Mapeo Objeto_Relacional
Hibernate - Parte 1

1. Introducción

En este tema veremos otra forma de acceder a una base de datos relacional, utilizando mapeo Objeto-Relacional también denominado ORM (Object-Relational Mapping).

2. El desfase objeto-relacional

Con el desarrollo de software Orientado a Objetos ha surgido un problema con respecto al almacenamiento y gestión de datos.

Por un lado tenemos que los datos suelen estar almacenados en bases de datos relacionales siguiendo el paradigma relacional, basado en principios matemáticos con tablas y relaciones entre estas. Por otro lado tenemos los programas que siguen el paradigma POO, se basa en clases, objetos y herencia, estos dos paradigmas no conviven fácilmente.

Esto se ve por ejemplo en la manera de acceder a los datos. Mientras que en el mundo OO los objetos se relacionan a través de propiedades, en el mundo relacional las tablas se unen a través de claves ajenas.

Este problema es conocido como desfase objeto-relacional. Las herramientas ORM tratan de solucionar este problema.

3. El Mapeo Objeto-Relacional. Características

El mapeo Objeto-Relacional consiste en una interfaz que traduce la lógica relacional en lógica de objetos, es una herramienta que nos permite convertir los elementos de la base de datos relacional en objetos.

Por ejemplo, las tablas pasan a ser clases, las filas objetos, los procedimientos almacenados métodos, etc.

3.1. Pros

La principal ventaja del ORM es una integración del acceso a datos relacionales dentro de un proyecto orientado a objetos, y permite el uso de características propias de la POO como son la herencia y el polimorfismo.

Se reduce el tiempo de desarrollo de software, el acceso a los datos es transparente para el programador.

EL programador puede centrar sus esfuerzos en la lógica de negocio.

El código queda mucho más limpio y se redacta menos código, lo cual minimiza la probabilidad de errores.

Las aplicaciones son totalmente independientes de los sistemas en los que están almacenados los datos. Se puede migrar de una base de datos a otra sin tocar nada del código.

Permiten convertir datos entre sistemas incompatibles.

3.2. Contras

La traducción que realiza la herramienta emplea más tiempo por lo cual el rendimiento de las consultas puede decaer.

Muchas bases de datos optimizan las consultas y pueden mejorar los tiempos de respuesta. Al generar SQL en caliente se pierde esta oportunidad de optimización.

Algunas herramientas ORM no optimizan la consulta para la base de datos utilizada sino que generan un SQL genérico lo que resulta en una muy mal utilización de funciones específicas de la base de datos.

Al tiempo empleado en el desarrollo hay que sumar el tiempo empleado en la configuración del mapeo. Con muchas herramientas hay que crear archivos de configuración que le van a indicar a la herramienta cómo tiene que mapear cada clase a sus respectivas tablas.

Para terminar hay que tener en cuenta que las herramientas ORM han madurado mucho durante los últimos años, muchas se utilizan en sistemas grandes con muchas transacciones por segundo. El rendimiento ha sido su talón de Aquiles pero muchos de los problemas de rendimiento han sido superados utilizando caché, guardando los datos frecuentemente utilizados en memoria etc. La ganancia en tiempo de desarrollo es muy grande para desechar este tipo de herramientas, en algunos casos en que el rendimiento no es suficiente siempre existe la posibilidad de no utilizar la herramienta en esos casos.

4. Herramientas ORM

Se han desarrollado muchas herramientas, algunas gratuitas, otras no, para determinados lenguajes de programación o determinados entornos de programación, pero todas tienen como puntos comunes que realizan el mapeo Objeto-Relacional y casi todas incluyen un lenguaje de consultas orientado a objetos propio que es totalmente independiente de la base de datos que se vaya a usar.

Estas son algunas que se pueden encontrar actualmente en el mercado:

- Para **Python** tenemos:
  - **STORM**
    https://storm.canonical.com/
  - **SQLOBJECT**
    http://www.sqlobject.org/
- Para **Java**:
  - **HIBERNATE** con HQL (Hibernate Query Language)
    http://www.hibernate.org/
  - **NHIBERNATE** HIBERNATE para .NET
  - **Oracle Toplink** integrado en NetBeans
- Para **Php**:
  - **Doctrine** con DQL (Doctrine Query Language) inspirado en HQL, lo incluye el framework Symfony.
    http://www.doctrine-project.org/
  - **Propel**
    http://propelorm.org/
    Es otro framework ORM para PHP 5 y superior, lo incluye el framework Symfony.
  - **ADOdb Active Record**
    http://phplens.com/lens/adodb/docs-active-record.htm
- Para **.NET**:
  - **NHIBERNATE**
  - **LINQ** http://msdn.microsoft.com/es-es/library/bb397897(v=vs.100)

Nosotros, ya que empezamos el estudio de manejo de conexiones en Java desde NetBeans, seguiremos con Java y practicaremos con Hibernate.

5. HIBERNATE - Introducción

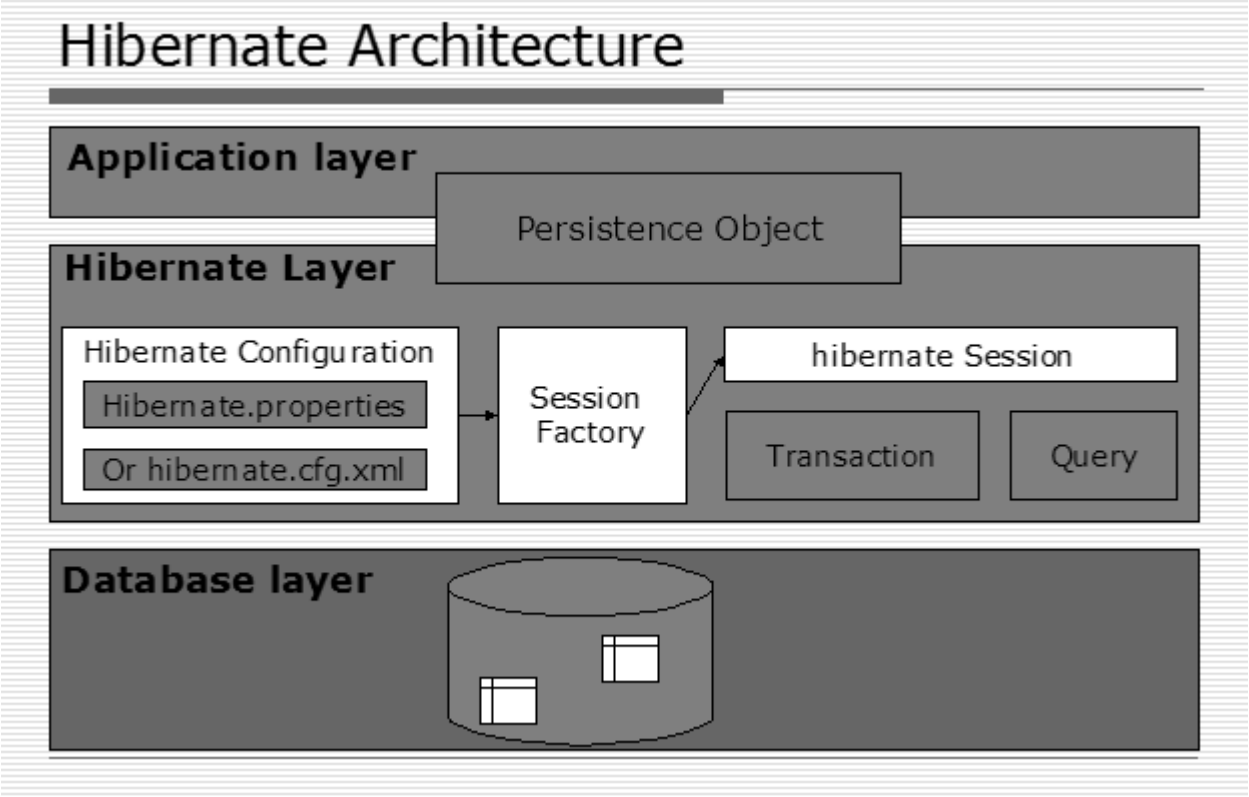
Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java (y disponible también para .Net) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante ficheros declarativos (XML) que permiten establecer esta relación o anotaciones en las clases (nosotros, en este tema solo veremos la primera opción).

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

Con Hibernate no emplearemos habitualmente SQL para acceder a datos sino que el propio motor de Hibernate que, 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.

5.1. Arquitectura de Hibernate

Hibernate parte de una filosofía de mapear objetos Java normales o más conocidos en la comunidad como "POJOs' (Plain Old Java Objects). Para almacenar y recuperar estos objetos de la base de datos, el desarrollador debe mantener una conversación con el motor de Hibernate mediante un objeto especial que es la sesión (clase Session) (equiparable al concepto de conexión de JDBC). Igual que con las conexiones JDBC hemos de crear y cerrar sesiones.



La clase **Session** (org.hibernate.Session) ofrece métodos como save(Object objeto), createQuery(String, consulta), beginTransaction(), close(), para interactuar con la BD tal 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(miObjecto), 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, quizá en cada petición.

Puede ser útil pensar en una sesión como en una colección de objetos cargados 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.

SessionFactory (org.hibernate.sessionFactory) configura Hibernate utilizando los archivos (hibernate.properties o hibernate.cfg.xml) y permite instanciar objetos Session.

Esta interfaz debe compartirse entre muchos hilos de ejecución. Normalmente hay una única SessionFactory para toda la aplicación, creada durante la inicialización de la misma, y se utiliza para crear todas las sesiones relacionadas con un contexto dado. Si la aplicación accede a varias bases de datos se necesitará una SessionFactory por cada base de datos.

Configuration (org.hibernate.cfg.Configuration) se utiliza para configurar Hibernate.

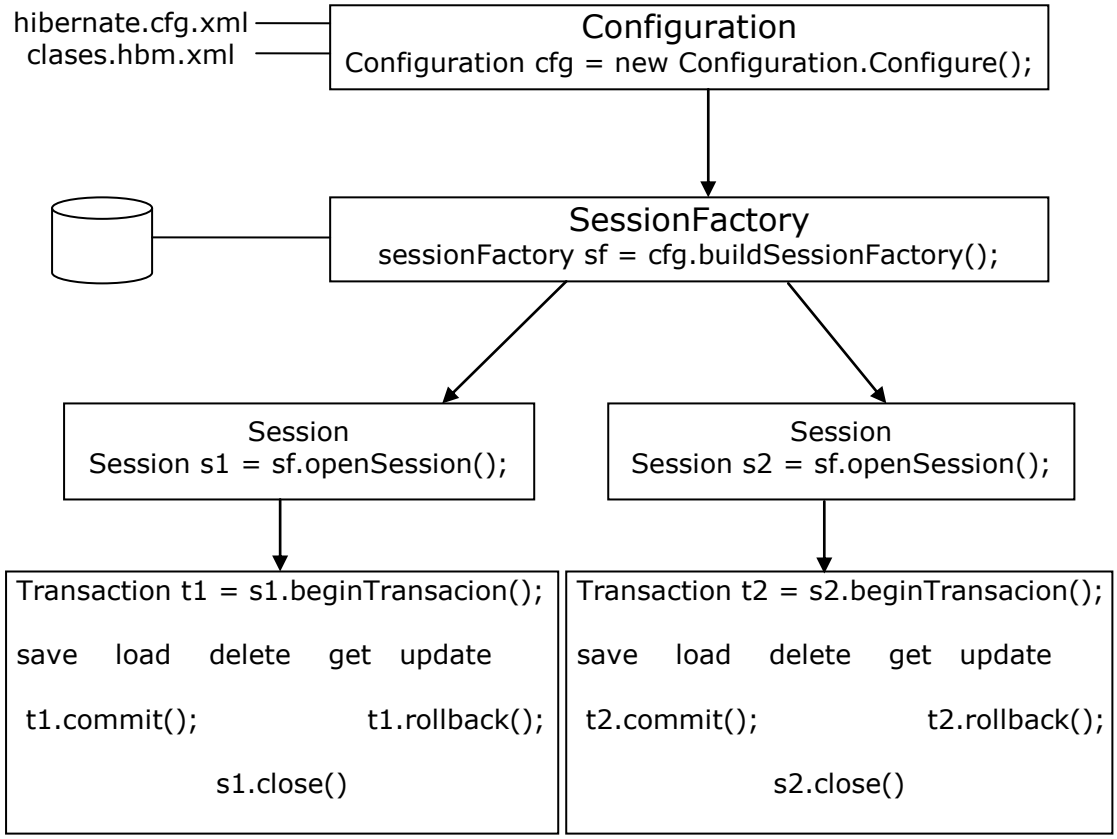
La aplicación utiliza una instancia de Configuration para especificar la ubicación de los documentos que indican el mapeado de los objetos y propiedades específicas de Hibernate, y a continuación crea la SessionFactory.

Transaction (org.hibernate.Transaction) nos permite asegurar que cualquier error que ocurra entre el inicio y final de la transacción produzca el fallo de la misma .

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

Query (org.hibernate.Query) permite realizar consultas a la base de datos y controla cómo se ejecutan dichas consultas. Las consultas se escriben en HQL o en el dialecto SQL nativo de la base de datos que estemos utilizando. Una instancia Query se utiliza para enlazar los parámetros de la consulta, limitar el número de resultados devueltos y para ejecutar dicha consulta.

El siguiente diagrama muestra cómo sería una aplicación con Hibernate:



6. Instalación y configuración de Hibernate

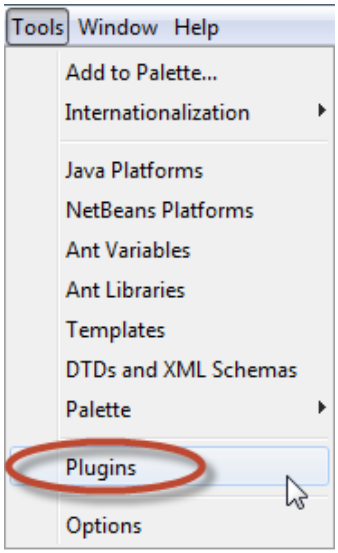
Vamos a instalar Hibernate sobre NetBeans.

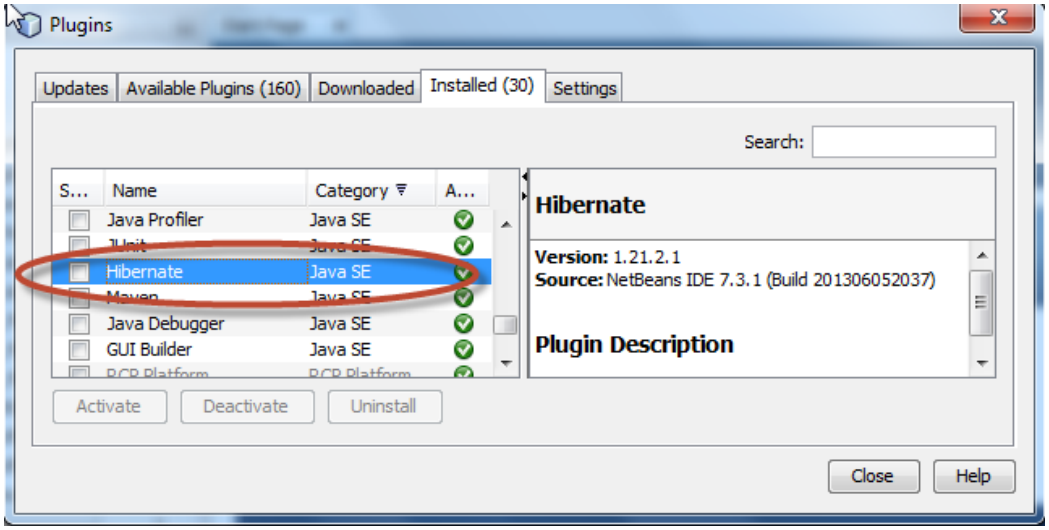
Para los ejemplos vamos a utilizar una base de datos MySQL de nombre *ejemplo*, similar a la creada en la unidad anterior, su propietario es el usuario *ejemplo* y la clave también *ejemplo*. Esta base de datos contiene las tablas empleados y departamentos siguientes:

CREATE TABLE departamentos (: dept_NO TINYINT NOT NULL PRIMARY KEY, dnombre VARCHAR (15) , loc VARCHAR (15),);	CREATE TABLE empleados (: Emp_no SMALLINT NOT NULL PRIMARY KEY, apellido VARCHAR (20) , oficio VARCHAR (15) , dir SMALLINT, fecha_alta DATE, salario FLOAT (6,2), comision FLOAT(6,2), dept_NO TINYINT REFERENCES departamentos);

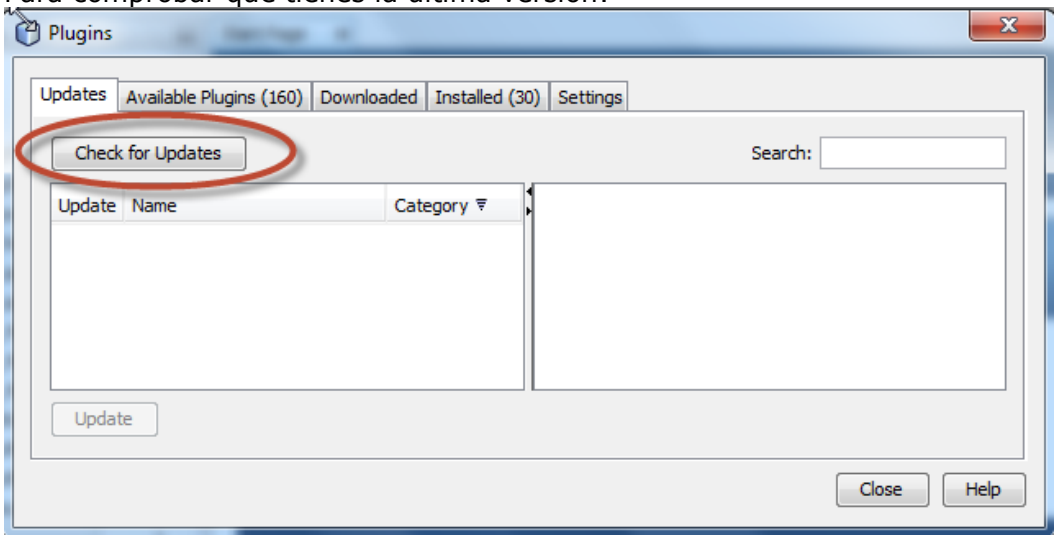
6.1. Instalación

Para trabajar con Hibernate se necesita tener en nuestro IDE el plug-in de hibernate, en el caso de la versión 7.3 de NetBeans lleva incorporado Hibernate, solo tienes que comprobar que tienes la última versión disponible.





Para comprobar que tienes la última versión:

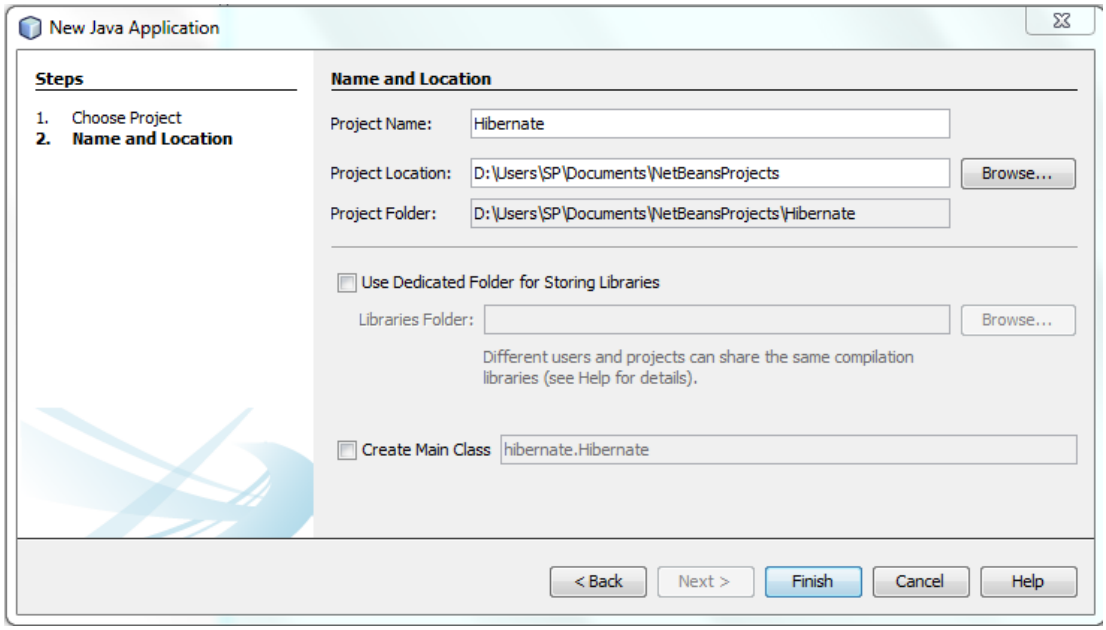


Si te quieres descargar Hibernate, esta es la página:
http://www.hibernate.org/downloads

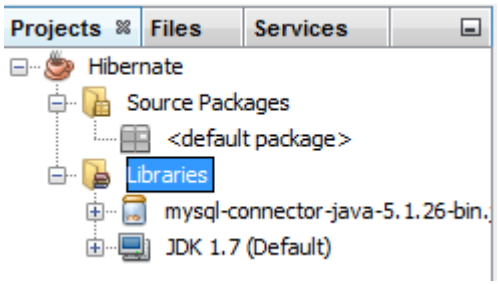
El archivo a descargar será un hibernate-release-XXXX.zip.
Una vez descargado lo descomprimimos donde nos guardamos todas nuestras librerías.

A continuación podemos empezar con la aplicación.

Crea un nuevo proyecto java desmarcando la casilla *Create Main File* .



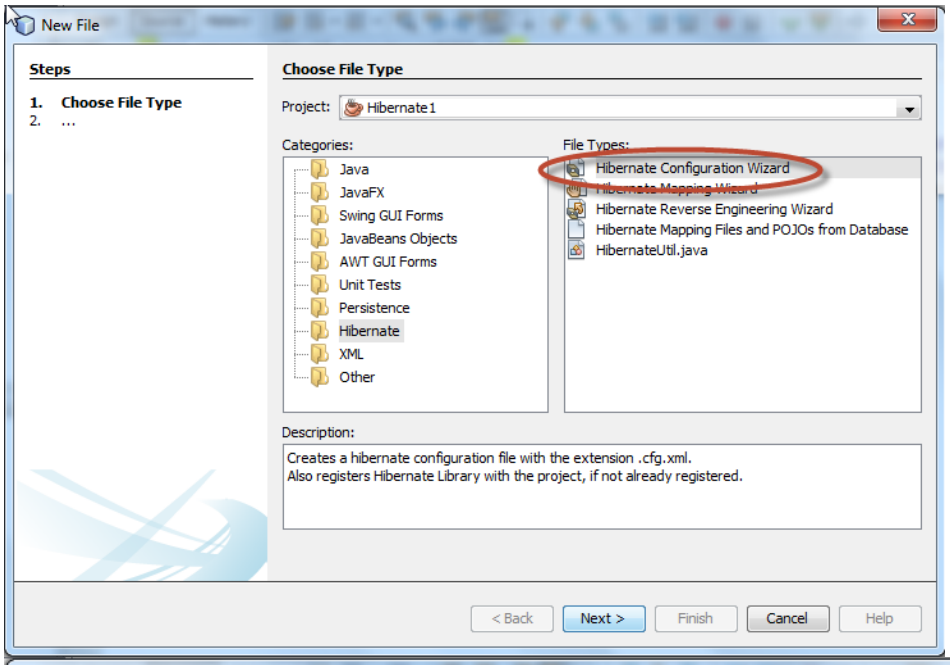
Como necesitaremos acceder a una base de datos, para poder establecer la conexión necesitamos añadir el driver correpondiente (en este caso mysql) a la libreria para que luego Hibernate pueda conectarse a nuestra base de datos MySQL. Hazlo como hicimos en el tema anterior, añadiendo el driver a la libreria

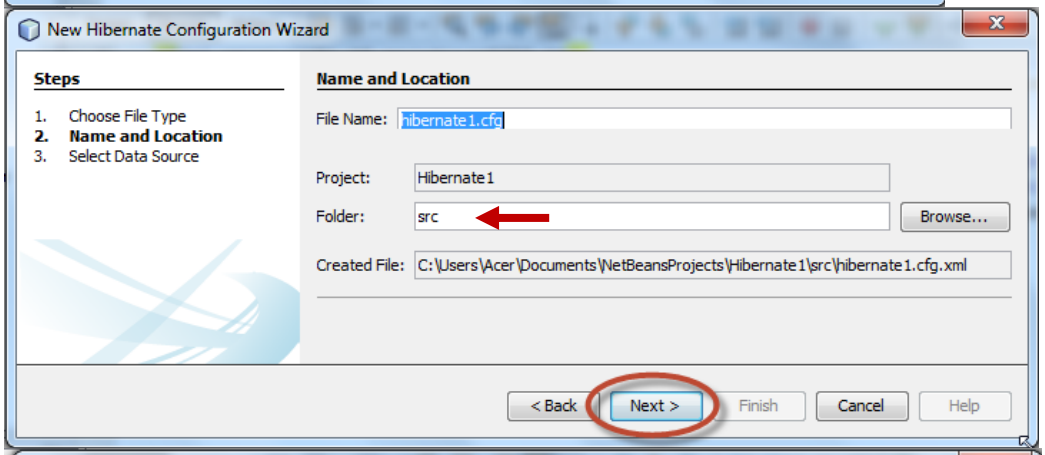


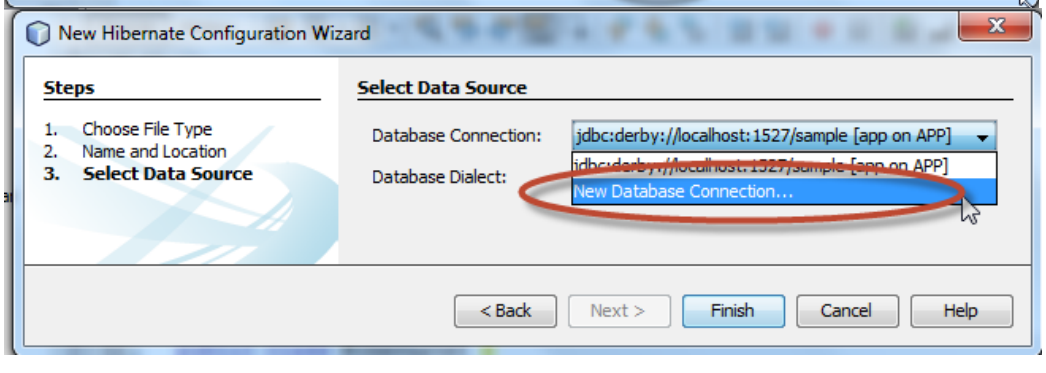
6.2. Generar el archivo de Configuración

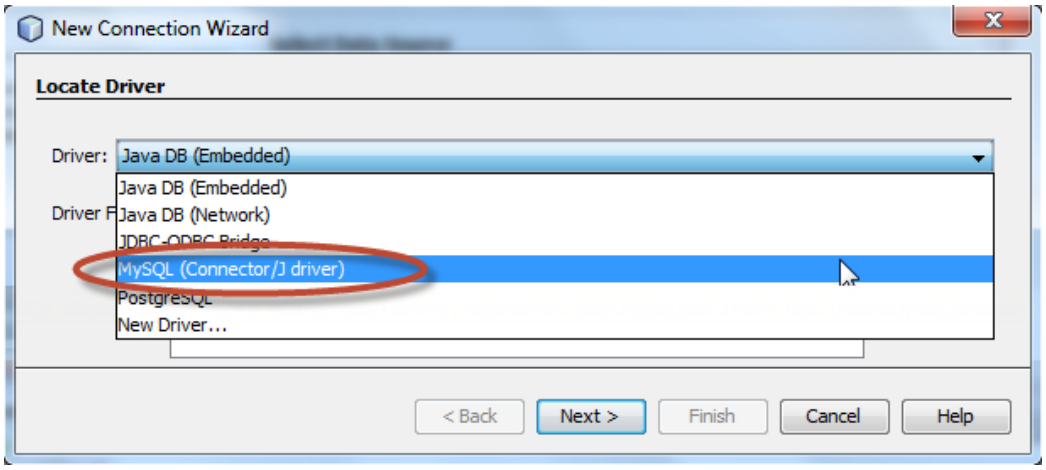
Una vez tenemos el driver mysql en nuestro proyecto hemos de **crear el fichero de configuración** de Hibernate **hibernate.cfg.xml**.

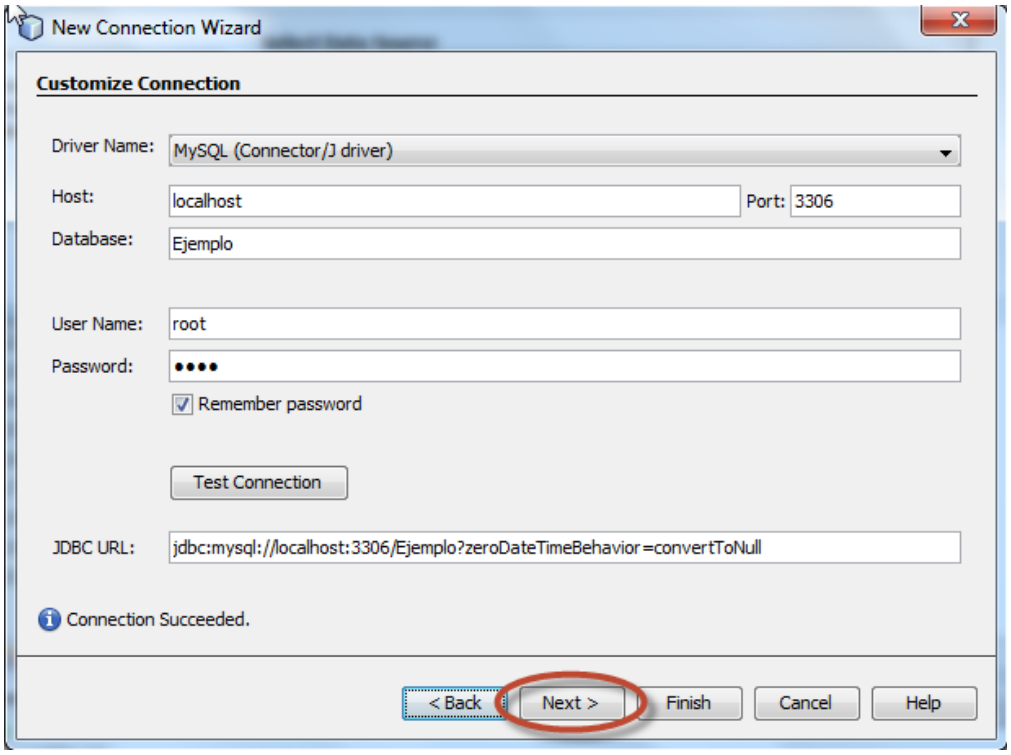
Clic derecho sobre el proyecto→ New→Other… → Hibernate → Hibernate Configuration Wizard:


New File
Steps
1. Choose File Type
2. ...
Choose File Type
Project: Hibernate1
Categories:
Java
JavaFX
Swing GUI Forms
JavaBeans Objects
AWT GUI Forms
Unit Tests
Persistence
Hibernate
XML
Other
File Types:
Hibernate Configuration Wizard
Hibernate Mapping Wizard
Hibernate Reverse Engineering Wizard
Hibernate Mapping Files and POJOs from Database
HibernateUtil.java
Description:
Creates a hibernate configuration file with the extension .cfg.xml.
Also registers Hibernate Library with the project, if not already registered.
< Back
Next >
Finish
Cancel
Help

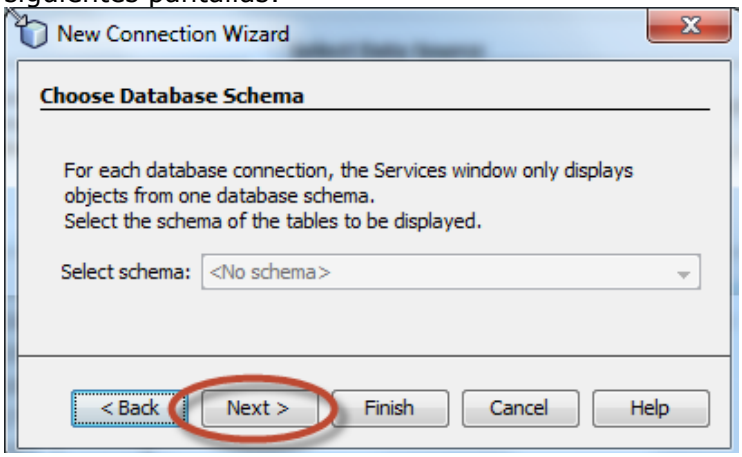

New Hibernate Configuration Wizard
Steps
1. Choose File Type
2. Name and Location
3. Select Data Source
Name and Location
File Name: hibernate1.cfg
Project: Hibernate1
Folder: src
Browse...
Created File: C:\Users\Acer\Documents\NetBeansProjects\Hibernate1\src\hibernate1.cfg.xml
< Back
Next >
Finish
Cancel
Help


New Hibernate Configuration Wizard
Steps
1. Choose File Type
2. Name and Location
3. Select Data Source
Select Data Source
Database Connection: jdbc:derby://localhost:1527/sample [app on APP]
Database Dialect:
New Database Connection...
< Back
Next >
Finish
Cancel
Help

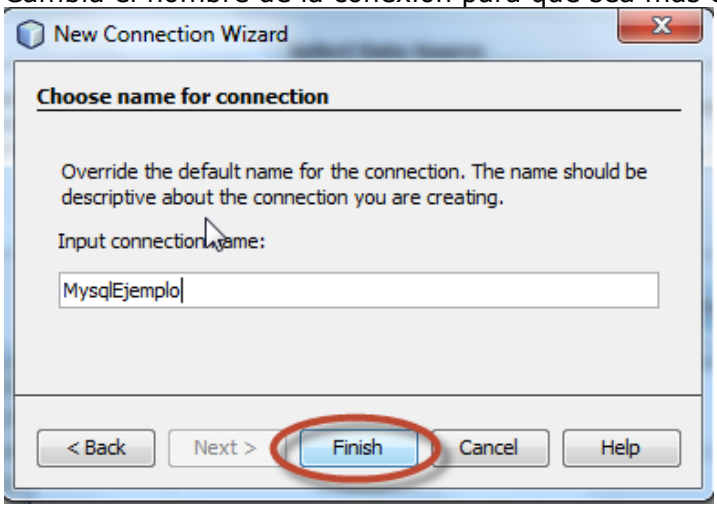

New Connection Wizard
Locate Driver
Driver: Java DB (Embedded)
Java DB (Embedded)
Java DB (Network)
MySQL (Connector/J driver)
New Driver...
< Back
Next >
Finish
Cancel
Help



Antes de pasar a la siguiente pantalla y para evitar posteriores errores y quebraderos de cabeza se recomienda **probar la conexión: Botón *Test Connection***. Continua pulsando *Next>* en las siguientes pantallas:



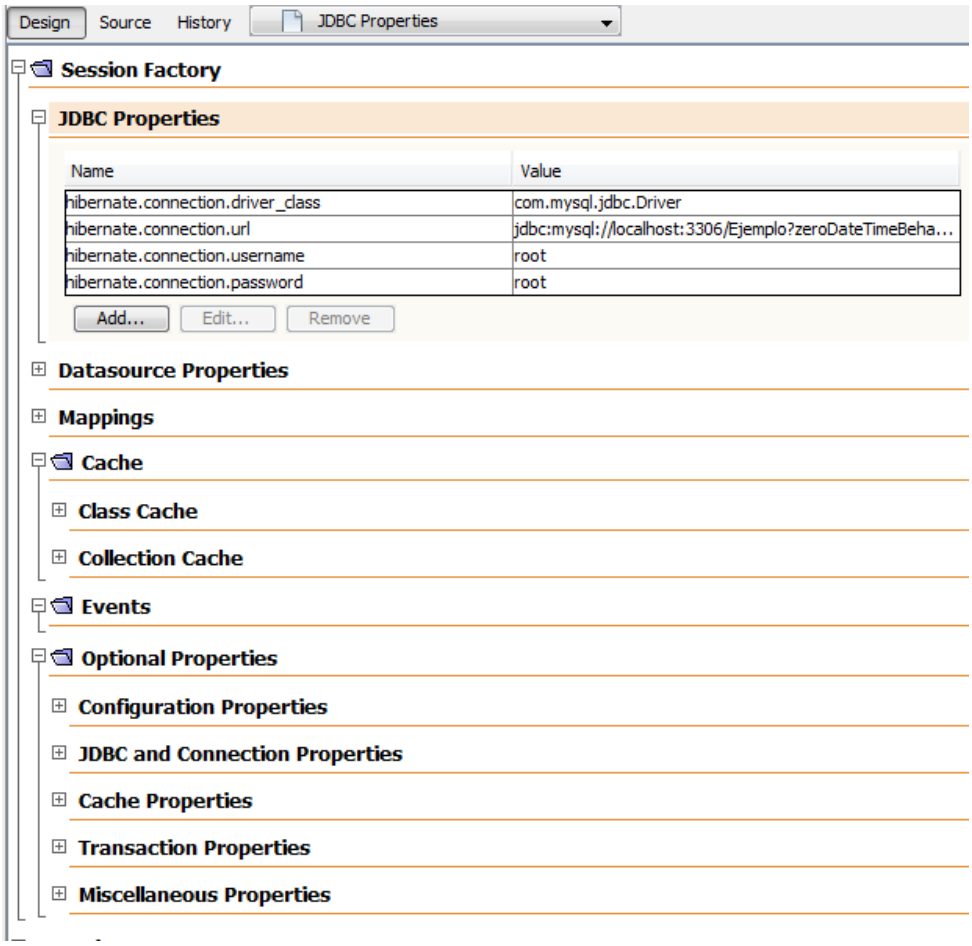
Cambia el nombre de la conexión para que sea más explicativo:



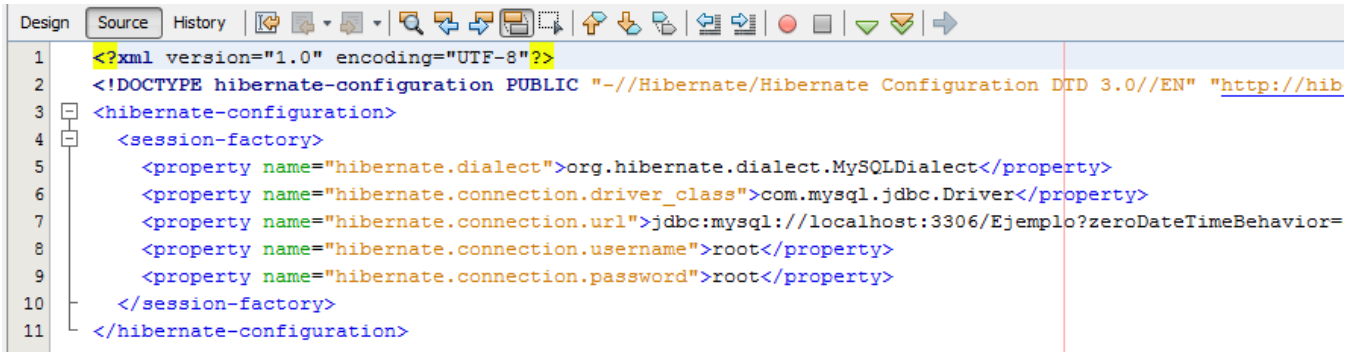
Al pulsar *Finish* se habrá creado el fichero de configuración.

La nueva conexión creada se podrá reutilizar en posteriores proyectos, en ese último caso, en lugar de elegir la opción New Database connection (tercera imagen página 7) se elegirá la conexión de la lista que aparece encima de la opción citada.

El resultado es el archivo hibernate.cfg.xml, que puedes visualizar en vista *Design*:



O si quieres ver el código fuente generado, lo tienes en la pestaña *Source*:

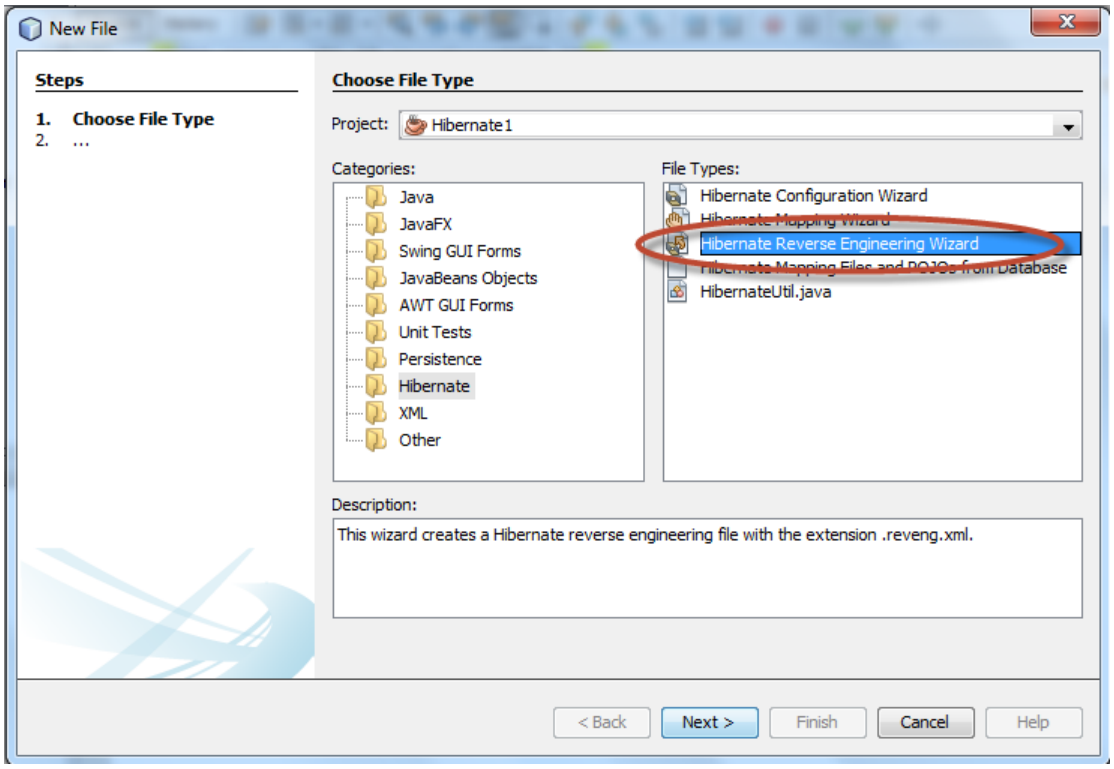


Un vez tenemos el fichero de configuración, debemos **crear el fichero de Reverse Engineering hibernate.reveng.xml**.

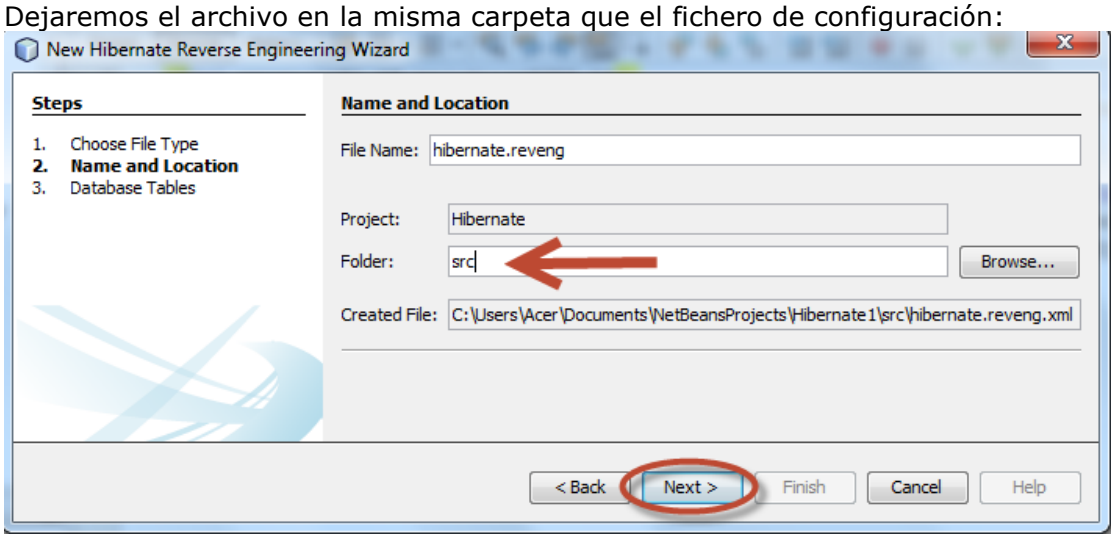
6.3. Generar el archivo de ingenieria inversa

Este archivo es el que usará Hibernate para generar las clases a partir de las tablas de la base de datos.

Como antes, clic derecho sobre el proyecto→ New→Other… → Hibernate → ahora Hibernate Reverse Engineering Wizard

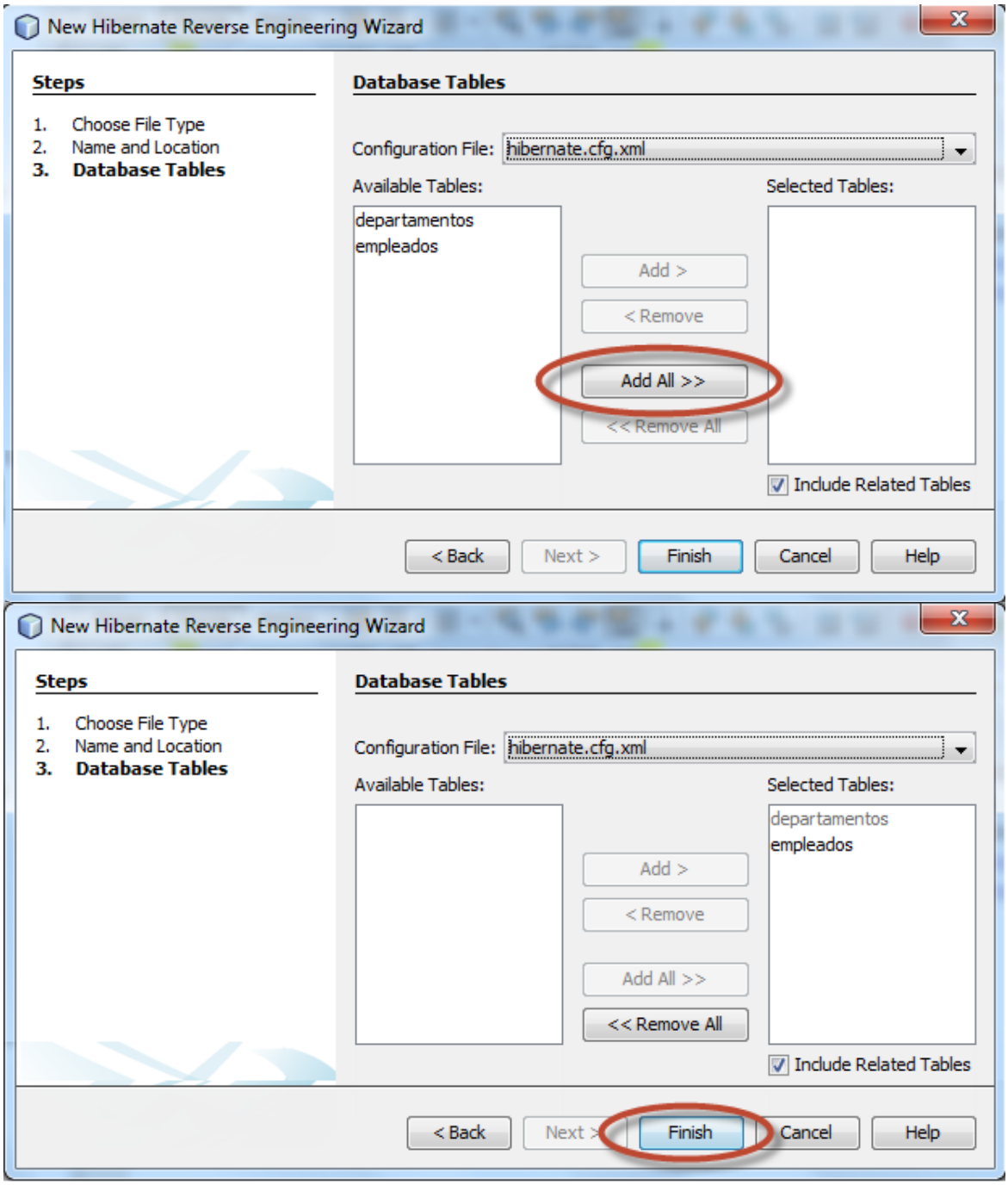


Dejaremos el archivo en la misma carpeta que el fichero de configuración:

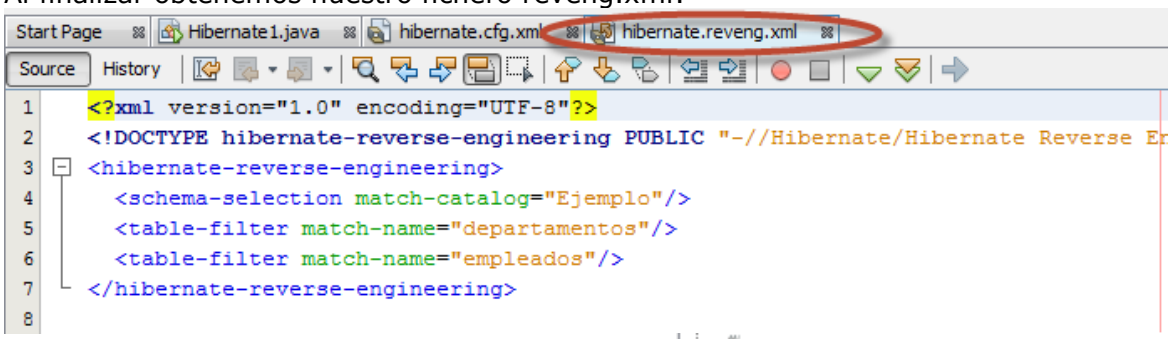


Hibernate se conecta a la base de datos para obtener las tablas y que seleccionemos las que queremos incluir en el mapa.

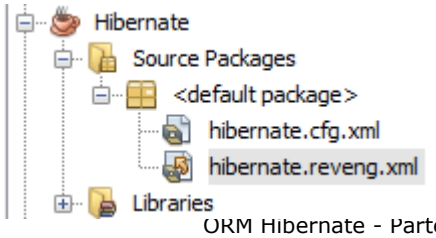
Nota. Si no aparece en la siguiente imagen la lista de tablas incluidas en la base de datos puedes tener un error de conexión, revisa la cadena de conexión que has indicado en el archivo .cfg.xml y comprueba que tengas agregado a la librería del proyecto el driver correspondiente.



Al finalizar obtenemos nuestro fichero reveng.xml:



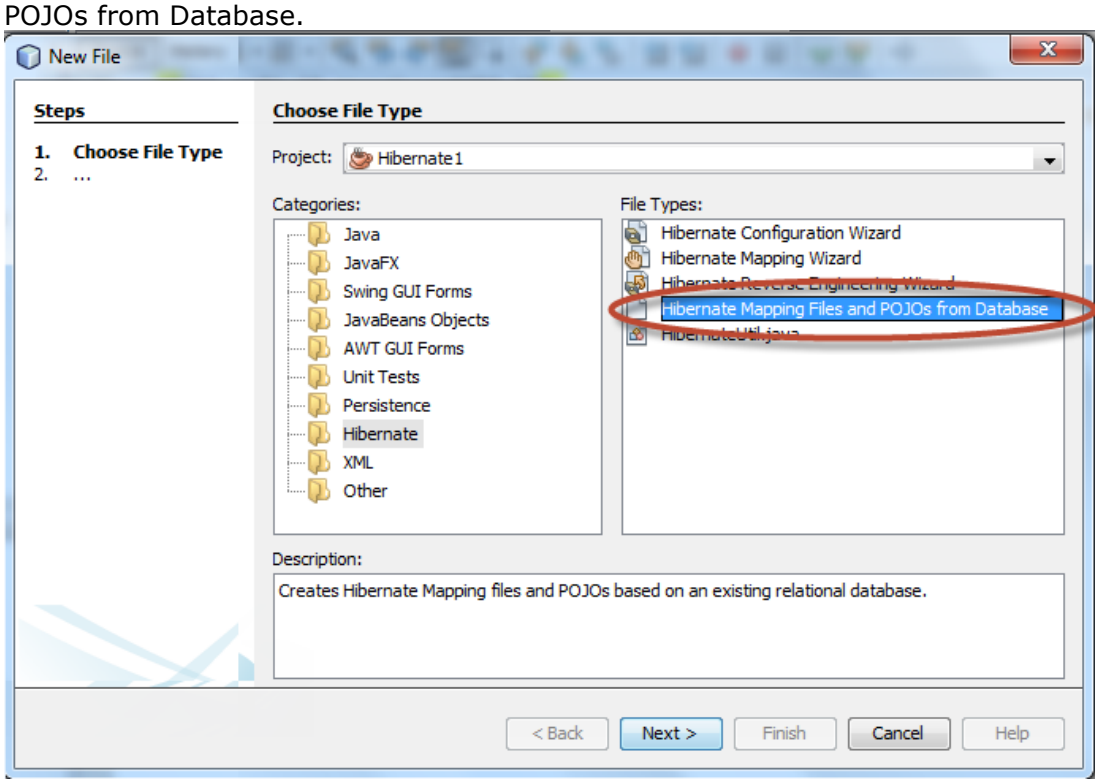
Esto es lo tenemos en el proyecto hasta ahora:

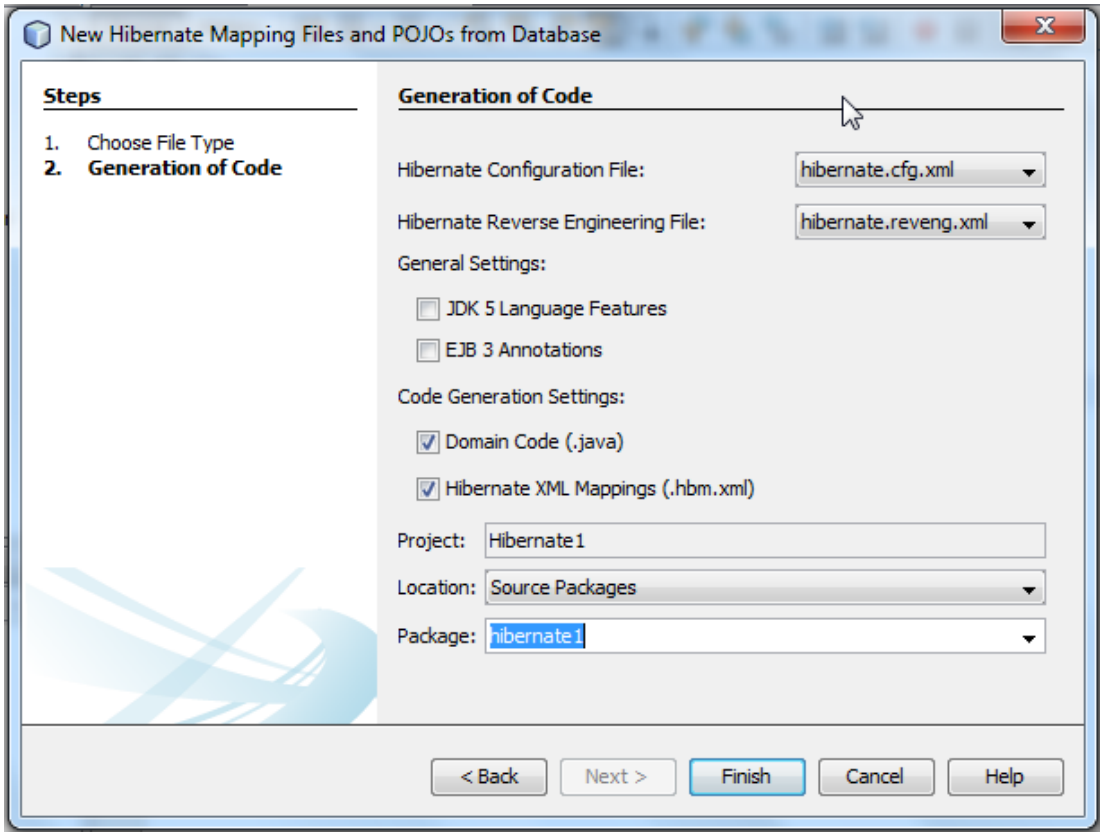


6.4. Generar las clases persistentes

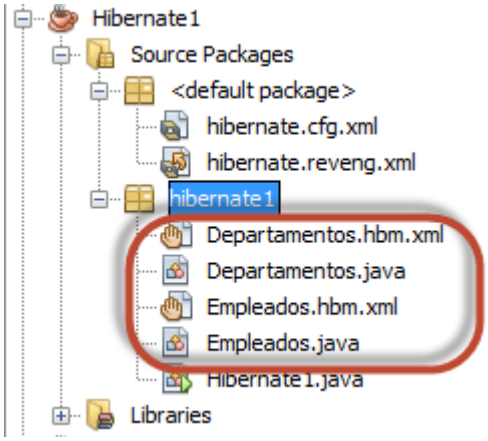
Hasta ahora hemos preparado un lado del mapa, indicando la base de datos que se mapea, las tablas que queremos añadir al mapa, ahora nos falta definir las clases correspondiente a esas tablas que permitirán definir los objetos que se usarán para manejar la información almacenada.

Clic derecho sobre el proyecto→ New→Other… → Hibernate → ahora Hibernate Mapping Files and POJOs from Database.





Al finalizar, se habrán creado las clases correspondientes a las tablas seleccionadas en el paso anterior e incluidas en el archivo de ingeniería inversa:



Por cada tabla se ha creado un .java con la definición de la clase.

En nuestro proyecto se han generado las clases Empleados.java y Departamentos.java. A esas clases se les llama clases persistentes porque sus objetos son objetos persistentes, objetos que se guardarán. La clase "representa" una tabla (su esquema), y los objetos de esa clase las filas de la tabla.

Todas las clases generadas tienen atributos, constructores y los métodos get y set para poder acceder a los atributos. Se utilizan convenciones de nombrado estándares de]avaBean para los métodos de propiedades getter y setter así como también visibilidad privada para los campos.

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, solo hay que poner la primera letra que los une en mayúsculas.
- Los métodos marcados como setter deben ser declarados como públicos, deyolver un 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.

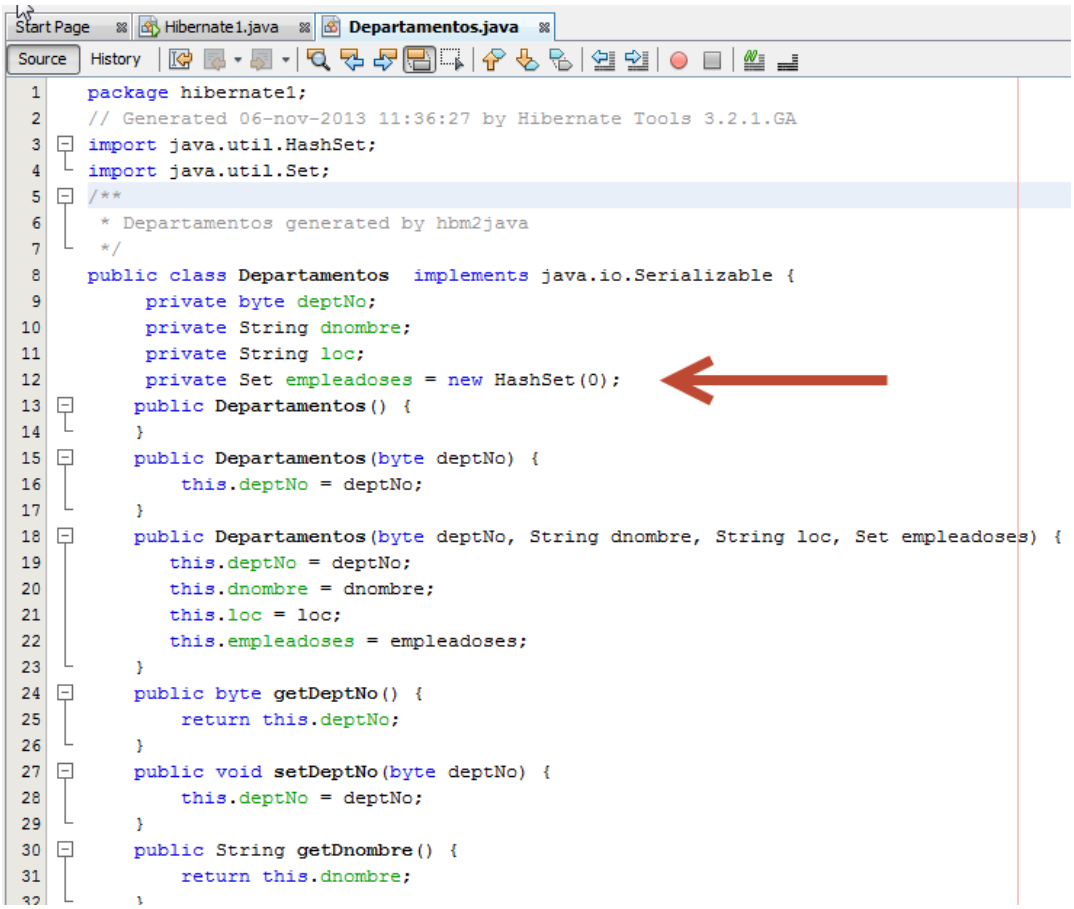
Como los atributos de los objetos son privados se crean métodos públicos para retornar el yalor de un atributo (método getter), o para cargar un valor a un atributo (método setter), por ejemplo el método getDnombre devuelve el nombre de un departamento (atributo dnombre) y el método setDnombre carga un valor en el atributo dnombre.

A estas reglas también se las llama modelo de programación POJO - Plain Old Java Object.

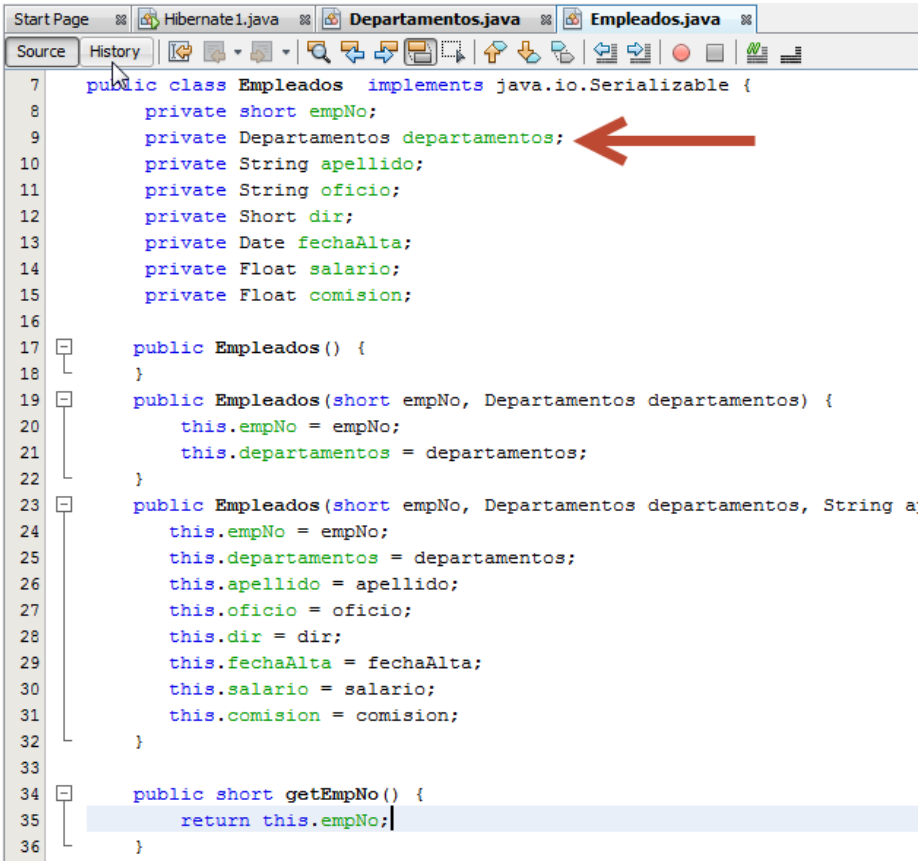
En la imagen siguiente puedes ver cómo se ha creado, una clase por cada tabla, en la que el equivalente a las columnas de la tabla son atributos de la clase.

La relaciones entre tablas se convierten en asociaciones entre clases. En la clase correspondiente a la tabla hija tendremos una asociación unidireccional (muchos-a-uno) y en la tabla padre tendremos una asociación uno-a-muchos.

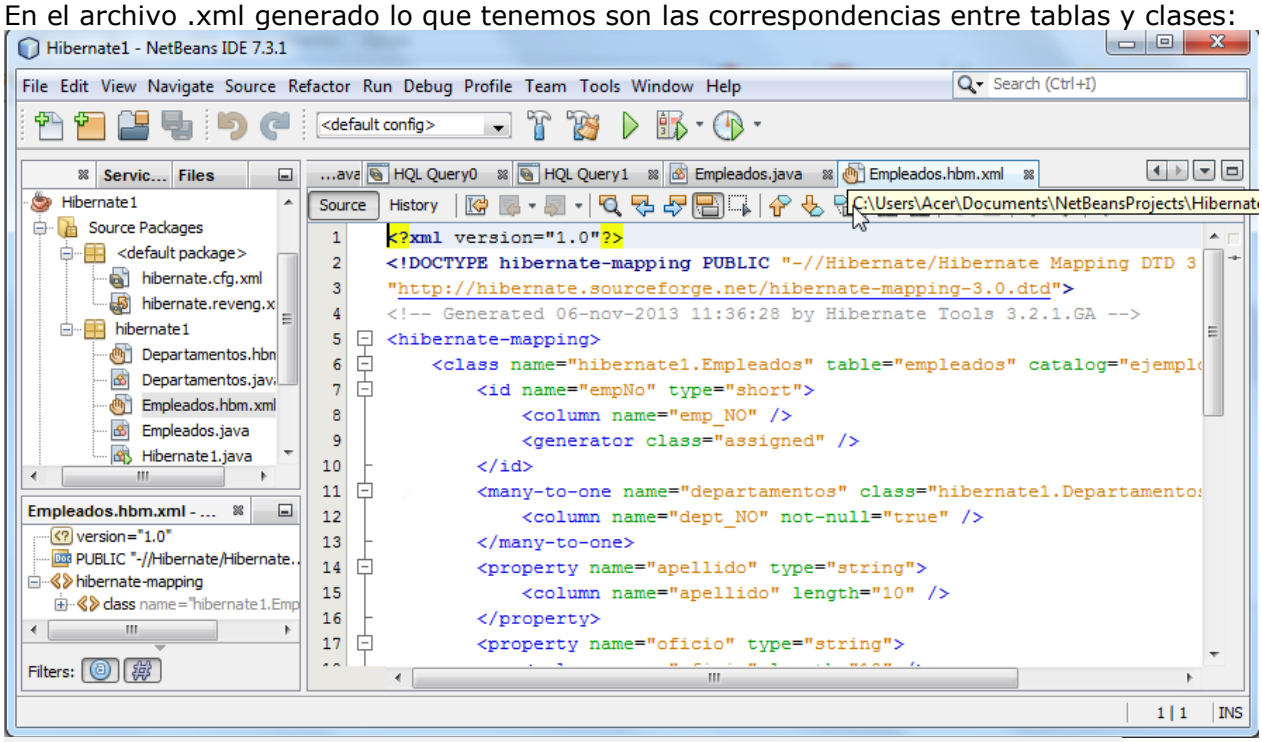
En este caso estoy en la tabla Departamentos, un departamento puede tener asignados varios empleados, luego la asociación es uno-a-muchos, el atributo será un Set:

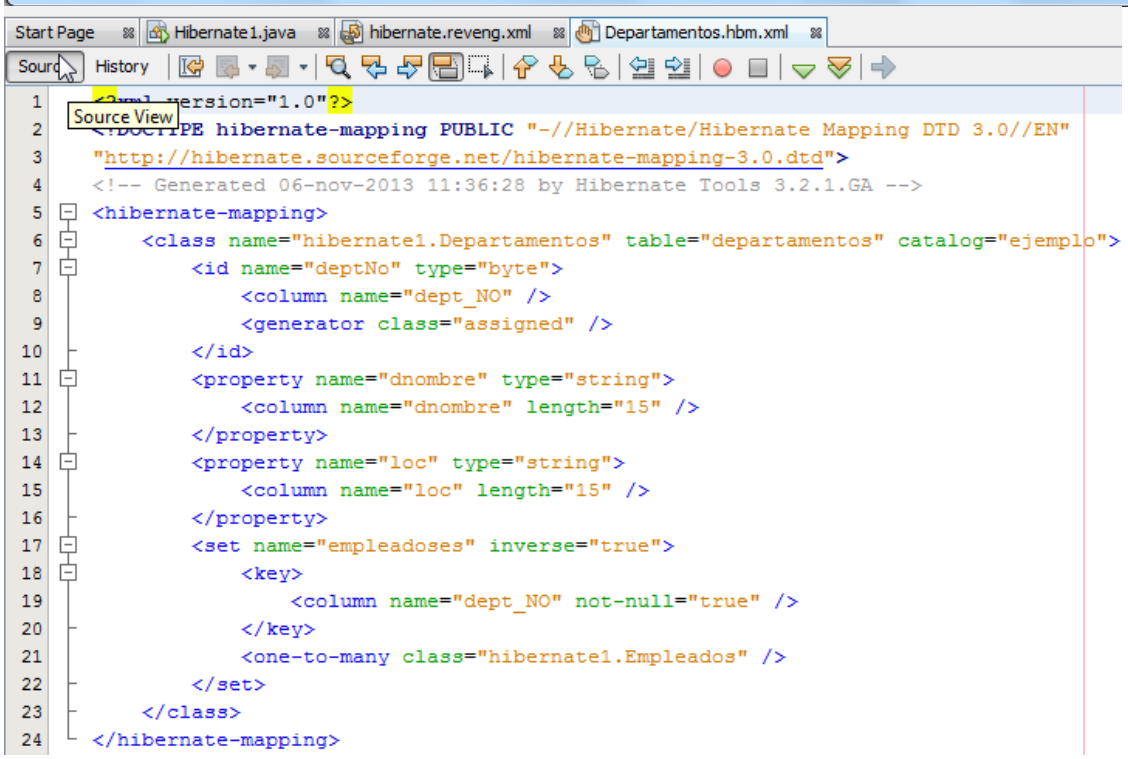


En la tabla Empleados tendremos una asociación muchos-a-uno, un empleado está asignado a un solo departamento, estamos en la tabla hija (que contiene la clave ajena), el atributo correspondiente es un atributo simple, pero ahora su tipo es de la clase Departamentos en vez de short (el tipo correspondiente a códigos de departamento).



En el archivo .xml generado lo que tenemos son las correspondencias entre tablas y clases:





Entre las etiquetas <hibernate-mapping> </ hibernate-mapping > de los ficheros XML se incluye un elemento class que hace referencia a una clase y contiene la correspondencias entre la tabla y la clase:
<class name="hibernate1.Departamentos" table="departamentos" catalog="ejemplo"> indica que la clase *Departamentos* se corresponde con la tabla *departamentos* del catálogo *ejemplo*.

<id...>...</id> que la propiedad identificadora *id* de la clase se corresponde con la columna de nombre *dept_NO*, esta información la genera por ser dept_NO clave principal.

<property...>...</property> que la propiedad *dnombre* de la clase se corresponde con la columna *dnombre* de la tabla, ... y así sucesivamente.

También tenemos definida la asociación entre Departamentos y empleados, en este caso es una uno-a-muchos (one-to-many) mientras que en empleados es una muchos a uno (many-to-one).

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

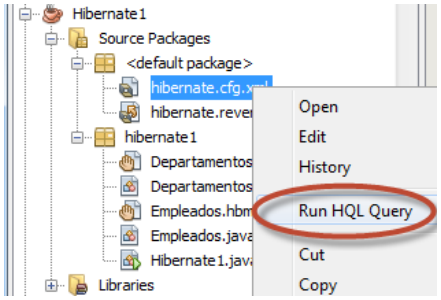
Nota. Tanto en el .java, como en el .hbm.xml, si no aparecen las características que hemos indicado referentes a la relación entre las dos tablas, es porque la relación no está definida en la base de datos (foreign key), revisa en la base de datos la definición de las tablas, la tabla hija deberá contener una clave ajena que haga referencia a la tabla padre, el campo de la tabla referenciada será generalmente su clave principal.

7. HQL - El lenguaje de consulta de Hibernate

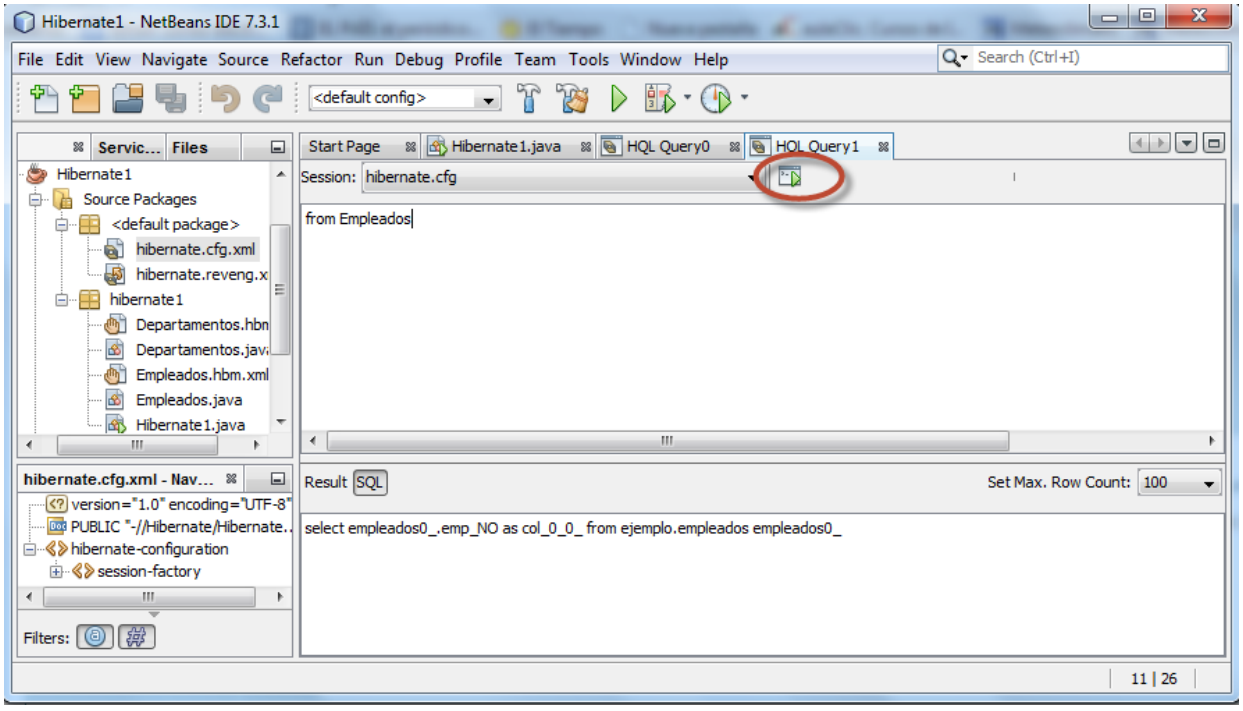
HQL (Hibernate Query Language) es el lenguaje incorporado a Hibernate para consultar los objetos de nuestro mapa.

Verás que la sintaxis HQL recuerda mucho el SQL, pero tiene diferencia, hay que pensar que ahora no estamos tratando tablas sino colecciones de objetos.

En Netbeans abrimos el editor de HQL haciendo clic derecho sobre el archivo de configuración de Hibernate y eligiendo la opción *Run HQL Query*.



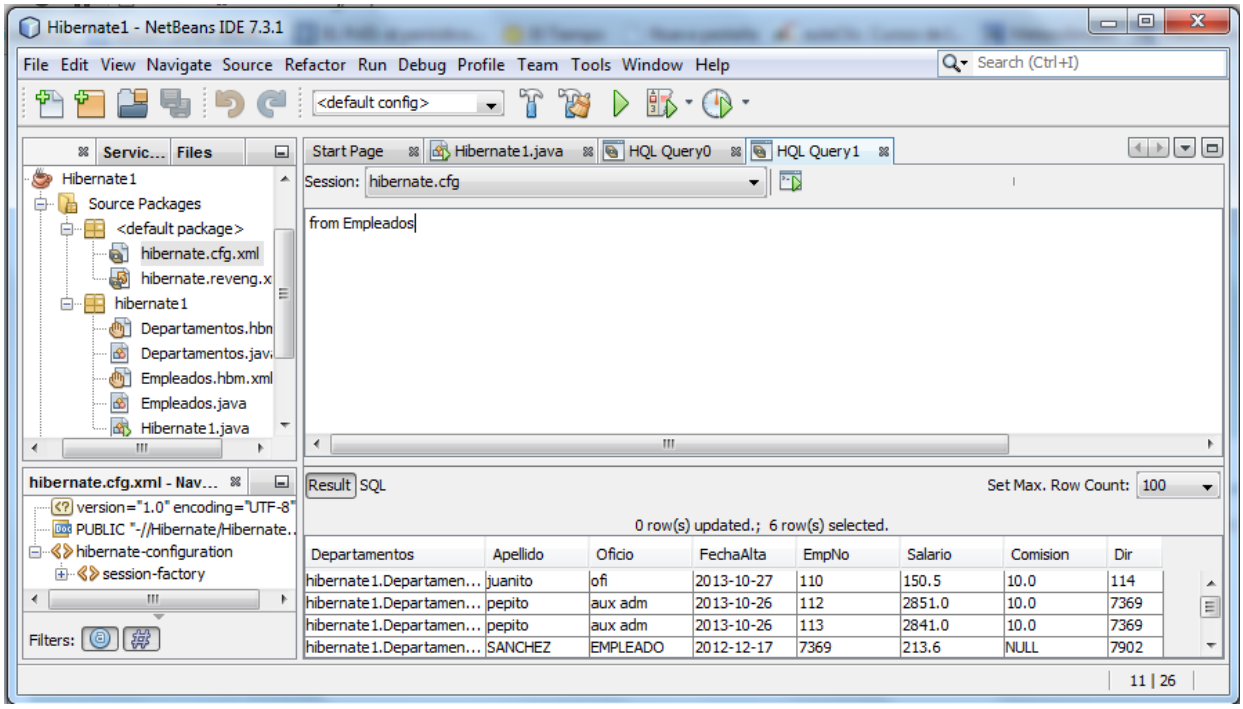
En la pestaña Session escribimos la sentencia HQL y con el icono señalado en la imagen la ejecutamos:



Al ejecutar, en la parte inferior se abre la pestaña *Result* donde se muestra el resultado de la consulta, como puedes ver en la imagen posterior.

Fíjate que en la columna Departamentos no se ve el código del departamento sino una referencia al objeto departamentos contenido en el empleado.

Lo que vemos y parecen filas realmente son instancias de objeto empleados.



A la hora de escribir código HQL tenemos que tener en cuenta que HQL no es sensible a las mayúsculas en las palabras reservadas, pero todo lo que se refiera a las clases definidas, son clases java y entonces será sensible a las mayúsculas.

Puedes escribir from Empleados o FROM Empleados, pero si la clase se llama Empleados no podrás escribir empleados.

Tampoco se puede utilizar el * como símbolo de "Todas las columnas".

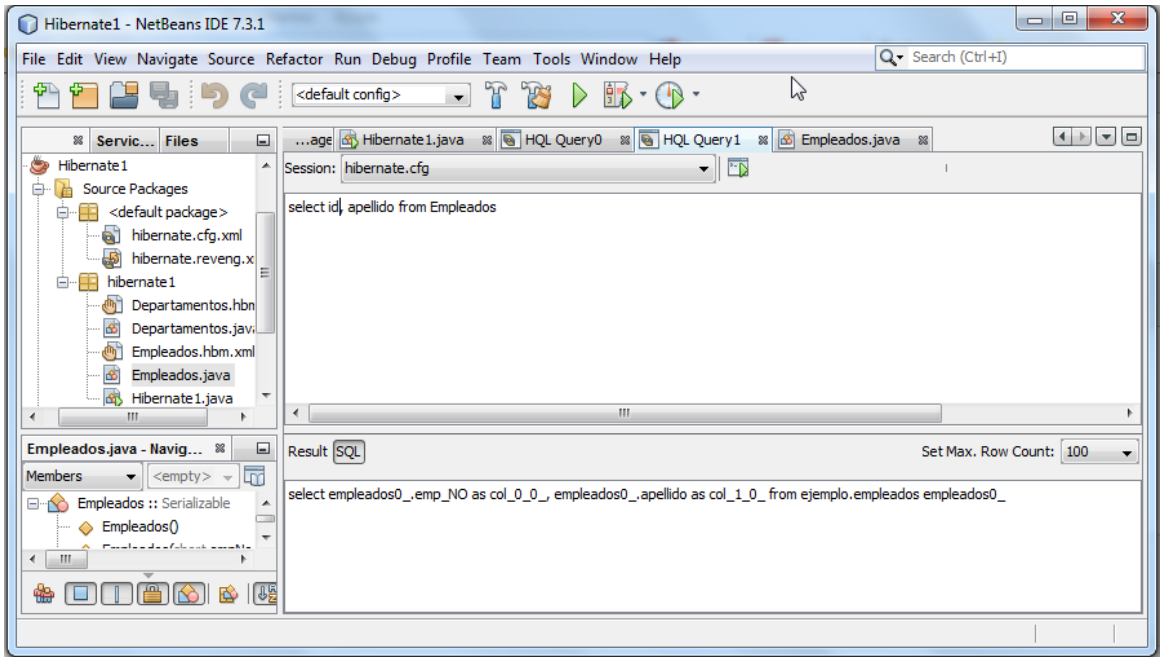
En este enlace tienes la referencia al lenguaje HQL:
http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html

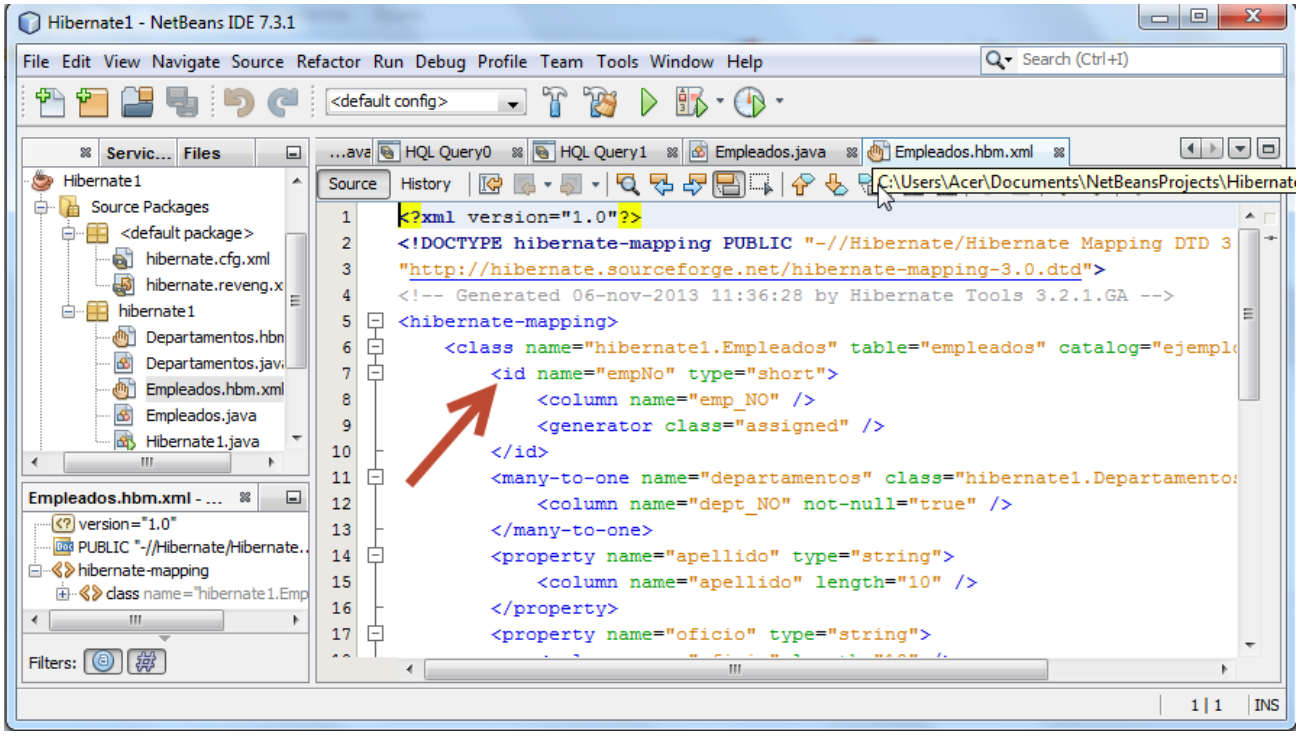
Un elemento interesante del lenguaje es la palabra id, se refiere al identificador de la tabla, su clave principal pero sin necesidad de indicar el nombre de la columna:

Nota. Si acabas de crear el proyecto con el mapa y quieres ejecutar HQL directamente, antes de poder ejecutar cualquier HQL, el proyecto tiene que estar compilado, opción Build sobre el proyecto.

La cláusula más simple que existe en Hibernate es **from**, que obtiene todas las instancias de una clase. Por ejemplo: from Empleados obtiene todas las instancias de la clase Empleados (equivalente a obtener todas las filas de la tabla EMPLEADOS).

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





Además de lo indicado anteriormente, aquí tienes un breve resumen del lenguaje que es mucho más extenso, pero nos limitaremos a estos conceptos:

La cláusula **order by** ordena los resultados de la consulta.

La cláusula **where** permite refinar la lista de instancias retornadas.
Por ejemplo:

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

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

```
from Empleados em.
```

Nota importante. Si utilizamos un alias, deberemos cualificar **siempre** las propiedades cuando queramos referirnos a ellas.
Por ejemplo: select apellido, salario from Empleados emp da error
select emp.apellido, emp.salario from Empleados emp es correcto

Pueden aparecer múltiples clases a la derecha de FROM, lo que causa un **producto cartesiano** / cross join:

from Empleados os em, Departamentos as dep.

Las consultas pueden **devolver múltiples objetos** y/o propiedades como un array de tipo Object[], una lista, un map o una clase.

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

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

Se puede utilizar alias para nombrar los atributos y expresiones.

Se pueden utilizar operadores aritméticos, de concatenación y funciones SQL reconocidas en la cláusula SELECT.
Ejemplos:

select avg(salario) as SalarioMedio, count(empNo) as NumEMpleados from Empleados
select avg(salario) + sum(salario)/ count(empNo) from Empleados
select count (distinct deptNo) from Empleados

Las expresiones utilizadas en la cláusula where pueden incluir:

operadores matemáticos: + , -, *, /

operadores de comparación: =, >=, <=, <> , !=, like

operadores lógicos: and, or, not

Paréntesis ( ) que indican agrupación

in, not in, between, is null, is not null, is empty, is not empty, member of, not member of
case ... when ... then ... else ... end, case when... then ... else ... end

concatenación de cadenas ... ... o concat(...,...)

current_date(), current_time(), current timestamp(), second(...), minute (...), hour (...), day (...), month (...), y year (...).

Cualquier función u operador definido por EJB-QL 3.0: substring(...) trim(), lower(), upper(), length(), locate(), abs() sqrt(), bit_length(), mod().

coalesce(), nullif().

str() para convertir valores numéricos o temporales a una cadena legible.

cast(... as ...) donde el segundo argumento es el nombre de un tipo de Hibernate.

extract(.. . from . ..) Estas dos últimas funciones solo se pueden usar si las soporta el SGBD subyacente.

lndex() se aplica a alias de una colección indexada unida.

Las funciones de HQL que tomen expresiones de ruta valuadas en colecciones: size(), minelement(), maxelement(), minindex(), maxindex(), junto con las funciones especiales elements() e indices, las cuales se pueden cuantificar utilizando some, all, exists, any, in.

Cualquier función escalar SQL soportada por la base de datos como trunc(), rtrim(), sin()...

parámetros posicionales JDBC: ?

parámetros con nombre: : name, : start_date y :x1

literales SQL 'algo', 56, 6.666+2,'1970-01-01 10:00:01.0'

constantes Jaya.

Se pueden definir consultas de resumen usando **group by** y **having**. Las funciones SQL y las funciones de agregado permitidas en las cláusulas order by y having son las permitidas en la bd subyacente. Ni la cláusula group by ni order by pueden contener expresiones aritméticas.
Ejemplos:

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

Para bases de datos que soportan subconsultas, Hibernate soporta **subconsultas**. Una subconsulta se debe encerrar entre paréntesis. Incluso se permiten subconsultas correlacionadas (la from de la subconsulta contiene una tabla de la consulta externa):
Ejemplos:

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

Cuando existen asociaciones entre clases podemos utilizar **joins** entre ellas. Los tipos de joins soportados son **inner join**, **left outer join** (left join), **right outer join** (right join), y **full join**.

La forma de utilizarlos es diferente a la usada en SQL (no olvidemos que no tratamos con tablas sino con objetos y colecciones de objetos).

El siguiente ejemplo 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 Departamento y un objeto Empleado:

```
from Departamentos as dep left outer join dep.enlace
```

En estos joins no es necesario especificar en la cláusula FROM todas las instancias (tablas) que se combinan, solo hay que hacer el join con el atributo enlace que es donde se define la asociación.

A la derecha de join se puede poner la cláusula **fetch**, entonces el resultado es diferente, no obtenemos dos objetos por cada fila de lo que sería la fila del resultado sino que un solo objeto que contiene en su interior una colección de los objetos relacionados.

La siguiente orden obtiene tantas instancias como resulte de combinar las tablas empleados y departamentos, incluyendo aquellos departamentos que no tengan empleados, pero solo del objeto Departamentos con los datos de sus empleados cargados en la colección (atributo enlace):

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

La salida de un join se puede ordenar:

```
from Departamentos as dep left outer join fetch dep.enlace order by dep.deptNo
```

Referencias:

http://www.cloudsopedia.com/tutorial/hibernate/index.php
http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html
Libro "Acceso a Datos" Alicia y Mª Jesús Ramos Martín Ed. Garceta