INTRODUCCIÓN A HIBERNATE[1]

Septiembre 2003

Traducción: Abril 2004

Francesc Rosés Albiol

Hibernate es un entorno de trabajo que tiene como objetivo facilitar la persistencia de objetos Java en bases de datos relacionales y al mismo tiempo la consulta de estas bases de datos para obtener objetos.

El objetivo de este artículo es familiarizar al lector con la filosofía utilizada en Hibernate y conseguir que sea capaz de utilizar Hibernate para desarrollar alguna aplicación. Hay, pues, un doble objetivo: "Filosófico" (observad las comillas) y práctico, Ambos objetivos van unidos. No pretendo entrar en la arquitectura de detalle usada por Hibernate ni en una discusión teórica. Partiré de mi experiencia práctica como desarrollador para ilustrar (y justificar, en parte) la filosofía de Hibernate. Y si parto de la experiencia como desarrollador. He de proporcionar los medios para que un desarrollador que lea este artículo pueda, después de hacerlo, usar Hibernate. Es, pues, un objetivo de este artículo conseguir que el lector sea capaz de escribir pequeñas aplicaciones de ABM (Altas, Bajas y Modificaciones).

Las posibilidades que ofrece Hibenate exceden con mucho las pretensiones de este artículo, así que me centraré sólo en las que cubren la casuística más habitual.

HIBERNATE: UN CASAMENTERO

Cualquier persona que se dedique al desarrollo objetual se encontrará con la problemática de implementar una persistencia de objetos en un soporte relacional.

El problema es que hay un divorcio total entre la estructura relacional y la objetual. Esto hace que el desarrollador opte con frecuencia por soluciones de compromiso que le hacen perder de vista el mundo objetual y la aplicación acaba siendo un batiburrillo infumable con una pseudoarquitectura objetual que hace daño a la vista.

Pongamos un ejemplo. Intentemos diseñar una base de datos para una escuela. Por una parte, tendríamos la lista de personas que pertenecen a la escuela. Cada persona tiene una sola categoría: profesor, alumno, personal de administración, etc. En la escuela, hay niveles: primero de EGB, segundo de EGB, etc. Además, hay clases. Cada clase pertenece a un nivel y tiene un nombre. Así, es posible que haya tres classes de primero de EGB: A, B y C. Pero una clase no es una clase sin alumnos y profesores. Así que tendremos que asignar personas a una clase. No entraremos aquí en un diseño minucioso ni nos preocupará si una persona está en una o más clases.

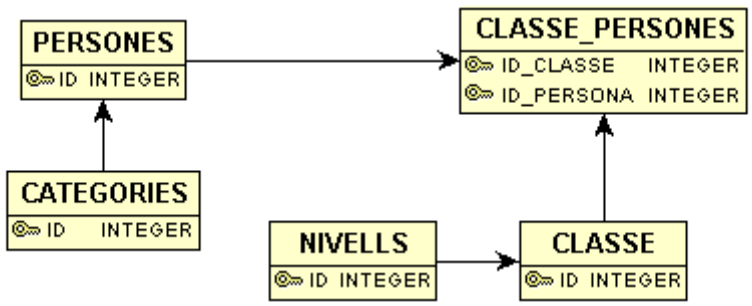
Así, podríamos pensar en una definición de tablas similar a esta:

```
CREATE TABLE CATEGORIAS
(
  ID INTEGER IDENTITY,
  CATEGORIA VARCHAR(50) NOT NULL
);
```

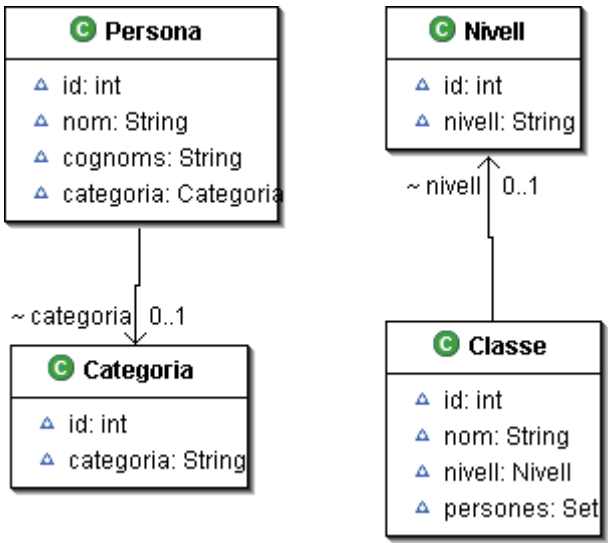
1 El presente artículo es una traducción del original en catalán escrito durante el mes de septiembre de 2003. En esta versión, no se han traducido los contenidos de las imágenes y el código fuente para castellano aún no está disponible. Debe considerarse, pues, esta versión como *draft*.

```
CREATE TABLE PERSONAS
(
  ID INTEGER IDENTITY,
  NOMBRE VARCHAR(30) NOT NULL,
  APELLIDOS VARCHAR(100) NOT NULL,
  ID_CATEGORIA INTEGER NOT NULL
);
ALTER TABLE PERSONAS ADD CONSTRAINT FK_P2C
  FOREIGN KEY ( ID_CATEGORIA )
   REFERENCES CATEGORIAS ( ID );
CREATE TABLE NIVELES
(
  ID INTEGER IDENTITY,
  NIVEL VARCHAR(30) NOT NULL
);
CREATE TABLE CLASE
(
  ID INTEGER IDENTITY,
  NOMBRE VARCHAR(20) NOT NULL,
  ID_NIVEL INTEGER NOT NULL
);
ALTER TABLE CLASE ADD CONSTRAINT FK_CLASE_NIVEL
  FOREIGN KEY ( ID_NIVEL )
   REFERENCES NIVELES ( ID );
CREATE TABLE CLASSE_PERSONAS
(
  ID_CLASE INTEGER NOT NULL,
  ID_PERSONA INTEGER NOT NULL,
  PRIMARY KEY (ID_CLASE,ID_PERSONA)
);
ALTER TABLE CLASE_PERSONAS ADD CONSTRAINT FK_CP_C
  FOREIGN KEY ( ID_CLASSE )
   REFERENCES CLASE ( ID );
ALTER TABLE CLASE_PERSONAS ADD CONSTRAINT FK_CP_P
  FOREIGN KEY ( ID_PERSONA )
   REFERENCES PERSONAS ( ID );
```

El siguiente gráfico muestra las relaciones entre tablas:



Pero el diseño objetual con el que nos gustaría trabajar no es un calco del diseño relacional. La siguiente figura ilustra nuestro diseño objetual con un diagrama de clases UML.



Observemos algunos cambios en nuestro diagrama de clases:

1. Los nombres de los objetos están en singular, mientras que los de las tablas están en plural. Así, entendemos que una instancia de un objeto se corresponde con un registro de una tabla. Una tabla es un conjunto de `Personas`. La clase `Persona` representa una sola persona.
2. La clase `Persona` no tiene un entero que apunte a la tabla `Categorias`. Tiene un objeto de tipo `Categoria`. Así, vemos que los tipos de las propiedades de un objeto no siempre se corresponden con los tipis de las columnas de las tablas. Un caso similar, lo encontraríamos en el objeto `Clase`. La propiedad correspondiente a la columna `ID_NIVEL INTEGER` es del tipo `Nivel`, no `int`.
3. En nuestro diseño del modelo relacional, tenemos la table `CLASE_PERSONAS` que es la encargada de almacenar las personas asociadas a cada clase. Desde el punto de vista objetual, el planteamiento es distinto. Cada clase tiene un conjunto, `Set`, de personas. No necesitamos crear un objeto puente que relacione clases y personas.

Conclusión: el modelo relacional y el objectual no acaban de casar.

Ahora bien, está claro que tendremos que hacer casar los dos modelos si queremos trabajar con comodidad; y este es el papel de Hibernate: hacer de casamentero. Hibernate se encarga de casar los dos modelos de manera que nosotros trabajemos desde Java como corresponde. Usando el modelo objetual.

Sin Hibernate, para añadir un registro a la tabla `Categorias` tendríamos que escribir algo similar a esto:[2]

```
[...]
Class.forName(“org.hsqldb.jdbcDriver”);
String url = “jdbc:hsqldb:./Databases/ESCUELA”;
Connection connection = DriverManager.getConnection(url, “sa”, “”);
String ins = “INSERT INTO CATEGORIAS VALUES(NULL, 'Profesor')”;
Statement stmt = null;
stmt = connection.createStatement();
stmt.executeUpdate(ins);
[...]
```

Con Hibernate, escribiríamos algo similar a esto:

```
[...]
```

2 Para simplificar, obvio el control de excepciones.

```
Configuration conf = new Configuration();
conf.addClass(Categoria.class);
SessionFactory sessionFactory = conf.buildSessionFactory();
Session session = sessionFactory.openSession();
Categoria categ = new Categoria();
categ.setCategoria(“Profesor”);
session.save(categ);
session.flush();
session.close();
[...]
```

Y si nos interesase añadir otra:

```
[...]
session = sessionFactory.openSession();
categ = new Categoria();
categ.setCategoria(“Alumno”);
session.save(categ);
session.flush();
session.close();
```

¿Cuál es la gran diferencia entre el código tradicional JDBC y el código que escribimos con Hibernate? “Ha desparecido el SQL”. Lo único que digo es que quiero “guardar” (*save*) un objecto.

Si lo que queremos es ejecutar una consulta a la base de datos, tendremos que utilizar un lenguaje de interrogación que “case” con el mundo objetual. Pedir una lista ordenada de profesores nos costaría tan poco como esto:

```
[...]
session = sessionFactory.openSession();
String qry = “FROM escuela.beans.Persona AS P
              WHERE P.categoria.categoria = 'Profesor'
              ORDER BY P.apellidos, P.nombre”;
List profes = session.find(qry);
session.close();
for (int i = 0; i < profes.size(); i++) {
  Persona p = (Persona)profes.get(i);
  System.out.println(p.getNombre() + “ “ + p.getApellidos() +
                     p.getCategoria().getCategoria());
}
```

Observamos aquí que la consulta se formula en base a objetos y que, una vez más, “ha desaparecido el SQL”. Del SQL, se ocupa Hibernate. Él casa nuestro mundo objectual con el relacional. Pedimos objetos y expresamos las condiciones de la consulta objetualmente. Hibernate se encarga de transformar nuestra consulta al dialecto SQL[3] que toque.

3 Hibernate soporta los siguientes dialectos SQL:
- DB2
- MySQL
- SAP DB
- Oracle (qualsevol versió)
- Oracle 9
- Sybase
- Sybase Anywhere
- Progress
- Mckoi SQL
- Interbase
- Pointbase
- PostgreSQL
- HypersonicSQL (HSQLDB)
- Microsoft SQL Server
- Ingres

¿QUÉ HAY QUE PRESENTAR PARA CASARSE?

Como hemos visto, Hibernate casa el mundo objetual con el relacional. Ahora bien, para casarse hay que presentar una serie de documentación que es la que posibilita la boda.

¿Qué ha de saber Hibernate para casar los dos modelos?

1. **Quién está detrás del modelo relacional**:
    1. Qué gestor de bases de datos está detrás
    2. A qué base de datos me conecto
    3. Cómo me conecto
    4. Etc.
2. **Cómo se emparejan propiedades y campos de tablas**:
    1. Clave primaria:
        1. ¿Cuál es la propiedad que se corresponde con la clave primaria de la tabla correspondiente?
        2. ¿Qué método deberé utilizar para generar un nuevo valor de la clave primaria?
        3. Etc.
    2. Otras propiedades:
        1. Cómo empareja una propiedad con un campo de una tabla de la base de datos
            1. ¿Cómo se llama la columna correspondiente?
            2. ¿Qué tipo tiene?
            3. ¿Es obligatoria?
            4. Etc.
    3. Cómo gestiona las relaciones entre tablas
        1. ¿Es una relació “uno-a-uno”,
        2. “uno-a-muchos”,
        3. “muchos-a-muchos”?, etc.

Para responder a las dos grandes preguntas, se utilizan dos archivos distintos:

1. El **archivo de propiedades de Hibernate** (`Hibernate.properties`) es el encargado de determinar los aspectos relacionados con el gestor de bases de datos y las conexiones con él.
2. Los **archivos que que definen el emparejamiento** (*mapping*) de propiedades con tablas y columnas (`*.hbm.xml`)

Una boda sencilla

Para no complicar las cosas de entrada, vamos a plantear un archivo de propiedades de Hibernate sencillo que nos permita interactuar con HSQLDB y un archivo de emparejamiento simple entre la clase `Categoria` y la tabla `Categorias`.

- Informix
- FrontBase

Para que tenga lugar la boda, hay que presentar dos documentos:

- **`Hibernate.properties`**: En el que se dice qué gestor de bases de datos usaremos y a qué base de datos nos conectaremos y cómo lo haremos.
- **`NombreDeClasse.hbm.xml`**: En el que se describe cómo se relacionan clases y tablas y propiedades y columnas.

A continuación, los describiremos con más detalle.

El archivo de propiedades de Hibernate: "Hibernate.properties"

En este archivo, se pueden especificar muchísimas cosas. Nos limitaremos, por ahora, a proporcionar los datos necesarios para que Hibernate se pueda conectar a nuestra base de datos HSQLDB. Este sería, pues, el mínimo necesario para funcionar:

```
## HypersonicSQL
hibernate.dialect net.sf.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class org.hsqldb.jdbcDriver
hibernate.connection.username sa
hibernate.connection.password
hibernate.connection.url jdbc:hsqldb:hsql://localhost
```

La configuración es sencilla. Especificamos el dialecto SQL y los datos necesarios para poder establecer una conexión con la base de datos vía JDBC (Driver, URL, usuario y contraseña).

El archivo de emparejamiento (mapping): "Categoria.hbm.xml"

Este es el archivo que casa los mundos relacional y objetual. El ejemplo, sencillísimo, que presentamos aquí es el que se corresponde con la clase `Categoria`.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
  <class name="escuela.beans.Categoria" table="categorias">
    <id name="id" type="integer" column="ID" unsaved-value="-1">
      <generator class="identity"/>
    </id>

    <property name="categoria" column="CATEGORIA"
              type="string"
              unique=”true”
              not-null="true"/>
  </class>
</hibernate-mapping>
```

Como vemos, para definir los emparejamientos se usa un documento XML.

Cabecera XML

Éste tiene una cabecera fija, típica de todos los documentos XML que no describiremos.

El elemento <hibernate-mapping>

Después de la cabecera, entramos en la parte de descripción de los emparejamientos, delimitada por el elemento **`<hibernate-mapping>`**. Este elemento permite especificar, mediante atributos, diversas características como el esquema de la base de datos. No nos entretendremos en describirlo.

El elemento <class>

Como estamos emparejando una clase con una table, hemos de especificar cuáles son. Lo hacemos usando el elemento **`<class>`**:

```
<class name="escuela.beans.Categoria" table="categorias">
```

Indicamos el nombre de la clase (**`name`**`=escuela.beans.Categoria`), y el nombre de la tabla con la que se empareja (**`table`**`=”categorias”`).

El elemento <id>

Una vez emparejados objeto y tabla, hay que emparejar las propiedades del objeto con las columnas de la tabla. En nuestro ejemplo, observamos que para describir este emparejamiento se utilizan dos elementos XML distintos: `<id>` y `<property>`.

El primero, **`<id>`**, empareja una de las propiedades con la clave primaria de la tabla:

```
<id name="id" type="integer" column="ID" unsaved-value="-1">
```

Identificamos la propiedad que se empareja con la clave primaria mediante el atributo XML **`name`**. En nuestro caso, la propiedad a relacionar con la clave primaria es `id`.

Esta propiedad se empareja con la columna `ID` de la tabla: **`column`**`=”ID”`. Como el nombre de la propiedad y el de la columna coinciden, en este caso, nos podríamos ahorrar el atributo `column`.

También, y opcionalmente, especificamos el tipo de datos con el que estamos trabajando. En este caso, con enteros. Como es un tipo básico, no necesitaríamos usar el atributo **`type`**. Hibernate sabe deducir el tipo de datos que estamos utilizando a partir de la introspección. Hibernate analiza en tiempo de ejecución las clases Java y obtiene la información que necesita. Aquí, a guisa de ejemplo, especificamos el tipo de datos Hibernate: `type=”integer”`.[4]

En nuestro ejemplo, hemos vuelto a utilizar un atributo opcional: **`unsaved-value`**. Este atributo permite especificar qué valor tiene el campo clave cuando aún no ha sido guardado en la base de datos. En nuestro caso, `-1`. Así, de entrada, parece un atributo poco útil, pero no es así. La classe `Session` de Hibernate dispone de un método muy práctico: `saveOrUpdate(Object)`. Este

4 Hibernate utiliza los siguientes tipos:
- `integer`, `long`, `short`, `float`, `double`, `character`, `byte`
- `boolean`, `yes_no`, `true_false`: alternativas para `boolean` o `java.lang.Boolean`.
- `string`: empareja `java.lang.String` y `VARCHAR`.
- `date`, `time`, `timestamp`: empareja `java.util.Date` y sus subclases con los tipos SQL `DATE`, `TIME` y `TIMESTAMP` (o equivalente).
- `calendar`, `calendar_date`: emparejan java.util.Calendar con SQL `TIMESTAMP` y `DATE`.
- `big_decimal`: empareja `java.math.BigDecimal` con `NUMERIC` (o `NUMBER` de Oracle).
- `locale`, `timezone`, `currency`: empareja `java.util.Locale`, `java.util.TimeZone` y `java.util.Currency` con `VARCHAR` (o `VARCHAR2` de Oracle)
- `class`: empareja `java.lang.Class` con `VARCHAR` (o `VARCHAR2` de Oracle)
- `binary`: empareja matrices (*arrays*) de bytes con un tipo SQL binario adecuado
- `serializable`: empareja tipos Java serializables con un tipo SQL binario adecuado
- `clob`, `blob`: emparejamientos para las clases JDBC `java.sql.Clob` y `java.sql.Blob`

método distingue si un objeto debe ser guardado por primera vez en la base de datos o bien modificado porque ya está en ella, en función de lo que especifiquemos en el atributo `unsaved-value`. En nuestro caso, la propiedad `id` de una `Categoria` que aún no haya sido guardada en la base de datos tendrá un valor -1. Esto indicará a Hibernate que hay que guardarla; y al hacerlo, la propiedad `id` de esta `Categoria` pasará a tener un valor diferente de `-1`, por ejemplo, `25`. Si recuperamos de la base de datos esta categoría, tendremos un objeto `Categoria` con un `id` igual a `25`. Si modificamos la propiedad `categoria` y después utilizamos el método `saveOrUpdate()`, Hibernate comprobará que la propiedad `id` no vale `-1` y deducirá que en vez de guardar, ha de modificar.

El atributo `unsaved-value` puede tener los siguientes valores predefinidos: `any`, `none` y `null`. Por defecto, `null`.

El subelemento <generator>

Este elemento nos permite definir cómo se generan los valores de las claves primarias. Hibernate nos ofrece diversos métodos de generación de valores para las claves primarias.

```
<generator class="identity"/>
```

El método se especifica mediante el atributo **`class`**. En nuestro ejemplo, usamos el método **`identity`**, ya que HSQLDB soporta claves primarias de tipo **IDENTITY**.[5]

Hay más muchos métodos predefinidos y, si fuera necesario, los podríamos definir nosotros. Algunos, como `identity`, generan claves directamente y otros necesitan la ayuda de tablas auxiliares para generarlas. Sería demasiado largo comentarlos todos, así que nos centraremos en un par de métodos.[6]

El método **`native`** es bastante interesante. Deja que Hibernate escoja entre los métodos `identity`, `sequence` o `hilo`,[7] en función de las características del gestor de bases de datos con el que trabajemos.

El método **`assigned`** deja que sea nuestra aplicación la que asigne un identificador al objeto antes de guardarlo en la base de datos.

El elemento <property>

Sirve para emparejar aquellas propiedades que no forman parte de la clave primaria con las correspondientes columnas de una (o más) tablas.

Aquí tenemos el emparejamiento propuesto para la propiedad `categoria` de nuestro ejemplo.

```
<property name="categoria" column="CATEGORIA"
          type="string"
          unique="true"
          not-null="true"/>
```

Como en el caso del elemento `id`, el atributo **`name`** indica el nombre de la propiedad, el atributo **`column`**, el nombre de la columna de la tabla `categorias` y el atributo **`type`** nos indica el tipo de Hibernate.

El atributo **`unique`**, opcional, nos permite especificar si se admiten valores duplicados o no. En

5 También las soportan los siguientes gestores: DB2, MySQL, MS SQL Server y Sybase.

6 La documentación de Hibernate comenta extensamente cada uno de los métodos de generación de valores para claves primarias.

7 El método `sequence` usa una *sequence* en DB2, PostgeSQL, Oracle, SAP DB, McKoi o Interbase. El método `hilo` necesita una tabla auxiliar para generar las claves.

nuestro caso, no.

El atributo opcional **`not-null`**, nos permite especificar la obligación de que la propiedad tenga un valor no nulo.

VIDA MATRIMONIAL

Bien, una vez casados, veamos qué hay que hacer...

Clases persistentes – Java Beans

Para que una clase pueda beneficiarse de las ventajas de Hibernate, es necesario que cumpla con una única condición de los Java Beans: Las propiedades objeto de persistencia deben ser accesibles mediante métodos `get` y `set`. Sencillísimo.

Así, la clase `Categoria` quedaria més o menys como sigue:

```
/*
 * Creada el 24/09/2003
 * (c) Francesc Rosés
 */
package escuela.beans;

/**
 * Categoría de personal.
 * @author Francesc Rosés
 */
public class Categoria {
  /** Identificador único */
  int id;


  /**
   * Devuelve el contenido de la propiedad <code>categoria</code>.
   * @return Categoria
   */
  public String getCategoria() {
    return categoria;
  }

  /**
   * Devuelve el contenido de la propiedad <code>id</code>.
   * @return Id
   */
  public int getId() {
    return id;
  }

  /**
   * Asigna un nuevo valor a la propiedad <code>categoria</code>.
   * @param nuevaCategoria
   */
  public void setCategoria(String nuevaCategoria) {
    categoria = nuevaCategoria;
  }

  /**
   * Asigna un nuevo valor a la propiedad <code>id</code>.
   * @param nuevoId
```

```
   */
  public void setId(int nuevoId) {
    id = nuevoId;
  }

  /**
   * Devuelve una tira que representa el objecto. En este caso,
   * concatena las propiedades <code>id</code> y
   * <code>categoria</code>.
   * @return Una representación textual del objecto.
   */
  public String toString() {
    return getId() + " " + getCategoria();
  }
}
```

Altas, bajas, modificacions (ABM) y consultas

En cualquier aplicación que use Hibernate aparecen cuatro objetos básicos:

1. **`Configuration:`** es el objeto que contiene la información necesaria para conectarse a la base de datos. Es el encargado de leerse el archivo `Hibernate.properties`.
   También es el encargado de procesar la información correspondiente a los aparejamientos. Es el encargado de leerse y verificar los archivos de emparejamiento `nombreDeClasse.hbm.xml`.
   ```
   Configuration conf = new Configuration();
   conf.addClass(escuela.beans.Categoria.class);
   ```
2. **`SessionFactory:`** es una fábrica de `Sessions`. Un objeto `Configuration` es capaz de crear una `SessionFactory` ya que tiene tota la información necesaria.
   Normalmente, una aplicación sólo tiene una `SessionFactory`.
   ```
   SessionFactory sessionFactory = conf.buildSessionFactory();
   ```
3. **`Session:`** La principal interficie entre la aplicación Java e Hibernate. Es la que mantiene las conversaciones entre la aplicación y la base de datos. Permite añadir, modificar y borrar objetos en la base de datos.
   ```
   Session session = sessionFactory.openSession();
   ```
4. **`Transaction:`** Como su nombre indica, se encarga de la transaccionalidad. Permite definir unidades de trabajo.
   ```
   Transaction tx = session.beginTransaction();
   [...]session.save(unaCategoria);
   tx.commit();
   [...]
   tx.rollback();
   ```

Un posible patrón de aplicación

A continuación, expondré un esquema (discutible, sin duda) que es el que acostumbro a usar en las aplicaciones Hibernate.

Primero, defino las 3 propiedades básicas y las asigno a `null`:

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

Después, defino los métodos de acceso a estas propiedades siguiendo siempre el mismo patrón: si la propiedad es nula, la creo y la devuelvo. Si no es nula, simplemente la devuelvo:

```
/**
```

```
     * [1] Construye y/o devuelve una Configuration.
     */
    public Configuration getConfiguration() {
      if (configuration == null) {
        // [1.1] Aquí lee el archivo de propiedades de Hibernate
        configuration = new Configuration();

        // [1.2] Aquí lee el archivo de emparejamientos Categoria.hbm.xml
        try {
          configuration.addClass(escola.beans.Categoria.class);
        } catch (MappingException e) {
            // Actuamos en consecuencia
          }
      }
      return configuration;
    }

    /**
     * [2] Construye y/o devuelve una SessionFactory.
     */
    public SessionFactory getSessionFactory() {
      if (sessionFactory == null) {
        try {
          sessionFactory = getConfiguration().buildSessionFactory();
        } catch (HibernateException e) {
            // Actuamos en consecuencia
          }
      }
      return sessionFactory;
    }

    /**
     * [3] Construye y/o devuelve una Session.
     */
    public Session getSession() {
      // [3.1] La crea si es nula o está cerrada
      if (session == null || !session.isOpen()) {
        try {
          session = getSessionFactory().openSession();
        } catch (HibernateException e) {
            // Actuem en conseqüència
          }
      }
      return session;
    }
```

[1] Lo primero que hacemos es crear una instancia de `Configuration`. Esta clase, como hemos visto, es la encargada de procesar el archivo de propiedades, `hibernate.properties` y el de emparejamiento, `nombreDeClasse.hbm.xml`.

[1.1] En el momento de crearla, Hibernate va a buscar el archivo `Hibernate.properties` al directorio padre del *package* en el que se encuentra nuestra clase. Así, si nuestra clase se encuentra en el *package* `escuela.abm`, y la estructura de directorios es:

```
C:\HIBERNATE_ARTICLE\escuela\abm\CategoriaABM.java
```

El archivo `hibernate.properties` tiene que estar en

```
C:\HIBERNATE_ARTICULO
```

[1.2] Una vez creada la instancia de `Configuration`, le pasamos la información sobre el emparejamiento:

```
configuration.addClass(escuela.beans.Categoria.class);
```

Esto hace que Hibernate vaya a buscar el archivo de emparejamiento en el mismo directorio en el que se encuentra la clase:

```
C:\HIBERNATE_ARTICLE\escuela\beans\Categoria.hbm.xml
```

[2] En estos momentos, Hibernate ya sabe cómo conectarse a la base de datos que nos interessa y cómo se emparejan propiedades y columnas. Pero, como hemos visto más arriba, para comunicarnos con Hibernate desde nuestra aplicación, necesitamos una `Session`, y para crearla, necesitamos una fábrica de `Session`s. Una `SessionFactory`:

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

[3] Una vez tenemos la fábrica, ya podemos crear `Session`s:

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

Altas

Añadir una nueva `Categoria` a la base de datos es sencillo:

```
public void anadir(String categ) {
  // [1] Construimos una nueva Categoria
  Categoria categoria = new Categoria();
  categoria.setCategoria(categ);

  // [2] Creamos una Session
  Session sesion = getSession();
  if (sesion == null) {
    System.out.println("Error abriendo sesión.");
    return;
  }

  Transaction tx = null;
  try {
    // [3] Iniciamos una unidad de trabajo
    tx = sessio.beginTransaction();
    // [4] Guardamos la nueva Categoria en la base de datos
    sesion.save(categoria);
    // [5] Confirmamos la transacción
    tx.commit();
  } catch (HibernateException e) {
      if (tx != null) {
        try {
          [6] Tiramos para atrás la transacción
          tx.rollback();
        } catch (HibernateException e) {
          }
      }
    }
    finally {
      try {
        [7] Cerramos la sesión
        sessio.close();
      } catch (HibernateException e) {
        }
    }
}   // afegir()
```

[1] Para añadir una nueva `Categoria`, primero hay que crearla. En nuestro ejemplo, la creamos a partir del parámetro `categ` del método.

[2] Hemos visto que la `Session` es el puente entre nuestra aplicación y Hibernate. Usamos el método `getSession()`, descrito más arriba, para obtener una `Session`.

[3] Iniciamos una unidad de trabajo antes de añadir la `Categoria`:[8]

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

[4] Guardamos `categoria` en la base de datos.

```
Sessio.save(categoria);
```

Observamos que no hay ninguna instrucción SQL. Simplemente, estamos persistiendo un objeto.

[5] Si no ha habido ningún problema, confirma la transacción a la base de datos, haciendo un `commit()` y cierra la unidad de trabajo. En definitiva, hace un `SQL COMMIT`, pero además, sincroniza la sesión con la base de datos ejecutando implícitamente el método `flush()` de la clase `Session`.

[6] Si ha habido algún problema, tira para atrás la transacción y cierra la unidad de trabajo. En definitiva, hace un `ROLLBACK`.

[7] Para acabar, pase lo que pase, cerramos la sesión.

Bajas

El esquema general para dar de baja una categoría es básicamente el mismo que el de darla de alta. Me centraré, pues, en la parte específica de darla de baja:

```
public void borrar(int idCategoria) {
  [...]
  Categoria unaCategoria = new Categoria();
  unaCategoria.setId(idCategoria);
  sesion.beginTransaction();
  sesion.delete(unaCategoria);
  sesion.commit();
  [...]
}
```

En el ejemplo anterior, creamos una `Categoria` y le asignamos el identificador que se nos pasa como parámetro, Este identificador es el que usará Hibernate para eliminar la `Categoria` de la base de datos.

Hay tres estrategias para borrar registros. Por ejemplo, podemos usar una consulta basándonos en el identificador de clave primaria:

```
public void borrar(int idCategoria) {
  [...]
  sesion.beginTransaction();
  Categoria unaCategoria = (Categoria)sesion.load(Categoria.class,
                                                  new Integer(id));
  sesion.delete(unaCategoria);
  sesion.commit();
  [...]
}
```

Fijémonos en el método `load()` de `Session`. Este método nos devuelve un objeto a partir de un

8 Podríamos no usar unidades de trabajo, pero creo que siempre es interesante su uso. Si no nos interesase usarlas, el código sería parecido a éste:

```
sesion.save(categoria);
sesion.flush();
sesion.close();
```

identificador de clave primaria, Como es lógico, le hemos de pasar la clase (`Categoria.class`) para que sepa en qué tabla ha de buscar. Recordemos que en el archivo de emparejamiento (`Categoria.hbm.xml`) hemos emparejado la clase `Categoria` con la tabla `CATEGORIAS`. Con esta información, Hibernate genera una instrucción SQL similar a esta:

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

Hibernate nos proporciona un lenguaje de consulta orientado a objetos bastante avanzado. Se llama HQL y hablaremos de él con más detalle, más adelante.

El método `delete()` admite una consulta en HQL como parámetro. En este caso, eliminaría todos los objetos devueltos por la consulta:

```
[...]
String sel = "FROM Categoria AS C WHERE C.id = " + idCategoria;
sesion.delete(sel);
[...]
```

En este caso, sólo borraría un registro. Si, por ejemplo, nos interesase borrar todos los registros que tuvieran "r" en el campo CATEGORIA, sólo tendríamos que cambiar la consulta:

```
[...]
String sel = "FROM Categoria AS C WHERE C.categoria LIKE "%r%";
sesion.delete(sel);
[...]
```

Fijémonos un poco en la sintaxis de la consulta. Se parece bastante a la sintaxi SQL habitual, pero está adaptada al mundo de los objetos:

- No hay `SELECT` ya que no devuelve valores de columnas, sino objetos.
- No especificamos ninguna tabla en el `FROM`. Especificamos una clase. Es a partir de esta clase y de las definiciones que hemos hecho en el archivo de emparejamientos que Hibernate deducirá qué tablas están implicadas.
- No usamos nombres de columnas en el `WHERE`. Usamos propiedades de objetos.

Modificaciones

La estrategia va a ser la misma que para las bajas:

```
public void cambiar(int idCategoria) {
  [...]
  sesion.beginTransaction();
  Categoria unaCategoria = (Categoria)sesion.load(Categoria.class,
                                                  new Integer(id));
// Cambiamos una propiedad del objeto
unaCategoria.setCategoria("Una nueva categoría");

// Lo modificamos en la base de datos
  sesion.update(unaCategoria);
  sesion.commit();
  [...]
}
```

¿Añadir o modificar?

Más arriba, hemos visto que el elemento `<id>` del archivo de emparejamientos (`Categoria.hbm.xml`) tiene un atributo que permite que Hibernate pueda decidir si hay que guardar un objeto o modificarlo: `unsaved-value`.

En nuestro ejemplo, hemos especificado que si la propiedad `id` de `Categoria` vale `-1`, hay que añadir el objeto y si no, hay que modificarlo en la base de datos.

```
public void anadirOCambiar(Categoria unaCategoria) {
  [...]
  sesion.beginTransaction();

// Lo modificamos o lo añadimos a la base de datos dependiendo del
// valor de la propiedad “id”
  sesion.saveOrUpdate(unaCategoria);
  sesion.commit();
  [...]
}
```

UNA BODA MÁS COMPLICADA

El ejemplo anterior era bastante sencillo. Es el caso típico del mantenimiento de tablas auxiliares del tipo `codigo` y `descripcion`. En nuestro modelo de datos, hay un caso prácticamente idéntico: el de la tabla `NIVELLS`.

Pero las cosas no son siempre tan sencillas. En nuestro modelo, encontramos un caso igualmente típico, pero un poc más complicado: el de la tabla `PERSONAS`. En esta tabla encontramos una referencia a la tabla `CATEGORIAS`. Toda persona tiene una categoría, y en el modelo relacional, lo señalamos con un identificador de categoría: la columna ID_CATEGORIA.

La representación objetual, sin embargo, no cuadra con la relacional. Las propiedades de la clase `Persona` serían:

```
public class Persona {
  /** Identificador de persona */
  int    id;
  /** Nombre de la persona */
  String nom;
  /** Apellidos de la persona */
  String apellidos;
  /** Categoría laboral de la persona */
  Categoria categoria;

  [...]
}
```

Aquí, la categoría no es un entero que apunta a la tabla de categorías, sino un objeto de tipo `Categoria`.

Para emparejar el objeto `Persona` con las tablas del mundo relacional, tenemos que describir esta relación en el archivo `Persona.hbm.xml`. Reproduzco a continuación un fragmento de dicho archivo:

```
<hibernate-mapping>
    <class name="escuela.beans.Persona" table="personas">
        <id name="id" unsaved-value="-1">
            <generator class="identity"/>
        </id>
        <property name="nombre"/>
        <property name="apellidos"/>
        <many-to-one name="categoria" column="CATEGORIA_ID"/>
    </class>
</hibernate-mapping>
```

Observem que excepto el emparejamiento de la propiedad `categoria`, todo es igual que en el ejemplo anterior.

Para expresar la relación de pa propiedad `categoria`, usamos el elemento `<many-to-one>`. Es decir, estamos describiendo una relación "**muchos-a-uno**".

Para hacerlo, decimos que la propiedad de `Persona` que participa en la relación `many-to-one` se llama `categoria` (`name="categoria"`). Con esto, Hibernate ya sabe que es de la clase `Categoria`, y que esta clase está ligada a la tabla `CATEGORIAS`. Lo que no sabe es qué campo de la tabla `PERSONAS` está emparejado con la clave primaria de la tabla `CATEGORIAS`. Por eso le decimos que la columna relacionada es `CATEGORIA_ID` (`column="CATEGORIA_ID"`).

Ahora, Hibernate ya dispone de todos los elementos para gestionar la persistencia de objetos de tipo `Persona`.[9]

Todo lo que hemos explicado más arriba sobre altas, bajas y modificaciones para objetos de tipo `Categoria` es aplicable a objetos de tipo `Persona`. Por ejemplo, para añadir una persona a a base de datos:

```
// [1] Obtenemos una categoría de la base de datos
Categoria categ = null;
categ = (Categoria)unaSesion.load(Categoria.class, new Integer(3));

// [2] Creamos una Persona
Persona p = new Persona();
p.setNombre("Un nombre");
p.setCognoms("Los dos apellidos");
p.setCategoria(categ);

// [3] Iniciamos una unidad de trabajo
Transaction tk = null;
tk = unaSesion.beginTransaction();

// [4] Guardamos el objeto Persona en la base de datos
unaSesion.save(p);

// [5] Cerramos la unidad de trabajo
tk.commit();
```

Más preguntas

Avancemos alguna cosa más del lenguaje de interrogación HQL, ahora que tenemos una relación "muchos a uno". Una de las consultas típicas a la tabla PERSONAS puede ser "*Quiero obtener todas las personas que sean profesores*".

Si utilizáramos SQL y la categoría de "profesor" tuviera un ID = 3, tendríamos:.

```
SELECT NOMBRE, APELLIDOS FROM PERSONAS WHERE CATEGORIA_ID = 3
```

O también:

```
SELECT NOMBRE, APELLIDOS FROM PERSONAS P, CATEGORIAS C
WHERE P.ID_CATEGORIA = C.ID AND C.CATEGORIA = 'Profesor'
```

En HQL, la cosa tendría esta pinta:

```
FROM Persona P WHERE P.categoria.id = 3
```

o bien:

9 En nuestro modelo relacional de ejemplo, la tabla `CLASES` también presenta una relación del tipo "muchos a uno" para el campo `ID_NIVEL`, que apunta al campo `ID` de la tabla `NIVELES`.

```
FROM Persona P WHERE P.categoria.categoria = 'Profesor'
```

Fijémonos en la lectura puramente objetual. El primer ejemplo se podría leer así: “Dame todas la `Personas` de las cuales su propiedad `categoria` tenga una propiedad `id` con valor `3`”. Y el segundo: “Dame todas las `Persona`s de las cuales su propiedad `categoria` tenga una propiedad `categoria` con valor `'Profesor'`”.

El primer caso es sencillo. Hibernate puede obtener respuestas de la misma tabla `PERSONAS` buscando aquellos registros con un valor 3 en la columna `ID_CATEGORIA`. Se correspondería con el rimer ejemplo de consulta SQL:

```
SELECT NOMBRE, APELLIDOS FROM PERSONAS WHERE CATEGORIA_ID = 3
```

El segundo es más complicado ya que el valor “Profesor” no está en la tabla `PERSONAS`, sino en la tabla `CATEGORIAS`. Hibernate tendrá que utilizar la información que tiene sobre las relaciones entre ambas tablas para obtener la información que le pedimos. Sin embargo, la formulación de la consulta en HQL es sencilla y clara- Esta consulta se correspondería con el segundo ejemplo de consulta SQL:

```
SELECT NOMBRE, APELLIDOS FROM PERSONAS P, CATEGORIAS C
WHERE P.ID_CATEGORIA = C.ID AND C.CATEGORIA = 'Profesor'
```

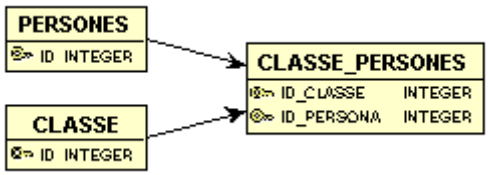
Como podéis ver, la formulación de la misma consulta en SQL es bastante más compleja.

UNA BODA AÚN MÁS COMPLICADA

Hasta ahora, hemos visto cómo casábamos un objeto con una tabla (el caso de `Categoria/CATEGORIAS`) y un objeto con dos tablas (`Persona/PERSONAS/CATEGORIAS`). Son casos típicos de cualquier aplicación y relativamente sencillos.

Pero muestro maridaje no acaba aquí. Hemos de casar el objeto `Classe` con el model relacional. Si observamos nuestro modelo relacional, veremos que la tabla CLASSE está relacionada con la tabla NIVELES. Es una relación idéntica a la descrita para `Persona/CATEGORIAS`. Toda persona tiene una categoría laboral y toda clase tiene un nivel (Primero de EGB, Segundo de EGB, etc.). Hasta aquí, nada nuevo, pero en nuestro modelo relacional aparece la tabla `CLASE_PERSONAS` que nos permite asociar personas a una clase.

La siguiente figura ilustra las relaciones entre la tabla CLASE_PERSONAS y las tablas `PERSONAS` y `CLASE`.



Observamos que la tabla `CLASE_PERSONAS` no aparece en nuestro modelo objetual. El objeto `Clase` tiene las siguientes propiedades:

```
public class Classe {
  /** Identificador único de la clase */
  int    id;
  /** Nombre de la clase. Por ejemplo, "C" */
  String nombre;
```

```
    /** Nivel de la clase. Por ejemplo, "Octavo de EGB" */
    Nivel nivel;
    /** Conjunto de personas asociadas a la clase */
    Set    personas = new HashSet();
```

La relación entre el objeto `Clase` y la tabla `NIVELES` se establece mediante la propiedad `nivel`. Y el archivo de emparejamiento `Clase.hbm.xml` la describe así:

```
<class name="escuela.beans.Clase" table="clase">
[...]
<many-to-one name="nivel" column="ID_NIVEL"/>
```

Donde se indica que la propiedad `nivel` (que Hibernate sabe que es del tipo `Nivel`) se obtiene a partir del identificador `ID_NIVEL` de la tabla `CLASES`.

La relación entre `Clase` y `CLASE_PERSONAS` se establece mediante la propiedad `personas`, un `java.util.Set`. El archivo de emparejamiento `Clase.hbm.xml` la describe así:

```
<class name="escuela.beans.Clase" table="clase">
[...]
<set name="personas" table="CLASE_PERSONAS">
  <key column="ID_CLASE"/>
  <many-to-many class="escuela.beans.Persona" column="ID_PERSONA"/>
</set>
```

Donde se indica que:

- Estamos describiendo la relación entre la clase escuela.beans.Clase y la tabla CLASE:
  `<class name="escuela.beans.Clase" table="clase">`
- La propiedad `personas` se relaciona con la tabla `CLASE_PERSONAS`:
  `<set name="personas" table="CLASE_PERSONAS">`
- La columna clave de la tabla `CLASE_PERSONAS` que se relaciona con la clave primaria del objeto `Clase` es `ID_CLASSE`:
  `<key column="ID_CLASSE"/>`
- La propiedad `personas` tiene una relación “muchos-a-muchos” con la columna `ID_PERSONA` de la tabla `CLASE_PERSONAS`:
  `<many-to-many class="escuela.beans.Persona" column="ID_PERSONA"/>`
- La propiedad `personas` contiene objetos de tipo `escuela.beans.Persona`, con lo cual, Hibernate tiene elementos suficientes para saber que ha de relacionar `Persona.id` con la columna `ID_PERSONA` de la tabla `CLASE_PERSONAS`.

El siguiente fragmento de código indica cómo se guardaría un objeto de tipo `Clase` en la base de datos:

```
// [1] Obtenemos un Nivel de la base de datos (p.e. “EGB1”)
Nivel nivel = null;
nivel = (Nivel)unaSesion.load(Nivel.class, new Integer(1));

// [2] Obtenemos unas cuantas personas de la base de datos
Persona p1 = (Persona)unaSesion.load(Persona.class, new Integer(1));
Persona p2 = (Persona)unaSesion.load(Persona.class, new Integer(2));
Persona p3 = (Persona)unaSesion.load(Persona.class, new Integer(3));

// [3] Creamos una instancia de Clase
Clase clase = new Clase();

// [4] Le asignamos un nombre
clase.setNombre(“A”);
```

```
// [5] Le asignamos el nivel
clase.setNivel(nivel);

// [6] Le añadimos las personas
clase.getPersonas().add(p1);
clase.getPersonas().add(p2);
clase.getPersonas().add(p3);

// [7] Guardamos el objeto “clase” en la base de datos
Transaction tk = unaSesion.beginTransaction();
unaSesion.save(clase);
tk.commit();
unaSesion.close();
```

Los puntos del **[1]** al **[5]**, son evidentes y han sido comentados más arriba para otras clases.

El punto **[6]** muestra cómo se añade un objeto al `Set` que contiene la lista de personas asociadas a una clase.

El punto **[7]** muestra cómo se realiza la persistencia de un objeto en la base de datos. Observamos que el código es el mismo para cualquier objeto. No hay diferencia con el que usábamos con la clase `Categoria`:

```
unaSesion.save(classe);
```

Ahora bien, Hibernate, en este caso, y basándose en el archivo de emparejamientos `Clase.hbm.xml`, hace muchas más cosas que en el caso de `Categoria`:

1. Inserta un nuevo registro en la tabla `CLASE` usando las propiedades `id`, `nombre` y `nivel.id` del objeto `Clase`.
2. Inserta tres (3) nuevos registros en la tabla `CLASE_PERSONAS` asignando a todos ellos el valor de la propiedad `id` del objeto `clase` para la columna `ID_CLASE` y a cada registro el valor de la propiedad `id` de cada uno de los objetos de tipo `Persona` que contiene la propiedad `Personas` del objeto `clase`.

El resultado es que la tabla CLASE tendría los siguientes registros:

```
ID        = 0
NOMBRE    = 'A'
ID_NIVEL  = 1
```

y la tabla `CLASE_PERSONAS` tendría los siguientes tres nuevos registros:

```
ID_CLASE   = 0
ID_PERSONA = 1

ID_CLASE   = 0
ID_PERSONA = 2

ID_CLASE   = 0
ID_PERSONA = 3
```

Y todo esto, con una sola operación:

```
unaSesion.save(clase);
```

Hagámoslo más práctico: el atributo “cascade”

En el ejemplo anterior de inserción de clases a la base de datos, observamos que en el punto **[2]** obtenemos de la base de datos los objetos Persona que formarán parte de la `Clase`. Así, estas personas existen previamente en la base de datos.

Pero ¿qué pasaría si estas personas todavía no estuvieran entradas en la base de datos? La respuesta más obvia es que no funcionaría. Para que funcionara tendríamos que:

1. Añadir cada una de las nuevas personas a la base de datos.
2. Añadir estas personas a nuestra instancia de `Clase`..
3. Guardar nuestra `Clase` en la base de datos.

Este método es correcto, pero un poco pesado.

Hay una manera más práctica que delega el trabajo a Hibernate. En el elemento

```
<set name="personas" table="CLASE_PERSONAS">
```

podemos añadir el atributo **`cascade`** para indicar que haga las operaciones en cascada. Este atributo puede tener los siguientes valores:

- **`save-update`**: para aplicarlo a las operaciones de inserción o de modificación
- **`delete`**: para aplicarlo a las operaciones de borrado
- **`all`**: para aplicarlo a cualquier operación
- **`none`**: para que no se realice nunca ninguna operación en cascada

En nuestro ejemplo, aplicaríamos la opción `save-update`:

```
<set name="personas" table="CLASE_PERSONAS" cascade="save-update">
```

Así, nuestra operación del punto **`[2]`** podría quedar así:

```
// [2] Creamos tres nuevas personas
Persona p1 = new Persona();
Persona p2 = new Persona();
Persona p3 = new Persona();

p1.setId(-1);
p1.setNom(“Nom1”);
p1.setCognoms(“Cognoms1”);
p2.setId(-1);
p2.setNom(“Nom2”);
p2.setCognoms(“Cognoms2”);
p3.setId(-1);
p3.setNom(“Nom3”);
p3.setCognoms(“Cognoms3”);

// [6] Añadamos las personas a la clase
classe.getPersones().add(p1);
classe.getPersones().add(p2);
classe.getPersones().add(p3);

// [7] Guardamos el objeto “clase” en la base de datos
Transaction tk = unaSesion.beginTransaction();
unaSesion.save(classe);
tk.commit();
unaSesion.close();
```

Al guardar la clase en la base de datos Hibernate

1. Crea un registro para cada una de las tres nuevas personas en la tabla PERSONAS.
2. Añade un nuevo registro en la tabla CLASES.
3. Añade tres nuevos registros en la tabla CLASE_PERSONAS.

Fijémonos que hemos asignado un valor `-1` a la propiedad `id` de cada una de las nuevas personas. Éste es el valor que hemos definido para el atributo `unsaved-value` en el archivo de emparejamientos de la clase `Persona`. Esto hace que Hibernate sepa que se trata de personas que aún no están en la base de datos y, por consiguiente, las añade.

Lo mismo podríamos hacer con la propiedad `nivel` del objeto `Clase`:

```
<many-to-one name="nivel" column="ID_NIVEL" cascade="save-update"/>
```

En este caso, si Hibernate encuentra un nivel con un `id = -1`, decide que primero lo tiene que añadir a la tabla `NIVELES` y después añadir la clase.

EL LENGUAJE DE INTERROGACIÓN DEL MUNDO OBJETUAL: EL HQL

Esparcidos por los ejemplos, hemos visto casos de uso del lenguaje HQL. Estos casos de uso nos dan, a mi entender, una idea de por dónde va el HQL, pero no son suficientes para entrever todas sus posibilidades.

En este capítulo trataré de justificar la necesidad del HQL, de sistematizar un poco lo que hemos ido viendo en capítulos anteriores y mostrar algunos aspectos útiles de HQL que aún no hemos visto.

Hay que decir, sin embargo, que este artículo sólo es introductorio y que no pretende cubrir todas las posibilidades del HQL. Éstas son tantas, que sería necesario otro artículo para describirlas.

¿Qué es el HQL y qué sentido tiene?

El HQL (*Hibernate Query Language*) es un lenguaje de interrogación. En el mundo relacional disponemos del SQL (*Structured Query Language*) que nos permite obtener información haciendo preguntas basadas en las tablas y sus columnas. El equivalente en el mundo objetual es el HQL, que nos permite hacer preguntas basadas en los objetos y sus propiedades.

Una vez más, Hibernate se encarga de casar los dos mundos. Traduce las consultas que hacemos desde el mundo objetual en HQL al lenguaje de interrogación del mundo relacional, el SQL, y transforma los resultados obtenidos en el mundo relacional (filas y columnas) en aquello que tiene sentido en el mundo objetual: objetos.

El concepto de “traducción” es importante para entender qué hace Hibernate y uno de los sentidos de HQL. Hemos visto más arriba la equivalencia entre una consulta SQL y una en HQL. Así, la consulta

```
FROM Persona P WHERE P.categoria.id = 3
```

se podría “traducir” a

```
SELECT ID, NOMBRE, APELLIDOS FROM PERSONAS WHERE CATEGORIA_ID = 3
```

y la consulta

```
FROM Persona P WHERE P.categoria.categoria = 'Profesor'
```

se podría “traducir” a

```
SELECT ID, NOMBRE, APELLIDOS FROM PERSONAS P, CATEGORIAS C
WHERE P.ID_CATEGORIA = C.ID AND C.CATEGORIA = 'Profesor'
```

Los ejemplos que acabamos de ver son sencillos y la traducción también lo es. Fijémonos que se usa

una sintaxis SQL estándar. Pero si hemos tenido que trabajar con diversos gestores de bases de datos, habremos observado que hay tantas sintaxis SQL como gestores. Es decir, cada gestor tiene su "dialecto SQL" propio.

En una aplicación JDBC tradicional, si cambiamos el gestor de bases de datos y no hemos ido con cuidado con las instrucciones SQL, nos podemos ver obligados a tener que adaptar las instrucciones SQL a las peculiaridades del dialecto del nuevo gestor.

En una aplicación con Hibernate, el problema desaparece. Sólo hay que cambiar el "dialecto" el el archivo `hibernate.properties` e Hibernate se encarga de traducir el HQL al dialecto SQL que toque. Tenemos, pues, una relación "uno a muchos": lo escribimos una vez en HQL y lo podemos aplicar a muchos gestores/dialectos distintos.

Intentemos (especulemos) describir los pasos que hace Hibernate para resolver una consulta sencilla como esta:

```
FROM Persona P WHERE P.categoria.id = 3
```

1. Determina el dialecto SQL que estamos utilizando (en nuestro caso, HSQLDB) a partir de la información contenida en el archivo `hibernate.properties`:
   ```
   hibernate.dialect net.sf.hibernate.dialect.HSQLDialect
   ```
2. Traduce nuestra consulta al dialecto SQL que toque:
   ```
   SELECT ID, NOMBRE, APELLIDOS FROM PERSONAS WHERE CATEGORIA_ID = 3
   ```
3. Ejecuta la consulta vía JDBC y obtiene un `ResultSet`:
   ```
   3, 'Jaume', 'Figueres Audal'
   ```
4. Construye un objeto de tipo `Persona` a partir de los elementos del `ResultSet`:
   ```
   Persona persona = new Persona();
   persona.setId(3);
   persona.setNombre("Jaume");
   persona.setApellidos("Figueres Audal");
   ```
5. Devuelve el objeto `Persona` creado a partir del `ResultSet`.

Un intent ode sistematización del HQL

Esto suena a una descripción minuciosa y exhaustiva de la sintaxis HQL utilizando la última, definitiva e incomprensible notación formal. Pues no. Sólo quiero sistematizar un poco una parte de la casuística.

Basémonos en dos situaciones típicas de interrogación:

1. Conocemos el ID
2. No conocemos el ID (o queremos hacer preguntas complejas)

Para cada una de las situaciones, Hibernate proporciona métodos o objetos y métodos distintos. La siguiente tabla los resume y, a continuación, los comentamos:

Situación	***Solución Hibernate***
Conocemos el ID	`Session.load()`
No conocemos el ID	`Session.find()`, `Session.iterate()`, `Query`

Conocemos el ID: Session.load()

A lo largo del artículo, hemos visto diversos ejemplos de utilización de `load()`. En todos los ejemplos, hemos usado la misma sintaxis de `load()`:

```
Session.load(Classe.class, id)
```

Por ejemplo:

```
Categoria unaCategoria = (Categoria)sesion.load(Categoria.class,
                                                new Integer(3));
```

Pero el método `load()` tiene más posibilidades que nos pueden resultar cómodas. Por ejemplo,

```
Session.load(unObjeto, id);
```

En este caso, el objeto con el `id` indicado se cargará en `unObjeto`:

```
Categoria unaCategoria = new Categoria();
sesion.load(unaCategoria, new Integer(3));
```

No conocemos el ID: preguntas complejas

Si bien es cierto que la carga de un objeto a partir de su identificador único es una operación bastante frecuente en cualquier aplicación, no deja de ser un caso simple.

Si disponemos de un potente lenguaje de interrogación es para poder hacer preguntas complejas. El lenguaje SQL nos permite hacer preguntas muy, muy complejas, y su equivalente en el mundo objetual, el HQL, también.

No quiero describir aquí detalladamente la sintaxis HQL. Intentaré ejemplificar un abanico razonable de la casuística e invito al lector a que consulte el excelente manual de referencia de Hibernate.

Es necesario separar desde un principio lo que es el lenguaje de interrogación (HQL) de los métodos que Hibernate nos proporciona para ejecutar la pregunta expresada en este lenguaje.

Así, la consulta

```
String consulta = “FROM Categorias C WHERE C.categoria LIKE 'A%';
```

se puede ejecutar de diversas maneras:

```
/* 1 */ List categorias = sesion.find(consulta);
/* 2 */ Iterator iterador = sesion.iterate(consulta);
/* 3 */ Query q = sesion.createQuery(consulta);
        List categs = q.list();
```

Fijémonos que en los tres casos estamos ejecutando la misma consulta.

En primer lugar, nos centraremos en el lenguaje de interrogación HQL y después hablaremos de los diversos métodos que Hibernate nos ofrece para ejecutar las consultas.

A lo largo del artículo, hemos visto diversos ejemplos de HQL. Normalmente, han sido consultas simples que no ilustran todas las posibilidades del lenguaje de interrogación. Vamos a plantear algunas que muestren algunas posibilidades más del HQL.

Funciones escalares

HQL nos permite usar funciones escalares en nuestras consultas. Así, por ejemplo, podemos contar el número de personas de la escuela:

```
SELECT COUNT(*) FROM Persona
```

En nuestro ejemplo, no tenemos más números que los `id` de las clases, así que no puedo poner ejemplos demasiado interesantes ni prácticos de las funciones escalares, pero a guisa de ejemplo, podemos obtener la media de los `id` de `Persona`:

```
SELECT AVG(P.id) FROM Persona P
```

También los podemos sumar:

```
SELECT SUM(P.id) FROM Persona P
```

O obtener los valores máximo y mínimo:

```
SELECT MAX(P.id), MIN(P.id) FROM Persona P
```

Listado de colecciones

Listar los niveles de las diferentes `Clase`s es sencillo:

```
SELECT C.nivel FROM Classe C
```

Si `Nivel` dispone de un método `toString()`, podemos obtener un resultado similar a éste, haciendo un `System.out.println()` de los objetos `Nivel` retornados por la consulta:

```
0 EGB1
```

También es fácil obtener la lista de clases:

```
FROM Classe
```

En este caso, y suponiendo también que `Classe` y `Persona` disponen de los toString() pertinentes, obtendríamos un resultado similar a este:

```
1 0 EGB1 B
          34 Pérez Abril, Verónica 10 Alumne(-a)
          38 Rosales Ribas, Núria 10 Alumne(-a)
           9 Deulofeu Gai, Rosana 9 Professor(-a)
          13 Rosales Ribas, Francesc 9 Professor(-a)
```

Ahora bien, nos puede interesar listar todas las personas de una clase. Es decir, la colección de personas que pertenecen a una clase. Fijémonos que `Clase` tiene una propiedad `personas` que es un `Set`. Para obtener la lista de personas, tendremos que utilizar la función `elements()`.

```
SELECT elements(C.personas) FROM Clase C
```

Esto nos listaría todas las personas de las clases:

```
34 Pérez Abril, Verónica 10 Alumne(-a)
38 Rosales Ribas, Núria 10 Alumne(-a)
 9 Deulofeu Gai, Rosana 9 Professor(-a)
13 Rosales Ribas, Francesc 9 Professor(-a)
```

Contar Colecciones: size

Imaginémonos que nos interesa saber qué clases tienen más de 20 personas relacionadas. Hibernate nos ofrece la posibilidad de contar colecciones mediante una propiedad especial, `size`, o una función especial, `size()`. Así, podremos formular nuestra pregunta de dos maneras distintas:

```
FROM Clase C WHERE C.personas.size > 20
```

o bien

```
FROM Clase C WHERE size(C.personas) > 20
```

La cláusula Group By

Esta cláusula tiene el mismo sentido que en SQL, así que no nos entretendremos demasiado. Un ejemplo sencillo que nos cuente el número de personas por clase nos puede servir:

```
SELECT C.nivel, C.nombre, count(elements(C.personas))
FROM Clase C
GROUP BY C
```

Ejecución de consultas

Existen diversos métodos para ejecutar consultas. Hasta ahora todas las que hemos visto en los ejemplos utilizaban el método `Session.load()`, si la búsqueda se hacía a partir del id, o Session.find(), si ejecutábamos una consulta compleja expresada en HQL. Hay, sin embargo, más maneras de ejecutar una consulta.

En este capítulo, veremos cuáles son estos métodos y cómo se pueden pasar parámetros a una consulta.

El método “Session.find()”

Este método, lo hemos visto ya en la mayor parte de los ejemplos. Nos devuelve el resultado de la consulta en una `java.util.List`. Es bastante práctico si se devuelven pocos resultados, ya que los tiene que mantener en memoria:

```
List personas = sesion.find(“FROM Persona P WHERE P.nombre =
'Berta'”);
```

Ciertamente, `Berta` no es un nombre demasiado común y nos devolverá pocos resultados.

El método “Session.iterate()”

Este método mos devuelve un `java.util.Iterator` y es práctico si la consulta nos proporciona un gran número de resultados.

El iterador se encarga de cargar los objetos resultantes de la consulta uno a uno, a media que los vamos pidiendo:

```
Iterator iterador = sesion.iterate(“FROM Personas AS P
                                    ORDER BY P.apellidos, P.nombre”);
while (iterador.hasNext()) {
  // Obtiene el objeto
  Persona unaPersona = (Persona)iterador.next();

  // Alguna cosa no expresable en la consulta
  if (algoritmoComplicado(unaPersona)) {
    // Borramos la instancia actual
    iterador.remove();

    // No hace falta que sigamos
    break;
  }
}
```

La interficie “Query”

La interficie `Session`, como hemos visto, dispone de tres métodos de ejecución de consulta: `load()`, `find()` e `iterate()`.

La interficie `Query` también nos permite ejecutar consultas pero aporta algunas ventajas:

1. Podemos especificar el número máximo de registros que queremos que se nos devuelvan
2. Podemos especificar el primer registro que queremos obtener
3. Permite el uso de parámetros con nombre

Aquí tenemos un ejemplo:

```
Query q = sesion.createQuery(“FROM Persona AS P”);
q.setFirstResult(5);
q.setMaxResults(20);
List persones = q.list();
```

Parámetros posicionales

Todas las consultas que hemos visto hasta ahora eran fijas; es decir, sabíamos todo lo que necesitábamos para hacer la consulta. Por ejemplo:

```
List personas = sesion.find(“FROM Persona P WHERE P.nombre =
'Jaume'”);
```

Buscamos aquellas personas que se llaman `'Jaume'`.

Pero, a veces, nos puede interesar pasar el nombre como un parámetro. Por ejemplo, si tenemos la consulta ya definida y la ejecutamos diversas veces variando el nombre que buscamos.

El método `Session.find()` nos permite especificar parámetros:

```
String consulta = “FROM Persona P WHERE P.nombre = ?”;
List personas = sesion.find(consulta, unNombre, Hibernate.STRING);
[...]
List personas = sesion.find(consulta, otroNombre, Hibernate.STRING);
```

Fijémonos que en las dos ejecuciones de `find()` usamos el mismo enunciado de la consulta, pero en el primer caso preguntamos por `unNombre` y en el segundo por `otroNombre`.

En el enunciado de la consulta, usamos un parámetro por posición, igual que lo haríamos en JDBC, usando un signo de interrogación.

El ejemplo que hemos puesto es sencillo porque sólo tiene un parámetro. El método `find()` nos permite especificar el valor de este parámetro (`unNombre`, `otroNombre`) y nos obliga a especificar el tipo Hibernate del parámetro (`Hibernate.STRING`). La clase `Hibernate` define constantes estáticas que nos permiten acceder a instancias de `net.sf.hibernate.type.Type`, lo cual nos facilita la vida.

Si la consulta tuviera más de un parámetro, tendríamos que especificar los valores y los tipos mediante matrices de valores y de tipos:

```
String cons = “FROM Persona P WHERE P.nombre = ? AND P.apellidos
LIKE ?”
List personas = Sesion.list(cons,
                            new Object[] {“Jaume”, “R%”},
                            new Type[] {Hibernate.STRING,
                                        Hibernate.STRING});
```

Parámetros con nombre

Todos los que hemos trabajado con JDBC sabemos qué quiere decir tener que contar interrogantes en una consulta con 25 parámetros: acertar a la quinta.

La interficie `Query` nos permite especificar parámetros con nombre, además de los posicionales. Esto tiene varias ventajas:

- El orden en que aparezcan dentro de la tira de la consulta es indiferente
- Un mismo parámetro puede aparecer diversas veces en la misma consulta
- Si el gestor de bases de datos soporta *Scrollable Result Sets*, nos permite movernos arriba y abajo por los resultados de la consulta
- Son autoexplicativos

Para especificar un parámetro con nombre, sólo hay que poner dos puntos (`:`) seguidos del nombre del parámetro:

```
// Parámetro con nombre (preferido)
Query q = sesion.createQuery("FROM CATEGORIAS C " +
                             WHERE C.categoria = :nombreCategoria";
q.setString("nombreCategoria", "Profesor");
Iterator categorias = q.find();

// Parámetro posicional
Query q = sesion.createQuery("FROM CATEGORIAS C " +
                             WHERE C.categoria = ?";
q.setString(0, "Profesor");
Iterator categories = q.find();
```

También podemos pasar una lista de parámetros con nombre:

```
List nombresCateg = new ArrayList();
nombresCateg.add("Profesor");
nombresCateg.add("Alumno");

Query q = sesion.createQuery("FROM CATEGORIAS C " +
                             WHERE C.categoria IN
(:listaNombresCateg);
q.setParameterList("listaNombresCateg", nombresCateg);
```

En este caso, estamos buscando todas las categorías cuyos nombres (`C.categoria`) estén en la lista de nombres `nombresCateg`.

Iterar arriba y abajo

Si el driver JDBC de nuestro gestor de bases de datos soporta *Scrollable ResultSets*, podemos usar la interficie `Query` para obtener una instancia de `ScrollableResults` que nos permitirá disfrutar de las ventajas de los *Scrollable ResultSets*.

El siguiente ejemplo muestra cómo obtener una lista de las personas de la escuela ordenada por apellidos y nombre y ver cuáles son las que aparecerían primero en cada una de las páginas del listado.

```
// Número de personas que caben en una página de listado
int PERSONAS_POR_PAGINA = 20;
String sel = "SELECT P.nombre, P.apellidos, P " +
             "FROM Persona P " +
             "ORDER BY P.apellidos, P.nombre";
Query q = sesion.createQuery(sel);

ScrollableResults personas = q.scroll();

if (personas.first()) {
  // Encontramos el primer nombre que aparece en cada una de las
```

```
  // páginas del listado de personas
  Vector primerosNombresDePagina = new Vector();
  do {
    String nombre     = personas.getString(0);
    String apellidos  = personas.getString(1);
    primerosNombresDePagina.addElement(apellidos + ", " + nombre);
  } while (personas.scroll(PERSONAS_POR_PAGINA));

  // Ahora obtenemos la primera página del listado de personas
  Vector paginaPersonas = new Vector();
  personas.beforeFirst();
  int i = 0;
  while ((PERSONAS_POR_PAGINA > i++ ) && personas.next()) {
    paginaPersonas.addElement(personas.get(2));
  }
}
```

El comportamiento del método `scroll()` es similar al del método `iterate()` que hemos visto mas arriba excepto por el hecho que los objetos se pueden inicializar selectivamente con `get(int)` en vez de inicializar un registro entero cada vez como hace `iterate()`.

Lectura de resultados

Hemos visto ya cómo escribir consultas en HQL y cómo ejecutarlas, pero aún no hemos visto cómo leer los resultados de la ejecución de la consulta. Veámoslo.

En algunos de los ejemplos que hemos visto, ya leíamos resultados. Es el caso de aquéllos que utilizan el método `load()`, ya que este método siempre devuelve un solo objeto:

```
Persona p = sesion.load(Persona.class, 1);
```

También hemos visto aquellos casos de `find()` que nos devuelven una `java.util.List` con la lista de objetos resultante, todos del mismo tipo:

```
List personas = sesion.find("FROM Persona P WHERE p.nombre LIKE 'A%');
```

Los objetos resultantes están el la `List personas`. Sólo hay que recorrer la lista para acceder a ellos.

A veces, sin embargo, la consulta nos devuelve diversos objetos de diversos tipos. Por ejemplo, la consulta

```
SELECT C.nivel , C.nombre, count(elements(C.personas))
FROM Clase C
GROUP BY C
```

nos devolvería un objeto de tipo `Nivel`, otro de tipo `String` y un `Integer`. Podríamos leer los resultados de la consulta usando el método iterate():

```
String consulta = "SELECT C.nivel , C.nombre, " +
                        "count(elements(C.personas)) " +
                  "FROM Clase C " +
                  "GROUP BY C";

Iterator i = sesion.iterate(consulta);
while (i.hasNext()) {
  Object[] registro = (Object[])i.next();
  Nivell  nivel     = (Nivell)registro[0];
  String  nombre    = (String)registro[1];
  Integer personas  = (Integer)registro[2];
}
```

Como podemos ver, el iterador nos devuelve un registro por iteración y este registro es una matriz de objetos: uno por cada una de las columnas de la tabla de resultados. Sólo tenemos que hacer la conversión al tipo esperado (`Nivel`, `String` e `Integer` en nuestro ejemplo) y ya tendríamos los resultados.

¿QUÉ UTILIZO?

En este artículo, he utilizado la versión 2.0.3 de ***Hibernate***[10] (la última estable a fecha de hoy[11]) y como gestor de bases de datos, *HSQLDB* versión 1.7.1[12]; un magnífico y completo gestor de bases de datos escrito en Java. Hay que remarcar que tanto Hibernate como HSQLDB son productos de código abierto.

He utilizado como entorno de desarrollo Java ***Eclipse*** versió 2.1.2. Este entorno de desarrollo es completamente gratuito y, como Hibernate y HSQLDB, de código abierto. A mi entender, por el momento, es el mejor entorno de desarrollo para Java. Y, por supuesto, el que tiene mejor relación calidad-precio.

Para los diagramas UML, he utilizado un plugin de Eclipse llamado ***Eclipse UML plugin***, de OMONDO. Una buena herramienta de modelado con dos versiones: una de pago y otra gratuita. Yo, obviamente, uso la gratuita.

Para los diagramas de relaciones, he usado ***DbVisualizer*** versió 4.0.1 de Minq Software. Una herramienta realmente interesante para obtener información de bases de datos. La última versión, es capaz de construir y presentar unos diagramas de relaciones realmente sorprendentes. Hay una versión comercial (a precio razonable) y otra gratuita (con algunas funciones menos, pero igualmente interesante).

Para generar los archivos básicos de emparejamientos, he usado el plugin de Eclipse ***Hibernator***. versión 0.9.6. Es interesante, pero le faltan algunas cosas. Entre otras, documentación.

Para probar las consultas, he usado ***Hibern8IDE***. Una interficie gráfica que nos ayuda a hacer prototipos de las consultas. También Hibernator proporciona una interficie para probar consultas pero me parece menos potente, Hibern8IDE e Hibernator comparten una carencia: la documentación. Una vez bien configurado, Hibern8IDE es una herramienta auxiliar excelente.

¿QUÉ HAY QUE LEER?

Esta introducción a Hibernate no habría sido posible sin la excelente documentación oficial de Hibernate ni sin los artículos que terceras personas han escrito sobre el tema. El propio equipo de Hibernate proporciona una lista muy interesante.

Yo diría que es más conveniente pasar por los artículos introductorios antes que adentrarse en la documentación oficial de referencia.

10 http://www.hibernate.org
11 En el momento de la traducción, ya está disponible la versión 2.1.2, del 4 de febrero de 2004. Sin embargo, el artículo sigue teniendo vigencia.
12 http://hsqldb.sourceforge.net/

¿QUÉ NO HE EXPLICADO?

No he explicado un montón de cosas interesantes que se pueden hacer con Hibernate. No he hablado de la posibilidad de utilizar XDoclet con Hibernate, ni de cómo podemos especificar consultas con nombre en los archivos de emparejamiento, ni de las herramientas de generación de código como Middlegen o las que vienen con Hibernate.

Me he limitado a los casos más sencillos y habituales y no he hablado nada de colecciones ni de filtros. He obviado descaradamente los temas relacionados con el rendimiento y las capacidades de optimización de Hibernate.

En fin, la sensación que tengo es que he explicado un montón de cosas pero que hay muchas más que se han quedado en el tintero.

Me daría por satisfecho si este artículo motivara al lector a introducirse en el mundo de la persistencia objetual y en el entorno de Hibernate.

APÉNDICE: APLICACIÓN DE EJEMPLO

Para ilustrar los contenidos de este artículo, he desarrollado una pequeña aplicación de ejemplo implmentando la versión 1.7.1 de HSQLDB.[13] Podéis descargar los fuentes del ejemplo en http://www.froses.com/Assets/Files/Articles/Hibernate_Introduccion_es.zip.

La aplicación de ejemplo ha sido desarrollada con Eclipse 3.0 M8, que podéis descargar desde http://www.eclipse.org/downloads/index.php, aunque no es obligatorio.

Los fuentes de la aplicación han sido empaquetados en un archivo ZIP (Hibernate_Introduccion_es.zip) que, a grandes rasgos, contiene la siguiente estructura de directorios:

Directorio	*Descripción*
HIBERNATE_ARTICULO	Directorio principal. Contiene el archivo de propiedades de Hibernate (`hibernate.properties`) y el de *Log4Java* (`log4j.properties`) además del resto de directorios que comento a continuación.
Databases	Contiene la base de datos creada y el *script* SQL por si la queréis crear de nuevo (`ESCUELA.SQL`).
escuela.beans	Es el directorio del *package* de *beans* y de los archivos de emparejamiento de Hibernate (`Categoria.hbm.xml`, `Nivel.hbm.xml`, `Persona.hbm.xml` y `Clase.hbm.xml`)
escuela.abm	Es el directorio del *package* que contiene las clases que permiten realizar altas, bajas y modificaciones (ABM) para los distintos *beans*.

Para la creación de la base de datos, me remito al manual de HSQLDB. El archivo de configuración de Hibernate del ejemplo supone que el gestor de bases de datos está arrancado en modo servidor. Para acceder a la base de datos en otra modalidad, deberéis modificar la URL especificada en el archivo `hibernate.properties`.

13 La versión 1.7.1 de HSQLDB se puede descargar de http://sourceforge.net/projects/hsqldb/.