Marco Teórico

CAPÍTULO 2

CAPÍTULO 2 – Marco Teórico

El concepto fundamental en la mayoría de las aplicaciones que manejan información es la persistencia de los datos, ya que normalmente es almacenada en bases de datos relacionales debido a que la tecnología relacional es conocida por su calidad. El modelado se presenta en términos de tablas, columnas y filas; el lenguaje estándar para la interacción es SQL, no mostrando un explicito modelo de la realidad [Iverson 2004]. La popularidad de las bases de datos relacionales no es accidental, sino tienen un acercamiento flexible y robusto increíble para el manejo de datos [Bauer/King 2004]. Sin embargo los lenguajes de programación tienden a ser orientados a objetos debido a la similitud en el modelado de los cuerpos a como los percibimos de la realidad, las características y beneficios que ofrece cada modelo son de gran peso por lo que inevitablemente hacen que hoy en día se siga trabajando con esta incompatibilidad en los modelos. Cuando el programa requiere guardar algún objeto, este último tiene que ser descompuesto y almacenado. El costo de esta incompatibilidad se ve reflejado en que aproximadamente el 30% de la aplicación es dedicada a el puente entre los modelos [Bauer/King 2004].

2.1 ORM

El mapeo objeto/relacional (ORM) es la automatización y manejo transparente de la persistencia de objetos en una aplicación en Java a las tablas en una base de datos relacional. Un ORM en esencia se encarga de transformar los datos de una representación a otra [Bauer/King 2004]. En otras palabras el ORM realiza el marco de trabajo de la persistencia debido a que sabe como consultar la base de datos para recuperar los objetos Java, independientemente de saber como persistir los objetos en la representación de tuplas en la base de datos. El mapeo de los meta datos típicamente es en archivos XML. Hoy en día existen varias alternativas de herramientas de ORM disponibles, entre ellas encontramos *open source* y comerciales. Entre las más populares herramientas *open source* encontramos

principalmente a Hibernate del grupo JBoss, Torque y OJB del Apache DB Project y en el caso de las comerciales principalmente está disponible TopLink de Oracle.

Logo	Producto	Dirección	Pertenece	Licencia
	Hibernate	http://hibernate.org/	Grupo JBoss	Open source
	Torque	http://db.apache.org/torque/	Apache DB Project	Open source
	ObJectRelationalBridge – OJB	http://db.apache.org/ojb/	Apache DB Project	Open source
	Cayenne	http://www.objectstyle.org/cayenne/	ObjectStyle Group	Open source
	Castor	http://castor.codehaus.org/	ExoLab Group, Intalio Inc.	Open source
	TopLink	http://www.oracle.com/technology/ products/ias/toplink/	Oracle	Comercial
	JDX	http://softwaretree.com/	Software Tree	Comercial
	Cocobase	http://www.thoughtinc.com/ cber_index.html	THOUGHT Inc	Comercial

Tabla 2.1. ORMs actualmente

2.2 Hibernate

Hibernate es una herramienta *open source*, que realiza el mapeo entre las aplicaciones orientadas a objetos y la entidad-relación de las bases de datos en entorno Java; mantiene características fundamentales como: asociación, herencia, polimorfismo, composición y colecciones [Hibernate, 2006]; además aprovecha la madurez y estandarización de las bases de datos relacionales.

En la actualidad *Hibernate* ha tenido gran demanda por su reciente incorporación al *JBoss Group*, el respaldo que da este grupo de desarrolladores motiva a pensar que *Hibernate* será competitivo en el área de base de datos.

A continuación se mencionan algunos beneficios de los ORMs como *Hibernate*. Bauer y King definen los siguientes cuatro: [Bauer/King 2004]

Productividad. El código relacionado con la persistencia resulta tedioso para las aplicaciones de Java. *Hibernate* elimina la mayor parte de este código y permite concentrarse principalmente en el problema, sin importar la estrategia de programación que se este utilizando *top-down* (empezando del modelo global del sistema) o *bottom-up* (empezando con un esquema de base de datos existente).

Capacidad de mantenimiento (*Maintainability*). La cantidad de líneas de código (LOC) son un punto clave para hacer un sistema más comprensible; delegando las tareas de persistencia a *Hibernate* se reduce este número, debido a que el énfasis se hace en la lógica del programa no en la persistencia, haciendo más fácil su mantenimiento. Como se dijo automatizando la persistencia de los objetos se reduce sustancialmente las LOC independientemente de reducir la complejidad de la aplicación debido a la separación del modelo relacional del orientado a objetos y de esta manera facilitando el empleo más elegante a la programación orientada a objetos de Java.

Rendimiento. La persistencia implementada directamente por el programador puede realizar las mismas tareas con un desempeño muy similar a la persistencia automatizada, pero existen grandes diferencias entre éstas cuando se consideran algunas restricciones como el

tiempo de desarrollo de la aplicación que se traduce en dinero. En las aplicaciones que manejan datos persistentes es necesario realizar optimizaciones para mejorar el desempeño del sistema, es posible realizar varias optimizaciones de fácil implementación con código escrito directamente por el programador (por ejemplo usando SQL/JDBC), sin embargo con el simple hecho de usar un ORM se están implementando una gran cantidad de optimizaciones tanto sencillas como complicadas debido a que el equipo de *Hibernate* ha dedicado suficiente tiempo a la investigación de nuevas y mejores técnicas de optimización. Cuando en un proyecto se está restringido por el tiempo, la persistencia codificada directamente por el programador solamente permite realizar algunas optimizaciones y únicamente bajo ciertas condiciones; *Hibernate* nos provee de muchas más optimizaciones que son usadas todo el tiempo aumentando la productividad del desarrollador.

Independencia de proveedor. Una gran ventaja de los ORM es que cuentan con herramientas de soporte para la mayoría de los manejadores de base de datos y esto añade un alto nivel de portabilidad a la aplicación desarrollada, independientemente de facilitar su desarrollo.

Para clarificar el papel que *Hibernate* juega en las aplicaciones de Java, véase Figura 2.2 en la cual se muestra el acceso a la base de datos vía JDBC de la mayoría de aplicaciones de Java (ruta 1 del diagrama), en el caso de Hibernate los desarrolladores se liberan de realizar la escritura de código encargada de la integración del JDBC y enfocan su atención en escribir la presentación y la lógica de negocios de la aplicación (ruta 2) [Iverson 2004]

En la ruta 1 se observa que la capa de persistencia está directamente conectado al JDBC mediante sentencias SQL, por el contrario la ruta 2 muestra la independencia de la aplicación mostrando a *Hibernate* como el puente entre la aplicación y la base de datos encargándose de las sentencias SQL a través del JDBC hasta la creación, modificación y eliminación de objetos persistentes.

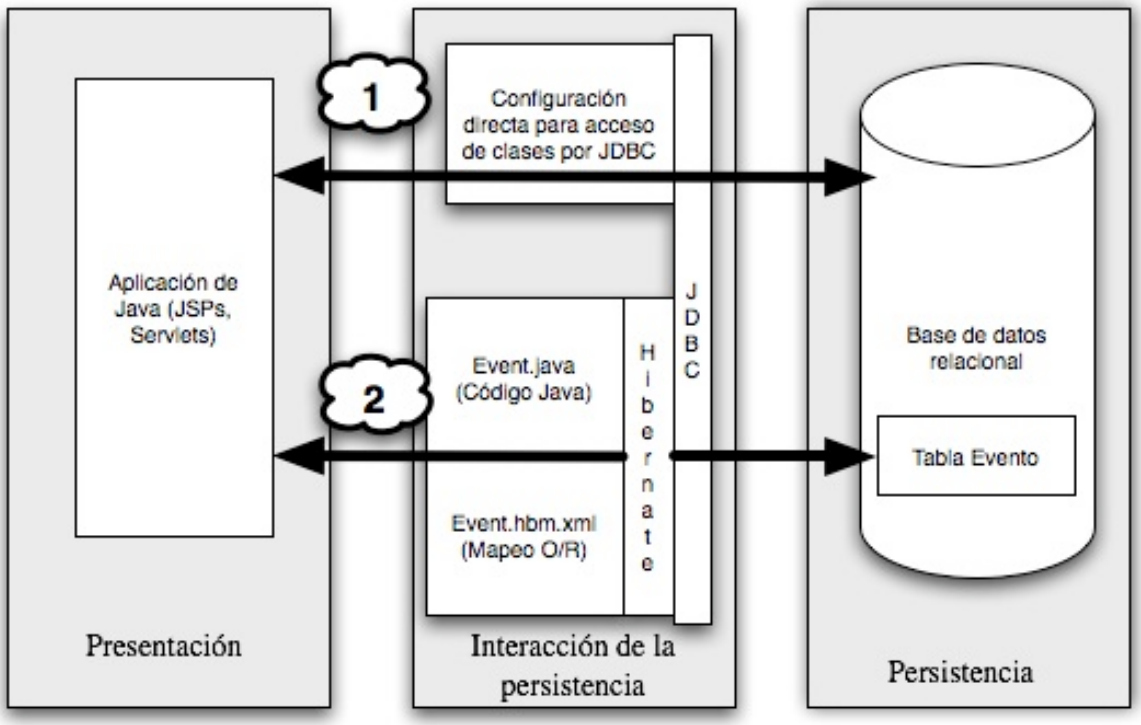

Figura 2.2 Esquema conceptual del papel de Hibernate en una aplicación.
(Adaptado de [Iverson 2004])

De esta forma los desarrolladores no necesitan conocer nada acerca del esquema de la base de datos, separando de manera clara y definida, la lógica de negocios de la aplicación del diseño de la base de datos.

2.2.1 Arquitectura de Hibernate

La visión de la arquitectura de muy alto nivel de *Hibernate* Figura 2.3 muestra los dos requerimientos indispensables para el mapeo entre los dos modelos.

El primer requerimiento son las propiedades (ilustrado como *hibernate.properties*) detallado más adelante. Las propiedades se refieren al archivo de configuración que contiene información sobre que base de datos se desea usar, el *driver* a utilizar para la comunicación, el

puerto, el usuario y contraseña; es decir la información necesaria para la conexión con la base de datos.

El segundo requerimiento es el *XML Mapping,* se refiere a los archivos de mapeo entre los *beans* de java hacia las tablas de la base de datos. Aquí se especifica el tipo de datos que se guardarán en la base de datos; hacia que tabla se dirigirá dicho *bean,* es decir la información necesaria para que *Hibernate* pueda interactuar con la base de datos.

Teniendo claro los dos elementos necesarios para la comunicación de Java con la base de datos, se continúa con un enfoque más profundo de la arquitectura.

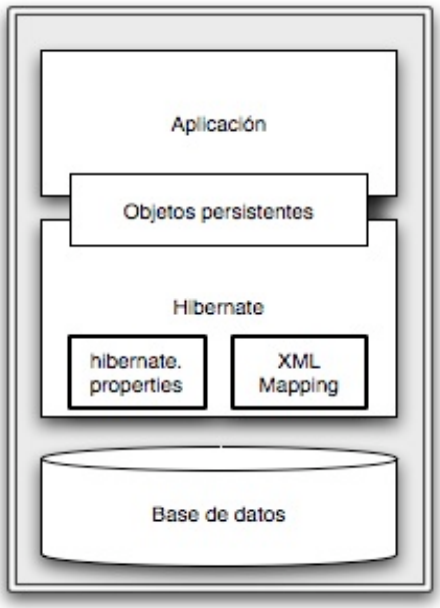

Figura 2.3 "Vista general de *Hibernate*"
(Adaptado de [Hibernate, 2006])

La figura 2.4 Ilustra los roles más importantes de las interfaces de *Hibernate* en la capa de negocios y de persistencia. Se muestra la capa de negocios sobre la capa de persistencia, debido a que la capa de negocios actúa como cliente de la capa de persistencia.

Las interfaces de *Session*, *Transaction* y *Query* son llamadas por la aplicación para realizar básicamente operaciones de CRUD (sus siglas en inglés de: create, read, update and delete) y ejecución de queries. Específicamente *Session* es la principal interfase entre Java e *Hibernate* ya que mantiene la comunicación entre la base de datos y la aplicación. *Transaction* permite a la aplicación definir unidades de trabajo y es asociada con una *Session* en particular. Por otro lado *Query* es una instancia obtenida por la *Session,* que nos permite ejecutar consultas con facilidad definiendo el número de máximo de resultados a ser devueltos, especificar el primer registro que se desea obtener, entre otros. Estas interfaces son la principal dependencia para el control de la aplicación.

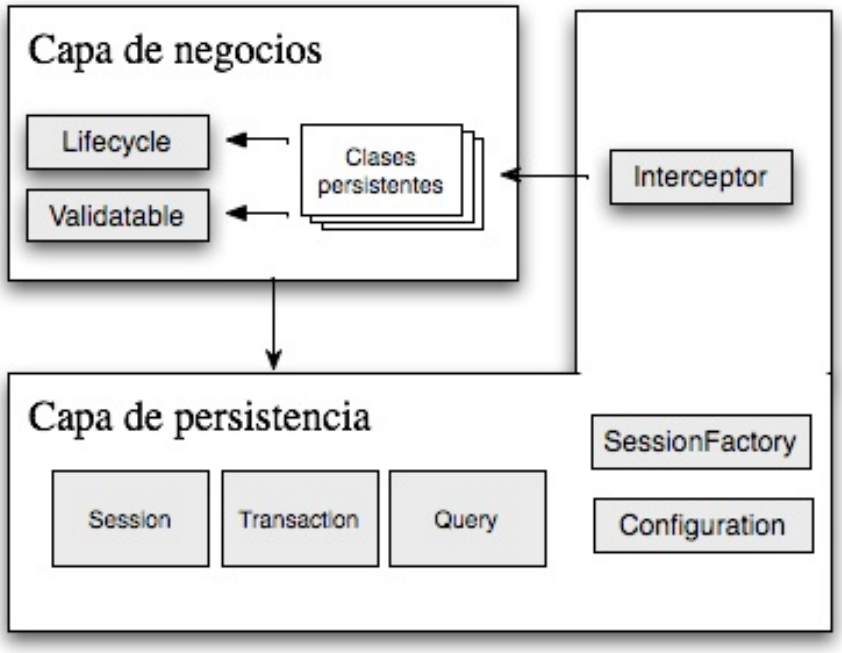

Figura 2.4 Roles en Hibernate
(Adaptado de [Iverson 2004])

Una instancia de *Configuration* permite a la aplicación especificar las propiedades y los documentos de mapeo a ser usados a la hora de crear una *SessionFactory*. Usualmente una aplicación crea una sola *Configuration* de esta manera solamente se crea una instancia de *SessionFactory* para que esta última inicialice la *Session*. Una vez creada la *SessionFactory*

no se puede regresar a la *Configuration*. La instancia de múltiples *Configuration*s es inevitable cuando se usa más de una base de datos en una misma aplicación

Interceptor, Lifecycle y Validatable son interfaces de *callback* de la *Session* para la persistencia de objetos.

2.2.2 Requerimientos para usar Hibernate

Hibernate es de distribución gratuita y *open source.* Disponible para su descarga en http://www.hibernate.org/. Su licencia LGPL es muy flexible y permite su uso tanto para proyectos *open source* como para proyectos comerciales.

Después de obtener una distribución de *Hibernate*, se procede a elegir un DBMS soportado por éste. Las bases de datos oficialmente soportadas por *Hibernate* 3.0 son: [Hibernate, 2006]

- Oracle 8i, 9i, 10g
- DB2 7.1, 7.2, 8.1
- Microsoft SQL Server 2000
- Sybase 12.5 (JConnect 5.5)
- MySQL 3.23, 4.0, 4.1, 5.0
- PostgreSQL 7.1.2, 7.2, 7.3, 7.4, 8.0, 8.1
- TimesTen 5.1
- HypersonicSQL 1.61, 1.7.0, 1.7.2, 1.7.3, 1.8
- SAP DB 7.3

En el contexto de una aplicación web usando Apache Tomcat se deben de incluir los archivos JAR disponibles en la carpeta lib de la distribución de *Hibernate* de la siguiente manera:

En el *classpath* de contexto de la aplicación en Apache Tomcat:


ant-antlr-1.6.3.jar


antlr-2.7.5H3.jar


dom4j-1.6.jar


commons-collections-2.1.1.jar


commons-logging-1.0.4.jar


ehcache-1.1.jar

Ahora de se necesita incluir tanto en el *classpath* de contexto de la aplicación como en el *classpath* global, el JDBC de la base datos que se haya decidido usar, para nuestro caso es:


com.mysql.jdbc.Driver.jar

Después de obtener las distribuciones correspondientes de *Hibernate*, la base de datos, el contenedor de JSPs y *servlets* Apache Tomcat específicamente para nuestro contexto, y haber incluido los archivos a los *classpath* adecuadamente, se puede iniciar el desarrollo de una aplicación.

2.2.3 Configuración de Hibernate

Hibernate por ser la capa de comunicación con la base de datos necesita conocer la información de conexión: qué DBMS se usará, a qué base de datos y cómo se conectará. Por estar diseñado para trabajar en diversos ambientes de desarrollo existen varias maneras de configurar sus propiedades:

1. Se puede hacer por instancia de *Configuration.setProperties*. (java.util.Properties) en la Figura 2.5 se muestra la configuración básica.

2. A través del archivo hibernate.properties ubicado en HIBERNATE_HOME/etc/. Este archivo contiene la configuración por default para la base de datos HypersonicSQL, independientemente de contar con diferentes opciones de configuración para utilizar otros DBMS soportado por *Hibernate.* Solo es necesario completar la información de conexión (como se muestra en la Figura 2.6)

3. Dando las propiedades al sistema usando java –Dproperty=value

4. Incluirlo en la *tag* <property> hibernate.cfg.xml

```
Configuration cfg = new Configuration()
.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
.setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
```

Figura 2.5: Ejemplo de configuración 1
Hibernate. Grupo JBoss
http://www.hibernate.org/hib_docs/v3/reference/en/html/session-configuration.html
Fecha de consulta: 05/12/05

```
## MySQL

hibernate.dialect org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost:3306/prueba
hibernate.connection.username=root
hibernate.connection.password=password
```

Figura 2.6. Ejemplo archivo configuración hibernate.properties para MySQL

2.2.4 Mapeo de Clases

Para que *Hibernate* se encargue del mapeo de las clases en Java es recomendable hacer un diseño de *JavaBean.* En la Figura 2.7 se puede observar un ejemplo con la clase *Event,* esta clase solo cuenta con dos atributos *id* de tipo *long* y *title* de tipo *string*.

Por convención los *JavaBean* contienen dos métodos para cara uno de sus correspondientes atributos, específicamente son los métodos públicos *getter* y *setter*. Todos los atributos de la clase son privados. Cabe destacar que no es obligatorio este diseño ya que *Hibernate* puede acceder a los métodos de la clase *private*, *public* o *protected* pero la práctica de este diseño le da al sistema robustez para *refactoring*. El constructor vacío es necesario para las clases persistentes debido a que *Hibernate* será el encargado de la creación de los objetos.

```
public class Event {
    private Long id;
    private String title;

    public Event() {}

    public Long getId() { … }

    private void setId(Long id) { … }
… }
```

Figura 2.7 Ejemplo JavaBean

Para que *Hibernate* pueda conocer como debe de almacenar los objetos de las clases persistentes, se debe crear para cada clase persistente un archivo de mapeo que por convención lleva el sufijo hbm.xml (Ver Figura7). La estructura básica del archivo está dada por el DTD proporcionado por *Hibenate*. Dentro de las *tags* de hibernate-mapping se debe

incluir el elemento clase, se debe de proporcionar el nombre de la clase y la tabla a la que se quiere asociar en el DBMS. Posteriormente se debe definir la *primary key* (identificador único de la instancia de la clase persistente) y la estrategia de la generación de los valores, *Hibernate* proporciona diez[1] estrategias para la generación dependiendo del tipo de dato y el DBMS usado, para delegar la generación al DBMS (independientemente del que se este utilizando). Para el mapeo de los atributos de las clases persistentes se definen dentro de la *tag* de *property,* el elemento indispensable es el de *name* se iguala al atributo de la clase a mapear y debe de iniciar con letra minúscula (esto se debe de tener en cuenta desde el desarrollo de la clase en Java). Hay elementos que pueden ser configurados opcionalmente como: *column* al no ser asignado su default es el *name.* El *type* puede ser asignado automáticamente por *Hibernate*[2] aunque para tipos de datos muy específicos es recomendable especificarlo manualmente.

En la Figura 2.8 se observa un ejemplo básico del mapeo de una clase de Java a partir del *bean Event*. La documentación completa sobre todos los atributos que se pueden especificar en el archivo de *mapping* XML se encuentran disponibles en el sitio oficial de *Hibernate* mencionado en el capítulo 2.2.2, algunos elementos del *mapping* no son lo suficientemente claros; en estos casos se puede recurrir a la consulta de la gran cantidad de fotos que se encuentran disponibles en Internet sobre este ORM. Existen numerosos libros que profundizan en el eso de la herramienta que pueden ser de gran utilidad para el desarrollo de una aplicación con este *framework* . Continuando con el *mapping* del *bean Event* se puede describir de la siguiente manera:

El elemento `<class>` especifica la clase de Java (bean) de la que se está haciendo el *mapping* y la tabla de la base de datos a relacionar en el mapeo, en este ejemplo la clase se llama *Event*.java y el nombre de la tabla en la base de datos se establece como: EVENTS. (no son necesarias las mayúsculas).

[1] Las estrategias de generación de *primary key* son incerment, identity, sequence, hilo, seqhilo, uuid.hex, uuid.string, native, assigned, foreing.
[2] Ejemplos de tipos pueden ser integer, string, character, date, timestamp, float, binary, serializable, object, blob.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-
mapping-3.0.dtd">

<hibernate-mapping>

    <class name="Event" table="EVENTS">

<id name="id" type="integer" column="ID">
            <generator class="native"/>
        </id>

<property name="title" type "string" unique="true" not-
null="true"/>
    </class>

</hibernate-mapping>
```

Figura 2.8 Ejemplo de Archivo.hbm.xml

El elemento <id> relaciona la llave primaria. Opcionalmente se puede especificar el tipo (*type*) de dato con el que se trabaja, en este caso con enteros; en caso de no estar especificado Hibernate lo podría deducir

El sub-elemento <generator> permite definir cómo se generarán los valores de las llaves primarias. Hibernate ofrece diversos métodos predefinidos para la generación como ya se había mencionado. El método que se muestra en el ejemplo *native* permite que Hibernate elija entre: *identity*, *sequence* o hilo en función de las características del DBMS [Hibernate, 2006].

El elemento <property> define las propiedades que no forman parte de la llave primaria, el atributo opcional *unique* permite definir si se permitirán valores duplicados o no, así como el atributo *not-null* permite especificar si la propiedad puede tener el valor de *null.*

El ejemplo anterior es bastante sencillo, existen mapeos más complicados en el caso de haber dependencias entre objetos, por ejemplo:

Considerando tener dos atributos más en la clase *Event: childEvents* (lista de *Events* hijos) y *fatherEvent* (padre de este *Event)*

```
Class Event {
…
Event fatherEvent;
List childEvents;
…
}
```

Figura 2.9 Ejemplo Extensión de clase Event

En este caso se deben de considerar:

El elemento `<many-to-one name= "fatherEvent" column= "ID">` se describe una relación de muchos a uno entre la misma clase; es decir el objeto *fatherEvent* puede tener muchos hijos, mientras que el *event* solo puede tener un padre. Se le tiene que especificar el campo de la tabla que contiene la llave primaria para ser relacionados.

El elemento:

```
<bag name="childEvents" table="Event" cascade="all">
     <key column= "ID" />
     <many-to-many class= "Event" column= "ID"/>
    </bag>
```

Se describe la relación muestra; la relación de *Event* con *childEvent*; el atributo de *cascade all* que se refiere a que realizará operaciones en cascada, es decir si se borra el objeto padre los eventos hijos serán borrados también.

2.2.5 Aplicación Hibernate

Después de tener el mapeo de .java a .hbm.xml un esquema para el uso de *Hibernate* en las aplicaciones puede ser:

- Definir las tres propiedades básicas:

```
Configuration config = null;
SessionFactory sessionFactory = null;
Session session = null;
```

- Métodos de acceso de acceso a las propiedades

  - Carga la información de la BD:

```
config = new Configuration();
```

  - Lee archivo clase.hbm.xml:

```
config.addClass(clase.class);
```

  - Construye la *SessionFactory*:

```
sessionFactory = getConfiguration().buildFactory();
```

  - Construye la *Session*:

```
Session = getSessionFactory().openSession();
```

Después de tener el acceso a las propiedades y configuración, se puede empezar a insertar, borrar, actualizar y consultar.

Insertar.

Para hacer persistente un objeto se crea una instancia que en este caso se trata de *Event* (1), se crea la *session* (2), se inicia la unidad de trabajo (3), se guarda el objeto *event* (4) cabe destacar que no se utiliza en ningún momento SQL, se confirma si la transacción se realizo de manera exitosa (5) (en caso de que no haber sido exitosa se debe considerar hacer un *rollback* en este momento), se cierra la *session* (6)

```
Event event = new Event();  // (1)
Session session = getSessionFactory().openSession();  //
(2)
Transaction tx = session.beginTransaction();  // (3)
session.save(event);  // (4)
tx.commit();  // (5)
session.close();  // (6)
```

Figura 2.10 Ejemplo save

Borrar

El esquema general para borrar un objeto de la base de datos es básicamente el mismo que el de insertar, a la instancia del objeto *Event* se le asigna el parámetro *id* del objeto que se desea eliminar de la base de datos y este objeto se le manda al método *delete()*

```
Event event = eventoBorrar;
Transaction tx = session.beginTransaction();
session.delete(event);
tx.commit();
```

Figura 2.11 Ejemplo delete

Se puede borrar un objeto a través de la llave primaria con la ayuda del método *load*() de *Session*, este método nos devuelve un objeto a partir del id (llave primaria), se tiene que especificar de que tipo es el objeto para que *Hibernate* sepa en que tabla buscar.

Figura 2.12 Ejemplo delete

Hibernate internamente creará el SQL necesario para recuperar el objeto por ejemplo:

```
SELECT * FROM EVENT WHERE ID = miID
```

Figura 2.13 Ejemplo SQL generado por Hibernate

En caso de que se necesite borrar más de un objeto, *Hibenate* dispone de un lenguaje de consulta orientado a objetos *Hibernate Query Languege* (HQL). En el siguiente ejemplo se muestra un ejemplo para borrar solo un registro

```
String sel = "FROM Event AS e WHERE e.id = " + miID;
session.delete(sel);
```

Figura 2.14 Ejemplo Delete con HQL de un registro

Un ejemplo para borrar más de uno es el siguiente:

```
String sel = "FROM Event AS e WHERE e.name LIKE = %r%";
session.delete(sel);
```

Figura 2.15 Ejemplo Delete con HQL de varios registros

En el caso de querer hacer actualizaciones se puede hacer como se muestra:

```
Event event = (Event)session.load(Event.class, new
Integer(id));
event.setName("nombreNuevo");
session.update(event);
```

Figura 2.16 Ejemplo update

Para recuperar objetos se utiliza el *Hibernate query* expresado en HQL. Como se muestra a continuación se pueden recuperar los resultados ordenados por algún criterio, en este caso ascendente.

```
List childEvents =
newSession.find("from Event as e order by e.name
asc");
…
```

Figura 2.17 Ejemplo find

El HQL proporcionado por Hibernate es capaz de soportar cláusulas como por ejemplo: *GROUP BY, COUNT, AND*, ?, %, *AVG, SUM, MAX, MIN*, ente otras.

Hibernate permite habilitar individualmente una caché para cada entidad. Independiente de poder habilitar una caché de segundo nivel es necesario establecer la

estrategia de concurrencia que *Hibernate* utilizará para sincronizar la caché de primer nivel con la caché de segundo nivel y a su vez la caché de segundo nivel con la base de datos. Hay cuatro estrategias de concurrencia predefinida (se configuran en el archivo de mapeo):

transactional: Es el nivel más estricto. Esta estrategia sólo se puede utilizar en *clusters*, *(*con caché distribuida*)*

read-write: Mantiene un aislamiento hasta el nivel de *commited*, utilizando un sistema de marcas de tiempo (*timestamps*). Esta estrategia no se puede utilizar en clusters.

nonstrict read-write: No ofrece ninguna garantía de consistencia entre la caché y la base de datos. Para sincronizar los objetos de la caché con la base de datos se utilizan time-outs, cuando termina el timeout se recargan los datos. Con esta estrategia se tiene un intervalo en el cual se tiene riesgo de obtener objetos desfasados. Cuando *Hibernate* realiza una operación de *flush*() en una sesión, se invalidan los objetos de la caché de segundo nivel. Aún así, esta es una operación asíncrona y no tenemos garantía de que otro usuario no pueda leer datos erróneos. Ideal para almacenar datos que no sean críticos.

read-only: Es la estrategia de concurrencia menos estricta. Ideal para datos que nunca cambian.

2.3 Indexación

Según la mayoría de estudios que se han estado realizando en los últimos años la recuperación y organización de la información es uno de los aspectos que han cobrado mayor relevancia, por lo cual ha motivado la aparición de sistemas que clasifiquen automáticamente documentos basándose en el análisis de contenidos procesados principalmente en criterios estadísticos, centrados en la frecuencia de aparición de términos en los documentos [Araujo, 2003]. Se puede pensar en extraer todos los términos de cada documento e indexarlos (a manera de datos estructurados) para obtener una lista invertida que permita ir con los términos hacia los documentos que los contienen, a esta actividad se le conoce como indexamiento a texto completo [Proal, 2003].

La indexación es un conjunto de técnicas que permiten realizar búsquedas en un grupo de documentos de forma rápida, para ello se utilizan estructuras de datos llamadas índices. Un índice organiza la información de los documentos de forma que, dada una palabra o una consulta (conjunto de palabras), sea posible determinar en qué documentos aparecen [Rijsbergen, 1979].

2.3.1 Lucene

Es una herramienta de recuperación de la información (RI) *open source*, del proyecto Jakarta de Apache desarrollado en Java, sigue un modelo de recuperación vectorial que permite tanto la indexación como la búsqueda de documentos [Paul, 2004].

El modelo de recuperación vectorial también llamado modelo de espacios vectoriales; fue propuesto en 1975, hoy en día este modelo tiene variantes y es el más popular. La idea principal es que dos documentos son similares si sus vectores apuntan hacia la misma dirección. Un aspecto importante es la determinación del peso de cada término, típicamente basado en su frecuencia: el peso aumenta entre más ocurrencias hay en el documento dado y por el contrario el peso disminuye entre más aparece en los demás documentos [Proal, 2005]

Las principales características de Lucene son: [Araujo, 2003]

Indexación incremental vs. Indexación por lotes. El término de indexación por lotes se utiliza para referirse a procesos en los cuales se ha creado el índice para un conjunto de documentos, el añadir documentos es tan complicado que se opta por re-indexar todos los documentos nuevamente. En la indexación incremental se pueden añadir fácilmente documentos a un índice ya creado con anterioridad.

Contenido Etiquetado. Algunas herramientas, tratan los documentos como simples flujos de palabras. Lucene permite dividir el contenido de los documentos en campos y así poder realizar consultas con un mayor contenido semántico. Esto es, se pueden buscar términos en

los distintos campos del documento concediéndole más importancia según el campo en el que aparezca.

Concurrencia. Es capaz de gestionar que varios usuarios puedan buscar en el índice de forma simultánea, así como también que un usuario este modificando el índice mientras otro usuario lo consulta.

Lematización. Eliminación de palabras comunes en el lenguaje como artículos, conjunciones, preposiciones, etc. [Proal, 2003]. Para lograr esto Lucene trabaja con una lista de *stop words*, las cuales son proporcionadas por el desarrollador, permitiendo que estas palabras coincidan con el idioma en el que se van a introducir los documentos, al coincidir con el idioma éstas puedan ser eliminadas del índice.