

UD4. MAPEO OBJETO-RELACIONAL (Herramientas)

1. Introducción
2. Herramientas ORM. Características
3. Arquitectura Hibernate
4. Instalación y configuración de Hibernate
5. Programación con Hibernate
6. Estructura de los ficheros de mapeo
7. Clases persistentes
8. Sesiones y objetos Hibernate
 - 8.1 Transacciones y sesiones
 - 8.2 Estados de un objeto Hibernate
 - 8.3 Carga de objetos persistentes
 - 8.4 Almacenamiento, modificación y borrado de objetos persistentes
9. Consultas
 - 9.1 Parámetros en las consultas
 - 9.2 Consultas sobre clases no asociadas
 - 9.3 Funciones de grupo en las consultas
 - 9.4 Objetos devueltos por las consultas
10. Insert, Update y Delete
11. Resumen de lenguaje HQL
 - 11.1 Asociaciones y uniones
12. Ejemplos resueltos
13. Bibliografía

1. Introducción.

A la hora de almacenar los datos de un programa orientado a objetos en una base de datos relacional, surge un inconveniente debido a incompatibilidad de sistemas de tipos de datos. En el software orientado a objetos, la información se representa como **clases y objetos**. En las bases de datos relacionales, como **tablas y sus restricciones**. Por tanto, para almacenar la información tratada en un programa orientado a objetos en una base de datos relacional es necesaria **una traducción entre ambas formas**.

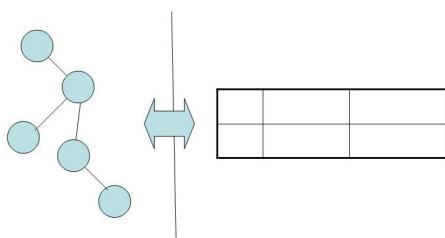
Las bases de datos del tipo relacional sólo permiten guardar tipos de datos primitivos (enteros, cadenas de texto, etc.) por lo que no se puede guardar de forma directa los objetos de la aplicación en las tablas. Por tanto, se debe convertir los valores de los objetos en valores simples que puedan ser almacenados en una base de datos (y poder recuperarlos más tarde).

El **mapeo objeto-relacional (ORM) - Object Relational Mapping-** soluciona este problema. *Es una técnica de programación que se utiliza con el propósito de convertir datos entre el sistema de tipos utilizado en el lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia.* En la práctica, esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional que posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo) en la base de datos relacional.

Por tanto, para acceder de forma efectiva a la base de datos relacional 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 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 (o registros) pasan a ser objetos que podemos manejar con facilidad.**

Existen diversos paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

Aplicación ---- Mapeo ---- Base de datos Relacional
(objetos) ----- (tablas)



2. 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. **Cada fila en la tabla se mapea a un objeto y cada columna a una propiedad:**

Clase – ----- Tabla

Objeto ----- Fila

Propiedad -- Columna

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. Sólo será necesario cambiar alguna línea en el fichero de configuración.

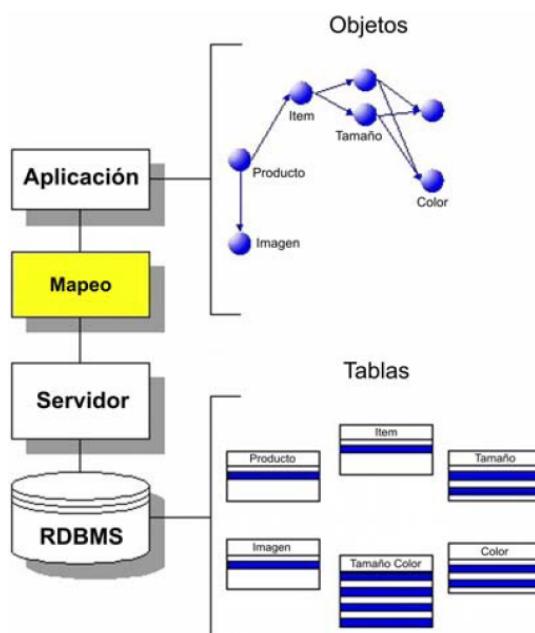
Algunas de **las ventajas que aportan estas herramientas** son:

- 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
- Lenguaje propio para realizar consultas
- Incentivan la portabilidad y escalabilidad de los programas de software.

Entre los **inconvenientes** se puede citar que:

- Tiempo utilizado en el aprendizaje. Este tipo de herramientas suelen ser complejas por lo que su correcta utilización lleva un tiempo que hay que emplear en ver el funcionamiento correcto y ver todo el partido que se le puede sacar.
- Menor rendimiento (aplicaciones algo más lentas). Esto es debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá de transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.
- Sistemas complejos. Normalmente la utilidad de ORM desciende con la mayor complejidad del sistema relacional.

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.



Nosotros veremos Hibernate.

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.

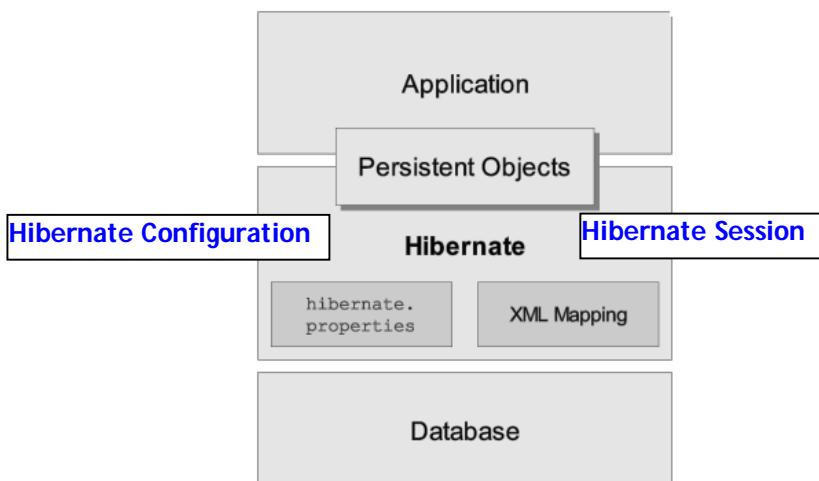
Se está convirtiendo en el estándar de facto para almacenamiento persistente cuando queremos independizar la capa de negocio del almacenamiento de la información. Esta capa de persistencia permite abstraer al programador Java de las particularidades de una determinada base de datos, proporcionando clases 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 la base de datos.

3. Arquitectura Hibernate.

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*.



- Hibernate parte de una filosofía de **mapear objetos Java normales** o más **conocidos como POJOs (Plain Old Java Objects)**. (Los POJOs son las clases java que se corresponden con tablas de la BD)
- 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 clase **Session (org.hibernate.Session)** ofrece métodos como: `save(Object objeto)`, `createQuery(String Consulta)`, `beginTransaction()`, `close()`, etc. para interactuar con la base de datos, tal y como se hace con una conexión JDBC, con la diferencia de 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 Hibernate son las siguientes:

- **SessionFactory** (*org.hibernate.SessionFactory*). Esta interface permite obtener instancias *Session*. 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.
- **Configuration** (*org.hibernate.cfg.Configuration*). Esta interface se utiliza para configurar Hibernate. La aplicación utiliza una instancia de **Configuration** para especificar la ubicación de los documentos que indican el mapeo de los objetos y propiedades específicas de Hibernate, y a continuación crea la **SessionFactory**.
- **Query** (*org.hibernate.Query*). Esta interface permite realizar consultas a la base de datos y controla cómo se ejecutan dichas consultas. Las consultas se escriben en HQL o SQL 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.
- **Transaction** (*org.hibernate.Transaction*). Esta interface nos permite asegurar que cualquier error entre el inicio y final de la transacción produzca el fallo del sistema.

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

Tutorial de Hibernate

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

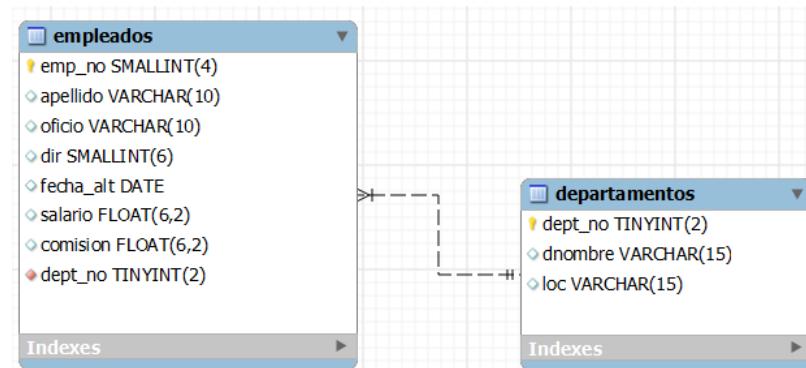
4. Instalación y configuración de Hibernate.

En este apartado vamos a instalar y configurar Hibernate en el entorno NetBeans y en Eclipse.

Instalaciones previas a Hibernate en nuestro equipo para los ejemplos:

- JDK 6 o 7 (Java Development Kit - Kit de Desarrollo para Java)
- IDE NetBeans 7.2 (Eclipse Juno SR1)
- Base de datos MySQL 5.5

Para los ejemplos, trabajaremos con la base de datos **empresaz**, cuyo esquema relacional es el siguiente:



```
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,
    FOREIGN KEY (dept_no) REFERENCES
    departamentos(dept_no)
);
```

```
CREATE TABLE departamentos (
    dept_no     TINYINT(2) NOT NULL PRIMARY KEY,
    dnombre     VARCHAR(15),
    loc         VARCHAR(15)
);
```

Las tareas de instalación y configuración, similares en ambos IDEs consistirán básicamente en los siguientes pasos:

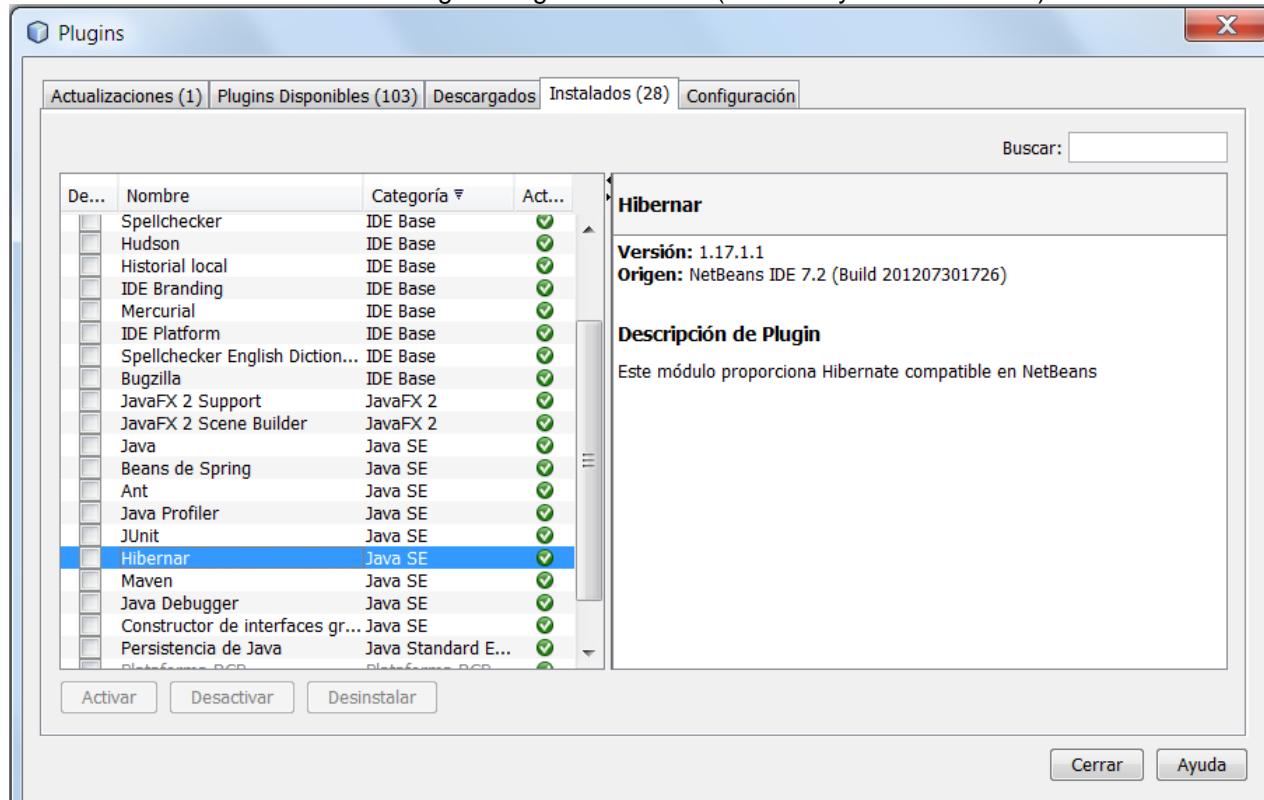
1. Instalación del plugin Hibernate en el IDE
2. Configuración del driver MySQL en el IDE (conexión a la BD MySQL)
3. Creación de un proyecto java
4. **Configuración de Hibernate**
 - a. Creación del archivo de configuración **hibernate.cfg.xml** (fichero XML con información sobre la conexión de base de datos, las asignaciones de recursos y otras propiedades de conexión).
 - b. Modificar el archivo de configuración *hibernate.cfg.xml* para habilitar la **depuración de consultas SQL** o bien crear el fichero XML **Hibernate Console Configuration**.
 - c. Crear los archivos de mapeo y clases java (POJOs). Para ello se crea el archivo de ingeniería inversa **hibernate.reveng.xml**
 - d. Crear la clase **HibernateUtil.javaHelperFile** para el manejo de sesiones.

4.1 Instalación y configuración de Hibernate en NetBeans

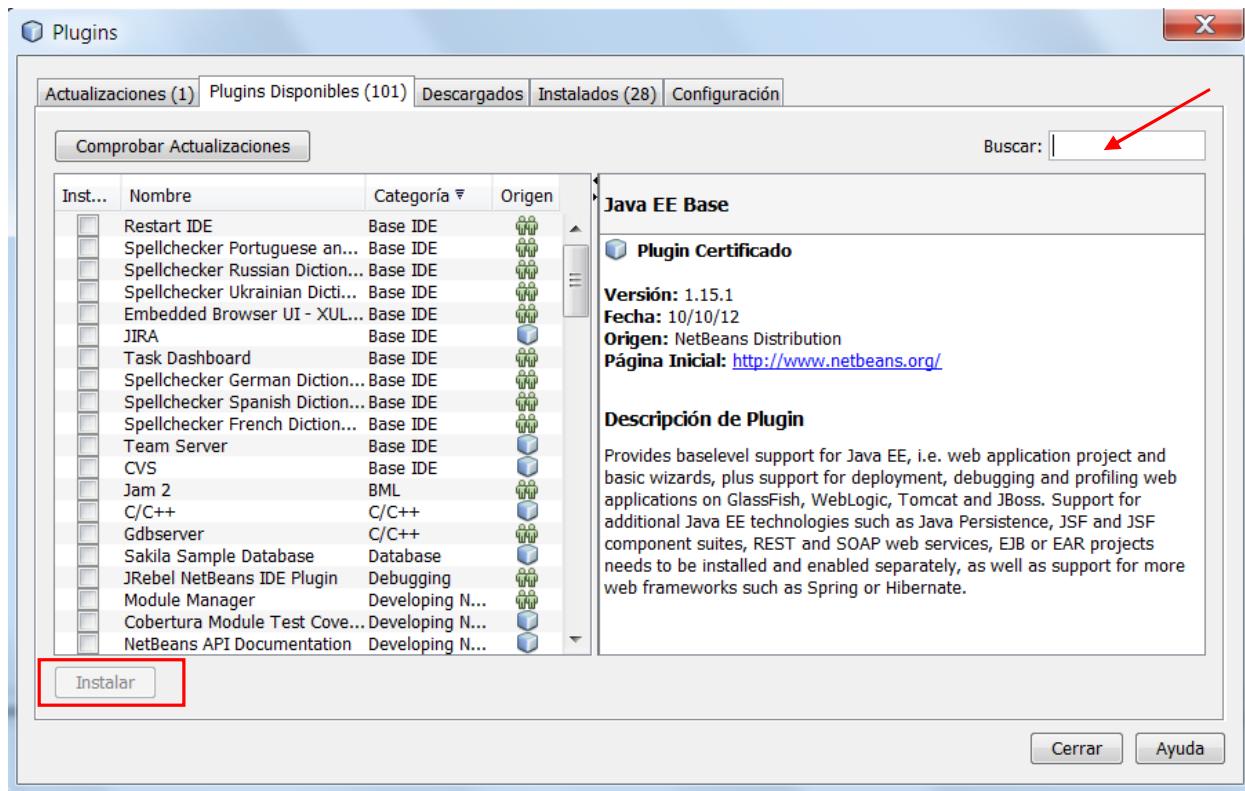
1. Instalación del Plugin Hibernate

Al instalar NetBeans, normalmente se instalan una serie de plugins, entre ellos Hibernate. Para comprobar si está instalado el plugin Hibernate en NetBeans:

- Desde herramientas/Plugins/Plugins instalados (vemos si ya está instalado)

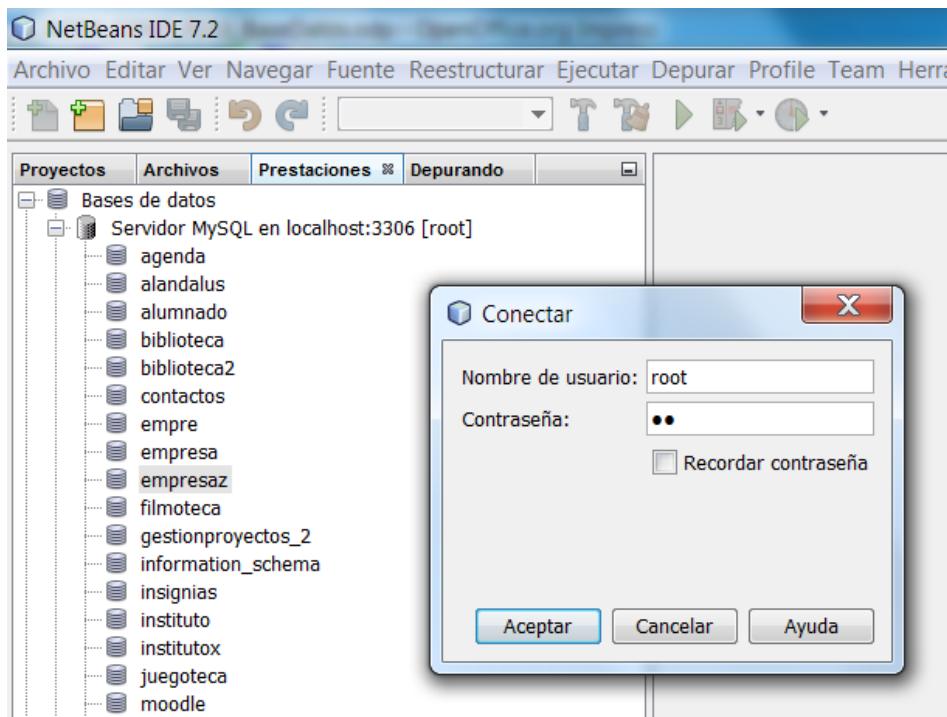


- Si no está instalado, desde Plugins Disponibles, lo buscamos y lo instalamos

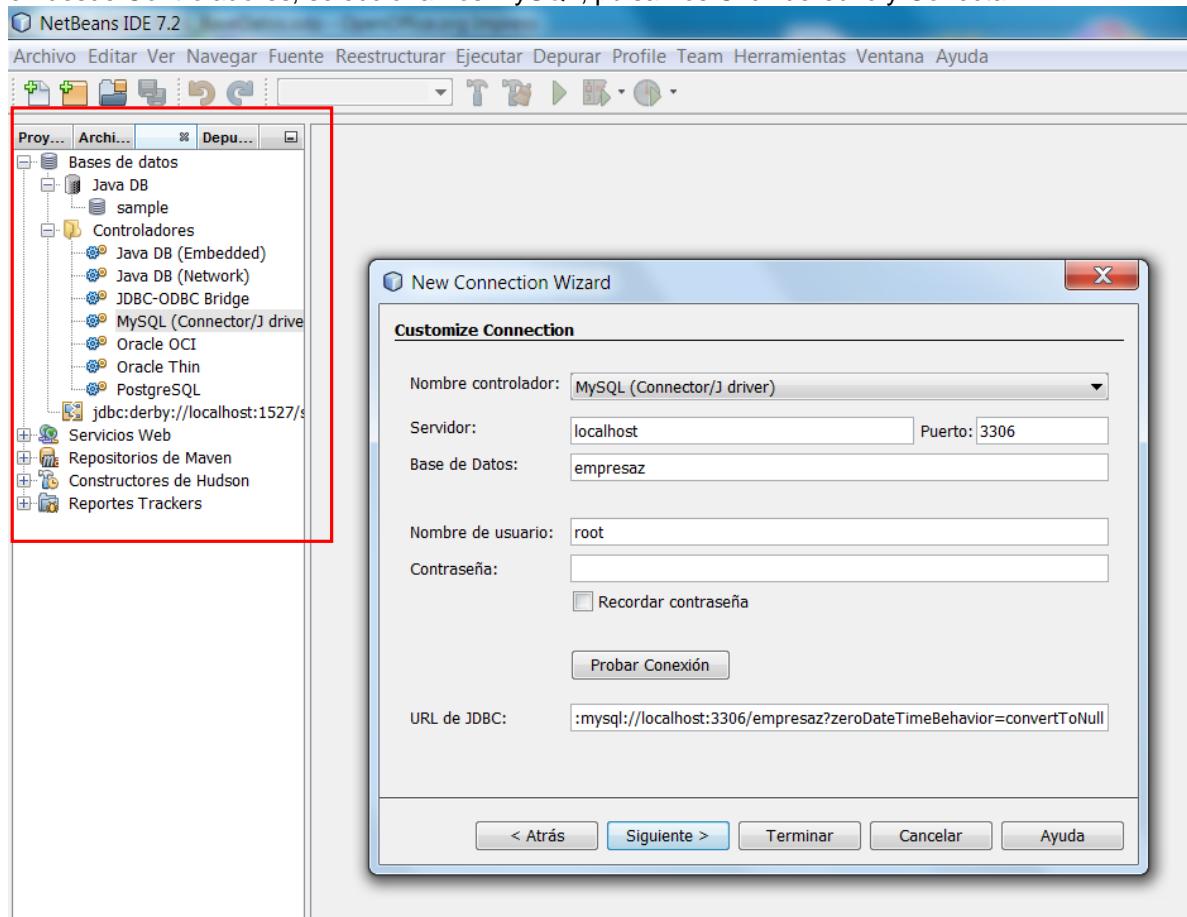


2. Conexión con la base de datos MySQL

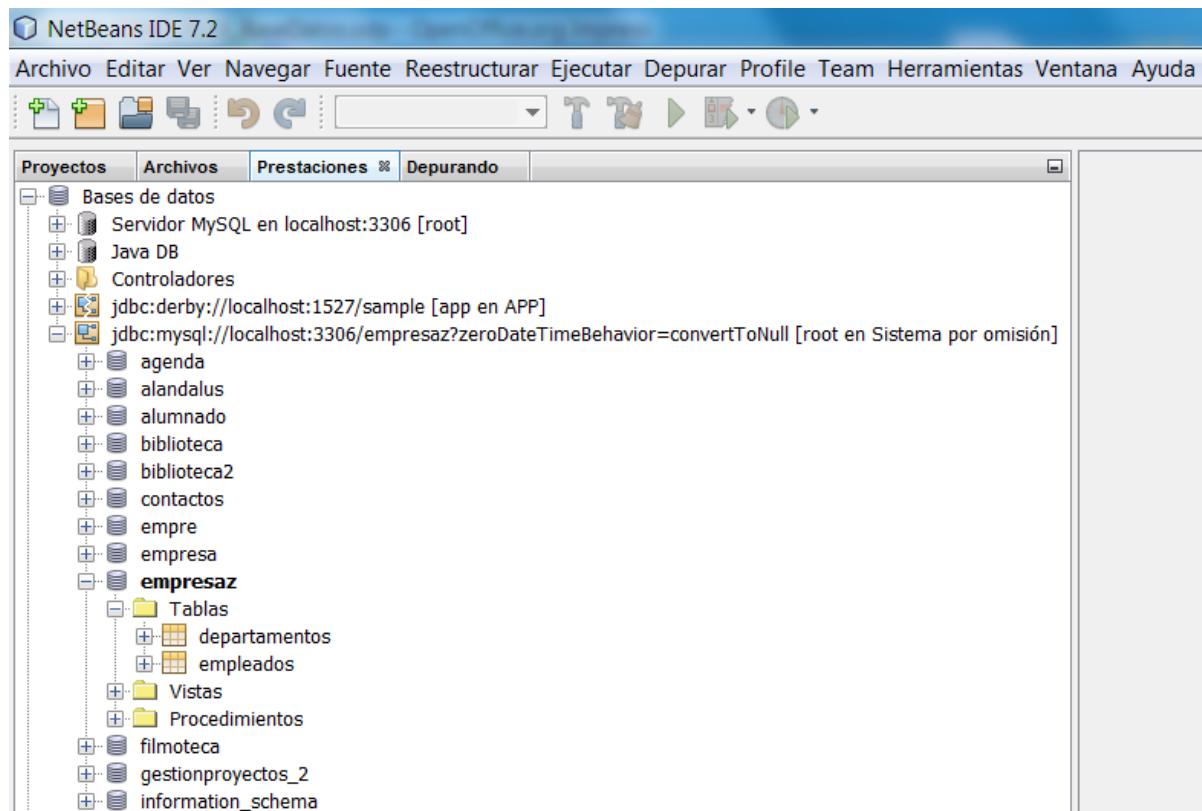
- Iniciamos el servidor de bases de datos MySQL y comprobamos que tenemos la BD empresaz.
- En la ventana Proyectos/Archivos/Prestaciones, seleccionamos la pestaña Prestaciones y expandimos el nodo de Bases de Datos.
- Seleccionamos la BD empresaz. Pulsamos Click derecho y conectar (conectamos como usuario root) y Aceptar.



O bien desde Controladores, seleccionamos MySQL, pulsamos Click derecho y Conectar:



Le damos a siguiente y terminar y ya tenemos la conexión a la BD.



3. Creación del proyecto

- Creamos un Proyecto o Aplicación Java de nombre **Ud4_Hibernate_01**
- Para poder utilizar Hibernate en un proyecto JSE, es necesario añadir las **librerías de Hibernate** a dicho proyecto.

Las librerías de Hibernate están incluidas en NetBeans y pueden añadirse a cualquier proyecto haciendo click derecho en el nodo “Bibliotecas”, de la ventana de proyectos, seleccionando añadir Biblioteca y seleccionando biblioteca Hibernate.

NetBeans incluye asistentes que ayudan a crear los ficheros de Hibernate que se necesitan en el proyecto. Los asistentes se pueden usar para crear el archivo de configuración de Hibernate.

4. Configuración de Hibernate

a) Creación del archivo de configuración de Hibernate (`hibernate.cfg.xml`)

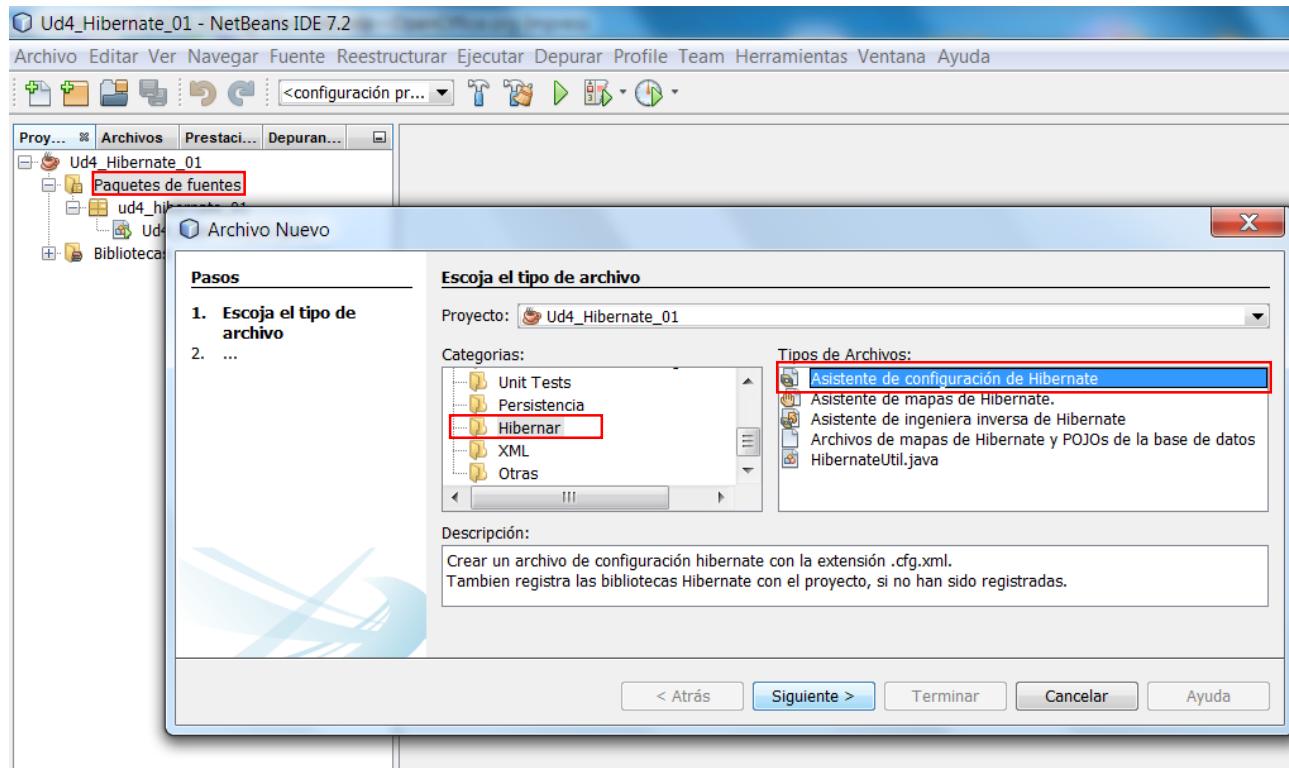
El archivo de configuración de Hibernate (`hibernate.cfg.xml`) contiene información sobre la conexión de la base de datos, los recursos de mapeo y otras propiedades de la conexión. Cuando se crea el archivo de configuración de Hibernate usando el asistente, podemos especificar la conexión a la base de datos, eligiendo de una lista de conexiones de bases de datos registradas en el IDE.

Cuando se genera el archivo de configuración, el IDE añade de forma automática detalles de la conexión e información basada en la conexión de la base de datos seleccionada. El IDE añade también las bibliotecas de Hibernate en el proyecto. Después de crear el fichero de configuración, éste puede ser editado usando el editor interactivo, o editar directamente el código XML.

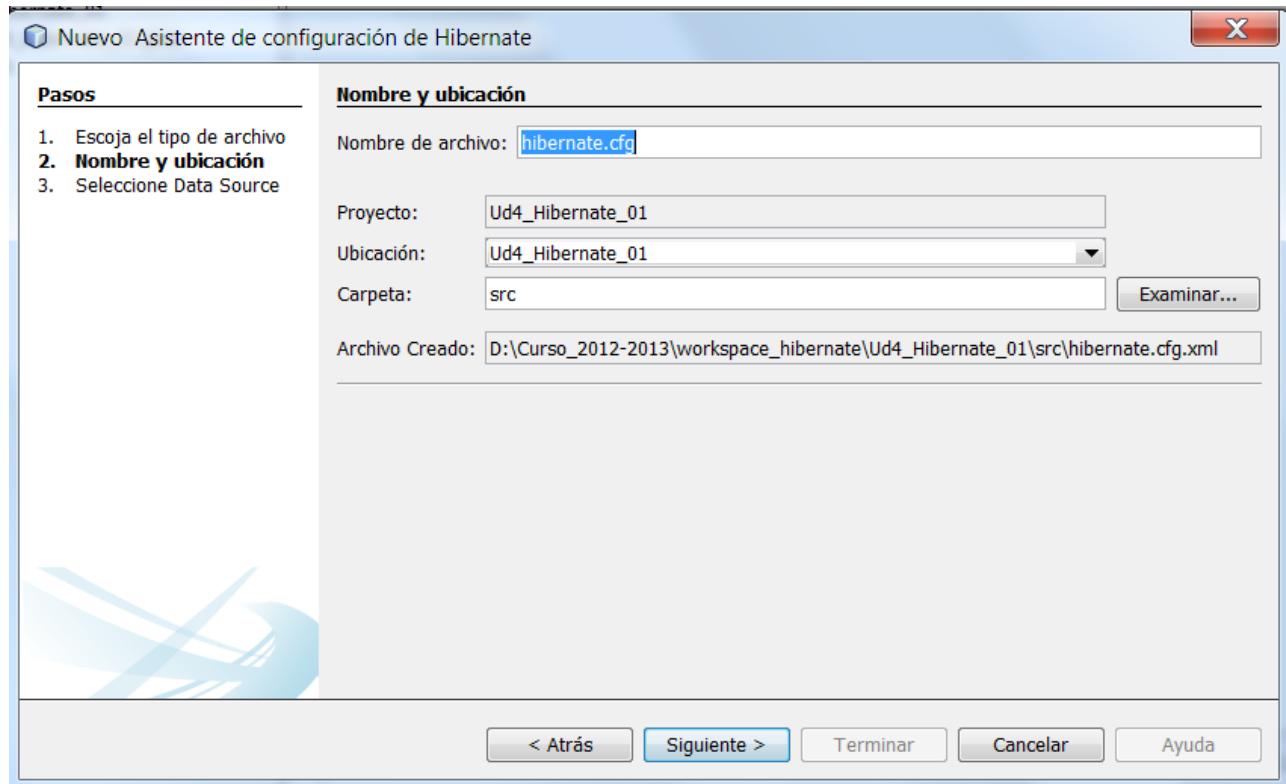
Los **pasos a seguir para crear el fichero de configuración de Hibernate** son los siguientes:

Dentro de la ventana de proyectos, hacemos click derecho sobre el nodo de “**Paquetes de fuentes**” y seleccionamos en el menú **Nuevo → Otro**, para abrir el asistente de Nuevo Fichero.

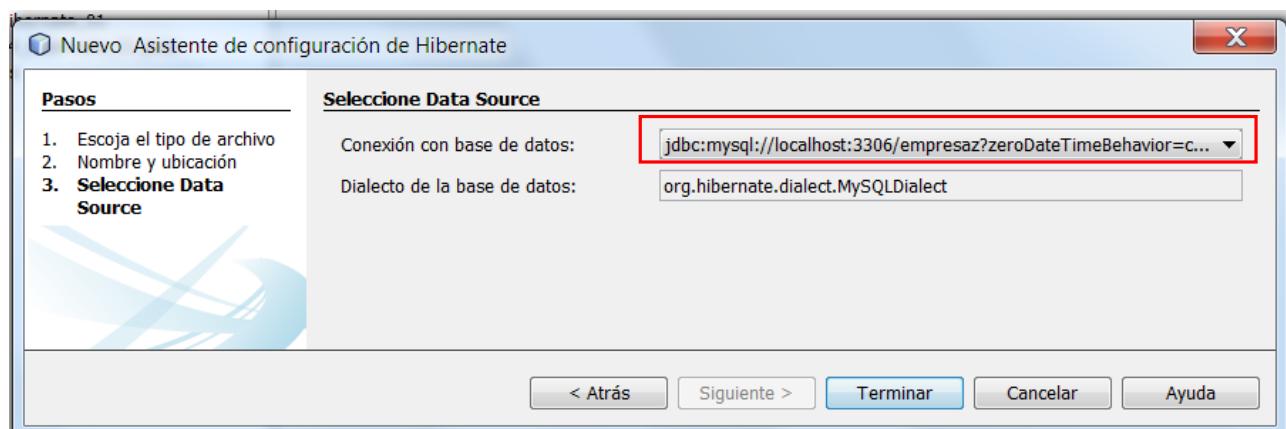
Seleccionamos **Hibernate → Asistente de configuración** y pulsamos **siguiente**.



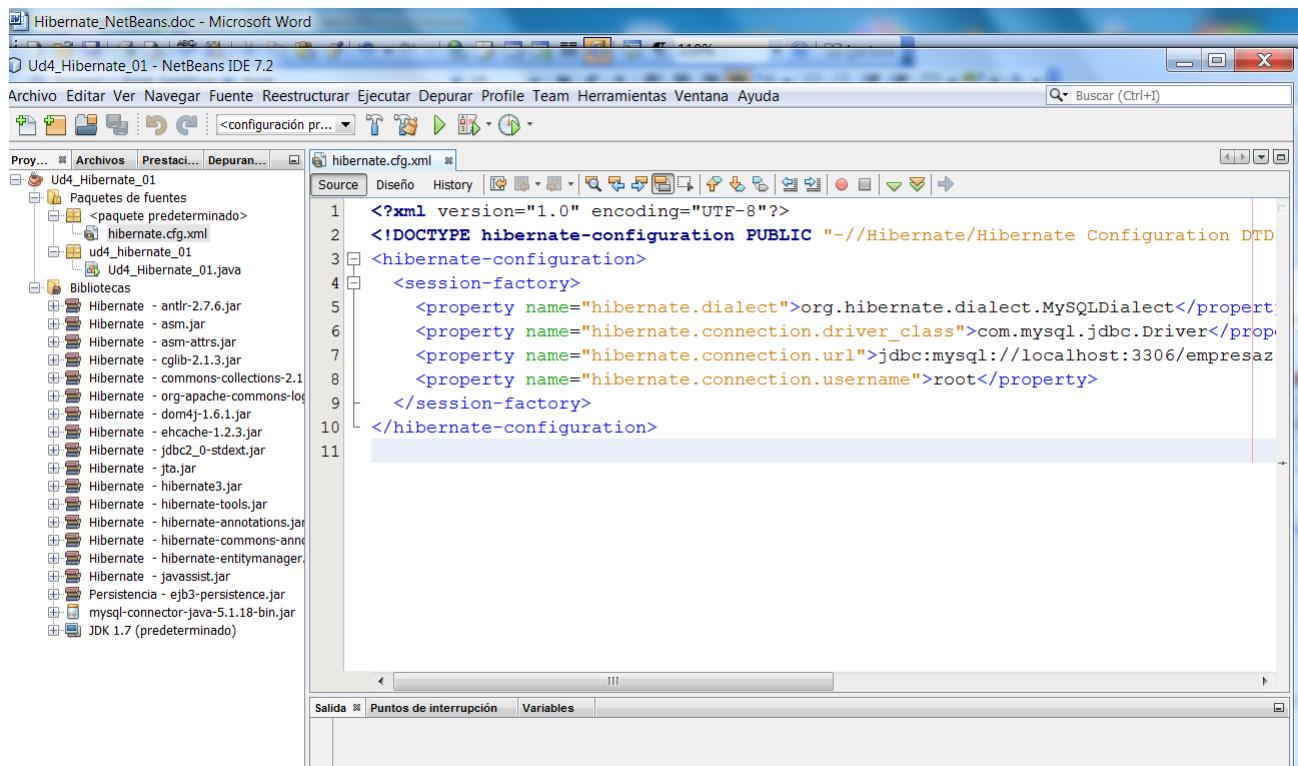
En la siguiente pantalla dejamos los valores por defecto y pulsamos **siguiente**



Seleccionamos la BD mysql **empresaz** y pulsamos **terminar**.



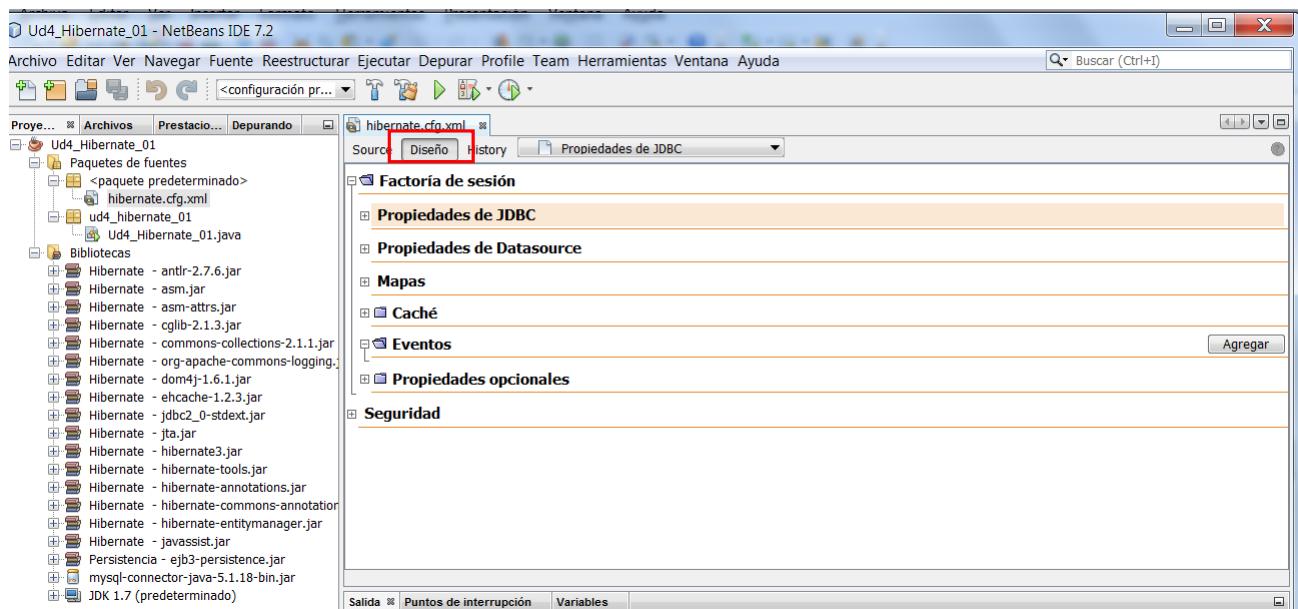
Se nos muestra el fichero de configuración de Hibernate y podemos comprobar desplegando el nodo Bibliotecas que se han añadido los ficheros JAR de Hibernate que son necesarios en la aplicación y el fichero JAR de conexión a MySQL:



b) modificar el fichero de configuración hibernate.cfg.xml para habilitar la depuración de consultas SQL

Para poder establecer las diferentes consultas que se desean realizar en la base de datos, es necesario especificar en el fichero hibernate.cfg.xml las propiedades necesarias para poder habilitar la depuración de consultas SQL.

Abrimos el fichero **hibernate.cfg.xml**, seleccionando la **pestaña de Diseño**. Para poder abrir el fichero, se puede expandir en el nodo de Archivos de Configuración, de la ventana de proyectos, y hacer doble click en hibernate.cfg.xml.



- Habilitar el procedimiento de depuración para consultas SQL

Expandimos **Propiedades de configuración** en el nodo **Propiedades Opcionales**

Hacemos click en **Agregar** y agregamos Propiedad Hibernate. En el cuadro de diálogo seleccionamos hibernate.show.sql y ponemos el valor de la propiedad a true. Con esta propiedad habilitamos el seguimiento de la depuración en las consultas SQL

The screenshot shows the 'Propiedades Opcionales' (Optional Properties) configuration interface. The 'Propiedades de configuración' (Configuration Properties) section is highlighted with a red box. It contains two properties:

Nombre	Valor
hibernate.dialect	org.hibernate.dialect.MySQLDialect
hibernate.show_sql	true

Below this section are other collapsed sections: 'Propiedades de conexión y JDBC', 'Propiedades de la caché', and 'Propiedades de transacción'.

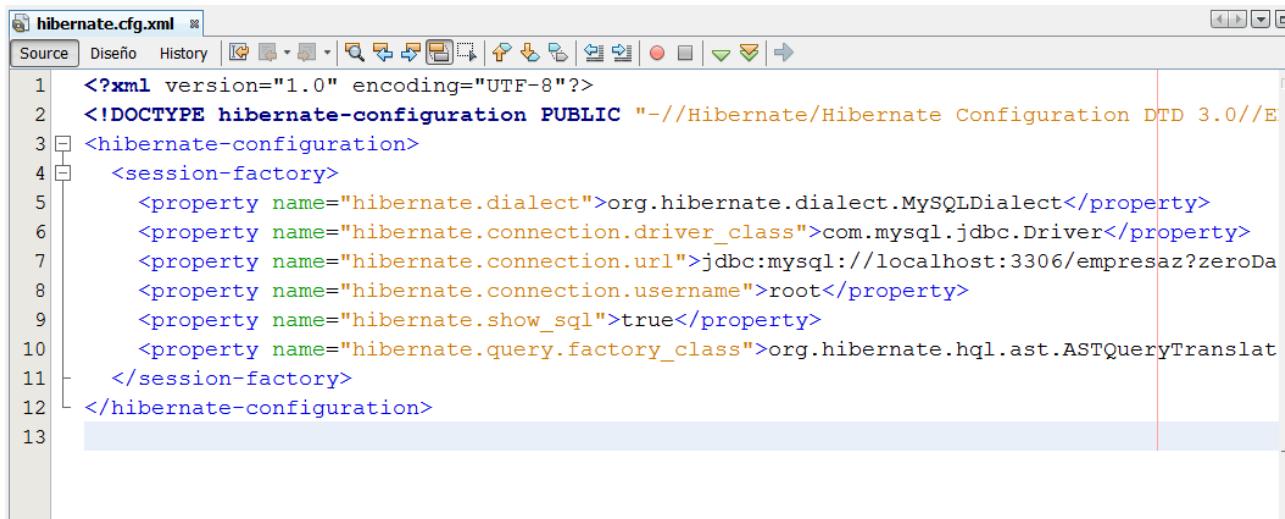
A continuación seleccionamos el nodo de “**Propiedades varias**” y agregamos hibernate.query.factory_class en la lista desplegable ‘Nombre de la Propiedad’.

The screenshot shows the 'Propiedades Opcionales' configuration interface. The 'Propiedades varias' (Various Properties) section is highlighted with a red box. It contains one property:

Nombre	Valor
hibernate.query.factory_class	org.hibernate.hql.ast.ASTQueryTranslatorFactory

Below this section are other collapsed sections: 'Propiedades de configuración', 'Propiedades de conexión y JDBC', 'Propiedades de la caché', and 'Propiedades de transacción'.

Si hacemos click en la pestaña **Source** del fichero de configuración, podremos ver el editor XML. La apariencia del fichero es la que se muestra a continuación. **Guardamos todos los cambios** realizado en el fichero.



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3  <hibernate-configuration>
4    <session-factory>
5      <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
6      <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
7      <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/empresaz?zeroDateTimeBehavior=convertToNull</property>
8      <property name="hibernate.connection.username">root</property>
9      <property name="hibernate.show_sql">true</property>
10     <property name="hibernate.query.factory_class">org.hibernate.hql.ast.ASTQueryTranslatorFactory</property>
11   </session-factory>
12 </hibernate-configuration>
13

```

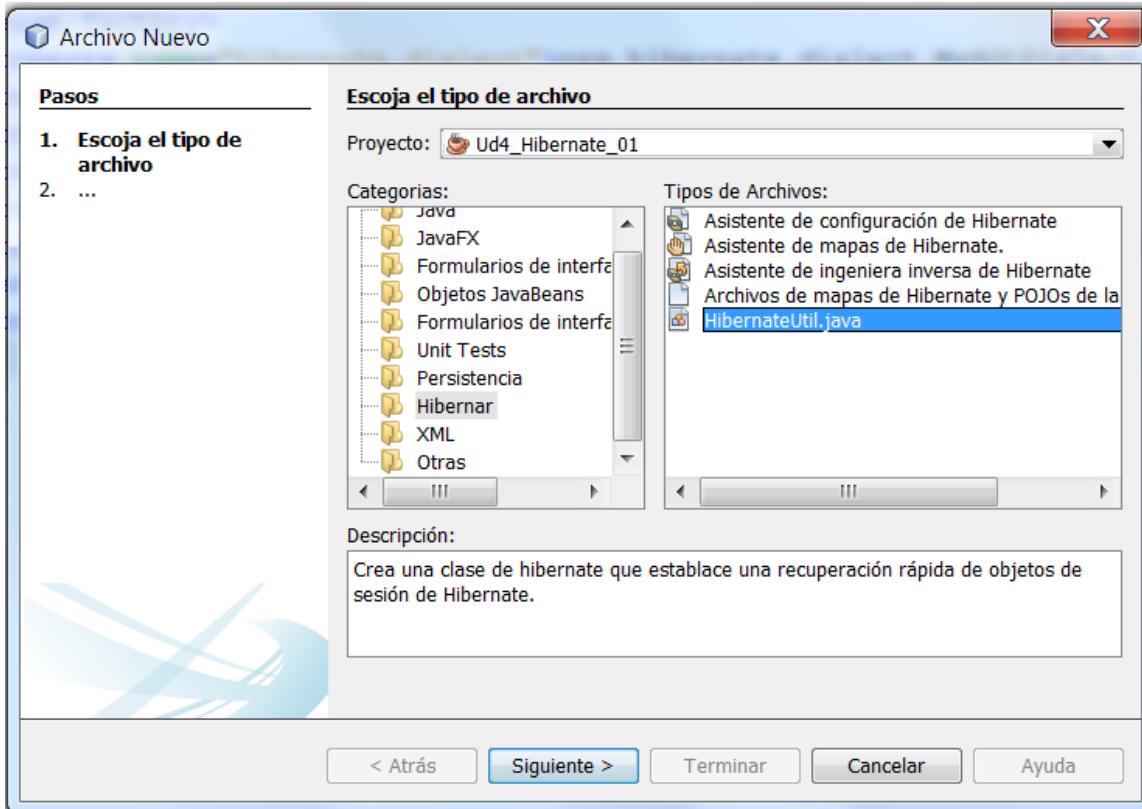
c) Crear la clase `HibernateUtil.java`

Para usar Hibernate es necesario crear una clase de ayuda (helper) para manejar el inicio y el acceso del objeto **SessionFactory** de Hibernate, para obtener un objeto **Session**.

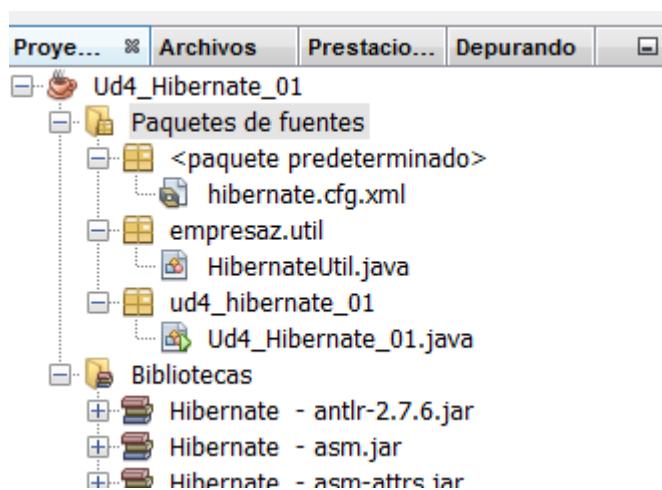
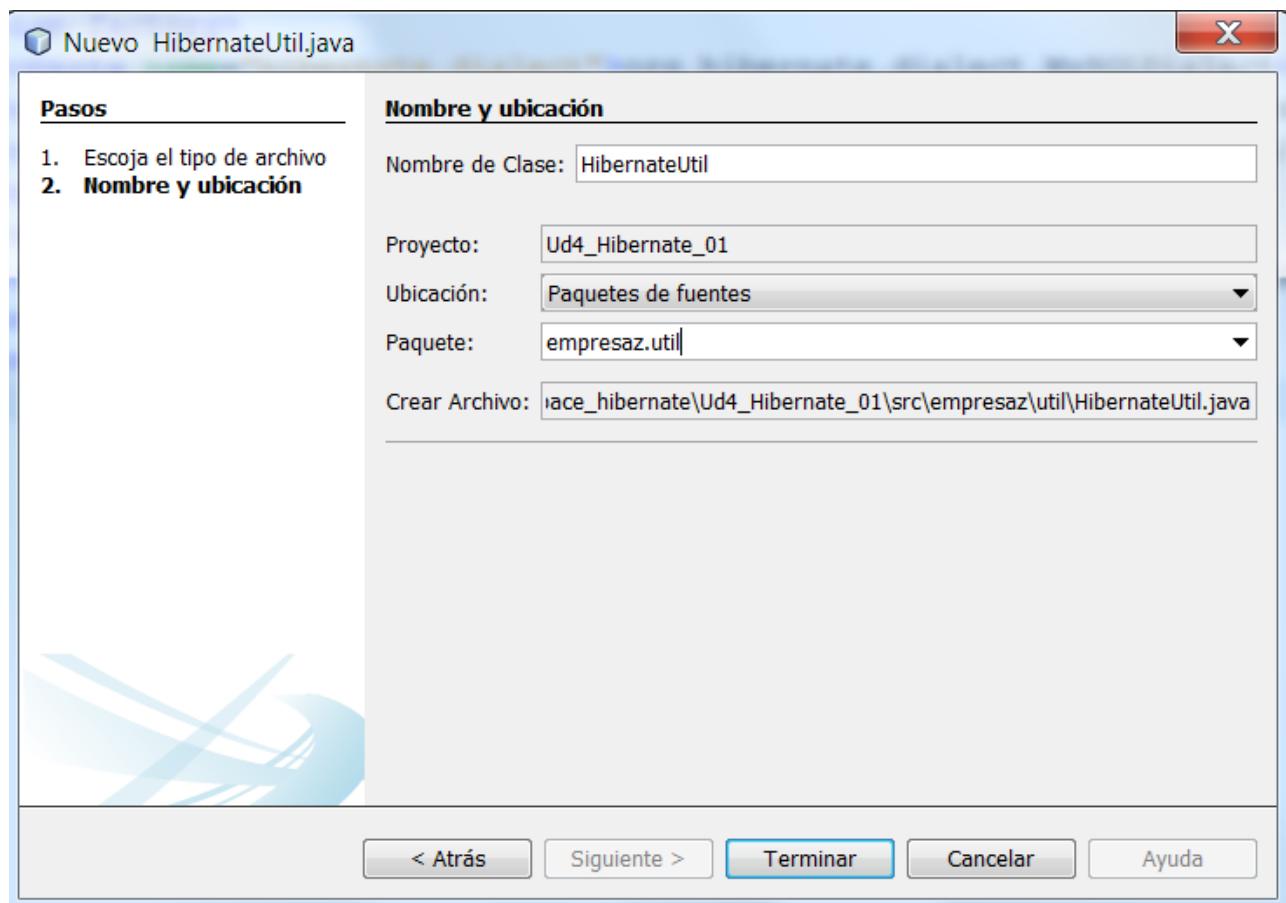
Esta clase llama al método de Hibernate **`configure()`**, que selecciona el fichero **`hibernate.cfg.xml`** y a partir de él construye `SessionFactory` para obtener el objeto `Session`.

En el nodo de **Paquetes de Fuentes** de la ventana Proyectos, pulsamos el botón derecho del ratón y seleccionamos Nuevo → Otro, para abrir el asistente de creación de archivos.

Seleccionamos Hibernate en la lista de Categorías y a **HibernateUtil.java** en la Lista de Tipos de Archivos.



Como Nombre de clase ponemos **HibernateUtil** y Paquete ponemos **empresaz.util**. Pulsamos a continuación el botón Terminar.



d) Archivos de mapeo hibernate y clases java

Podemos utilizar el asistente de Ingeniería Inversa y el Mapeo de Archivos de Hibernate y POJO desde una base de datos y crear múltiples POJO's y archivos de mapeo, basados en las tablas de la base de datos que estemos seleccionando.

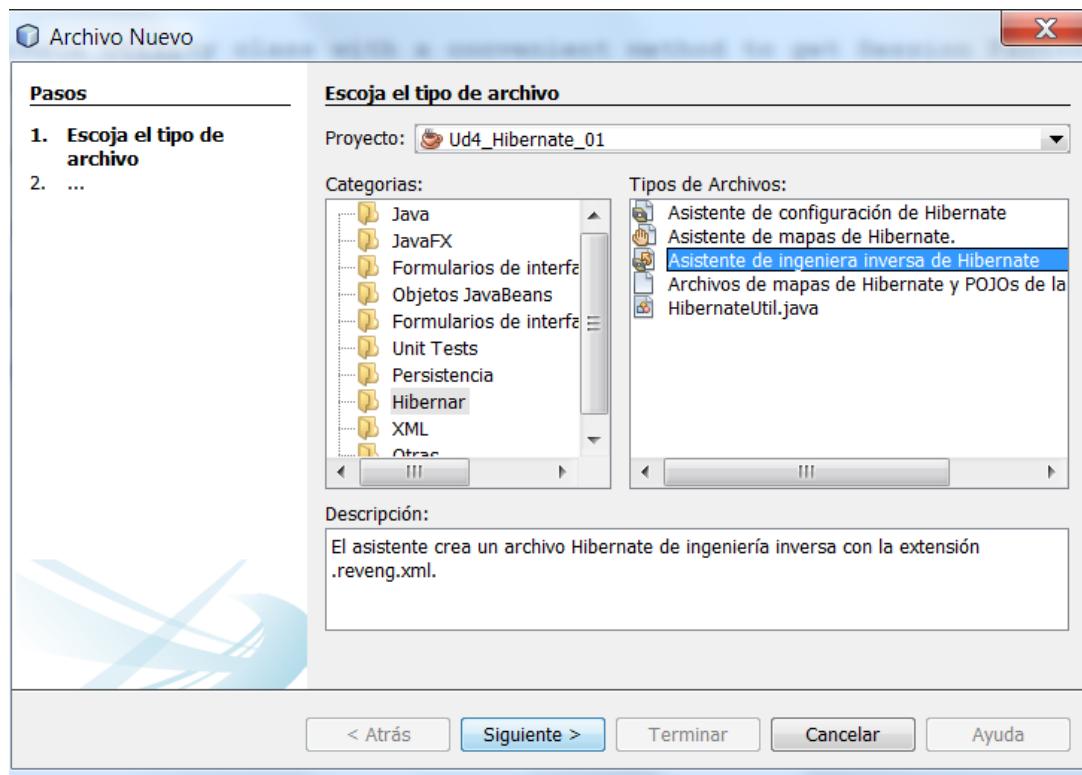
Como alternativa, podemos utilizar los asistentes de NetBeans para ayudarnos a crear POJO's individuales y archivos de mapeo.

El archivo de ingeniería inversa (**hibernate.reveng.xml**) es un archivo XML que puede ser usado para modificar la configuración por defecto cuando se generan los archivos de Hibernate desde los metadatos de bases de datos especificadas en hibernate.cfg.xml.

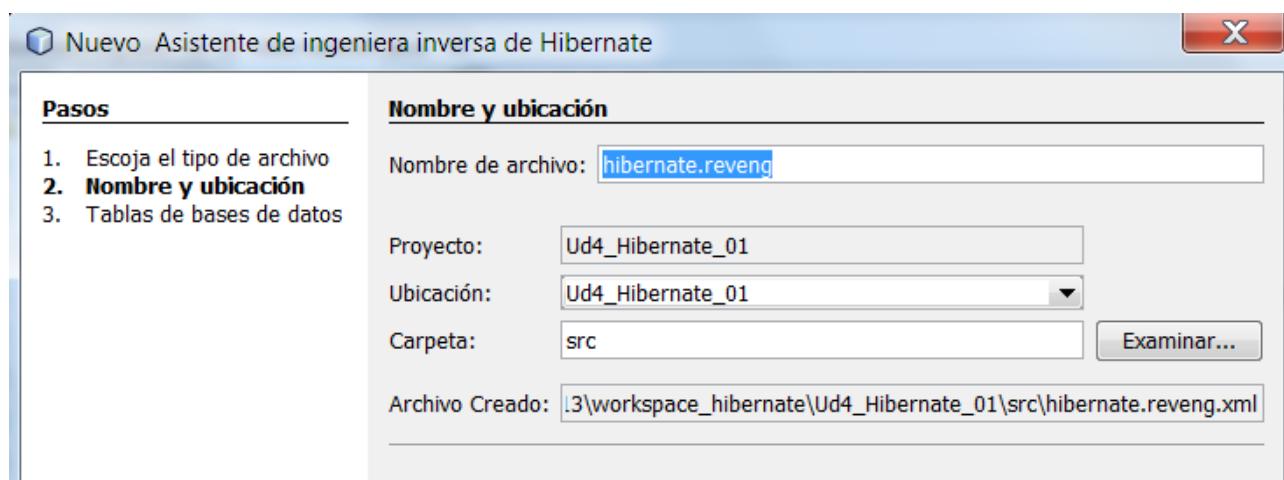
El asistente genera los archivos con la configuración por defecto. Podemos modificar el archivo para especificar de forma explícita el esquema de la base de datos que es usado, para filtrar las tablas que se desean dejar fuera puesto que no van a ser usadas y para especificar como los tipos JDBC van a ser mapeados a tipos Hibernate.

Para **crear el fichero de ingeniería inversa**, procedemos de la siguiente forma:

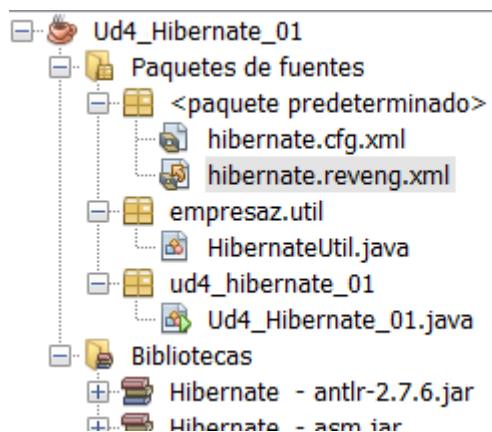
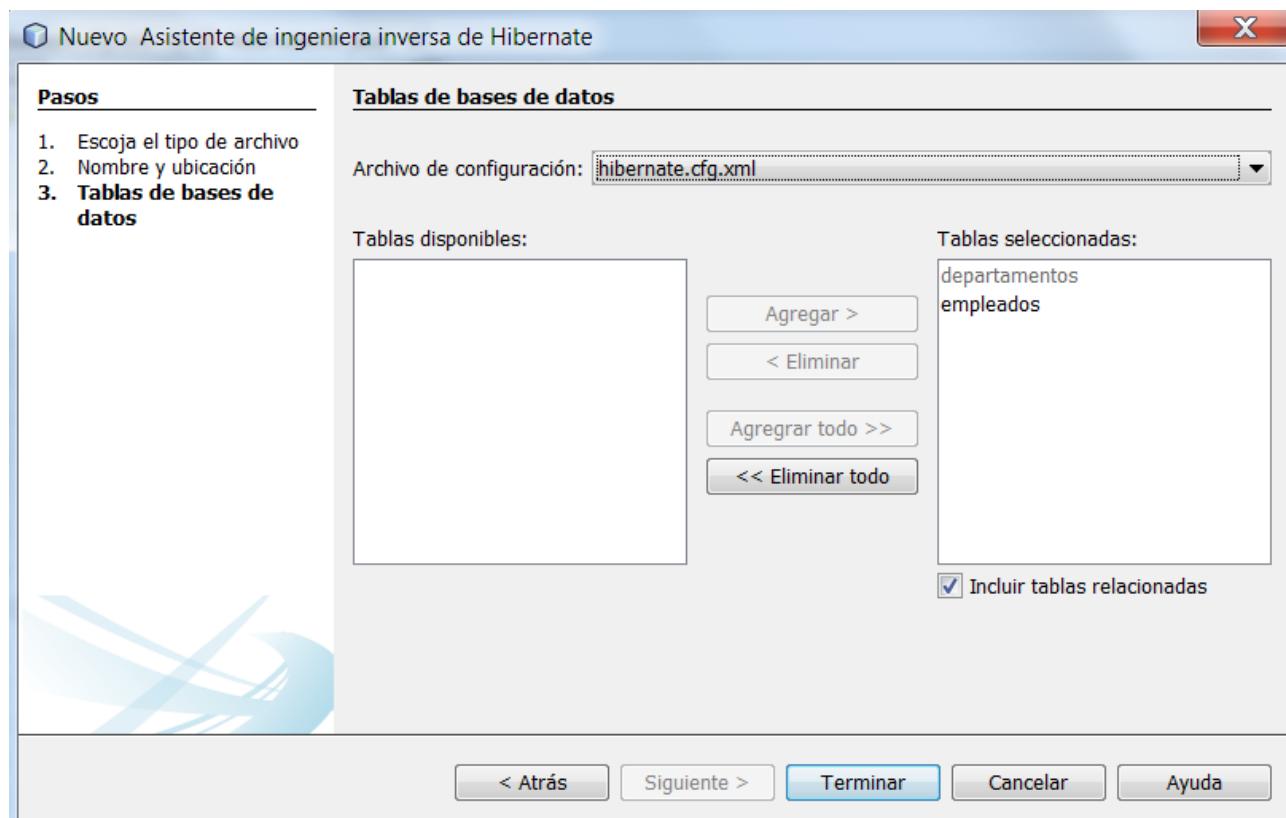
Hacemos click derecho sobre el nodo Paquetes de Fuentes, de nuestro proyecto, Nuevo→Otros, y después Hibernate→ **Asistente de ingeniería inversa de Hibernate**. Pulsamos siguiente.



Dejamos como nombre de archivo '**hibernate.reveng**' y pulsamos siguiente.



En la siguiente pantalla seleccionamos las dos tablas: EMPLEADOS y DEPARTAMENTOS, y pulsamos Terminar.



```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate Reverse Engineering//EN" "http://www.hibernate.org/dtd/hibernate-reverse-engineering.dtd">
<hibernate-reverse-engineering>
    <schema-selection match-catalog="empresaz"/>
    <table-filter match-name="empleados"/>
    <table-filter match-name="departamentos"/>
</hibernate-reverse-engineering>

```

Creación de los archivos de mapeo hibernate y POJOs

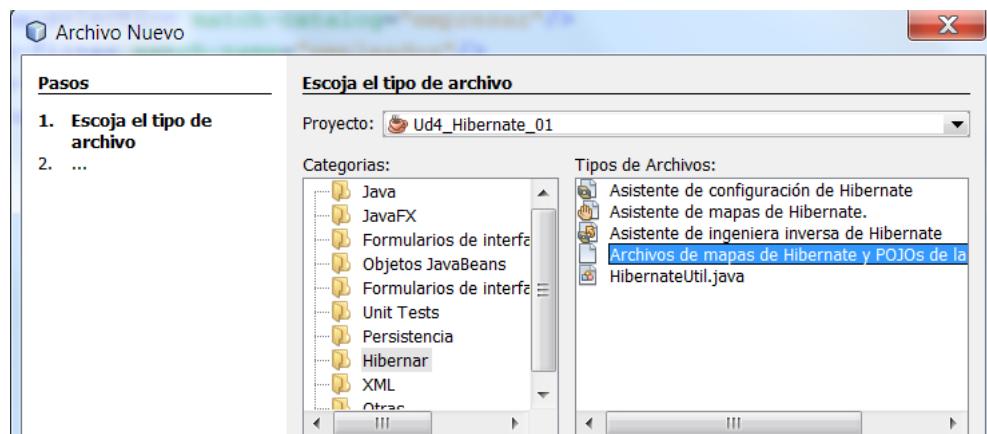
El asistente de archivos de mapeo y POJOs desde una base de datos, permite generar ficheros basados en tablas de la base de datos. Cuando usamos el asistente, NetBeans genera POJOs y archivos de mapeo basados en las tablas de la base de datos especificadas en **hibernate.reveng.xml** y los añade a las entradas de mapeo de **hibernate.cfg.xml**.

Cuando usamos el asistente podemos elegir los archivos que queremos que NetBeans genere y seleccionar las opciones de generación de código

Para la creación de los archivos de mapeo:

Hacemos click sobre el nodo Paquetes de fuentes, dentro de la ventana de proyectos.

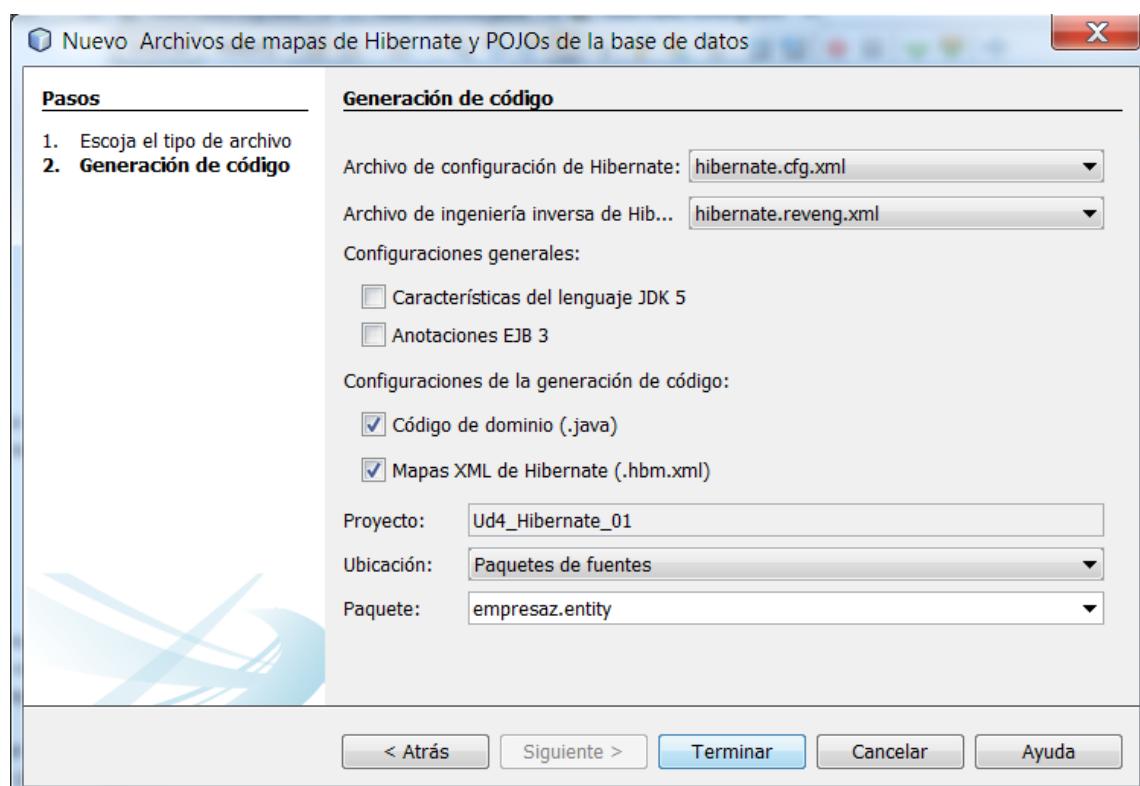
Seleccionamos Hibernate en Categoría y **Archivos de mapas de Hibernate y POJOs de base de datos**. Pulsamos siguiente.

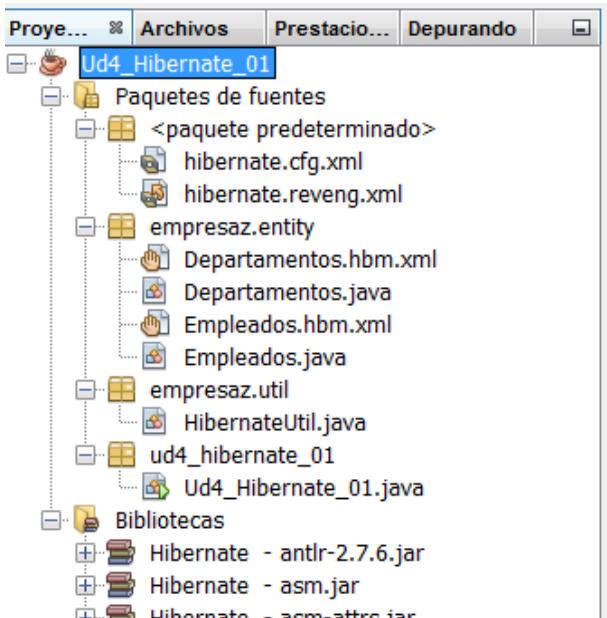


Seleccionamos hibernate.cfg.xml de la lista de Archivos de Configuración Hibernate, si no está seleccionado.

Seleccionamos hibernate.reveng.xml de la lista de Archivos de ingeniería inversa y nos aseguramos de que esté seleccionada la opción “Código de dominio (Java)” y la opción “Mapas XML de Hibernate”.

Escribimos empresaz.entity en el nombre del paquete y pulsamos el botón Terminar





Se habrá generado el paquete `empresaz.entity` con las clases Java de las tablas **EMPLEADOS** (`Empleados.java`) y **DEPARTAMENTOS** (`Departamentos.java`) que contiene los 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

Ya tenemos creada toda la estructura necesaria para crear la aplicación Java utilizando Hibernate y para probar las consultas HQL sobre la base de datos.

Vemos las clases generadas:

En **Empleados.java**, desaparece el atributo número de departamento y aparece un atributo de nombre 'departamentos', que es un objeto Departamentos y hace referencia al Departamento del empleado

```
public class Empleados implements java.io.Serializable {
    private short empNo;
    private Departamentos departamentos;
    private String apellido;
    private String oficio;
    private Short dir;
    private Date fechaAlt;
    private Float salario;
    private Float comision;
```

En **Departamentos.java** aparece un nuevo atributo 'empleadoses', tipo Set, para almacenar los empleados del departamento.

```
public class Departamentos implements java.io.Serializable {
    private byte deptNo;
    private String dnombre;
    private String loc;
    private Set empleadoses = new HashSet(0);
```

Mediante estos atributos se establecen las relaciones entre estos objetos.

También se generan los métodos **get** y **set** para estos atributos:

En Departamentos.java

```
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 getEmpleadoses() {
    return this.empleadoses;
}

public void setEmpleadoses(Set empleadoses) {
    this.empleadoses = empleadoses;
}
```

En Empleados.java

```
public short getEmpNo() {
    return this.empNo;
}

public void setEmpNo(short empNo) {
    this.empNo = empNo;
}

public Departamentos getDepartamentos() {
    return this.departamentos;
}

public void setDepartamentos(Departamentos departamentos) {
    this.departamentos = departamentos;
}
```

Vemos los archivos de mapeo generados

Departamentos.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 09-ene-2013 11:32:31 by Hibernate Tools 3.2.1.GA -->
<hibernate-mapping>
    <class name="empresaz.entity.Departamentos" table="departamentos" catalog="empresaz">
        <id name="deptNo" type="byte">
            <column name="dept_no" />
            <generator class="assigned" />
        </id>
        <property name="dnombre" type="string">
            <column name="dnombre" length="15" />
        </property>
        <property name="loc" type="string">
            <column name="loc" length="15" />
        </property>
        <set name="empleadoses" inverse="true">
            <key>
                <column name="dept_no" not-null="true" />
            </key>
            <one-to-many class="empresaz.entity.Empleados" />
        </set>
    </class>
</hibernate-mapping>
```

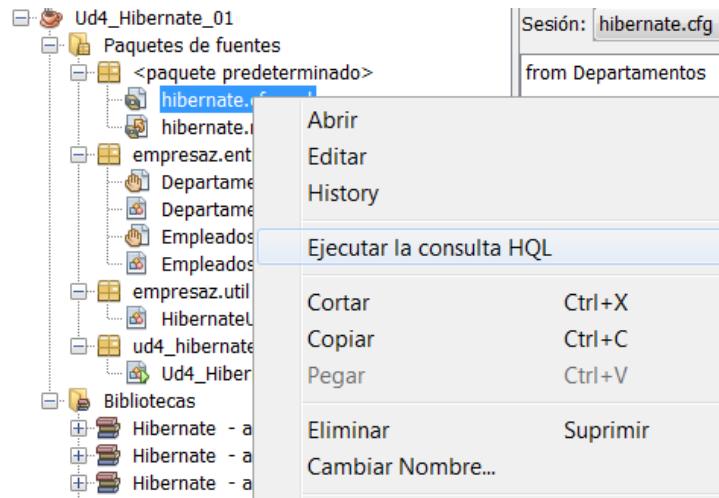
Empleados.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 09-ene-2013 11:32:31 by Hibernate Tools 3.2.1.GA -->
<hibernate-mapping>
    <class name="empresaz.entity.Empleados" table="empleados" catalog="empresaz">
        <id name="empNo" type="short">
            <column name="emp_no" />
            <generator class="assigned" />
        </id>
        <many-to-one name="departamentos" class="empresaz.entity.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 name="comision" type="java.lang.Float">
            <column name="comision" precision="6" />
        </property>
    </class>
</hibernate-mapping>
```

5 . Realizar consultas para probar conexión.

Vamos a realizar consultas en HQL para comprobar si la conexión a la base de datos funciona correctamente

- Sobre **hibernate.cfg.xml**, click derecho y seleccionamos ‘Ejecuta consulta HQL’.



Aparecerán las ventanas de consulta HQL. Redactamos la consulta

`from Departamentos`

y click en botón ‘Ejecutar consulta’

The screenshot shows the NetBeans IDE interface after executing the HQL query. The code editor still contains 'from Departamentos'. Below it, the 'Resultado' tab of the results window is active, displaying a table with 65 rows of department data. The table has columns: DeptNo, Dnombre, Loc, and Emplead... (partially visible). The data includes various department names like 'CONTABIL...', 'INVESTIG...', 'VENTAS', etc., located in cities like 'SEVILLA', 'MADRID', 'ALMERÍA', etc. A red box highlights the 'Ejecutar' button in the toolbar above the results table.

DeptNo	Dnombre	Loc	Emplead...
10	CONTABIL...	SEVILLA	[empresa...]
15	INFORMÁ...	MADRID	[]
20	INVESTIG...	MADRID	[empresa...]
25	INFORMA...	MADRID	[]
28	ANDROID	ALMERÍA	[]
30	VENTAS	BARCELONA	[empresa...]
35	INFORMA...	MADRID	[]
38	ANDROID	ALMERÍA	[]
39	ANDROID	ALMERÍA	[]
40	PRODUCCI...	BILBAO	[empresa...]
41	ANDROID	ALMERÍA	[]
42	ANDROID	ALMERÍA	[]
45	INFORMA...	MADRID	[]
49	ANDROID	ALMERÍA	[]
51	ANDROID	ALMERÍA	[]
61	ANDROID	ALMERÍA	[]
62	ANDROID	ALMERÍA	[]
63	ANDROID	ALMERÍA	[]
64	ANDROID	ALMERÍA	[]
65	ANDROID	ALMERÍA	[]

Ten en cuenta que las consultas HQL:

- El * no se puede utilizar a la derecha de SELECT (select * from Empleados, da error)
- Diferencia mayúsculas y minúsculas. (Hay que tener cuidado con las mayúsculas y minúsculas, respetando los nombres de las clases y las propiedades).

Otra consulta: Departamentos que empiezan por A

```
from Departamentos d where d.dnombre like 'A%'
```

Sesión: hibernate.cfg

```
from Departamentos d where d.dnombre like 'A%'
```

Resultado SQL

0 Fila(s) actualizada(s); 12 fila(s) seleccionada(s).

DeptNo	Dnombre	Loc	Emplead...
28	ANDROID	ALMERÍA	[]
38	ANDROID	ALMERÍA	[]
39	ANDROID	ALMERÍA	[]
41	ANDROID	ALMERÍA	[]
42	ANDROID	ALMERÍA	[]
49	ANDROID	ALMERÍA	[]
51	ANDROID	ALMERÍA	[]
61	ANDROID	ALMERÍA	[]
62	ANDROID	ALMERÍA	[]
63	ANDROID	ALMERÍA	[]
64	ANDROID	ALMERÍA	[]
65	ANDROID	ALMERÍA	[]

Otras consultas que se pueden realizar desde el entorno, al estilo de SQL son:

```
select empNo, apellido, salario from Empleados where dept_no=10
```

Sesión: hibernate.cfg

```
select empNo, apellido, salario from Empleados where dept_no=10
```

Resultado SQL

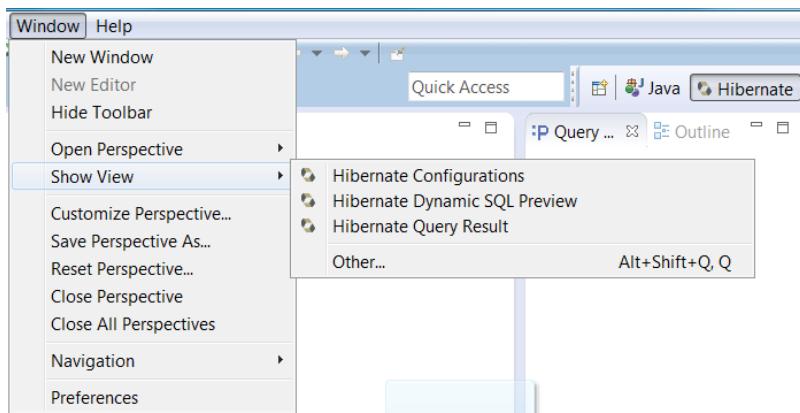
0 Fila(s) actualizada(s); 7 fila(s) seleccionada(s).

Columna 1	Columna 2	Columna 3
1000	AMARO	2721.74
1001	ALONSO	2101.22
2000	CASTRO	2644.16
2001	CASTILLO	2023.64
3000	HERNANDEZ	4195.51
3001	HERMIDA	642.97
4455	PEPE	1500.0

4.2 Instalación y configuración de Hibernate en Eclipse

1- Instalar el plugin de Hibernate para Eclipse. (en Eclipse SR1 Juno para JEE)

- Teniendo una conexión a Internet, iniciamos eclipse y pulsamos: Help → Install New Software
- Rellenamos el campo Work With con la URL: <http://download.jboss.org/jbosstools/updates/stable/> y pulsamos el botón Add, nos pide un nombre, ponemos Hibernate y pulsamos OK.
- Aparecen la lista de plugins. Pulsamos en flechita DataServices y seleccionamos solo HibernateTools. El resto de opciones las desmarcamos. Pulsamos Next y empieza la descarga.
- Una vez descargado, se visualiza una ventana con los detalles del elemento a instalar, pulsamos de nuevo Next. A continuación aceptamos la licencia y pulsamos el botón Finish. Comienza el proceso de instalación que tardará unos minutos.
- Una vez instalado pide reiniciar Eclipse. Para comprobar que se ha instalado correctamente podemos pulsar en la opción de menú Windows → Show View → Other, y deben aparecer las opciones de Hibernate. También se puede comprobar desde el menú Window → Open Perspective → Other → Hibernate.

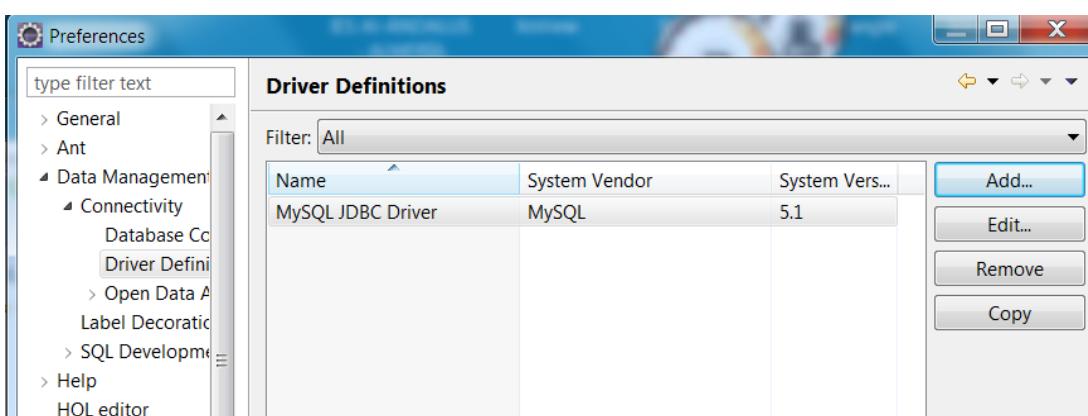


2- Configuración del Driver MySQL en Eclipse.

Una vez instalado Hibernate, hay que configurarlo para que se comunique con MySQL.

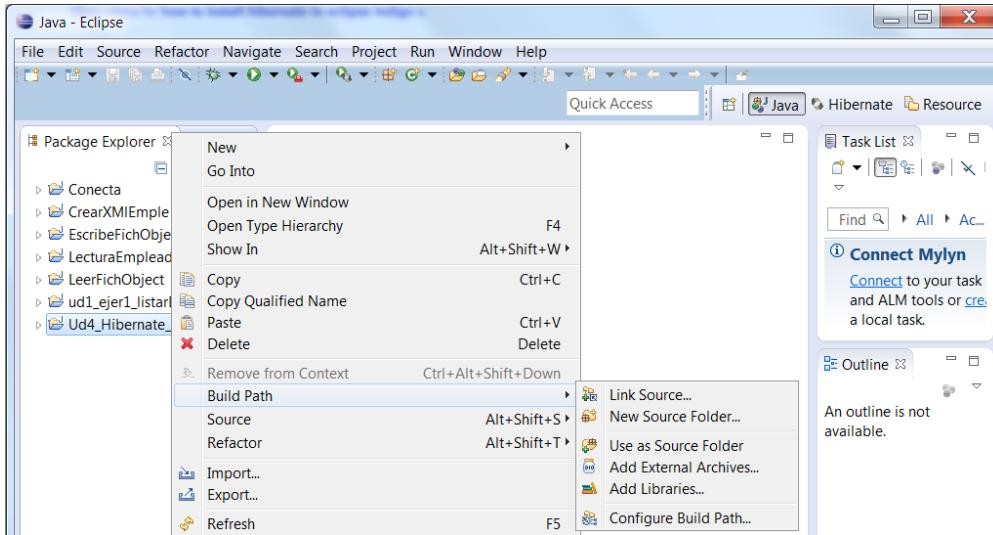
Hay que tener instalado el conector JDB de MySQL (mysql-connector-java-5.1.22.zip), lo descomprimimos en la misma carpeta de eclipse.

- En eclipse, desde el menú Window → Preferences → Data Management → Connectivity → Driver Definitions se pulsa Add
- Desde la pestaña Name/Type se selecciona MySQL JDBC Driver en la versión 5.1 (o le ponemos el nombre)
- Desde la pestaña JAR List pulsamos el botón Add Jar/Zip para localizar el conector anterior, se debe seleccionar el fichero mysql-connector-java-5.1.22.zip (en la carpeta de eclipse). Si se visualiza otro driver mysql-connector-java-5.1.xx.jar, lo eliminaremos primero, y añadiremos el nuestro.
- Desde la pestaña properties puede ser necesario establecer la BD empresaz.

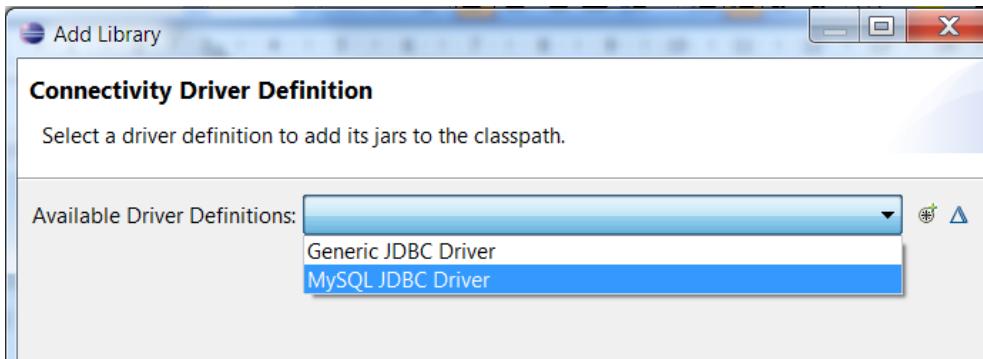


3- Crear proyecto y configurar Hibernate para que se comunique con MySQL

- Crear un proyecto Java de nombre Ud4_Hibernate_01. Hacer click derecho sobre el proyecto y seleccionar Build Path → Add Libraries



Seleccionamos la opción Connectivity Driver Definitions y después MySQL JDBC Driver



El proyecto debe mostrar un aspecto similar al siguiente:

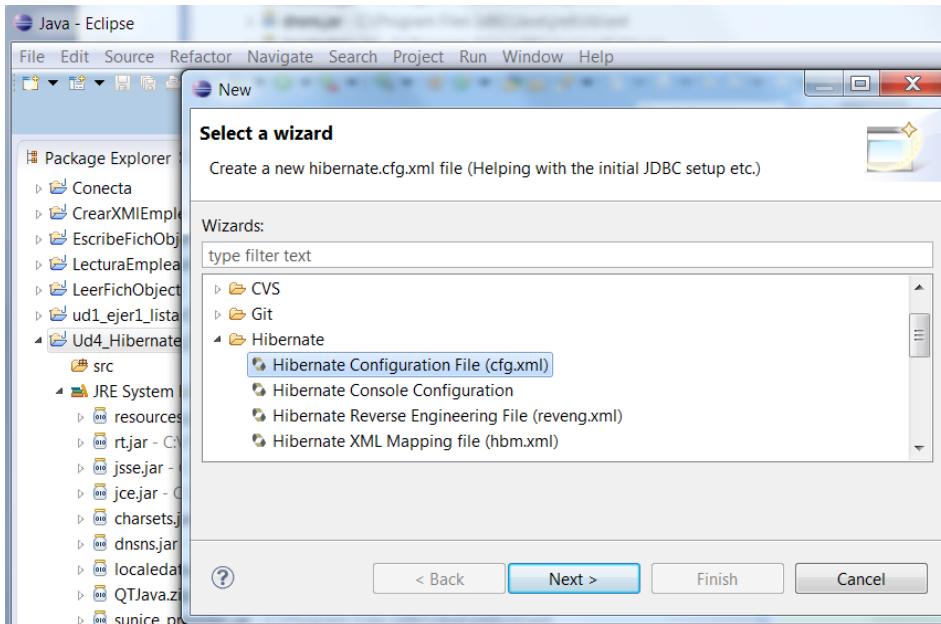


4- Configuración de Hibernate

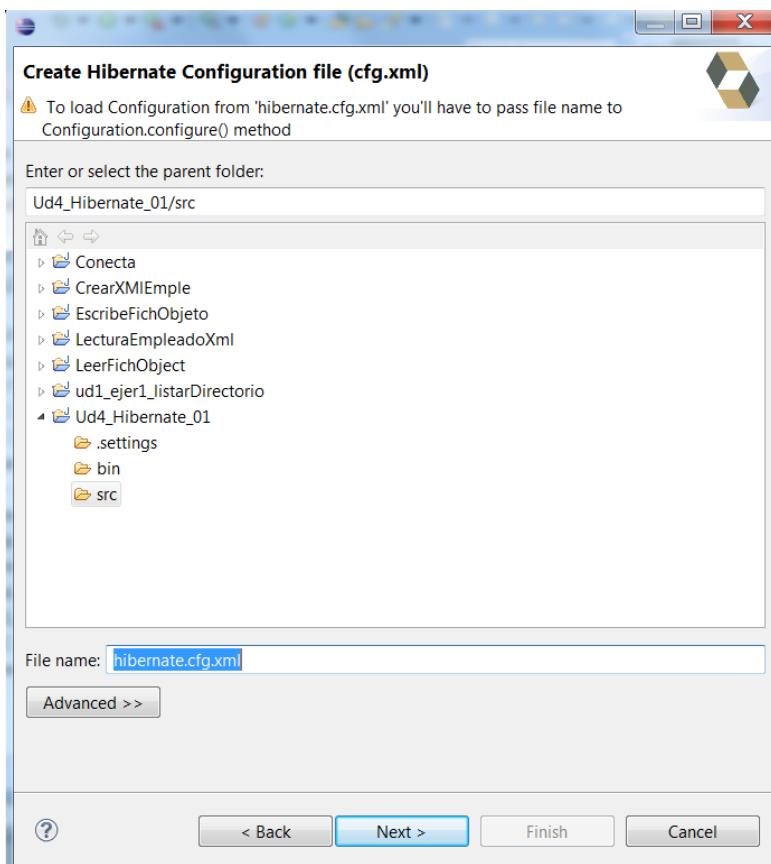
Una vez que tenemos el driver MySQL en nuestro proyecto, hemos de crear el fichero de configuración de Hibernate **hibernate.cfg.xml** (fichero XML con todo lo necesario para realizar la conexión a la base de datos).

a) Crear el fichero hibernate.cfg.xml

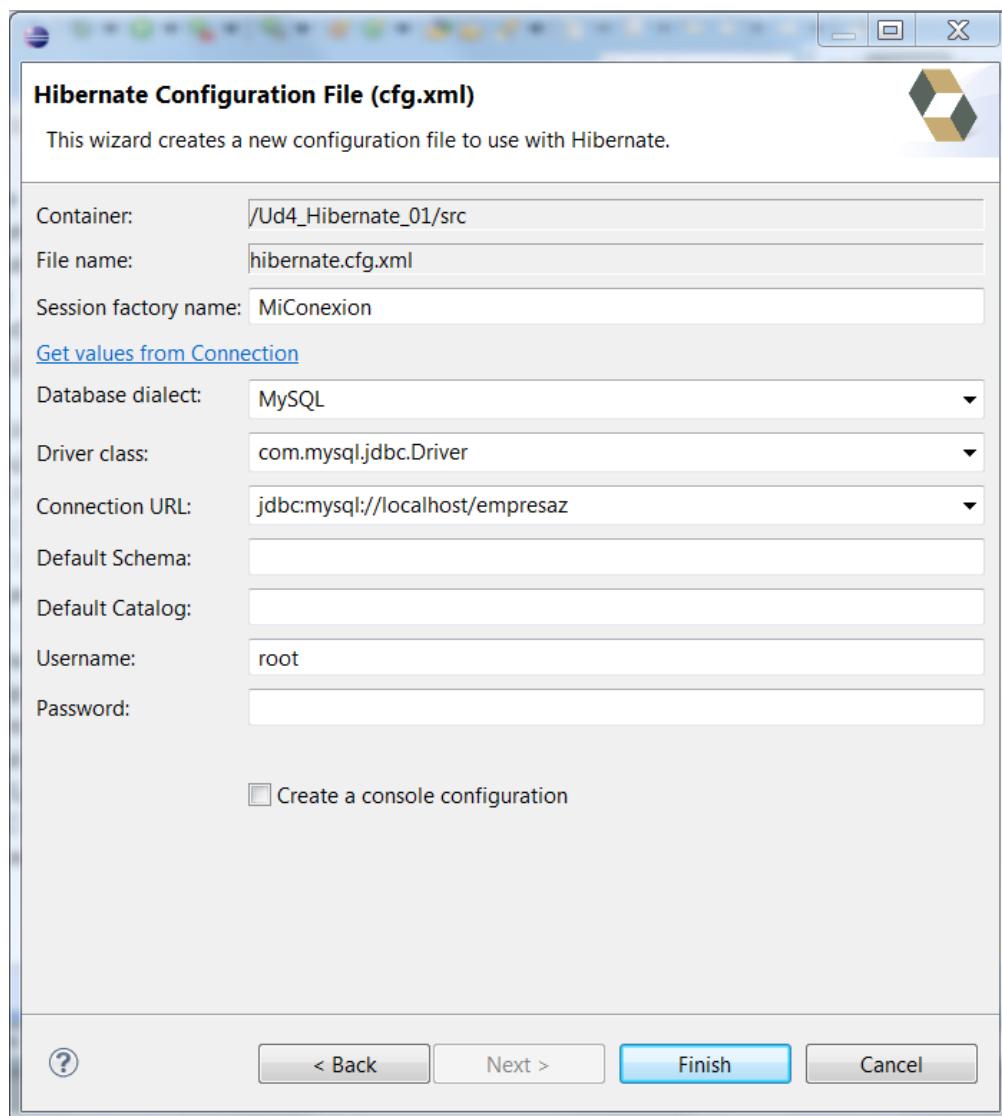
- seleccionamos proyecto y pulsamos: New→Other→Hibernate Configuration File (cfg.xml)



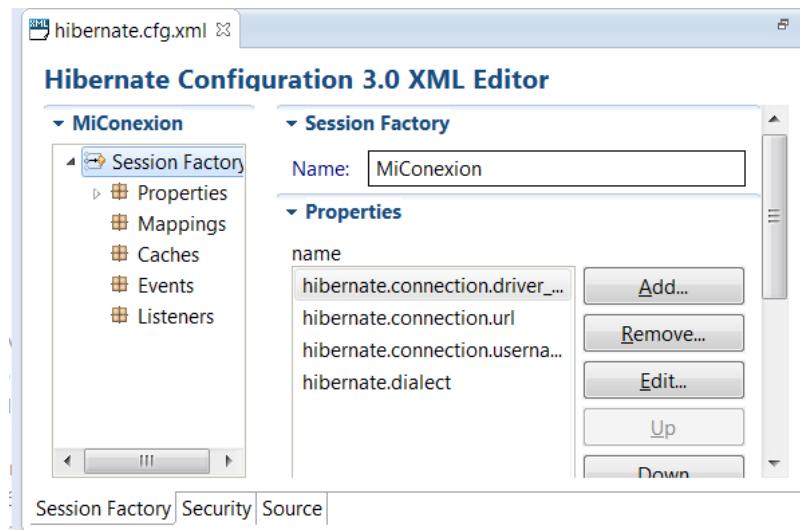
Lo creamos en la carpeta **src**:



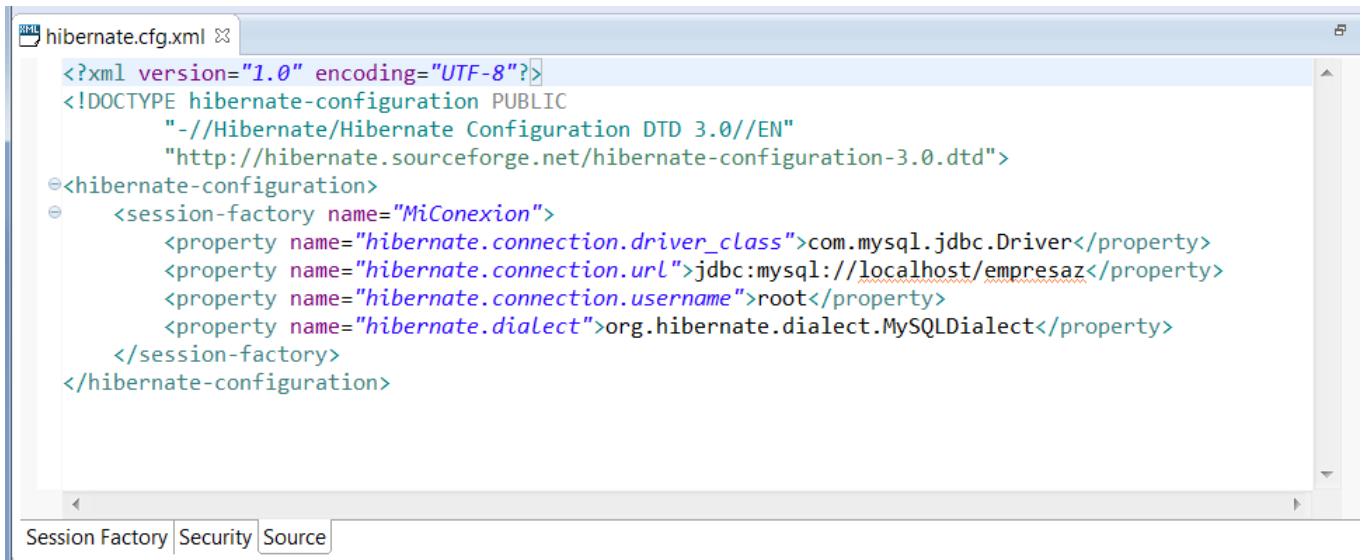
Pulsamos Next, y cumplimentamos los datos de conexión:



Al pulsar sobre **Finish** aparece el editor de configuración de Hibernate



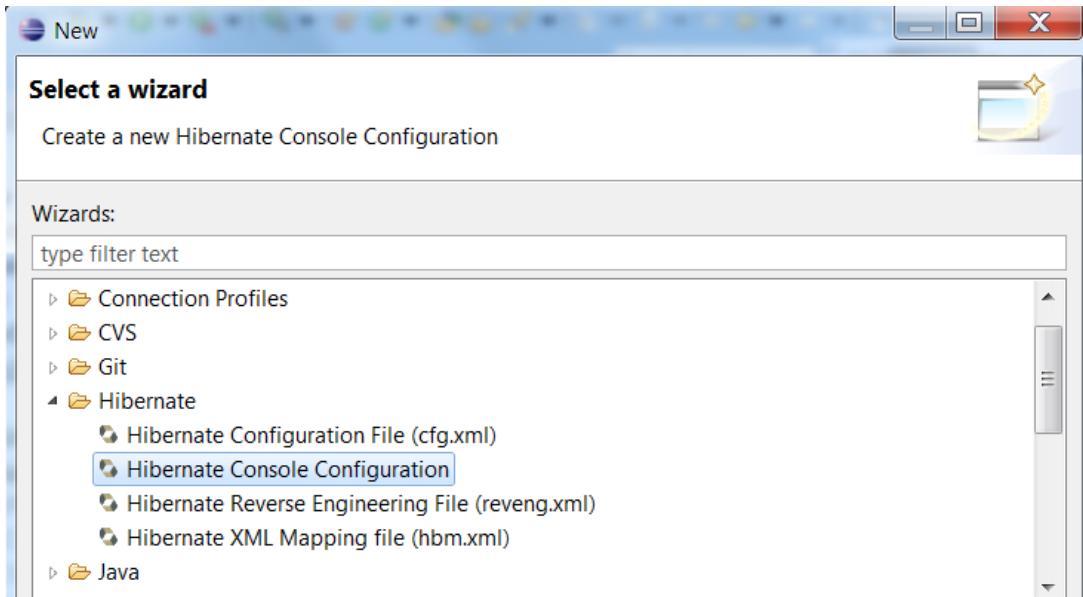
Desde la pestaña Source, se puede ver la configuración del fichero XML cfg.xml



```
<?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.url">jdbc:mysql://localhost/empresaz</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
</session-factory>
</hibernate-configuration>
```

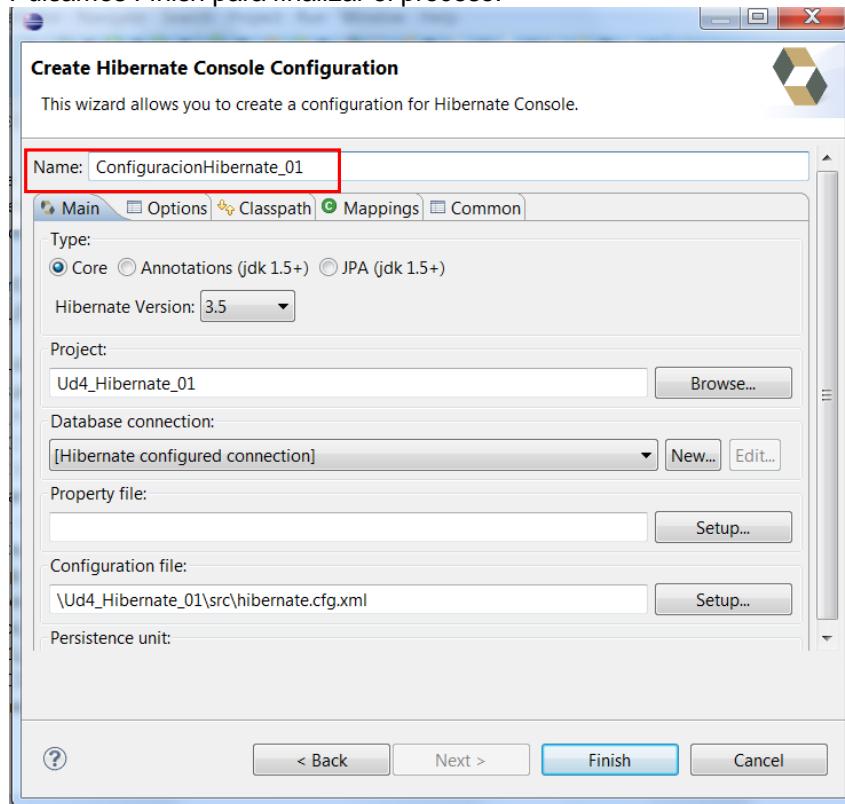
b) Crear el fichero XML Hibernate Console Configuration

Seleccionamos el proyecto, click derecho y seleccionamos New→Other→Hibernate→Hibernate Console Configuration y Next



Aparece la siguiente ventana donde ponemos el nombre (en Name: ConfiguracionHibernate_01) Y nos aseguramos de que aparece el fichero de configuración creado anteriormente, hibernate.cfg.xml.

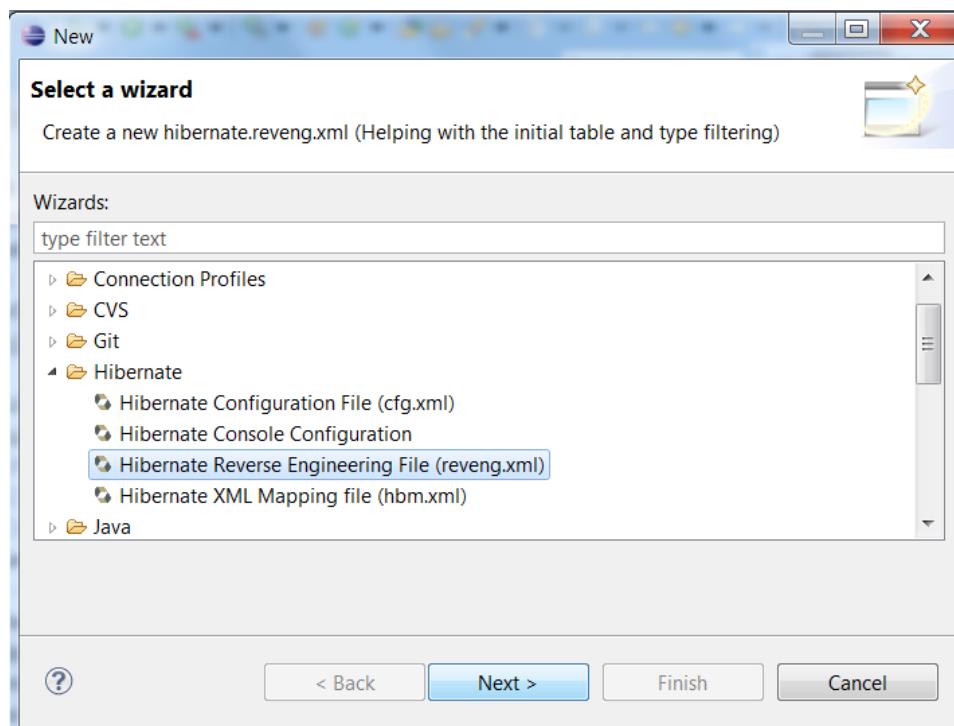
Pulsamos Finish para finalizar el proceso.



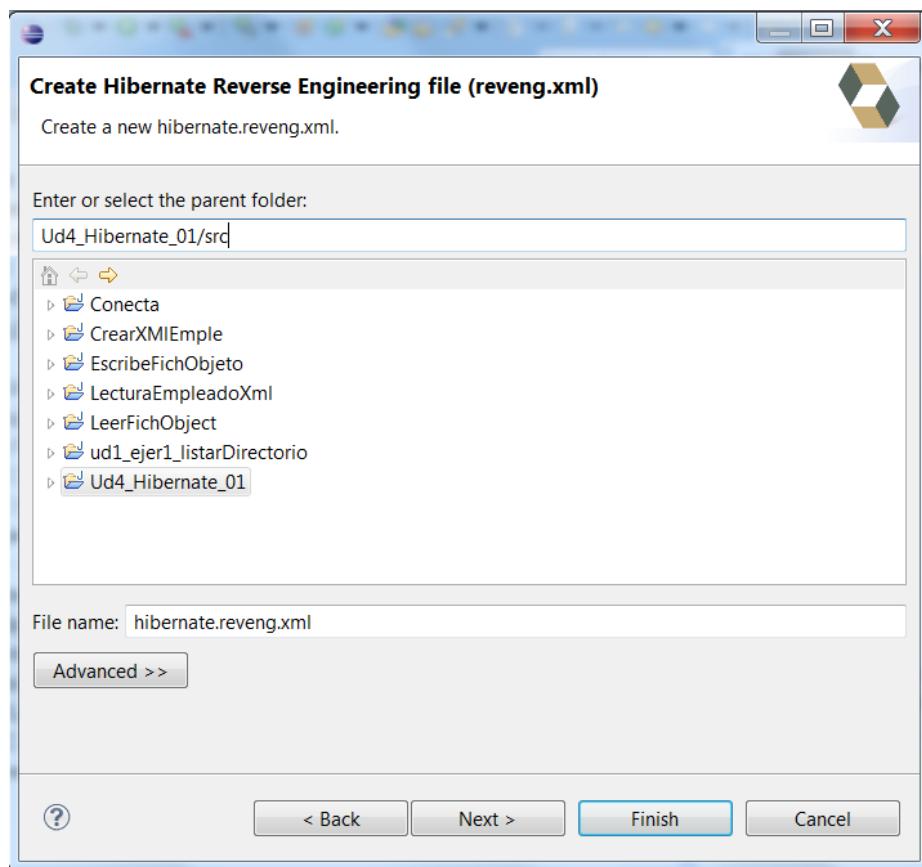
c) Crear el fichero XML Hibernate Reverse Engineering (reveng.xml)

Este fichero es el encargado de crear las clases de nuestras tablas MySQL

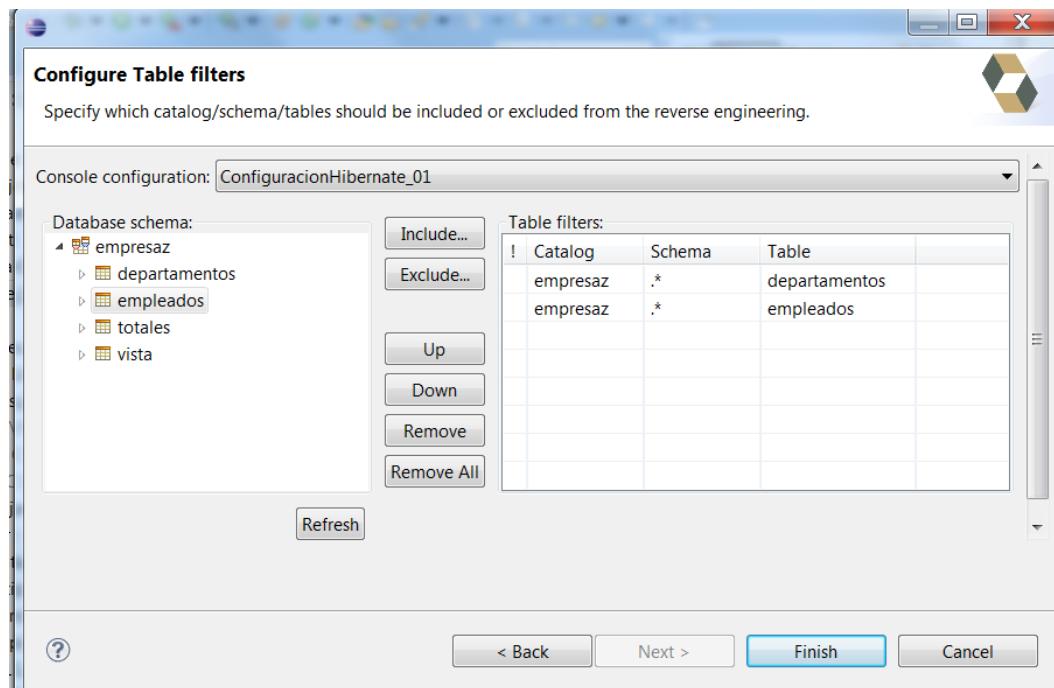
Pulsamos click derecho sobre el proyecto, New→Other→Hibernate→ Hibernate Reverse EngineeringFile (reveng.xml). Pulsamos el botón Next .



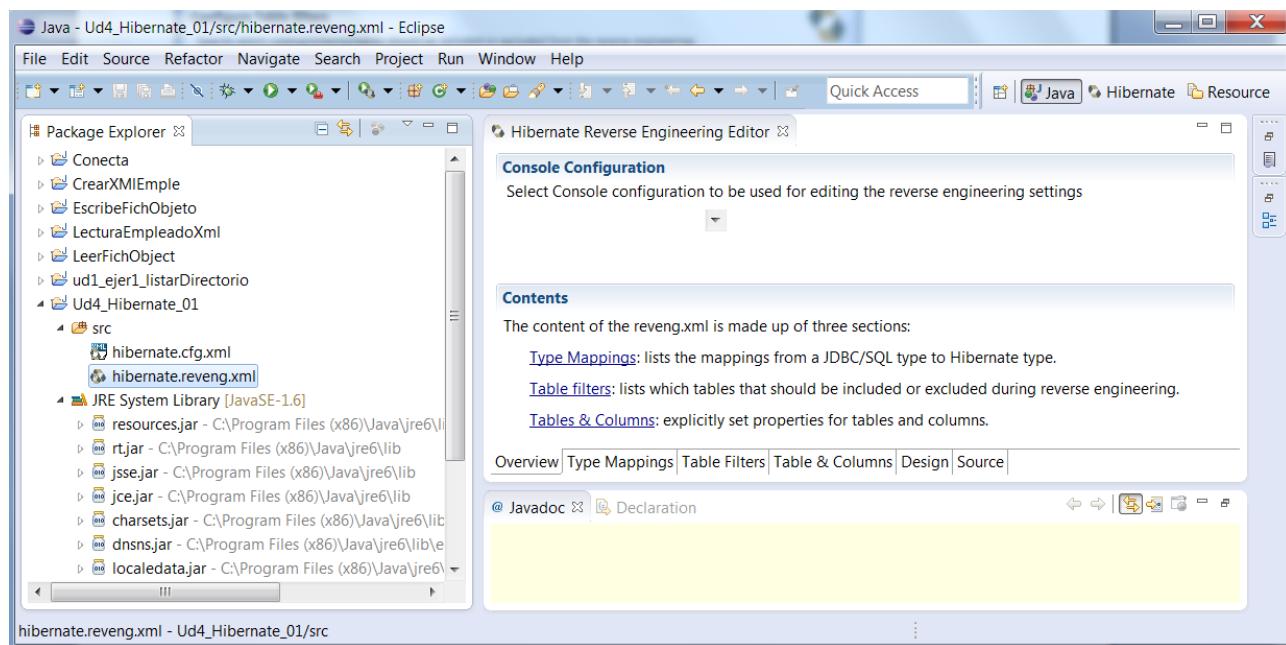
Nos pide que indiquemos dónde se va a guardar el fichero. Se debe guardar en la misma carpeta que el fichero hibernate.cfg.xml, en este caso src.



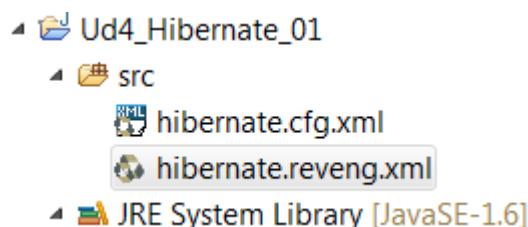
Pulsamos Next, y en la siguiente ventana, seleccionamos nuestra conexión y las tablas que nos interesan de la base de datos empresaz



Pulsamos Finish y aparece la ventana del editor de Hibernate Reverse Engineering



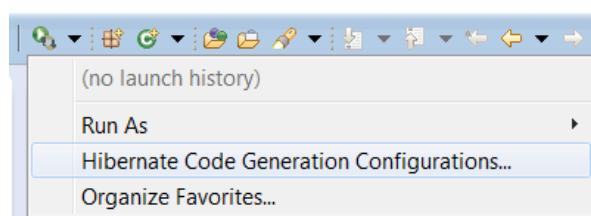
Y desde la pestaña source, se puede ver y editar el fichero XML reveng.xml generado:

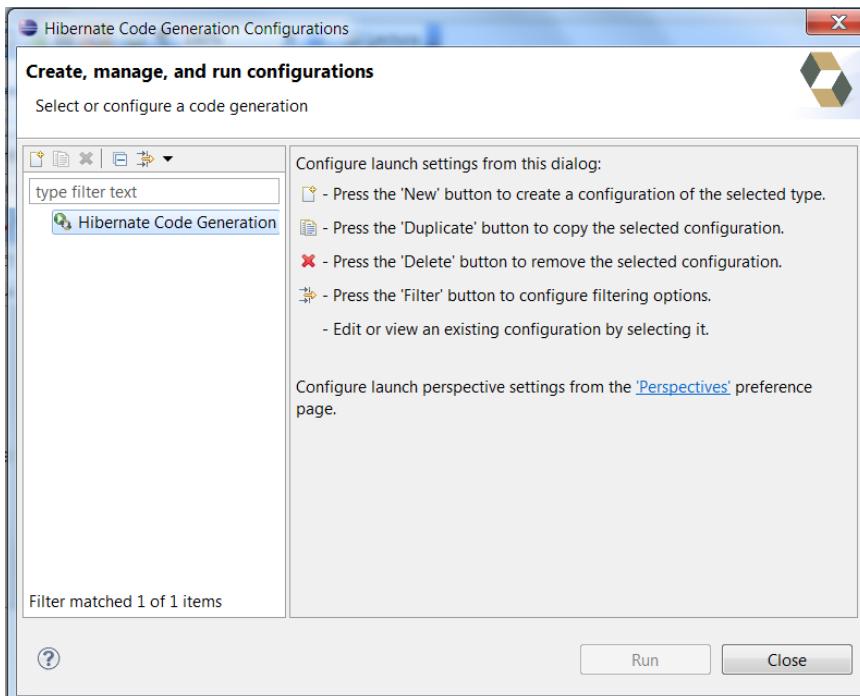


d) Generar las clases de la base de datos

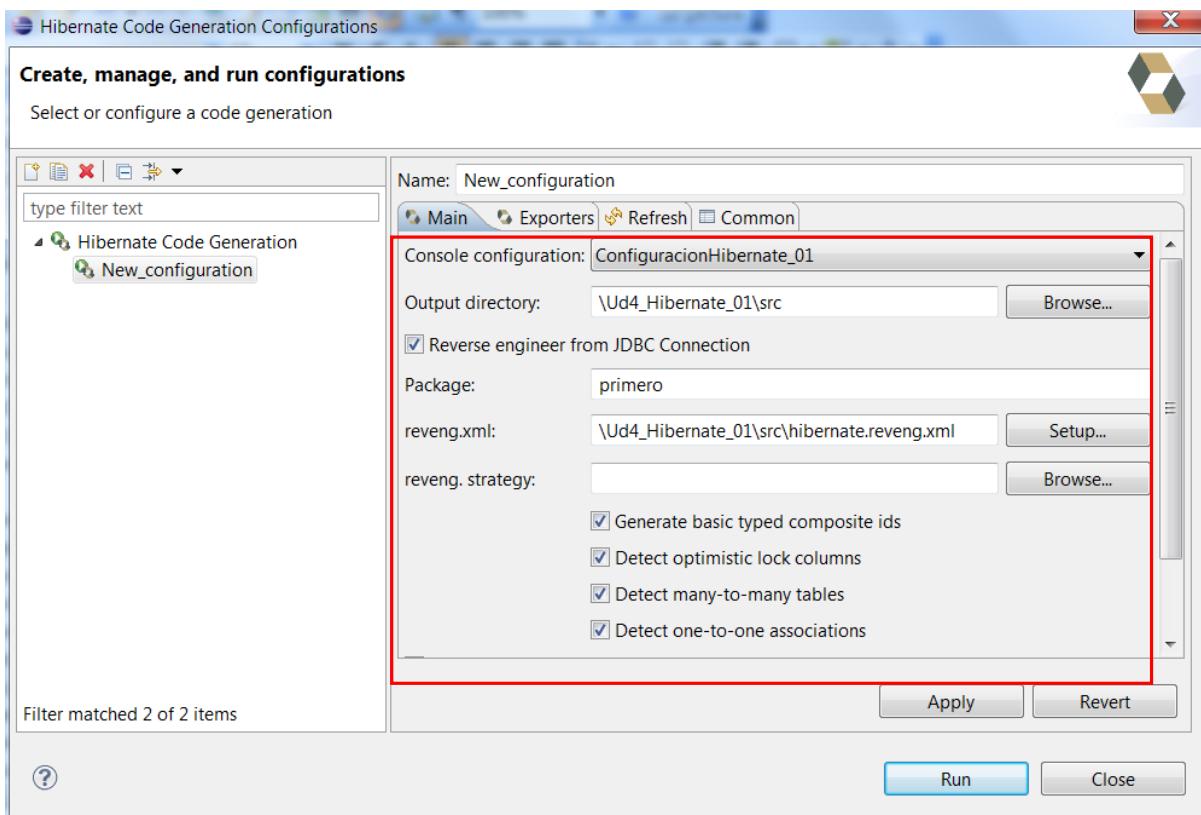
El siguiente paso es generar las clases de la base de datos empresaz.

Para ello pulsamos la flechita situada a la derecha del botón Run As y seleccionamos Hibernate Code Generation Configurations

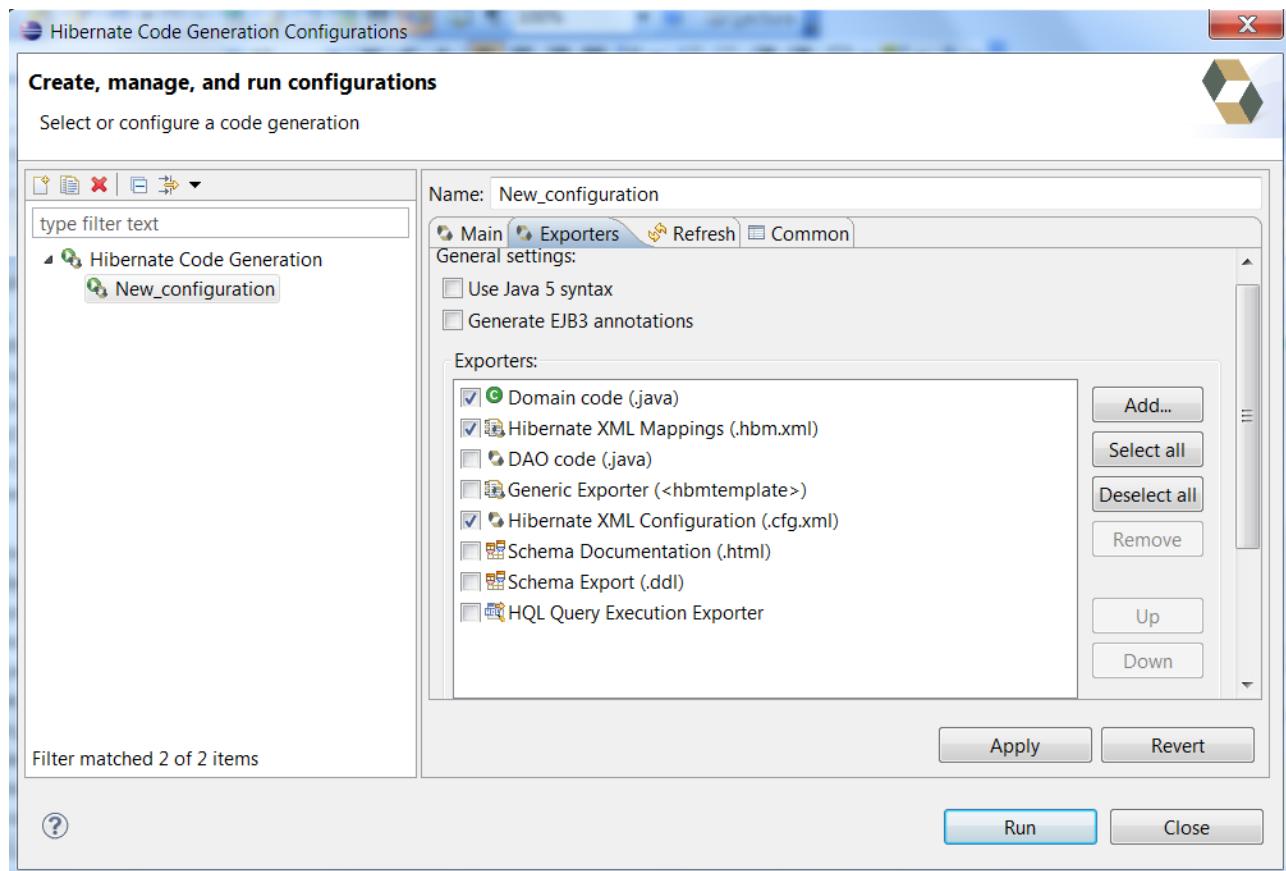




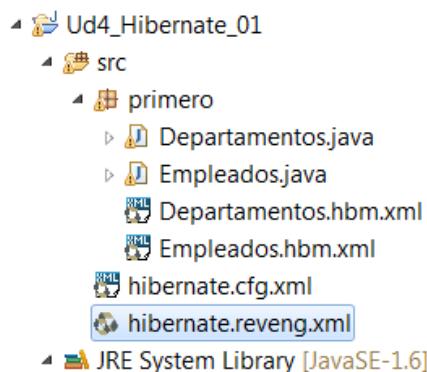
Hacemos doble click en Hibernate Code Generation. En la pestaña Main indicamos los siguientes valores:



En la pestaña Exporters se indican los ficheros que queremos generar, se marcan las casillas : Domain code, Hibernate XML Mappings e Hibernate XML Configuration. Una vez seleccionados se pulsa Apply y después Run.



Tras Run, se habrá generado el paquete primero con las clases Java de las tablas EMPLEADOS (**Empleados.java**) y DEPARTAMENTOS (**Departamentos.java**) que contiene los 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:



Pueden aparecer algunos Warnings en las clases Departamentos y Empleados, por ejemplo “*The serializable class Departamentos does not declare a static final serialVersionUID field of type long*”, esto es porque las clases implementan la interfaz Serializable. Para solucionarlo se pulsa con el ratón sobre el símbolo de warning para que nos dé las posibles soluciones. Por ejemplo se añade:

```
private static final long serialVersionUID = 1L;
```

Igual en los otros warnings:
`@SuppressWarnings("rawtypes")`

Se observa variación en los tipos generados para la base de datos.

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

    private static final long serialVersionUID = 1L;
    private byte deptNo;
    private String dnombre;
    private String loc;
    @SuppressWarnings("rawtypes")
    private Set empleadoses = new HashSet(0);
```

Por ejemplo, en **Departamentos.java** aparece un nuevo atributo 'empleadoses', tipo Set, para almacenar los empleados del departamento.

En **Empleados.java**, desaparece el atributo número de departamento y aparece un atributo de nombre 'departamentos', que es un objeto Departamentos y hace referencia al Departamento del empleado.

```
/*
 * Empleados generated by hbm2java
 */
@SuppressWarnings("serial")
public class Empleados implements java.io.Serializable {

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

Mediante estos atributos se establecen las relaciones entre estos objetos. También se generan los métodos **get** y **set** para estos atributos:

En Empleados.java

```
public Departamentos getDepartamentos() {
    return this.departamentos;
}

public void setDepartamentos(Departamentos departamentos) {
    this.departamentos = departamentos;
}
```

En Departamentos.java

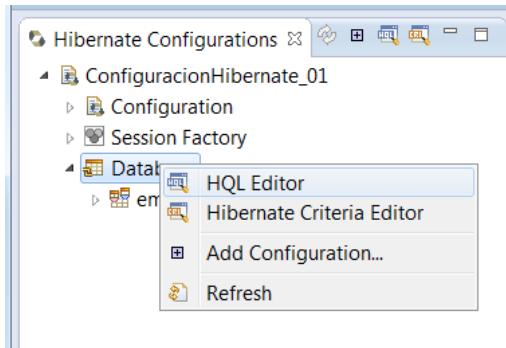
```
@SuppressWarnings("rawtypes")
public Set getEmpleadoses() {
    return this.empleadoses;
}

@SuppressWarnings("rawtypes")
public void setEmpleadoses(Set empleadoses) {
    this.empleadoses = empleadoses;
}
```

5. Realizar consultas para probar conexión

Vamos a realizar consultas en HQL para comprobar si la conexión a la base de datos funciona correctamente.

- Abrimos la perspectiva de Hibernate desde el menú Window→Open Perspective→Other→Hibernate
- Pulsamos en nuestra configuración 'ConfiguracionHibernate_01', después click derecho sobre Database→HQL Editor

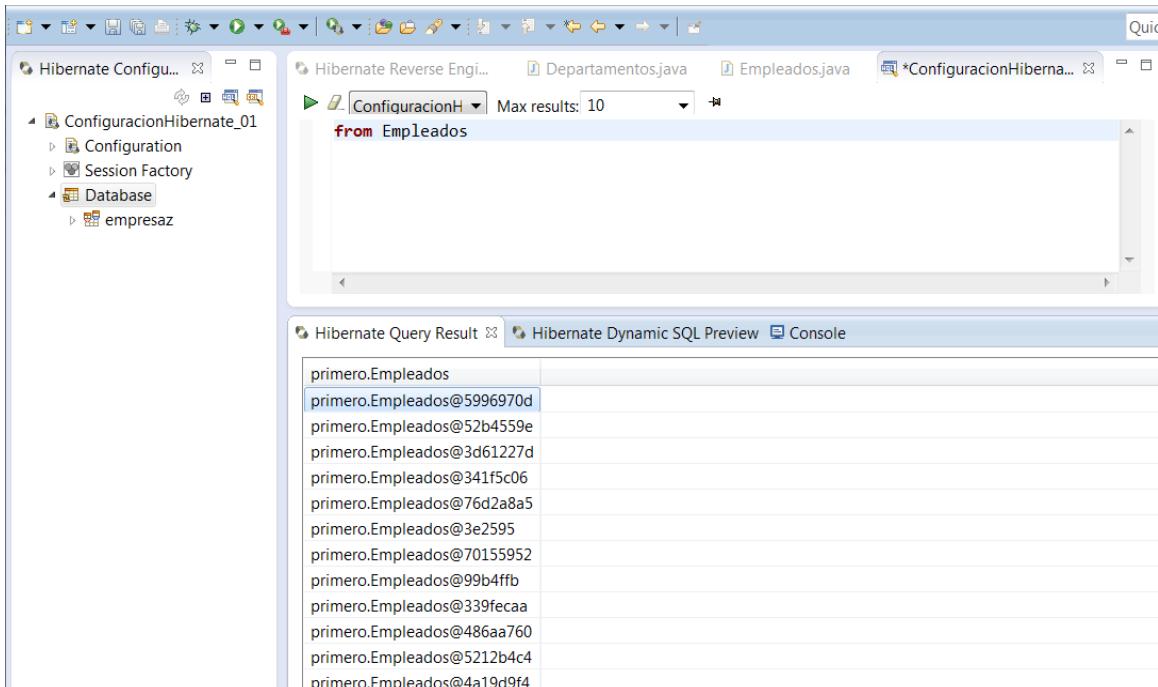


Nos pregunta si queremos abrir la sesión, pulsamos en Yes.

A continuación escribimos el siguiente código HQL desde la pestaña ConfiguracionHibernate_01:

`from Empleados`, y pulsamos el botón '**ejecutar la consulta**'.

En la pestaña Hibernate Query Result aparecen los resultados de la consulta.



Otras consultas que se pueden realizar desde entorno, al estilo de SQL son:

The screenshot shows the Hibernate Query Editor interface. At the top, there are tabs for 'Hibernate Reverse Engi...', 'Departamentos.java', and 'Empleados.java'. Below the tabs, a query is entered in the editor pane:

```
select dnombre, loc, deptNo from Departamentos
```

Below the editor, there are three tabs: 'Hibernate Query Result', 'Hibernate Dynamic SQL Preview', and 'Console'. The 'Hibernate Query Result' tab is selected, displaying the following data:

0	1	2
CONTABILIDAD	SEVILLA	10
INFORMÁTICA	MADRID	15
INVESTIGACIÓN	MADRID	20
INFORMÁTICA	MADRID	25
ANDROID	ALMERÍA	28
VENTAS	BARCELONA	30
INFORMÁTICA	MADRID	35
ANDROID	ALMERÍA	38
ANDROID	ALMERÍA	39
PRODUCCIÓN	BILBAO	40

The screenshot shows the Hibernate Query Editor interface again. The configuration tab is now labeled 'ConfiguracionHibern...' and contains the following query:

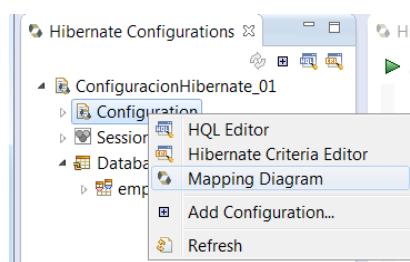
```
select empNo, apellido, salario from Empleados where dept_no=10
```

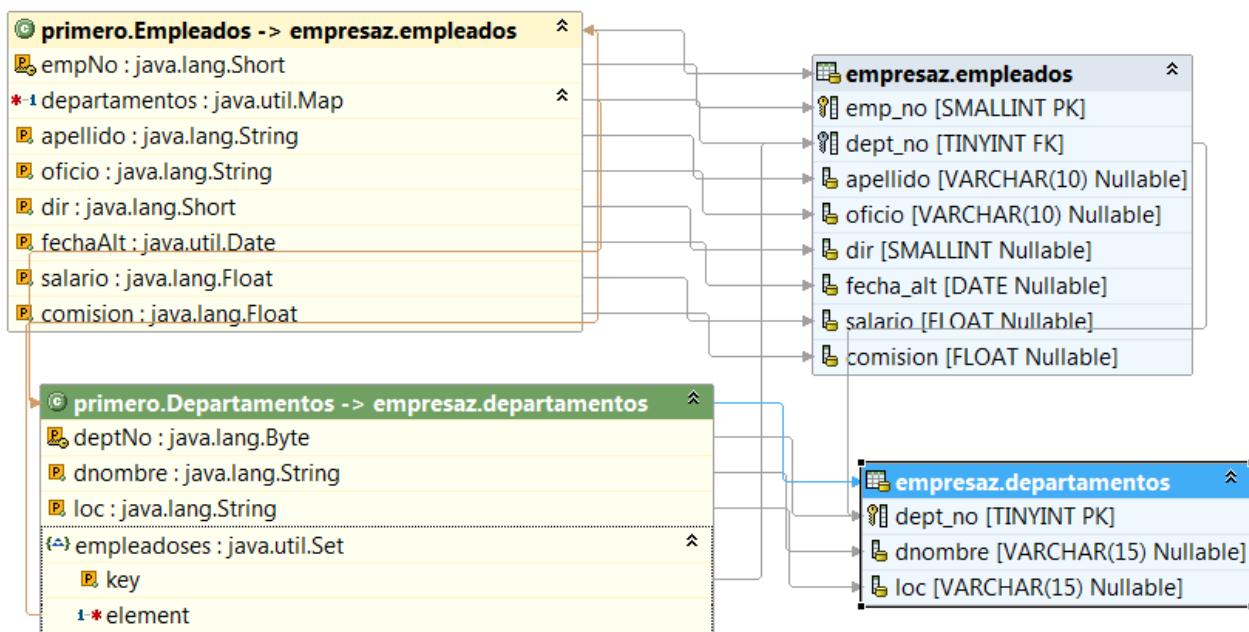
The results are displayed in the 'Hibernate Query Result' tab:

0	1	2
1000	AMARO	2721.74
1001	ALONSO	2101.22
2000	CASTRO	2644.16
2001	CASTILLO	2023.64
3000	HERNANDEZ	4195.51
3001	HERMIDA	642.97

- Ten en cuenta que el * no se puede utilizar a la derecha de SELECT (select * from Empleados, da error)
- Hay que tener cuidado con las mayúsculas y minúsculas, respetando los nombres de las clases y las propiedades.

Desde la perspectiva Hibernate y pulsando con el botón derecho del ratón en Configuration, seleccionamos Mapping Diagram para visualizar el diagrama de mapeo entre las clases java y las tablas de la base de datos. Se muestra un diagrama similar al siguiente:



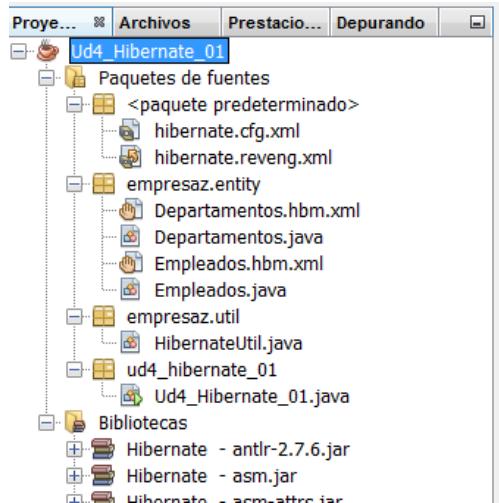


5. Programación con Hibernate

Haremos un ejemplo muy sencillo para probar el funcionamiento correcto de la configuración realizada tanto en NetBeans como en Eclipse.

5.1 En NetBeans

La estructura del proyecto Java **Ud4_Hibernate_01**, configurado para trabajar con Hibernate en el apartado anterior, es la siguiente:



Donde hemos dejado todo configurado para comenzar a programar.

Realizaremos un ejemplo sencillo para ilustrar cómo consultar todos los departamentos y cómo insertar un nuevo objeto Departamentos en la tabla departamentos.

Creamos el método *listadoDepartamentos()* y el método *insertDepartamento()* dentro de **Ud4_Hibernate_01.java**

Obtiene el listado de todos los departamentos de la base de datos

```
public static void listadoDepartamentos() {
    //Obtiene sesión y la abre
    SessionFactory sfactory = HibernateUtil.getSessionFactory();
    Session session = sfactory.openSession();
    System.out.println("=====");
    System.out.println("Listado de Departamentos");

    Departamentos depar = new Departamentos();
    //consulta con HQL
    Query q = session.createQuery("from Departamentos");
    //obtiene un iterador para recorrer el resultado de la consulta
    Iterator<?> iter = q.iterate();
    //mientras haya resultados
    while (iter.hasNext()) {
        // extraer el siguiente objeto
        depar = (Departamentos) iter.next();
        System.out.println(depar.getDeptNo() + "*" + depar.getDnombre());
    }
    System.out.println("=====");
    //cierra sesión
    session.close();
}
```

import: org.hibernate.SessionFactory, org.hibernate.classic.Session, org.hibernate.Query, java.util.Iterator

Inserta un nuevo objeto Departamentos en la base de datos

```

public static void insertaDepartamento() {
    Session session = null;
    try {
        //obtiene una sesión
        SessionFactory sfactory = HibernateUtil.getSessionFactory();
        //abre la sesión
        session = sfactory.openSession();
        //comienza una transacción
        Transaction tx = session.beginTransaction();
        System.out.println("Inserta una fila en la tabla Departamentos");
        //inserta el departamento 60 en la tabla departamentos
        Departamentos dep = new Departamentos();
        dep.setDeptNo((byte) 60);
        dep.setDnombre("MARKETING");
        dep.setLoc("GUADALAJARA");
        session.save(dep); //almacena el objeto
        tx.commit();
    } catch (HibernateException he) {
        System.out.println("Error " + he.getCause() + ", " + he.getMessages());
    } finally {
        session.close();
    }
}

```

import: org.hibernate.SessionFactory, org.hibernate.classic.Session, org.hibernate.Transaction, org.hibernate.HibernateException

5.2 En Eclipse.

1. Es necesario descargar la última distribución de Hibernate desde la URL.

<http://sourceforge.net/projects/hibernate/files/hibernate3/>

2. Creamos una carpeta de nombre Hibernate dentro de la carpeta de eclipse (**C:\eclipse\Hibernate**) y dentro de ella copiamos las librerías que necesitamos. Descomprimimos el fichero descargado, por ejemplo en Hibernate.

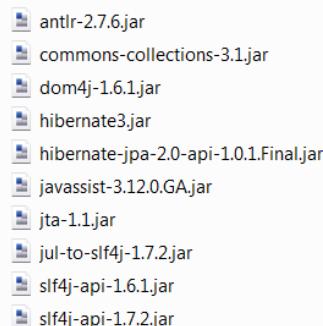
3. En la carpeta descomprimida buscamos el fichero **hibernate3.jar** y lo copiamos a **C:\eclipse\Hibernate**

4. En la carpeta descomprimida, seleccionamos todos los ficheros de **\lib\required** y los copiamos a **C:\eclipse\Hibernate**

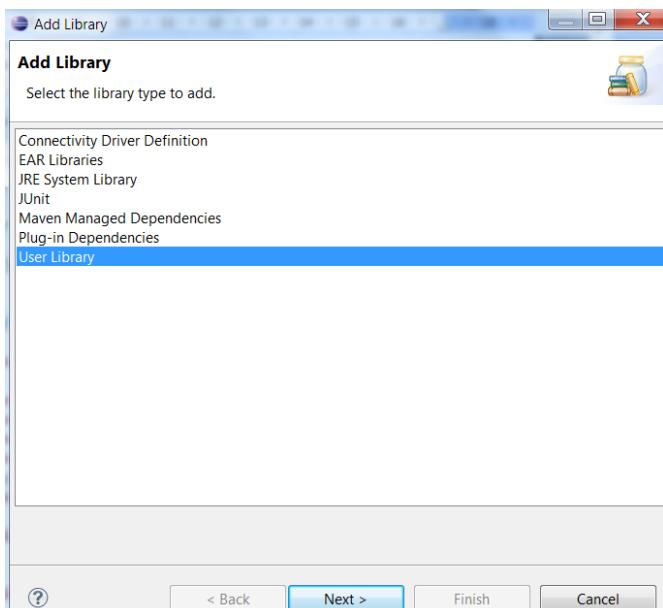
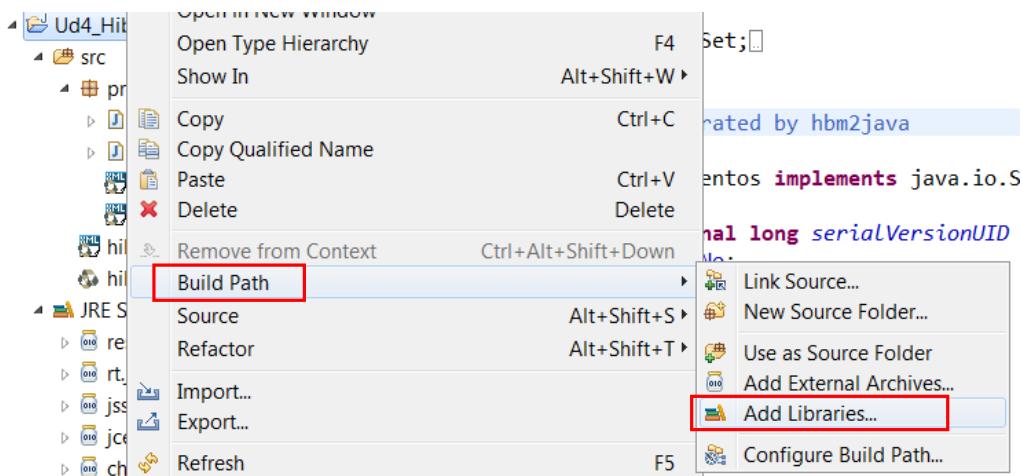
5. En la carpeta descomprimida, seleccionamos todos los ficheros de **\lib\jpa** y los copiamos a **C:\eclipse\Hibernate**.

6. Descargamos la última versión de *la librería slf4j* desde la URL: <http://www.slf4j.org/download.html> para descargar el fichero **slf4j-simple-1.7.2.zip**, lo descomprimimos y localizamos los ficheros **slf4j-simple-1.7.2.jar** y **slf4j-api-1.7.2.jar** para copiarlos a **C:\eclipse\Hibernate**

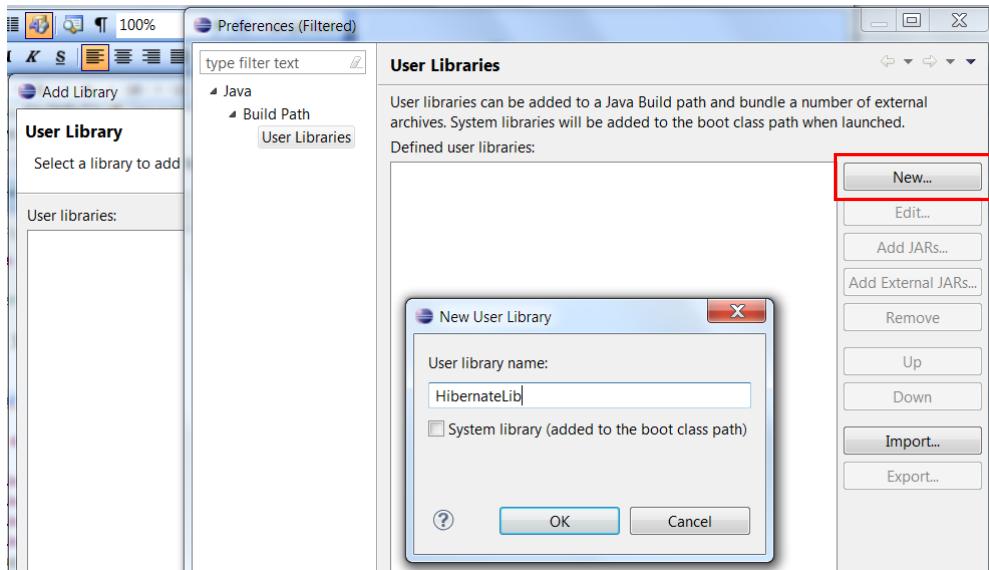
7. El contenido final de la carpeta **C:\eclipse\Hibernate** será:



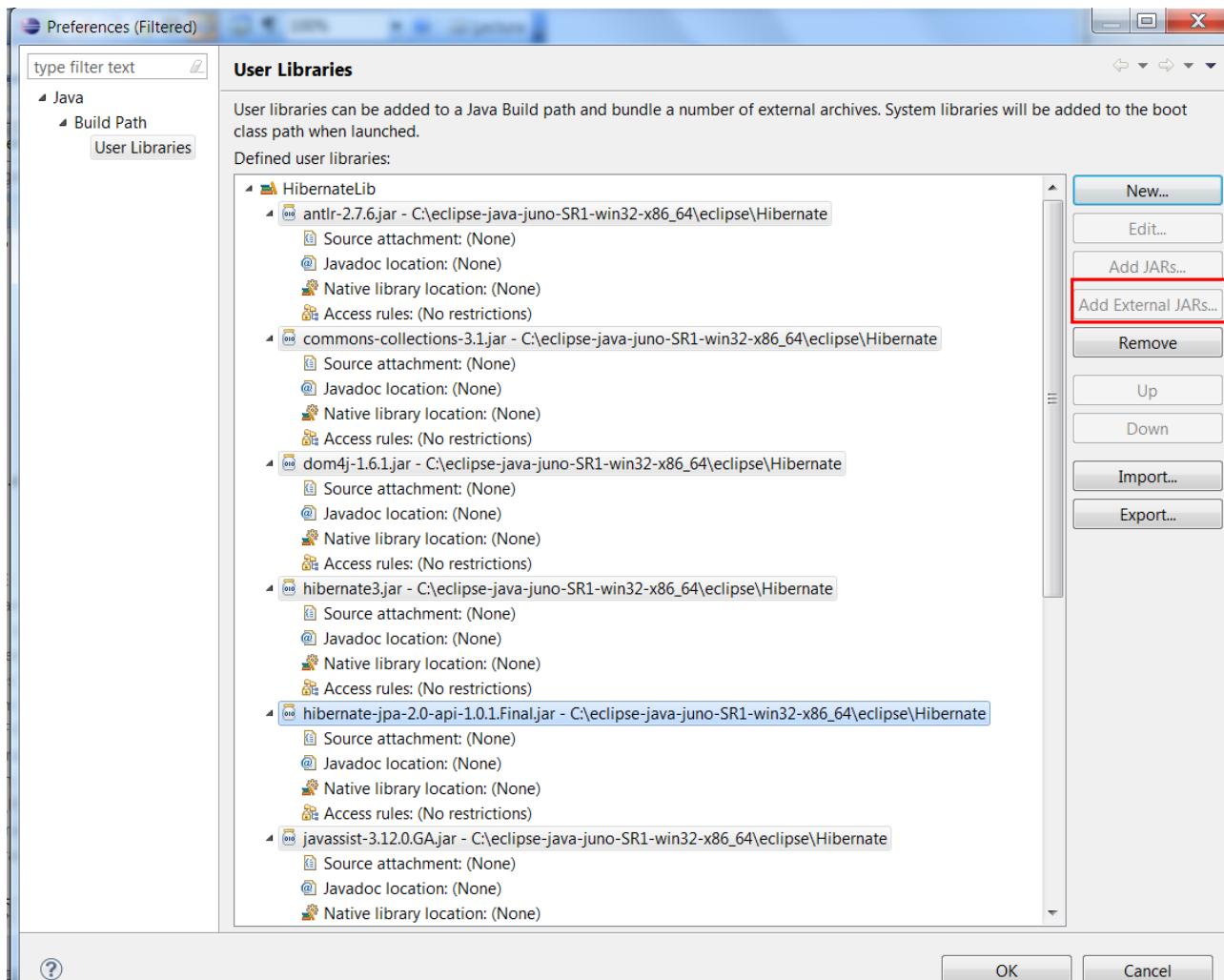
8. Desde Eclipse, hacemos click derecho sobre el proyecto y pulsamos Build Path→Add Libraries. Se visualiza una nueva ventana desde la que elegimos User Library y pulsamos Next



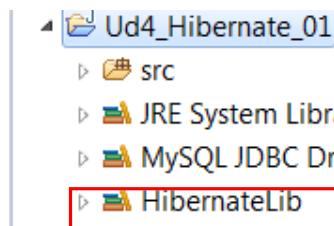
9. En la siguiente ventana pulsamos User Libraries, a continuación pulsamos New, nos pedirá el nombre de la librería que queramos agregar, escribimos por ejemplo **HibernateLib** y pulsamos el botón OK.



10. Seguidamente pulsamos el botón Add External JARs y seleccionamos todos los ficheros de nuestra carpeta **C:\eclipse\Hibernate**. Pulsamos OK y a continuación Finish.



En nuestro proyecto aparecerá la nueva librería:



Con esto, ya podemos crear nuestro primer programa Java, que permitirá comunicarnos con la base de datos **empresaz**.

11. Lo primero será **crear una instancia a la base de datos para poder trabajar con ella**, y que se utilizará a lo largo de toda la aplicación. Creamos para ello la clase de ayuda **HibernateUtil.java** tal y como hicimos en NetBeans. A este tipo de clases se las denomina también *Singleton*.

El **Singleton** es un patrón de diseño 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.

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 puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido y privado)

Esta clase accede a **SessionFactory** para obtener un objeto sesión. La clase llama a **configure()** y carga el fichero de configuración (**hibernate.cfg.xml**) y entonces construye la SessionFactory para obtener el objeto sesión.

Denominamos a la clase **HibernateUtil.java** y la incluimos en el **paquete primero** del proyecto

HibernateUtil.java

```
package primero;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory;
    static {
        try{
            sessionFactory=new Configuration().configure().buildSessionFactory();
        }catch(Throwable ex){
            System.err.println("Fallo inicio SessionFactory" + 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.

Creamos la clase principal **Main.java** en el paquete default Package o bien el paquete primero.

El programa de prueba consiste simplemente en consultar todos los departamentos de la BD y en insertar un nuevo objeto Departamentos en la base de datos. El código es el siguiente

(ponemos la clase Main.java en el **paquete primero**):

Consulta y listado de todos los departamentos de la base de datos

```
public static void ListadoDepartamentos() {
    //Obtiene sesión y la abre
    SessionFactory sfactory = HibernateUtil.getSessionFactory();
    Session session = sfactory.openSession();
    System.out.println("=====");
    System.out.println("Listado de Departamentos");

    Departamentos depart = new Departamentos();
    //consulta con HQL
    Query q = session.createQuery("from Departamentos");
    //obtiene un iterador para recorrer el resultado de la consulta
    Iterator<?> iter = q.iterator();
    //mientras haya resultados
    while (iter.hasNext()) {
        //extraer el siguiente objeto
        depart = (Departamentos) iter.next();
        System.out.println(depart.getDeptNo() + "*" + depart.getDnombre());
    }
    System.out.println("=====");
    //cierra sesión
    session.close();
}
```

Con la línea `SessionFactory sfactory= HibernateUtil.getSessionFactory()`, obtenemos la sesión creada por el Singleton o clase de ayuda. Esta instrucción se usará a lo largo de todas las clases en las que deseamos 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="MiConexion">

Por

<session-factory>

Cuando se cambia algún fichero de configuración hay que pulsar en el botón derecho en el nombre del proyecto y después en la opción Refresh.

Para insertar objeto Departamentos en la base de datos:

```
public static void insertaDepartamento() {
    Session session = null;
    Transaction tx = null;
    try {
        // obtiene una sesión a la BD
        SessionFactory sfactory = HibernateUtil.getSessionFactory();
        // abre la sesión
        session = sfactory.openSession();
        // comienza la transacción
        tx = session.beginTransaction();

        System.out.println("Inserta una fila en la tabla Departamentos");
        // inserta el departamento 66 en la tabla departamentos
        Departamentos dep = new Departamentos();
        dep.setDeptNo((byte) 66);
        dep.setDnombre("MARKETING");
        dep.setLoc("GUADALAJARA");
        session.save(dep); // almacena el objeto
        tx.commit();
    } catch (HibernateException e) {
        System.out.println("ERROR1: " + e.getCause() + ", " + e.getMessage());
        if (tx != null)
            tx.rollback();
    } finally {
        session.close();
    }
}
```

La ejecutamos y comprobamos que se ha insertado el departamento.

6. Estructura de los ficheros de mapeo

Las aplicaciones Hibernate hacen uso de ficheros de mapeo que contienen metadatos que definen los mapeos objeto/relacional para las clases Java. Un fichero de mapeo tiene el sufijo .hbm.xml.

Las aplicaciones Hibernate utilizan unos **ficheros de mapeo**, que:

- Contienen metadatos que definen los mapeos objeto/relacional para las clases Java.
- Dentro de cada fichero, se mapean a tablas de la base de datos las clases que se van a persistir y las propiedades se definen con mapeos de campo/columna y claves primarias.
- Están en formato XML y tienen la extensión **.hbm.xml**.

En el proyecto anterior se han creado los **ficheros de mapeo**:

- *Empleados.hbm.xml*, asociado a la tabla EMPLEADOS
- *Departamentos.hbm.xml*, asociado a la tabla DEPARTAMENTOS.

Estos ficheros se guardan en el mismo directorio que las **clases Java (POJOs) que representan a las tablas**

- *Empleados.java*, que representa un objeto Empleados
- *Departamentos.java*, que representa un objeto Departamentos

Todos los ficheros forman parte del mismo paquete

Ese paquete en nuestros ejemplos es: *primero* (Eclipse) o *empresaz.entity* (NetBeans).

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://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 10-dic-2012 12:13:55 by Hibernate Tools 3.4.0.CR1 -->
<hibernate-mapping>
  <class name="primero.Departamentos" table="departamentos" catalog="empresaz">
    <id name="deptNo" type="byte">
      <column name="dept_no" />
      <generator class="assigned" />
    </id>
    <property name="dnombre" type="string">
      <column name="dnombre" length="15" />
    </property>
    <property name="loc" type="string">
      <column name="loc" length="15" />
    </property>
    <set name="empleadoses" table="empleados" inverse="true">
      <key>
        <column name="dept_no" not-null="true" />
      </key>
      <one-to-many class="primero.Empleados" />
    </set>
  </class>
</hibernate-mapping>
```

Y la descripción de la **tabla DEPARTAMENTOS**:

TABLE departamentos (
dept_no TINYINT(2) NOT NULL PRIMARY KEY,
denombre VARCHAR(15),
loc VARCHAR(15)
);

Veamos el **significado del contenido del fichero XML**:

- <hi bernate-mappi ng>: todos los ficheros de mapeo comienzan y terminan con esta etiqueta.
- <cl ass>: 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:

```
<cl ass name="pri mero. Departamentos" tabl e="departamentos" catal og="empresaz">
```

- <i d>: 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 *autoincrement* indicaría que es un identificador *autogenerated* por la base de datos. Este atributo se correspondería con la columna *dept_no* (*dept_no TINYINT(2) NOT NULL PRIMATY KEY*) de la tabla DEPARTAMENTOS:

```
<i d name="deptNo" type="byte">
  <col umn name="dept_no" />
  <generator cl ass="assi gned" />
</i d>
```

- El resto de atributos se indican en la etiqueta *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 definiría así:

```
<property name="dnombre" type="string">
  <col umn name="dnombre" length="15" />
</property>
```

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

```
<property name="loc" type="string">
  <col umn name="loc" length="15" />
</property>
```

- Hay un nuevo atributo indicado mediante la etiqueta *set*, que permite relacionar al departamento con sus empleados. En el mapeo se define una lista de empleados para un departamento. La relación es **one-to-many**, asociación **uno-a-muchos** (*un departamento puede tener muchos empleados*). El atributo de nombre *empleadoses* es una lista de instancias de la clase *pri mero.Empleados*:

```
<set name="empl eadoses" tabl e="empl eados" i nverse="true">
  <key>
    <col umn name="dept_no" not-nul l="true" />
  </key>
  <one-to-many cl ass="pri mero.Empl eados" />
</set>
```

Los **tipos 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 Java a tipos SQL y viceversa. De nuevo *Hibernate tratará de determinar el tipo correcto de conversión y de mapeo por sí mismo*.

En el caso del fichero **Empleados.hbm.xml**, la estructura es:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 10-dic-2012 12:13:55 by Hibernate Tools 3.4.0.CR1 -->
<hibernate-mapping>
  <class name="primero.Empleados" table="empleados" catalog="empresaz">
    <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 name="comision" type="java.lang.Float">
      <column name="comision" precision="6" />
    </property>
  </class>
</hibernate-mapping>
```

Y la descripción de la tabla Empleados:

```
TABLE empleados (
  emp_no      SMALLINT(4) UNSIGNED NOT NULL PRIMARY KEY,
  apellido    ARCHAR(10),
  oficio      VARCHAR(10),
  dir         SMALLINT,
  fecha_alt   DATE,
  salario     FLOAT(6,2),
  comision    FLOAT(6,2),
  dept_no     TINYINT(2) NOT NULL,
  FOREIGN KEY (dept_no) REFERENCES departamentos(dept_no)
);
```

Aquí se observa la relación **many-to-one**, es una relación unidireccional **muchos-a-uno** (*muchos empleados pertenecen a un departamento*). 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.

7. Clases persistentes

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

```
<hibernate-mapping>
    <class name="primero.Departamentos" table="departamentos" catalog="empresaz">
        <class name="primero.Empleados" table="empleados" catalog="empresaz">
```

En nuestro proyecto de ejemplo se han generado las clases **Empleados.java** y **Departamentos.java**. A estas clases se las llama, **clases persistentes**, 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, y por tanto se pueden crear objetos Empleados y Departamentos a partir de ellas. Tienen unos atributos y métodos `getter` y `setter` 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 atributos de los objetos privados se crean métodos para retornar un valor de un atributo, métodos `getter`, o para cargar un valor a un atributo, métodos `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">
    <column name="dept_no" />
    <generator class="assigned" />
</id>
```

- Al igual que con el elemento `id`, el atributo `name` del elemento `property` le dice a Hibernate qué métodos `getter` y `setter` utilizar.

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

Así que en este caso, Hibernate buscará los métodos `getDnombre()`, `setDnombre()`, `getLoc()` y `setLoc()`.

Convenciones de nombrado estándares de JavaBean para los métodos de propiedades getter y setter:

- Si la propiedad no es de tipo booleana, el nombre del método que lee el valor debe estar precedido por `get`. Por ejemplo: `getDnombre()` es un nombre válido.
- Si la propiedad es un booleano, el prefijo es `get` o `is`. Por ejemplo, `getCasado()` o `isCasado()`, son nombres válidos para una propiedad booleana.
- El método para almacenar el valor debe tener el prefijo `set`. Por ejemplo, `setDnombre()` es un nombre válido para la propiedad *dnombre*.
- Para completar el nombre de un método `getter` o `setter`, se pone la primera letra que los une en mayúsculas.
- Los métodos marcados como `setter` deben ser declarados como *públicos*, devolver un *tipo void* y recibir un *argumento del tipo de propiedad* al que van a dar valor.
- Los métodos de tipo `getter` deben ser declarados como *públicos*, *no aceptan argumentos* y *devuelven un valor del mismo tipo que el que recibe el método setter*.

Departamentos.java

```
package primero;
import java.util.HashSet;
/**
 * Departamentos generated by hbm2java
 */
public class Departamentos implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private byte deptNo;
    private String dnombre;
    private String loc;
    @SuppressWarnings("rawtypes")
    private Set empleadoses = new HashSet(0);

    public Departamentos() {
    }
    public Departamentos(byte deptNo) {
        this.deptNo = deptNo;
    }
    public Departamentos(byte deptNo, String dnombre, String loc,
                         Set 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 getEmpleadoses() {
        return this.empleadoses;
    }
    public void setEmpleadoses(Set empleadoses) {
        this.empleadoses = empleadoses;
    }
}
```

8. Sesiones y Objetos Hibernate

Para poder utilizar los mecanismos de **persistencia de Hibernate** se debe:

- inicializar el entorno Hibernate
- 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();
//obtiene un objeto session
Session session = sessionFactory.openSession();
```

La llamada **Configuration().configure()** carga el fichero de configuración **hibernate.cfg.xml** e inicializa el entorno de Hibernate.

Una vez inicializada la configuración se crea el ejemplar **SessionFactory**, que normalmente solo se crea una vez y se utiliza para crear todas las sesiones relacionadas con un contexto dado.

Esto es lo que se hizo en el proyecto inicial al crear la clase **HibernateUtil.java**, para crear una vez el ejemplar de **SessionFactory**:

```
//inicializa el entorno Hibernate y crea un ejemplar de SessionFactory
SessionFactory session = HibernateUtil.getSessionFactory();
//obtiene un objeto session
Session session = session.openSession();
```

8.1 Transacciones y Sesiones

Un objeto **Session** de Hibernate, una sesión, representa una única unidad de trabajo para un almacén de datos dado y lo abre un ejemplar de **SessionFactory**. Se deben cerrar las sesiones cuando se haya completado todo el trabajo de una transacción.

La sesión nos permite representar el gestor de persistencia, ya que dispone de una API básica que nos permite cargar y guardar objetos.

La sesión está formada internamente por una cola de sentencias SQL que son necesarias ejecutar para poder sincronizar el estado de la sesión con la base de datos. Asimismo, la sesión contiene una lista de objetos persistentes. Una sesión corresponde con el primer nivel de caché.

Una **transacción** es un conjunto de órdenes que se ejecutan formando un unidad de trabajo, en forma indivisible oatómica.

Para realizar con éxito la gestión de transacciones, éstas se van a basar en el uso del objeto **Session**.

El siguiente código ilustra una **sesión de persistencia** de Hibernate:

```
SessionFactory sfactory = HibernateUtil.getSessionFactory();
Session session = sfactory.openSession(); //comienza la sesión
Transaction tx;
try {
    tx = session.beginTransaction(); //comienza la transacción
    //código de persistencia (/get/load/delete/save/update)
    tx.commit();
}
catch (HibernateException e) {
    if (tx!=null)
        tx.rollback();
}
finally {
    session.close();
}
```

- El método **commit()** confirma una transacción
- El método **rollback()** deshace una transacción.

Cuando se crea el objeto **Session**, se le asigna la conexión de la base de datos que va a utilizar. Una vez obtenido el objeto **Session**, se crea una nueva unidad de trabajo o **Transaction** utilizando el método `beginTransaction()`.

- Dentro del contexto de la transacción creada, se pueden invocar los métodos de gestión de persistencia proporcionados por el objeto **Session**, para recuperar, añadir, eliminar o modificar el estado de instancias de clases persistentes.
- También se pueden realizar consultas.
- Si las operaciones de persistencia no han producido ninguna excepción, se invoca el método `commit()` de la unidad de trabajo para confirmar los cambios realizados. En caso contrario, se realiza un `rollback()` para deshacer los cambios producidos.
- Sobre un mismo objeto **Session** pueden crearse varias unidades de trabajo. Finalmente se cierra el objeto **Session** invocando su método `close()`.

8.2 Estados de un objeto Hibernate

Hibernate define y soporta los siguientes estados de objeto:

- **Transitorio (Transient)**: El objeto está recién creado y no ha sido enlazado con el gestor de persistencia.
- **Persistente**: El objeto está enlazado con una sesión. Todos los cambios que se realicen serán persistentes.
- **Separado (Detached)**: El objeto es persistente y sigue en memoria después de que termine la sesión. En este caso existe en Java y en la base de datos.

Vemos con más detalle estos estados:

- **Transitorio**. 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.
 - Utilizaremos una **Session** de Hibernate para hacer un objeto persistente y dejar que Hibernate se ocupe de las declaraciones SQL que necesitan ejecutarse para esta transición.
 - Los objetos recién instanciados de una clase persistente Hibernate los considera como transitorios.
 - Podemos hacer que un objeto transitorio sea persistente asociándolo a una sesión:

```
Departamentos dep = new Departamentos();
dep.setDeptNo((byte) 66);
dep.setDnombre("MARKETING");
dep.setLoc("GUADALAJARA");
session.save(dep); // hace que la instancia u objeto sea persistente
```

- **Persistente**. Un objeto persistente tiene una representación en la base de datos y un valor identificador.
 - Puede haber sido guardado o cargado en memoria, 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.

Resumiendo, 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**. Una instancia separada es un objeto que se ha hecho persistente, pero su **Session** 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 reunida a una nueva **Session** más tarde, haciéndola persistente de nuevo (con todas las modificaciones).

8.3 Carga de objetos persistentes

Existen dos métodos de la clase **Session** que se encargan de recuperar un objeto persistente por identificador: **load()** y **get()**. Ambos toman un objeto y cargan el estado dentro de un objeto recién instanciado de esa clase, en estado persistente. La diferencia entre ellos radica en cómo indican que un objeto no se encuentra en la base de datos y que **load()** puede devolver un **objeto proxy**.

- **load()** lanza una excepción *ObjectNotFoundException* si la fila no existe.

```
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());
```

- **get()** devuelve *null* si no existe una fila correspondiente al objeto.

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

- La recuperación de un objeto por su identificador no es tan flexible como el uso de consultas arbitrarias.

EJEMPLO. El siguiente ejemplo comprueba si el departamento 11 existe, si existe visualiza sus datos, si no existe visualiza un mensaje:

```
package primero;

import org.hibernate.SessionFactory;
import org.hibernate.classic.Session;

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

        System.out.println("=====");
        System.out.println("DATOS DEL DEPARTAMENTO 11");

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

El resultado de ejecución será el siguiente: (no existe el departamento 11)

```
=====
DATOS DEL DEPARTAMENTO 11
El departamento no existe
=====
```

EJEMPLO. 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:

```
package primero;

import java.util.Iterator;
import java.util.Set;
import org.hibernate.SessionFactory;
import org.hibernate.Session;

public class ListadoDep {
    public static void main(String[] args){
        SessionFactory sesion = SessionFactoryUtil.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.getNombre());
        System.out.println("Localidad: "+dep.getLoc());

        System.out.println("=====");
        System.out.println("Empleados del departamento 10");

        Set<Empleados> listaemple = dep.getEmpleados(); //obtiene empleados
        Iterator<Empleados> it = listaemple.iterator();
        System.out.println("Número de empleados: "+listaemple.size() );
        while(it.hasNext()){
            Empleados emple = new Empleados();
            emple=it.next();
            System.out.println(emple.getApellido() + " * " + emple.getSalario());
        }
        System.out.println("=====");
        session.close();
    }
}
```

El resultado de ejecución es similar al siguiente:

```
=====
DATOS DEL DEPARTAMENTO 10
Nombre Dep: CONTABILIDAD
Localidad: SEVILLA
=====
Empleados del departamento 10
Número de empleados: 7
HERNANDEZ * 4195.51
PEPE * 1500.0
CASTILLO * 2023.64
HERMIDA * 642.97
ALONSO * 2101.22
CASTRO * 2644.16
AMARO * 2721.74
=====
```

Referencia API de Hibernate:
<http://docs.jboss.org/hibernate/core/3.5/javadoc/>

8.4 Almacenamiento, modificación y borrado de objetos persistentes

Los siguientes métodos de la clase **Session** permiten guardar, modificar y eliminar objetos de la base de datos.

- **Método `save()`**. Permite almacenar objetos en contexto de persistencia.

Para almacenar objetos persistentes, los pasos generales son los siguientes:

1. Se instancia un objeto nuevo (estado transitorio).
2. Se obtiene una sesión y se comienza la transacción, inicializando el contexto de persistencia.
3. Una vez obtenida la sesión, se llama al método **`save()`**, el cual introduce el objeto en el contexto de persistencia. Este método devuelve el identificador del objeto persistido.
4. Para que los **cambios sean sincronizados en las bases de datos**, es necesario realizar el **`commit()`** de la transacción. Dentro del objeto sesión se llama al método **`flush()`**. Es posible llamarlo explícitamente. En este momento, se obtiene la conexión JDBC a la bases de datos para poder ejecutar la oportuna sentencia.
5. Finalmente, la sesión se cierra, con el objeto de liberar el contexto de persistencia, y por tanto, devolver la referencia del objeto creado al estado disociado.

El siguiente ejemplo crea un nuevo objeto Departamentos y se lo entrega a Hibernate, en ese momento 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 en contexto de persistencia
tx.commit(); //confirma transacción (sincronización con base de datos)
session.close();
```

Si ya existe el objeto en la base de datos se lanza una excepción de violación de integridad referencial.

- **Método `delete()`**. Permite eliminar objetos de la base de datos. Para realizar el borrado, primero debe haber sido cargado éste con el método `get()` o `load()`, a continuación podemos borrarlo con `delete()`.

El siguiente ejemplo borra el empleado cuyo número de empleado (`empNo`) es 8000:

```
Empl eados em = new Empl eados();
em = (Empl eados) session.load(Empl eados.class, (short) 8000);
session.delete(em); // el imina el objeto
```

Si no existe el objeto, se lanza una excepción `ObjectNotFoundException` que habrá que gestionar en un `try-catch`

- **Método `update()`**. Permite modificar objetos de la base de datos. Al igual que en el borrado, primero hay que cargar el objeto con `load()` o `get()`, a continuación realizar las modificaciones con los métodos `setter` y por último utilizar el método `update()`.

```
Empl eados em = new Empl eados();
em = (Empl eados) session.load(Empl eados.class, (short) 8000);
System.out.println("Salario antiguo: " + em.getSalario());
System.out.println("Comisión antigua: " + em.getComision());
em.setSalario((float) 3500);
em.setComision((float) 150);
session.update(em); // modifica el objeto

System.out.println("Salario nuevo: " + em.getSalario());
System.out.println("Comisión nueva: " + em.getComision());
```

Si no existe el objeto, se lanza una excepción `ObjectNotFoundException` que habrá que gestionar en un `try-catch`

9. Consultas

Hibernate soporta un lenguaje de consultas 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 son representadas con una instancia de la **interface Query**.
 - La interface **Query** ofrece métodos para ligar parámetros por nombre o por posición (al estilo de JDBC con marcadores ?), ejecutar y manejar el resultado de la consulta.
 - La interface Query obtiene la consulta utilizando un objeto Session.
- Para obtener una consulta usaremos el método **Session.createQuery("cadenaConsulta")** de la siguiente forma:

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

- Para ejecutar y recuperar los datos de la consulta usaremos el método **iterate()**:

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

- El método **iterate()** ejecuta la consulta y devuelve un objeto *Iterator* Java para recorrer los resultados de la consulta. En este caso Hibernate ejecuta la consulta obteniendo solo los **ids** de las entidades y en cada llamada el método *Iterator.next()* ejecuta la consulta propia para obtener la entidad completa. Esto implica una importante cantidad de accesos a la base de datos, pero la ventaja es que no 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 JDBC subyacente 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()*:

```
q.setFetchSize(5);
```

Consultar la API de Hibernate. Interface Query

<http://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/Query.html>

EJEMPLO. Consulta de todas las filas de la tabla Departamentos:

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

```
Departamentos depar = new Departamentos();
Query q = session.createQuery("from Departamentos");
q.setFetchSize(5);
Iterator iter = q.iterator();
while(iter.hasNext()){
    //extraer objeto
    depar=(Departamentos) iter.next();
    System.out.println(depar.getDeptNo() + "*" + depar.getDnombre());
}
```

- El método **uniqueResult()** ofrece un atajo si sabemos que la consulta devolverá un único objeto, no siendo necesario obtener un objeto *Query* para iterar sobre él.

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 de nombre Contabilidad.

En estos ejemplos se utiliza un parámetro por posición (?) en el criterio where de la consulta HQL. Mediante métodos setxxxx(posición, valor) se asigna el valor al parámetro.

En este caso **setInteger(0, 10)** asigna el valor 10 al parámetro de posición 0.

```
//visualiza los datos del departamento 10
Departamentos depar = (Departamentos) session.createQuery("from Departamentos"
+ " where deptNo=?").setInteger(0, 10).uniqueResult();
```

```
System.out.println(depar.getDeptNo() + "*" + depar.getLoc() + "*" + depar.getDnombre());
```

En este caso `setString(0, "Contabilidad")` asigna el valor "Contabilidad" al parámetro de posición 0.

```
//visualiza los datos del departamento de nombre Contabilidad
Departamentos depar = (Departamentos) session.createQuery("from Departamentos"
    + " where dnombre=?").setString(0, "Contabilidad").uniqueResult();
System.out.println(depar.getDeptNo() + "*" + depar.getLoc() + "*" + depar.getDnombre());
```

Producido en ambos casos el siguiente resultado:

```
=====
10*SEVILLA*CONTABILIDAD
=====
```

El siguiente ejemplo realiza una consulta para visualizar el apellido y el salario de los empleados del departamento 10:

```
System.out.println("=====");
System.out.println("Empleados Departamento 10");
Empleados emple = new Empleados();
//consulta con parámetros
Query q = session.createQuery("from Empleados where departamentos.deptNo=?");
//asignación del valor al parámetro
q.setInteger(0, 10);
//filas recuperadas en cada acceso a la BD
q.setFetchSize(5);
//iterador para ejecutar y recorrer la consulta
Iterator<?> iter = q.iterator();
//mientras haya resultados
while(iter.hasNext()){
    emple=(Empleados) iter.next(); //extraer objeto
    System.out.println(emple.getApellido() + "*" + emple.getSalario());
}
System.out.println("=====");
```

Producido una salida similar a la siguiente:

```
=====
Empleados Departamento 10
AMARO*2721.74
ALONSO*2101.22
CASTRO*2644.16
CASTILLO*2023.64
HERNANDEZ*4195.51
HERMIDA*642.97
=====
```

9.1 Parámetros en las consultas

Con `createQuery()` se pueden enlazar valores a los parámetros con nombre, que son identificados de la forma:

- `:nombre` en la cadena de la consulta, o a los
- `parámetros ?` de estilo JDBC, como has visto en ejemplos anteriores.

Hibernate numera los parámetros ? desde 0 en adelante, 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 el que aparecen en la cadena de consulta
- Pueden aparecer múltiples veces en la misma petición
- Son auto-documentados.

Algunos métodos que se pueden usar para asignar valores a los parámetros son:

MÉTODO	DESCRIPCIÓN
setCharacter(int posición, char valor)	Asigna valor a un parámetro de tipo CHAR
setCharacter(String nombre, char valor)	
setdate(int posición, Date fecha)	Asigna valor a un parámetro de tipo DATE
setDate(String nombre, Date fecha)	
setDouble(int posición, double valor)	Asigna valor a un parámetro de tipo decimal (en MySQL tipo FLOAT)
setDouble(String nombre, double valor)	
setInteger(int posición, int valor)	Asigna valor a un parámetro de tipo entero
setInteger(String nombre, int valor)	
setString(int posición, String valor)	Asigna valor a un parámetro de tipo VARCHAR
setString(String nombre, String valor)	
setParameterList(String nombre, Collection valores)	Asigna una lista de valores a un parámetro
posición => indica la posición del parámetro dentro de la consulta, empieza en 0	
nombre => indica el nombre (:nombre) del parámetro dentro de la consulta	

Vemos algunos **EJEMPLOS**:

- Empleados cuyo número de departamento sea 10 y el oficio AUXILIAR

```
//utilizando parámetros por nombre
Query q = session.createQuery("from Empl eados where departamentos.deptNo= :ndep "
                           + "and oficio= :ofi ");
q.setInteger("ndep", 10);
q.setString("ofi", "AUXILIAR");
```

```
//utilizando parámetros por posición
Query q = session.createQuery("from Empl eados where departamentos.deptNo= ? "
                           + "and oficio= ?");
q.setInteger(0, 10);
q.setString(1, "AUXILIAR");
```

Empleados cuyo número de departamento sea 10 o 20

```
//lista de parámetros nombrados
List<Byte> numeros = new ArrayList <Byte> ();
numeros.add((byte)10);
numeros.add((byte)20);
q = session.createQuery("from Empl eados where " +
                      "departamentos.deptNo in (:listadep)");
q.setParameterList("listadep", numeros);
```

- Empleados cuya fecha de alta es anterior a 2010-12-31

```
//Parámetros de tipo date
SimpleDateFormat formatoDelTexto = new SimpleDateFormat("yyyy-MM-dd");
String strFecha = "2010-12-31";
Date fecha = null;
try{
    fecha=formatoDelTexto.parse(strFecha);
} catch(ParseException ex){ ex.printStackTrace(); }

q = session.createQuery("from Empl eados where fechaAlt < ?)");
q.setDate(0, fecha);
```

9.2 Consultas sobre clases no asociadas

Los ejemplos anteriores se han realizado sobre las clases Empleados y Departamentos. Estas clases son clases asociadas, ya que provienen de un mapeo entre tablas relacionadas mediante una clave foránea y como resultado del mapeo se obtuvo en la clase Empleados el atributo `departamentos` que es una referencia a la clase Departamentos:

```
public class Empleados implements Serializable {
    private short empNo;
    private Departamentos departamentos;
    private String apellido;
    private String oficio;
    private Short dir;
    private Date fechaAlt;
    private Float salario;
    private Float comision;
    . . .
}
```

```
public class Departamentos implements Serializable {
    private byte deptNo;
    private String dnombre;
    private String loc;
    private Set empleadoses = new HashSet(0);
    . . .
}
```

```
CREATE TABLE empleados (
    emp_no      SMALLINT(4) NOT NULL PRIMARY KEY,
    apellido    CHAR(10),
    oficio      VARCHAR(10),
    dir         SMALLINT,
    fecha_alt   DATE,
    salario     FLOAT(6,2),
    comision    FLOAT(6,2),
    dept_no     TINYINT(2) NOT NULL,
    FOREIGN KEY (dept_no) REFERENCES
    departamentos(dept_no)
);
```

```
CREATE TABLE departamentos (
    dept_no     TINYINT(2) NOT NULL PRIMARY KEY,
    denombre    VARCHAR(15),
    loc         VARCHAR(15)
);
```

En este caso, la consulta para obtener los datos de los empleados (apellido, salario) y de sus departamentos (nombre del departamento y localidad) sería de la siguiente forma:

```
System.out.println("Empleados y sus departamentos");
Empleados emple = new Empleados();

//consulta sobre clases asociadas
Query q = session.createQuery("from Empleados order by apellido");

// iterador para ejecutar y recorrer la consulta
Iterator<?> iter = q.iterator();

while (iter.hasNext()) {
    emple=(Empleados) iter.next(); //extraer objeto

    System.out.println(emple.getApellido() + "*" + emple.getSalario() + "*"
        + emple.getDepartamentos().getDnombre() + "*" + emple.getDepartamentos().getLoc());
}
```

Observa que los objetos que devuelve la consulta son objetos de la clase Empleados.

Supongamos el caso en que **las clases no están asociadas**, esto es, las clases resultantes del mapeo no contienen atributos que hagan referencia a otras clases (provienen de un mapeo de tablas que no están relacionadas por clave foránea):

```
public class Empleados implements
java.io.Serializable {
    private short empNo;
    private byte deptNo;
    private String apellido;
    private String oficio;
    private Short dir;
    private Date fechaAlt;
    private Float salario;
    private Float comision;
```

```
public class Departamentos implements
java.io.Serializable {
    private byte deptNo;
    private String nombre;
    private String loc;
```

```
CREATE TABLE empleados (
emp_no      SMALLINT(4) NOT NULL PRIMARY KEY,
apellido    ARCHAR(10),
oficio      VARCHAR(10),
dir         SMALLINT,
fecha_alt   DATE,
salario     FLOAT(6,2),
comision    FLOAT(6,2),
dept_no     TINYINT(2) NOT NULL REFERENCES
departamentos(dept_no)
);
```

```
CREATE TABLE departamentos (
dept_no     TINYINT(2) NOT NULL PRIMARY KEY,
denombre    VARCHAR(15),
loc         VARCHAR(15)
);
```

Para recuperar los mismos datos que antes mediante una consulta, ahora:

- Deben intervenir ambas clases.
- Además no tenemos asociada ninguna clase a los atributos que devuelve esa consulta. En estos casos podemos utilizar la **clase Object**.
 - Utilizamos la clase Object de la siguiente forma:

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 una array de objetos, donde el primer elemento del array pertenece a la clase Empleados y el segundo a la Departamentos:

```
// consulta sobre varias clases
Query q= session.createQuery("from Empleados e, Departamentos d " +
                            " where e.deptNo=d.deptNo order by e.apellido");

// iterador para ejecutar y recorrer la consulta
Iterator<?> iter = q.iterate();

while(iter.hasNext()){
    //extraer array de objetos
    Object[] par = (Object[]) iter.next();
    //asigna cada parte al correspondiente tipo de objeto
    Empleados em = (Empleados) par[0];
    Departamentos de = (Departamentos) par[1];

    System.out.println(em.getApellido()+"*"+em.getSalario()+"*"+
                       de.getNombre()+"*"+de.getLoc());
}
```

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, o de agregación, como AVG(), SUM(), COUNT(), MAX(), MIN(), se puede recoger como un único valor utilizando el método ***uniqueResult()***.

EJEMPLOS:

- Este ejemplo muestra el total de empleados:

```
// Total de empleados
Query q = session.createQuery("select count(*) from Empl eados");
Long total = (Long) q.uniqueResult();
System.out.println("Total empleados: " + total);
```

- Este ejemplo muestra el salario medio de los empleados:

```
//salario medio de los empleados
Query q= session.createQuery("select avg(salario) from Empl eados");
Double salarioMedio = (Double) q.uniqueResult();
System.out.println("Salario medio: "+ salarioMedio);
```

- Este ejemplo muestra el total de empleados del departamento 10

Sin parámetros

```
//Total de empleados del departamento 10
Query q= session.createQuery("select count(*) from Empl eados where " +
                           "departamentos.deptNo=10");
Long total = (Long) q.uniqueResult();
System.out.println("Total empleados Depar 10: "+ total);
```

Usando parámetros por posición

```
//Total de empleados del departamento 10
Query q= session.createQuery("select count(*) from Empl eados where " +
                           "departamentos.deptNo=?");
Long total = (Long) q.setInteger(0, 10).uniqueResult();
System.out.println("Total empleados Depar 10: "+ total);
```

Si en la consulta intervienen varias funciones de agregado ¿cómo recuperamos los datos?

```
//Salario máximo, mínimo y medio de los empleados
Query q= session.createQuery("select max(salario), min(salario), avg(salario)" +
                           " from Empl eados");
```

Cuando en la consulta intervienen varias funciones de grupo o agregado y además devuelve varias filas podemos utilizar objetos devueltos por las consultas, tal y como veremos en el siguiente apartado.

9.4 Objetos devueltos por las consultas

Supongamos que a partir de las tablas EMPLEADOS y DEPARTAMENTOS queremos 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, podemos crear una y utilizarla sin necesidad de mapearla.

Cada fila devolverá un objeto de esa clase. Por ejemplo, creamos la **clase Totales** en el paquete **primero (o Ud4_Hibernate_01)** con 4 atributos: numero, cuenta, media, nombre (para guardar los datos del número de departamento, número de empleados, media de salario y nombre del departamento) y los getter y setter asociados.

Totales.java

```
public class Totales{
    private Object cuenta;
    private Object numero;
    private Double media;
    private String nombre;

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

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

    public Object getNumero() {return this.numero;}
    public void setNumero (final Object 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 **Totales.java**, procedemos de la siguiente forma:

Consulta donde para cada departamento se muestra: número de departamento, nombre, total de empleados y media de salario

```
System.out.println("Departamento, nombre, total empleados y salario medio");

Query q = session.createQuery("select new primero.Totales("
    + "departamentos.deptNo, count(empNo), "
    + "avg(salario), departamentos.dnombre)"
    + "from Empleados "
    + "group by departamentos.deptNo");

Iterator<?> iter = q.iterator();
while (iter.hasNext()) {
    Totales tot = (Totales) iter.next();
    System.out.println("Número Dep: " + tot.getNumero() +
                       " Nombre: " + tot.getNombre() +
                       " Nº emple: " + tot.getCuenta() +
                       " Salario medio: " + tot.getMedia());
}
```

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 vió anteriormente en el apartado 10.2).

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 indexando por posición.

Consulta donde para cada departamento se muestra: número de departamento, nombre, total de empleados y media de salario

```
System.out.println("Departamento, nombre, total empleados y salario medio");

Query q = session.createQuery("select departamentos.deptNo, count(empNo), " +
    " avg(salario), departamentos.dnombre" +
    " from Empleados" +
    " group by departamentos.deptNo");

Iterator<?> iter = q.iterate(); //Ejecución de consulta;
while (iter.hasNext()) {
    Object[] filaActual = (Object[]) iter.next(); //Acceso a una fila
    System.out.println("Dep: "+filaActual[0] + "* Nombre: "+filaActual[3] +
        " * Empleados: "+filaActual[1] +
        " * Salario Medio: "+filaActual[2]);
}
```

La misma consulta pero en el caso de que las clases Empleados y Departamentos no estén asociadas o relacionadas (provienen del mapeo de tablas sin claves ajena) sería:

```
System.out.println("Departamento, nombre, total empleados y salario medio");

Query q = session.createQuery("select de.deptNo, count(em.empNo), " +
    " avg(em.salario), de.dnombre as nombre " +
    "from Departamentos as de, Empleados as em" +
    " where de.deptNo=em.deptNo" +
    " group by de.deptNo");

Iterator<?> iter = q.iterate(); //Ejecución consulta;
while (iter.hasNext()) {
    Object[] filaActual = iter.next(); //Acceso a una fila
    System.out.println("Dep: "+filaActual[0] + "* Nombre: "+filaActual[3] +
        " * Empleados: "+filaActual[1] +
        " * Salario Medio: "+filaActual[2]);
}
```

Obteniendo en los tres casos un resultado similar al siguiente:

```
=====
Departamento, nombre, total empleados y salario medio
Número Dep: 10 Nombre:CONTABILIDAD Nº emple: 6 Salario medio: 2388.206604
Número Dep: 20 Nombre:INVESTIGACIÓN Nº emple: 3 Salario medio: 1620.0
Número Dep: 30 Nombre:VENTAS Nº emple: 3 Salario medio: 1933.333333
Número Dep: 40 Nombre:PRODUCCIÓN Nº emple: 2 Salario medio: 2975.0
=====
```

10. Insert, Update y Delete

Con el lenguaje HQL también podemos realizar operaciones INSERT, UPDATE y DELETE.

La sintaxis para las operaciones **UPDATE** y **DELETE** es la siguiente:

```
UPDATE [FROM] NombreEntidad set propiedad= nuevo_valor [WHERE condicion]

DELETE [FROM] NombreEntidad [WHERE condicion]
```

Donde:

- FROM y WHERE son opcionales
- 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.

- Para ejecutar un UPDATE o DELETE en HQL, utilizaremos el método **Query.executeUpdate()**.
- El valor **int** retornado por este método indica el número de entidades afectadas por la operación.
- Se debe realizar el **commit** para confirmar la transacción.

Vemos algunos **EJEMPLOS**:

- Modificar el salario de un empleado (el empleado CASTRO)

```
Sessi on sessi on = sfactory.openSessi on();
Transacti on tx = sessi on. begi nTransacti on();

String hql Modi f="update Empl eados e set e. sal ari o= :nuevoSal where e. apel l i do = :ape";
Query q= sessi on. createQuery(hql Modi f);
q.setDoubl e("nuevoSal ", 2500);
q.setString("ape", "CASTRO");
int filasModi f =q. executeUpdate();

System. out. println("Filas modifi cadas: " + filasModi f);
tx. commi t();
sessi on. cl ose();
```

- Eliminar los empleados de un departamento (el departamento 20)

```
Session sessi on = sfactory.openSessi on();
Transacti on tx = sessi on. begi nTransacti on();

String hql Del = "del ete Empl eados e where e. departamentos. deptNo = ?";
int filasDel = session. createQuery(hql Del )
.setInteger(0, 20)
.executeUpdate();
System.out.println("Filas elimi nadas: "+ filasDel); //número de filas afectadas

tx. commi t();
sessi on. cl ose();
```

Consulta para más información

<http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/batch.html#batch-direct>

Sintaxis para la operación **INSERT**

INSERT INTO NombreEntidad (lista_propiedades) sentencia_select

Donde:

- Solo se soporta la forma `INSERT INTO --- SELECT ...` no la forma `INSERT INTO ... VALUES`. Es decir, solo se pueden insertar datos procedentes de otra tabla (debe estar mapeada en nuestro proyecto).
- 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 propiedad `id` hay dos opciones: se puede especificar en la `lista_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 `AUTOINCREMENT PRIMARY KEY` en MySQL, la clave primaria se crea de forma automática sin necesidad de dar valor).

```
//insertamos los departamentos de la tabla NUEVOSDEP
//la tabla tiene que estar mapeada a nuestro proyecto
Session session = sfactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Departamentos (deptNo, dnombre, loc)" +
    " select n.deptNo, n.dnombre, n.loc from Nuevosdep n";
int createEntities = session.createQuery(hqlInsert).executeUpdate();
System.out.println("Filas insertadas: "+createEntities); //número de filas afectadas

tx.commit();
session.close();
```

Para mapear la tabla `nuevosdep` al proyecto puedes proceder de la siguiente forma:

- Genera de nuevo el fichero de configuración y el fichero de ingeniería inversa incluyendo además de las tablas anteriores la tabla `nuevosdep`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//
<hibernate-reverse-engineering>
<schema-selection match-catalog="empresaz"/>
<table-filter match-name="nuevosdep"/>
<table-filter match-name="empleados"/>
<table-filter match-name="departamentos"/>
</hibernate-reverse-engineering>
```

- Genera de nuevo los POJOS y los ficheros de mapeo, ubicándolos en el mismo paquete que antes.

	<pre><hibernate-mapping> <class name="empresaz.entity.Nuevosdep" table="nuevosdep" catalog="empresaz"> <id name="deptNo" type="byte"> <column name="dept_no" /> <generator class="assigned" /> </id> <property name="dnombre" type="string"> <column name="dnombre" length="15" /> </property> <property name="loc" type="string"> <column name="loc" length="15" /> </property> </class> </hibernate-mapping></pre>
--	--

Consulta para más información

<http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/batch.html#batch-direct>

11. Resumen del lenguaje HQL

Las consultas HQL no son sensibles a mayúsculas, a excepción de los nombres de las clases y propiedades Java.

- La cláusula mas 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.
- Para obtener determinadas propiedades (columnas) en una consulta utilizamos la cláusula SELECT. No se puede utilizar el *.

```
from Empl eados where departamentos.deptNo=10 order by apellido
```

```
select apellido, salario from Empl eados
where departamentos.deptNo=10 order by 2 desc
```

- Se pueden asignar alias a las clases usando la cláusula AS o sin ella:

```
from Empl eados as em o bi en from Empl eados em
```

- Las consultas pueden retornar múltiples objetos y/o propiedades, como un array de tipo `Object[]`, una lista, o una clase. 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 ..)`
- Se pueden utilizar alias para nombrar los atributos y expresiones.

```
select avg(salario) as med, count(empNo) as total from Empl eados
```

```
select count (distinct departamentos.deptNo) from Empl eados
```

- Las expresiones utilizadas en la cláusula WHERE pueden incluir entre otros:

- Operadores matemáticos: +, -, *, /
- Operadores de comparación: =, >, <, >=, <=, !=, like
- Operadores lógicos: and, or, not
- in, not in, between, is null, is not null,
- coalesce() (devuelve el primer valor distinto de null de sus argumentos)
- nullif() (devuelve null si el valor del argumento 1 coincide con valor argumento 2)
- Funciones de HQL
- Funciones SQL
- Parámetros posicionales JDB
- Constantes Java

Ejemplos.

```
select apellido, oficio from Empl eados where departamentos.deptNo in (10, 20)
```

```
select apellido, oficio from Empl eados where departamentos.deptNo not in (10, 20)
```

```
select apellido, oficio, salario from Empl eados
where salario between 2000 and 3000
```

```
select apellido, oficio from Empl eados where comision is null
```

```
select lower(apellido), coalesce(comision, 0) from Empl eados
```

```
select upper(apellido), nullif(comision, 0) from Empl eados
```

```
select apellido from Empl eados 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. Departamentos en los que el salario medio de los empleados es superior a 2000 euros.

```
select departamentos.dnombre, avg(salario) from Empleados
group by departamentos.deptNo
having avg(salario) > 2000
```

o bien, si las clases no están asociadas

```
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.

Ejemplos.

a) Empleados con salario superior al salario medio

```
select apellido, salario from Empleados
where salario >
(select avg(salario) from Empleados)
```

b) Empleados del mismo departamento que 'Gabarro'

```
select apellido, salario from Empleados
where departamentos.deptNo=
(select departamentos.deptNo from Empleados
where apellido like 'Gabarro' )
```

c) Empleados con salario mayor que cualquiera de los empleados del departamento 30.

```
select apellido, salario, departamentos.deptNo from Empleados
where salario >
(select max(salario) from Empleados
where departamentos.deptNo =30)
```

d) Apellido, oficio, salario y fecha del empleado empleado más antiguo de la empresa.

```
select apellido, oficio, salario, fechaAlta from Empleados
where fechaAlta =
(select min(fechaAlta) from Empleados)
```

11.1 Asociaciones y Uniones (JOINS)

Para poder realizar en HQL asociaciones con JOIN las tablas mapeadas deben estar relacionadas mediante claves primaria y ajena.

En el ejemplo de mapeo de las tablas MySQL (InnoDB) de la base de datos **empresaz** vimos como se generaban de forma automática las asociaciones entre las clases POJO generadas a partir de tablas que provenían de interrelaciones con claves foráneas. Estas asociaciones también se pueden realizar de manera manual, cuando las tablas subyacentes no están relacionadas mediante claves foráneas, como por ejemplo para motor de almacenamiento MyISAM.

En el caso de la base de datos **empresaz**, tenemos

<pre>CREATE TABLE empleados (emp_no SMALLINT(4) NOT NULL PRIMARY KEY, apellido ARCHAR(10), oficio VARCHAR(10), dir SMALLINT, fecha_alt DATE, salario FLOAT(6,2), comision FLOAT(6,2), dept_no TINYINT(2) NOT NULL, FOREIGN KEY (dept_no) REFERENCES departamentos(dept_no));</pre>	<pre>CREATE TABLE departamentos (dept_no TINYINT(2) NOT NULL PRIMARY KEY, denominbre VARCHAR(15), loc VARCHAR(15));</pre>
--	--

Tabla **DEPARTAMENTOS**: clave primaria columna **dept_no**

Tabla **EMPLEADOS**: clave primaria columna **emp_no**, clave ajena columna **dept_no** que referencia a la tabla DEPARTAMENTOS

Existe una relación uno-a-muchos (one-to-many) entre las clases persistentes Departamentos y Empleados, lo que significa que en la clase Departamentos tendremos una colección de objetos de la clase Empleados, es decir, por cada departamento podemos tener muchos empleados.

Las líneas en el fichero XML **Departamentos.hbm.xml** que indican esto son:

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

Donde:

- la **colección** se indica mediante la etiqueta **set**, que se le da un nombre (**empleados**),
- 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** (**dept_no**),
- el **tipo de relación** (one-to-many) y
- la **clase con la que se establece la relación** (<paquete>.Empleados).

En la clase **Departamentos.java** aparece la colección (**empleados**) con los setter y los getter:

```
public class Departamentos implements Serializable {
    -----
    private Set empleados = new HashSet(0);

    -----
    public Set getEmpleados() {
        return this.empleados;
    }
    public void setEmpleados(Set empleados) {
        this.empleados = empleados;
    }
}
```

Teniendo estas configuraciones se pueden realizar consultas con JOIN. Los tipos de JOIN soportados son:

- **INNER JOIN.**
- **LEFT OUTER JOIN (LEFT JOIN)**
- **RIGHT OUTER JOIN (RIGHT JOIN)**

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 enlace, en nuestro ejemplo empleados, que es donde se define la asociación y tenemos la colección.

INNER JOIN.

from Departamentos as dep **inner join** dep.empleados

Devuelve tantas instancias de dos objetos como resulte de combinar las tablas EMPLEADOS y DEPARTAMENTOS, incluyendo solo aquellos departamentos que tengan empleados asociados. Se obtiene por cada fila un objeto Departamentos (con índice 0) y un objeto Empleados (con índice 1).

Resultado SQL												Est
0 Fila(s) actualizada(s); 11 fila(s) seleccionada(s).												
DeptNo	Dnombre	Loc	Empleados	Dir	Salario	Comision	EmpNo	Apellido	Oficio	FechaAlt	Departamentos	
10	CONTABILIDAD	SEVILLA	[empresaz.entity.Empleados... 1	2721.74	0.0	1000	AMARO	AUXILIAR	2010-01-02	empresaz.entity.Depart...		
10	CONTABILIDAD	SEVILLA	[empresaz.entity.Empleados... 3000	2101.22	0.0	1001	ALONSO	AUXILIAR	2012-11-28	empresaz.entity.Depart...		
10	CONTABILIDAD	SEVILLA	[empresaz.entity.Empleados... 2	2500.0	0.0	2000	CASTRO	AUXILIAR	2011-01-02	empresaz.entity.Depart...		
10	CONTABILIDAD	SEVILLA	[empresaz.entity.Empleados... 3000	2023.64	500.0	2001	CASTILLO	AUXILIAR	2012-11-28	empresaz.entity.Depart...		
10	CONTABILIDAD	SEVILLA	[empresaz.entity.Empleados... 3	4195.51	0.0	3000	HERNANDEZ	CONTABLE	2001-01-02	empresaz.entity.Depart...		
10	CONTABILIDAD	SEVILLA	[empresaz.entity.Empleados... 3000	642.97	0.0	3001	HERMIDA	AUXILIAR	2012-11-28	empresaz.entity.Depart...		
30	VENTAS	BARCELONA	[empresaz.entity.Empleados... 7	1200.0	10.0	7000	GOMEZ	VENDEDOR	2002-01-02	empresaz.entity.Depart...		
30	VENTAS	BARCELONA	[empresaz.entity.Empleados... 6001	1100.0	0.0	7001	GABARRO	SUPERVIS.	2012-11-28	empresaz.entity.Depart...		
30	VENTAS	BARCELONA	[empresaz.entity.Empleados... 8	3500.0	150.0	8000	BORREGO	VENDEDOR	2002-01-02	empresaz.entity.Depart...		
40	PRODUCCIÓN	BILBAO	[empresaz.entity.Empleados... 6	3450.0	20.0	6000	DE ANDRES	GERENTE	2001-01-02	empresaz.entity.Depart...		
40	PRODUCCIÓN	BILBAO	[empresaz.entity.Empleados... 6000	2500.0	0.0	6001	DEMARIA	SUBDIREC.	2012-11-28	empresaz.entity.Depart...		

LEFT JOIN

from Departamentos as dep **left join** dep.empleados

Devuelve tantas instancias de dos objetos como resulte de combinar las tablas EMPLEADOS y DEPARTAMENTOS, incluyendo aquellos departamentos que no tengan empleados. Se obtiene por cada fila un objeto Departamentos y un objeto Empleados.

primero.Departamentos	primero.Empleados
primero.Departamentos@68022720	primero.Empleados@6556db38
primero.Departamentos@68022720	primero.Empleados@29526d82
primero.Departamentos@68022720	primero.Empleados@629386ff
primero.Departamentos@68022720	primero.Empleados@438401e8
primero.Departamentos@68022720	primero.Empleados@f9948f9
primero.Departamentos@68022720	primero.Empleados@1885db6f
primero.Departamentos@7c0960fc	null
primero.Departamentos@4bbe8002	null
primero.Departamentos@47c62a27	primero.Empleados@22f98b41
primero.Departamentos@47c62a27	primero.Empleados@2d256a57
primero.Departamentos@47c62a27	primero.Empleados@73688ce8
primero.Departamentos@54fb0e89	null
primero.Departamentos@2e776ff5	primero.Empleados@235755a8
primero.Departamentos@2e776ff5	primero.Empleados@3ab89ab8
primero.Departamentos@2db5c426	null
primero.Departamentos@7860b23b	null
primero.Departamentos@14557d3b	null
primero.Departamentos@7890b6ea	null
primero.Departamentos@64d8fa7c	null

Cláusula FETCH

Al utilizar “fetch conseguimos que con una select se devuelva una lista de objetos tipo-1 y la colección de objetos tipo-2 asociada a cada uno de ellos.

Es decir, estamos recuperando la información y se está creando la asociación entre los objetos de tipo-1 y tipo-2

```
from Departamentos as dep inner join fetch dep.empleados
```

Obtiene una lista con tantas instancias como resulte de combinar las tablas EMPLEADOS Y DEPARTAMENTOS, pero incluyendo sólo los departamentos que tienen asociados empleados y se recupera además la colección de empleados de cada departamento.

Empleados	DeptNo	Dnombre	Loc
[empresaz.entity.E...	10	CONTABILIDAD	SEVILLA
[empresaz.entity.E...	30	VENTAS	BARCELONA
[empresaz.entity.E...	40	PRODUCCIÓN	BILBAO

```
from Departamentos as dep left outer join fetch dep.empleados
```

Obtiene tantas instancias como resulte de combinar las tablas EMPLEADOS y DEPARTAMENTOS, incluyendo aquellos departamentos que no tengan empleados, pero ahora en cada fila solo se devuelve el objeto Departamentos con los datos de sus empleados cargados en la colección (atributo empleados).

DeptNo	Dnombre	Loc	Empleados
10	CONTABILIDAD	SEVILLA	[empresaz.entity.Empleado...
15	INFORMÁTICA	MADRID	[]
20	INVESTIGACIÓN	MADRID	[]
30	VENTAS	BARCELONA	[empresaz.entity.Empleado...
39	ANDROID	ALMERÍA	[]
40	PRODUCCIÓN	BILBAO	[empresaz.entity.Empleado...
70	INFORMÁTICA	TOLEDO	[]
80	I+D+I	Madrid	[]
81	PUBLICIDAD	Almería	[]
82	CONTRATOS	Granada	[]
86	MARKETING	GUADALAJARA	[]

CLáusula ORDER BY

```
from Departamentos as dep left outer join fetch dep.empleados order by dep.dnombre
```

Consulta el siguiente enlace para más detalle sobre HQL

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

EJEMPLOS en JAVA.

- **Consulta con inner join.** Recupera los datos de la consulta mediante dos objetos, el primero con los datos del departamento y el siguiente con los del empleado. Se recuperan sólo los departamentos con empleados.

```

SessionFactory sfactory = HibernateUtil.getSessionFactory();
Session session = sfactory.openSession();

System.out.println("=====");
System.out.println("Listado de Departamentos con INNER JOIN ");
Query q = session.createQuery("from Departamentos as dep inner join dep.empleados " +
    "order by dep.dnombre");
Iterator iter = q.iterator();
while (iter.hasNext()) {
    Object[] par = (Object[]) iter.next(); // obtener los dos objetos
    Departamentos de = (Departamentos) par[0]; // objeto Departamentos
    Emplados em = (Emplados) par[1]; // siguiente objeto Emplados

    System.out.println(de.getDnombre() + "*" + de.getLoc() + "*"
        + em.getApellido() + "*" + em.getSalario());
}
System.out.println("=====");
session.close();

```

Obteniendo un resultado similar al siguiente: (11 filas: 11 empleados con sus departamentos

```

=====
Listado de Departamentos con INNER JOIN
CONTABILIDAD*SEVILLA*AMARO*2721.74
CONTABILIDAD*SEVILLA*ALONSO*2101.22
CONTABILIDAD*SEVILLA*CASTRO*2500.0
CONTABILIDAD*SEVILLA*CASTILLO*2023.64
CONTABILIDAD*SEVILLA*HERNANDEZ*4195.51
CONTABILIDAD*SEVILLA*HERMIDA*642.97
PRODUCCIÓN*BILBAO*DE ANDRES*3450.0
PRODUCCIÓN*BILBAO*DEMARIA*2500.0
VENTAS*BARCELONA*GOMEZ*1200.0
VENTAS*BARCELONA*GABARRO*1100.0
VENTAS*BARCELONA*BORREGO*3500.0
=====
```

- **Consulta con left join.** Recupera los datos de la consulta mediante dos objetos, el primero con los datos del departamento y el siguiente con los del empleado. Se recuperan todos los departamentos

```

SessionFactory sfactory = HibernateUtil.getSessionFactory();
Session session = sfactory.openSession();
System.out
    .println("=====");
System.out.println("Listado de Departamentos con LEFT JOIN ");
Query q = session.createQuery("from Departamentos as dep left join dep.empladoses order by "
    + "dep.dnombre");
Iterator iter = q.iterator();
while (iter.hasNext()) {
    Object[] par = (Object[]) iter.next(); // obtener los dos objetos
    Departamentos de = (Departamentos) par[0]; // objeto Departamentos
    Emplados em = (Emplados) par[1]; // objeto Emplados
    if (em != null && de != null)
        System.out.println(de.getDnombre() + "*" + de.getLoc() + "*"
            + em.getApellido() + "*" + em.getSalario());
    if (em == null && de != null) // departamento sin empleados
        System.out.println(de.getDnombre() + "*"
            + de.getLoc() + " No hay empleados..");
}
System.out.println("=====");
session.close();

```

Obteniendo un resultado similar al siguiente: (19 filas: 11 empleados con sus departamentos + 8 departamentos sin empleados)

```
=====
Listado de Departamentos con LEFT JOIN
ANDROID*ALMERÍA No hay empleados..
CONTABILIDAD*SEVILLA*AMARO*2721.74
CONTABILIDAD*SEVILLA*ALONSO*2101.22
CONTABILIDAD*SEVILLA*CASTRO*2500.0
CONTABILIDAD*SEVILLA*CASTILLO*2023.64
CONTABILIDAD*SEVILLA*HERNANDEZ*4195.51
CONTABILIDAD*SEVILLA*HERMIDA*642.97
CONTRATOS*Granada No hay empleados..
I+D+I*Madrid No hay empleados..
INFORMÁTICA*MADRID No hay empleados..
INVESTIGACIÓN*MADRID No hay empleados..
MARKETING*GUADALAJARA No hay empleados..
PRODUCCIÓN*BILBAO*DE ANDRES*3450.0
PRODUCCIÓN*BILBAO*DEMARIA*2500.0
PUBLICIDAD*Almería No hay empleados..
VENTAS*BARCELONA*GOMEZ*1200.0
VENTAS*BARCELONA*GABARRO*1100.0
VENTAS*BARCELONA*BORREGO*3500.0
=====
```

- **Consulta con left join fetch.** Recupera los datos de la consulta mediante fetch, lo que indica que por cada departamento se carga una colección con los datos de los empleados de ese departamento:
 - La construcción fetch **no puede utilizarse en consultas Query ejecutadas con iterate()**. **Se ejecutará** la consulta mediante **list()**. Con list() al ejecutarse la consulta se carga en memoria la colección asociada.
 - Las consultas que hacen uso de una recuperación inmediata o temprana de colecciones usualmente retornan duplicados de los objetos raíz, pero con sus colecciones inicializadas.
 - Se pueden **filtrar** los duplicados con un **Set**

```
q = session.createQuery("from Departamentos as dep left join fetch dep.empleadoses order by"
                      + " dep.dnombre");
Departamentos depar = new Departamentos();
Empl eados emp e = new Empl eados();
// fetch no se puede utilizar en una consulta con iterate()

//Ejecuta la consulta eliminando duplicados
Set uni queq = new HashSet(q.list());

Iterator<?> di ter = uni queq.iterator(); // iterador para recorrer el resultado de la consulta
while (di ter.hasNext()) {
    depar = (Departamentos) di ter.next(); // obtiene objeto Departamentos
    System.out.println("*** " + depar.getDeptNo() + "*" + depar.getDnombre() + "*" +
    depar.getLoc());
    Set set = depar.getEmpl eadoses(); // obtiene los empleados del departamento
    System.out.println("Número de empleados: ." + set.size());
    // si hay empleados en el departamento
    if (set.size() != 0) {
        Iterator<?> ei ter = set.iterator(); // iterador para la colección de empleados
        while (ei ter.hasNext()) { // mientras hay empleados
            emp e = (Empl eados) ei ter.next(); // obtener un empleado
            System.out.println(emp e.getApel lido() + "****" + emp e.getOfici o());
        }
    }
}
System.out.println("-----");
```

Obteniendo un resultado similar al siguiente:

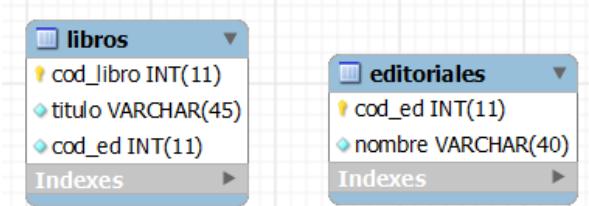
```
=====
Listado de Departamentos con LEFT JOIN FETCH
*** 39*ANDROID*ALMERÍA
Número de empleados:.0
-----
*** 10*CONTABILIDAD*SEVILLA
Número de empleados:.6
AMARO****AUXILIAR
HERMIDA****AUXILIAR
CASTRO****AUXILIAR
ALONSO****AUXILIAR
HERNANDEZ****CONTABLE
CASTILLO****AUXILIAR
-----
*** 80*I+D+I*Madrid
Número de empleados:.0
-----
*** 15*INFORMÁTICA*MADRID
Número de empleados:.0
-----
*** 70*INFORMÁTICA*TOLEDO
Número de empleados:.0
-----
*** 20*INVESTIGACIÓN*MADRID
Número de empleados:.0
-----
*** 86*MARKETING*GUADALAJARA
Número de empleados:.0|
-----
*** 40*PRODUCCIÓN*BILBAO
Número de empleados:.2
DE ANDRES****GERENTE
DEMARIA****SUBDIREC.
-----
*** 30*VENTAS*BARCELONA
Número de empleados:.3
BORREGO****VENDEDOR
GOMEZ****VENDEDOR
GABARRO****SUPERVIS.
```

Consulta el siguiente enlace para más información sobre los diferentes tipos de Mapeos de asociación
(uno-a-uno, uno-a-muchos, muchos-a-uno, muchos-a-muchos)

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

12. Ejemplos Resueltos

EJEMPLO 1. Base de datos **bibliotecaz** (**bibliotecaz.sql**). Modificaremos los ficheros de mapeo de forma manual para poder realizar JOINs en nuestra Aplicación Java.



```
CREATE TABLE libros(
    cod_libro int(11) NOT NULL PRIMARY KEY,
    titulo varchar(45) NOT NULL,
    cod_ed int(11) NOT NULL REFERENCES editoriales(cod_ed)
) ENGINE=MyISAM;
```

```
CREATE TABLE editoriales (
    cod_ed int(11) NOT NULL PRIMARY KEY,
    nombre varchar(40) NOT NULL
) ENGINE=MyISAM;
```

Una vez creado un proyecto Hibernate para trabajar con la base de datos MySQL bibliotecaz, hemos obtenido los siguientes ficheros de mapeo y POJOs:

Editoriales.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 23-ene-2013 11:31:05 by Hibernate Tools 3.2.1.GA -->
<hibernate-mapping>
    <class name="bibliotecaz.entity.Editoriales" table="editoriales" catalog="bibliotecaz">
        <id name="codEd" type="int">
            <column name="cod_ed" />
            <generator class="assigned" />
        </id>
        <property name="nombre" type="string">
            <column name="nombre" length="40" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Libros.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 23-ene-2013 11:31:05 by Hibernate Tools 3.2.1.GA -->
<hibernate-mapping>
    <class name="bibliotecaz.entity.Libros" table="libros" catalog="bibliotecaz">
        <id name="codLibro" type="int">
            <column name="cod_libro" />
            <generator class="assigned" />
        </id>
        <property name="titulo" type="string">
            <column name="titulo" length="45" not-null="true" />
        </property>
        <property name="codEd" type="int">
            <column name="cod_ed" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Editoriales.java

```
package bibliotecaz.entity;
// Generated 23-ene-2013 11:31:05 by Hibernate Tools 3.2.1.GA
/**
 * Editoriales generated by hbm2java
 */
public class Editoriales implements java.io.Serializable {
    private int codEd;
    private String nombre;
    public Editoriales() {
    }
    public Editoriales(int codEd, String nombre) {
        this.codEd = codEd;
        this.nombre = nombre;
    }
    public int getCodEd() {
        return this.codEd;
    }
    public void setCodEd(int codEd) {
        this.codEd = codEd;
    }
    public String getNombre() {
        return this.nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Libros.java

```
package bibliotecaz.entity;
public class Libros implements java.io.Serializable {
    private int codLibro;
    private String titulo;
    private int codEd;
    public Libros() {
    }
    public Libros(int codLibro, String titulo, int codEd) {
        this.codLibro = codLibro;
        this.titulo = titulo;
        this.codEd = codEd;
    }
    public int getCodLibro() {
        return this.codLibro;
    }
    public void setCodLibro(int codLibro) {
        this.codLibro = codLibro;
    }
    public String getTitulo() {
        return this.titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public int getCodEd() {
        return this.codEd;
    }
    public void setCodEd(int codEd) {
        this.codEd = codEd;
    }
}
```

Observa, que al no estar relacionadas las tablas mediante claves ajenas, en el fichero de mapeo **Editoriales.hbm.xml** no hay ninguna referencia mediante una colección (set) a los libros relacionados.

1. Creamos esta asociación manualmente. Añadimos las siguientes líneas al fichero **Editoriales.hbm.xml**, antes de la finalización de la clase:

```

<set name="enlace" table="libros" inverse="true">
    <key>
        <column name="cod_ed" not-null="true" />
    </key>
    <one-to-many class="bibliotecaz.entity.Libros" />
</set>
</class>
</hibernate-mapping>

```

Donde la colección se indica mediante la **etiqueta set**, que se le da un nombre (**enlace**), se indica el nombre de la tabla de donde se tomará esa colección de objetos (**libros**), la columna de la tabla por la que se relacionan (**cod_ed**), el tipo de relación (**one-to-many**) y la clase con la que se establece la relación (**bibliotecaz.entity.Libros**).

2. También es necesario añadir a la clase **Editoriales.java** la colección (**enlace**) con los **setter** y los **getter**:

```

private Set enlace = new HashSet(0);

public Set getEnlace() {
    return enlace;
}
public void setEnlace(Set enlace) {
    this.enlace = enlace;
}

```

Una vez realizados estos cambios se pueden realizar consultas con JOINS. Por ejemplo, listar todas las editoriales y los títulos publicados. Si la editorial no tiene libros asociados también aparece en el listado con el comentario “No hay libros ..”

```

public static void consultaJoin(SessionFactory sfactory) {
    Session session = sfactory.openSession();
    System.out
        .println("=====");
    System.out.println("Listado de Editoriales con LEFT JOIN ");
    Query q = session
        .createQuery("from Editoriales as ed left join ed.enlace order by "
            + "ed.nombre");
    Iterator iter = q.iterate();
    while (iter.hasNext()) {
        Object[] par = (Object[]) iter.next(); // obtener los dos objetos
        Editoriales ed = (Editoriales) par[0]; // primer objeto
        // Departamentos
        Libros li = (Libros) par[1]; // siguiente objeto Libros
        if (li != null && ed != null) {
            System.out.println(ed.getNombre() + "*" + li.getTitulo());
        }
        if (li == null && ed != null) // editorial sin libros
        {
            System.out.println(ed.getNombre() + " No hay Libros..");
        }
    }
    System.out
        .println("=====");
    session.close();
}

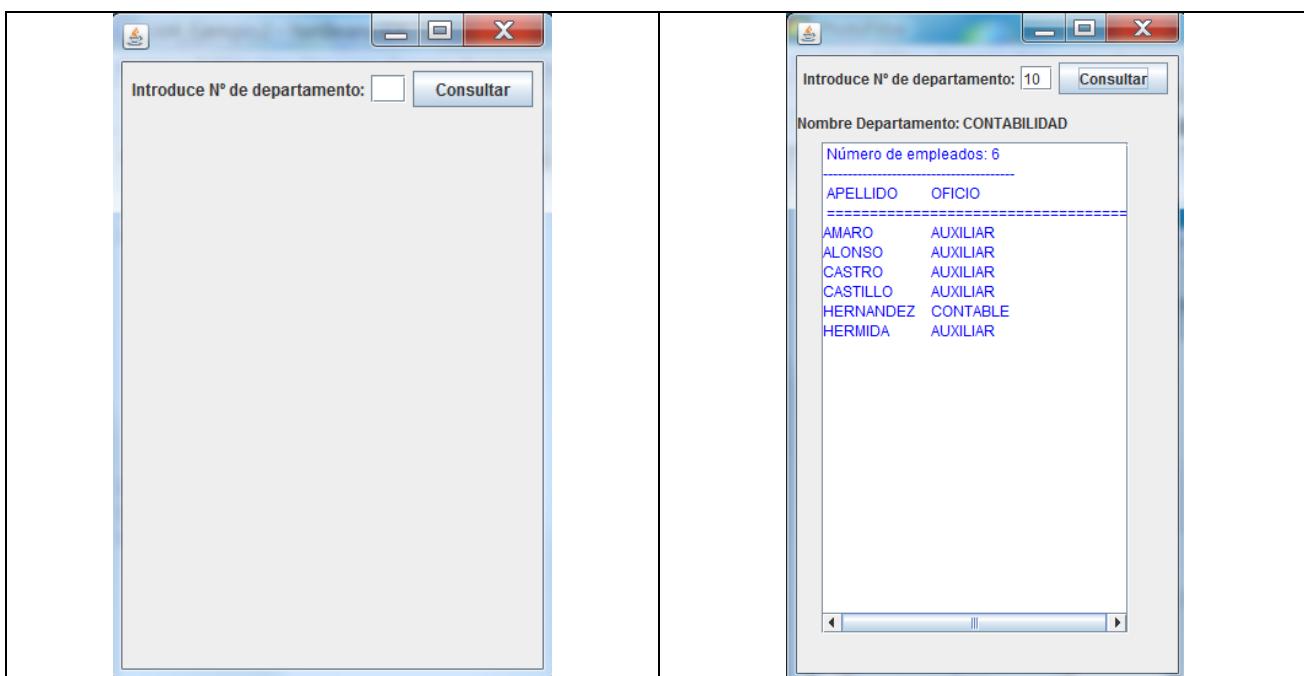
```

EJEMPLO 2. Base de datos **empresaz** (**empresaz.sql**, **insert_empresaz.sql**).

Aplicación para consultar los empleados de un departamento.

<code>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, FOREIGN KEY (dept_no) REFERENCES departamentos(dept_no));</code>	<code>CREATE TABLE departamentos (dept_no TINYINT(2) NOT NULL PRIMARY KEY, dnombre VARCHAR(15), loc VARCHAR(15));</code>
---	--

En este ejemplo, simplemente se ha incluido un sencillo interfaz gráfico a la aplicación para consultar los empleados de un departamento:



Crea un proyecto de nombre **Ud4_Ejemplo2** y realiza su configuración Hibernate con la siguiente estructura. Las clases **Ud4_Ejemplo2.java** y **Pantalla.java** contiene el código de la aplicación.

	<p>Se crearán dos clases para gestionar la aplicación:</p> <p>Ud4_Ejemplo2.java: clase principal con el método main(). Crea una instancia de la clase Pantalla y llama al método iniciar() que dibuja la pantalla y espera la introducción de un valor y pulsemos el botón consultar.</p> <p>Pantalla.java: en la que se definen varios métodos:</p> <ul style="list-style-type: none"> - iniciar(): inicializa la pantalla - actionPerformed(ActionEvent e): controla la acción a realizar cuando pulsemos el botón. - visualizarDep(int dep): recibe el número de un departamento y visualiza sus datos. - visualizarEmp(int dep): recibe el número de un departamento y visualiza los datos de sus empleados
--	---

El código es el siguiente de las clases **Ud4_Ejemplo2.java** y **Pantalla.java** es el siguiente:

```
package Ud4_Ejemplo_2;
public class Main {
    public static void main(String[] args) {
        final Pantalla ventana = new Pantalla();
        ventana.iniciar();
    }
}
```

```
package Ud4_Ejemplo_2;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.Rectangle;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Iterator;
import java.util.List;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
import empresaz.entity.Departamentos;
import empresaz.entity.Empleados;
import empresaz.util.HibernateUtil;
public class Pantalla extends JFrame implements ActionListener{
    JLabel etiqueta=new JLabel ("");
    JLabel label = new JLabel ("Introduce Nº de departamento:");
    JTextField depar = new JTextField(2);
    JButton boton = new JButton ("Consultar");
    SessionFactory sfactory = HibernateUtil.getSessionFactory();
    //contenedor para botones
    public void iniciar(){
        getContentPane().setLayout(new GridLayout(10, 1));
        JPanel panelBotones = new JPanel (new FlowLayout());
        panelBotones.add(label);
        panelBotones.add(depar);
        panelBotones.add(boton);
        getContentPane().add(panelBotones);
        getContentPane().add(etiqueta);
        setVisible(true);
        pack();
        boton.addActionListener(this); //pulsamos botón
    }
    //acción cuando pulsamos el botón
    @Override
    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==boton){ //se pulsa el botón
            etiqueta.setText("Departamento a consultar: " + depar.getText());
            int dep;
            try{
                dep=Integer.parseInt(depar.getText());
                visualizarDep(dep); //visualiza datos del departamento
            }catch (java.lang.NumberFormatException ex){
                etiqueta.setText("Departamento erróneo");
            }
        }
    }
}
```

```

//Localizar datos del departamento
void visualizarDep (int dep){ //visualiza datos del dep
    Session session = sfactory.openSession();

    Departamentos depart = (Departamentos) session.createQuery(
        "from Departamentos where de.deptNo = ?")
        .setInteger(0, dep).uniqueResult();

    if(depart!=null){
        etiqueta.setText("Nombre Departamento: "+depart.getDnombre());
        visualizarEmp(dep); //visualiza los empleados
    }else{
        etiqueta.setText("No existe el departamento "+ dep);
    }
    session.close();
}

//Localizar datos de los empleados
void visualizarEmp(int dep){ //visualiza datos de los empleados del dep
    JTextArea area = new JTextArea();
    JScrollPane scroll = new JScrollPane(area);
    //prepara el área para escribir los empleados
    area.setBounds(new Rectangle(20, 70, 250, 400));
    scroll.setBounds(new Rectangle(20, 70, 250, 400));
    area.setEditable(false);
    getContentPane().add(scroll, null);
    getContentPane().setLayout(null);
    area.setForeground(Color.blue);

    //Sesión Hibernar
    Session session = sfactory.openSession();

    Query q = session.createQuery("from Empleados where "
        + "departamentos.deptNo = :ndep");
    //asigna parámetros a consulta
    q.setInteger("ndep", dep);

    Empleados emple = new Empleados();

    //ejecuta la consulta con método list()
    List<Empleados> lista = q.list();

    area.append(" Número de empleados: " + lista.size()+"\n");
    area.append("-----");
    area.append("\n APELLIDO\tOFICIO");
    area.append("\n =====");

    //iterador para recorrer el resultado de la consulta
    Iterator<?> iter = lista.iterator();
    while(iter.hasNext()){
        emple = (Empleados)iter.next();
        String cad="\n "+emple.getApellido()+"\t"+emple.getOficio();
        area.append(cad);
    }
    session.close();
}
}

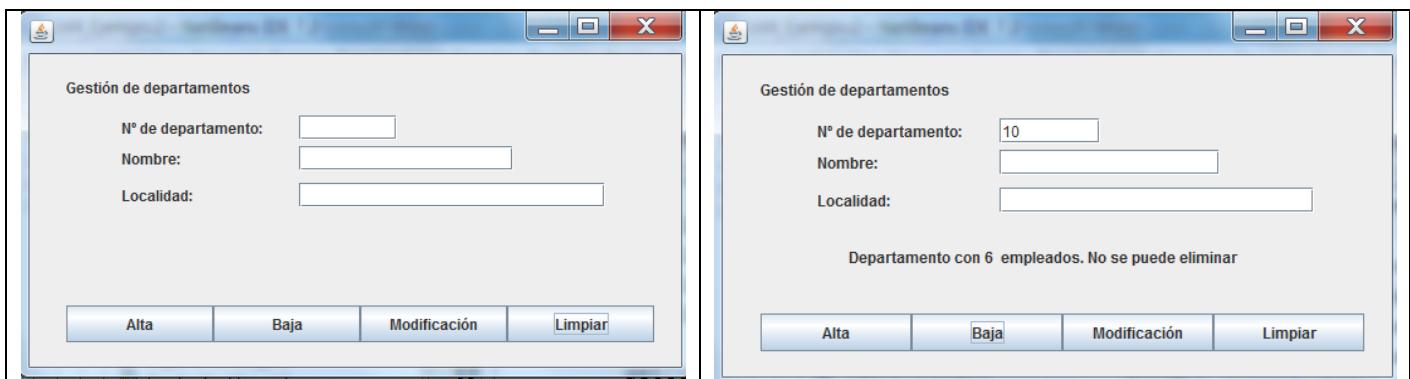
```

Puedes modificar este y el siguiente ejemplo utilizando una interfaz de usuario diseñada mediante las herramientas que estés viendo en el módulo de “Desarrollo de Interfaces”.

EJEMPLO 3. Base de datos **empresaz** (**empresaz.sql**, **insert_empresaz.sql**).

Aplicación para realizar altas, bajas y modificaciones de departamentos.

Al ejecutar el programa se muestra una pantalla donde se visualizan 3 campos de entrada para introducir datos del departamento y 4 botones que nos permitirán introducir, eliminar y modificar un departamento. Además un botón para limpiar pantalla. Después de realizar un alta, baja o modificación se visualiza un mensaje indicando la acción realizada. Igualmente, se indica mediante un mensaje el motivo por el cual no se puede insertar, eliminar o modificar un departamento



Como en la aplicación anterior, la clase **Pantalla2.java** es la encargada de realizar la operaciones de altas, bajas y modificaciones mediante los métodos: **insertaDep()**, **bajaDep()**, **modificaDep()**

```
//insertar nuevos departamentos
void insertaDep(int num, String nom, String loc) {
    SessionFactory sfactory = HibernateUtil.getSessionFactory();
    Session session = sfactory.openSession();
    Transaction tx = session.beginTransaction();

    //Consulta para ver si existe el departamento
    Departamentos depart = (Departamentos) session.createQuery(
        "from Departamentos where deptNo= ?")
        .setInteger(0, num)
        .uniqueResult();

    //si el departamento ya existe
    if (depart != null) {
        etiqueta.setText("Departamento existente. No se puede dar de alta");
        tx.rollback();
    } else { //insertar un nuevo departamento
        Departamentos d = new Departamentos();
        if(nom.length()>15) {
            nom=nom.substring(0, 15);
        }
        d.setDnombre(nom);
        if(loc.length()>15) {
            loc=loc.substring(0, 15);
        }
        d.setLoc(loc);
        d.setDeptNo((byte)num);
        session.save(d);
        tx.commit();
        etiqueta.setText("Departamento insertado");
    }
    session.close();
}
```

```

//eliminar departamentos
void bajaDep(int num) {
    SessionFactory sfactory = HibernateUtil.getSessionFactory();
    Session session = sfactory.openSession();
    Transaction tx = session.beginTransaction();

    Departamentos de=new Departamentos();
    de=(Departamentos) session.get(Departamentos.class, (byte) num);
    if(de!=null){
        //comprobamos si tiene empleados
        Query q=session.createQuery(" select count (empNo)"
            + " from Empleados where departamentos.deptNo=?")
            .setInteger(0, num);
        Long numdep= (Long) q.uniqueResult();
        if(numdep==0){
            session.delete(de); //elimina objeto
            tx.commit();
            etiqueta.setText("Eliminado");
        } else{
            etiqueta.setText("Departamento con " + numdep + " empleados."
                + " No se puede eliminar");
            tx.rollback();
        }
    }else{
        etiqueta.setText("Departamento inexistente - no se puede eliminar");
        tx.rollback();
    }
    session.close();
}

//modificar departamentos
void modificaDep(int num, String nom, String loc) {
    SessionFactory sfactory = HibernateUtil.getSessionFactory();
    Session session = sfactory.openSession();
    Transaction tx = session.beginTransaction();

    @SuppressWarnings("UnusedAssignment")
    Departamentos de = new Departamentos();
    de=(Departamentos) session.get(Departamentos.class, (byte) num);
    if(de!=null){
        if(nom.length()>15) {
            nom=nom.substring(0, 15);
        }
        de.setDnombre(nom);
        if(loc.length()>15) {
            loc=loc.substring(0, 15);
        }
        de.setLoc(loc);
        session.update(de);
        tx.commit();
        etiqueta.setText("Departamento modificado");
    }else{
        etiqueta.setText("Departamento inexistente - no se puede modificar");
        tx.rollback();
    }
    session.close();
}

```

13. Bibliografía.

- Acceso a Datos , Editorial Garceta
- Java, cómo programar, Edición 9, Editorial DEITEL
- <http://www.dzone.com/tutorials/java/hibernate/hibernate-tutorial/hibernate-tutorial.html>
- <http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=hibernate>
- <http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/index.html>
- <http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/index.html>