

4

Herramientas de mapeo objeto-relacional

OBJETIVOS DEL CAPÍTULO

- ✓ Instalar y configurar una herramienta ORM.
- ✓ Definir ficheros de mapeo.
- ✓ Aplicar mecanismos de persistencia a los objetos.
- ✓ Desarrollar aplicaciones que modifican y recuperan objetos persistentes.
- ✓ Desarrollar aplicaciones que realizan consultas usando el lenguaje HQL.
- ✓ Gestionar las transacciones.

En el capítulo anterior se han descrito las ventajas de utilizar el mismo modelo orientado a objetos a la hora de programar (POO) y a la hora de almacenar los datos (SGBD-OO). El trabajo conjunto de POO y SGBD-OO simplifica enormemente el desarrollo de software ya que, en teoría, con ellos no es necesaria una conversión entre el modelo de POO (por ejemplo con Java) y el modelo de base de datos (por ejemplo, Matisse).

Pese a las ventajas de la homogeneidad entre modelos, no siempre es posible trabajar con sistemas OO, o bien por razones de rendimiento, o bien por estar creada la estructura de las bases de datos antes que el software que la gestiona, o bien por otras razones. En el desarrollo de aplicaciones es muy común utilizar lenguajes OO (Java, C#, etc.) y almacenar los datos en sistemas relacionales (Oracle, MySQL, etc.).

En estos casos es necesario convertir objetos, con los que se trabaja a nivel de programación, a un modelo relacional. Más concretamente, es obligada una conversión o mapeo de los atributos de los objetos a las tablas y datos del modelo relacional.

En este capítulo se trata este proceso de mapeo, mostrando las alternativas más comunes en concordancia con las herramientas utilizadas en el resto del libro.

4.1 CONCEPTO DE MAPEO OBJETO-RELACIONAL (*OBJECT-RELATIONAL MAPPING [ORM]*)

El mapeo objeto-relacional (más conocido por su nombre en inglés, *Object-Relational Mapping*, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación que permite convertir datos entre el sistema de tipos utilizado en un lenguaje de programación, normalmente orientado a objetos, y el utilizado en una base de datos relacional. En la práctica este mapeo crea una base de datos orientada a objetos virtual sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos, por ejemplo la herencia y el polimorfismo.⁵⁵

Actualmente, las bases de datos relacionales (no las objeto-relacionales) solo pueden guardar datos primitivos, por lo que no se pueden guardar objetos de la aplicación directamente en la base de datos. Lo que se hace con el mapeo ORM es convertir los datos del objeto en datos primitivos que sí se pueden almacenar en las tablas correspondientes de una base de datos relacional. Cuando se desee recuperar los datos del sistema relacional, lo que se hace es obtener los datos primitivos de la base de datos y volver a construir el objeto. *El objetivo del mapeo ORM es ofrecer un proceso transparente de conversión de datos relativionales a objetos y viceversa.*

Entre las ventajas principales del mapeo objeto-relacional se pueden mencionar las siguientes:

- ✓ Rapidez en el desarrollo. La mayoría de las herramientas actuales permiten la creación del modelo por medio del esquema de la base de datos, leyendo el esquema se puede crear el modelo adecuado.
- ✓ Abstracción de la base de datos. Al utilizar un sistema ORM, lo que se consigue es abstraer la aplicación del sistema gestor de base de datos que se utilice. Por tanto, si en el futuro se cambia de sistema gestor, el cambio no debería afectar al código del programa desarrollado.

⁵⁵ Se puede entender que en ORM se simula un sistema OO como el mostrado en el capítulo anterior.

- ✓ Reutilización. Al utilizar un sistema ORM se pueden utilizar los métodos de un objeto de datos desde distintas zonas de la aplicación, incluso desde aplicaciones distintas.
- ✓ Mantenimiento del código. El mapeo facilita el mantenimiento del código debido a la correcta ordenación de la capa de datos, haciendo que el mantenimiento sea mucho más sencillo.
- ✓ Lenguaje propio para realizar las consultas. Los mecanismos de mapeo ofrecen su propio lenguaje para hacer las consultas, lo que hace que los usuarios dejen de utilizar las sentencias SQL para que pasen a utilizar el lenguaje propio de cada herramienta.⁵⁶

Por contra, entre las desventajas del mapeo objeto-relacional destacan:

- ✓ El tiempo utilizado en el aprendizaje. Este tipo de herramientas suelen ser complejas, por lo que su correcta utilización lleva un tiempo no despreciable.
- ✓ Aplicaciones algo más lentas. Esto es debido a que sobre todas las consultas que se hagan sobre la base de datos, el sistema primero deberá transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

En la actualidad hay muchos tipos de marcos de trabajo (también denominados *frameworks*) que permiten el mapeo objeto-relacional, atendiendo al lenguaje que se esté utilizando. Algunos de los más utilizados son *Doctrine*, *Propel*, *Hibernate* y *LINKQ*.

4.2 CARACTERÍSTICAS DE LAS HERRAMIENTAS ORM. HERRAMIENTAS ORM MÁS UTILIZADAS

Como se ha comentado en el Capítulo 3, al trabajar con programación orientada a objetos y bases de datos relacionales se utilizan paradigmas y formas de pensar distintas. El modelo relacional trata con relaciones y conjuntos de datos. El paradigma OO trata con clases de objetos, objetos, atributos, métodos y asociaciones entre objetos. Un mapeo objeto-relacional (ORM) tiene como misión evitar estas diferencias.

Las diferencias entre modelos se manifiestan de la siguiente manera: si se tienen objetos en una aplicación orientada a objetos y se desea que estos sean persistentes, normalmente se abrirá una conexión JDBC,⁵⁷ se creará una sentencia SQL (INSERT INTO) y se copiarán todos los valores de los atributos de los objetos en la base de datos relacional. Esto podría ser fácil para un objeto pequeño, sin embargo se complica cuando el objeto tiene un número elevado de atributos. Además, para más dificultad, los objetos no están aislados, sino que están relacionados entre sí. De esta manera, por ejemplo, si una clase *álbum de música* tiene varias canciones asociadas, también deben ser “insertadas” cada una de las canciones, complicando todavía más el proceso de conversión. Lo mismo se puede aplicar para el proceso inverso. Se haría una recuperación de los datos de la base de datos con una conexión JDBC y sentencias SQL (SELECT) y se asociarían los datos recuperados con atributos de objetos.

⁵⁶ Como se ha comentado en capítulos anteriores, esta ventaja también puede ser vista como un inconveniente al obligar al programador a conocer más de un lenguaje.

⁵⁷ Como las mostradas en el Capítulo 2.

Con una herramienta ORM el proceso es mucho más sencillo. Teóricamente, a partir de los objetos Java, por ejemplo *album1* se puede hacer la persistencia en un sistema relacional ejecutando: *orm.save(album1)*. Esta sentencia generará automáticamente todo el SQL necesario para almacenar el objeto. El proceso inverso es igual de sencillo. Por ejemplo, si se desea crear un objeto *album1* de tipo *Album* a partir del almacenado en el sistema relacional con un *id* de objeto determinado, la sentencia del ORM sería: *album1=orm.load(album1.class, objectId)*;

Como se ha comentado anteriormente, un ORM suele aportar un lenguaje de consulta propio para recuperar los datos de la base de datos relacional como objetos. El siguiente ejemplo recupera una lista de objetos *Album* cuyo *título* es *Black*.

```
List lAlbum = orm.find("FROM Album object WHERE object.titulo like "Black");
```

Esta sentencia traduce automáticamente la consulta a SQL y recupera los datos como si fueran objetos de tipo *Album*, también de manera transparente para el programador.

4.3 INSTALACIÓN Y CONFIGURACIÓN DE UNA HERRAMIENTA ORM

En esta sección, para tener una herramienta de referencia para abordar el resto del capítulo, se presenta Hibernate como ORM que permite el mapeo objeto-relacional en Java. Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL, y es ampliamente utilizado en desarrollos profesionales. Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado y para poder adaptarse a su uso sobre una base de datos ya existente. Esta herramienta ofrece también un lenguaje de consulta de datos llamado HQL (*Hibernate Query Language*).

4.3.1 INSTALACIÓN MANUAL

Para instalar Hibernate (versión 4)⁵⁸ y poder utilizarlo, por ejemplo, en el IDE Netbeans 7.1.2 (o inferior) se pueden seguir los siguientes pasos:

- 1** Acceder al sitio de Hibernate.⁵⁹ Navegar a Downloads->Hibernate->JBoss Community.
- 2** Descargar la última versión. En este caso se utilizará la versión 4.1.9.
- 3** Una vez descargado el fichero *hibernate-release-4.1.9.Final.zip* se descomprime. En el fichero se encuentran 3 carpetas:

- *lib*: contiene las librerías (*jars*) Java con el código de Hibernate. Las más destacables son:
 - *lib\required*: contiene los *jars* que siempre deben usarse en Hibernate. Es decir, siempre debemos incluir estos ficheros *jars* en todos los proyectos que usen Hibernate.

⁵⁸ Toda la información sobre Hibernate 4 puede ser consultada en <http://docs.jboss.org/hibernate/orm/4.1/javadocs/>

⁵⁹ <http://www.hibernate.org/downloads>

- *lib\jpa*: las librerías necesarias para usar JPA con Hibernate.
 - *lib\optional*: contiene las librerías que añaden nuevas funcionalidades a Hibernate, como poder usar el *pool* de conexiones *C3PO* o el sistema de caché *EhCache*.
 - *lib\envers*: contiene librerías que permiten realizar auditorías sobre los datos que se hacen persistentes.
- *documentation*: contiene la documentación sobre Hibernate. De entre la documentación destaca:
- *documentation/devguide/en-US/html_single/index.html*: Guía del desarrollador. *Hibernate Developer Guide*.
 - *documentation/quickstart/en-US/html_single/index.html*: Guía de inicio. *Hibernate Getting Started Guide*.
 - *documentation/javadocs/index.html*: JavaDoc de las clases Java. *Hibernate JavaDoc (4.1.9.Final)*.
- *project*: contiene principalmente el código fuente y ficheros de configuración de las distintas bases de datos.

4 En este paso se copian todos los ficheros *jar* que se encuentran en la carpeta *lib\required* en la carpeta *lib* de nuestro proyecto Java.

5 Copiar el fichero *hibernate-entitymanager-4.1.9.Final.jar* de la carpeta *lib\jpa* también en la carpeta *lib* de nuestro proyecto Java.

6 En este paso es necesario indicar a NetBeans 7.1.2 que se desea usar todas esas librerías, para ello una opción sería con el botón derecho pulsar sobre el árbol en el nodo **Libraries** y seleccionar la opción de menú **Add Jar/Folder...**⁶⁰

7 Seleccionar todos los ficheros *jar* anteriores (pasos 4 y 5) y pulsar el botón **Abrir**.

Realizados estos pasos se tienen todas las librerías de Hibernate en un proyecto Java.

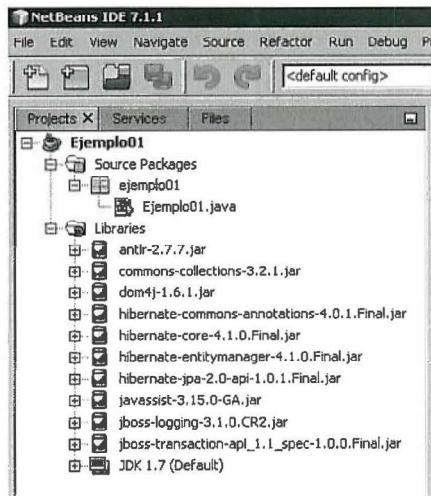


Figura 4.1. Librerías Hibernate en el proyecto Netbeans

60 En la Sección 2.2.4 se muestra este proceso para añadir librerías de JDBC, pero con otros pasos.

4.3.2 USAR NETBEANS CON J2EE

Una solución más sencilla que la anterior es descargarse una versión de IDE NetBeans que contenga ya el paquete Hibernate. En los ejemplos siguientes se usará NetBeans con J2EE 7.2.1,⁶¹ que, entre otras cosas, ya contiene Hibernate 4.

4.4 ESTRUCTURA DE FICHEROS DE HIBERNATE. MAPEO Y CLASES PERSISTENTES

En esta sección se muestran los ficheros más destacados en Hibernate que permiten el mapeo entre los objetos Java y las tablas del sistema gestor relacional. En concreto, Hibernate tiene dos clases importantes:

- Las clases Java, que representan los objetos que tienen correspondencia con las tablas de la base de datos relacional.
- El fichero de mapeo (*.hbm.xml*), que indica el mapeo entre los atributos de una clase y los campos de la tabla relacional con la que está asociado.

4.4.1 CLASES JAVA PARA REPRESENTAR LOS OBJETOS (POJO)

Las clases Java representan objetos en una aplicación que use Hibernate, objetos que se corresponden con la información almacenada en un sistema relacional. Las características de estas clases son:

- ✓ Deben tener un constructor público sin ningún tipo de argumentos.
- ✓ Para cada propiedad que se quiere hacer corresponder con un campo de una tabla relacional debe haber un método *get/set* asociado.
- ✓ Debe implementar la interfaz *Serializable*.

A estas clases Hibernate se refiere como POJO (*Plain Old Java Objects*). Un ejemplo de POJO con las características descritas sería el siguiente.

```
public class Albumes implements java.io.Serializable {  
  
    private int id;  
    private String titulo;  
    private String autor;  
  
    public Albumes() {
```

⁶¹ Esta versión de IDE NetBeans está disponible en <http://netbeans.org/downloads/>

```
}

public int getId() {
    return this.id;
}

public void setId(int id) {
    this.id = id;
}
public String getTitulo() {
    return this.titulo;
}

public void setTitulo(String titulo) {
    this.titulo = titulo;
}
public String getAutor() {
    return this.autor;
}

public void setAutor(String autor) {
    this.autor = autor;
}
```

Esta clase representa un álbum de música. Este álbum tiene como propiedades un identificador (*id*) un título y un autor. Como puede observarse es una clase “normal” en Java que incluye métodos para acceder a los atributos (*set/get*). Su única peculiaridad, exigida por Hibernate, es que implemente la interfaz *java.io.Serializable*.

4.4.2 FICHERO DE MAPEO ".HBM.XML"

Para cada clase que se quiere hacer persistente (por ejemplo, la clase *Albumes* de la sección anterior) se creará un fichero XML con la información que permitirá mapear esa clase a una base de datos relacional. Este fichero estará en el mismo paquete que la clase cuyos objetos se quieren hacer persistentes.

Un ejemplo de fichero *.hbm.xml* para la clase anterior (*Albumes*) y una supuesta tabla relacional (ALBUMES) sería:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 11-ene-2013 17:04:45 by Hibernate Tools 3.2.1.GA -->
```

```
<hibernate-mapping>
<class name="accesohibernate.Albumes" table="ALBUMES" schema="ROOT">
<id name="id" type="int">
<column name="ID" />
<generator class="assigned" />
</id>
<property name="titulo" type="string">
<column name="TITULO" length="30" />
</property>
<property name="autor" type="string">
<column name="AUTOR" length="20" />
</property>
</class>
</hibernate-mapping>
```

En el ejemplo, las etiquetas `<property>` contienen el nombre y tipo de los atributos que se quieren hacer persistentes (de la clase *Albumes*). Por su lado, las etiquetas `<column>` (dentro de `<property>`) contienen el nombre y el tipo del campo de la base de datos en el que se almacenarán los atributos de la clase *Albumes* (en el ejemplo, *titulo* se almacenará en *TITULO*). Por su lado, la etiqueta `<class>` contiene el nombre de la tabla sobre la que se mapea (*ALBUMES*) y el esquema en el que se encuentra (ROOT). En este ejemplo la tabla *ALBUMES* debe estar creada antes de hacer el mapeo. Sin embargo, es posible también que una aplicación la cree automáticamente después con la información de mapeo.

4.4.3 CREAR FICHEROS DE MAPEO CON NETBEANS

Desde la IDE NetBeans 7.2.1 es sencillo crear los ficheros con las clases Java (POJO) y el fichero *.hbm.xml* para el mapeo. Existen muchas alternativas para crear estos ficheros, sin embargo, en esta sección se mostrará una en concreto: partiendo de una base de datos relacional con una tabla (*ALBUMES*) se crea la clase Java asociada y el fichero de mapeo.

Antes de explicar los pasos a seguir es necesario crear el proyecto Java y una conexión a una base de datos.

- Crear con NetBeans un proyecto tipo aplicación Java (Java Application). Para este ejemplo se le ha llamado *accesoHibernate*⁶²
- Crear una base de datos en el propio entorno de IDE NetBeans (JavaDB). Para el ejemplo que se detalla se ha creado una base de datos llamada “discografía” que tiene una tabla *ALBUMES* para el esquema ROOT. Se han creado en *ALBUMES* tres campos: ID, clave primaria tipo numérico, TITULO, tipo VARCHAR(30), y AUTOR, tipo VARCHAR(30). La Figura 4.2 muestra en NetBeans la estructura creada.

62 En el código asociado a este capítulo se puede encontrar el proyecto *accesoHibernate* completo.

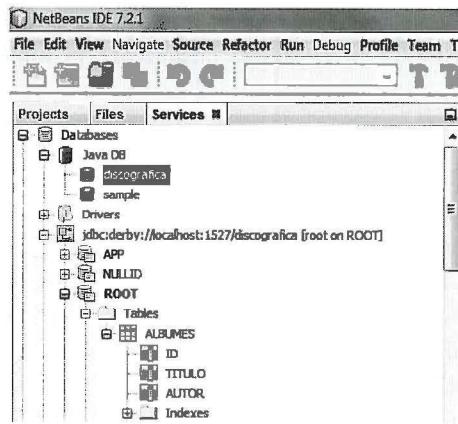


Figura 4.2. Base de datos Java DB

Una vez creados el proyecto y la conexión a la base de datos se generan los ficheros necesarios para el mapeo.

Fichero de configuración (hibernate.cfg)

Al proyecto Java *accesoHibernate* se le añade un nuevo archivo (hibernate.cfg). Este archivo contendrá la información necesaria para conectar a la base de datos sobre la que se hará la persistencia.

Para crearlo se pueden seguir los siguientes pasos: (1) ir al menú *Fichero (File)* -> *Nuevo archivo* y seleccionar dentro de las posibilidades que ofrece (**otros...**) un tipo de archivo de configuración de Hibernate (**hibernate configuration wizard**) que se encuentra dentro de la carpeta *Hibernate*. (2) Se pulsa en **Siguiente**. (3) El nombre por defecto es *hibernate.cfg*. Sin cambiar nada se pulsa de nuevo en **Siguiente**. (4) De la lista desplegable se selecciona la conexión a la base de datos, que ha debido ser creada con anterioridad (aunque también se puede crear en ese momento). (5) Una vez seleccionada se le da a **Terminar**. Se habrá creado en el paquete por defecto del proyecto (*default package*) un fichero *hibernate.cfg.xml* con la configuración para la conexión a la base de datos (dónde encontrarla y el usuario y clave para acceder). La Figura 4.3 muestra cómo quedaría el proyecto con el nuevo fichero.

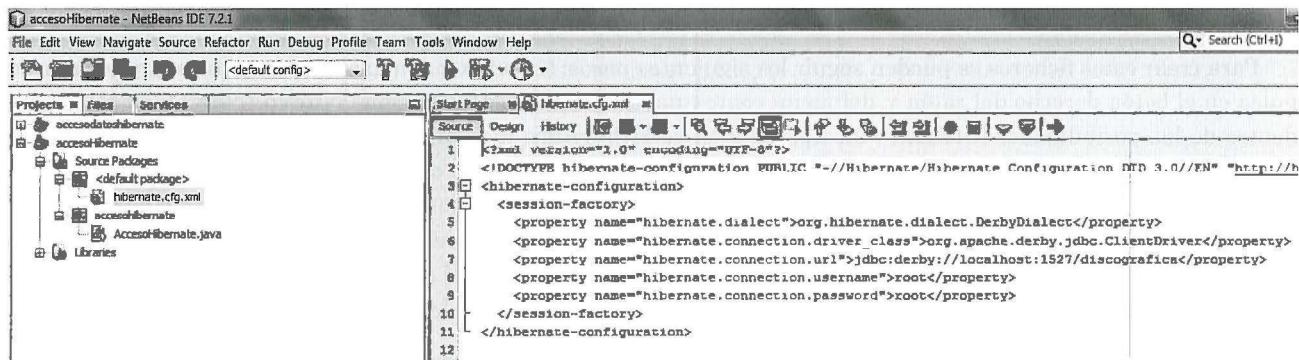


Figura 4.3. Estructura del fichero hibernate.cfg.xml

Fichero de ingeniería inversa (`hibernate.reveng`)

Este archivo indica el esquema (ROOT) en el que se encontrará la tabla a mapear y el nombre de la tabla (ALBUMES).

Para crearlo se siguen los siguientes pasos: (1) seleccionando el paquete **default package** se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y, se selecciona dentro de las posibilidades (**otros...**), un tipo de archivo **Hibernate Reverse Engineering Wizard** dentro de la carpeta **Hibernate**. (2) Se pulsa en **Siguiente**. (3) El nombre por defecto del fichero será *hibernate.revenge*. Sin cambiar nada se pulsa de nuevo en **Siguiente**. (4) Se selecciona la tabla que se quiere mapear (llevándola a la otra lista con **añadir -> Add**). Para este ejemplo es ALBUMES, que fue creada con anterioridad en la base de datos. (5) Una vez seleccionada se le da a **Terminar**. Con estos pasos se habrá creado en el paquete por defecto del proyecto (*default package*) un fichero *hibernate.revenge.xml* con el nombre del esquema (ROOT) y de la tabla que se desea mapear (ALBUMES). La Figura 4.4 muestra cómo quedaría el proyecto con el nuevo fichero.

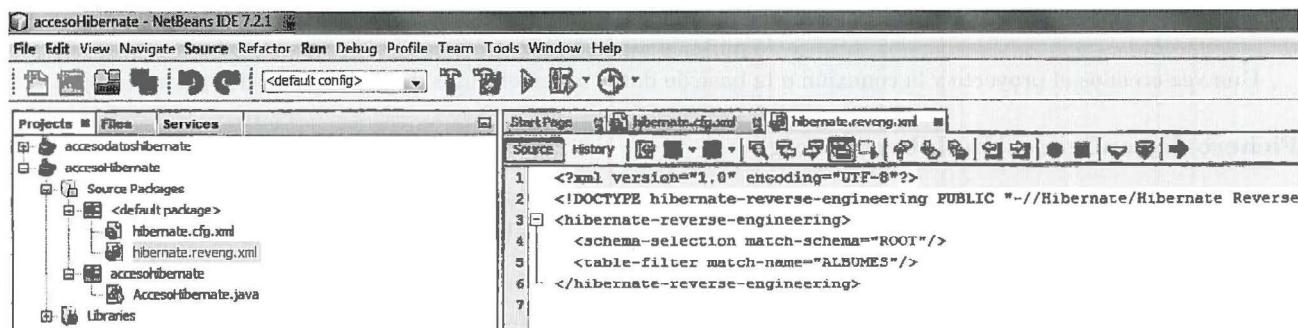


Figura 4.4. Estructura del fichero *hibernate.revenge.xml*

Ficheros POJO y de mapeo

En este paso se creará el fichero POJO, que es la clase Java obtenida a partir de la tabla de la base de datos y el fichero de mapeo (*.hbm.xml*) que contiene las reglas de mapeo para convertir datos de la tabla relacional con los atributos de la clase POJO.

Para crear estos ficheros se pueden seguir los siguientes pasos: (1) seleccionando el paquete **default package** se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y, se selecciona dentro de las posibilidades (**otros...**), un tipo de archivo **Hibernate mapping files and POJO from Database** dentro de la carpeta **Hibernate**. (2) Se pulsa en **Siguiente**. (3) En la ventana que aparece se puede observar como ya salen los ficheros de configuración creados en los pasos anteriores, que son los que necesita Hibernate para crear la clase Java (POJO) a partir de la tabla ALBUMES y el fichero de mapeo entre esa clase y la tabla. Para terminar de configurar esta parte solo es necesario indicar el nombre de un paquete en el que almacenar los nuevos ficheros que se van a crear. En el ejemplo es *accesohibernate*. (4) Una vez seleccionado se le da a **Terminar**.

Con estos pasos se habrán creado en el paquete *accesohibernate* dos ficheros, uno es la clase Java (POJO) obtenida a partir de la tabla ALBUMES (*Albumes.java*) y otro es el fichero de mapeo entre la tabla y la clase (*Albumes.hbm.xml*). Hay que observar que estos ficheros son los mismos que los descritos en las secciones anteriores.

La Figura 4.5 muestra cómo quedaría el proyecto con los dos nuevos ficheros.

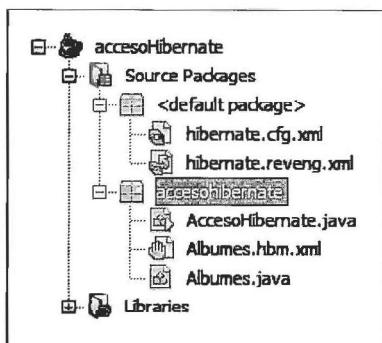


Figura 4.5. Fichero POJO y de mapeo en el proyecto

Si se desea añadir posteriormente nuevos POJO de tablas creadas en la base de datos se puede hacer repitiendo estos pasos para las nuevas tablas.

Fichero HibernateUtil.java

Este fichero se utiliza para gestionar las conexiones que se hacen a las bases de datos y que permitirán mapear los objetos en las tablas correspondientes. Para crear estos ficheros se pueden seguir los siguientes pasos: (1) seleccionando el paquete **accesoHibernate** se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y, se selecciona dentro de las posibilidades (**otros**), un tipo de archivo **HibernateUtil.java**. (2) Se pulsa en **Siguiente**. (3) Se cambia el nombre por defecto (*NewHibernateUtil*) por *HibernateUtil* y también se selecciona un paquete **accesoHibernate** (creado antes) que será donde se almacene el fichero. (4) Una vez hecho se le da a **Terminar**.

Con estos pasos se habrá creado un fichero *HibernateUtil.java* dentro del paquete *accesoHibernate*.

ACTIVIDADES 4.1



- Repetir los pasos mostrados en esta sección para crear una aplicación Java que acceda a una base de datos creada con JavaDB que está disponible dentro del propio entorno de NetBeans. La base de datos debe tener dos tablas: Album y Cancion. La tabla Album puede ser como la mostrada en la sección. La tabla Cancion representa a las canciones de un álbum y tendrá los siguientes atributos (id del álbum, id de la canción [clave], título de la canción y duración).

4.5 SESIONES. OBJETO PARA CREARLAS

Una vez creada la estructura de archivos de Hibernate, el siguiente paso es explotar todas las posibilidades que ofrece. Para ello, la clase más utilizada con Hibernate es *Session*, localizada en el paquete *org.hibernate.Session*. Esta clase contiene métodos para leer, guardar o borrar entidades sobre la base de datos.

Para crear una sesión con *Session*, primeramente se utiliza una factoría que gestiona las sesiones abiertas. Esta factoría está localizada en el fichero *HibernateUtil* creado en el último paso de la sección anterior. El siguiente código muestra un ejemplo de cómo se puede abrir una conexión usando *HibernateUtil* y la clase *Session*.

```
Session session = HibernateUtil.getSessionFactory().openSession();  
session.close();
```

En el ejemplo, la sesión abierta está materializada en el objeto *session*. Después de usar una sesión se debe cerrar con el método *close()*.

La clase *Session*⁶³ tiene como métodos más destacables.

- *beginTransaction()*: método para hacer transacciones. Las transacciones se hacen con el objeto *Transaction* (*org.hibernate.Transaction*) que tiene, entre otros, métodos para confirmar una transacción (*commit()*) y para anularla (*rollback()*).
- *save()*: método para hacer un objeto persistente en la base de datos.
- *delete()*: método para eliminar los datos de un objeto en la base de datos.
- *update()*: método para modificar un objeto.
- *get()*: método para recuperar un objeto.
- *createQuery()*: método para crear una consulta HQL (*Hibernate Query Language*) y ejecutarla sobre la base de datos. La consulta se hace con el objeto *Query* (*org.hibernate.Query*). Entre sus métodos más destacados *list()* es el que devuelve dentro de un objeto *java.util.List* el resultado de una consulta.

4.6 CARGA, ALMACENAMIENTO Y MODIFICACIÓN DE OBJETOS

En esta sección se muestra cómo usar Hibernate para las operaciones básicas sobre una base de datos (guardar un objeto, recuperarlo, modificarlo y eliminarlo).

⁶³ Más información sobre la clase *Session* puede ser encontrada en <http://docs.jboss.org/hibernate/orm/4.1/javadocs/>

4.6.1 GUARDAR

Para guardar un objeto (hacerlo persistente) se usa el método `save(Object)`. Los pasos son: (1) crear un objeto y posteriormente hacerlo persistente con el método `save`. Para que la persistencia tenga éxito es necesario abrir previamente una transacción y confirmarla al final para materializar los datos en la base de datos.

Sobre el ejemplo creado en la sección anterior, el siguiente código muestra un método `annadir_album()` que crea un nuevo objeto `Albumes` y lo guarda (`save()`) en la base de datos.

```
public static void annadir_album(int id, String tit, String aut)
{
    Transaction tx=null;
    Session session = HibernateUtil.getSessionFactory().openSession();
    tx=session.beginTransaction(); //Crea una transacción
    Albumes a = new Albumes(id);
    a.setAutor(aut);
    a.setTitulo(tit);
    session.save(a); //Guarda el objeto creado en la BBDD.
    tx.commit(); //Materializa la transacción
    session.close();
}
```

El ejemplo crea una `Transaction tx` que será la que materializa la operación de persistencia `save()` cuando todo ha sido correcto. El resultado lo muestra como salida de texto.

ACTIVIDADES 4.2



- ▶ Para el proyecto creado en la Actividad 4.1, crear métodos que permitan hacer persistentes objetos `Album` y `Cancion`.



PISTA

En el código disponible para este capítulo hay un proyecto llamado `accesoHibernate` que puede servir de ayuda para dar los primeros pasos. Sin embargo, el proyecto no incluye la base de datos ni la conexión, la cual se debe crear antes de ejecutarlo.

4.6.2 LEER

Para recuperar un objeto de la base de datos se usa el método `get(Class, clave primaria del objeto)`. El siguiente código muestra un ejemplo de utilización con un método `recuperar_Album()` al que se le pasa como parámetro la clave primaria (`id`) del objeto a recuperar.

```
public static void recuperar_album(int id)
{
    Transaction tx=null;
    Session session = HibernateUtil.getSessionFactory().openSession();
    Albumes a ;
    a=(Albumes)session.get(Albumes.class,id);
    System.out.println ("Autor: " + a.getAutor());
    session.close();

}
```

En el ejemplo, el objeto recuperado es guardado en la variable `a` de tipo `Albumes`. Para evitar errores siempre es necesario hacer un *cast* para indicar que el resultado de `get()` es un objeto de tipo `Albumes`.

ACTIVIDADES 4.3



- Para el proyecto creado en la Actividad 4.2 crear métodos que permitan recuperar los objetos persistentes `Album` y `Cancion`.

4.6.3 ACTUALIZAR

Para modificar un objeto existente en la base de datos se usa el método `update(Objeto)`. El siguiente código muestra un ejemplo de utilización con un método `modificar_Album()` al que se le pasa como parámetros la clave primaria (`id`) y los nuevos valores para `titulo` y `autor`.

```
public static void modificar_album(int id, String tit, String aut)
{
    Transaction tx=null;
    Session session = HibernateUtil.getSessionFactory().openSession();
    tx=session.beginTransaction(); //Crea una transacción
    Albumes a = new Albumes(id);
    a.setAutor(aut);
    a.setTitulo(tit);
    session.update(a); //Modifica el objeto con Id indicado
    tx.commit(); //Materializa la transacción
    session.close();

}
```

Evidentemente, hay que tener en cuenta que la clave primaria de los objetos no se puede modificar, ya que es la única referencia que se tiene para localizarlos únicamente.

Para que un objeto pueda ser modificado debe existir con anterioridad (haber hecho un *save()* del objeto). Sin embargo, en el desarrollado es posible que no se sepa si el objeto existe o no. *Session* ofrece un método llamado *saveOrUpdate (Object)* que si el objeto no existe lo guarda (*save()*) y si existe lo modifica (*update()*).

ACTIVIDADES 4.4



- Añadir al proyecto creado en la Actividad 4.3 métodos que permitan modificar los atributos de una canción dada por su ID.

4.6.4 BORRAR

Para borrar un objeto desde la base de datos el método que se utiliza es *delete (Object object)*, al que se le pasa el objeto a borrar. El siguiente código muestra el método *borrar_Album()* que borra un objeto con *id* que se le pasa como parámetro.

```
public static void borrar_album(int id)
{
    Transaction tx=null;
    Session session = HibernateUtil.getSessionFactory().openSession();
    tx=session.beginTransaction(); //Crea una transacción
    Albumes a = new Albumes(id);
    session.delete(a);
    System.out.println ("Objeto borrado");
    tx.commit(); //Materializa la transacción
    session.close();
}
```

En el ejemplo se puede observar que no es necesario rellenar todos los campos del objeto *a* para eliminarlo. Solo es necesario darle valor al atributo que hace de clave primaria, en este caso *id*.

ACTIVIDADES 4.5



- Añadir al proyecto creado en la Actividad 4.4 métodos que permitan eliminar objetos *Album* y *Cancion* dados por su ID.

4.7

CONSULTAS HQL (*HIBERNATE QUERY LANGUAGE*)

Como se ha comentado anteriormente, un ORM suele aportar un lenguaje de consulta propio para recuperar como objetos los datos de la base de datos relacional. Ese lenguaje suele ser SQL o una aproximación a SQL.

Para el caso de Hibernate el lenguaje utilizado se llama HQL (*Hibernate Query Language*). Este lenguaje es una versión de la sintaxis de SQL, adaptada para devolver objetos. Se puede decir que más que una versión de SQL es una versión de OQL, visto en el Capítulo 3.

La principal particularidad de HQL es que las consultas se realizan sobre los objetos Java creados en la aplicación, es decir las entidades que se hacen persistentes en Hibernate (o POJOs). Esto hace que HQL tenga las siguientes características:

- ✓ Los tipos de datos son los de Java.
- ✓ Las consultas son independientes del lenguaje de SQL específico de la base de datos.
- ✓ Las consultas son independientes del modelo de tablas de la base de datos. No se necesita conocer el modelo, lo que hace de la independencia una ventaja.
- ✓ Es posible tratar con las colecciones de Java (`java.io.List`, por ejemplo).
- ✓ Es posible navegar entre los distintos objetos en la propia consulta.

Además de estas características, para trabajar con HQL es necesario tener en cuenta una serie de consideraciones:⁶⁴

- **Mayúsculas:** el lenguaje no es sensible a mayúsculas y minúsculas en lo que corresponde con las palabras reservadas del lenguaje. Sin embargo, sí es sensible para el caso de los nombres de las clases y de sus atributos.
- **Filtrado:** como en SQL se pueden hacer criterios de selección con la cláusula *WHERE*. Sin embargo, no hay que confundir que las propiedades y nombre de los objetos hacen referencia a los POJO y no a las tablas de la base de datos (como sí ocurre con SQL). En las cláusulas WHERE los literales van entre comas simples (') y los decimales se expresan con punto (.). Además, se pueden usar operadores =,<,>,<=,>=, != (distinto) y like para cadenas. Los criterios se pueden concatenar con operadores lógicos AND, OR y NOT.
- **En la cláusula SELECT** se suele poner una referencia a un objeto. Pero también se pueden usar funciones de agregación sobre atributos de las clases. Algunas son: AVG (media aritmética), SUM(suma de valores), COUNT (contar elementos). En realidad permite la gran mayoría de las que permite SQL.

⁶⁴ No es objetivo de este capítulo profundizar en el lenguaje HQL. Todos los detalles sobre HQL pueden ser consultados en <http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/queryhql.html>

4.7.1 EJECUTAR HQL DESDE JAVA

Desde un proyecto Java con Hibernate se pueden ejecutar consultas HQL utilizando la clase *Query*⁶⁵ (*org.hibernate.Query*). Esta clase representa una consulta HQL y la ejecuta para devolver el resultado como objetos Java. Las consultas *Query* son creadas con el método *createQuery()* de la clase *Session*.

Uno de los métodos más usados de la clase *Query* es *list()*. Este método devuelve un *java.io.List* (Colección) con los objetos devueltos por una consulta HQL.

El siguiente ejemplo muestra un método *consulta()* que ejecuta una consulta HQL para obtener todos los objetos *Albumes* cuyo *título* contiene la palabra *love* (select a from *Albumes* a where *titulo* like '%love%').

```
public static void consulta()
{
    String c= "select a from Albumes a where titulo like '%love%';
    Session session = HibernateUtil.getSessionFactory().openSession();
    Query q= session.createQuery(c);
    List results = q.list();
    Iterator albumesiterator= results.iterator();
    while (albumesiterator.hasNext())
    {
        Albumes a2= (Albumes)albumesiterator.next();
        System.out.println ( " - " + a2.getTitulo () );
    }
    session.close();
}
```

En el ejemplo, primero se abre una sesión. Seguidamente se crea una consulta *Query* con el HQL de *c*. El método *q.List()* devuelve como *java.io.List* el resultado de la ejecución de la consulta, es decir una lista de objetos *Albumes* que satisfacen la consulta. Al ser una lista, esta es recorrida con una clase *Iterator* para obtener los valores individuales.

ACTIVIDADES 4.6



- Añadir al proyecto de la Actividad 4.5 un método que permita ejecutar una consulta que obtenga el título de los *Albumes* almacenados cuyo título empiece por "C".

⁶⁵ Más información sobre la clase *Query* puede ser encontrada en <http://docs.jboss.org/hibernate/orm/4.1/javadocs/>



PISTA

La consulta podría ser:

```
SELECT a.titulo from Albumes a WHERE a.titulo like 'C%'  
Con esta consulta, la lista de resultados no será una lista de objetos Albumes, sino una lista de objetos  
String.
```

4.8 CONCLUSIONES Y PROPUESTAS PARA AMPLIAR

En este capítulo se han mostrado las características principales de un ORM, concretando su trabajo con uno de los ORM más destacados: Hibernate 4. El trabajo con Hibernate está muy extendido dentro del desarrollo de aplicaciones. Ofrece una alternativa al uso de bases de datos OO para trabajar con Java. La abstracción que permite Hibernate al darle al programador la posibilidad de olvidarse del sistema relacional subyacente y centrarse en el trabajo con los objetos, abre un abanico importante de posibilidades en el desarrollo de aplicaciones multiplataforma.

Sin embargo, como ocurre con el resto de capítulos, tratar todo lo que Hibernate ofrece bien puede ocupar todo un libro. Por ello, este capítulo se ha centrado únicamente en las alternativas más básicas que muestren una idea global de lo que Hibernate puede ofrecer. Con esta base, profundizar en Hibernate es más sencillo.

A continuación se muestran líneas posibles de ampliación de los contenidos:⁶⁶

- ✓ Estudiar HQL y las posibilidades que ofrece para consultas avanzadas que reúnan varios objetos.
- ✓ Ampliar el conocimiento sobre los métodos y clases de Hibernate.

⁶⁶ Estos contenidos se pueden ampliar con <http://hibernate.org/>