ejemplo, si el classpath es el directorio actual, el fichero Message.hbm.xml debería estar en el sub-directorio hello. En este ejemplo, utilizamos también las propiedades del sistema de la máquina virtual para determinar otras opciones de configuración (que también hubieran podido definirse antes en el código de la aplicación o como opciones de inicio).

El código anterior puede escribirse utilizando el estilo de programación denominado method chaining, que está soportado en la mayoría de interfaces de Hibernate. De esta forma, no necesitamos declarar una variable local para configuración. El ejemplo anterior quedaría como sigue:

[32] (CCIA, 2005-2006)

Ejemplo 17 Inicialización de Hibernate Versión 2

```
SessionFactory ses = new Configuration()
.addResource ("test/Mensaje.hbm.xml")
.setProperties (System.getProperties() )
.buildSessionFactory();
```

Por convención los ficheros de mapeado xml llevan la extensión hbm.xml. Otra convención es tener un fichero de mapeado por clase, en lugar de tener todos los mapeados en un único fichero (lo cual es posible, pero se considera un mal estilo). Nuestro ejemplo tiene solamente una clase persistente situada en el mismo directorio que dicha clase, pero si tuviésemos más clases persistentes, con un fichero xml para cada una de ellas, ¿dónde deberíamos situar dichos ficheros de configuración?

En la documentación de Hibernate se recomienda que el fichero de mapeado para cada clase persistente se sitúe en el mismo directorio de dicha clase. Se pueden cargar múltiples ficheros de mapeado con sucesivas llamadas a addResource(). De forma alternativa, también se puede utilizar el método addClass(), pasando como parámetro una clase persistente: así dejaremos que Hibernate busque el documento de mapeado por nosotros:

Ejemplo 18 Método addClass de Hibernate

```
SessionFactory sess = new Configuration()
.addClass(tesis.prueba.Usuario.class)
.addClass(tesis.prueba.Rol.class)
.addClass(tesis.prueba.Dependencia.class)
.setProperties(System.getProperties() )
.buildSesionFactory();
```

El método addClass() asume que el nombre del fichero de mapeado termina con la extensión .hbm.xml y que está desplegado junto con el fichero .class al que hace referencia.

En una instancia de configuration también podemos especificar las propiedades de configuración (mediante setProperties). En el ejemplo anterior, pasamos como parámetro las propiedades del sistema, pero podemos especificar propiedades concretas en forma de parejas (nombre_propiedad, valor_propiedad) como en el siguiente caso:

Ejemplo 19 Método setProperty Hibernate

```
SessionFactory sess = new Configuration()
.addClass (“tesis.prueba.Usuario.class")
.addClass (“tesis.prueba.Rol.class")
.setProperty ("hibernate.connection.datasource",
"java:comp/env/jcbc/prueba")
.setProperty("hibernate.dialect","org.hibernate.dialect.OracleDBD
ialect")
.setProperty ("hibernate.order_updates", "true")
.buildSessionFactory();
```

Acabamos de ver cómo crear una SessionFactory. La mayoría de las aplicaciones necesitan crear una SessionFactory. Si fuese necesario crear otra SessionFactory (si por ejemplo, hubiese varias bases de datos), habría que repetir el proceso. En este caso cada SessionFactory estaría disponible para una base de datos y lista para producir Sessions para trabajar con esa base de datos en particular y con un conjunto de ficheros de correspondencia de clases.

Por su puesto, configurar Hibernate requiere algo más que indicar cuáles son los documentos de mapeado. También es necesario especificar cómo se obtienen las conexiones a las bases de datos, el manejo de transacciones, entre otras cosas. De todas las opciones de configuración, las opciones de la base de datos son las más importantes. Estas opciones son distintas según estemos en un entorno gestionado o no gestionado. En nuestro caso, solamente vamos a comentar el segundo caso.

4.6.3. Configuración de la conexión de base de datos[33]

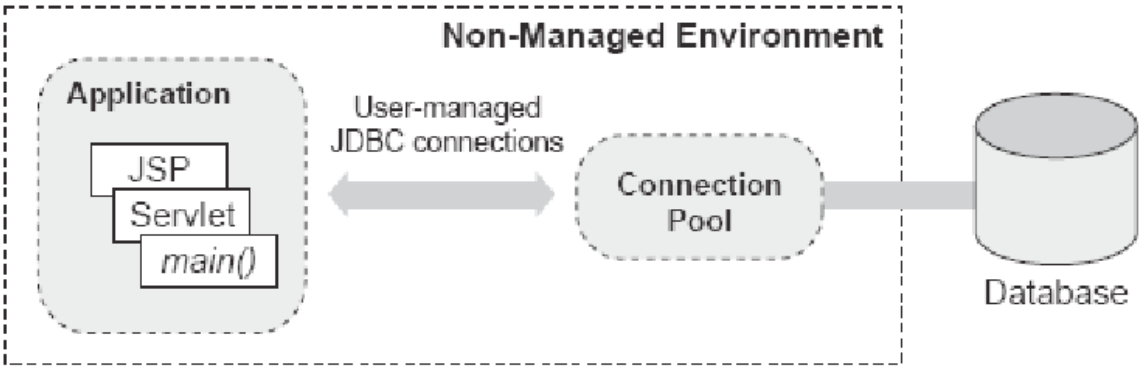
En un entorno no gestionado, como por ejemplo un contenedor de servlets, la aplicación es la responsable de obtener las conexiones JDBC. Hibernate es parte de la aplicación, por lo que es responsable de obtener dichas conexiones. Generalmente, no es conveniente crear una conexión cada vez que se quiere interactuar con la base de datos. En vez de eso, las aplicaciones Java deberían usar un pool de conexiones. Hay tres razones por las que usar un pool:

- Conseguir una nueva conexión es caro.
- Mantener muchas conexiones ociosas tiene un alto costo.
- Crear la preparación de sentencias es también caro para algunos drivers.

La siguiente figura muestra el papel de un pool de conexiones JDBC en un entorno de ejecución de una aplicación web (sin utilizar Hibernate). Ya que este entorno es no gestionado, no implementa el pooling de conexiones, por lo que la aplicación debe implementar su propio algoritmo de pooling o utilizar alguna librería como por ejemplo el pool de conexiones de libre distribución C3P0. Sin Hibernate, el código de la aplicación normalmente llama al pool de conexiones para obtener las conexiones JDBC y ejecutar sentencias SQL.

[33] (CCIA, 2005-2006)

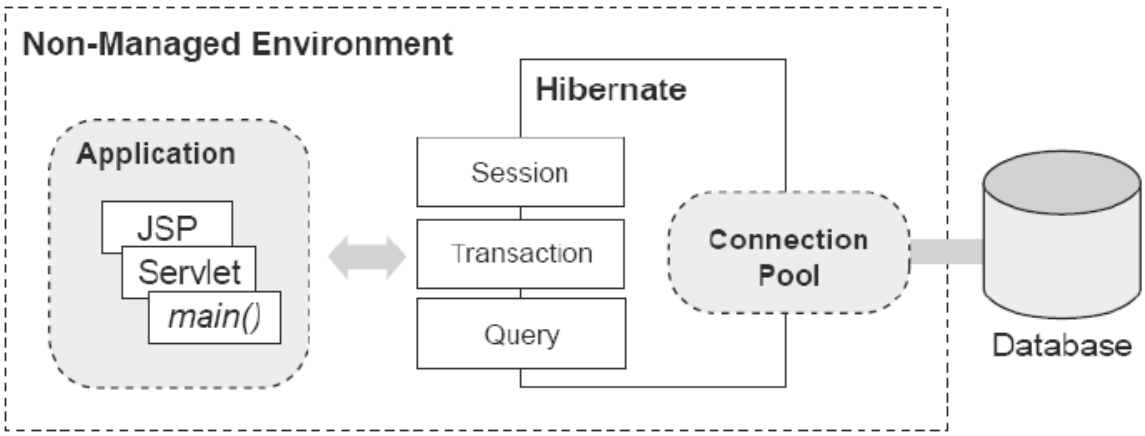
Figura 14 Pool de conexiones JDBC en un entorno no gestionado



Fuente: (Gaving King, 2007, pág. 45)

Con Hibernate, este escenario cambia: Hibernate actúa como un cliente del pool de conexiones JDBC, tal y como se muestra en la siguiente Figura. El código de la aplicación utiliza los API's Session y Query para las operaciones de persistencia y solamente tiene que gestionar las transacciones a la base de datos, idealmente, utilizando el API Hibernate Transaction.

Figura 15 Hibernate con un pool de conexiones en un entorno no gestionado



Fuente: (Gaving King, 2007, pág. 46)

Hibernate define una arquitectura de plugins que permite la integración con cualquier pool de conexiones. Puesto que Hibernate ya incluye soporte para C3P0, vamos a ver cómo usarlo.

Hibernate actualizará la configuración del pool por nosotros con las propiedades que determinemos. Un ejemplo de un fichero hibernate.properties utilizando C3P0 se muestra en el siguiente listado:

Ejemplo 20 Fichero hibernate.properties utilizando C3P0 Hibernate

```
hibernate.connection.driver_class = org.oracle.Driver
hibernate.connection.url = jdbc:oracle://localhost/pruebadb
hibernate.conection.password =  ”secret”
hibernate.connection.username = “usuario”
hibernate.dialect = net.sf.hibernate.dialect.PosgreSQLDialect
hibernate.c3p0.max_size=30
hibernate.c3p0.timeout=400
hibernate.c3p0.min_size=10
hibernate.c3p0.idle_test_period=3500
hibernate.c3p0.max_elements=60
```

Estas líneas de código especifican la siguiente información:

- El nombre de la clase Java que implementa el Driver JDBC (el fichero JAR del driver debe estar en el classpath de la aplicación.

- La URL JDBC que especifica el host y nombre de la base de datos para las conexiones JDBC.

- El usuario de la base de datos

- La contraseña de la base de datos para el usuario especificado

- Un Dialect para la base de datos. A pesar de esfuerzo de estandarización de ANSI, SQL se implementa de forma diferente por diferentes vendedores, por lo que necesitamos especificar un Dialect. Hibernate soporta el SQL de las bases de datos más populares.

- El número mínimo de conexiones JDBC que C3P0 mantiene preparadas.

- El número máximo de conexiones en el pool. Se lanzará una excepción si este número se sobrepasa en tiempo de ejecución.

- El período de tiempo (en este caso, 5 minutos o 300 segundos) después del cual una conexión no usada se eliminará del pool.

- El número máximo de sentencias preparadas que serán almacenadas en una memoria intermedia (caché). Esto es esencial para un mejor rendimiento de Hibernate.

- El tiempo en segundos que una conexión debe estar sin utilizar para que se valide de forma automática dicha conexión.

El especificar las propiedades de la forma hibernate.c3p0.* selecciona C3P0 como el pool de conexiones para Hibernate (sin necesidad de ninguna otra acción). Otros pools de conexiones soportados son Apache DBCP y Proxool.

4.6.4. Uso de configuraciones basadas en XML

Podemos utilizar un fichero de configuración XML para configurar completamente una SessionFactory. A diferencia del fichero hibernate.properties, que contiene solamente parámetros de configuración, el fichero hibernate.cfg.xml puede especificar también la ubicación de los

documentos de mapeado. Muchos usuarios prefieren centralizar la configuración de Hibernate de esta forma, en vez de añadir parámetros a Configuration en el código de la aplicación.

Ejemplo 21 Configuración Xml Hibernate

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-
2.0.dtd">
<hibernate-configuration>
<session-factory name="java:/hibernate/HibernateFactory">
<property name="connection.datasource">
java:/comp/env/jdbc/testDB
</property>
<property name="show_sql">true</property>
<property name="dialect">
net.sf.hibernate.dialect.OracleDialect
</property>
<mapping resource="test/Usuario.hbm.xml"/>
<mapping resource="test/Empleado.hbm.xml"/>
<mapping resource="test/Jefe.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

La declaración <!DOCTYOPE se usa por el analizador de XML para validar este documento frente a la DTD de configuración de Hibernate. El atributo “name” opcional es equivalente a la propiedad hibernate.session_factory_name y se usa para el enlazado JNDI de la SessionFactory.

Las propiedades de Hibernate pueden especificarse sin el prefijo Hibernate. Los nombres de las propiedades y valores son, por otro lado, idénticos a las propiedades de configuración especificadas mediante programación.

Los documentos de mapeado pueden especificarse como recursos de la aplicación. Ahora podemos inicializar Hibernate utilizando SessionFactory sessions = new Configuration().configure().buildSessionFactory();.

Cuando se llama a configure(), Hibernate busca un fichero denominado hibernate.cfg.xml en el classpath. Si se quiere utilizar un nombre de fichero diferente o hacer que Hibernate busque en un subdirectorio, debemos pasar una ruta al método “configure(): SessionFactory ses = new Configuration() .configure("/hibernate-config/configHibernate.cfg.xml).buildSessionFactory();”.

Utilizar un fichero de configuración XML ciertamente es más cómodo que usar un fichero de propiedades o mediante programación. El hecho de que se puedan tener ficheros de mapeado externos al código fuente de la aplicación es el principal beneficio de esta aproximación. Así, por ejemplo, podemos usar diferentes conjuntos de ficheros de mapeado (y diferentes opciones de configuración), dependiendo de la base de datos que utilicemos y el entorno (Desarrollo o producción), y cambiar entre ellos mediante programación. Si tenemos ambos ficheros, hibernate.properties, e hibernate.cfg.xml, en el classpath, las asignaciones del fichero de configuración XML prevalecen sobre las del fichero de propiedades. Esto puede resultar útil si queremos guardar algunas propiedades base y sobrescribirlas para cada despliegue con un fichero de configuración XML.

4.6.5. Configuración de logging[34]

Hibernate (y muchas otras implementaciones de ORMs) ejecuta las sentencias SQL de forma asíncrona. Una sentencia INSERT normalmente no se ejecuta cuando la aplicación llama a Session.save(); una sentencia UPDATE

[34] (CCIA, 2005-2006)

no se ejecuta cuando la aplicación llama a Item.addBid(). En vez de eso, las sentencias SQL se ejecutan al final de una transacción.

Esta característica evidencia el hecho de seguir una traza y depurar el código ORM es a veces no trivial. Una forma de ver qué es lo que está sucediendo internamente en Hibernate es utilizar el mecanismo de *logging*. Para ello tendremos que asignar a la propiedad hibernate.show_sql el valor true, lo que permite hacer logs en la consola del código SQL generado.

Hay veces en las que mostrar el código SQL no es suficiente. Hibernate "muestra" todos los eventos interesantes utilizando el denominado commons-logging de Apache, una capa de abstracción que redirige la salida al log4j de Apache (si colocamos log4j.jar en le classpath) o al logging de JDK1.4 (si estamos ejecutando bajo JDK1.4 o superior y log4j no está presente). Es recomendable utilizar log4j, ya que está más maduro, es más popular, y está bajo un desarrollo más activo.

Para ver las salidas desde log4j necesitamos un fichero denominado log4j.properties en nuestro classpath. El siguiente ejemplo redirige todos los mensajes log a la consola:

Ejemplo 22 Redireccionar el log de Hibernate a la salida de la consola

```
###Dirigir los mensajes a la consola###
log4j.appender.stdout.Target = System.out
##Enviarlo a consola
log4j.appender.stdout = org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
###Opciones de root###
log4j.rootLogger=warn, stdout
###Opciones de Hibernate Log.###
log4j.logger.net.sf.hibernate=info
###cache, log###
log4j.logger.net.sf.hibernate.ps.PreparedStatementCache=error
###parámetros log JDBC###
log4j.logger.net.sf.hibernate.type=info
```

Con esta configuración, no aparecerán muchos mensajes de log en tiempo de ejecución. Si reemplazamos info por debug en la categoría log4.logger.net.sf.hibernate se mostrará el trabajo interno de Hibernate.

4.7. Trabajando con objetos mapeados

Al trabajar con objetos mapeados, Hibernate provee una gran cantidad de funciones y formas de mapeo para manipular los datos resultantes, se pueden establecer relaciones de many to many, one to many, one to one con el atributo inverse le indicamos a Hibernate que la relación es bidireccional.

Con el atributo cascade=”save-update”, le indicamos a hibernate que persista todos los extremos de las relaciones sin necesidad de realizar un save explicito, con el atributo “all-delete-orphan” indicamos que el padre es el responsable del ciclo de vida del hijo.

Hibernate soporta modelos de granulo fino, por lo que se pueden mapear varias clases en una sola tabla, también cuenta con persistencia transitiva lo que significa que todos los objetos desde una instancia persistente, se vuelven persistentes.

4.7.1. Crear un objeto

Si se desea insertar un objeto a la base de datos, basta con que la clase este mapeada e incluida en el archivo de configuración de Hibernate, luego con la interfaz session, se procede a ejecutar la función saveOrUpdate.

Ejemplo 23 Insertar un fila en Hibernate

```
Alumno Alumno1 = new Alumno();
Alumno1.setName(“Victor Gonzalez“);
Alumno1.setCarne(“200230367”);
```

```
Configuration config = new Configuration();
//Se crea la sesión
SessionFactory fact = config.buildSessionFactory();
//Se abre la sesión
Session ses = fact.openSession();
ses.saveOrUpdate(Alumno1);
ses.flush();
ses.close();
```

4.7.2. Modificar un objeto

Por defecto se escriben todas los campos de la tabla, con la propiedad "dynamic-update = true", solo se modifican los campos que han sido modificados, es decir se genera una sentencia update que contiene solo las columnas que fueron modificadas.

Ejemplo 24 Modificar un objeto con Hibernate

```
Alumno1.setCarne("200230368");

Configuration configuracion = new Configuration();
//Se crea la sesión
SessionFactory fact = configuracion.buildSessionFactory();
//Se abre la sesión
Session ses = fact.openSession();
ses.saveOrUpdate(Alumno1);
ses.flush();
ses.close();
```

4.7.3. Obtener una lista de objetos

En Hibernate existen varias formas de obtener los objetos de la base de datos.

- Navegar en el grafo de objetos persistidos, empezando por un objeto ya cargado.

- Cargándolo por medio de su identificador ya sea por medio de get(Class, Id), o por medio de load(Class, Id), load a diferencia de

get envía una excepción si no se encontrase el objeto, mientras que get retorna null.

- Utilizar HQL, Utilizar la API Criteria que permite hacer consultas, permite especificar restricciones dinámicamente sin trabajar directamente con cadenas, parseado en tiempo de compilación.

Ejemplo 25 Búsqueda con interfaz criteria

```
Criteria criterios = session.createCriteria(Usuario.class);
criterios.add( Expression.like("carne", "200230367") );
List resultado = criterios.list();
```

Ejemplo 26 Busqueda con HQL

```
Query query = session.createQuery("from Alumno u where u.carne =
:fcarne");
query.setString("fcarne", "200230367");
List result = q.list();
```

4.7.4. Eliminar un objeto

Para eliminar un objeto de la base de datos con Hibernate, simplemente es necesario ejecutar la función delete desde la interfaz sessión, esto produce que el objeto se vuelva transitivo

Ejemplo 27 Borrar un objeto con Hibernate

```
session.delete(Alumno1);
```

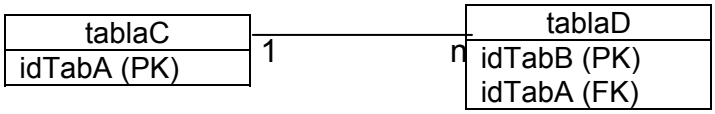
4.7.5. Colecciones[35]

Cuando se tienen relaciones de uno a muchos entre dos clases que son persistentes, como ejemplo C y D, dentro de la clase C, se debe colocar una colección (Array) de Objetos de la clase D.

[35] (García, 2006)

Podemos visualizar esta relación en la base de datos por medio de dos tablas: tablaC y tablaD, donde en tablaD se encuentra un campo que es la llave extranjera de la tabla tablaC.

Ejemplo 28 Mapeo Hibernate colecciones




```
PK = Llave primaria
FK = Llave extranjera
```

```
<hibernate-mapping>
     <!—Inicio Hibernate mapping -->
    <class table="tablaC" name="classA">
         <!-- Llave primaria -->
        <id name="idTabA" column="idTabA">
            <generator class="native"/>
        </id>

        <!-- Array de objetos de B -->
        <set name="ArraysB">
            <one-to-many class="classB" />
            <key column="idTabA" />
        </set>
     </class>
</hibernate-mapping>
```

```
<hibernate-mapping>
    <class name="classB" table="tablaD">
        <!-- *Llave primaria* -->
        <id name="idTabB" column="idTabB">
            <generator class="native"/>
        </id>
     </class>
</hibernate-mapping>
```

De manera predeterminada, cuando intentemos recuperar los objetos de la clase classA, Hibernate 3 realizará una consulta al repositorio de datos para extraer la información correspondiente a la tabla tablaC, pero Hibernate no extraerá la información inmediatamente, solamente proporcionará un proxy.

Este proxy es el encargado de realizar las consultas al repositorio de datos cuando verdaderamente se intente acceder a la colección de Objetos de la D.

Según esta forma de acceso, para extraer la información de la tablaC y sus objetos relacionados de tablaD es necesario realizar n+1 (donde n es el número de registros de classC) accesos al repositorio de datos.

En la mayoría de casos esta forma de acceso es adecuada, puesto que no se recuperan absolutamente todos los registros de la tabla “tablaD” hasta que sean realmente necesarios. Esta forma de acceso evita traer todo el repositorio de datos a memoria cuando se tienen muchas tablas relacionadas.

Este mecanismo puede ser inadecuado, ya que este realiza múltiples accesos pues cuando se pudiera realizar la extracción de datos con una sola consulta, pero mejoramos el rendimiento de nuestra aplicación al momento de realizar consultas por separado.

4.7.6. Fetch & Lazy

Para mejorar el rendimiento de las aplicaciones, debemos comprender como y cuando Hibernate obtiene la información del repositorio de datos.

4.7.6.1. Lazy Loading

Al momento de realizar una relación en Hibernate utilizamos el atributo “lazy” para definir cuando se obtendrá esta información. Por defecto la propiedad “lazy” tiene el valor “true”, esto quiere decir que la colección de Objetos se recupera cuando se realice una acción sobre este.

Si el atributo “lazy” lo establecemos false, en el momento de realizar la consulta traería el objeto principal con su colección de objetos (Este era la opción por defecto de la versión 2 de Hibernate).

4.7.6.2. Fetch

Por otra parte, el atributo “fetch” define como obtendrá Hibernate la información del repositorio de datos. Por defecto esta propiedad tiene el valor “select”. Cuando se tiene el valor “select” en esta propiedad, Hibernate realiza un acceso a la base de datos para para obtener cada objeto classC.

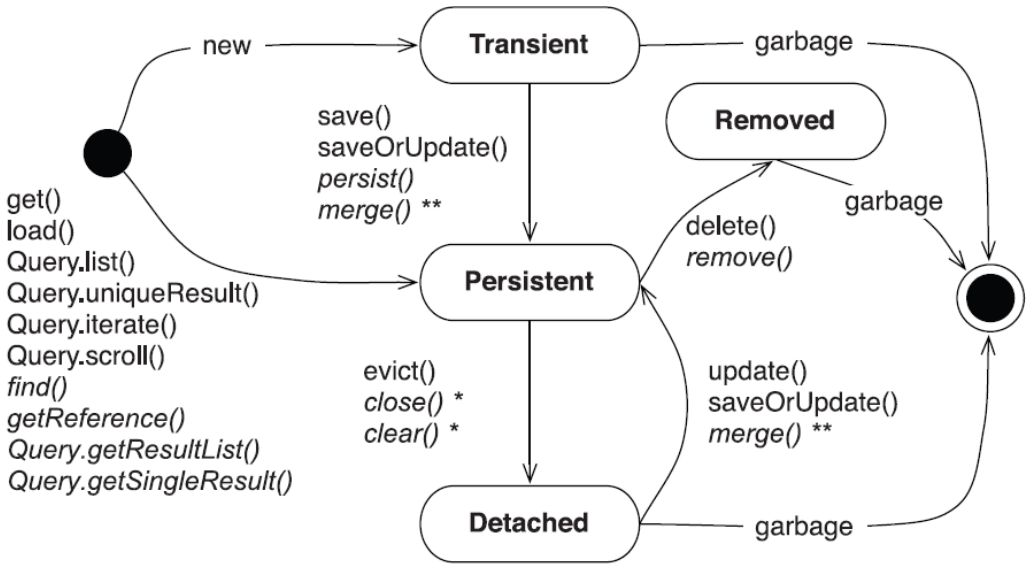
Al establecer fetch con valor “join”, Hibernate obtendrá todos los datos con una sola consulta, simplemente realiza un left outer join con las tablas relacionadas en una misma sentencia SQL.

4.7.7. Estados de los objetos en Hibernate

Las diferentes soluciones ORM usan diferentes terminologías y definen diferentes estados y estados de transición para el ciclo de vida de los objetos. Por otra parte, los estados de los objetos utilizados pueden ser internamente diferentes a lo presentado en la aplicación del cliente. Hibernate define solo 4 estados, ocultando la complejidad de la implementación interna del código del cliente.

Los estados de los objetos definidos por Hibernate se presentan en la figura siguiente, se puede notar que cada método invocado por el administrador de persistencia es el disparador para la transición de estado. El Api en Hibernate es el objeto Session.

Figura 16 Estados de objetos en Hibernate y sus disparadores



Fuente: (Gaving King, 2007, pág. 386)

4.8. Caché

La arquitectura de caché de Hibernate es realmente potente. Su caché de dos niveles ofrece una gran flexibilidad al tiempo que un gran rendimiento tanto en sistemas standalone como en cluster. [36]

Al comprender completamente el funcionamiento de las caches de Hibernate, se podrá obtener el máximo de posibilidades, teniendo un aumento significativo en la escalabilidad y el rendimiento de nuestras aplicaciones, evitando así problemas de sincronización y concurrencia.

[36] http://www.ajlopez.net/TemaArticulos.php?Id=4

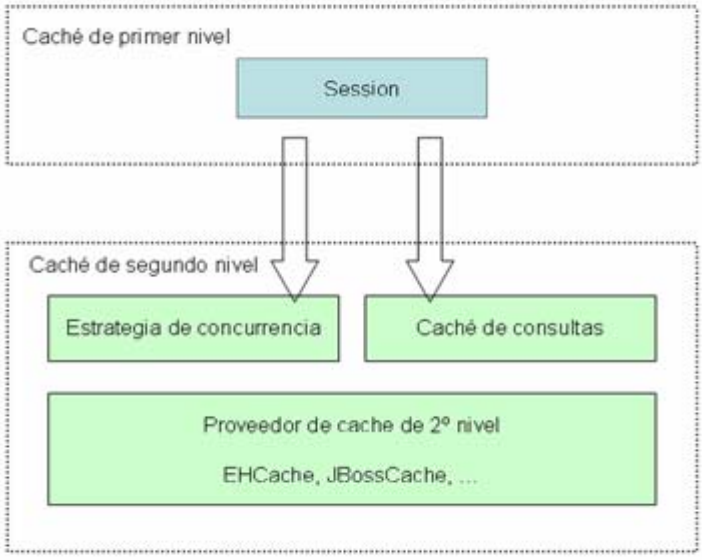
4.8.1. La arquitectura de caché de Hibernate

La arquitectura de caché de Hibernate se compone de dos niveles, cada uno con tareas my diferentes. El primer nivel se refiere a la caché de sesión, esta viene activada por defecto y no se puede desactivar.

La caché de segundo nivel en Hibernate se maneja de forma externa por medio de proveedores de caché, el alcance que podría tener es de proceso o de clúster. El uso de la caché de segundo nivel es opcional y puede ser asignada clase por clase o realizando una asociación base.

En la siguiente figura, se puede observar cómo está constituida la arquitectura de caché de dos niveles de Hibernate, tal como se describe anteriormente la caché de primer nivel se refiere a la sesión misma e interactúa con la caché de segundo nivel.

Figura 17. Arquitectura de caché de Hibernate



Fuente: http://www.lifia.info.unlp.edu.ar/es/difusion/20071024.htm

4.8.2. Cachés y concurrencia

Cualquier implementación ORM que permita múltiples unidades de trabajo para compartir la misma instancia de persistencia, debe proveer algún tipo de nivel de bloqueo para asegurar la sincronización de accesos concurrentes. Usualmente esta es implementada utilizando candados de lectura y escritura junto con la detección de deadlock. Implementaciones como Hibernate que mantienen un conjunto de instancias por unidad de trabajo, evitan estos tipos de problema en gran medida.

Un mecanismo de caché, se utiliza para almacenar en memoria objetos que pueden ser utilizados posteriormente, aumentando con esto el rendimiento de las aplicaciones y reduciendo los accesos a las bases de datos. Cuando se utiliza un mecanismo de caché, la forma de interactuar sería la siguiente:

1. Se realiza la consulta para obtener los objetos.

2. Se realiza una búsqueda dentro de la caché de primer nivel, para verificar si los objetos solicitados se encuentran allí, si esto es así entonces se devuelven de forma directa. Si dichos objetos no estuvieran en la caché, se procede a realizar la recuperación desde algún recurso externo y se guardan en la caché, por si se llegaran a necesitar en un futuro, nos ahorraríamos este paso, que sin lugar a duda es más costoso.

4.8.3. La caché de primer nivel[37]

La caché de session o caché de primer nivel, asegura que cuando la aplicación requiera el mismo objeto persistente en dos ocasiones en una particular sesión, este devuelve la misma instancia. Esto ayuda a no crear tráfico innecesario en la base de datos, pero lo más importante de esto es lo siguiente:

- La capa de persistencia no es vulnerable a desbordamiento de pila en el caso de tener referencia circular en el grafo de objetos.

- No puede existir conflicto al finalizar una transacción, esto se debe a que solo un objeto representa una fila en la base de datos. Todos los cambios en los objetos pueden escribirse de forma segura en la base de datos.

- Los cambios realizados en una particular unidad de trabajo están disponibles inmediatamente para todo lo que se ejecute dentro de la misma.

No se necesita realizar nada en especial para activar la caché de sesión. Esta siempre está activada y por razones evidentes no se puede desactivar.

Cada vez que se realice una acción sobre un objeto tales como: save(), update() o saverOrUpdate() y siempre que se recupere un objeto utilizando load(), find(), list(), itérate() o filter(), este objeto es agregado a la caché de primer nivel. El estado del objeto será sincronizado con la base de datos cuando se llama el método flush(). Si no se desea que la sincronización ocurra, o si se está utilizando un número grande de objetos y se necesita liberar

[37] (Py, 2005)

memoria, se puede llamar la instrucción evict() para remover los objetos y sus colecciones de la caché de primer nivel, esto puede ser útil en muchas situaciones.

Cuando se necesite realizar update o deletes masivos, no es recomendable utilizar la caché de primer nivel, esto puede causar un error de tipo OutOfMemoryException, es mejor realizar estas operaciones mediante procedimientos almacenados que realicen las modificaciones o eliminaciones directamente en la base de datos. Si por el contrario es necesario tener los objetos en memoria, se recomienda ejecutar la sentencia evict inmediatamente para liberar memoria. Para eliminar completamente todos los objetos de la caché de sesión, se debe llamar la instrucción Session.clear().

Cuando se requiere una sesión, Hibernate necesita obtener una conexión al repositorio de datos del pool de conexiones de Hibernate. Estas conexiones se liberan solamente cuando se cierra la sesión utilizando la instrucción close(). Es evidente que si se consumen muchas conexiones del pool, este podría quedarse sin conexiones. Hibernate proporciona la instrucción *disconnect()* que permite desconectar por momentos las conexiones, evitando que el pool se quede sin conexiones para servir.

4.8.4. La caché de segundo nivel

La caché de segundo nivel de Hibernate, tiene alcance de proceso o clúster; todas las sesiones comparten la misma caché de segundo nivel. La caché de segundo nivel tiene el alcance de SessionFactory.

Las instancias persistentes son guardadas en la caché de segundo nivel de forma separada. La separación es un proceso a nivel de bits como la

serialización (este algoritmo es mucho más eficiente que la serialización de Java).

La implementación interna del alcance proceso/cluster no es muy importante, es más importante el uso de las políticas, estrategias y proveedores físicos de caché.

Dependiendo el tipo de información que se maneje, requerirá un tipo especial de políticas de caché. En las aplicaciones, las relaciones de escritura/lectura y el tamaño de las tablas son variantes, además algunas tablas pueden ser compartidas con otras aplicaciones externas. La caché de segundo nivel es configurable para soportar este tipo de granularidad, esto permite desactivar la caché de segundo nivel para clases que son criticas, tal es el caso de las tablas financieras.

La política de caché consiste en realizar las actividades siguientes:

- Activar la caché de segundo nivel.
- Establecer la estrategia de concurrencia de Hibernate
- Configurar la política de expiración de la caché (tiempo de espera, LRU).
- Establecer el formato físico de la caché (archivos de índices, memoria, cluster replicado)

No es conveniente habilitar la caché para todas las clases, la caché solo debe estar disponible para aquellas clases cuyos accesos de lecturas sean considerables. Si se tienen clases en las cuales se realizan muchas actualizaciones, no se recomienda que se habilite la caché de segundo nivel.

La caché de segundo nivel puede ser un problema cuando se comparte con aplicaciones que realizan muchos acceso de escritura a las tablas.

La configuración de la caché de hibérnate, se realiza en dos simples pasos. Primero se debe decidir cual estrategia de concurrencia utilizar. Después de esto, se necesita configurar el tiempo de expiración y los atributos del formato físico de la caché utilizada por el proveedor de caché.

Este tipo de caché, ayuda a resolver el problema de las actualizaciones concurrentes, además esta se acoplara a la caché de sesión resolviendo todos los fallos que en esta se produzcan.

4.8.4.1. Caché en Hibernate

Hibernate permite habilitar individualmente la caché para cada una de las entidades. De este modo, se puede decidir que clases se beneficiarán del uso de una caché, ya que como se explicaba anteriormente, puede que no todas las clases de nuestro sistema se beneficien. Además de esto, al habilitar la caché es necesario establecer la estrategia de concurrencia que Hibernate utilizará para sincronizar la caché de primer nivel con la caché de segundo nivel, y ésta última con la base de datos. Hay cuatro estrategias de concurrencia predefinidas, a continuación aparecen listadas por orden de restricciones en términos de aislamiento transaccional.[38]

- **transactional**: Es una estrategia que solo puede ser utilizada en clúster con cachés distribuidas, garantiza un nivel de aislamiento hasta *repeatable read*, si es necesario. Es el nivel más estricto, se recomienda su uso cuando no podamos permitirnos datos que queden desfasados.[50]

[38] http://javahispano.org/articles.article.action?id=95

- **read-write**: Esta estrategia es recomendable en la misma situación que la transactional, con la única diferencia es que esta no puede ser utilizada con cachés distribuidas, su alcance es hasta el nivel de commit y utiliza un sistema de marcas de tiempo.

- **Nonstrict-read-write**: No garantiza la consistencia entre la caché y la base de datos. Si existe la posibilidad de acceso concurrente a una entidad, se debe configurar el tiempo de expiración suficientemente corto. Se debe utilizar esta estrategia cuando los datos raramente cambian (horas, días o semanas) y la información histórica no es de crítica importancia. Hibernate invalida la caché de un objeto que sea modificado y volcado a la base de datos, pero esta operación se realiza de forma asíncrona, sin ningún bloqueo de caché o garantía que la información recuperada por otro usuario sea la última versión.

- **read-only**: Es la estrategia de concurrencia para datos que nunca cambian. Esta estrategia se utiliza solo para referencia de datos.

4.8.4.2. Proveedores de caché

Para la caché de segundo nivel de Hibernate, se debe elegir un proveedor pues Hibernate no incluye ningún proveedor de caché, pero si posee gran soporte para estos. Los proveedores que se pueden utilizar con Hibernate son los siguientes:

- EHCache es destinado a un proceso simple en una sola JVM. Puede mantener la caché en memoria o en disco, y soporta la caché de consultas de Hibernate.

- OpenSymphony OSCache es una librería que soporta caché en memoria y en disco en una sola JVM, con un amplio conjunto de políticas de expiración y soporta la caché de consultas.

- SwarmCache es un clúster de caché basado en JGroups. Puede ser utilizada en un clúster pero no soporta la caché de consultas de Hibernate.

- JBossCache está completamente orientado a un clúster transaccional replicado de caché. También está basado en la librería JGroups multicast. Suporta la caché de consulta de Hibernate. Asume que los relojes de sincronización están en el clúster.

Existe una forma de hacer adaptaciones de otros productos simplemente implementando la interfaz net.sf.hibernate.cache.CacheProvider.

No todos los proveedores de caché son compatibles con todas las estrategias de concurrencias, la siguiente matriz ayuda a elegir una apropiada combinación.

Tabla VI. Soporte de estrategias de concurrencia

Cache Provider	read-only	nonstrict-read-write	read-write	transactional
EHCache	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBossCache	X			X

Fuente: Gaving King, C. B. (2007) P. 184

La configuración de la caché consiste básicamente en dos pasos:

- En los archivos de mapeo para las clases persistentes, elegir la estrategia de concurrencia que se desea utilizar.

- Habilitar el proveedor de caché en la configuración global de Hibernate y personalizar las configuración especifica del proveedor de caché, por ejemplo si se desea utilizar el proveedor OSCache, se necesita editar el archivo oscache.properties o EHCache el archivo ehcache.xml en el classpath.

Para establecer la configuración del proveedor se necesita agregar la siguiente línea en el archivo global de configuración de Hibernate.

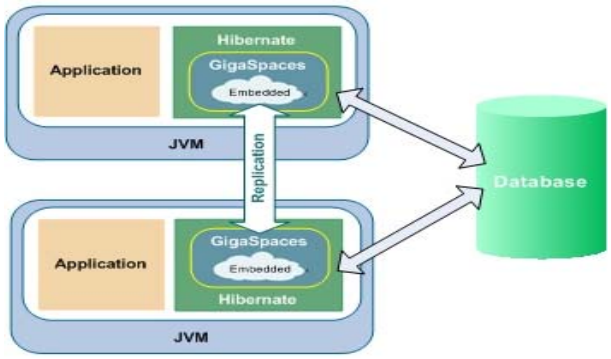
```
hibernate.cache.provider_class =net.sf.ehcache.hibernate.Provider
```

4.8.5. Caches distribuidas

En una aplicación empresarial donde existen decenas de miles de usuarios, es necesario que las aplicaciones se ejecuten en un ambiente distribuido, en otras palabras en un *clúster*. En estas circunstancias, la única solución es utilizar la caché de segundo nivel.

Existen dos formas de implementar una caché de segundo nivel forma distribuida. La primera y más sencilla, consiste en colocar una instancia del proveedor de caché en cada nodo, esperando que cada instancia en los nodos se sincronicen entre ellos, esta forma es muy simple pues no existen retardos de sincronización.

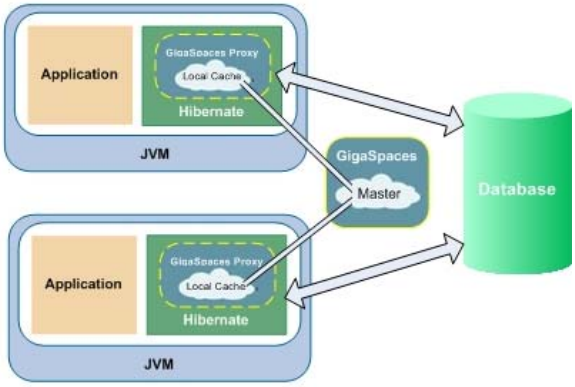
Figura 18. Modelo de caché Hibernate, replicación embebida



Fuente: http://www.gigaspaces.com

La segunda forma consiste en permitir que un proveedor de caché más sofisticado realice el trabajo de sincronización entre las cachés de los diferentes nodos. Uno de los proveedores que se recomienda para Hibernate es JBossCache, ya que es un proveedor totalmente transaccional que se basa en la librería de *multicasting*, *JGroups*.

Figura 19. Modelo de caché Hibernate, topología Master-Local



Fuente: http://www.gigaspaces.com

De la misma forma en que se configura el proveedor EHCache, se necesita añadir solamente una línea al archivo de configuración de hibérnate para activar el proveedor de caché:

```
hibernate.cache.provider_class=net.sf.hibernate.cache.TreeCacheProvider
```

La configuración de JBossCache se realiza mediante la utilización del archivo *treecache.xml*, este archivo debe establecerse en cada uno de los nodos del *clúster*. Este archivo es más complejo que el de EHCache pues en este se debe especificar la configuración del clúster, las políticas de sincronización entre los nodos, etc. El siguiente ejemplo se como podría ser la configuración de JBossCache.

Ejemplo 29 Configuración de Regiones de Caché JBossCache

```
<server>
   <attribute name="CacheMode">REPL_SYNC</attribute>

   <attribute name="EvictionPolicyConfig">
     <config>
       <attribute name="wakeUpIntervalSeconds">10</attribute>
       <region name="/_default_">
         <attribute name="timeToIdleSeconds">500</attribute>
         <attribute name="maxNodes">4000</attribute>
       </region>
       <region name="/org/hibernate/auction/model/Category">
         <attribute name="timeToIdleSeconds">4000</attribute>
         <attribute name="maxNodes">400</attribute>
       </region>
       <region name="/org/hibernate/auction/model/Bid">
         <attribute name="timeToIdleSeconds">1800</attribute>
         <attribute name="maxNodes">4000</attribute>
       </region>
     </config>
   </attribute>
   ................
 </server>
```

Fuente: http://javahispano.org/articles.article.action?id=95

4.8.6. Caché de consultas

Hibernate cuenta con una caché de consultas ejecutadas en tiempo reciente, de forma que si estas se vuelven a utilizar, Hibernate recupera los resultados de forma más ágil. Este tipo de caché solo se recomienda cuando se tiene una aplicación que realizase muchas consultas y que además estas consultas se repitan constantemente. Cuando se tengan sentencias como actualización, inserción y eliminación, no es muy conveniente activar la caché de consultas de Hibernate pues este invalida automáticamente en la sesión todas las consultas que tengan que ver con dichas operaciones.

La forma de habilitar la caché de consultas de Hibernate es muy sencilla, para ello simplemente debemos agregar en el archivo de configuración de Hibernate la sentencia que habilita dicha caché:

```
hibernate.cache.use_query_cache=true
```

Para aprovechar la caché de consultas de Hibernate, se debe utilizar la interfaz Query, pues de cualquier otra forma Hibernate ignora la consulta y no la almacena en la caché de consultas. Un ejemplo sería el siguiente:

```
  Query queryJefes = session.createQuery("from Jefes j where
j.nombre = :nombre");
  queryJefes.setString("nombre","VICTOR");
  queryJefes.setCacheable(true);
```

4.9. Quiénes utilizan Hibernate

A continuación se listan algunos sitios públicos que utilizan Hibernate.

Tabla VII Listado de empresas que utilizan Hibernate.

Wilos - http://www.wilos-project.org/
Company name, Location: SoftSlate Commerce, NY, USA
2Fi Business Solutions Ltd. (Hong Kong)
argus Barcelona (Europe)
GPI Argentina, La PLata, Buenos Aires, Argentina
TDC Internet (Warsaw, Poland)
Proyecto Open Source itracker
TerraContact Inc.,
LF Inc. (Tampa, FL)
Ubik-Ingénierie, ubik-ingenierie.com, Roubaix, France
Sony Computer Entertainment Europe, SCEE, Studio Liverpool, Liverpool, United Kingdom
Elastic Path Software (Vancouver, BC, Canada)
Skillserv, skillserv.com, San Francisco, California, USA
Fedelta POS, fedeltapos.com, Brisbane, Australia
AT&T Labs, Tampa (Florida)
AonCHOR http://www.aonchor.aon.com
Jteam (Amsterdam, The Netherlands)
1Genia (Paris, France)

PriceWaterhouseCoopers (Tampa, Florida)
Intrasoft International (Belgium, Brussels)
Church and People (New York)
Burgerweeshuis (Netherlands, Deventer)
Cisco Learning Institute (Phoenix, AZ USA)
Pyromod Software Inc, Creator of BestCrosswords.com, Montreal, Canada. (http://www.pyromod.com)
Open Lab S.r.l (Florence I)
DriveNow (Australia)
Crank Clothing, t-shirts and apparel
Mailvision, End-to-End SIP solutions, Israel. (http://www.mailvision.com)

Fuente: http://www.hibernate.org/Documentation/WhoUsesHibernate

4.10. Licencia

Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.[39]

4.11. Soporte y capacitación para Hibernate

Todo el soporte y consultoría para Hibernate es proporcionado mediante Jboss, una división de Red Hat.

El soporte profesional ayuda a sobrellevar todos los problemas relacionados con Hibernate, incluyendo bugs y administración de parches, soporte de producción y asistencia en despliegue. Están disponibles tres niveles de soporte, que van desde soporte de 8x5 con respuesta a las 24 horas a 24x7 con respuesta a las 2 horas.

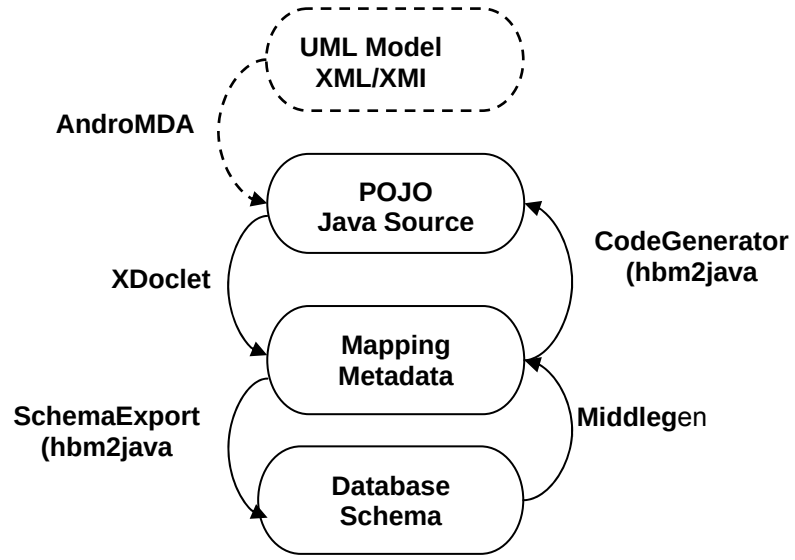
Jboss básicamente ofrece servicios como: Certificaciones, cursos de capacitación y Consultoría.

[39] http://es.wikipedia.org/wiki/Hibernate

4.12. Herramientas de apoyo

Hibernate cuenta con muchas herramientas de apoyo para generar el xml que lo describe y ahorra mucho trabajo al momento de mapear las tablas, algunas generan el xml y el código java a partir del esquema de base de datos, otros generan el xml a partir de metadata creada en las clases java, etc.,

Figura 18 Ciclo de Generación de código y Herramientas



Estas herramientas nos ayudan desde que se inicia el ciclo de vida del software con el modelado. Poseidon puede generar los modelos de entidad relación, luego estos son traducidos por androMDA a objetos planos de Java y luego utilizando XDoclet se crean los descriptores xml, estas tareas pueden ser automatizadas a través de tareas ANT.

Una opción alternativa consiste en crear los scripts para la base de datos mediante una herramienta CASE, luego generar los descriptores xml mediante Middlegen y por último generar los objetos planos de Java con hbm2java.

A continuación se listan algunas de las herramientas que se pueden utilizar de apoyo para Hibernate.

- Hibernate Tools: Ahorra mucho trabajo puesto que al utilizarlo con Ant o con Eclipse puede generar los archivos xml y pojos, correspondiente a las tablas. Referencia: http://www.hibernate.org/255.html.

- Hibernator: Herramienta que se Utiliza para generar los descriptores xml y los pojos a partir de un esquema de base de datos. Referencia: http://hibernator.sourceforge.net/

- XDclet: Es una herramienta para la generación de código o XML a partir de “doclets”, marcas que se incluyen en los comentarios de un programa. Referencia: http://xdoclet.sourceforge.net/olddocs/

- AndroMDA: Se pronuncia "Andromeda” es un programa informático de tipo *framework*, de generación extensible de código que se adhiere al paradigma de la arquitectura dirigida por modelos. Referencia: http://www.javahispano.org/canyamo.action

- Middlegen: Es un producto Open source de generación de código, este genera los archivos de configuración utilizando herramientas como JDBC, Velocity, Ant y XDoclet.

5. ANÁLISIS COMPARATIVO

Luego de haber concluido con las características principales de Hibérnate e Ibatis, entramos a una sección muy importante; el análisis comparativo. Este análisis se basa en los distintos parámetros que se deben considerar al momento elegir un ORM, estas características principales nos garantizan un ORM maduro que nos puede aportar muchos beneficios al momento de construir nuestras aplicaciones.

5.1. Factores a considerar al elegir un Orm

a) Curva de aprendizaje.

Al referirse a curva de aprendizaje, se refiere a la facilidad con la que se puede aprender una herramienta. Tanto Hibernate como Ibatis, son fáciles de aprender, la mayor parte de configuraciones vienen por defecto y son intuitivas Ibatis saca una ligera ventaja sobre Hibernate, su configuración es más sencilla, Hibernate por otra parte posee configuraciones más avanzadas en el manejo de la caché, las relaciones entre tablas, los updates automáticos, esto ocasiona configuraciones no tan sencillas para el usuario.

b) Facilidad de Debug

Cuando se está desarrollando, algo muy importante son las herramientas que proveen para encontrar errores dentro de nuestras aplicaciones, podemos pasar mucho tiempo tratando de encontrar un error muy sencillo si no se cuentan con estas herramientas, máxime trabajando con *framework's,* cuya interioridad es desconocida,

dependemos mucho de la información que ellos nos puedan brindar para encontrar errores.

Cuando se trabaja con Hibernate e Ibatis estos nos envían mensajes de información, debug, warnings para poder detectar problemas en nuestro código, ambos utilizan log4j, el cual es un potente y conocido proyecto para el debugeo de aplicaciones. Entre las características principales de esta herramienta están: diferentes niveles de trazas, redirección de las trazas a diferentes salidas (base de datos, consola, archivo), configuración por medio de ficheros, filtros por categoría y diferentes formatos de visualización.

c) Facilidad de Debug (errores SQL)

Cuando se está trabajando con un ORM, las sentencias SQL las maneja dicha herramienta, pero si existiese algún error de sintaxis o semántico, la herramienta debe proveer información explicita de lo que está ocurriendo.

En los Orm analizados en ambos se puede activar que nos brinden información detallada de lo que está ocurriendo con nuestro SQL.

d) Rendimiento

Es bien sabido que en términos de rendimiento utilizar un jdbc es mejor, pero de igual forma si se sabe configurar bien los mecanismos de caché que proveen los *framework's* ORM, vamos a tener resultados sorprendentes en el rendimiento de nuestras aplicaciones.

Hibernate posee una arquitectura de caché muy potente, ya que posee una arquitectura de caché de dos niveles que brinda una gran flexibilidad y alto rendimiento para nuestras aplicaciones; sin embargo un punto a favor de Ibatis es que además de poseer un mecanismo de caché, el código SQL se crea en tiempo de desarrollo, mientras que Hibernate construye las consultas SQL en tiempo de ejecución, en base a los descriptores xml de las tablas mapeadas y de las sentencias HQL, además Hibernate almacena gran cantidad de información en memoria referente a las relaciones, atributos y configuraciones de las tablas mapeadas, el cual tiene un costo de memoria y repercute en el rendimiento de la aplicación, especialmente al inicio cuando se cargan todos los descriptores xml.

e) Mapeo Objeto Relacional

La función principal de un Orm es realizar el mapeo objeto relacional, esto incluye los objetos y sus relaciones, cada *framework* realiza este trabajo de distintas formas, algunos no están completos y no permiten el mapeo del dominio completo de tablas. Hibernate en cuanto a mapeo Objeto Relacional es mejor que Ibatis, pues se puede tener el mapeo de todo el dominio relacional.

f) Cantidad de Trabajo

Cuando se intenta utilizar una herramienta ORM, uno de los principales objetivos es reducir el trabajo al momento de desarrollar o de dar un mantenimiento de una aplicación. En Ibatis e Hibernate muestran un gran ahorro de trabajo al obtener los datos en forma de listas de objetos. Una de las principales ventajas de Ibatis sucede cuando se tienen los SQL’s ya definidos, el trabajo se reduce a pegar el SQL en los descriptores XML.

g) Integración

Se dice que un ORM puede integrarse fácilmente, cuando sin tantas modificaciones se puede incorporar dicha tecnología para hacer uso de los beneficios de arquitectura que proveen.

En este caso Ibatis aventaja a Hibernate, pues en Ibatis se siguen utilizando las mismas consultas que se tenían, por lo que se prefiere Ibatis en cuanto a actualizaciones de software se refiere.

h) Intrusión

La intrusión se refiere a que tan invasivo es nuestro *framework* en nuestra aplicación, Hibernate e Ibatis utilizan reflection para crear los objetos y de esta forma no tener que heredar de clases externas, los objetos utilizados son simples pojo's.

i) Reusabilidad

Cuando se trabaja en componentes, estos se pueden volver a reutilizar, esto ahorra mucho trabajo, pues no es conveniente volver a realizar algo ya realizado, la mayoría de *framework's* promueven esta característica pues brindan un bajo acoplamiento entre capas. La reusabilidad, depende mucho de la forma en la que se programa aunque algunos *framework* permiten que sus componentes sean más reutilizables que otros.

j) Soporte para Transacciones

La información almacenada en una base de datos necesita ser protegidos por una transacción. Esto permite múltiples inserciones, modificaciones y borrados con la seguridad de que todo o se ejecuta o falla, como si fuera una sola entidad coherente.

Esta característica es vital en un ORM, aquí el trabajo se complica pues debe mantener las transacciones sobre los objetos modificados, además realizar transacciones locales, globales, genéricas y automáticas.

Hibernate e Ibatis, proveen mecanismos que facilitan el uso de transacciones, por otro lado, Ibatis posee Dao que se encarga del manejo de transacciones independiente de que tecnología de persistencia utilicemos.

k) Escalabilidad

Un Orm debe permitir el crecimiento de la aplicación sin tener que volverse a diseñar la aplicación, en este caso Hibernate posee una caché de doble capa que podría ser utilizada en un cluster e Ibatis soporta cacheado de consultas, incluso distribuido para entornos con contenedores en clúster.

l) Facilidad de refactorización

La refactorización es la capacidad de reestructurar el código fuente, modificando la estructura interna sin cambiar el comportamiento externo. Esta característica es muy importante en los Orm, pues para poder realizar actualizaciones y mejoras en los atributos no funcionales, es necesario modificar la estructura interna de la aplicación sin cambiar la lógica de negocio. Como ejemplo tunning de las consultas.

La refactorización en Ibatis es mejor que Hibernate, pues se pueden realizar mantenimientos a las consultas, sin tener que tocar una sola línea de código fuente.

m) Persistencia transitiva (estilo de cascada)

Persistencia por alcanzabilidad, todos los objetos alcanzables desde una instancia persistente se hacen persistentes. Es un algoritmo recursivo.

Hibernate provee persistencia transitiva, esta propiedad en Hibernate es muy útil pues a veces necesitamos grabar el objeto padre y con esto queremos que automáticamente todos los hijos sean persistentes.

n) Herramientas de apoyo (e.j. generación de esquema BD)

Cuando los archivos de configuración son descriptores XML, es muy tedioso crear configuraciones a mano de cada tabla que se quiera utilizar. Por eso, aunque un ORM sea una maravilla, hay que tener en cuenta que debe contar con herramientas que permitan fácilmente obtener los descriptores XML a partir del esquema de base de datos, o inclusive al revés, a partir de los archivos de configuración crear el esquema de base de datos respectivos.

Tanto Ibatis como Hibernate cuentan con herramientas muy buena de generación de archivos de configuración, Ibatis por su parte tiene a Abator, una muy buena herramienta que no solo genera pojo's y descriptores XML con las operaciones CRUD de las tablas, sino que a la vez genera la configuración y clases para implementar el patrón Dao, este patrón puede implementar varias plantillas de manejo de persistencia, tanto de Ibatis como de Hibernate, entre otras.

Hibernate también cuenta con varias herramientas de mapeo muy buenas, entre ellas están: androMDA, XDoclet, Hibernator, entre otros.

o) Características Adicionales

Además de lo anteriormente mencionado, un ORM debe proveer características adicionales como: devolver listas de Objetos, devolver las propiedades de los objetos en XML, crear proxies de objetos, Map, Collections.

Hibernate e Ibatis tienen la posibilidad de devolver la información obtenida de la base de datos en cualquiera de las formas mencionadas anteriormente, Hibernate tiene más características pero en términos de mapeo objeto/relacional, esto le permite a Hibernate poseer un mayor control de los objetos y sus relaciones obtenidas de la base de datos (ej. Conocer que objetos son transitivos y cuales son persistentes).

p) Documentación y Soporte

La documentación y el soporte de Hibernate e Ibatis, es muy extensa, por tratarse de software muy popular, libre y tener el apoyo de empresas como Jboss y Apache poseen una gran comunidad de usuarios y expertos dispuestos a compartir sus experiencias.

5.2. Cuándo y Cómo elegir un ORM

Haciendo uso de los *framework's* ORM de forma adecuada, realmente se logra una reducción de un 30% a un 40% de código, aparte de esto se tiene una aplicación mas mantenible pues los ORM permiten separar el acceso a Datos de la lógica de negocio, más ordenada al establecer cada cosa en su lugar, pudiendo dividir la aplicación en distintas tareas, algo muy importante cuando se trabajan proyectos grandes.

Tantas ventajas que nos proveen los ORM, sería algo no muy inteligente no considerar utilizar estas herramientas en nuestras aplicaciones, pero para poder tener éxito en la creación de aplicaciones con estas, se debe tomar una decisión muy importante: "Que framework ORM debo utilizar", esto determinara el éxito de nuestra aplicación.

Por lo general, las malas experiencias con ORM's se dan cuando se toman decisiones equivocadas en cuanto a cual utilizar, se deben conocer cuáles son las características de cada ORM y tener la mayor información acerca de estos para tomar la mejor decisión.

En conclusión antes de tomar una decisión, mínimo se deben realizar las siguientes tareas:

- Investigar y comparar, se debe conocer muy bien cuáles son las ventajas y desventajas de cada uno, por ejemplo Ibatis es sencillo de utilizar y permite una separación más apropiada del SQL utilizado.

- Experiencia de los Desarrolladores, sin duda un punto clave es que tanta experiencia tienen los desarrolladores en las herramientas a elegir, pues es muy diferente utilizar una

herramienta completamente nueva a utilizar alguna que ya se conoce.

- Documentación y Soporte, la herramienta elegida debe contar con soporte quien nos respalde en cualquier problema y documentación que nos permita obtener el mejor provecho de ella.

- Lo último no es siempre lo mejor, no se debe caer en la tentación de elegir la tecnología más reciente, puesto que pueden ser no muy maduras, hay poca documentación, puede que no sea la herramienta adecuado a lo que queremos, etc.

- Cada proyecto es diferente, recordemos que la arquitectura de software debe satisfacer tanto los requerimientos funcionales como los requerimientos no funcionales de nuestra aplicación, y que cada proyecto tiene diferentes requerimientos, distintas restricciones técnicas y de negocio, llegamos a la conclusión que no podemos utilizar la misma técnica para resolver problemas muy diferentes.

- Rendimiento vs Facilidad de desarrollo, algunas veces tenemos que sacrificar el rendimiento por la facilidad con la que podemos crear aplicaciones de manera rápida y mejor construidas (en términos de diseño).

- SQL sigue siendo necesario, al momento de elegir una herramienta ORM el utilizar SQL no debe ser el que determine que ORM utilizar, puesto que SQL sigue siendo el lenguaje estándar de consultas, a menos que se piense utilizar distintos dialectos para el acceso a datos.

- Considerar Alternativas, no podemos olvidar que existen otras alternativas a las bases de datos relaciones, por ejemplo: Bases de datos orientadas a objetos, XML, XMLBD.

5.3. Matriz comparativa

La matriz presentada a continuación muestra la comparación entre Ibatis e Hibernate, tomando en cuenta los Factores descritos anteriormente que se deben considerar al momento de la elección de un ORM.

Tabla VIII Matriz Comparativa

	JDBC	HIBERNATE	IBATIS
Curva de aprendizaje	*****	****	****
Rendimiento	*****	***	****
Mapeo objeto relacional	-	*****	**
Facilidad de debug	****	****	***
Facilidad de debug (errores SQL)	****	***	****
Cantidad de trabajo	-	****	****
Integración	*****	**	*****
Pool de conexiones	-	****	**
Intrusión	*****	*****	*****
Carga perezosa	-	*****	*
Caché	-	****	***
Caché de terceros	-	*****	*****
Múltiples lenguajes	-	*****	*****
Transacciones distribuidas	-	****	****
Simplicidad	-	***	*****
Batches	-		
Persistencia transitiva	-	*****	**
Reusabilidad	*	****	****
Escalabilidad	*	*****	*****
Soporte para transacciones	***	***	****
Facilidad de refactorización	*	***	*****
Herramientas de apoyo	-	*****	****

5.4. Interpretación de resultados

A lo largo de la investigación, podemos observar que ambos ORM's son muy buenos cumpliendo con casi todas las características que un ORM debe cumplir, mas sin embargo se pueden notar algunas diferencias que inclinan la balanza hacia uno u otro al momento de elegir cual utilizar.

Hibernate tiene soporte para la mayoría de bases de datos, posee un lenguaje de consultas propio que proporciona una conexión elegante entre el el modelo relacional y el paradigma orientado a objetos. Hibernate ofrece mecanismos para recuperar y actualizar los datos de manera sencilla, control de transacciones, pool de conexiones a base de datos, consultas declarativas y programáticas y un manejo de relaciones entre entidades declarativas.

Hibernate es menos invasivo que otro *framework's* ORM, pues utiliza Reflection, esto nos permite trabajar con objetos simples de java, pudiendo incluir en ellos: herencia, asociación, polimorfismo, composición y collection de java, además la generación del SQL lo realiza cuando crea la sesión para una mejora en el rendimiento, Hibernate es muy bueno en cuanto a mapeo objeto relacional se refiere, pero le hace falta funcionalidad y capacidad en el manejo de transacciones y conexiones.

Por otra parte, Ibatis trabaja con Jdbc por lo que no requiere de plugins para poder trabajar, posee una configuración de caché bastante simple, un control de transacciones locales y Globales, descriptores XML simples, soporta Maps, Collection, List, y tipos primitivos (Integer, String, etc.), soporta mapeo de objetos complejos, no requiere cambios drásticos para modificar el modelo de objetos, ni tampoco el diseño de base de datos, se conoce a plenitud el SQL que se está utilizando, por lo que se puede optimizar, rediseñar, etc.

En conclusión, es recomendable utilizar Ibatis cuando se tienen sistemas con muchas llamadas a procedimientos almacenados y sentencias SQL y como único mecanismo para obtener y modificar datos, en ese caso Hibernate no sería el ORM adecuado, Hibernate sería una buena opción cuando no se dan estas situaciones.

CONCLUSIONES

1. La utilización de un ORM permite desacoplar los componentes de software de tal forma que vamos a poder reutilizar dichos componentes de código, esto representa ahorros en costos económicos y de tiempo a corto plazo.

2. Al utilizar un ORM no se debe pensar que se va a ignorar el SQL. El mapeo Objeto/Relacional es simplemente para hacer más fácil el manejo de datos persistentes; no libera del uso de este.

3. Al momento de elegir un *framework* ORM, se debe tener en cuenta la arquitectura de software, la cual debe satisfacer tanto los requerimientos funcionales como los requerimientos no funcionales del proyecto.

4. Las malas experiencias con *framework's* ORM suelen ser resultado de intentar utilizarlos donde no es apropiado.

5. En Hibernate, la utilización adecuada de la caché de segundo nivel, puede corregir todos los problemas de la caché de primer nivel, aunque se debe de tener en cuenta que en ocasiones cuando se trate con aplicaciones sencillas, quizás no sea tan necesaria.

6. Hibernate proporciona una independencia entre la base de datos y la lógica de negocio, y un desarrollo rápido puesto que cubre de un 80% - 90% de la persistencia de nuestras aplicaciones, además cuenta con un

lenguaje de consultas propio “HQL” que lo hace multi-motor de base de datos, es menos intrusivo que otros *framework’s* ORM, puesto que utiliza Reflection y la genera de bytecodes en tiempo de ejecución.

7. Simplicidad es la mayor ventaja de Ibatis sobre otras herramientas de mapeo Objeto Relacional.

8. Es recomendable utilizar Ibatis cuando se tienen sistemas con muchas sentencias SQL y llamadas a procedimientos almacenados.

RECOMENDACIONES

1. Se debe utilizar un *framework* ORM, pues simplifica grandemente el desarrollo de la capa de persistencia, tratándose de una idea madura y que cada día gana más popularidad.

2. Los productos ORM son diferentes, por lo que se debe evaluar antes del inicio del desarrollo del proyecto qué herramienta satisface los requerimientos del mismo.

3. En un *framework* ORM se debe analizar las interioridades de su funcionamiento, pues una mala configuración ocasiona un deterioro en el rendimiento de la aplicación.

4. No se debe olvidar la caché de consultas de Hibernate, pues por muy sencilla que parezca puede darnos grandes resultados en el rendimiento de nuestra aplicación.

5. El utilizar SQL no debe ser lo que determine que ORM utilizar, puesto que SQL sigue siendo el lenguaje estándar de consultas, a menos que se piense utilizar distintos dialectos para el acceso a datos.

6. Hoy en día existen muchas herramientas ORM, tanto comerciales como código abierto. Es importante solo debe elegir una, además no es recomendable hacer una herramienta propia puesto que los resultados obtenidos nunca se comparan con las soluciones genéricas existentes.

BIBLIOGRAFÍA

1. Begin, C., Goodin, B., & Larry, M. (2007). iBATIS in Action. New York: Manning Publications Co.

2. CCIA, D. (2005-2006). www.jtech.ua.es/j2ee/2006-2007/doc/sesion03-apuntes.pdf. Recuperado el 18 de 06 de 2008, de www.jtech.ua.es/j2ee/2006-2007/doc/sesion03-apuntes.pdf: www.jtech.ua.es/j2ee/2006-2007/doc/sesion03-apuntes.pdf

3. Clinton, B. (18 de 02 de 2006). iBATIS Data Access Objects, Developer Guide.

4. Chambi Aruquipa, M., & Marquez Granado, E. (01 de Septiembre de 2005). Benchmaking de Frameworks de Desarrollo.

5. Fundation, A. S. (s.f.). Recuperado el 01 de 08 de 2008, de http://ibatis.apache.org/overview.html

6. Fundation, A. S. (s.f.). Ibatis Apache Org. Recuperado el 06 de 04 de 2008, de http://ibatis.apache.org/overview.html

7. García, A. P. (14 de 04 de 2006). http://www.adictosaltrabajo.com. Recuperado el 08 de 06 de 2008, de http://www.adictosaltrabajo.com: http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=hibernateJoin

8. Gaving King, C. B. (2007). Java Persistence With Hibernate. New York: Manning Publications Co.

9. Hibernate. (s.f.). Arquitectura Hibernate. Recuperado el 12 de 06 de 2008, de http://www.hibernate.org/hib_docs/reference/en/html/architecture.html

10. Husted, T., & Clinton, B. (s.f.). iBATIS Wiki. Recuperado el 17 de 02 de 2008, de http://opensource.atlassian.com/confluence/oss/display/IBATIS/Home

11. Jboss. (s.f.). Arquitectura Hibernate. Recuperado el 25 de 05 de 2008, de http://www.hibernate.org/hib_docs/reference/en/html/architecture.html

12. Jboss. (s.f.). http://www.hibernate.org/. Recuperado el 06 de 06 de 2008, de Hibernate: http://www.hibernate.org/113.html

13. Marquez Granado, M., & Aruquipa Chambi, M. (01 de 09 de 2005). Benchmaking de Frameworks de Desarrollo. Recuperado el 25 de 07 de 2008, de http://pgi.umsa.bo/enlaces/investigacion/pdf/INGSW3_44.pdf?PHPSESSID=6dea1abac4f1f4f71ea069d648fc73d6

14. Mota, K. (01 de 10 de 2006). Tutorial iBATIS SQL Maps.

15. Peak, P., & Heudecker, N. (2006). Hibernate Quickly. 209 Bruce Park Avenue: Manning Publications Co.

16. Pizarro, P. (2005). ORM, Object-Relational Mapping. Mendoza, Argentina.

17. Py, M. (18 de 01 de 2005). Cachés, concurrencia e Hibernate. Recuperado el 25 de 01 de 2008, de http://www.javahispano.org/contenidos/es/caches__concurrencia_e_hibernate/

18. Sevilla, E. U. (s.f.). Diseño de la capa de datos. De objetos a datos.

19. Sourceforge. (s.f.). iBATIS Data Mapper Developer Guide. Recuperado el 05 de 03 de 2008, de http://ibatisnet.sourceforge.net/DevGuide.html

20. Universidad Femenina del Sagrado, C. (s.f.). Persistencia Hibernate. Recuperado el 25 de 08 de 2008, de www.unife.edu.pe/ingenieria/desarrollo.doc

21. Urrizola, S. (24 de 10 de 2008). Hibernate cache. Recuperado el 15 de 08 de 2008, de http://www.lifia.info.unlp.edu.ar/es/difusion/20071024.htm

22. Viscosu, G. (s.f.). www.db4o.com. Recuperado el 03 de 08 de 2008, de www.db4o.com/espanol/db4o%20Whitepaper%20-%20Bases%20de%20Objetos.pdf

23. Wapedia. (s.f.). wappedia. Recuperado el 05 de 07 de 2008, de http://wapedia.mobi/es/IBATIS#1