

CAPÍTULO 3

HERRAMIENTAS DE MAPEO OBJETO-RELACIONAL (ORM)

OBJETIVOS

- Instalar y configurar una herramienta ORM.
- Interpretar y definir los ficheros de mapeo.
- Aplicar mecanismos de persistencia.
- Desarrollar aplicaciones para insertar, modificar y recuperar objetos persistentes.
- Realizar consultas con el lenguaje de la herramienta ORM.
- Gestionar transacciones.

CONTENIDOS

- Concepto de mapeo objeto-relacional.
- Herramientas objeto-relacional.
- Ficheros de mapeo. Elementos, propiedades.
- Clases persistentes.
- Sesiones; estados de un objeto.
- Carga, almacenamiento y modificación de objetos.
- Consultas SQL.
- Lenguajes propios de la herramienta ORM.

RESUMEN

En este capítulo aprenderemos a instalar una herramienta de mapeo que nos permitirá crear una capa de acceso a los datos de una base de datos relacional, de tal forma que las tablas se transformarán en clases y las filas de las tablas serán objetos. Realizaremos programas Java que accederán a la base de datos relacional usando orientación a objetos.

3.1. INTRODUCCIÓN

En este tema aprenderemos a acceder a una base de datos relacional utilizando el lenguaje orientado a objetos Java. Para acceder de forma efectiva a la base de datos desde un contexto orientado a objetos, es necesaria una interfaz que traduzca la lógica de los objetos a la lógica relacional. Esta interfaz se llama **ORM (Object Relational Mapping)** y es la herramienta que nos sirve para transformar representaciones de datos de los Sistemas de Bases de Datos Relacionales a representaciones de objetos; es decir, las tablas de nuestra base de datos pasan a ser clases y las filas de las tablas (o registros) objetos que podemos manejar con facilidad.

3.2. CONCEPTO DE MAPEO OBJETO-RELACIONAL

Según la Wikipedia 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 para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia, véase Figura 3.1. En la práctica esto 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 (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.



Figura 3.1. Mapeo objeto-relacional.

3.3. HERRAMIENTAS ORM. CARACTERÍSTICAS

Las herramientas ORM nos permiten crear una capa de acceso a datos; una forma sencilla y válida de hacerlo es crear una clase por cada tabla de la base de datos y mapearlas una a una. Estas herramientas aportan un lenguaje de consultas orientado a objetos propio y totalmente independiente de la base de datos que usemos, lo que nos permitirá migrar de una base de datos a otra sin tocar nuestro código, solo será necesario cambiar alguna línea en el fichero de configuración.

Algunas de las **ventajas** que aportan estas herramientas son:

- Ayudan a reducir el tiempo de desarrollo de software.
- Abstracción de la base de datos.
- Reutilización.

- Permiten la producción de mejor código.
- Son independientes de la Base de Datos, funcionan en cualquier BD.
- Lenguaje propio para realizar las consultas.
- Incentivan la portabilidad y escalabilidad de los programas de software.

Uno de los **inconvenientes** es que las aplicaciones son algo más lentas debido a que 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.

Existen muchas herramientas ORM, algunas son: *Doctrine*, *Propel* o *ADOdb Active Record* para incluir en proyectos PHP, *LINQ* desarrollado por Microsoft para el mapeo objeto-relacional para los lenguajes Visual Basic .Net y C#, *Hibernate* desarrollado para la tecnología Java y disponible también para la tecnología .NET con el nombre de *Nhibernate*, es software libre bajo la licencia GNU LGPL. Además de estos que hemos nombrado hay otros muchos como pueden ser *QuickDB*, *iPersist*, *Java Data Objects*, *Oracle Toplink*, etc. En este tema estudiaremos Hibernate que es uno de los ORM más populares.

Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java (y disponible también para .Net) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante ficheros declarativos (XML) que permiten establecer estas relaciones, Figura 3.2.

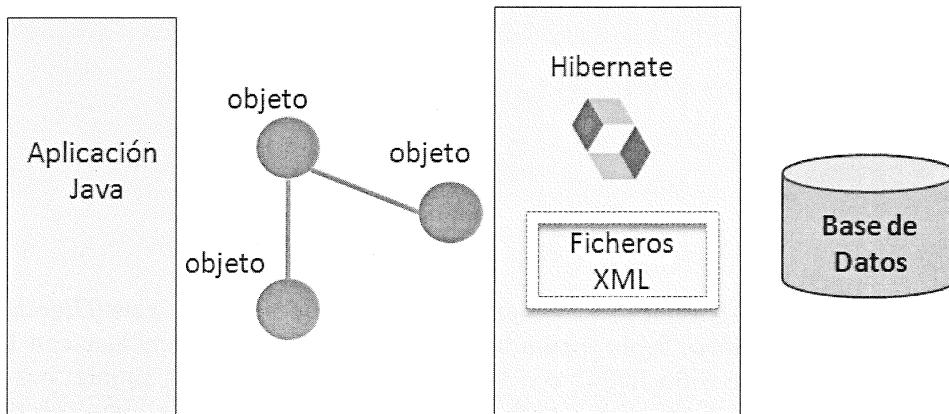


Figura 3.2. Hibernate, mapeo objeto-relacional.

Se está convirtiendo en el estándar de facto para almacenamiento persistente cuando queremos independizar la capa de negocio del almacenamiento de la información. Esta capa de persistencia permite abstraer al programador Java de las particularidades de una determinada base de datos proporcionando clases que envolverán los datos recuperados de las filas de las tablas. Hibernate busca solucionar la diferencia entre los dos modelos de datos usados para organizar y manipular datos: el modelo de objetos proporcionado por el lenguaje de programación y el modelo relacional usado en las bases de datos.

Con Hibernate no emplearemos habitualmente SQL para acceder a datos, sino que el propio motor de Hibernate, mediante el uso de factorías (patrón de diseño **Factory**) y otros elementos de programación construirá esas consultas para nosotros. Hibernate pone a disposición del diseñador un lenguaje llamado **HQL (Hibernate Query Language)** que permite acceder a datos mediante POO.

3.4. ARQUITECTURA HIBERNATE

Hibernate parte de una filosofía de mapear objetos Java normales o más conocidos en la comunidad como "POJOs" (*Plain Old Java Objects*). Para almacenar y recuperar estos objetos de la base de datos, el desarrollador debe mantener una conversación con el motor de Hibernate mediante un objeto especial que es la **sesión** (clase **Session**) (equiparable al concepto de conexión de JDBC). Igual que con las conexiones JDBC hemos de crear y cerrar sesiones. La arquitectura Hibernate se muestra en la Figura 3.3, donde se observan varias capas. Entre la capa de Hibernate y la de base de datos se muestran diferentes APIs Java que usan Hibernate para interactuar con la base de datos.

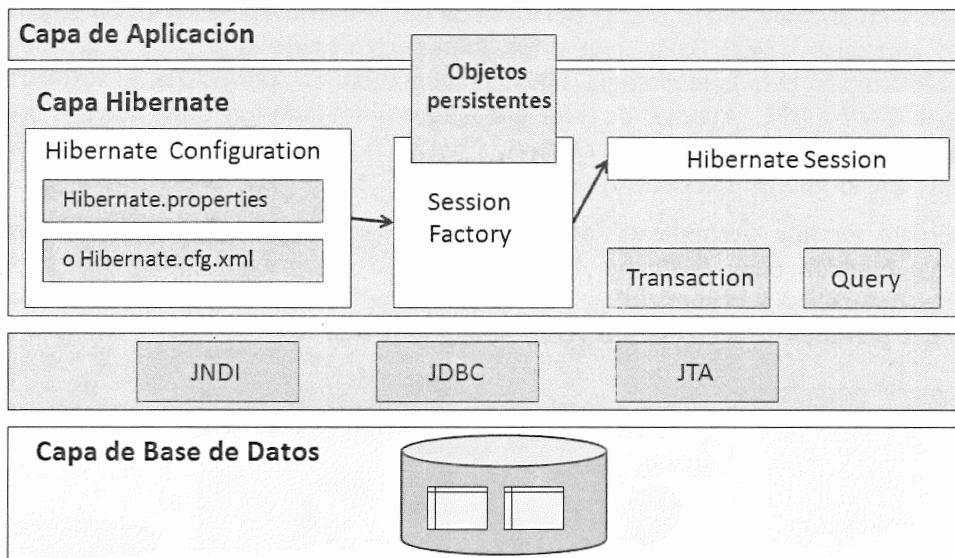


Figura 3.3. Arquitectura Hibernate.

La clase **Session** (`org.hibernate.Session`) ofrece métodos como `save(Object objeto)`, `createQuery(String consulta)`, `beginTransaction()`, `close()`, etc. para interactuar con la BD tal como se hace con una conexión JDBC, con la diferencia que resulta más simple; por ejemplo, guardar un objeto consiste en algo así como `session.save(miObjeto)`, sin necesidad de especificar una sentencia SQL. Una instancia de **Session** no consume mucha memoria y su creación y destrucción es muy barata. Esto es importante, ya que nuestra aplicación necesitará crear y destruir sesiones todo el tiempo, quizás en cada petición. Puede ser útil pensar en una sesión como en una caché o colección de objetos cargados (a o desde una base de datos) relacionados con una única unidad de trabajo. Hibernate puede detectar cambios en los objetos pertenecientes a una unidad de trabajo.

Las interfaces de Hibernate son los siguientes:

- La interfaz **SessionFactory** (`org.hibernate.SessionFactory`) permite obtener instancias **Session**. Esta interfaz debe compartirse entre muchos hilos de ejecución. Normalmente hay una única **SessionFactory** para toda la aplicación, creada durante la inicialización de la misma, y se utiliza para crear todas las sesiones relacionadas con un contexto dado. Si la aplicación accede a varias bases de datos se necesitará una **SessionFactory** por cada base de datos.

- La interfaz **Configuration** (`org.hibernate.cfg.Configuration`) se utiliza para configurar 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 **SessionFactory**.
- La interfaz **Query** (`org.hibernate.Query`) permite realizar consultas a la base de datos y controla cómo se ejecutan dichas consultas. Las consultas se escriben en HQL o en el dialecto SQL nativo de la base de datos que estemos utilizando. Una instancia **Query** se utiliza para enlazar los parámetros de la consulta, limitar el número de resultados devueltos y para ejecutar dicha consulta.
- La interfaz **Transaction** (`org.hibernate.Transaction`) nos permite asegurar que cualquier error que ocurra entre el inicio y final de la transacción produzca el fallo de la misma.

Hibernate hace uso de APIs de Java, tales como JDBC, JTA (*Java Transaction Api*) y JNDI (*Java Naming Directory Interface*).

La Figura 3.4 muestra cómo sería una aplicación con Hibernate.

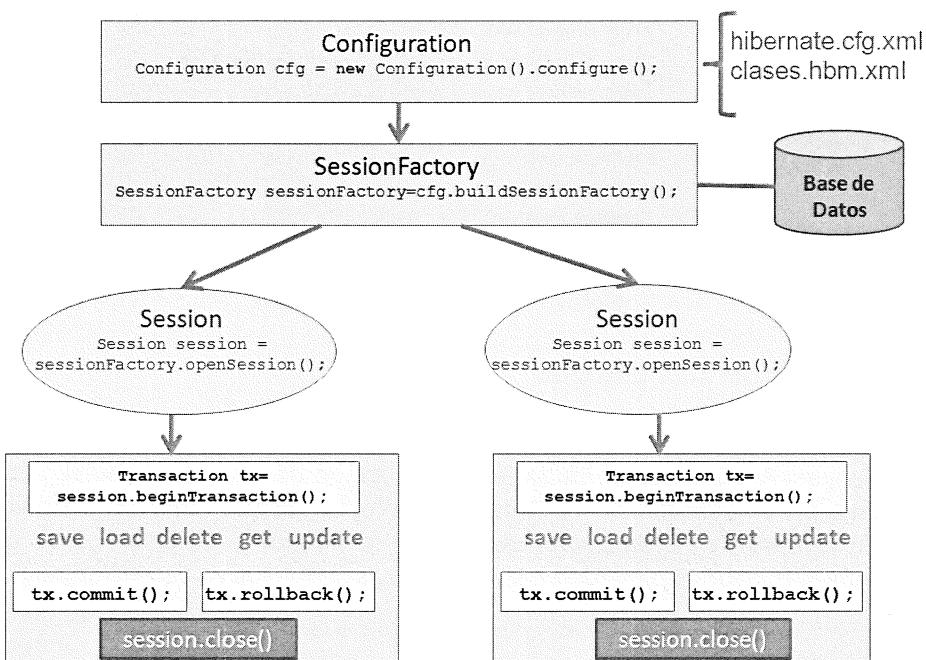


Figura 3.4. Aplicación con Hibernate.

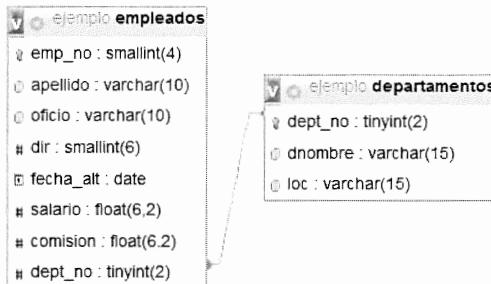
3.5. INSTALACIÓN Y CONFIGURACIÓN DE HIBERNATE

En esta apartado vamos a instalar y configurar Hibernate en el entorno Eclipse (en este caso se han hecho las pruebas en la versión Mars). Para los ejemplos vamos a utilizar la base de datos MySQL de nombre *ejemplo*, creada en la Capítulo anterior, recordemos que su propietario era el usuario *ejemplo* y la clave la misma que el nombre del usuario. Se creaban las tablas *empleados* y *departamentos*, las relaciones se muestran en la Figura 3.5. La creación de las tablas es la siguiente, la clave ajena en la tabla *empleados* debe crearse dando nombre a la restricción:

```

CREATE TABLE departamentos (
    dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
    dnombre  VARCHAR(15),
    loc      VARCHAR(15)
) ENGINE=InnoDB;
CREATE TABLE empleados (
    emp_no     SMALLINT(4)  NOT NULL PRIMARY KEY,
    apellido   VARCHAR(10),
    oficio     VARCHAR(10),
    dir        SMALLINT,
    fecha_alt  DATE,
    salario    FLOAT(6,2),
    comision   FLOAT(6,2),
    dept_no    TINYINT(2) NOT NULL,
    CONSTRAINT fkdep FOREIGN KEY (dept_no)
        REFERENCES departamentos (dept_no)
) ENGINE=InnoDB;

```

Figura 3.5. Base de datos *ejemplo*.

3.5.1. Instalación del plugin

Para instalar el plugin de Hibernate en Eclipse se necesita tener conexión a Internet. Primero iniciamos Eclipse. A continuación pulsamos en la opción del menú horizontal **Help-> Install New Software**, rellenamos el campo **Work With** con la siguiente URL: <http://download.jboss.org/jbosstools/updates/stable/> y pulsamos el botón *Add*, nos pide un nombre, escribimos por ejemplo *Hibernate*, y pulsamos el botón *OK*, véase Figura 3.6.

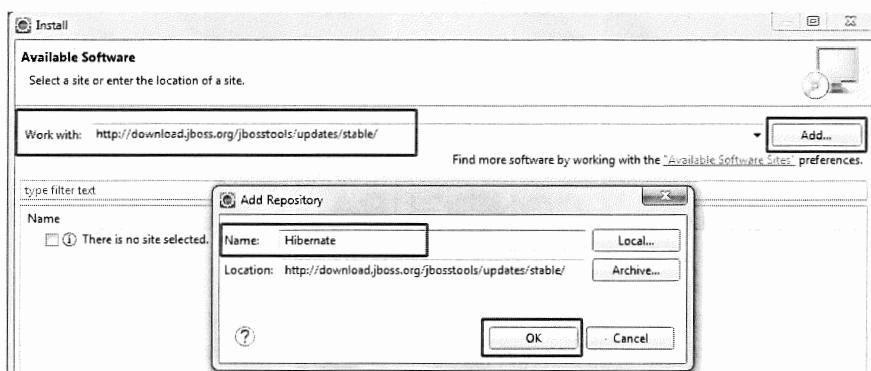


Figura 3.6. Instalar plugin de Hibernate

Al rato aparece la lista de plugins. Pulsamos en la flechita que aparece a la izquierda de **JBoss Data Services Development** y seleccionamos **Hibernate Tools**. A continuación pulsamos al botón *Next* (Figura 3.7), comienza el proceso de descarga.

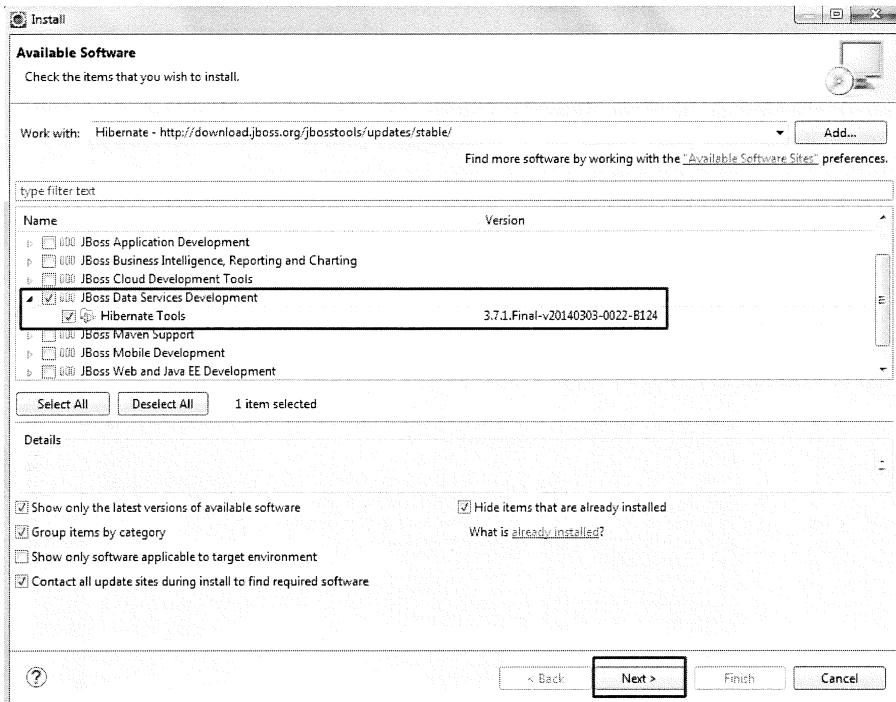


Figura 3.7. Seleccionar plugin Hibernate Tools.

Una vez descargado se visualiza una ventana con los detalles del elemento a instalar, pulsamos de nuevo en *Next*. A continuación aceptamos la licencia y pulsamos el botón *Finish*. El proceso de instalación comienza, puede tardar un rato. Durante la instalación nos pide confirmación para continuar, ya que el plugin contiene software sin firmar.

Una vez instalado nos pide reiniciar Eclipse. Para comprobar que se ha instalado correctamente podemos pulsar en la opción de menú **Windows -> Show View -> Other**, deben aparecer las opciones de Hibernate, Figura 3.8.

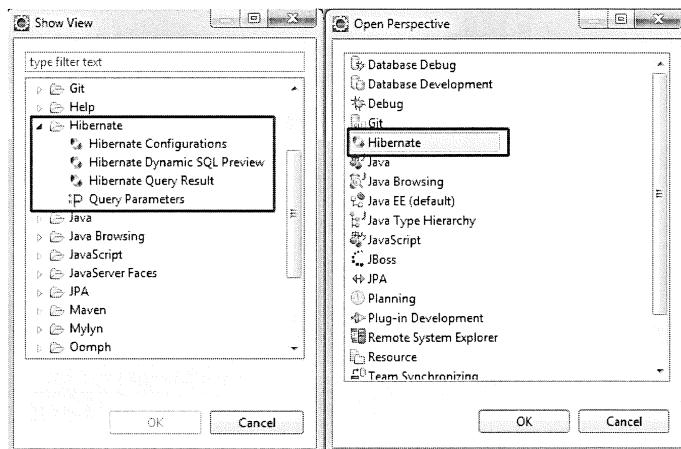


Figura 3.8. Comprobar la instalación de Hibernate

También se puede comprobar desde el menú **Window-> Perspective -> Open Perspective -> Other-> Hibernate**, véase Figura 3.8

3.5.2. Configuración del driver MySQL

Una vez instalado Hibernate, el siguiente paso es configurarlo para que se comunique con MySQL. Utilizaremos la base de datos *ejemplo*. En primer lugar hemos de descargarnos el driver MySQL desde la URL <http://dev.mysql.com/downloads/connector/j/>. En el capítulo anterior ya lo usamos, debemos tenerlo localizado en alguna carpeta.

A continuación desde el menú: **Window -> Preferences -> Data Management -> Connectivity-> Driver Definitions** se pulsa el botón *Add*:

- Desde la pestaña **Name/Type** se selecciona de la lista *Vendor Filter* la opción **MySQL**. Y a continuación en la lista de drivers seleccionamos **MySQL JDBC Driver** en la versión 5.1, véase Figura 3.9

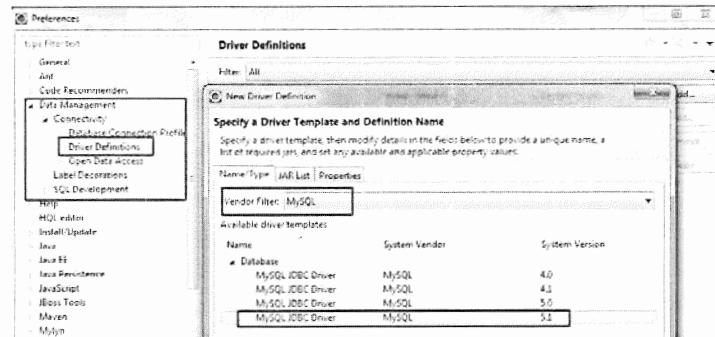


Figura 3.9. Comprobar la instalación de Hibernate

- Desde la pestaña **JAR List** pulsamos el botón *Add JAR/Zip* para localizar el conector **mysql-connector-java-5.1.38-bin.jar**, se debe buscar la carpeta donde tengamos dicho conector. Si se visualiza el driver **mysql-connector-java-5.1.0-bin.jar** lo eliminamos pulsando el botón *Remove JAR/Zip*, Figura 3.10. Se pulsa el botón *OK*. Y de nuevo *OK* para salir de esta pantalla.

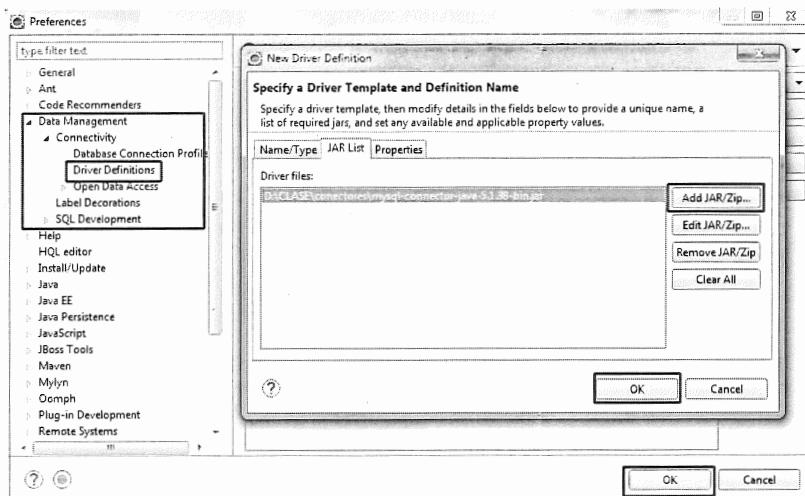


Figura 3.10. Instalación del driver MySQL

A continuación vamos a crear un proyecto Eclipse (le damos el nombre de *Proyecto1*) y configuraremos Hibernate para que se comunique con MySQL y nos cree las clases correspondientes de cada tabla de la base de datos *ejemplo*. Pulsamos en el menú **File-> New -> Project->Java Project** y pulsamos el botón *Next*, nos pide el nombre del proyecto, por ejemplo, escribimos *Proyecto1* y pulsamos el botón *Finish*. Pregunta si queremos abrir la perspectiva asociada al proyecto, pulsamos el botón *Yes*.

Ahora hemos de agregar el driver MySQL, para ello seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos **Build Path-> Add Libraries**. Se visualiza una ventana desde la que hemos de elegir la opción **Connectivity Driver Definition** y pulsar el botón *Next*. A continuación se visualiza una nueva ventana donde aparecen las opciones disponibles para conectarnos a una fuente de datos, elegimos de la lista la opción **MySQL JDBC Driver** y pulsamos el botón *Finish*, véase Figura 3.11.

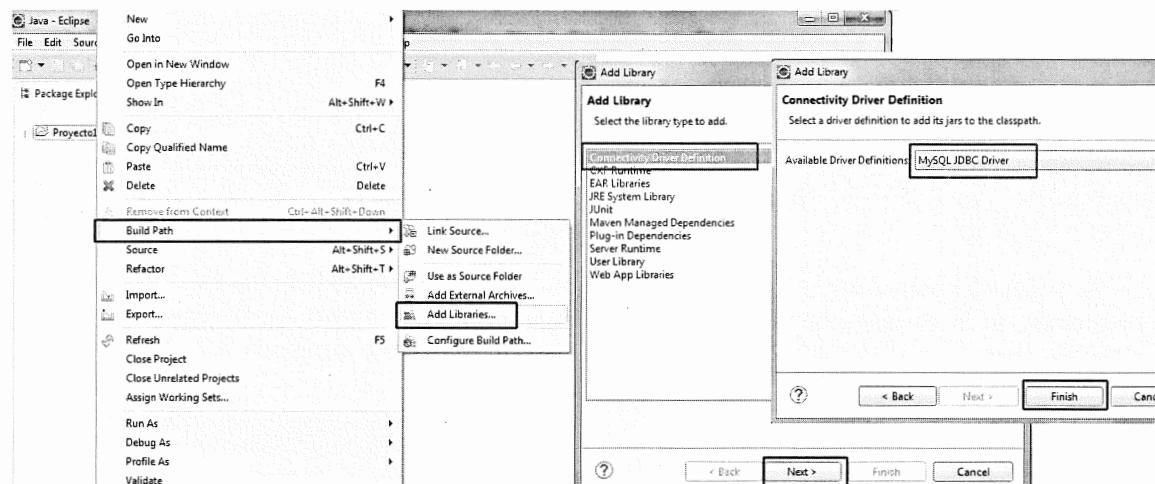


Figura 3.11. Agregar la librería MySQL JDBC al proyecto

El proyecto debe mostrar un aspecto similar al de la Figura 3.12.

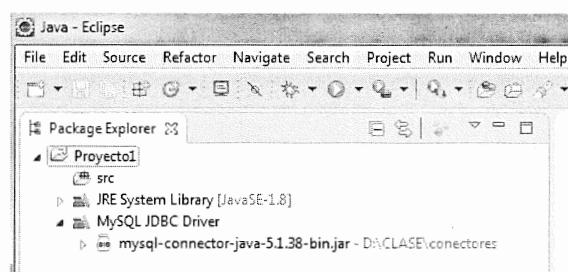
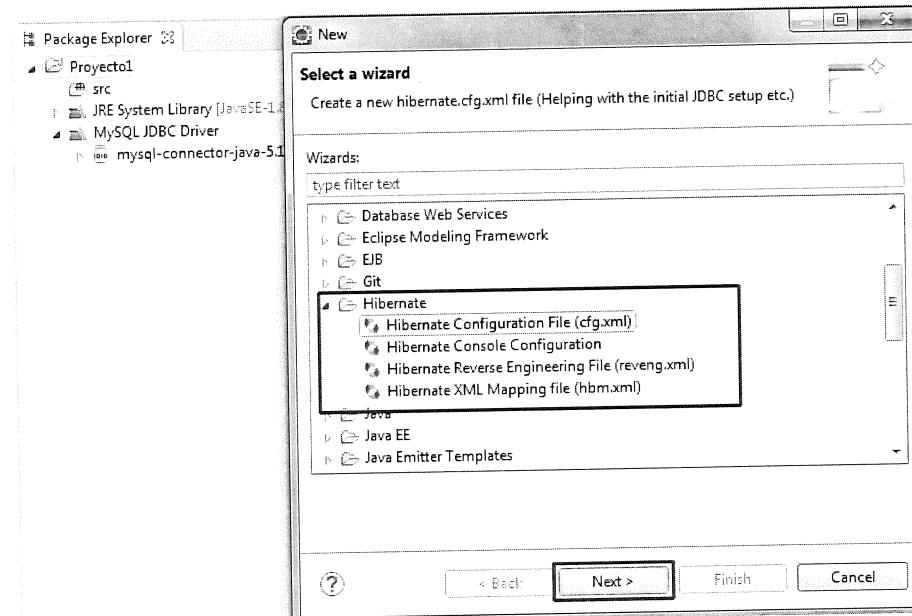


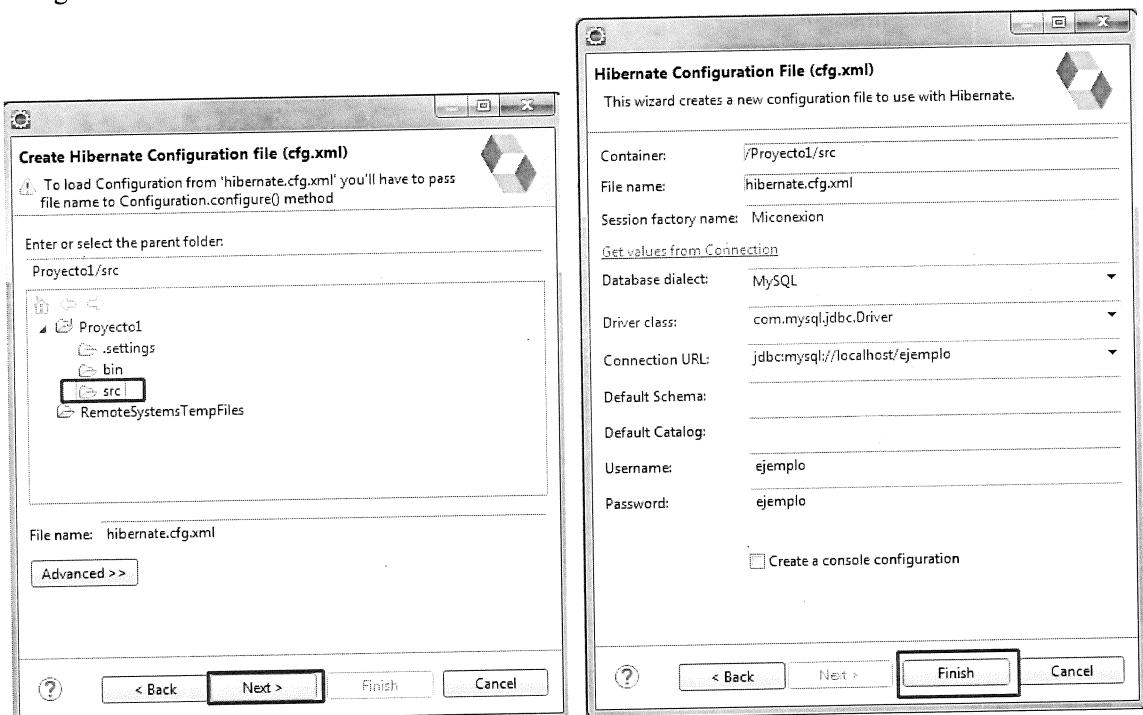
Figura 3.12. Proyecto con la librería MySQL.

3.5.3. Configuración de Hibernate

Una vez que tenemos la librería MySQL en nuestro proyecto hemos de crear el fichero de configuración de Hibernate llamado **hibernate.cfg.xml**. Seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y pulsamos sobre: **New-> Other->Hibernate->Hibernate Configuration File (cfg.xml)**, Figura 3.13. Este fichero es un XML que contiene todo lo necesario para realizar la conexión a la base de datos.

Figura 3.13.Crear fichero *hibernate.cfg.xml*.

Se pulsa el botón *Next*. A continuación nos pide donde crear el fichero, en este ejemplo lo crearemos en la carpeta por defecto llamada *src*, pulsamos de nuevo *Next*, véase Figura 3.14. Seguidamente hemos de escribir los datos para poder realizar la conexión a la base de datos, véase Figura 3.14.

Figura 3.14.Datos para la configuración del fichero *hibernate.cfg.xml*.

Los campos a rellenar son los siguientes:

- **Session factory name:** aquí se escribe el nombre de nuestra conexión a MySQL, en este caso se le ha dado el nombre *Miconexion*.
- **Database dialect:** desde esta lista se elige como se comunica JDBC con la base de datos, se elige *MySQL*.
- **Driver Class:** se selecciona la clase de JDBC que se va a usar para la conexión, se elige **com.mysql.jdbc.Driver**.
- **Connection URL:** se elige la ruta de conexión a nuestra base de datos, se elige de la lista la opción: *jdbc:mysql://<hostname>/<database>* y se cambia por *jdbc:mysql://localhost/ejemplo*.
- **Username:** usuario que se conectará a la base de datos *ejemplo*, en este caso el nombre se usuario es *ejemplo*.
- **Password:** clave del usuario que realiza la conexión con MySQL, en este caso es *ejemplo*.

La casilla **Create a console configuration** crea el fichero **XML Hibernate Console Configuration** con el mismo nombre que el proyecto Eclipse, podemos marcarla, pulsar el botón **Next** y finalizar o seguir el paso siguiente para crear el fichero.

Para terminar se pulsa el botón **Finish**. Se visualiza el editor de configuración de Hibernate, véase Figura 3.15.

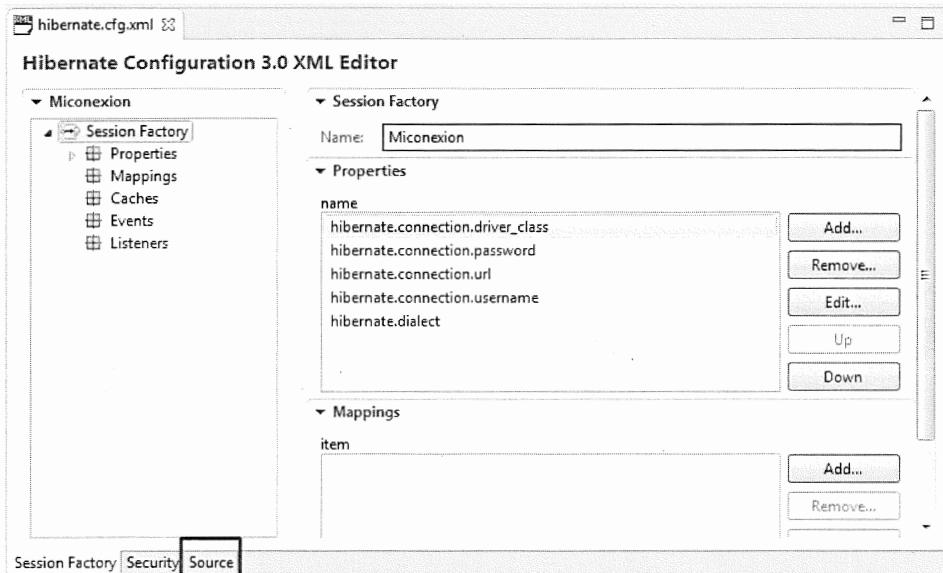


Figura 3.15. Editor de configuración de Hibernate.

Desde la pestaña **Source** se puede editar el fichero XML **cfg.xml** generado:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
```

```

<hibernate-configuration>
    <session-factory name="Miconexion">
        <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property
name="hibernate.connection.password">ejemplo</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost/ejemplo</property>
        <property
name="hibernate.connection.username">ejemplo</property>
        <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    </session-factory>
</hibernate-configuration>

```

Una vez creado el fichero **hibernate.cfg.xml** hemos de crear el fichero **XML Hibernate Console Configuration**. Seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos: **New-> Other->Hibernate->Hibernate Console Configuration**, se pulsa el botón **Next**, Figura 3.16.

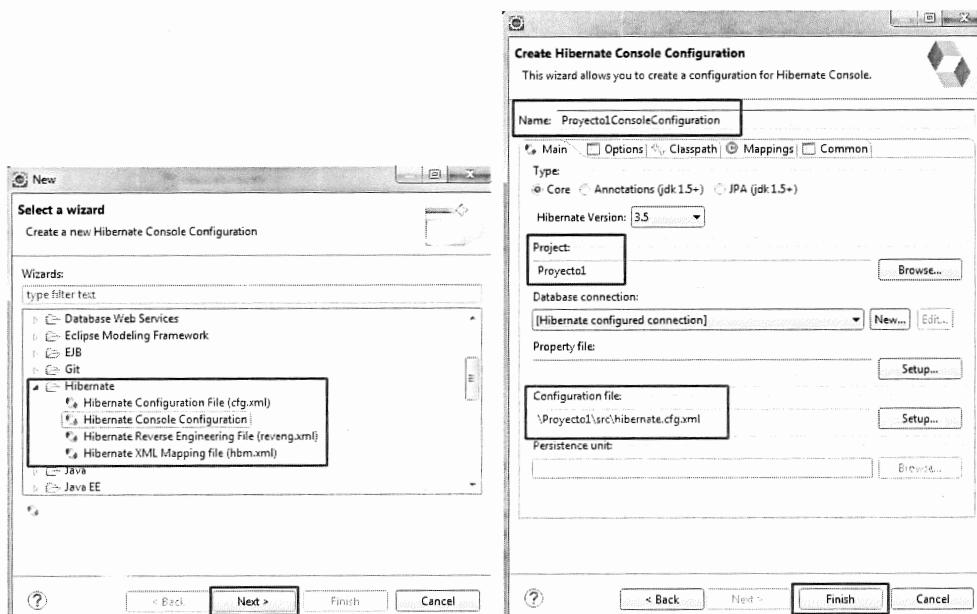


Figura 3.16.Crear Hibernate Console Configuration.

Se abre una nueva ventana, en el campo **Name** escribimos un nombre para nuestra configuración, por ejemplo *Proyecto1ConsoleConfiguration*. Nos aseguramos que en el campo **Project** aparezca nuestro proyecto y en el campo **Configuration file** aparezca el fichero de configuración creado anteriormente (**hibernate.cfg.xml**). Pulsamos **Finish** para terminar el proceso de creación.

Por último, hemos de crear el fichero **XML Hibernate Reverse Engineering (reveng.xml)** que es el encargado de crear las clases de nuestras tablas MySQL. Pulsamos el botón derecho del ratón sobre el proyecto y seleccionamos: **New-> Other->Hibernate-> Hibernate Reverse**

Engineering File(reveng.xml). Pulsamos el botón *Next* y nos pide que indiquemos donde se va a guardar el fichero, se debe guardar en la misma carpeta que el fichero **hibernate.cfg.xml**, en este caso en la carpeta *src*, Figura 3.17.

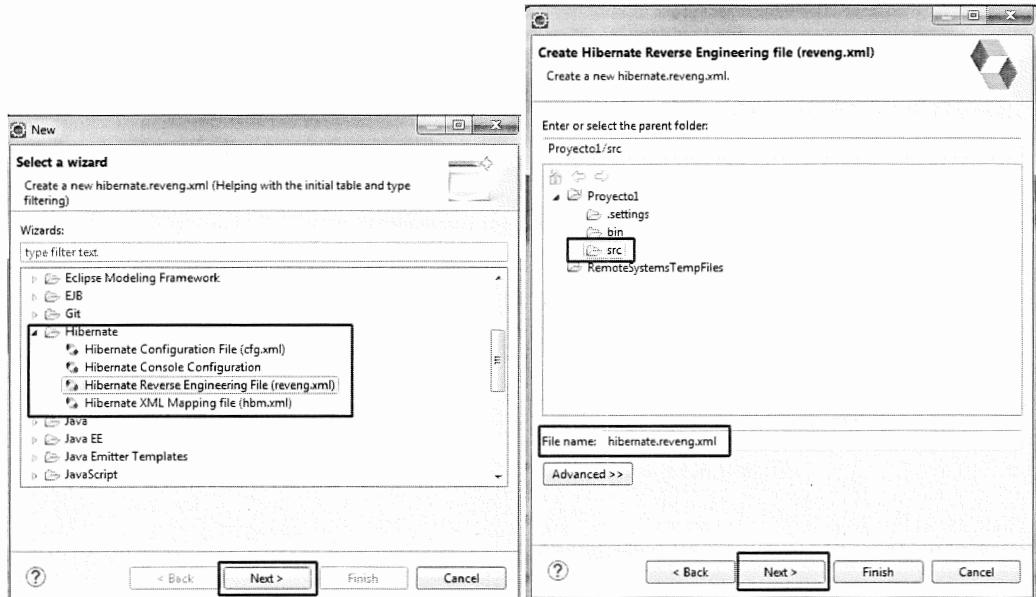


Figura 3.17. Crear fichero reveng.xml.

Pulsamos *Next*, se visualiza una nueva ventana desde donde indicaremos las tablas que queremos mapear. Desde la lista **Console configuration** seleccionamos el nombre que dimos al fichero *Hibernate Console Configuration*, en este caso *Proyecto1ConsoleConfiguration*, y pulsamos el botón *Refresh* para que muestre la base de datos *ejemplo* y sus tablas, véase Figura 3.18. Seleccionamos una a una o todas las tablas y pulsamos el botón *Include*. Para finalizar pulsamos el botón *Finish*.

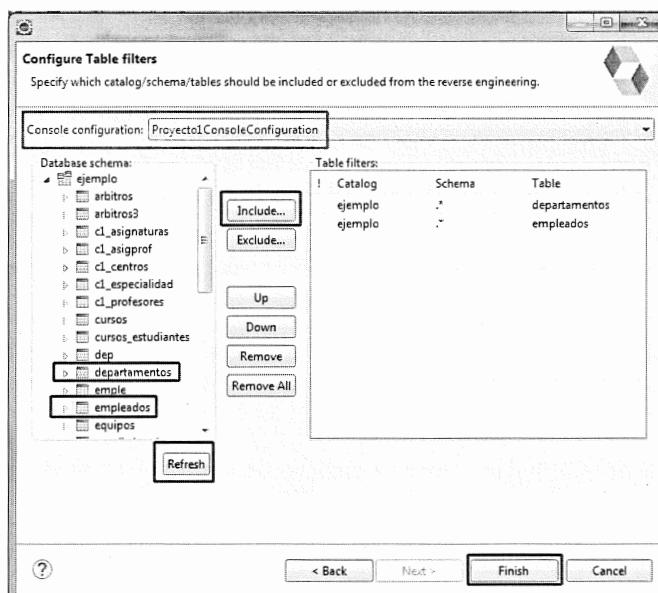


Figura 3.18. Tablas a mapear con Hibernate.

Se visualiza el editor de *Hibernate Reverse Engineering*, véase Figura 3.19.

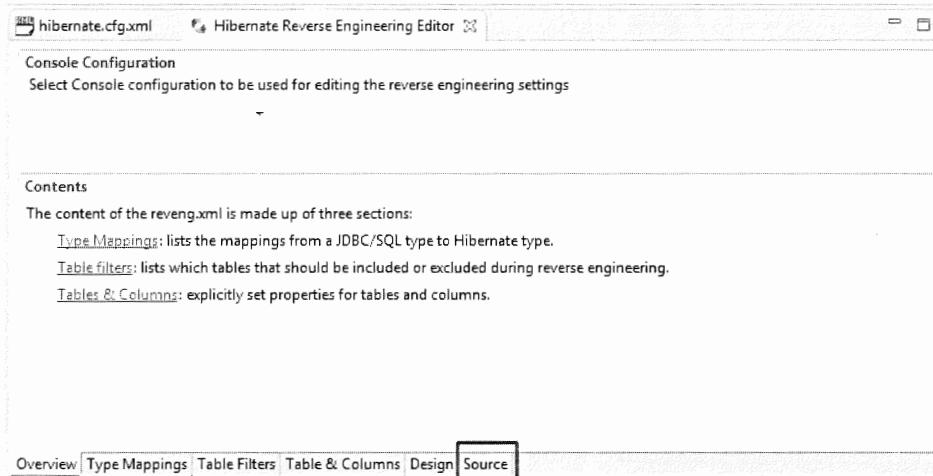


Figura 3.19. Editor *Hibernate Reverse Engineering*.

Desde la pestaña **Source** se puede editar el fichero XML **reveng.xml** generado:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate
Reverse Engineering DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-reverse-engineering-
3.0.dtd" >

<hibernate-reverse-engineering>
  <table-filter match-catalog="ejemplo" match-name="departamentos"/>
  <table-filter match-catalog="ejemplo" match-name="empleados"/>
</hibernate-reverse-engineering>
```

ACTIVIDAD 3.1

Crea un nuevo proyecto Java para acceder a Oracle usando Hibernate, le damos el nombre *Proyecto1OracleHibernate*. Accederemos a las tablas EMPLEADOS Y DEPARTAMENTOS del usuario de nombre EJEMPLO creado en la Capítulo 2. Seguimos el epígrafe 3.5.2 para configurar el driver Oracle:

- Desde la pestaña *Name/Type* se selecciona *Oracle Thin Driver* en la versión 11. (Si de la lista *Available Driver Definitions* no aparece driver para Oracle pulsamos en el botón *New Driver Definition* localizado al lado de la lista).

- Desde la pestaña *JAR List* pulsamos el botón *Add JAR/Zip* para localizar el conector **ojdbc6.jar**. Si se visualiza el driver **ojdbc14.jar** lo eliminaremos pulsando el botón *Remove JAR/Zip*.

Seguimos el Epígrafe 3.5.3 para crear el fichero de configuración de Hibernate, escribimos los siguientes valores:

- *Session factory name*: MiConexionOracle

- *Database Dialect*: Oracle 10g

- *Driver Class*: oracle.jdbc.driver.OracleDriver
- *Conection URL*: jdbc:oracle:thin:@localhost:1521:XE (versión Oracle Express)
- *Default Schema*: EJEMPLO
- *Username*: EJEMPLO
- *Password*: EJEMPLO

3.5.4. Generar las clases de la base de datos

El siguiente paso es generar las clases de nuestra base de datos *ejemplo*. Para ello pulsamos en la flechita situada a la derecha del botón *Run As* y seleccionamos **Hibernate Code Generation Configurations**, Figura 3.20.

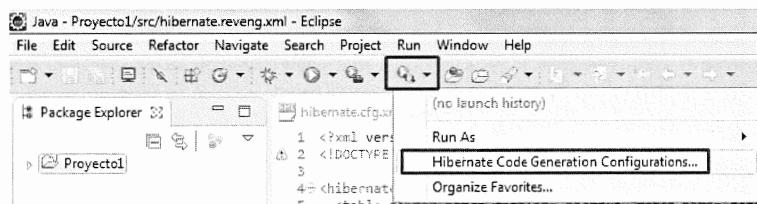


Figura 3.20. Botón *Run As...*

Desde la ventana que aparece hacemos doble clic en la opción: **Hibernate Code Generation** que se visualiza en el marco de la izquierda. Al hacer doble clic se visualizan varias pestañas. Desde la pestaña **Main** configuramos los siguientes campos, véase Figura 3.21 (el resto se dejan los valores por defecto):

- *Name*: escribimos un nombre para esta configuración, por ejemplo, *ConfiguracionProyecto1*.
- *Console configuration*: seleccionamos de la lista *Proyecto1ConsoleConfiguration*.
- *Output directory*: debe ser la carpeta *src*, pulsando en el botón *Browse* localizamos la carpeta *\Proyecto1\src*.
- *Package*: escribimos el nombre del paquete donde se crearán las clases, por ejemplo, *primero*.
- *reveng.xml*: localizamos el fichero **reveng.xml** creado anteriormente. Pulsando en el botón *Setup* se puede localizar.

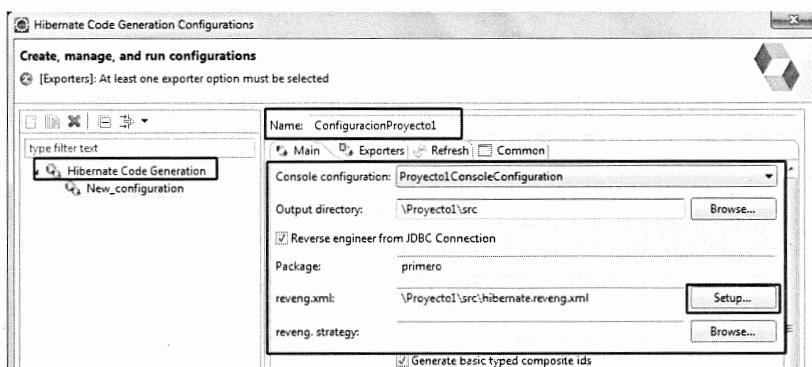


Figura 3.21. Pestaña *Main*.

Desde la pestaña **Exporters** se indica los ficheros que queremos generar, se marcan las casillas: *Use Java 5 syntax*, *Domain code*, *Hibernate XML Mappings* e *Hibernate XML Configuration*. Una vez seleccionadas se pulsa el botón *Apply* y posteriormente se pulsa *Run*, véase Figura 3.22.

Al ejecutarse nos genera un paquete llamado *primero* con las clases Java de las tablas *empleados* (*Empleados.java*) y *departamentos* (*Departamentos.java*) que contienen los métodos *getters* y *setters* de cada campo de la tabla; y los ficheros XML, *Departamentos.hbm.xml* y *Empleados.hbm.xml* que contienen la información del mapeo de su respectiva tabla, véase Figura 3.23.

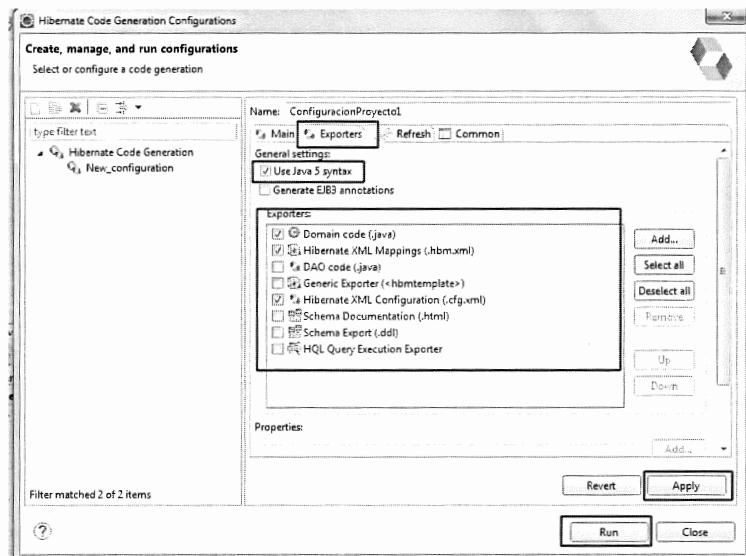


Figura 3.22.Pestaña Exporters.

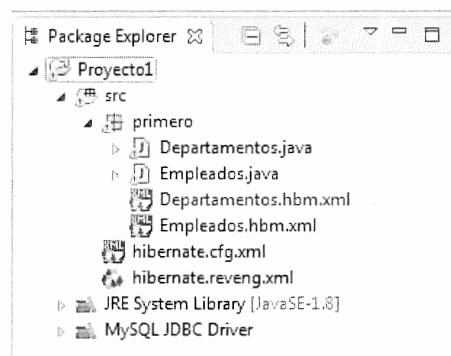


Figura 3.23. Proyecto1 con las clases Java y ficheros *hbm.xml* generados.

Para cada clase se generan una serie de atributos que tienen que ver con las columnas de la tabla que mapean y las relaciones de claves ajenas, varios constructores; y los métodos *getters* y *setters*. En la clase *Departamentos* se observan los siguientes atributos:

```
private byte deptNo;
private String dnombre;
private String loc;
private Set<Empleados> empleadoses = new HashSet<Empleados>(0);
```

Los atributos *deptNo*, *dnombre* y *loc* se corresponden con las columnas de la tabla. El atributo *empleadoses* define la relación de clave ajena entre las tablas *empleados* y *departamentos*. Este atributo sirve para almacenar los empleados del departamento.

En la clase *Empleados* se observan los siguientes atributos:

```
private short empNo;
private Departamentos departamentos;
private String apellido;
private String oficio;
private Short dir;
private Date fechaAlt;
private Float salario;
private Float comision;
```

El atributo de número de departamento (columna DEPT_NO de la tabla) no aparece, en su lugar aparece un atributo de nombre *departamentos* que es un objeto de la clase *Departamentos* y que hace referencia al departamento del empleado. Recordemos que la columna DEPT_NO de la tabla *empleados* es la clave ajena que referencia a la tabla *departamentos*, al mapear dicha columna se genera un atributo del tipo *Departamentos*.

3.5.5. Primera consulta en HQL

A continuación vamos a realizar consultas en HQL para comprobar si la conexión a la base de datos funciona correctamente. Abrimos la perspectiva de Hibernate desde la opción de menú **Window-> Perspective-> Open Perspective-> Other-> Hibernate**. Desde la pestaña **Hibernate Configurations** pulsamos en la configuración de nombre *Proyecto1ConsoleConfiguration* con el botón derecho del ratón y seleccionamos **HQL Editor**, véase Figura 3.24.

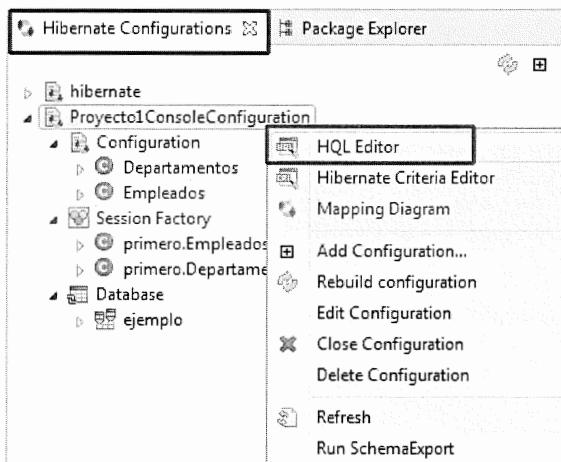


Figura 3.24. Pestaña *Hibernate Configurations*.

Se abre una nueva pestaña con el mismo nombre que la configuración de Hibernate. Desde aquí podemos escribir sentencias HQL para consultar la base de datos. A continuación escribimos el siguiente código HQL para consultar los empleados: *from Empleados*, y pulsamos el botón con la flechita verde ▶ para ejecutar la consulta, véase Figura 3.25. Desde la pestaña

Hibernate Query Result se puede ver el resultado de la consulta. Si seleccionamos una fila, en el panel **Property** veremos el contenido de la misma.

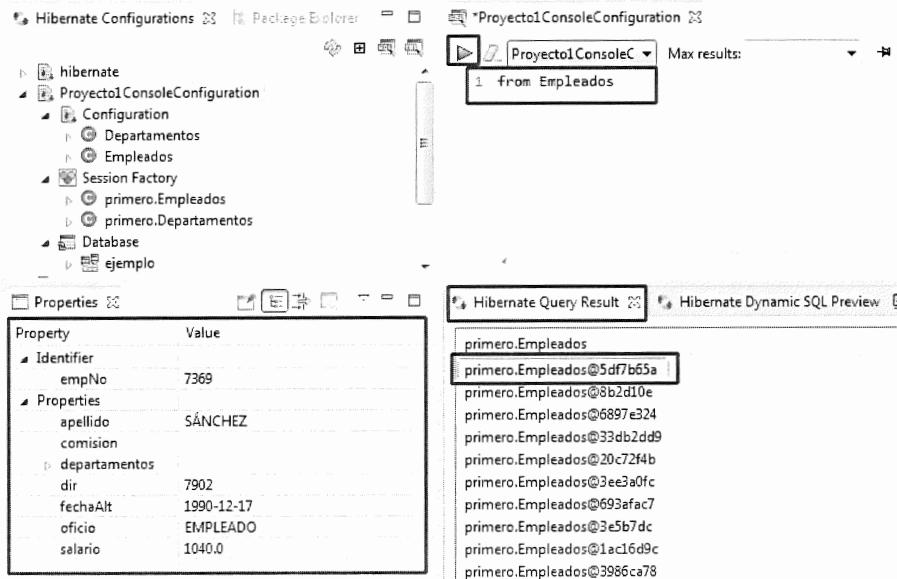


Figura 3.25. Ejecución de una consulta en HQL.

Desde este entorno también se pueden realizar consultas al estilo SQL, respetando los nombres de las clases y de los atributos de las mismas, por ejemplo:

```
select dnombre, loc, deptNo from Departamentos
select empNo, apellido, salario from Empleados where dept_no = 10
```

Ten en cuenta que el * no se puede utilizar a la derecha de SELECT, este ejemplo: `select * from Empleados` produce un error. Las siguientes SELECT son también erróneas porque no respetan los nombres de las clases y los atributos:

```
from departamentos
select Dnombre from Departamentos
from Departamentos as dep where dep.dept_no = 10
select emp_no, apellido, salario from Empleados where emp_no = 7839
```

Para comentar líneas usamos doble guion --. Al final del capítulo encontrarás un resumen de sentencias en HQL.

ACTIVIDAD 3.2

Realiza consultas HQL con las tablas mapeadas. Prueba estas consultas:

`from Empleados as e where e.departamentos.deptNo = 10`

`from Departamentos where deptNo = 10`

`from Departamentos as d join d.empleadoses`

`from Departamentos as d left outer join d.empleadoses`

```
select avg(salario), count(empNo) from Empleados where departamentos.deptNo = 10
```

```
select apellido, salario from Empleados as e where e.departamentos.deptNo =10
```

```
select departamentos.dnombre, avg(salario), count(empNo) from Empleados group by departamentos.deptNo
```

Desde la perspectiva Hibernate, pulsamos con el botón derecho del ratón en **Configuration** y seleccionamos **Mapping Diagram**, se muestra el diagrama de mapeo entre las clases Java y las tablas de la base de datos, véase Figura 3.26.

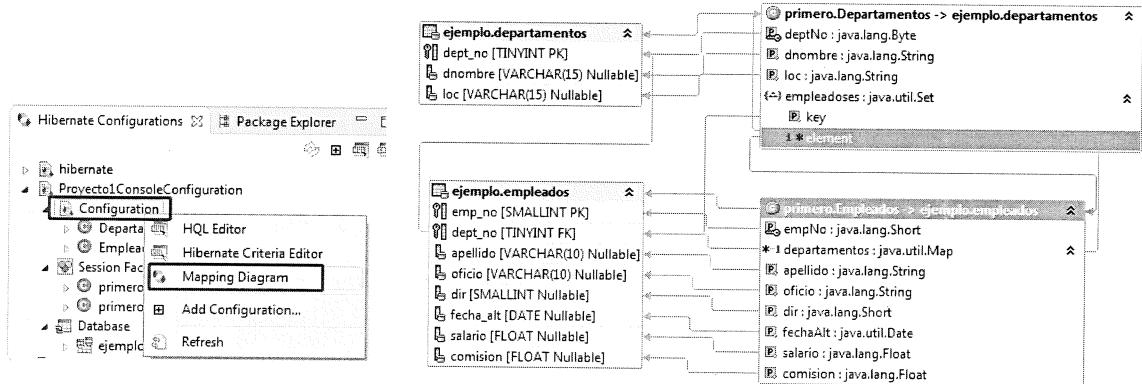


Figura 3.26. Diagrama de mapeo.

3.5.6. Empezando a programar con Hibernate en Eclipse

Con la configuración realizada hasta ahora no se puede programar en Java, es necesario realizar los siguientes pasos:

1. Desde la URL <http://hibernate.org/orm/downloads/> descargamos la última distribución estable de Hibernate. En este caso se ha bajado la versión: **hibernate-release-5.1.0.Final.zip**.
2. A continuación hemos de preparar la distribución para agregarla a Eclipse. Creamos una carpeta dentro de Eclipse con nombre *Hibernate* y aquí es donde copiaremos las librerías que vamos a necesitar, nos debe quedar: *D:\eclipse\Hibernate*. Descomprimimos el ZIP en esta carpeta (en Linux lo descargamos en *opt/eclipse/Hibernate*).
3. Desde la carpeta *D:\eclipse\Hibernate\hibernate-release-5.1.0.Final\lib\required* seleccionamos todos los ficheros y los copiamos a nuestra carpeta *D:\eclipse\Hibernate*.
4. Descargamos la última versión de la librería *slf4j* desde la URL <http://www.slf4j.org/download.html>, se ha descargado el fichero **slf4j-1.7.21.zip** (en Linux **slf4j-1.7.21.tar.gz**), lo descomprimimos y localizamos los ficheros **slf4j-api-1.7.21.jar** y **slf4j-simple-1.7.21.jar** para copiarlos en la carpeta *D:\eclipse\Hibernate*.
5. Los ficheros JAR que deben contener la carpeta son los que se muestran en la Figura 3.27.

6. Desde Eclipse hacemos clic con el botón derecho del ratón en nuestro proyecto y pulsamos en **Build Path → Add Libraries**. Se visualiza una nueva ventana desde la que elegimos **User Library**. En este paso crearemos una librería con todos los JAR que necesita Hibernate, pulsamos el botón *Next* (Figura 3.28).

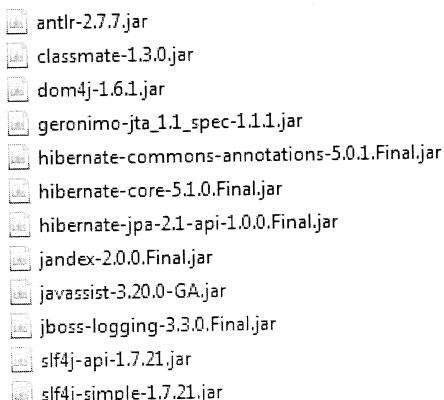


Figura 3.27. Ficheros JAR necesarios.

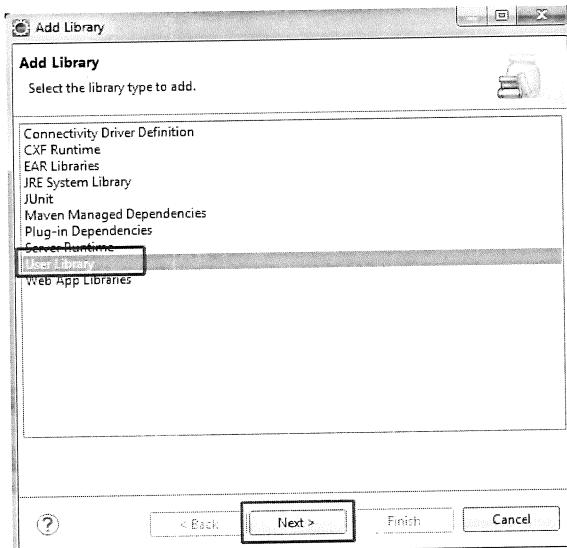


Figura 3.28. Añadir librería de usuario.

7. Desde la siguiente ventana pulsamos el botón *User Libraries*, a continuación pulsamos el botón *New*, nos pedirá el nombre de la librería que queremos agregar, escribimos por ejemplo: **HibernateLib** y pulsamos el botón *OK*, véase Figura 3.29.

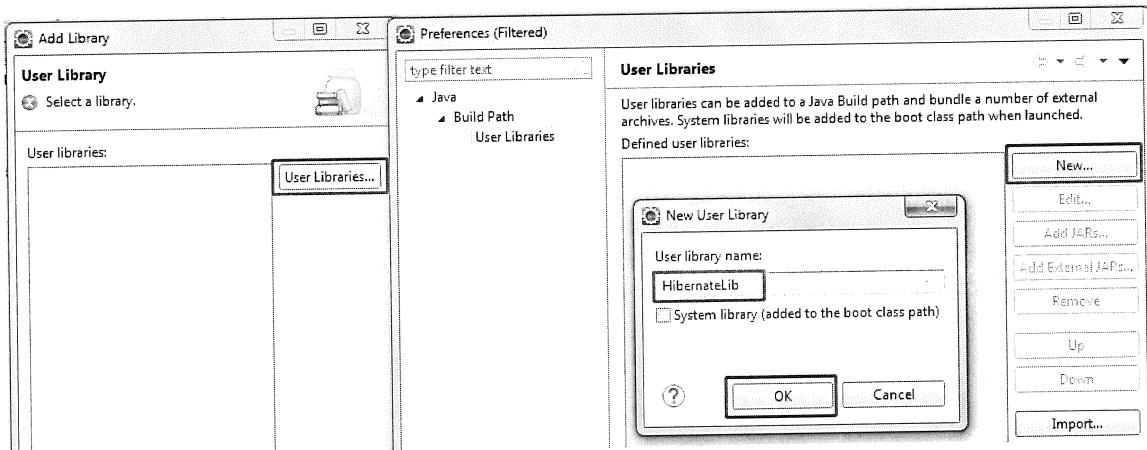


Figura 3.29. Nombrar la librería con los jar para Hibernate.

8. Seguidamente pulsamos el botón *Add External JARs* para localizar los ficheros JAR de nuestra carpeta *Hibernate*, los seleccionamos todos y pulsamos el botón *Abrir*. Se mostrará una pantalla similar a la mostrada en la Figura 3.30. Pulsamos el botón *OK* y a continuación el botón *Finish* para finalizar. En nuestro proyecto aparecerá la nueva librería

Con esto ya podemos crear el primer programa Java en nuestro proyecto que nos va a permitir comunicarnos con nuestra base de datos.

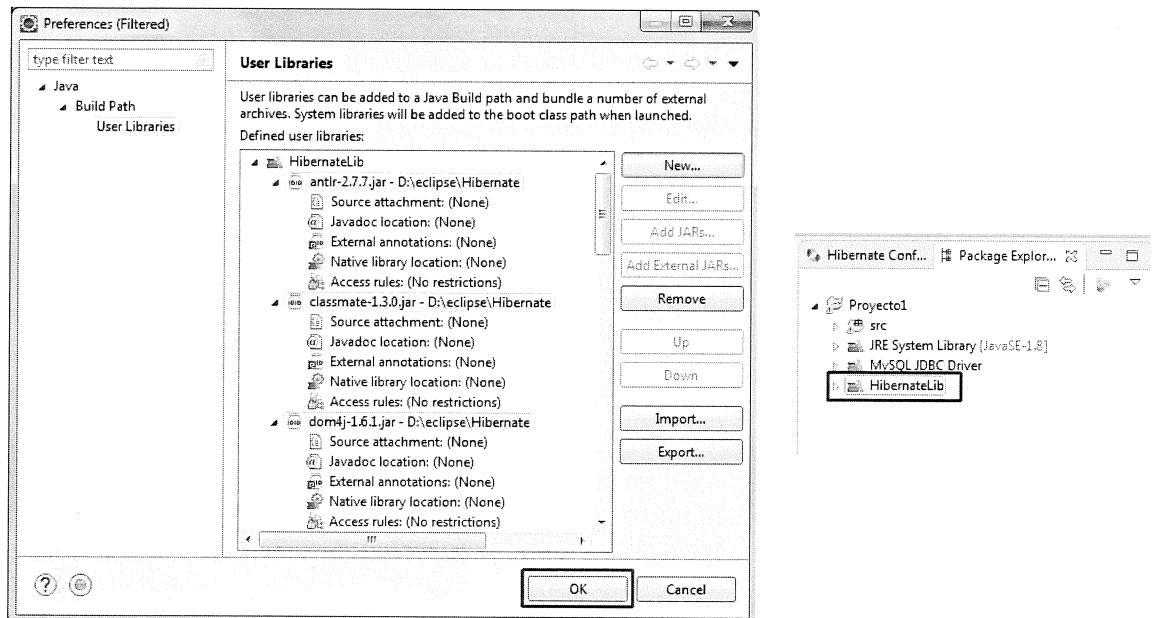


Figura 3.30. Librería *HibernateLib* con los JAR y en el proyecto Eclipse.

En primer lugar crearemos una instancia a la base de datos para poder trabajar con ella y que se utilizará a lo largo de toda la aplicación. Necesitaremos crear un **Singleton**.

El **Singleton** es un patrón de diseño diseñado para restringir la creación de objetos pertenecientes a una clase. Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella (así tenemos un único objeto creado de una clase).

El patrón **Singleton** se implementa creando en nuestra clase un método que crea una instancia del objeto, solo si todavía no existe alguna. Para asegurar que la clase no pueda ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

Nuestro **Singleton** será una clase de ayuda que accede a **SessionFactory** para obtener un objeto sesión, hay una única **SessionFactory** para toda la aplicación. En la clase se define una variable estática llamada *sessionFactory* que recoge el objeto **SessionFactory** devuelto por el método *buildSessionFactory()*; este objeto se crea a partir del fichero de configuración (*hibernate.cfg.xml*). El método *getSessionFactory()* devuelve el valor de la variable estática definida, o lo que es lo mismo, devuelve el objeto **SessionFactory** creado. El nombre de la clase es *HibernateUtil.java* y se incluirá en el paquete *primero* de nuestro proyecto, el código es el siguiente:

```
package primero;

import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory = buildSessionFactory();
```

```

private static SessionFactory buildSessionFactory() {
    try {
        //Create the SessionFactory from hibernate.cfg.xml
        return new Configuration().configure().buildSessionFactory(
            new StandardServiceRegistryBuilder().configure().build());
    }
    catch (Throwable ex) {
        // Make sure you log the exception, as it might be swallowed
        System.err.println("Initial SessionFactory creation failed." + ex);
        throw new ExceptionInInitializerError(ex);
    }
}

public static SessionFactory getSessionFactory() {
    return sessionFactory;
}
}//

```

Con esta clase podemos obtener la sesión actual desde cualquier parte de nuestra aplicación. Ahora pulsando con el botón derecho del ratón en nuestro proyecto creamos una clase de nombre *Main.java*. Esta se creará en el *default package* del proyecto, o en otro paquete que definamos. El método *main()* inserta una fila en la tabla *departamentos*. El código es el siguiente:

```

import primero.*;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class Main {
    public static void main(String[] args) {

        //obtener la sesión actual
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        //crear la sesión
        Session session = sesion.openSession();
        //crear una transacción de la sesión
        Transaction tx = session.beginTransaction();

        System.out.println("Inserto una fila en la tabla DEPARTAMENTOS.");

        Departamentos dep = new Departamentos();
        dep.setDeptNo((byte) 62);
        dep.setDnombre("MARKETING");
        dep.setLoc("GUADALAJARA");

        session.save(dep);
        tx.commit();
        session.close();

        System.exit(0);
    }
}

```

Con la siguiente línea `SessionFactory sesion = HibernateUtil.getSessionFactory();` se obtiene la sesión creada por el **Singleton**. Esta instrucción se usará a lo largo de todas las clases en las que deseemos realizar operaciones con nuestra base de datos. Antes de ejecutar la aplicación debemos editar el fichero **hibernate.cfg.xml** y cambiar la línea:

```
<session-factory name ="Miconexion">
```

Por la siguiente, donde se elimina el parámetro *name*:

```
<session-factory>
```

Ya que si no se mostrará el siguiente error en la ejecución:

```
WARN: HHH000277: Could not bind factory to JNDI
org.hibernate.engine.jndi.JndiException: Error parsing JNDI name
[Miconexion]
```

Durante la ejecución también aparecerán errores warnings del tipo: *WARN: HHH90000012: Recognized obsolete hibernate namespace http://hibernate.sourceforge.net/hibernate-configuration. Use namespace http://www.hibernate.org/dtd/hibernate-configuration instead. Support for obsolete DTD/XSD namespaces may be removed at any time.* Para eliminar dicho error hemos de cambiar las cabeceras DOCTYPE de los ficheros XML, se deben actualizar las URL de los namespaces. Cambiamos en el fichero **hibernate.cfg.xml** la línea:

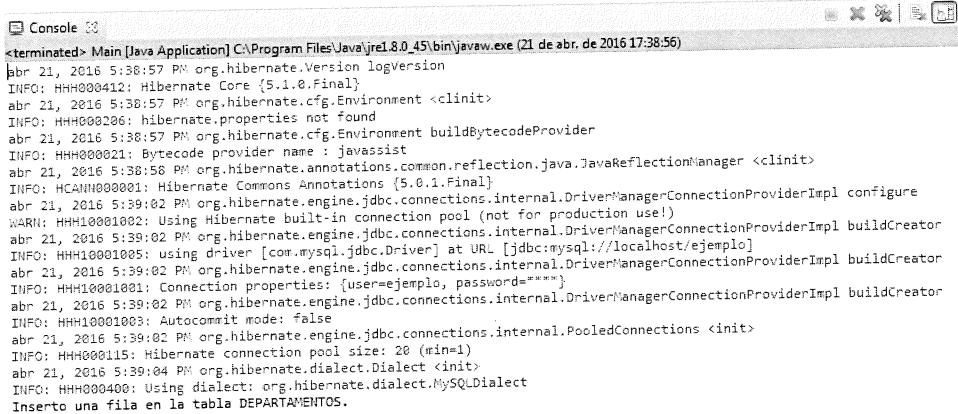
```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

Por esta otra:

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
```

En los ficheros de mapeo **Empleados.hbm.xml** y **Departamentos.hbm.xml** cambiamos la URL por la siguiente: <http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd>. Y en el fichero **hibernate.reveng.xml** escribimos la siguiente: <http://www.hibernate.org/dtd/hibernate-reverse-engineering-3.0.dtd>.

Cuando se cambia algún fichero de configuración hay que pulsar con el botón derecho en el nombre del proyecto y luego en la opción **Refresh** (o pulsar F5). La ejecución del programa mostraría en consola la imagen de la Figura 3.31.



```

Console 33
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (21 de abr. de 2016 17:38:56)
abr 21, 2016 5:38:57 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.1.0.Final}
abr 21, 2016 5:38:57 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
abr 21, 2016 5:38:57 PM org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
abr 21, 2016 5:38:58 PM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.jdbc.Driver] at URL [jdbc:mysql://localhost/empleo]
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.jdbc.Driver] at URL [jdbc:mysql://localhost/empleo]
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: Connection properties: {user=rejemplio, password=""}
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
abr 21, 2016 5:39:04 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
Inserto una fila en la tabla DEPARTAMENTOS.

```

Figura 3.31.Consola de la ejecución del programa.

El siguiente ejemplo inserta un empleado en la tabla *empleados* en el departamento 10, para el departamento será necesario crear un objeto de tipo *Departamentos* y asignar como número de departamento el valor 10, es lo que se hace en el método *setDeptNo()* de esta clase:

```

import primero.*;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class MainEmpleado {
    public static void main(String[] args) {
        //obtener la sesión actual
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        //crear la sesión
        Session session = sesion.openSession();
        //crear una transacción de la sesión
        Transaction tx = session.beginTransaction();

        System.out.println("Inserto un EMPLEADO EN EL DEPARTAMENTO 10.");
        Float salario = new Float(1500); // inicializo el salario
        Float comision = new Float(10); // inicializo la comisión

        Empleados em = new Empleados(); // creo un objeto empleados
        em.setEmpNo((short) 4455); // el número de empleado es 4455
        em.setApellido("PEPE"); // el nombre es PEPE
        em.setDir((short) 7499); // el director es el empleado 7499
        em.setOficio("VENDEDOR"); // el oficio es VENDEDOR

        em.setSalario(salario);
        em.setComision(comision);

        //se crea un objeto Departamentos para asignárselo al empleado
        Departamentos d = new Departamentos();
        d.setDeptNo((byte) 10); //el número de dep es 10
        em.setDepartamentos(d);

        //fecha de alta, calculamos fecha actual
    }
}

```

```

java.util.Date hoy = new java.util.Date();
java.sql.Date fecha = new java.sql.Date(hoy.getTime());
em.setFechaAlt(fecha);

session.save(em);
tx.commit();
session.close();
System.exit(0);
}
}

```

En los ejemplos anteriores hemos usado los siguientes métodos:

- ***save()***: Este método de la sesión (interface **Session**) lo usaremos para guardar el objeto, le pasamos como argumento el objeto a guardar: ***save(Object object)***.
- ***commit()***: Este método hace un commit de la transacción actual. La transacción se crea al método ***beginTransaction()*** de la sesión actual. Es necesario para que los datos se almacenen en la BD.
- ***close()***: Este método se utiliza para cerrar la sesión.

¡ ¡INTERESANTE !!

Documentación Java sobre Hibernate: <https://docs.jboss.org/hibernate/orm/current/javadocs/>

Referencia en español de Hibernate: <https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/>

Documentación sobre las distintas versiones de Hibernate incluyendo la actual: <https://docs.jboss.org/hibernate/orm/>

Si los ejemplos anteriores los ejecutamos más de una vez se producirán excepciones de error, ya que el departamento a insertar o el empleado a insertar existe. La excepción es la siguiente **org.hibernate.exception.ConstraintViolationException**, y se produce al hacer el commit en el método ***tx.commit()***. También se producirá error si al insertar un empleado creamos un objeto departamento que no exista, entonces, se produce la siguiente excepción: **org.hibernate.TransientPropertyValueException** esta vez sobre el método ***session.save(em)***. El siguiente código controlaría las excepciones de la existencia de un empleado y de la no existencia del departamento en el programa de inserción de un empleado:

```

try {
    session.save(em);
    try {
        tx.commit();
    } catch (ConstraintViolationException e) {
        System.out.println("EMPLEADO DUPLICADO");
        System.out.printf("MENSAJE: %s%n", e.getMessage());
        System.out.printf("COD ERROR: %d%n", e.getErrorCode());
        System.out.printf("ERROR SQL: %s%n",
                          e.getSQLException().getMessage());
    }
} catch (TransientPropertyValueException e) {
    System.out.println("EL DEPARTAMENTO NO EXISTE");
    System.out.printf("MENSAJE: %s%n", e.getMessage());
    System.out.printf("Propiedad: %s%n", e.getPropertyName());
}

```

```

} catch (Exception e) {
    System.out.println("ERROR NO CONTROLADO....");
    e.printStackTrace();
}
}

```

3.6. ESTRUCTURA DE LOS FICHEROS DE MAPEO

Hibernate utiliza unos ficheros de mapeo para relacionar las tablas de la base de datos con los objetos Java. Estos ficheros están en formato XML y tienen la extensión **.hbm.xml**. En el proyecto anterior se han creado los ficheros: **Empleados.hbm.xml** y **Departamentos.hbm.xml**, el primero asociado a la tabla de *empleados* y el segundo a la tabla *departamentos*. Estos ficheros se guardan en el mismo directorio que las clases Java *Empleados.java* y *Departamentos.java*. Estas clases representan un objeto *empleados* y un objeto *departamentos* respectivamente y todos los ficheros forman parte del paquete **primero**. La Figura 3.32 muestra la correspondencia entre los ficheros de mapeo y las clases generadas.

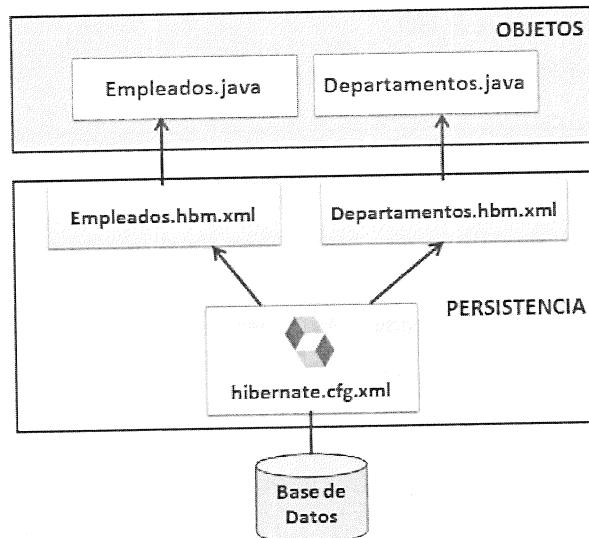


Figura 3.32. Ficheros de mapeo y clases Java.

La estructura del fichero **Departamentos.hbm.xml** es la siguiente:

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 21-abr-2016 9:48:48 by Hibernate Tools 3.4.0.CR1 -->
<hibernate-mapping>
    <class name="primero.Departamentos" table="departamentos"
          catalog="ejemplo">

        <id name="deptNo" type="byte">
            <column name="dept_no" />
            <generator class="assigned" />
        </id>
        <property name="dnombre" type="string">
            <column name="dnombre" length="15" />
        </property>
    </class>
</hibernate-mapping>

```

```

<property name="loc" type="string">
    <column name="loc" length="15" />
</property>
<set name="empleadoses" table="empleados" inverse="true"
      lazy="true" fetch="select">
    <key>
        <column name="dept_no" not-null="true" />
    </key>
    <one-to-many class="primero.Empleados" />
</set>
</class>
</hibernate-mapping>

```

Veamos el significado del contenido del fichero XML:

- **hibernate-mapping**: todos los ficheros de mapeo comienzan y acaban con esta etiqueta.
- **class**: esta etiqueta engloba a la clase con sus atributos, indicando siempre el mapeo a la tabla de la base de datos. En **name** se indica el nombre de la clase y en **table** el nombre de la tabla a la que representa este objeto, en **catalog** se indica el nombre de la base de datos:

```
<class name="primero.Departamentos" table="departamentos"
      catalog="ejemplo">
```

Dentro de **class** distinguimos la etiqueta **id** en la cual se indica en **name** el campo que representa al atributo clave en la clase y en **column** su nombre sobre la tabla, en **type** el tipo de datos. En **id** además tenemos la propiedad **generator** que indica la naturaleza del campo clave. En este caso es **assigned** porque es el usuario el que se encarga de asignar la clave. Si fuese **increment** indicaría que es un identificador autogenerado por la base de datos. Este atributo se correspondería con la columna *dept_no* (*dept_no* TINYINT(2) NOT NULL PRIMARY KEY) de la tabla *departamentos*:

```

<id name="deptNo" type="byte">
    <column name="dept_no" />
    <generator class="assigned" />
</id>

```

El resto de atributos se indican en las etiquetas **property** asociando el nombre del campo de la clase con el nombre de la columna de la tabla y el tipo de datos. La columna *dnombre* de la tabla *departamentos* (*dnombre* VARCHAR(15)) se define así:

```

<property name="dnombre" type="string">
    <column name="dnombre" length="15" />
</property>

```

La columna *loc* de la tabla se declara de forma similar:

```

<property name="loc" type="string">
    <column name="loc" length="15" />
</property>

```

La etiqueta **set** se utiliza para mapear colecciones. Dentro de **set** se definen varios atributos. En **name** se indica el nombre del atributo generado, el nombre de la tabla de donde se tomará la colección se declara con el atributo **table**. En el elemento **key** se define el nombre de la columna identificadora en la asociación, en este caso la columna *dept_no* de la tabla *departamentos*. El elemento **one-to-many** define la relación, en este caso es una asociación *uno-a-muchos*, es decir, un departamento puede tener muchos empleados. En **class** se indica de qué tipo son los elementos de la colección. Resumiendo, este mapeo indica que la clase *Departamentos.java* tiene un atributo de nombre **empleadoses** que es una lista de instancias de la clase **primero.Empleados**:

```
<set name="empleadoses" table="empleados" inverse="true"
      lazy="true" fetch="select">
    <key>
      <column name="dept_no" not-null="true" />
    </key>
    <one-to-many class="primero.Empleados" />
</set>
```

Los tipos que declaramos y utilizamos en los ficheros de mapeo no son tipos de datos Java. Tampoco son tipos de base de datos SQL. Estos tipos se llaman **tipos de mapeo Hibernate**, convertidores que pueden traducir de tipos de datos de Java a SQL y viceversa. De nuevo, Hibernate tratará de determinar el tipo correcto de conversión y de mapeo por sí mismo.

La estructura del fichero **Empleados.hbm.xml** es la siguiente:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 21-abr-2016 9:48:48 by Hibernate Tools 3.4.0.CR1 -->
<hibernate-mapping>
  <class name="primero.Empleados" table="empleados"
        catalog="ejemplo">
    <id name="empNo" type="short">
      <column name="emp_no" />
      <generator class="assigned" />
    </id>
    <many-to-one name="departamentos" class="primero.Departamentos"
                  fetch="select">
      <column name="dept_no" not-null="true" />
    </many-to-one>
    <property name="apellido" type="string">
      <column name="apellido" length="10" />
    </property>
    <property name="oficio" type="string">
      <column name="oficio" length="10" />
    </property>
    <property name="dir" type="java.lang.Short">
      <column name="dir" />
    </property>
    <property name="fechaAlt" type="date">
      <column name="fecha_alt" length="10" />
    </property>
    <property name="salario" type="java.lang.Float">
      <column name="salario" precision="6" />
    </property>
```

```

</property>
<property name="comision" type="java.lang.Float">
    <column name="comision" precision="6" />
</property>
</class>
</hibernate-mapping>

```

En este fichero nos encontramos con la relación **many-to-one** es una asociación unidireccional **muchos-a-uno** (una clave ajena en una tabla referencia la columna (o columnas) de la clave primaria de la tabla destino); se utiliza para definir una relación muchos a uno entre las dos clases Java (es decir, muchos empleados pertenecen a un departamento). En el atributo **name** se indica el nombre del atributo en la clase Java y en **class** se indica la clase a la que referencia. En el elemento **column name** se indica el nombre de la columna de la tabla *empleados* (en este caso es *dept_no*). El mapeo indica que la clase *Empleados.java* tiene un atributo de nombre **departamentos** que es una instancia de la clase **primero.Departamentos**:

```

<many-to-one name="departamentos" class="primero.Departamentos"
    fetch="select">
    <column name="dept_no" not-null="true" />
</many-to-one>

```

El atributo **fetch** (por defecto es *select*) escoge entre la recuperación de unión exterior (outer-join) o la recuperación por selección secuencial.

3.7. CLASES PERSISTENTES

Hemos visto que entre las etiquetas **<hibernate-mapping>** **</hibernate-mapping>** de los ficheros XML se incluye un elemento **class** que hace referencia a una clase:

```

<class name="primero.Departamentos" table="departamentos"
    catalog="ejemplo">

<class name="primero.Empleados" table="empleados" catalog="ejemplo">

```

En nuestro proyecto se han generado las clases *Empleados.java* y *Departamentos.java*. A estas clases se les llama **clases persistentes**, son las clases que implementan las entidades del problema, deben implementar la interfaz **Serializable**. Equivalen a una tabla de la base de datos, y un registro o fila es un objeto persistente de esa clase. Estas clases representan un objeto *empleados* y un objeto *departamentos*, por lo tanto, podemos crear objetos empleados y departamentos a partir de ellas. Tienen unos atributos y unos métodos *get* y *set* para acceder a los mismos.

Utilizan convenciones de nombrado estándares de JavaBean para los métodos de propiedades *getter* y *setter* así como también visibilidad privada para los campos. Al ser los atributos de los objetos privados se crean métodos públicos para retornar un valor de un atributo, método *getter*, o para cargar un valor a un atributo, método *setter*, por ejemplo, el método *getDnombre()* devuelve el nombre de un departamento (atributo *dnombre*) y el método *setDnombre()* carga un valor en el atributo *dnombre*. A estas reglas también se las llama modelo de programación **POJO** (*Plain Old Java Object*).

Para completar el nombre de un método *getter* o *setter*, solo hay que poner la primera letra que los une en mayúsculas. Si nos fijamos en el fichero **Departamentos.hbm.xml**, el elemento **id** es la declaración de la propiedad identificadora, el atributo de mapeo *name="deptNo"* declara el nombre de la propiedad JavaBean y le dice a Hibernate que utilice los métodos *getDeptNo()* y *setDeptNo()* para acceder a la propiedad:

```
<id name="deptNo" type="byte">
```

Al igual que con el elemento **id**, el atributo **name** del elemento **property** le dice a Hibernate qué métodos *getter* y *setter* utilizar. Así que en este caso, Hibernate buscará los métodos *getDnombre()*, *setDnombre()*, *getLoc()* y *setLoc()*:

```
<property name="dnombre" type="string">
<property name="loc" type="string">
```

La clase *Departamentos.java* es la siguiente:

```
package primero;
// Generated 21-abr-2016 9:48:48 by Hibernate Tools 3.4.0.CR1

import java.util.HashSet;
import java.util.Set;

/**
 * Departamentos generated by hbm2java
 */
public class Departamentos implements java.io.Serializable {

    private byte deptNo;
    private String dnombre;
    private String loc;
    private Set<Empleados> empleadoses = new HashSet<Empleados>(0);

    public Departamentos() {
    }

    public Departamentos(byte deptNo) {
        this.deptNo = deptNo;
    }

    public Departamentos(byte deptNo, String dnombre,
                         String loc, Set<Empleados> empleadoses) {
        this.deptNo = deptNo;
        this.dnombre = dnombre;
        this.loc = loc;
        this.empleadoses = empleadoses;
    }

    public byte getDeptNo() {
        return this.deptNo;
    }

    public void setDeptNo(byte deptNo) {
        this.deptNo = deptNo;
    }
}
```

```

public String getDnombre() {
    return this.dnombre;
}

public void setDnombre(String dnombre) {
    this.dnombre = dnombre;
}

public String getLoc() {
    return this.loc;
}

public void setLoc(String loc) {
    this.loc = loc;
}

public Set<Empleados> getEmpleadoses() {
    return this.empleadoses;
}

public void setEmpleadoses(Set<Empleados> empleadoses) {
    this.empleadoses = empleadoses;
}
}

```

ACTIVIDAD 3.3

Realiza el mapeo de las tablas de PRODUCTOS, CLIENTES y VENTAS del capítulo anterior y estudia los ficheros de mapeo y las clases generadas.

3.8. SESIONES Y OBJETOS HIBERNATE

Para poder utilizar los mecanismos de persistencia de Hibernate se debe inicializar el entorno Hibernate y obtener un objeto **Session** utilizando la clase **SessionFactory** de Hibernate. El siguiente fragmento de código ilustra este proceso:

```

// Inicializa el entorno Hibernate
Configuration cfg = new Configuration().configure();

// Crea el ejemplar de SessionFactory
SessionFactory sessionFactory = cfg.buildSessionFactory(
    new StandardServiceRegistryBuilder().configure().build() );

// Obtiene un objeto Session
Session session = sessionFactory.openSession();

```

La llamada a **Configuration().configure()** carga el fichero de configuración **hibernate.cfg.xml** e inicializa el entorno de Hibernate. Se necesita crear un objeto del tipo **StandardServiceRegistry** que contiene la lista de servicios que utiliza Hibernate para crear el ejemplar de **SessionFactory**; este normalmente solo se crea una vez y se utiliza para crear todas las sesiones relacionadas con un contexto dado. Es lo que se hizo en el *Proyecto1* al crear la clase **HibernateUtil.java**, para crear una vez el ejemplar de **SessionFactory**.

3.8.1. Transacciones

Un objeto **Session** de Hibernate representa una única unidad de trabajo para un almacén de datos dado y lo abre un ejemplar de **SessionFactory**. Al crear la sesión se crea la transacción para dicha sesión. Se deben cerrar las sesiones cuando se haya completado todo el trabajo de una transacción. El siguiente código ilustra una sesión de persistencia de Hibernate:

```
Session session = sesion.openSession();           //crea la sesión
Transaction tx = session.beginTransaction(); //crea la transacción
//Código de persistencia
.
.
.
tx.commit();          //valida la transacción
session.close(); //finaliza la sesión
```

El método ***beginTransaction()*** marca el comienzo de una transacción. El método ***commit()*** valida una transacción, y ***rollback()*** deshace la transacción.

3.8.2. Estados de un objeto Hibernate

Hibernate define y soporta los siguientes estados de objeto:

- **Transitorio (Transient)**: Un objeto es transitorio si ha sido recién instanciado utilizando el operador *new*, y no está asociado a una **Session** de Hibernate. No tiene una representación persistente en la base de datos y no se le ha asignado un valor identificador. Las instancias transitorias serán destruidas por el recolector de basura si la aplicación no mantiene más una referencia. Utiliza la **Session** de Hibernate para hacer un objeto persistente (y deja que Hibernate se ocupe de las declaraciones SQL que necesitan ejecutarse para esta transición). Las instancias recién instanciadas de una clase persistente, Hibernate las considera como **transitorias**. Podemos hacer una instancia **transitoria persistente** asociándola con una sesión:

```
//Inserto el departamento 60 en la tabla DEPARTAMENTOS
Departamentos dep = new Departamentos();
dep.setDeptNo((byte) 60);
dep.setDnombre("MARKETING");
dep.setLoc("GUADALAJARA");
session.save(dep); //save() hace que la instancia sea persistente
```

- **Persistente (Persistent)**. Un objeto estará en este estado cuando ya está almacenado en la base de datos. Puede haber sido guardado o cargado, sin embargo, por definición, se encuentra en el ámbito de una **Session**. Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo. En definitiva, los objetos transitorios solo existen en memoria y no en un almacén de datos, han sido instanciados por el desarrollador sin haberlos almacenado mediante una sesión. Los persistentes se caracterizan por haber sido ya creados y almacenados en una sesión o bien devueltos en una consulta realizada con la sesión.
- **Separado (Detached)**. Un objeto está en este estado cuando cerramos la sesión mediante el método ***close()*** de **Session**. Una instancia separada es un objeto que se ha hecho persistente, pero su sesión ha sido cerrada. La referencia al objeto todavía es

válida, por supuesto, y la instancia separada podría incluso ser modificada en este estado. Una instancia separada puede ser asociada a una nueva **Session** más tarde, haciéndola persistente de nuevo (con todas las modificaciones).

Fuente: <http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/objectstate.html>

3.8.3. Carga de objetos

Para la carga de objetos usaremos los siguientes métodos de **Session**:

MÉTODO	DESCRIPCIÓN
<T> T load(Class<T> Clase, Serializable id)	Devuelve la instancia persistente de la clase indicada con el identificador dado. La instancia tiene que existir, si no existe el método lanza una excepción
Object load(String nombreClase, Serializable id)	Similar al método anterior, pero en este caso indicamos en el primer parámetro el nombre de la clase en formato de <i>String</i>
<T> T get(Class<T> Clase, Serializable id)	Devuelve la instancia persistente de la clase indicada con el identificador dado. Si la instancia no existe, devuelve <i>null</i>
Object get(String nombreClase, Serializable id)	Similar al método anterior, pero en este caso indicamos en el primer parámetro el nombre de la clase

El siguiente ejemplo utiliza el método *load()* para obtener los datos del departamento 20. En el primer parámetro se indica la clase *Departamentos* y en el segundo el número de departamento que se quiere recuperar, se hace un *cast* para convertirlo al tipo de dato definido en el atributo identificador de la clase (*deptNo*) que es *byte*. Si la fila no existe se lanza la excepción **ObjectNotFoundException**:

```
// Visualiza los datos del departamento 20
Departamentos dep = new Departamentos();
try {
    dep = (Departamentos) session.load(Departamentos.class, (byte) 20);
    System.out.printf("Nombre Dep: %s%n", dep.getDnombre());
    System.out.printf("Localidad: %s%n", dep.getLoc());
} catch (ObjectNotFoundException o) {
    System.out.println("NO EXISTE EL DEPARTAMENTO!!!");
}
```

Usando el segundo formato del método *load()* la obtención del departamento quedaría así:

```
dep = (Departamentos) session.load("primero.Departamentos", (byte) 20);
```

El método *load()* lanza la excepción **ObjectNotFoundException** si la fila no existe. Si no tenemos la certeza de que la fila exista debemos utilizar el método *get()*, que llama a la base de datos inmediatamente y devuelve *null* si no existe la fila correspondiente, el siguiente ejemplo comprueba si el departamento 11 existe. Si existe visualiza sus datos, y si no existe visualiza un mensaje indicándolo:

```
Departamentos dep = (Departamentos)
    session.get(Departamentos.class, (byte) 11);
if (dep==null) {
    System.out.println("El departamento no existe");
}
else
{
    System.out.printf("Nombre Dep: %s%n", dep.getDnombre());
    System.out.printf("Localidad: %s%n", dep.getLoc());
}
```

ACTIVIDAD 3.4

Visualiza los datos del empleado con número: 7369.

El siguiente ejemplo obtiene los datos del departamento 10 y el APELLIDO y SALARIO de sus empleados, para obtener los empleados usamos el método **getEmpleados()** de la clase *Departamentos* y usamos un **Iterator** para recorrer la lista de empleados.

```
import java.util.Iterator;
import java.util.Set;

import primero.*;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

public class ListadoDep {
    public static void main(String[] args) {
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        Session session = sesion.openSession();

        System.out.println("=====");
        System.out.println("DATOS DEL DEPARTAMENTO 10.");

        Departamentos dep = new Departamentos();
        dep = (Departamentos) session.load(Departamentos.class,
                                         (byte) 10);
        System.out.println("Nombre Dep:" + dep.getDnombre());
        System.out.println("Localidad:" + dep.getLoc());

        System.out.println("=====");
        System.out.println("EMPLEADOS DEL DEPARTAMENTO 10.");

        // obtenemos empleados
        Set<Empleados> listaemple = dep.getEmpleados();
        Iterator<Empleados> it = listaemple.iterator();

        System.out.printf("Número de empleados: %d %n",
                          listaemple.size());
        while (it.hasNext()) {
            Empleados emple = it.next();
            System.out.printf("%s * %.2f %n",
                            emple.getApellido(), emple.getSalario());
        }
    }
}
```

```

        System.out.println("=====");
        session.close();
        System.exit(0);
    }
}

```

La ejecución muestra la siguiente salida:

```

=====
DATOS DEL DEPARTAMENTO 10.
Nombre Dep:CONTABILIDAD
Localidad:SEVILLA
=====
EMPLEADOS DEL DEPARTAMENTO 10.
Número de empleados: 4
REY * 4100,00
MUÑOZ * 1690,00
PEPE * 1500,00
CEREZO * 2885,00
=====
```

¡ ¡INTERESANTE !!

Consulta la API de Hibernate: <https://docs.jboss.org/hibernate/orm/current/javadocs/>

ACTIVIDAD 3.5

A partir de las tablas mapeadas de PRODUCTOS, CLIENTES y VENTAS obtén un listado de ventas de un cliente. El cliente se obtendrá como un argumento de *main()*. La información a visualizar es similar a la mostrada en el Ejercicio 7 del Capítulo anterior.

3.8.4. Almacenamiento, modificación y borrado de objetos

Para almacenamiento, modificación y borrado de objetos usamos los siguientes métodos **Session**:

MÉTODO	DESCRIPCIÓN
Serializable save(Object obj)	Guarda el objeto que se pasa como argumento en la base de datos. Hace que la instancia transitoria del objeto sea persistente.
void update(Object objeto)	Actualiza en la base de datos el objeto que se pasa como argumento. El objeto a modificar debe ser cargado con el método <i>load()</i> o <i>get()</i>
void delete(Object objeto)	Elimina de la base de datos el objeto que se pasa como argumento. El objeto a eliminar debe ser cargado con el método <i>load()</i> o <i>get()</i>

El siguiente ejemplo crea un nuevo objeto *Departamentos* y lo almacena en la base de datos usando el método *save()*. Hibernate se encarga de SQL y ejecuta un INSERT en la base de datos:

```
Departamentos dep = new Departamentos();
dep.setDeptNo((byte) 70);
dep.setDnombre("INFORMÁTICA");
dep.setLoc("TOLEDO");
session.save(dep); //almacena el objeto
```

Para realizar el borrado de un objeto, primero debe ser cargado con el método ***load()*** o el método ***get()*** y a continuación podemos borrarlo con el método ***delete()***, hemos de asegurarnos de que no existan referencias de otros objetos al que se va a borrar, de lo contrario se producirá una excepción. El siguiente ejemplo borra el empleado cuyo número de empleado (*empNo*) es 7369:

```
Empleados em = new Empleados();
em = (Empleados) session.load(Empleados.class, (short)7369);
session.delete(em); //elimina el objeto
```

El siguiente ejemplo muestra la eliminación de un departamento controlando distintas excepciones, que el departamento no exista y que tenga empleados. La ejecución mostrará el mensaje: *NO SE PUEDE ELIMINAR, TIENE EMPLEADOS*:

```
import org.hibernate.ObjectNotFoundException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.exception.ConstraintViolationException;

import primero.Departamentos;
import primero.HibernateUtil;

public class BorradoDep {
    public static void main(String[] args) {
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        Session session = sesion.openSession();
        Transaction tx = session.beginTransaction();

        //DEPARTAMENTO A ELIMINAR
        Departamentos de = (Departamentos)
            session.load(Departamentos.class, (byte) 10);
        try {
            session.delete(de); // elimina el objeto
            tx.commit();
            System.out.println("Departamento eliminado...");
        } catch (ObjectNotFoundException o) {
            System.out.println("NO EXISTE EL DEPARTAMENTO...");
        } catch (ConstraintViolationException c) {
            System.out.println("NO SE PUEDE ELIMINAR, TIENE EMPLEADOS");
        } catch (Exception e) {
            System.out.println("ERROR NO CONTROLADO....");
            e.printStackTrace();
        }
        session.close();
        System.exit(0);
    }
}
```

Para modificar un objeto, igual que para borrarlo, primero hemos de cargarlo, a continuación realizamos las modificaciones con los métodos *setter*, y por último, utilizamos el método *update()* para modificarlo. El siguiente ejemplo modifica el salario y el departamento del empleado 7369, sumamos 1000 al salario y le asignamos el departamento 30:

```
Empleados em = new Empleados();
try {
    em = (Empleados) session.load(Empleados.class, (short) 7369);
    System.out.printf("Modificación empleado: %d%n", em.getEmpNo());
    System.out.printf("Salario antiguo: %.2f%n", em.getSalario());
    System.out.printf("Departamento antiguo: %s%n",
                      em.getDepartamentos().getDnombre());
    //nuevo salario
    float NuevoSalario = em.getSalario() + 1000;
    em.setSalario(NuevoSalario);

    //nuevo departamento
    Departamentos dep = (Departamentos)
        session.get(Departamentos.class, (byte) 30);
    if (dep == null) {
        System.out.println("El departamento NO existe");
    } else {
        em.setDepartamentos(dep);
        session.update(em); // modifica el objeto
        tx.commit();
        System.out.printf("Salario nuevo: %.2f%n", em.getSalario());
        System.out.printf("Departamento nuevo: %d%n",
                          em.getDepartamentos().getDnombre());
    }
}

} catch (ObjectNotFoundException o) {
    System.out.println("NO EXISTE EL EMPLEADO...");
} catch (ConstraintViolationException c) {
    System.out.println("NO SE PUEDE ASIGNAR UN DEPARTAMENTO
QUE NO EXISTE....");
} catch (Exception e) {
    System.out.println("ERROR NO CONTROLADO....");
    e.printStackTrace();
}
```

ACTIVIDAD 3.6

Sube el salario a todos los empleados del departamento 10. El salario se recibe como argumento de *main()*. Muestra el apellido y salario del empleado antes y después de actualizar.

Partimos de las tablas mapeadas de PRODUCTOS, CLIENTES y VENTAS. Realiza un programa Java para insertar varios productos, otro para insertar varios clientes y un tercero para insertar varias ventas a un cliente controlando que el cliente exista, el producto exista y tenga stock (*stockactual – cantidad >= stockminimo*) y la venta no esté duplicada.

3.9. CONSULTAS

Hibernate soporta un lenguaje de consulta orientado a objetos denominado **HQL (Hibernate Query Language)** fácil de usar pero potente a la vez. Este lenguaje es una extensión orientada a objetos de SQL. Las consultas HQL y SQL nativas son representadas con una instancia de `org.hibernate.Query`. Esta interfaz ofrece métodos para ligar parámetros, manejo del conjunto resultado, y para la ejecución de la consulta real. Siempre obtiene una `Query` utilizando el objeto `Session` actual. Algunos métodos importantes de esta interfaz son los siguientes:

MÉTODO	DESCRIPCIÓN
<code>Iterator iterate()</code>	Devuelve en un objeto <code>Iterator</code> el resultado de la consulta
<code>List list()</code>	Devuelve el resultado de la consulta en un <code>List</code>
<code>Query setFetchSize (int size)</code>	Fija el número de resultados a recuperar en cada acceso a la base de datos al valor indicado en <code>size</code>
<code>int executeUpdate()</code>	Ejecuta la sentencia de modificación o borrado. Devuelve el número de entidades afectadas
<code>String getQueryString()</code>	Devuelve la consulta en un <code>String</code>
<code>Object uniqueResult()</code>	Devuelve un objeto (cuando sabemos que la consulta devuelve un objeto) o nulo si la consulta no devuelve resultados
<code>Query setCharacter(int posición, char valor)</code>	Asigna el <i>valor</i> indicado en el método a un parámetro de tipo CHAR <i>posición</i> , indica la posición del parámetro dentro de la consulta, empieza en 0
<code>Query setCharacter(String nombre, char valor)</code>	<i>nombre</i> es el nombre (se indica como <code>:nombre</code>) del parámetro dentro de la consulta
<code>Query setDate(int posición, Date fecha)</code> <code>Query setDate(String nombre, Date fecha)</code>	Asigna la <i>fecha</i> a un parámetro de tipo DATE
<code>Query setDouble(int posición, double valor)</code> <code>Query setDouble(String nombre, double valor)</code>	Asigna <i>valor</i> a un parámetro de tipo decimal (en MySQL tipo FLOAT)
<code>Query setInteger(int posición, int valor)</code> <code>Query setInteger(String nombre, int valor)</code>	Asigna <i>valor</i> a un parámetro de tipo entero
<code>Query setString(int posición, String valor)</code> <code>Query setString(String nombre, String valor)</code>	Asigna <i>valor</i> a un parámetro de tipo VARCHAR
<code>Query setParameterList(String nombre, Collection valores)</code>	Asigna una colección de valores al parámetro cuyo nombre se indica en <i>nombre</i>
<code>Query setParameter(int posición, Object valor)</code>	Asigna el <i>valor</i> al parámetro indicado en <i>posición</i>
<code>Query setParameter(String nombre, Object valor)</code>	Asigna el <i>valor</i> al parámetro indicado en <i>nombre</i>
<code>int executeUpdate()</code>	Ejecuta una sentencia UPDATE o DELETE, devuelve el número de entidades afectadas por la operación
Consulta la API de Hibernate: https://docs.jboss.org/hibernate/orm/current/javadocs/	

Para realizar una consulta usaremos el método `createQuery()` de la interface `SharedSessionContract`, se le pasará en un `String` la consulta HQL:

```
Query createQuery(String queryString)
```

Por ejemplo, para hacer una consulta sobre la tabla `departamentos`, mapeada con la clase `Departamentos` se escribe lo siguiente:

```
Query q = session.createQuery("from Departamentos");
```

Para recuperar los datos de la consulta usaremos el método `list()`:

```
List <Departamentos> lista = q.list();
```

O el método `iterate()`:

```
Iterator iter = q.iterate();
```

El método `list()` devuelve en una colección todos los resultados de la consulta, en la colección se encuentran instanciadas todas las entidades que corresponden al resultado de la ejecución de la consulta. Este método realiza una única comunicación con la base de datos en donde se traen todos los resultados y requiere que haya memoria suficiente para almacenar todos los objetos resultantes de la consulta. Si la cantidad de resultados es extensa, el retraso del acceso a la base de datos será notorio.

El método `iterate()` devuelve un iterador Java para recuperar los resultados de la consulta. En este caso Hibernate ejecuta la consulta obteniendo solo los ids de las entidades, y en cada llamada al método `Iterator.next()` ejecuta la consulta propia para obtener la entidad completa. Esto implica una mayor cantidad de accesos a la base de datos y, por tanto, mayor tiempo de procesamiento total. La ventaja de este método es que no se requiere que todas las entidades estén cargadas en memoria simultáneamente. Se puede utilizar el método `setFetchSize()` para fijar la cantidad de resultados a recuperar en cada acceso a la base de datos, de esta manera no se hará un acceso a la base de datos en cada llamada al método `Iterator.next()`. El siguiente ejemplo obtiene 10 filas en cada acceso a la base de datos:

```
q.setFetchSize(10);
```

El siguiente ejemplo realiza una consulta de todas las filas de la tabla `departamentos`:

```
import java.util.Iterator;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

import primero.Departamentos;
import primero.HibernateUtil;

public class ListaDepartamentos {
    public static void main(String[] args) {
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        Session session = sesion.openSession();
```

```

Query q = session.createQuery("from Departamentos");
List <Departamentos> lista = q.list();

// Obtenemos un Iterador y recorremos la lista.
Iterator <Departamentos> iter = lista.iterator();
System.out.printf("Número de departamentos: %d%n", lista.size());

while (iter.hasNext())
{
    //extraer el objeto
    Departamentos depar = (Departamentos) iter.next();
    System.out.printf("%d, %s%n",
                      depar.getDeptNo(), depar.getDnombre());
}
session.close();
System.exit(0);
}
}

```

Visualiza la siguiente información:

```

Número de departamentos: 5
10, CONTABILIDAD
20, INVESTIGACIÓN
30, VENTAS
40, PRODUCCIÓN
60, MARKETING

```

El ejemplo con el método *iterate()* quedaría así:

```

Query q = session.createQuery("from Departamentos");
q.setFetchSize(10);
Iterator iter = q.iterate();

while (iter.hasNext())
{
    Departamentos depar = (Departamentos) iter.next();
    System.out.printf("%d, %s%n", depar.getDeptNo(), depar.getDnombre());
}

```

El ejemplo siguiente visualiza el apellido y salario de los empleados del departamento 20:

```

Query q = session.createQuery
        ("from Empleados as e where e.departamentos.deptNo = 20");
List<Empleados> lista = q.list();

Iterator<Empleados> iter = lista.iterator();
while (iter.hasNext()) {
    Empleados emp = (Empleados) iter.next(); // extraer el objeto
    System.out.printf("%s, %.2f %n",
                      emp.getApellido(), emp.getSalario());
}

```

El método ***uniqueResult()*** ofrece un atajo si sabemos que la consulta devolverá un objeto. Los siguientes ejemplos obtienen los datos de un único departamento, el primero visualiza los datos del departamento 10 y el segundo los datos del departamento con nombre VENTAS:

```
//Visualiza los datos del departamento 10
String hql = "from Departamentos as dep where dep.deptNo = 10";
Query q = session.createQuery(hql);

Departamentos dep = (Departamentos) q.uniqueResult();
System.out.printf("%d, %s, %s%n", dep.getDeptNo(),
                  dep.getLoc(), dep.getDnombre());

//Visualiza los datos del departamento de nombre VENTAS
hql = "from Departamentos as dep where dep.dnombre = 'VENTAS' ";
q = session.createQuery(hql);

dep = (Departamentos) q.uniqueResult();
System.out.printf("%d, %s, %s%n", dep.getDeptNo(),
                  dep.getLoc(), dep.getDnombre());
```

ACTIVIDAD 3.7

Realiza una consulta con ***createQuery()*** para obtener los datos del departamento 20 y visualiza también el apellido de sus empleados.

Partimos de las tablas mapeadas de PRODUCTOS, CLIENTES y VENTAS. Realiza un programa Java que muestre la misma información que el Ejercicio 7 del Capítulo 2 (usa una consulta HQL para obtener la información).

3.9.1. Parámetros en las consultas

Hibernate soporta parámetros con nombre y parámetros de estilo JDBC (?) en las consultas HQL. Los parámetros con nombre son identificadores de la forma **:nombre** en la cadena de consulta. Hibernate numera los parámetros desde cero, el primero que aparece estará en la posición 0, el siguiente en la posición 1, y así sucesivamente. Las ventajas de los parámetros con nombre son las siguientes:

- son insensibles al orden en que aparecen en la cadena de consulta,
- pueden aparecer múltiples veces en la misma petición,
- son autodocumentados.

Para asignar valores a los parámetros se utilizan los métodos ***setXXX*** vistos en la tabla anterior. La sintaxis más simple de utilizar estos parámetros es usando el método ***setParameter()***, por ejemplo, la siguiente consulta utiliza el parámetro nombrado **:numemple** y muestra el apellido y oficio del empleado con número 7369:

```
String hql = "from Empleados where empNo = :numemple";
Query q = session.createQuery(hql);

q.setParameter("numemple", (short) 7369);
Empleados emple = (Empleados) q.uniqueResult();
System.out.printf("%s, %s %n", emple.getApellido(), emple.getOficio());
```

El siguiente ejemplo consulta los empleados cuyo número de departamento es 10 y el oficio DIRECTOR:

```
String hql2 = "from Empleados emp where
    emp.departamentos.deptNo = :ndep and emp.oficio = :ofi";
Query q = session.createQuery(hql2);

q.setParameter("ndep", (byte) 10);
q.setParameter("ofi", "DIRECTOR");
List <Empleados> lista = q.list();
```

El mismo ejemplo con parámetros posicionales de estilo JDBC (el uso de estos parámetros se considera obsoleto por lo que se recomienda usar los parámetros nombrados) quedaría así:

```
String hql = "from Empleados emp
    where emp.departamentos.deptNo = ? and emp.oficio = ?";
Query q = session.createQuery(hql);
q.setParameter(0, (byte) 10);
q.setParameter(1, "DIRECTOR");
```

Para asignar valor a los parámetros también podemos usar los demás métodos *setXXX*, por ejemplo, para dar valor al número de departamento usamos el método *setInteger()* y para el oficio *setString()*:

```
q.setInteger("ndep", (byte) 10);
q.setString("ofi", "DIRECTOR");
```

El siguiente ejemplo usa el método *setDate()* para asignar valor a un parámetro nombrado de tipo *Date*. Obtiene los empleados cuya fecha de alta es 1991-12-03:

```
SimpleDateFormat formatoDelTexto = new SimpleDateFormat("yyyy-MM-dd");
String strFecha = "1991-12-03";
Date fecha = null;
try {
    fecha = formatoDelTexto.parse(strFecha);
} catch (ParseException ex) {
    ex.printStackTrace();
}
String hql = "from Empleados where fechaAlt = :fechalta";

Query q = session.createQuery(hql);
q.setDate("fechalta", fecha);
```

El siguiente ejemplo asigna a un parámetro nombrado llamado *:listadep* una colección de valores llamada *numeros* con los valores 10 y 20 para obtener aquellos empleados cuyo número de departamento sea 10 o 20; se usa el método *setParamererList()*:

```
List <Byte> numeros = new ArrayList <Byte> ();
numeros.add((byte)10);
numeros.add((byte)20);

String hql = "from Empleados emp
    where emp.departamentos.deptNo in (:listadep)
    order by emp.departamentos.deptNo ";
```

```
Query q = session.createQuery(hql);
q.setParameterList("listadep", numeros);
```

3.9.2. Consultas sobre clases no asociadas

Si queremos recuperar los datos de una consulta en la que intervienen varias tablas y no tenemos asociada a ninguna clase los atributos que devuelve esa consulta podemos utilizar la clase **Object**. Los resultados se reciben en un array de objetos, donde el primer elemento del array se corresponde con la primera clase que ponemos a la derecha de FROM, el siguiente elemento con la siguiente clase y así sucesivamente. El siguiente ejemplo realiza una consulta para obtener los datos de los empleados y de sus departamentos. El resultado de la consulta se recibe en un array de objetos, donde el primer elemento del array pertenece a la clase *Empleados* y el segundo a la clase *Departamentos*:

```
String hql="from Empleados e, Departamentos d
           where e.departamentos.deptNo = d.deptNo order by apellido";
Query cons = session.createQuery(hql);
Iterator q = cons.iterate();
while (q.hasNext()) {
    Object[] par =(Object[]) q.next();
    Empleados em = (Empleados) par[0];
    Departamentos de = (Departamentos) par[1];
    System.out.printf( "%s, %.2f, %s, %s %n",
                       em.getApellido(), em.getSalario(), de.getDnombre(), de.getLoc());
}
```

Para estos casos se puede crear una vista a partir de las tablas y realizar el mapeo de la vista con Hibernate.

3.9.3. Funciones de grupo en las consultas

Los resultados devueltos por una consulta HQL o SQL en la que se ha utilizado una función de grupo como por ejemplo *avg()*, *sum()*, *count()*, etc. se pueden recoger como un único valor utilizando el método ***uniqueResult()***. El siguiente ejemplo muestra el salario medio de los empleados:

```
// MOSTRAR SALARIO MEDIO DE LOS EMPLEADOS
String hql = "select avg(em.salario) from Empleados as em";
Query cons = session.createQuery(hql);
Double suma = (Double) cons.uniqueResult();
System.out.printf("Salario medio: %.2f%n", suma);
```

Si en la consulta intervienen varias funciones de grupo y además devuelve varias filas, podemos utilizar objetos devueltos por las consultas. Por ejemplo, a continuación se muestra el salario medio y el número de empleados por cada departamento:

```
//SALARIO MEDIO Y EL NÚMERO DE EMPLEADOS POR DEPARTAMENTO
String hql = "select e.departamentos.deptNo, avg(salario), count(empNo)
              from Empleados e group by e.departamentos.deptNo ";
```

```

Query cons = session.createQuery(hql);
Iterator iter = cons.iterate();

while (iter.hasNext()) {
    Object[] par = (Object[]) iter.next();
    Byte depar = (Byte) par[0];
    Double media = (Double) par[1];
    Long cuenta = (Long) par[2];
    System.out.printf("Dep: %d, Media: %.2f, Nº emp: %d %n",
                      depar, media, cuenta);
}

```

3.9.4. Objetos devueltos por las consultas

Anteriormente vimos cómo se pueden tratar los resultados obtenidos por una SELECT que no está asociada a ninguna entidad. Supongamos que a partir de las tablas *empleados* y *departamentos* quiero obtener una consulta en la que aparezcan el nombre del departamento, su número, el número de empleados y el salario medio. Como los datos de esta consulta no están asociados a ninguna clase, puedo crear una y utilizarla sin necesidad de mapearla. Cada fila devolverá un objeto de esa clase. Por ejemplo, creo la clase *Totales* en el paquete *primero* con 4 atributos: *numero*, *cuenta*, *media* y *nombre* (para guardar los datos de el número de departamento, número de empleados, la media de salario y el nombre del departamento) y los constructores, *getter* y *setter* asociados. La clase *Totales.java* es la siguiente:

```

package primero;
public class Totales {
    private Long cuenta; //número empleados
    private Byte numero; //número departamento
    private Double media; //media salario
    private String nombre; //nombre departamento

    public Totales( Byte numero, Long cuenta,
                    Double media, String nombre) {
        this.cuenta = cuenta;
        this.media = media;
        this.nombre = nombre;
        this.numero = numero;
    }

    public Totales() {}

    public Long getCuenta() {return this.cuenta;}
    public void setCuenta( Long cuenta) {this.cuenta = cuenta; }

    public Byte getNumero() {return this.numero;}
    public void setNumero( Byte numero) {this.numero = numero; }

    public Double getMedia() {return this.media;}
    public void setMedia(final Double media) {this.media = media; }

    public String getNombre() {return this.nombre;}
    public void setNombre(final String nombre) {this.nombre = nombre;}
}

```

Para hacer uso de la clase anterior construimos la consulta HQL de la siguiente manera:

```
String hql= "select new primero.Totales(" +
    " d.deptNo, count(e.empNo), coalesce(avg(e.salario),0) , "+
    " d.dnombre) "+
    " from Empleados as e right join e.departamentos as d "+
    " group by d.deptNo, d.dnombre ";

Query cons = session.createQuery(hql);
Iterator q = cons.iterate();
while (q.hasNext()) {
    Totales tot =(Totales) q.next();
    System.out.printf(
        "Numero Dep: %d, Nombre: %s, Salario medio: %.2f, N° emple: %d%n",
        tot.getNumero(), tot.getNombre(),
        tot.getMedia(), tot.getCuenta());
}
```

También podemos recuperar los valores de una consulta que no está asociada a ninguna clase mediante un array de objetos, clase **Object** (un ejemplo similar se vio anteriormente). Los resultados se reciben en un array de objetos, donde el primer elemento del array se corresponde con la primera fila, el siguiente con la siguiente fila. Dentro de cada fila será necesario acceder a los atributos o columnas mediante otro array de objetos:

```
String hql = "select d.deptNo, count(e.empNo), "+
    " coalesce(avg(e.salario),0), d.dnombre "+
    " from Empleados as e right join e.departamentos as d "+
    " group by d.deptNo, d.dnombre ";

Query cons = session.createQuery(hql);

List <Object[]> filas = cons.list(); // Todas las filas

for (int i = 0; i < filas.size(); i++) {
    Object[] filaActual = filas.get(i); // Acceso a una fila
    System.out.printf(
        "Numero Dep: %d, Nombre: %s, Salario medio: %.2f, N° emple: %d%n",
        filaActual[0], filaActual[3], filaActual[2], filaActual[1]);
}
```

3.10. INSERT, UPDATE y DELETE

Con el lenguaje HQL también podremos realizar operaciones INSERT, UPDATE y DELETE. La sintaxis para las operaciones UPDATE y DELETE es la siguiente:

(**UPDATE** | **DELETE**) [FROM] NombreEntidad [WHERE condición]

Donde:

- La palabra clave FROM es opcional.
- La cláusula WHERE también es opcional.

- Solo puede haber una entidad mencionada en la cláusula FROM y puede tener un alias, en ese caso cualquier referencia a la propiedad tiene que ser calificada utilizando ese alias. Si el nombre de la entidad no tiene un alias, entonces, es ilegal calificar cualquier referencia de la propiedad.

No se puede especificar ninguna asociación en una consulta masiva de HQL. Se pueden utilizar subconsultas en la cláusula WHERE (las subconsultas pueden contener asociaciones).

Para ejecutar un UPDATE o DELETE en HQL, utilizaremos el método `executeUpdate()` que devuelve el número de entidades afectadas por la operación; no hemos de olvidar realizar el commit para validar la transacción. Veamos algunos ejemplos, dentro de la misma transacción modificamos un empleado y eliminamos los empleados del departamento 20:

```
Transaction tx = session.beginTransaction();
//Modificamos el salario de GIL
String hqlModif = "update Empleados set salario = :nuevoSal
                  where apellido = :ape";
Query q1 = session.createQuery(hqlModif);
q1.setParameter("nuevoSal", (float) 2500.34);
q1.setString("ape", "GIL");

int filasModif = q1.executeUpdate();
System.out.printf("FILAS MODIFICADAS: %d%n", filasModif);

//Eliminamos los empleados del departamento 20
String hqlDel = "delete Empleados e
                  where e.departamentos.deptNo = :dep";
Query q = session.createQuery(hqlDel);
q.setInteger("dep", 20);

int filasDel = q.executeUpdate();
System.out.printf("FILAS ELIMINADAS: %d%n", filasDel);

tx.commit(); // valida la transacción
```

Con `tx.rollback()` se deshace la transacción.

La sintaxis para la operación INSERT es la siguiente:

```
INSERT INTO NombreEntidad (lista de propiedades) sentencia_select
```

Donde:

- Solo se soporta la forma INSERT INTO ... SELECT ..., no la forma INSERT INTO ... VALUES ... Es decir, solo podemos insertar datos procedentes de otra tabla.
- La *lista de propiedades* es análoga a la lista de columnas en la declaración INSERT de SQL.
- La *sentencia_select* puede ser cualquier consulta SELECT de HQL válida, hay que tener en cuenta que los tipos devueltos por la consulta coincidan con los esperados por el INSERT.

- Para el caso de la propiedad **id** hay dos opciones: se puede especificar en la lista de propiedades (en tal caso su valor se toma de la expresión de selección correspondiente) o se puede omitir de la lista de propiedades (en este caso se utiliza un valor generado). Esta última opción solamente está disponible cuando se utilizan generadores de id que operan en la base de datos (por ejemplo, cuando se usa AUTO_INCREMENT PRIMARY KEY en MySQL, la clave primaria se crea de forma automática sin necesidad de dar valor).

Ejemplo: desde SQL creo la siguiente tabla e inserto varias filas:

```
CREATE TABLE nuevos (
    dept_no TINYINT(2) NOT NULL PRIMARY KEY,
    dnombre VARCHAR(15),
    loc     VARCHAR(15)
) ENGINE=InnoDB;

INSERT INTO nuevos VALUES (51,'PERSONAL','MADRID');
INSERT INTO nuevos VALUES (52,'NÓMINAS','TOLEDO');
INSERT INTO nuevos VALUES (53,'OCIO','BARCELONA');
```

A continuación añado la siguiente línea al fichero **hibernate.reveng.xml**:

```
<table-filter match-catalog="ejemplo" match-name="nuevos"/>
```

A continuación generamos la nueva clase desde **Hibernate Code Generation Configurations**. Se tiene que generar la clase y el fichero **nuevos.hbm.xml**. Por último, ejecuto el código Java para insertar los datos de esa tabla en *Departamentos*:

```
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Departamentos (deptNo, dnombre, loc)"
    + " select n.deptNo, n.dnombre, n.loc from Nuevos n";

Query cons = session.createQuery( hqlInsert );
int filascreadas = cons.executeUpdate();

tx.commit(); // valida la transacción

System.out.printf("FILAS INSERTADAS: %d%n",filascreadas);
```

3.11. RESUMEN DEL LENGUAJE HQL

Las consultas en HQL no son sensibles a mayúsculas, a excepción de los nombres de las clases y propiedades Java. Podemos escribir FROM, from, SELECT, sElect, etc.

La cláusula más simple que existe en Hibernate es **from**, que obtiene todas las instancias de una clase, por ejemplo, **from Empleados** obtiene todas las instancias de la clase *Empleados* (en SQL, obtiene todas las filas de la tabla *empleados*). La cláusula **order by** ordena los resultados de la consulta.

La cláusula **where** permite refinar la lista de instancias retornadas y **order by** ordena la lista.

Ejemplos:

```
from Empleados where deptNo = 10 order by apellido
from Empleados as em where deptNo = 10 order by 1 desc
from Empleados as em where em.deptNo = 10
```

Podemos asignar alias a las clases usando la cláusula **as**: *from Empleados as em*, o sin usar dicha cláusula: *from Empleados em*.

Pueden aparecer múltiples clases a la derecha de FROM, lo que causa un producto cartesiano o una unión "cruzada" (cross join): *from Empleados as em, Departamentos as dep*.

Para obtener determinadas propiedades (columnas) en una consulta utilizamos la cláusula SELECT: *select apellido, salario from Empleados*, obtiene los atributos *apellido* y *salario* de la clase *Empleados*.

Las consultas pueden retornar múltiples objetos y/o propiedades como un array de tipo **Object[]**, una lista, una clase, etc. En apartados anteriores vimos algunos ejemplos.

Las funciones de grupo soportadas son las siguientes, la semántica es similar a SQL:

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

Se puede utilizar alias para nombrar los atributos y expresiones. Se pueden utilizar operadores aritméticos, de concatenación y funciones SQL reconocidas en la cláusula SELECT. Veamos algunos ejemplos:

```
select avg(salario) as med, count(empNo) as c from Empleados
select avg(salario), count(empNo) from Empleados
select avg(salario) + sum(salario), count(empNo) from Empleados
select apellido || '*' || oficio as campo from Empleados
select count(distinct deptNo) from Empleados
```

Las expresiones utilizadas en la cláusula **where** incluyen lo siguiente¹:

- Operadores matemáticos: +, -, *, /.
- Operadores de comparación binarios: =, >=, <=, <>, !=, like.
- Operadores lógicos and, or, not.
- Paréntesis () que indican agrupación.
- *in, not in, between, is null, is not null, is empty, is not empty, member of y not member of ..*
- Caso "simple", *case ... when ... then ... else ... end*, y caso "buscado", *case when ... then ... else ... end*.

¹Fuente: <http://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/queryhql.html#queryhql-expressions>

- Concatenación de cadenas ...||... o *concat(...,...)*.
- *current_date()*, *current_time()* y *current_timestamp()*.
- *second(...)*, *minute(...)*, *hour(...)*, *day(...)*, *month(...)*, y *year(...)* .
- Cualquier función u operador definido por EJB-QL 3.0: *substring()*, *trim()*, *lower()*, *upper()*, *length()*, *locate()*, *abs()*, *sqrt()*, *bit_length()*, *mod()* .
- *coalesce()* y *nullif()*.
- *str()* para convertir valores numéricos o temporales a una cadena legible.
- *cast... as ...*, donde el segundo argumento es el nombre de un tipo de Hibernate, y *extract... from ...* si *cast()* y *extract()* es soportado por la base de datos subyacente.
- La función *index()* de HQL, que se aplica a alias de una colección indexada unida.
- Las funciones de HQL que tomen expresiones de ruta valuadas en colecciones: *size()*, *minelement()*, *maxelement()*, *minindex()*, *maxindex()*, junto con las funciones especiales *elements()* e índices, las cuales se pueden cuantificar utilizando *some*, *all*, *exists*, *any*, *in*.
- Cualquier función escalar SQL soportada por la base de datos como *sign()*, *trunc()*, *rtrim()* y *sin()*.
- Parámetros posicionales JDBC ?.
- Parámetros con nombre *:name*, *:start_date* y *:x1*
- Literales SQL '*foo*', 69, *6.66E+2*, '1970-01-01 10:00:01.0'.
- Constantes Java.

Ejemplos:

```
from Empleados where deptNo in (10,20)
from Empleados where deptNo not in (10,20)
from Empleados where salario between 2000 and 3000
from Empleados where salario not between 2000 and 3000
from Empleados where comision is null
from Empleados where comision is not null
select lower(apellido), coalesce(comision, 0) from Empleados
select apellido from Empleados where apellido like 'A%'
```

Se pueden agrupar consultas usando **group by** y **having**. Las funciones SQL y las funciones de agregación SQL están permitidas en las cláusulas **having** y **order by**, si están soportadas por la base de datos subyacente. Ni la cláusula **group by** ni la cláusula **order by** pueden contener expresiones aritméticas. Ejemplos:

```
select de.dnombre, avg(em.salario)
from Empleados em, Departamentos de
where em.deptNo = de.deptNo
group by de.dnombre
having avg(em.salario) > 2000
```

Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas dentro de consultas. Una subconsulta se debe encerrar entre paréntesis. Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias en la consulta exterior). Ejemplos:

```
from Empleados as em where em.salario >
    (select avg(em2.salario) from Empleados em2
     where em2.deptNo=em.deptNo)

from Empleados as em where em.salario >
    (select avg(salario) from Empleados)
```

3.11.1. Asociaciones y uniones (joins)

En los mapeos realizados sobre las tablas las asociaciones de claves ajena se generan de forma automática. A continuación vamos a ver cómo se realizan asociaciones de forma manual sobre tablas que no tienen clave ajena definida. Por ejemplo, la tabla *empleados* tiene la columna *dir* que representa el director del empleado y es un número de empleado. Entonces podemos decir que un empleado que es director puede tener a cargo otros empleados, tenemos pues, una relación de uno a muchos (**one-to-many**) entre dos clases persistentes *Empleados*.

Para mapear esta relación añadimos las siguientes líneas al fichero XML **Empleados.hbm.xml** antes de la finalización de la definición de la clase (etiqueta `</class>`):

```
<set name="empleacargo" table="empleados" >
    <key column="dir" />
    <one-to-many class="primero.Empleados" />
</set>
```

En donde la colección se indica mediante la etiqueta `<set> </set>`, que se le da un nombre (*empleacargo*), se indica el nombre de la tabla de donde se tomará esa colección de objetos (*empleados*), la columna de la tabla por la que se relacionan (*dir*), el tipo de relación (**one-to-many**) y la clase con la que se establece la relación (*primero.Empleados*). También es necesario añadir a la clase *Empleados.java* la colección (*empleacargo*) con los métodos *set* y los *get*:

```
private Set<Empleados> empleacargo = new HashSet<Empleados>(0);

public Set<Empleados> getEmpleacargo() {
    return empleacargo;
}

public void setEmpleacargo(Set<Empleados> empleacargo) {
    this.empleacargo = empleacargo;
}
```

Una vez realizados los cambios se pueden realizar consultas con joins. Los tipos de joins soportados son *inner join*, *left outer join (left join)*, *right outer join (right join)*. Aunque la forma de utilizarlos es diferente a la usada en SQL. En estos joins no es necesario especificar en la cláusula *from* las instancias (tablas) que se combinan, solo hay que hacer el join con el atributo donde se define la asociación. El siguiente ejemplo:

```
from Empleados as emp join emp.empleacargo
```

Devuelve tantas instancias de dos objetos *Empleados* como resulte de combinar la tabla *empleados* consigo misma mediante las columnas *dir* y *emp_no*. El primer objeto resultante representa el director del empleado y el segundo el empleado. La orden anterior se corresponde con la siguiente orden en SQL:

```
SELECT dire.emp_no, dire.apellido, em.emp_no, em.apellido
FROM empleados dire, empleados em
WHERE dire.emp_no = em.dir
```

En la consulta anterior faltan los empleados que no tienen director, para que se muestren ejecutamos la consulta con *right join*, además la salida se puede ordenar:

```
from Empleados as emp right join emp.empleacargo order by emp.empNo
```

En SQL quedaría así:

```
SELECT dire.emp_no, dire.apellido, em.emp_no, em.apellido
FROM empleados dire
RIGHT JOIN empleados em ON dire.emp_no = em.dir
```

El siguiente ejemplo muestra los datos de los empleados (número de empleado y apellido) y los de su director, la salida se ordena por director:

```
String hql = "from Empleados as emp right join emp.empleacargo
            order by emp.empNo";
Query cons = session.createQuery(hql);
Iterator q = cons.iterate();

while (q.hasNext()) {
    Object[] par = (Object[]) q.next();
    Empleados dir = (Empleados) par[0];//director
    Empleados em = (Empleados) par[1]; //empleado

    if(dir!=null)
        System.out.printf("Empleado: %d, %s, DIRECTOR: %d, %s %n",
                           em.getEmpNo(), em.getApellido(),
                           dir.getEmpNo(), dir.getApellido());
    else
        System.out.printf("Empleado %d, %s, SIN DIRECTOR.%n",
                           em.getEmpNo(), em.getApellido());
}
```

El siguiente ejemplo muestra los empleados, si el empleado es director muestra los que tiene a su cargo:

```
String hql = "from Empleados ";
Query cons = session.createQuery(hql);

List<Empleados> lis = cons.list();
Iterator<Empleados> ite = lis.iterator();
System.out.println("=====");

while (ite.hasNext()) {
```

```

Empleados emple = (Empleados) ite.next();
if (emple != null) {
    Set acargo = emple.getEmpleacargo(); //empleados a cargo
    if (acargo.size() == 0) { // no son directores
        System.out.printf("EMPLEADO: %d, %s %n",
                           emple.getEmpNo(), emple.getApellido());
        System.out.println("=====");
    } else {
        System.out.printf("DIRECTOR: %d, %s %n",
                           emple.getEmpNo(), emple.getApellido());

        System.out.println("A cargo: " + acargo.size());

        Iterator it = acargo.iterator();
        while (it.hasNext()) { // recorrer los empleados a cargo
            Empleados em = (Empleados) it.next();
            System.out.printf("\t %d, %s %n",
                           em.getEmpNo(), em.getApellido());
        }
        System.out.println("=====");
    }
}
}
//
```

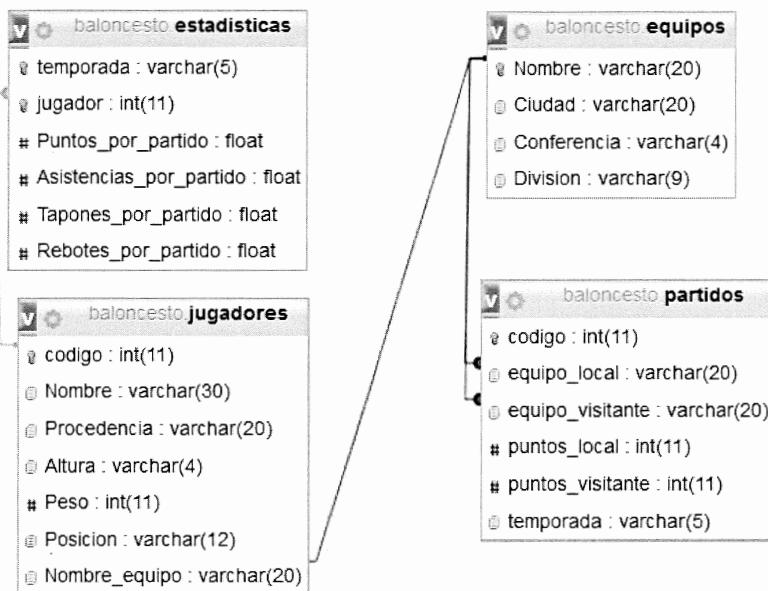
En Hibernate (y en general en las bases de datos) existen 4 tipos de relaciones: *uno-a-uno*, *uno-a-muchos*, *muchos-a- uno* y *muchos-a-muchos*; en los ejemplos solo se ha usado **one-to-many** que en el modelo relacional es una asociación uno a muchos. A la hora de definir la asociación se puede especificar más atributos, en este ejemplo solo se han indicado los necesarios.

El lenguaje HQL es mucho más extenso, aquí se han expuestos las nociones mínimas para empezar a trabajar.

Más información: <http://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/queryhql.html>

COMPRUEBA TU APRENDIZAJE

1. ¿Qué ventajas aportan las herramientas de mapeo objeto-relacional?
2. Cita alguno de los inconvenientes del uso de herramientas de mapeo objeto-relacional.
3. Busca en Internet más herramientas ORM, haz una lista e indica algunas características de ellas, plataformas sobre las que se utilizan, lenguajes, si son de software libre o propietario, etc.
4. Disponemos de la siguiente base de datos de nombre **baloncesto** y usuario y clave con el mismo nombre. Las tablas se muestran en la Figura 3.33. Son las siguientes:

Figura 3.33. Base de datos *baloncesto*.

EQUIPOS - Contiene información de los equipos que participan en la liga de baloncesto.

```

CREATE TABLE equipos (
    Nombre      varchar(20) NOT NULL,
    Ciudad      varchar(20) DEFAULT NULL,
    Conferencia varchar(4)  DEFAULT NULL,
    Division    varchar(9)  DEFAULT NULL,
    PRIMARY KEY (Nombre) )engine=innodb;
  
```

JUGADORES - Contiene información de los datos de los jugadores de los equipos. La altura viene en pies y el peso en libras. Al lado se indica la conversión a metros y gramos respectivamente.

```

CREATE TABLE jugadores (
    codigo int NOT NULL,
    Nombre           varchar(30) DEFAULT NULL,
    Procedencia     varchar(20) DEFAULT NULL,
    Altura          varchar(4)  DEFAULT NULL, -- en pies, 1 pie 0,3048 metros
    Peso            int DEFAULT NULL, -- en libras 1 libra 453.59 gramos
    Posicion        varchar(12) DEFAULT NULL,
    Nombre_equipo   varchar(20) DEFAULT NULL,
    PRIMARY KEY (codigo),
    FOREIGN KEY (Nombre_equipo) References equipos(Nombre)
)engine=innodb;
  
```

ESTADÍSTICAS – Esta tabla contiene la información de las estadísticas de los jugadores. La media de puntos por partido, las asistencias realizadas, los tapones, los rebotes, etc.

```

CREATE TABLE estadisticas (
    temporada      varchar(5) NOT NULL ,
    jugador        int NOT NULL ,
    Puntos_por_partido number(5,2) DEFAULT NULL,
  
```

```
Asistencias_por_partido number(5,2) DEFAULT NULL,  
Tapones_por_partido      number(5,2) DEFAULT NULL,  
Rebotes_por_partido      number(5,2) DEFAULT NULL,  
PRIMARY KEY (temporada,jugador),  
FOREIGN KEY (jugador) REFERENCES Jugadores(Codigo)  
)engine=innodb;
```

PARTIDOS – Esta tabla contiene la información de los partidos disputados, los equipos y los puntos. Las columnas son:

```
CREATE TABLE partidos (  
    codigo          int NOT NULL,  
    equipo_local    varchar(20) DEFAULT NULL,  
    equipo_visitante varchar(20) DEFAULT NULL,  
    puntos_local    number(5) DEFAULT NULL,  
    puntos_visitante number(5) DEFAULT NULL,  
    temporada       varchar(5) DEFAULT NULL,  
    PRIMARY KEY (codigo),  
    FOREIGN KEY (equipo_local) REFERENCES equipos(nombre),  
    FOREIGN KEY (equipo_visitante) REFERENCES equipos(nombre)  
)engine=innodb;
```

Mapea las tablas de la base de datos y estudia las clases generadas.

- Realiza un programa Java que admita un argumento desde *main()*, este argumento es el código de un jugador. El programa debe mostrar las estadísticas del jugador. Se deben controlar situaciones de error: si no se introduce ningún parámetro o el parámetro no es correcto (debe ser numérico) se debe mostrar un mensaje de error; si el jugador no existe muestra un mensaje indicándolo. Por ejemplo, si el código de jugador es 227, se debe mostrar la siguiente información:

```
DATOS DEL JUGADOR: 227  
Nombre : Kirk Hinrich  
Equipo : Bulls  
Temporada Ptos Asis Tap Reb  
=====  
06/07 16.6 6.3 0.3 3.4  
03/04 12.0 6.8 0.3 3.4  
05/06 15.9 6.3 0.3 3.6  
07/08 12.0 6.0 0.3 3.4  
04/05 15.7 6.4 0.3 3.9  
=====  
Num de registros: 5  
=====
```

- Realiza un programa Java que muestre por cada equipo la lista de sus jugadores con la media de los puntos por partido. El listado debe aparecer ordenado por equipo. Debe mostrar también el número de equipos que hay. Ejemplo de salida del programa:

```
Número de Equipos: 30  
=====  
Equipo: 76ers  
120, Louis Amundson: 1,45  
125, Willie Green: 9,04
```

```

=====
Equipo: Bobcats
181, Derek Anderson: 11,20
184, Jermareo Davidson: 3,20
191, Adam Morrison: 11,80
.
.
.
=====
Equipo: Bucks
204, Royal Ivey: 3,93
.
.
.
```

7. Realiza un programa Java que inserte estadísticas para el jugador 123. Los datos a insertar son los siguientes: temporada 05/06: puntos por partido 7, rebotes 5; temporada 06/07 puntos por partido 10, tapones 3. Los valores no indicados tendrán valor 0. Transforma después el programa para que todos los valores a insertar se introduzcan a partir de los argumentos de *main()*. Controlar posibles errores, número de argumentos correctos, que el jugador exista, que la estadística no exista, etc.
8. Realiza un programa Java para visualizar los datos de los empleados. La pantalla debe presentar 5 botones que nos permitirán iniciar la visualización de empleados y movernos entre los empleados. Los campos que se muestran en la pantalla no son editables. El botón *Ejecutar Consulta y Primer Reg* deben mostrar el primer empleado (el que tiene menor número de empleado). Si no hay empleados visualizar mensaje indicándolo. El botón *Último Reg* muestra el último empleado (el que tiene el mayor número de empleado). El botón *Siguiente* muestra el siguiente empleado al que se está visualizando en este momento. Si se pulsa y estamos ante el último empleado se debe visualizar un mensaje indicándolo (por ejemplo: "No hay más empleados"). El botón *Anterior* muestra el anterior empleado al que se está visualizando. Si se pulsa y estamos ante el primer empleado se debe visualizar un mensaje indicándolo (por ejemplo: "primer empleado"). Al hacer clic en el botón de cierre de la ventana finalizará el programa. La pantalla es la siguiente:

CONSULTA DE EMPLEADOS

CONSULTA DE EMPLEADOS

Nº de empleado:	<input type="text"/>	<input type="button" value="Ejecutar Consulta"/>
Apellido:	<input type="text"/>	
Oficio:	<input type="text"/>	
Departamento:	<input type="text"/>	
Salario:	<input type="text"/>	

<input type="button" value="Primer Reg"/>	<input type="button" value="Siguiente"/>	<input type="button" value="Anterior"/>	<input type="button" value="Último Reg"/>
---	--	---	---

Figura 3.34. Ejercicio 8.

ACTIVIDADES DE AMPLIACIÓN

1. En una compañía de metro se dispone de una base de datos que contiene las tablas con la información necesaria para su gestión. Las tablas son las siguientes:

- Tabla **T_Lineas**, contiene la información de las distintas líneas de metro que gestiona la compañía. La clave es el **cod_linea**.
- Tabla **T_Estaciones**, contiene la información de las distintas estaciones de metro que existen y son gestionadas por la compañía. La clave es **cod_estacion**.
- Tabla **T_Linea_estacion**, contiene la información de las estaciones que tiene cada línea y el orden que hace cada estación dentro de la línea, este número no se puede repetir dentro de la misma línea. La clave está formada por el **cod_linea** y el **cod_estacion**.
- Tabla **T_Accesos**, contiene la información sobre los distintos accesos de entrada por los que se puede llegar a cada estación. Una estación puede tener distintos puntos de acceso. La clave es **cod_acceso**.
- Tabla **T_Trenes**, contiene la información sobre los distintos trenes que circulan por cada línea. En una línea circulan muchos trenes. Y un tren pertenece a una línea. La clave es **cod_tren**. Los trenes se guardan en cocheras.
- Tabla **T_Cocheras**, contiene la información sobre las cocheras y depósitos donde aparcان los trenes al final de la jornada. La clave es **cod_cochera**. En una cochera se guardan muchos trenes, y un tren se guarda en una cochera.
- Tabla **T_Viajes**, contiene la información sobre distintos viajes que ofrece la compañía. Cada viaje tiene una estación de destino y una estación de origen. La clave es **cod_viaje**.

Las tablas y sus relaciones se muestran en la siguiente Figura 3.35:

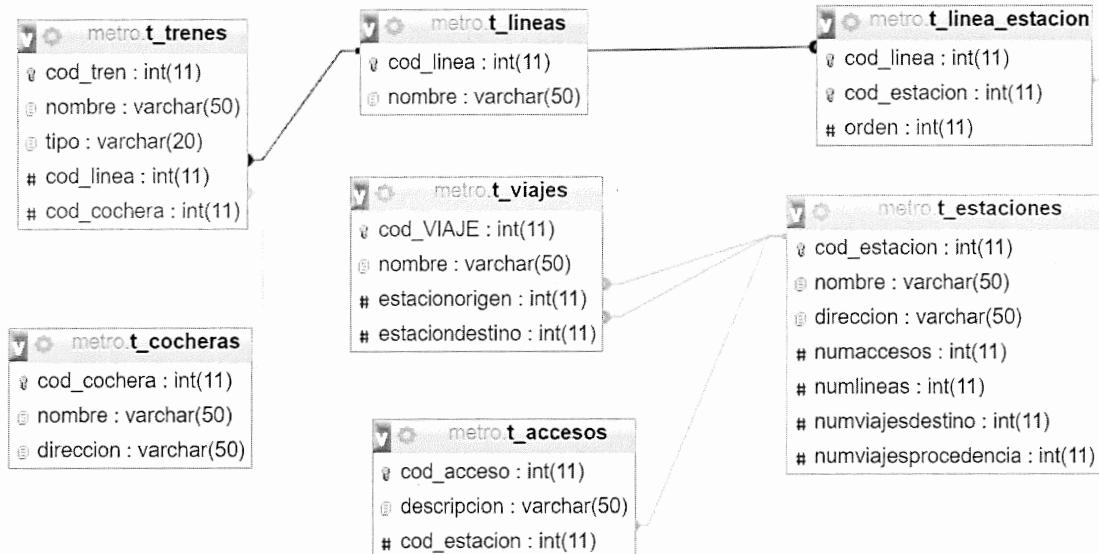


Figura 3.35. Modelo Actividad de ampliación.

Mapea las tablas utilizando Hibernate y realiza un proyecto Java con los siguientes métodos (si prefieres puedes hacer cada ejercicio en una clase):

2. Crea un método que reciba un número de línea, un número de estación, el orden y los inserte en la tabla T_LINEA_ESTACION. Antes de insertar hay que comprobar que la línea y la estación existan en las tablas correspondientes, que no exista el registro en la tabla T_LINEA_ESTACION, y que el orden sea correcto. Visualiza los mensajes de error que correspondan (línea inexistente, estación inexistente, registro ya existe...)
3. Crea un método para actualizar los campos *numaccesos*, *numlineas*, *numviajesdestino* y *numviajesprocedencia* de las estaciones de la tabla T_ESTACIONES. Estas columnas deben contener el número de accesos que tiene la estación (*numaccesos*), el número de líneas que pasan por la estación (*numlineas*), el número de viajes que la tienen como destino (*numviajesdestino*), y el número de viajes que la tienen como procedencia (*numviajesprocedencia*).
4. Crea un método que visualice por cada estación, el número de líneas que pasan por ella, el número de accesos que tiene, el número de viajes que tienen como destino la estación y los viajes con su nombre y su código. Y lo mismo con los viajes de procedencia. Obtén la siguiente salida por estación:

```
COD ESTACIÓN: xxxxxxx NOMBRE ESTACIÓN: xxxxxxxxxxxx
-----
Números de líneas que pasan: xxxxxx
Número de accesos que tiene: xxxxxx
NUM-Viajes-DESTINO: xxxxxxxx
COD-VIAJE      NOMBRE-VIAJE-DESTINO
-----
xxxxx     xxxxxxxxxxxxxxxxxxxxxx
xxxxx     xxxxxxxxxxxxxxxxxxxxxx

NUM-Viajes-PROCEDENCIA: xxxxxxxx
COD-VIAJE      NOMBRE-VIAJE-PROCEDENCIA
-----
xxxxx     xxxxxxxxxxxxxxxxxxxxxx
xxxxx     xxxxxxxxxxxxxxxxxxxxxx
```

5. Crea un método o una clase para crear el fichero *Viajes.xml* que debe contener la información de todos los viajes: el código, el nombre, y los nombres de la estación de destino y de procedencia. Para ello utiliza JAXB, y crea las clases correspondientes para crear el mapeo a XML. El fichero XML debe de tener la siguiente estructura:

```
<todoslosviajes>
  <viaje>
    <codigoviaje> </codigoviaje>
    <nombreviaje> </nombreviaje>
    <nombrestacionorigen> </nombrestacionorigen>
    <nombrestaciondestino> </nombrestaciondestino>
  </viaje>
</todoslosviajes>
```

