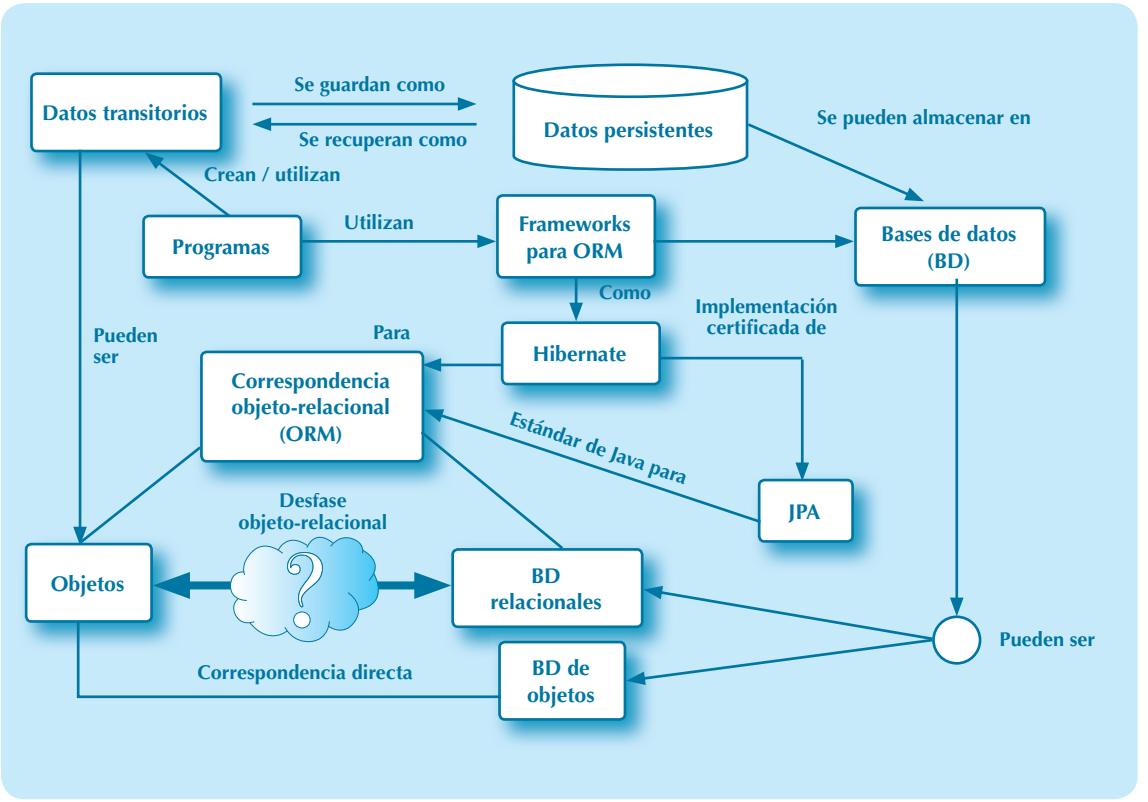
4

Correspondencia objeto-relacional

Objetivos

- ✓ Comprender el propósito y los fundamentos de la correspondencia objeto-relacional (ORM).
- ✓ Utilizar Hibernate para persistencia de objetos en un esquema relacional mediante ficheros de correspondencia hbm, incluyendo relaciones de uno a uno, de uno a muchos y herencia.
- ✓ Crear POJO (ficheros de clases de Java) apropiados para ORM con Hibernate, crear un esquema relacional para almacenamiento de objetos persistentes, y crear ficheros de correspondencia para establecer la correspondencia entre POJO y tablas del esquema relacional.
- ✓ Analizar el ciclo de vida de un objeto persistente en Hibernate y crear programas en Java que utilicen las principales operaciones disponibles en cada estado.
- ✓ Conocer los fundamentos de los lenguajes HQL y JPQL y crear programas que utilicen HQL.

Mapa conceptual



Glosario

Contexto de persistencia. Lleva el control de los cambios realizados sobre objetos persistentes asociados a él durante una sesión mediante un gestor de entidades, para reflejarlos en la base de datos.

Desfase objeto-relacional. Conjunto de dificultades conceptuales y técnicas para llevar a cabo la persistencia de objetos sobre bases de datos relacionales.

Ficheros de correspondencia o ficheros hbm. Ficheros propios de Hibernate en los que se especifica la correspondencia entre una clase y un esquema relacional (es decir, un conjunto de tablas relacionadas). Es decir, en un fichero de correspondencia se especifica cómo la información contenida en un objeto de una clase determinada se almacena en un esquema relacional.

HQL *(Hibernate Query Language)*, JPQL *(Java Persistence Query Language)*. Lenguajes similares a SQL pero que manejan conjuntos de objetos y sus atributos en lugar de conjuntos de filas y columnas. JPQL es un subconjunto de HQL y es parte de la especificación JPA.

JPA *(Java Persistence Arquitecture)*. API estándar de Java para ORM.

Objeto persistente. Objeto que tiene una representación persistente en la base de datos. Un objeto persistente está asociado a un contexto de persistencia que detecta los cambios que se realizan sobre él para, en su momento, reflejarlos en la base de datos.

ORM *(Object-Relational Mapping)*. Correspondencia objeto-relacional. Técnicas y herramientas que hacen posible la persistencia de objetos en un esquema relacional, basada en una correspondencia entre clases creadas con un lenguaje de programación orientado a objetos y tablas creadas en una base de datos relacional. La traducción correcta de *mapping* en español es *correspondencia*, y no *mapeo,* aunque esta última es, con mucho, la más habitual.

4.1. Correspondencia objeto-relacional

Los lenguajes orientados a objetos se popularizaron enormemente en los años noventa. C++ estaba disponible desde mediados de los ochenta como una extensión orientada a objetos del lenguaje C. Java se popularizó a mediados de los noventa, especialmente para aplicaciones de gestión empresarial con uso intensivo de bases de datos relacionales. Para entonces las bases de datos relacionales ya tenían un dominio absoluto como medio de almacenamiento de datos, que continúa en la actualidad. El lenguaje Java se ha seguido utilizando para este tipo de aplicaciones y ha sido dominante en el campo de las aplicaciones empresariales de gran envergadura, normalmente desplegadas sobre servidores de aplicaciones basados en Java EE (Java Enterprise Edition).

Bien es cierto que utilizar lenguajes orientados a objetos no significa necesariamente utilizar sus características orientadas a objetos. De hecho, gran parte de estas aplicaciones no lo hacen. Es decir, no utilizan objetos para representar los datos persistentes con los que trabajan, o bien no sacan partido de la herencia y otras posibilidades de estos lenguajes.

En cualquier caso, cada vez más se planteó la conveniencia o incluso la necesidad de almacenar objetos en bases de datos relacionales, incluso objetos complejos con referencias a otros objetos, o a colecciones de objetos, y objetos de clases sobre las que se definen otras subclases. La persistencia de objetos en bases de datos relacionales planteaba una serie de problemas, conocidos conjuntamente como *desfase objeto-relacional* (en inglés *object-relational impedance mismatch*) o *desajuste de impedancia objeto-relacional* (véase la figura 1.9).

Para resolver o para evitar estos problemas surgieron varios planteamientos:

1. *Bases de datos de objetos.* Almacenan directamente objetos. Esta parece en principio la solución ideal o la más natural, y esta idea surgió con fuerza en los noventa, durante el *boom* de la programación orientada a objetos. Pero el desarrollo de estas bases de datos planteaba problemas conceptuales y prácticos. Para empezar, la falta de un modelo formal ampliamente aceptado en el que basarse, al contrario que las bases de datos relacionales, basadas en el modelo relacional. También la falta de estándares ampliamente adoptados, como SQL para las bases de datos relacionales, a pesar del trabajo inicial de ODMG (Object Data Management Group). Este grupo se creó en 1991 y se disolvió en 2001. Hoy en día las bases de datos de objetos siguen teniendo un uso muy reducido.
2. *Bases de datos objeto-relacionales.* Son bases de datos relacionales con capacidades para gestionar objetos. En SQL:99 se introdujeron tipos estructurados definidos por el usuario,

entre ellos los tipos objetos (clases). Son una solución de compromiso y resuelven solo parcialmente el problema.

3. *ORM o correspondencia objeto-relacional*. Consiste en el establecimiento de una correspondencia entre clases definidas en un lenguaje de programación orientado a objetos, como Java, y tablas de una base de datos relacional, y en el uso de mecanismos para que las modificaciones sobre los objetos se registren en la base de datos y, a la inversa, para que se pueda recuperar en objetos la información registrada en la base de datos. Esto hace posible la persistencia de objetos en bases de datos puramente relacionales. La primera herramienta ORM importante que surgió fue Hibernate para el lenguaje Java en 2001, como alternativa a la persistencia de objetos proporcionada de Java EE, basada en EJB (Enterprise Java Beans). Después de Hibernate surgieron otras herramientas y *frameworks* ORM para Java, como Apache Cayenne, Apache OpenJPA, Oracle TopLink, etc. Utilizan en general JDBC para la interacción con la base de datos y funcionan con diferentes bases de datos. También existen soluciones ORM para otros lenguajes orientados a objetos como C++, PHP, Python, etc.

4.2. Hibernate

Hibernate es un *framework* de ORM para Java, distribuido bajo licencia LGPL 2.1 de GNU. Al estar distribuido bajo licencia LGPL, Hibernate se puede utilizar en aplicaciones comerciales sin necesidad de hacer público su código fuente.

Recurso web

En el sitio web de Hibernate ORM se puede encontrar el *software* y abundante documentación: http://hibernate.org/orm/.

Hibernate se desarrolló en 2001 como una alternativa a la persistencia de objetos proporcionada en Java EE mediante EJB. Java EE es una plataforma muy potente pero compleja y que necesita muchos recursos, pensada para aplicaciones empresariales de envergadura. Como alternativa, se desarrolló Hibernate, un *framework* que proporciona persistencia de objetos basada en ORM, pero sin necesidad de un servidor Java EE. Tiene soporte para transacciones, y utiliza técnicas para mejorar el rendimiento, como por ejemplo *pool* de conexiones, técnicas de *caching* y también *batching* (agrupación en lotes) para las operaciones sobre las bases de datos.

En 2010, Hibernate 3 (versión 3.5.0 y posteriores) se convirtió en implementación certificada de JPA 2.0. JPA es la API estándar de Java para persistencia de objetos, y parte de la especificación Java EE. A partir de entonces, Hibernate siempre ha proporcionado dos API, tanto la suya propia como la de JPA.

Hibernate tiene su propio lenguaje para manejar objetos persistentes, HQL, inspirado en SQL. Las sentencias de SQL operan sobre filas y columnas de tablas, mientras que las de HQL lo hacen sobre conjuntos de objetos persistentes y sus atributos. El lenguaje estándar de JPA para manejo de objetos persistentes, JPQL, es un subconjunto de HQL.

La siguiente ilustración muestra la arquitectura de Hibernate y sus principales componentes. Como ya se ha comentado, Hibernate proporciona tanto una API propia como una implementación de la API de JPA 2.0. En esta ilustración se muestran las clases y componentes de la API propia de Hibernate, pero son muy similares en concepto a los de la API de JPA.

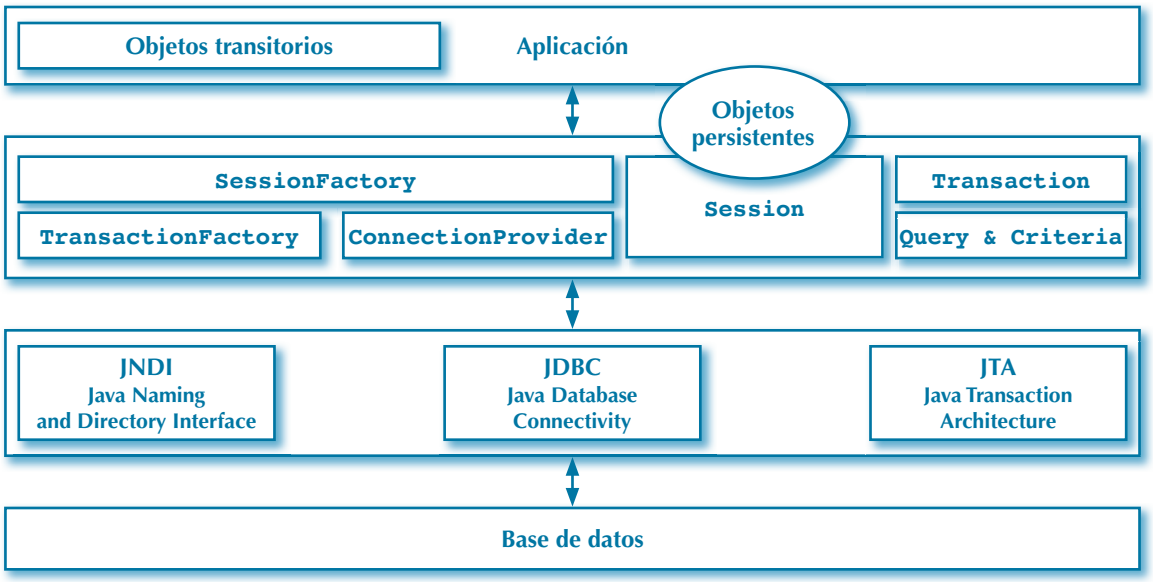


Figura 4.1
Arquitectura de Hibernate

Una aplicación que utiliza Hibernate trabaja con objetos tanto transitorios como persistentes. Los objetos persistentes están siempre asociados a una sesión (`Session`) creada por una `SessionFactory`. Una aplicación puede crear objetos transitorios y después convertirlos en persistentes para que después se almacenen en la base de datos. Puede también recuperar un objeto persistente desde la base de datos y realizar cambios sobre él, para que después se reflejen en la base de datos. Puede agrupar operaciones sobre los objetos persistentes dentro de transacciones (`Transaction`) que están siempre asociadas a una sesión. Puede también utilizar sentencias de HQL mediante la clase `Query`. Hibernate interactúa con la base de datos mediante JDBC. Hibernate puede utilizar internamente API de Java tales como JNDI (Java Naming and Directory Interface) para fuentes de datos (`DataSource`) y JTA (Java Transaction Arquitecture) para transacciones.

4.3. Iniciación a la correspondencia objeto-relacional con Hibernate

La correspondencia entre objetos de Java y datos en un esquema relacional (es decir, un conjunto de tablas) es compleja si se tienen en cuenta todos los posibles casos. Pero para empezar con lo más sencillo, en general se puede decir que para cada clase existe una tabla en el esquema relacional que permite almacenar los objetos de esa clase.

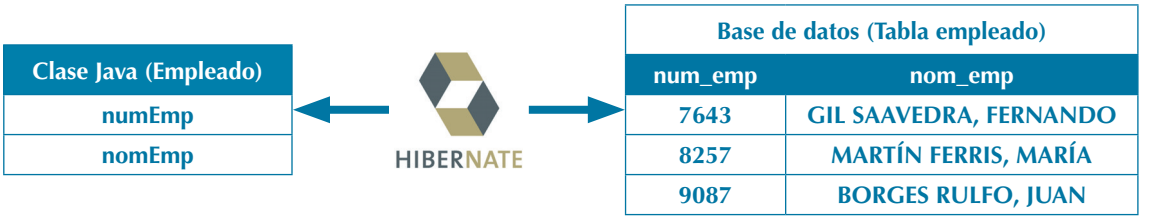


Figura 4.2
Correspondencia entre una clase de Java sencilla y una tabla de la base de datos

La correspondencia de otras características de las clases es menos directa. A saber:

- Colecciones de objetos, que representan relaciones de uno a muchos o de muchos a muchos entre objetos.
- Referencias a objetos, que representan relaciones de uno a uno o de muchos a uno entre objetos.
- Herencia.

Hibernate permite establecer una correspondencia objeto-relacional en la que basar la persistencia de objetos sobre bases de datos relacionales no solo cuando se parte de cero, sino también cuando se parte de una base de datos ya existente o cuando se quiere añadir persistencia para clases ya creadas. Esto es muy importante porque no siempre se desarrolla una aplicación desde cero, sino que a menudo debe construirse sobre algo ya existente.

Hibernate impone, eso sí, algunas condiciones tanto a las tablas como a las clases. Si no se crean de cero, sino que vienen dadas, y no las cumplen, puede ser necesario un trabajo previo de adecuación.

- Las tablas deben tener clave primaria. En una base de datos relacional no es obligatorio que todas las tablas la tengan, pero en la práctica casi siempre es así. Cuando no existe, se puede crear una nueva clave primaria autogenerada (se habló de ellas en el capítulo previo dedicado a bases de datos relacionales). Esto es sencillo y normalmente no obliga a hacer ningún cambio en las aplicaciones que utilizan la tabla. En correspondencia, las clases deben tener un atributo para almacenar un identificador único.

RECUERDA

✓ Un clave primaria autogenerada para una tabla se define sobre una columna numérica, de manera que el sistema gestor de base de datos asigna un nuevo valor para esa columna para cada nueva fila insertada en la tabla. En MySQL se crean con la opción `AUTO_INCREMENT` en la definición de la columna.

- Los valores de la columna o columnas que forman la clave primaria no pueden cambiar. Esto significa, por ejemplo, que si la clave primaria es el DNI, este no se puede cambiar, ni siquiera para corregir errores o para rellenar con ceros por la izquierda, no al menos con una aplicación basada en Hibernate. En la práctica, lo más habitual es tener para

cada tabla una clave primaria autogenerada, lo que evita este tipo de inconvenientes. Si en este caso particular se quisieran evitar duplicidades de DNI, que podrían darse si no es clave primaria, se podría definir un índice único para ese campo, por ejemplo: `CREATE UNIQUE INDEX i_cliente_dni ON cliente(dni)`.

- Las clases deben implementar la interfaz `Serializable`. Para ello basta con incluir `implements Serializable` en la definición de la clase.
- Las clases deben cumplir una serie de convenciones tales como la existencia de método `getX` y `setX` (*gettters* y *setters*) para obtener y asignar, respectivamente, el valor de cada atributo `X`.

TOMA NOTA

Hay que tener muy presentes los problemas que pueden surgir debido a la nomenclatura de las tablas y los campos cuando se usan algunos sistemas operativos. En Windows y MacOS pueden surgir problemas si se usan mayúsculas para los nombres de tablas o de campos. El motivo, en última instancia, es que los datos se almacenan en sistemas de ficheros y su nombre se puede convertir a minúsculas. Para evitar problemas, lo mejor es utilizar nombres sin mayúsculas para las tablas y sus campos y, en general, para cualquier objeto de la base de datos.

La correspondencia se puede establecer mediante ficheros de correspondencia de tipo hbm *(Hibernate mapping)*. Estos son ficheros de XML en los que se especifica la correspondencia entre, por una parte, clases y sus atributos, y por otra parte, tablas y sus columnas. También, alternativamente, se puede establecer mediante JPA, por medio de anotaciones en las clases. Se puede incluso utilizar un mecanismo para unas clases y otro para otras.

4.4. Correspondencia objeto-relacional a partir de las tablas

En esta sección se explicará cómo con Hibernate se puede establecer la correspondencia para un conjunto de tablas relacionadas. Algunos IDE o entornos de desarrollo incluyen asistentes para establecer esta correspondencia, si bien puede ser necesario o conveniente ajustar posteriormente algunas cosas manualmente. Se explica-

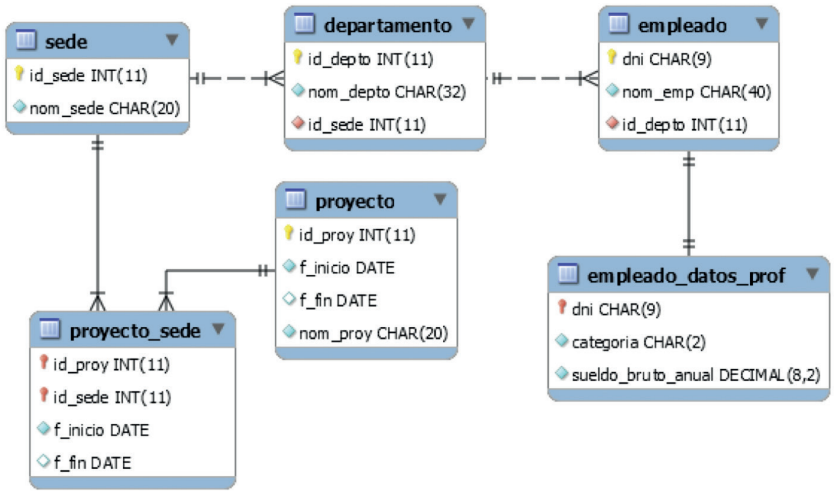


Figura 4.3
Esquema relacional para proyecto de ORM con Hibernate

rá cómo hacerlo en el IDE Netbeans. Se acabará con un pequeño programa de ejemplo que se basa en esta correspondencia para crear varios objetos relacionados y almacenarlos en la base de datos. Los nombres de las tablas se ponen en singular (`departamento` y no `departamentos`). Esto no es estrictamente necesario, pero hace que los nombres generados por Hibernate para la definición de las clases sean claros y descriptivos.

Con las siguientes sentencias de SQL se puede crear, en MySQL, el esquema relacional anterior y un usuario `libro_ad` con los permisos necesarios sobre él. Para ello se puede entrar con el usuario `root`.

```
create database proyecto_orm;
use proyecto_orm;
create table sede(
    id_sede integer auto_increment not null,
    nom_sede char(20) not null,
    primary key(id_sede)
);
create table departamento(
    id_depto integer auto_increment not null,
    nom_depto char(32) not null,
    id_sede integer not null,
    primary key(id_depto),
    foreign key fk_depto_sede(id_sede) references sede(id_sede)
);
create table empleado(
    dni char(9) not null,
    nom_emp char(40) not null,
    id_depto integer not null,
    primary key(dni),
    foreign key fk_empleado_depto(id_depto) references departamento(id_depto)
);
create table empleado_datos_prof(
    dni char(9) not null,
    categoria char(2) not null,
    sueldo_bruto_anual decimal(8,2),
    primary key(dni),
    foreign key fk_empleado_datosprof_empl(dni) references empleado(dni)
);
create table proyecto (
    id_proy integer auto_increment not null,
    f_inicio date not null,
    f_fin date,
    nom_proy char(20) not null,
    primary key(id_proy)
);
create table proyecto_sede (
    id_proy integer not null,
    id_sede integer not null,
    f_inicio date not null,
    f_fin date,
    primary key(id_proy, id_sede),
    foreign key fk_proysede_proy (id_proy) references proyecto(id_proy),
    foreign key fk_proysede_sede (id_sede) references sede(id_sede)
);
CREATE USER 'libro_ad'@'localhost' IDENTIFIED BY '(password)';
GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP,EXECUTE
    ON proyecto_orm.* TO 'libro_ad'@'localhost';
```

4.4.1. Creación de la conexión con la base de datos

Lo primero es crear en NetBeans una conexión con la base de datos. Habrá que seleccionar uno de los *drivers* disponibles en la pestaña "Services", o si ninguno de ellos permite la conexión, habrá que instalar antes un *driver* que la permita. En este caso se quiere establecer una conexión con un servidor MySQL 8.0. Se pueden consultar las propiedades de un *driver* pulsando sobre él con el botón derecho y seleccionando la opción "Customize". En este ejemplo, se puede ver que el nombre del fichero `jar` para el *driver* con nombre "MySQL (Connector/H driver)" incluye el número de versión 5.1.23.

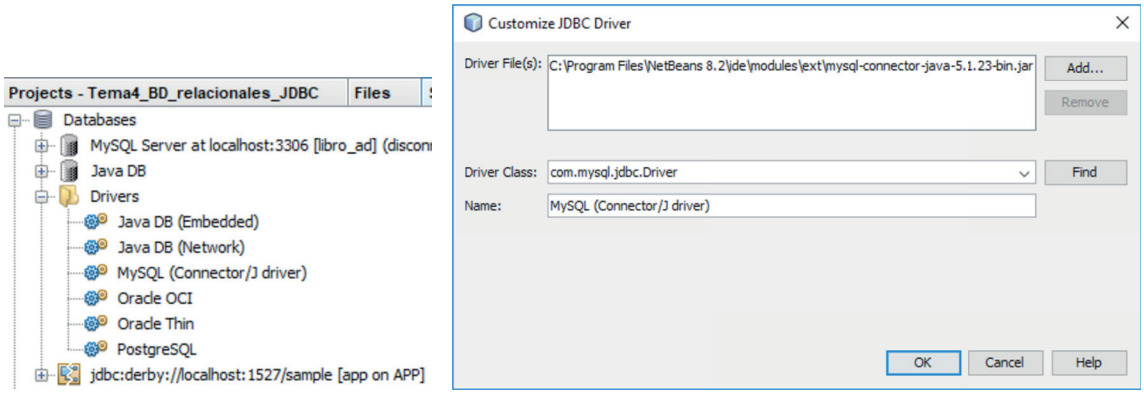


Figura 4.4
Consulta de los *drivers* instalados y de sus propiedades en NetBeans

Ni este *driver* ni ningún otro disponible permite la conexión con MySQL 8.0. Hay que añadir el *driver* apropiado, que viene empaquetado en un fichero `jar` disponible en la sección de descargas del sitio web de MySQL. Para ello, se pulsa con el botón derecho sobre "Drivers" y se selecciona la opción "New driver...". Por último, se selecciona el fichero `jar`. Hay que pulsar el botón "Find" para localizar la clase que implementa el *driver*. Conviene cambiarle el nombre para diferenciarlo del *driver* para versiones anteriores de MySQL. Se ha elegido "MySQL (Connector/J driver) [8.0]".

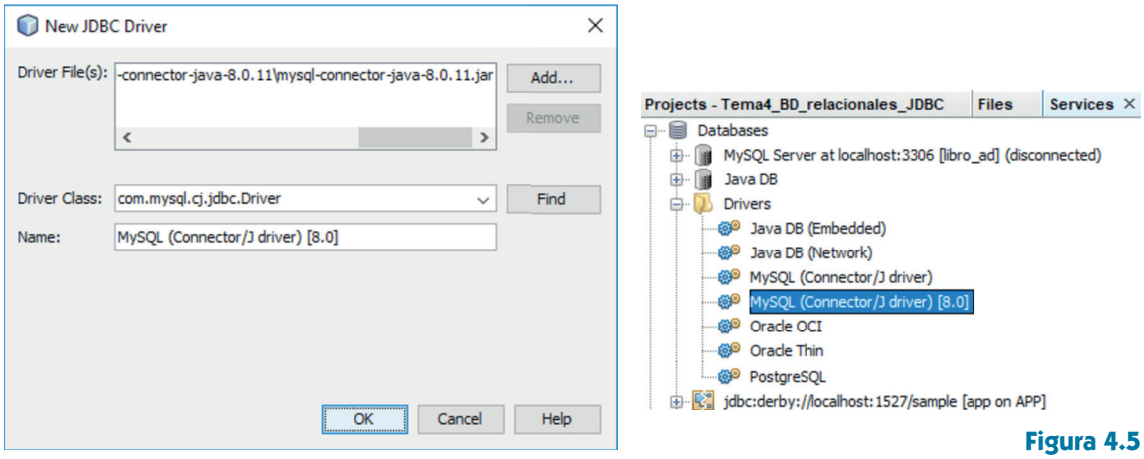


Figura 4.5
Instalación de un nuevo *driver* JDBC en NetBeans

Para crear la conexión a la base de datos, se pulsa con el botón derecho sobre "Databases" y se selecciona la opción "New connection…". Después se selecciona el *driver* y se especifican las opciones de conexión. Para MySQL 8.0 es necesario añadir algunas opciones en la URL de conexión y esta quedaría así:

```
jdbc:mysql://localhost:3306/proyecto_ORM?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
```

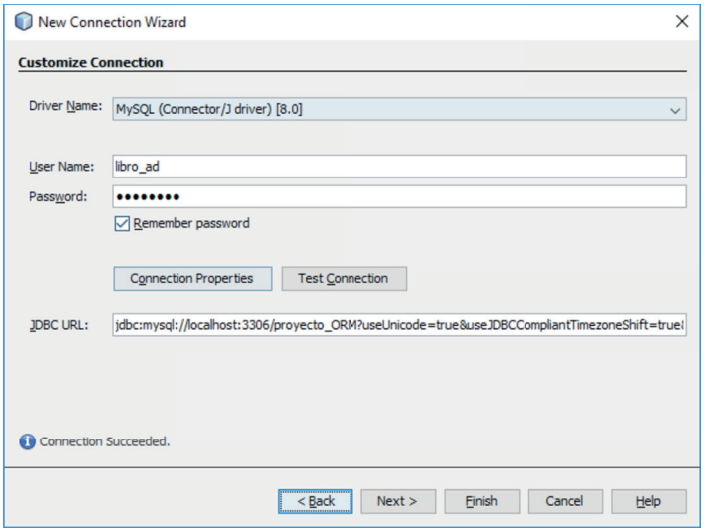


Figura 4.6
Datos de conexión para nueva conexión

Conviene verificar que la conexión se puede establecer pulsando el botón "Test Connection".

Por último, se especifica el nombre de la conexión. En este caso se le pone el nombre `proyecto_ORM`.

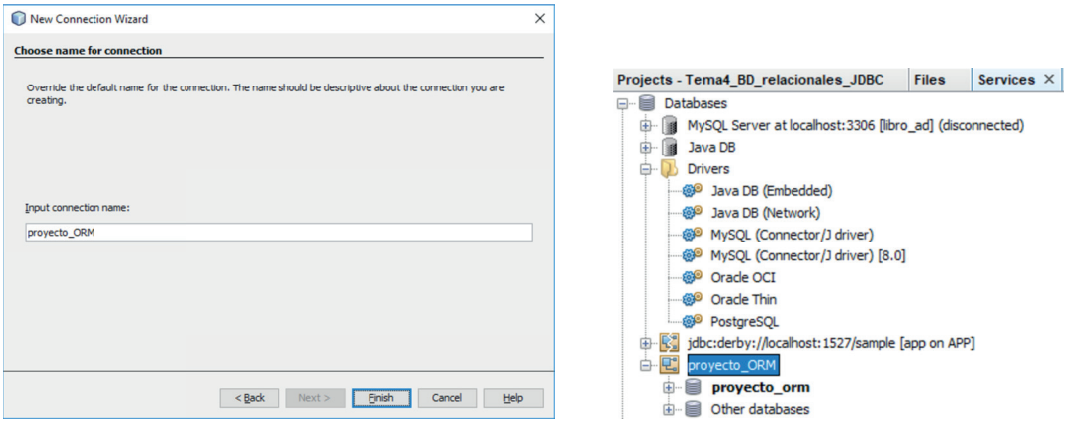


Figura 4.7
Nombre para nueva conexión

4.4.2. Creación del proyecto

Se trata de crear un proyecto en NetBeans como se ha venido haciendo hasta ahora, con nombre `ORM_conexion`.

Una vez creado el proyecto, se pueden utilizar distintos *wizards* o asistentes para automatizar la creación de todo lo necesario para que el proyecto pueda hacer uso de la persistencia de objetos con Hibernate.

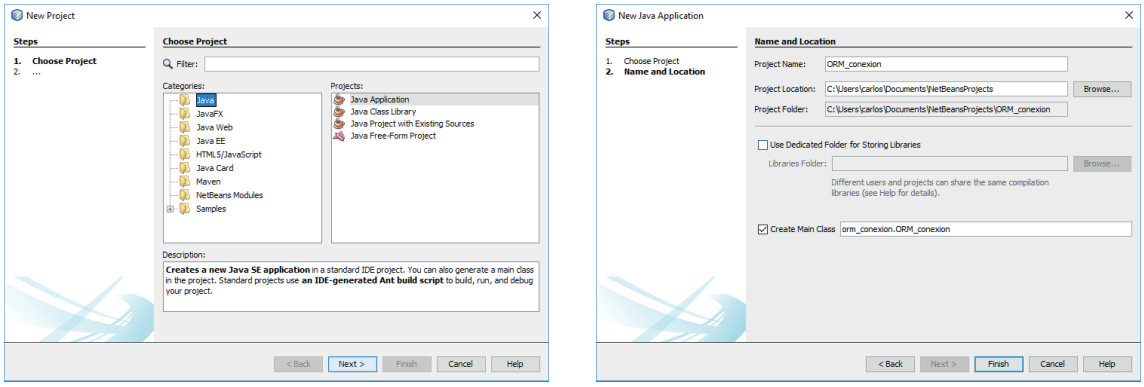


Figura 4.8
Nuevo proyecto en NetBeans

4.4.3. Fichero de configuración de Hibernate: `hibernate.cfg.xml`

Este fichero contiene todo lo necesario para establecer una conexión con la base de datos.

Para crear el fichero de configuración con Netbeans 8.2 se pulsa sobre el proyecto con el botón derecho del ratón y se selecciona la opción "New…", después "Other…", y después "Hibernate". Entonces se muestra una lista con todos los asistentes para Hibernate (figura 4.9).

Se selecciona "Hibernate Configuration Wizard". En los siguientes diálogos se indica `src` como directorio (*Folder*) para situar el fichero y como conexión la que se acaba de crear en el apartado anterior `proyecto_ORM` (figura 4.10). Eso es todo.

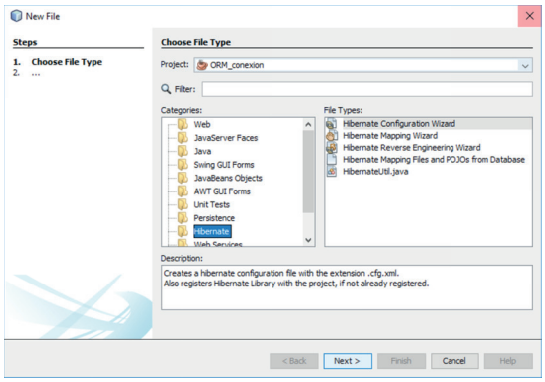


Figura 4.9
Asistentes para Hibernate en NetBeans 8.2

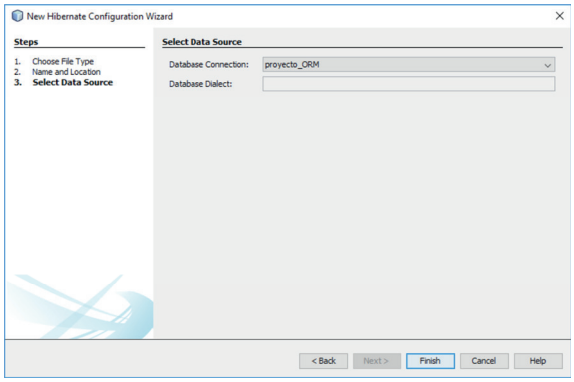


Figura 4.10
Asistente para creación de fichero de configuración `hibernate.cfg.xml`

El contenido del fichero de configuración recién creado es este. Como se puede ver, contiene los datos necesarios para establecer una conexión a la base de datos con JDBC.

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

```
        <session-factory> <property name="hibernate.connection.driver_class">
            com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.
            Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/
            proyecto_orm?(... URL de conexión, conteniendo opciones
            adicionales)</property>
        <property name="hibernate.connection.username">libro_ad</property>
        <property name="hibernate.connection.password">(password)</property>
    </session-factory>
</hibernate-configuration>
```

4.4.4. Fichero de ingeniería inversa `hibernate.reveng.xml`

En este fichero de configuración se especifica para qué clases y tablas se va a establecer la correspondencia, y suele tener por nombre `hibernate.reveng.xml` (figura 4.11).

Se puede crear este fichero con otro asistente para Hibernate. Es necesario especificar el fichero de configuración antes creado y las tablas para las que se va a establecer correspondencia. Como directorio para situar el fichero ("Folder") se indica `src`. Hay que elegir las tablas (figura 4.12).

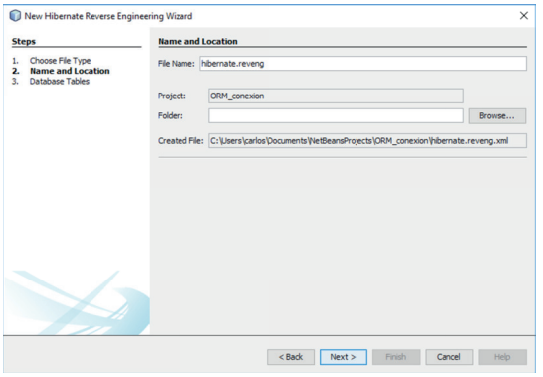


Figura 4.11
Asistente para creación del fichero de ingeniería inversa `hibernate.reveng.xml`

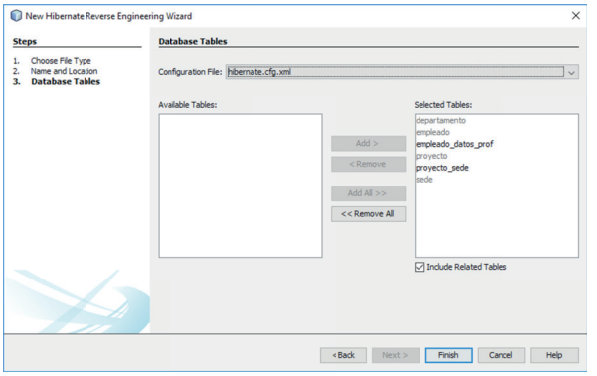


Figura 4.12
Selección de tablas para el fichero de ingeniería inversa

Si se muestra un mensaje de error indicando que los *drivers* no se han añadido al proyecto, hay que añadirlos como se explicó en el capítulo anterior. Si no se ven las tablas que cabría esperar ver, o bien si aparecen junto con el texto *(no primary key)*, probablemente se está trabajando en Windows y no se ha tenido la precaución de no utilizar mayúsculas para nombres de tablas y atributos.

A continuación, se muestra el contenido del fichero de ingeniería inversa recién creado. En él se indica la base de datos y las tablas que se van a hacer corresponder con objetos.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate
    Reverse Engineering DTD 3.0//EN" "http://hibernate.sourceforge.net/
    hibernate-reverse-engineering-3.0.dtd">
<hibernate-reverse-engineering>
    <schema-selection match-catalog="proyecto_orm"/>
    <table-filter match-name="proyecto"/>
```

```
    <table-filter match-name="empleado"/>
    <table-filter match-name="empleado_datos_prof"/>
    <table-filter match-name="proyecto_sede"/>
    <table-filter match-name="departamento"/>
    <table-filter match-name="sede"/>
</hibernate-reverse-engineering>
```

4.4.5. POJO (clases) y ficheros de correspondencia

Esta es la parte interesante. Finalmente se generan las clases y los ficheros de correspondencia con el asistente "Hibernate Mapping Files and POJOs from Database". POJO es el acrónimo de *plain old Java file,* algo así como "un fichero de Java de toda la vida, sin más". Son clases que incluyen un atributo por cada columna de la tabla y un método *getter* y *setter* para cada uno, es decir, un método para obtener su valor y otro para asignarlo.

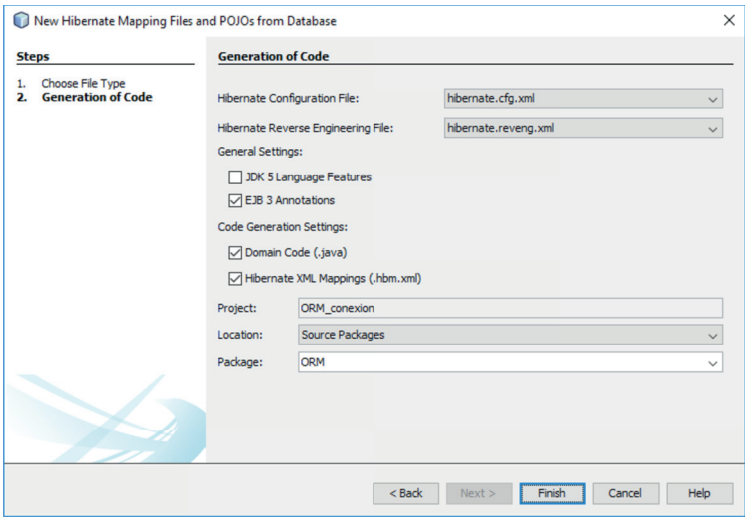


Figura 4.13
Asistente para la creación de clases (POJO) y ficheros de correspondencia

Hay que indicar el fichero de configuración antes creado, el de ingeniería inversa, y el nombre del paquete para incluir los POJO y los ficheros de correspondencia. En este ejemplo se opta por crear un nuevo paquete llamado `ORM`. La opción "EJB 3 Annotations" se puede utilizar para generar los ficheros con anotaciones de JPA. No supone ningún problema generarlos con esas anotaciones aunque no se vaya a utilizar JPA para la correspondencia. Como se verá en breve, en el fichero de configuración `hibernate.cfg.xml` se indicarán los ficheros hbm para la correspondencia de cada clase, con lo que se ignorarán estas anotaciones.

Además de crear las clases (POJO) y los ficheros de correspondencia, este asistente añade las siguientes líneas al fichero de configuración `hibernate.cfg.xml`, para establecer la correspondencia de las clases mediante los correspondientes ficheros hbm. Si para alguna clase se quisiera utilizar anotaciones en lugar de ficheros hbm, habría que eliminar del fichero de configuración la línea correspondiente a su fichero de correspondencia. Más adelante se verá en detalle el contenido de estos ficheros de correspondencia.

```
<mapping resource="ORM/Departamento.hbm.xml"/>
<mapping resource="ORM/Empleado.hbm.xml"/>
<mapping resource="ORM/EmpleadoDatosProf.hbm.xml"/>
<mapping resource="ORM/ProyectoSede.hbm.xml"/>
<mapping resource="ORM/Proyecto.hbm.xml"/>
<mapping resource="ORM/Sede.hbm.xml"/>
```

4.4.6. HibernateUtil.java

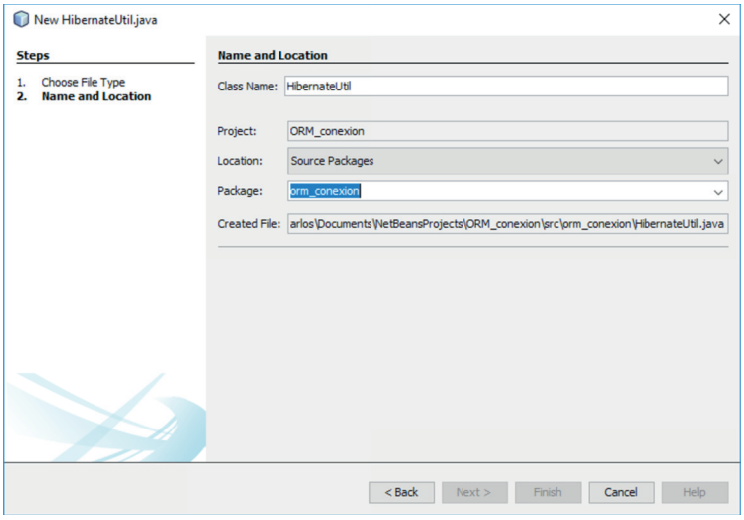


Figura 4.14
Asistente para la creación de `HibernateUtil.java`

Este es un fichero en el que se crea una clase que facilita enormemente la inicialización de Hibernate. Solo hay que indicar el nombre del fichero y el paquete en que se quiere crear.

Con esto se ha terminado de crear todo lo que necesita un programa que utilice Hibernate para persistencia de objetos. Solo falta escribir este programa en el fichero `ORM_conexion.java`. Pero antes se explicará cómo descargar y utilizar una versión reciente de Hibernate.

4.5. Descarga y uso de una versión reciente de Hibernate

Se puede comprobar que los asistentes han añadido los ficheros jar necesarios para Hibernate, pero que son de la versión 4.3. Se puede descargar la última versión estable de Hibernate de su página web. Es muy importante verificar sus prerrequisitos, especialmente en lo relativo a la versión de Java. La versión 5.3, que se va a emplear aquí, tiene como prerrequisito Java SE 8.

Una vez descargado el fichero zip y descomprimido, aparece un directorio `lib`, y dentro de él los directorios `required`, `optional` y algunos más.

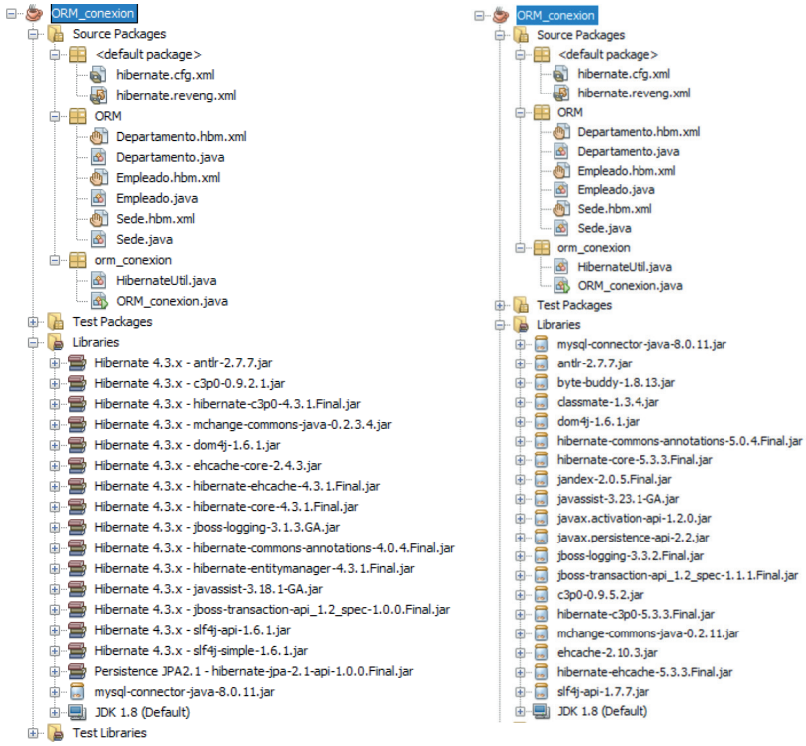


Figura 4.15
Cambios en el proyecto para usar una versión más reciente de Hibernate

Hay que reemplazar los ficheros `jar` de Hibernate añadidos por el asistente al proyecto por los de la versión más reciente. En el proyecto deben estar al menos los que vienen en el directorio `required`. Aparte de eso, se incluyen todos los equivalentes en la nueva versión a alguno de los que están en el proyecto. También se ha eliminado el *driver* para versiones anteriores de MySQL, `mysql-connector-java-5.1.23-bin.jar`, porque se usa el nuevo para la versión 8.0. En la figura 4.15 se muestra el proyecto antes y después de hacer estos cambios.

Ahora se plantea un pequeño problema, y es que no compila `HibernateUtil.java`. No se admite `AnnotationConfiguration`. Como con cualquier API, cuando se trabaja con Hibernate conviene tener a mano sus Javadocs.

Recurso web

Los Javadocs de Hibernate están disponibles en la descarga y también en Internet en las direcciones que se muestran a continuación:

http://docs.jboss.org/hibernate/orm/5.3/javadocs/
http://docs.jboss.org/hibernate/orm/4.3/javadocs

En los Javadocs de la versión más reciente de Hibernate no aparece `AnnotationConfiguration`. Pero este código se creó para la versión 4.3 de Hibernate. En los Javadocs de esta versión dice que está obsoleta (*deprecated*), y que su funcionalidad se ha pasado a `Configuration`. El problema se resuelve cambiando `AnnotationConfiguration` por `Configuration`.

Los programas de ejemplo que se hagan a partir de ahora se harán en un proyecto como este, en el fichero `ORM_conexion.java` o en otro en la misma ubicación.

4.6. Programa de ejemplo para persistencia de objetos con Hibernate

Finalmente, un programa de ejemplo que crea una sede, un departamento de esa sede y un empleado de ese departamento. Todas las operaciones se realizan en una sesión y, dentro de ella, en una transacción. La clase `HibernateUtil`, definida en `HibernateUtil.java`, permite abrir una `Session` con suma facilidad. Las clases para objetos persistentes (POJO) están en el paquete con nombre `ORM`, por lo que se prefija con `ORM` su nombre. Para crear objetos se utilizan sus creadores sin parámetros. Para asignar valores a sus atributos se utilizan métodos *setter*, y también para relacionarlos con otros objetos (por ejemplo, para asociar un empleado a su departamento). Para guardar objetos en la base de datos se utiliza el método `save()` de la sesión. Baste esto para una primera aproximación. En las secciones siguientes se explicarán los detalles de la correspondencia objeto-relacional y otros aspectos de Hibernate.

```
// Programa sencillo para ORM con Hibernate
package orm_conexion;
import org.hibernate.Session;
import org.hibernate.Transaction;
```

```
public class ORM_conexion {

    public static void main(String[] args) {

        Transaction t = null;
        try (Session s = HibernateUtil.getSessionFactory().openSession()) {

            t = s.beginTransaction();

            ORM.Sede sede = new ORM.Sede();
            sede.setNomSede("MÁLAGA");
            s.save(sede);

            ORM.Departamento depto = new ORM.Departamento();
            depto.setNomDepto("INVESTIGACIÓN Y DESARROLLO");
            depto.setSede(sede);
            s.save(depto);

            ORM.Empleado emp = new ORM.Empleado();
            emp.setDni("56789012B");
            emp.setNomEmp("SAMPER");
            emp.setDepartamento(depto);
            s.save(emp);

            t.commit();

        } catch (Exception e) {
            e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}
```

Como resultado de la ejecución del programa, este será el contenido de las tablas. Para todos los valores definidos como `auto_increment` se ha asignado valor 1, al ser la primera fila que se añade a la tabla.

sede

id_sede	nom_emp
1	MÁLAGA

departamento

id_depto	nom_depto	id_sede
1	I+D	1

empleado

dni	nom_emp	id_depto
56789012B	SAMPER	1

Actividades propuestas

4.1. Haz un programa que cree una nueva sede, dos departamentos para esta nueva sede y dos empleados para cada uno de estos departamentos. Verifica que los datos se crean correctamente comprobando el contenido de las tablas.

4.2. Ejecuta otra vez el programa de ejemplo, y verifica si se produce alguna excepción. Localiza el tipo de excepción. Cambia el programa para que este tipo de excepción se gestione de manera separada y se proporcione una información más concisa pero suficiente, en lugar de la muy prolija proporcionada por `printStackTrace()`. No se puede capturar directamente una excepción del tipo `ConstraintViolationException`. Hay que utilizar repetidamente el método

`getCause()` de `Exception` y verificar el tipo de excepción con `instanceof ConstraintViolationException`.
Haz que no puedan existir dos sedes distintas con idéntico nombre, y que no puedan existir dos departamentos distintos con idéntico nombre en una misma sede. La manera más sencilla es con índices únicos (sentencia `CREATE UNIQUE INDEX` de SQL). Verifica tu solución utilizando el programa de ejemplo inicial o pequeñas variaciones de él. Hay que introducir esta restricción en la propia base de datos, y hay que verificar que (en el caso en que se intente crear una nueva sede con el mismo nombre que una ya existente, y en el caso en que se intente crear un nuevo departamento en una sede con el mismo nombre que un departamento ya existente en esa sede) se produce una excepción y el programa la gestiona adecuadamente.

4.7. Ficheros hbm o de correspondencia de Hibernate

En este apartado se verá en detalle el contenido de los ficheros hbm (*Hibernate mapping files* o ficheros de correspondencia de Hibernate), que especifican la correspondencia entre clases de Java y tablas de una base de datos relacional. Se pondrá como ejemplo el fichero de correspondencia `departamento.hbm.xml`, que especifica la correspondencia entre la clase de Java `Departamento` y la tabla `departamento`. Pero la explicación no se limitará a lo que aparece en este fichero, sino que se pondrá todo en un contexto más amplio para explicar otras opciones y características de Hibernate.

La tabla `departamento` se definió de la siguiente manera en MySQL:

```
create table departamento(
    id_depto integer auto_increment not null,
    nom_depto char(32) not null,
    id_sede integer not null,
    primary key(id_depto),
    foreign key fk_depto_sede(id_sede) references sede(id_sede)
);
```

El POJO generado para ella, `Departamento.java`, es el siguiente:

```
/**
* Departamento generated by hbm2java
*/
@Entity
@Table(name="departamento",catalog="proyecto_orm")
public class Departamento implements java.io.Serializable {

    private Integer idDepto;
    private Sede sede;
    private String nomDepto;
    private Set empleados = new HashSet(0);

    public Departamento() {
    }
```

```
    public Departamento(Sede sede, String nomDepto) {
        this.sede = sede;
        this.nomDepto = nomDepto;
    }
    public Departamento(Sede sede, String nomDepto, Set empleados) {
        this.sede = sede;
        this.nomDepto = nomDepto;
        this.empleados = empleados;
    }

    @Id @GeneratedValue(strategy=IDENTITY)

    @Column(name="id_depto", unique=true, nullable=false)
    public Integer getIdDepto() {
        return this.idDepto;
    }

    public void setIdDepto(Integer idDepto) {
        this.idDepto = idDepto;
    }

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="id_sede", nullable=false)
    @OneToMany(fetch=FetchType.LAZY, mappedBy="departamento")
    public Sede getSede() {
        return this.sede;
    }

    public void setSede(Sede sede) {
        this.sede = sede;
    }

    @Column(name="nom_depto", nullable=false, length=32)
    public String getNomDepto() {
        return this.nomDepto;
    }

    public void setNomDepto(String nomDepto) {
        this.nomDepto = nomDepto;
    }

    @OneToMany(fetch=FetchType.LAZY, mappedBy="departamento")
    public Set getEmpleados() {
        return this.empleados;
    }

    public void setEmpleados(Set empleados) {
        this.empleados = empleados;
    }

}
```

Hibernate requiere la opción `implements java.io.Serializable`. Los atributos se declaran como privados y tienen métodos *getter* y *setter*.

Los nombres de los atributos de la clase se obtienen a partir de los nombres de los atributos de las tablas siguiendo sencillas reglas, de manera que al atributo `nom_depto` de la tabla `departamento`, por ejemplo, le corresponde el atributo `nomDepto` de la clase `Departamento`.

En la clase `Departamento` se reflejan las relaciones de uno a muchos en las que participa la tabla `departamento`, definidas mediante claves foráneas (`FOREIGN KEY` en SQL). La relación

de uno a muchos de un departamento con sus empleados se refleja en `Set empleados`, para los empleados del departamento. El nombre `empleados` es el resultado de añadir `s` al final del nombre de la tabla `empleado` para formar su plural.

PARA SABER MÁS

Reglas de nomenclatura de Hibernate

Hibernate utiliza reglas para la correspondencia entre nombres de tablas y nombres de columnas de tablas, por una parte, y nombres de clases y nombres de atributos de clases, por otra. Esto incluye la formación de los plurales de los nombres de clases para las colecciones, añadiendo *s* o *es*. El motivo para poner los nombres de las tablas en singular, si se tiene esa posibilidad, es que los plurales resultantes sean correctos. Que no lo sean, por supuesto, no impide que las clases funcionen como deben. Las reglas de nomenclatura se pueden cambiar, pero no es sencillo.

La relación de uno a muchos de una sede con sus departamentos se refleja en el atributo `Sede sede` en la clase `Departamento`, para la sede a la que pertenece el departamento.

Este es el contenido del fichero de correspondencia `departamento.hbm.xml`.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//
    EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated by Hibernate Tools 4.3.1 -->
<hibernate-mapping>
    <class name="ORM.Departamento" table="departamento" catalog="proyecto_orm"
        optimistic-lock="version">
        <id name="idDepto" type="java.lang.Integer">
            <column name="id_depto" />
            <generator class="identity" />
        </id>
        <many-to-one name="sede" class="ORM.Sede" fetch="select">
            <column name="id_sede" not-null="true" />
        </many-to-one>
        <property name="nomDepto" type="string">
            <column name="nom_depto" length="32" not-null="true" />
        </property>
        <set name="empleados" table="empleado" inverse="true" lazy="true"
            fetch="select">
            <key>
            <column name="id_depto" not-null="true" />
            </key>
            <one-to-many class="ORM.Empleado" />
        </set>
    </class>
</hibernate-mapping>
```

4.7.1. Correspondencia para las clases y atributos de clase

En el elemento `<class>` se indica la clase (`ORM.Departamento`), la tabla con la que se corresponde (`departamento`) y la base de datos en que está dicha tabla (`proyecto_ORM`).

A continuación, vienen una serie de elementos que establecen la correspondencia de los distintos atributos de la clase. Tienen algunos atributos comunes, tales como:

`name`: el nombre del atributo en la clase.
`type`: el tipo del atributo en la clase de Java. Hibernate hace corresponder a cada posible tipo para un atributo de una clase un tipo estándar de SQL.

El atributo de la tabla de la base de datos se indica con el elemento `<column>`. Este puede tener algún atributo con información adicional para algunos tipos. Para un `string`, por ejemplo, `length` indica la longitud del campo en la base de datos.

Los tipos de elementos que aparecen son:

a) `<id>` hace corresponder un atributo de la clase con uno que es clave primaria de la tabla:

Departamento.java	CREATE TABLE departamento
private Integer idDepto;	id_depto integer auto_increment not null, primary key(id_depto)

```
<id name="idDepto" type="java.lang.Integer">
    <column name="id_depto" />
    <generator class="identity" />
</id>
```

Departamento.hbm.xml

Empleado.java	CREATE TABLE empleado
private String dni;	dni char(9) not null, primary key(dni)

```
<id name="dni" type="string">
    <column name="dni" length="9"/>
    <generator class="assigned"/>
</id>
```

Empleado.hbm.xml

El valor del atributo `generator` hace referencia a una clase de Hibernate utilizada para generar valores para los campos de la clave primaria. Hay unas cuantas disponibles que deberían ser suficientes para cualquier situación. Pero si fuera necesario, se pueden crear nuevas a medida. Algunos de los valores disponibles son:

- `identity`: la columna se define en la base de datos como clave autogenerada, en el caso de MySQL con la opción `auto_increment`. La base de datos genera un valor para esta (`id_depto` de la tabla `departamento`), que se asigna al atributo correspondiente (`idDepto` de la clase `Departamento`).

- `assigned`: la aplicación debe proporcionar un valor para el campo. Este es el caso, por ejemplo, para el atributo `DNI` de la clase `Empleado`, que se corresponde con el atributo `DNI` de la tabla `empleado`.
- `sequence`: se puede utilizar con Oracle para generar un identificador numérico utilizando una secuencia de la base de datos.
- `native`: puede comportarse como `identity`, `sequence` o `hilo` (esta última opción no se verá aquí), dependiendo de la base de datos con la que se trabaje.
- `foreign`: usa el identificador de otro objeto asociado. Usado para relaciones de uno a uno, que se verán en breve.

b) `<property>` hace corresponder un atributo de la clase con uno de la tabla, sin más:

Departamento.java	CREATE TABLE departamento
private String nomDepto;	nom_depto char(32) not null

```
<property name="nomDepto" type="string">
   <column name="nom_depto" length="32" not-null="true" />
</property>
```

Departamento.hbm.xml

El atributo `not-null` se utiliza para indicar que no puede tomar valor nulo, lo que se corresponde con la opción `not null` en SQL.

4.7.2. Correspondencia para las relaciones

También debe establecerse una correspondencia para las relaciones. Es decir, también es necesario especificar cómo las relaciones entre distintos objetos se reflejan en la base de datos. Existen relaciones de uno a muchos, de uno a uno y de muchos a muchos. Hibernate permite especificar la correspondencia para todos estos tipos.

Para el ejemplo desarrollado se ha partido de un esquema relacional. Para ilustrar las relaciones se recurre a un diagrama entidad-relación (o diagrama E-R).

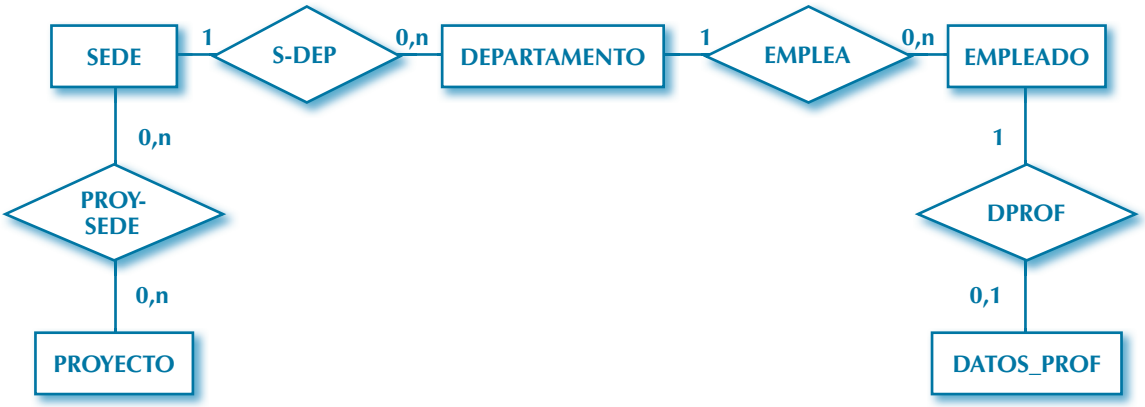


Figura 4.16
Diagrama E-R para el esquema relacional de ejemplo

RECUERDA

✓ En el módulo "Bases de Datos" de primer curso se estudian los diagramas E-R, su uso para el modelado conceptual de datos, y la manera de obtener un esquema relacional a partir de un diagrama E-R. En un diagrama E-R se pueden representar relaciones de muchos a muchos. No es así en un esquema relacional, por lo que, para poder representarlas en un esquema relacional, hay que descomponerlas previamente en dos relaciones de uno a muchos.

El diagrama E-R que se muestra en la figura 4.16 tiene relaciones de todos los tipos: de uno a muchos (entre `SEDE` y `DEPARTAMENTO` y entre `DEPARTAMENTO` y `EMPLEADO`), de uno a uno (entre `EMPLEADO` y `DATOS_PROF`) y de muchos a muchos (entre `SEDE` y `PROYECTO`).

Las relaciones entre dos clases deben definirse en Hibernate en los ficheros de correspondencia de ambas clases. Por este motivo, si a un objeto de la clase A le pueden corresponder varios de la clase B, la relación entre ambas clases se define de uno a muchos desde la clase A hacia la clase B en el fichero de correspondencia para la clase A, y de muchos a uno desde la clase B hacia la clase A en el fichero de correspondencia para la clase B.

A continuación, se explica la manera en que las relaciones de distintos tipos se han representado en los ficheros de correspondencia generados por Hibernate.

A) Relaciones de uno a muchos

`<many-to-one>` y `<one-to-many>` se usan para especificar dos relaciones recíprocas, la segunda de uno a muchos y la primera de muchos a uno. A una sede pueden corresponder varios departamentos (relación uno a muchos o *one-to-many*). Recíprocamente, varios departamentos pueden corresponder a una sede (relación muchos a uno o *many-to-one*). En última instancia, estas relaciones de uno a muchos, sus recíprocas de muchos a uno, sea entre instancias de objetos o entre filas de tablas, son una misma cosa, especificada en la tabla `departamento` con la clave foránea `fk_depto_sede(id_sede) references sede(id_sede)`, que establece que a cada fila de la tabla `departamento` le corresponde una en la tabla `sede`, determinada por el valor del atributo `id_sede`. Esto se refleja en la clase `ORM.Departamento` mediante un atributo `sede`, que indica la sede a la que pertenece el departamento, y en la clase `ORM.Sede` mediante una colección `departamentos`, que contiene los departamentos pertenecientes a la sede.

Departamento.java	CREATE TABLE departamento

`private Sede sede;`	`id_sede integer not null,`
	`foreign key fk_depto_sede(id_sede)`
	`references sede(id_sede)`

```
<many-to-one name="sede" class="ORM.Sede" fetch="select">
    <column name="id_sede" not-null="true" />
</many-to-one>
```

Departamento.hbm.xml

Sede.java	CREATE TABLE sede
private Set departamentos = new HashSet(0);	id_sede integer auto_increment not null, primary key(id_sede)

```
<set name="departamentos" table="departamento" inverse="true" lazy="true"
    fetch="select">
    <key>
        <column name="id_sede" not-null="true" />
    </key>
    <one-to-many class="ORM.Departamento" />
</set>
```

Sede.hbm.xml

El elemento `<set>` especifica el tipo de colección que se utilizará para almacenar los departamentos de una sede (`HashSet`) y cómo se pueden obtener en la base de datos (consultando la tabla `departamento` por `id_sede`, como se indica en `<key>`). `<one-to-many>` establece el tipo de objetos que se almacenarán en la colección (`ORM.Departamento`).

El atributo `lazy` toma por defecto valor `true`. Eso significa que, cuando se consulta la base de datos para obtener un objeto de tipo `ORM.Departamento`, no se obtienen los empleados del departamento en la colección `empleados`. En lugar de ello, Hibernate proporciona un proxy, que obtendrá los empleados de la base de datos solo cuando se consulte el contenido de la colección `empleados`.

El atributo `fetch` puede tomar los valores `select` y `join`. El valor elegido solo supone una diferencia cuando se consulta la base de datos para obtener varios departamentos. `select` significa que para cada departamento se hará una consulta de SQL para obtener sus empleados. `join` significa que en algunos casos (no siempre, depende del mecanismo utilizado para obtener los datos de los departamentos) se hará una única consulta de SQL con `left outer join`, lo que puede mejorar el rendimiento.

B) Relaciones uno a uno

Se especifica en el elemento `<one-to-one>` en los ficheros de correspondencia `Empleado.hbm.xml` y `EmpleadoDatosProf.hbm.xml`. Es una relación de uno a uno y no de uno a muchos porque la clave foránea se define desde la clave primaria de una tabla (`empleado_datos_prof`). No es completamente simétrica: a un objeto de la clase `EmpleadoDatosProf` le corresponde exactamente uno de la clase `Empleado` (cardinalidad 1), pero a uno de la clase `Empleado` puede no corresponderle ninguno de la clase `EmpleadoDatosProf` (cardinalidad 0,1). Esta asimetría se refleja en los ficheros de correspondencia, cuyos contenidos más relevantes se muestran a continuación:

Empleado.java	CREATE TABLE empleado
private EmpleadoDatosProf empleadoDatosProf;	dni char(9) not null, primary key(dni)

```
<id name="dni" type="string">
    <column name="dni" length="9" />
    <generator class="assigned" />
</id>
<one-to-one name="empleadoDatosProf" class="ORM.EmpleadoDatosProf">
</one-to-one>
```

Empleado.hbm.xml

EmpleadoDatosProf.java	CREATE TABLE empleado_datos_prof
private Empleado empleado;	dni char(9) not null, primary key(dni), foreign key fk_empleado_datosprof_ empl(dni) references empleado(dni)

```
<id name="dni" type="string">
    <column name="dni" length="9" />
    <generator class="foreign">
        <param name="property">empleado</param>
    </generator>
</id>
<one-to-one name="empleado" class="ORM.Empleado" constrained="true">
</one-to-one>
```

Empleado.hbm.xml

Las asimetrías se dan en `EmpleadoDatosProf.hbm.xml`, porque es en la tabla `Empleado_datos_prof` donde se define, desde su clave primaria, la clave foránea a la otra tabla `Empleado`. Esto significa que no puede existir un objeto de clase `EmpleadoDatosProf` sin un objeto correspondiente de clase `Empleado`, y se refleja con `constrained=true`. Además, el objeto de clase `EmpleadoDatosProf` debe tener el mismo identificador que el correspondiente de clase `Empleado`, y esto se refleja con `<generator class="foreign">`, lo que significa que el identificador (`dni`) se tomará del objeto de esta otra clase.

C) *Relaciones de muchos a muchos*

En los ficheros de correspondencia de Hibernate se pueden definir relaciones de este tipo mediante el elemento `<many-to-many>`. No se explica aquí cómo especificar la correspondencia para relaciones de muchos a muchos con Hibernate. Cuando se generan los ficheros de correspondencia a partir de un esquema relacional, no aparecen este tipo de relaciones porque, como se ha comentado, en un esquema relacional no se pueden representar directamente.

Cuando se obtiene un esquema relacional a partir de un diagrama E-R, una relación de muchos a muchos se descompone en dos relaciones de uno a muchos con una nueva entidad. En este caso, la relación de muchos a muchos `PROY-SEDE` entre `SEDE` y `PROYECTO` se descompone en dos relaciones de uno a muchos de `SEDE` y de `PROYECTO` a una nueva entidad, y en el esquema relacional se introducen tablas para `SEDE` (tabla `sede`), para `PROYECTO` (tabla `proyecto`) y para la nueva entidad (tabla `proyecto_sede`), y una restricción de clave foránea (`FOREIGN KEY`) para cada relación de uno a muchos.

Como recapitulación, se incluye un diagrama E-R similar al anterior, solo que con la relación de muchos a muchos entre `SEDE` y `PROYECTO` transformada en dos relaciones de uno a muchos con una nueva entidad (a la que corresponde la clase `ProyectoSede`). Se indican nombres de clases en lugar de entidades. Se indican junto a cada clase los atributos que se emplean para representar las relaciones con otras clases, los que son colecciones con letra en cursiva. Estos atributos son privados y no se puede acceder directamente a ellos, pero existe un método *getter* para recuperar cada uno (`getSede()` para `sede`, `getEmpleados()` para `empleados`, etc), y un método *setter* para modificarlos.

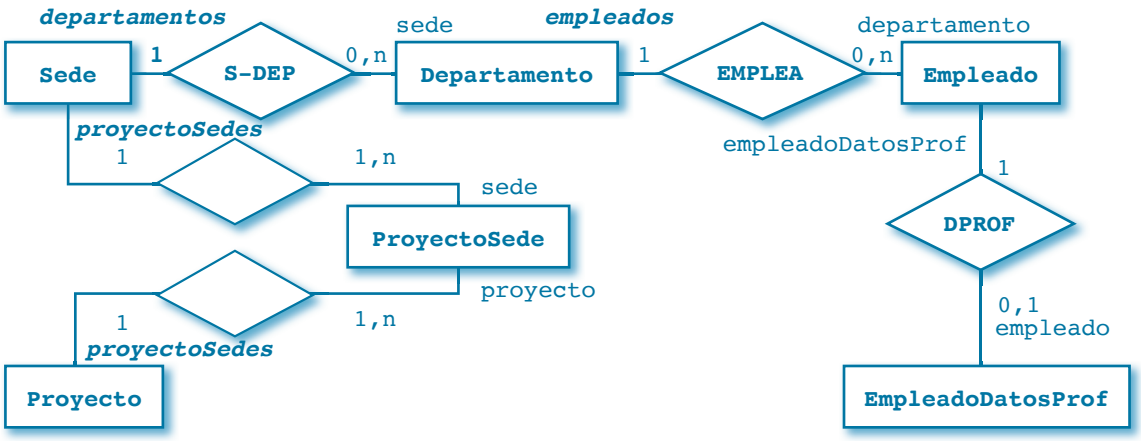


Figura 4.17
Atributos de clase para representación de las relaciones entre clases

Una vez recuperado un objeto persistente, estos atributos permiten acceder fácilmente a otros objetos relacionados, y crear nuevas asociaciones con otros, sin necesidad de operar con la base de datos, solo trabajando con clases de Java. Por ejemplo: a partir del DNI de un empleado se puede obtener su departamento con `getDepartamento()`, y a partir de él todos sus empleados con `getEmpleados()`, y para cada uno de ellos sus datos profesionales con `getEmpleadoDatosProf()`. También se podrían establecer nuevas relaciones entre objetos o modificar las que ya existen utilizando los correspondientes métodos *setter*.

TOMA NOTA

No es recomendable utilizar los métodos *setter* para asignar directamente un valor a las colecciones. Es mejor obtener la colección con el método *getter* y manejarla con los métodos propios de las colecciones tales como `add()` para añadir, etc. En breve se pondrán algunos ejemplos.

Si no se trabaja con Hibernate o una herramienta similar, habría que utilizar consultas SQL para obtener los departamentos de una sede, o los empleados de un departamento, y también para modificar estos datos. Para ello es necesario tener presente la estructura de la base de datos. Esta operativa no se presta a un desarrollo orientado a objetos. Los programas de este tipo tienden a utilizar muchas y a menudo complejas sentencias de SQL. La desventaja que puede tener utilizar Hibernate o *frameworks* similares es el menor rendimiento de las operaciones de consulta o modificación de datos. Para aplicaciones de envergadura que manejan bases de datos complejas y con grandes volúmenes de datos, es necesario optimizar las operaciones con la base de datos. Pero el margen de maniobra es menor, y para optimizar bien el rendimiento se necesita un conocimiento avanzado del *framework*. Lo que se gana en claridad y mantenibilidad, y las ventajas de utilizar la programación orientada a objetos, puede ir en detrimento del grado de control sobre la base de datos, de la flexibilidad y de la eficiencia en las operaciones con la base de datos. De todas formas, como se verá más adelante, Hibernate permite también realizar consultas directamente con SQL.

En los siguientes apartados se explica la manera de gestionar las relaciones de uno a muchos y de uno a uno.

4.8. Manejo de relaciones de uno a muchos entre objetos persistentes

Para reflejar este tipo de relaciones entre dos clases se utilizan atributos en ambas clases. En una de ellas el atributo es sencillo y en la otra es una colección.

El siguiente programa crea una nueva sede y tres departamentos de ella, y por último muestra los datos de la sede y de sus departamentos, e itera sobre la colección que en un objeto de tipo `Sede` permite acceder a sus departamentos. El programa no muestra los departamentos creados para la sede si no se utiliza `refresh()`, aunque estos se graban correctamente en la base de datos. La asignación de la sede a los departamentos en objetos de tipo `ORM.Departamento` no se refleja automáticamente en la colección de los departamentos para la sede en el objeto de tipo `ORM.Sede`. Con `s.refresh(sede)` se actualiza `sede` con los contenidos actuales de la base de datos. Es posible que los departamentos no se listen en el mismo orden en el que se han creado. Con Hibernate se puede establecer un orden para los objetos de una colección, pero una vez establecido, no se pueden recuperar en otro. Para el segundo y tercer departamento, se usan constructores que permiten especificar directamente sus datos, en lugar de usar el constructor sin parámetros y después métodos *setter,* como para el primero. Se ha hecho de las dos formas para ilustrar ambas formas de hacer lo mismo.

```
// Programa que crea una sede y luego varios departamentos asociados
package orm_colecciones_e_iteradores;

import java.util.Iterator;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ORM_colecciones_e_iteradores {

       public static void main(String[] args) {

           Transaction t = null;
           try (Session s = HibernateUtil.getSessionFactory().openSession()){
               t = s.beginTransaction();
               ORM.Sede sede = new ORM.Sede("TERUEL");
               s.save(sede);

               ORM.Departamento depto = new ORM.Departamento();
               depto.setNomDepto("I+D");
               depto.setSede(sede);
               s.save(depto);

               depto = new ORM.Departamento(sede, "MARKETING");
               s.save(depto);

               depto = new ORM.Departamento(sede, "QA");
               s.save(depto);

               s.refresh(sede);

               Iterator itDeptos = sede.getDepartamentos().iterator();
               System.out.println("Nueva sede [" + sede.getIdSede() + "] (" +
               sede.getNomSede()+")");
               System.out.println("----------------");

               while(itDeptos.hasNext()) {
                   ORM.Departamento unDepto = (ORM.Departamento) itDeptos.
                       next();
```

```
                System.out.println("Depto: [" + unDepto.getIdDepto() + "] ("
                    +
                unDepto.getNomDepto()+")");
            }

            t.commit();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}
```

El siguiente programa hace lo mismo, pero de distinta manera. Primero crea una nueva sede. Después crea uno a uno los nuevos departamentos y los añade a la colección que almacena los departamentos para la nueva sede. Para este programa no hace falta usar `refresh()` para que se muestre correctamente toda la información.

```
// Programa que crea una sede y varios departamentos que añade a su lista

package orm_colecciones_e_iteradores_2;

import java.util.Iterator;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class ORM_colecciones_e_iteradores_2 {

    public static void main(String[] args) {

        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();

            ORM.Sede sede = new ORM.Sede("VALENCIA");
            s.save(sede);

            ORM.Departamento depto1=new ORM.Departamento(sede, "I+D");
            s.save(depto1);
            sede.getDepartamentos().add(depto1);

            ORM.Departamento depto2=new ORM.Departamento(sede, "MARKETING");
            s.save(depto2);
            sede.getDepartamentos().add(depto2);

            ORM.Departamento depto3=new ORM.Departamento(sede, "QA");
            s.save(depto3);
            sede.getDepartamentos().add(depto3);

            Iterator<ORM.Departamento> itDeptos =
                sede.getDepartamentos().iterator();

            System.out.println("Nueva sede [" + sede.getIdSede() + "] (" +
                sede.getNomSede() + ")");
            System.out.println("-----------------");

            while(itDeptos.hasNext()) {
                ORM.Departamento unDepto = (ORM.Departamento) itDeptos.next();
                System.out.println("Depto: [" + unDepto.getIdDepto() + "] (" +
```

```
                unDepto.getNomDepto() + ")");
            }

            t.commit();
        } catch (Exception e) {
            if (t != null) {
                e.printStackTrace(System.err);
                t.rollback();
            }
        }
    }
}
```

4.9. Manejo de relaciones de uno a uno entre objetos persistentes

Para reflejar este tipo de relaciones entre dos clases se utilizan atributos sencillos en ambas clases. Hay una relación de este tipo entre las clases `Empleado` y `EmpleadoDatosProf`.

El siguiente programa de ejemplo crea un nuevo empleado y le asigna sus datos profesionales. El nuevo empleado se asigna a un departamento ya existente con identificador (`id_depto`) 1. Para obtener este departamento, se utiliza el método `get()`. Este método aún no se ha visto, pero su uso aquí es fácil de comprender. Es interesante que, para que se puedan recuperar los datos profesionales a partir del objeto de tipo `Empleado` mediante `getEmpleadoDatosProf()`, debe antes confirmarse la transacción con `commit()`. Esto es porque, al estar definido el generador para el identificador de `Empleado` como `assigned`, se posterga la grabación de los datos hasta entonces, y al coincidir con el identificador de `EmpleadoDatosProf` (generador de tipo `foreign`), solo se puede obtener cuando se haya grabado en la base de datos. Para obtener los datos profesionales a partir del objeto `emp` de tipo `Empleado` ha sido necesario `s.refresh(emp)`.

```
// Pro datos profesionales

package ORM_rel_uno_a_uno;

import java.math.BigDecimal;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ORM_rel_uno_a_uno {

    public static void main(String[] args) {

        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();

            ORM.Departamento depto = s.get(ORM.Departamento.class, 1);

            ORM.Empleado emp = new ORM.Empleado();
            emp.setDni("76543210S");
            emp.setDepartamento(depto);
            emp.setNomEmp("SILVA");
            s.save(emp);
```

```
            ORM.EmpleadoDatosProf datosProf = new ORM.EmpleadoDatosProf();
            datosProf.setEmpleado(emp);
            datosProf.setCategoria("A");
            datosProf.setSueldoBrutoAnual(new BigDecimal(52000.0));
            s.save(datosProf);

            t.commit();

            s.refresh(emp);
            System.out.println("Categoría del nuevo empleado: " +
                emp.getEmpleadoDatosProf().getCategoria() +
            ", DNI: " + emp.getEmpleadoDatosProf().getDni()
            );
        } catch (Exception e) {
        e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}
```

4.10. Sesiones y estados de los objetos persistentes

Ya se ha aprendido cómo se puede establecer la correspondencia objeto-relacional para clases de Java, y también para relaciones entre clases. Se han creado objetos transitorios y se han almacenado en la base de datos como objetos persistentes con Hibernate. Ahora es el momento de explicar en detalle el ciclo de vida de los objetos persistentes, los distintos estados en los que pueden estar, y las operaciones que permiten recuperar objetos persistentes de la base de datos, modificarlos y grabar estas modificaciones de nuevo en la base de datos.

Una clase para la que se establecen correspondencias mediante Hibernate se denomina *clase persistente,* y una instancia de una clase persistente, *objeto persistente*.

La sesión (interfaz `org.hibernate.Session`) es un componente esencial en Hibernate. Una sesión se construye sobre una conexión a la base de datos. Las transacciones creadas por los programas anteriores son transacciones de sesión, y pueden incluir varias transacciones de base de datos. Una sesión puede mejorar el rendimiento de las consultas en objetos persistentes utilizando una caché. La apertura de sesiones es muy rápida utilizando un *pool* de conexiones.

Una sesión, junto con un gestor de entidades asociado, constituye un *contexto de persistencia*. Un gestor de entidades (interfaz `javax.persistence.EntityManager`) lleva el control de los cambios que se realizan sobre objetos persistentes. La interfaz `Session` pertenece a la API propia de Hibernate, mientras que la interfaz `EntityManager` pertenece a la de JPA. Se puede obtener un `EntityManager` a partir de una `Session` y viceversa; existe, pues, un puente entre ambas API. Casi todo lo que se puede hacer utilizando una de estas API se puede hacer utilizando la otra, si bien es cierto que la interfaz de Hibernate ofrece más posibilidades, o al menos una manera más directa de hacer muchas cosas. Aquí, por brevedad y simplicidad, se explicará solo la interfaz `Session` de la API de Hibernate.

Los cambios que se realizan sobre objetos persistentes se reflejan, en última instancia, en la base de datos en tanto en cuanto están asociados a un contexto de persistencia.

Los objetos persistentes pueden estar en diferentes estados:

1. *Transitorio (transient)*. El objeto se acaba de crear y no está asociado con ningún contexto de persistencia. No está grabado en la base de datos. No tiene por qué tener un identificador único asignado, pero podría tenerlo si se ha especificado para su identificador el generador `assigned` (valor "`assigned`" para el atributo `generator` del elemento `id`), y se le ha asignado un valor.
2. *Gestionado (managed) o persistente (persistent)*. El objeto tiene un identificador asociado y está asociado con un contexto de persistencia, y está grabado en la base de datos. Cualquier cambio que se realice en el objeto se reflejará en la base de datos. No inmediatamente, sino en respuesta a determinadas operaciones sobre la sesión, típicamente cuando se cierra con `close()`.
3. *Separado (detached)*. El objeto tiene un identificador asociado, pero ya no está asociado con un contexto de persistencia, bien porque el contexto se ha cerrado, bien porque se desvinculó el objeto del contexto.
4. *Eliminado (removed)*. El objeto tiene un identificador asociado y está asociado con un contexto de persistencia, pero está pendiente de ser borrado de la base de datos.

Un objeto persistente puede cambiar de estado merced a distintas operaciones realizadas sobre él dentro de una sesión. Algunos de estos cambios de estado conllevan operaciones en la base de datos para grabar, modificar o borrar la representación del objeto en ella.

Los objetos en estados transitorio y separado, una vez que ya no están asociados a un contexto de persistencia, pueden ser eliminados por el recolector de basura de *Java (garbage collector)*, desde el mismo momento en que no exista ninguna referencia a ellos.

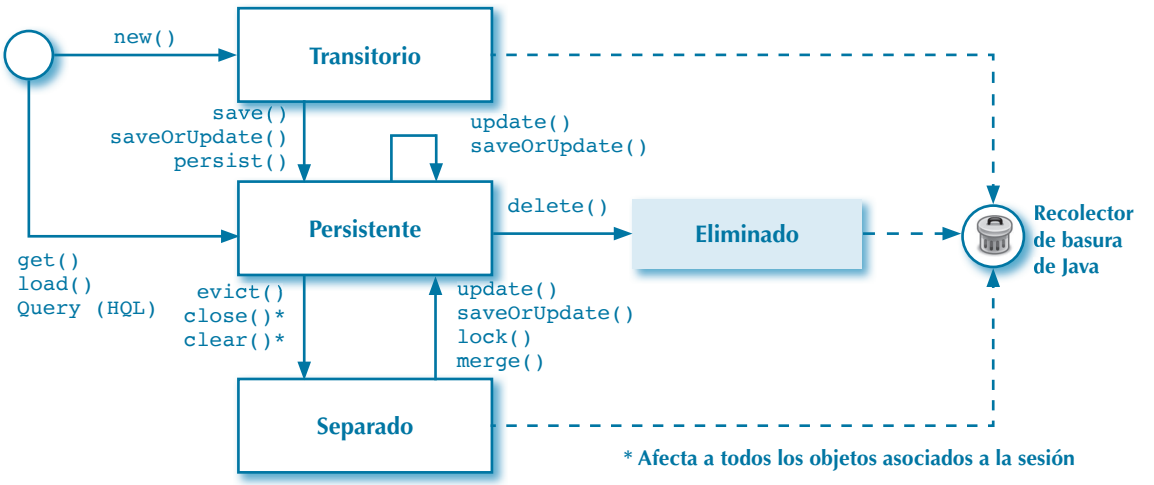


Figura 4.18
Diagrama de estados de un objeto persistente

En el resto de este apartado se ofrece una breve explicación de las operaciones que permiten crear objetos persistentes, cambiar su estado y realizar diversas operaciones con ellos, limitadas, como ya se ha comentado, a la API de Hibernate. Esta es una explicación general para proporcionar una primera introducción a las posibilidades de Hibernate. Por tanto, se omiten muchos detalles, muchas opciones de algunas operaciones, muchos casos particulares, muchas implicaciones que pueden ser relevantes en situaciones concretas, y muchas posibilidades de una herramienta tan potente y compleja como es Hibernate.

4.10.1. De transitorio a persistente con `save()`, `saveOrUpdate()` y `persist()`

Un objeto se puede pasar de estado transitorio a persistente con los métodos `save()`, `saveOrUpdate()` y `persist()` de `Session`. Si el identificador de la clase se genera automáticamente (es el caso con un valor `identity`, `sequence` o `native`, y algunos otros, para el atributo `generator`), no hay que asignar un valor al identificador único para la clase. En ese caso, `save()` y `saveOrUpdate()` devuelven el identificador una vez que se ha grabado el objeto en la base de datos, lo que se hace inmediatamente, mientras que `persist()` no devuelve este identificador, y en algunos casos se puede grabar el objeto posteriormente, lo que contribuye a mejorar el rendimiento. `persist()` solo puede utilizarse dentro de una transacción, al contrario que `save()` y `saveOrUpdate()`, que pueden utilizarse tanto dentro como fuera de una transacción.

```
ORM.Sede sede = new ORM.Sede();
sede.setNomSede("CUENCA");
Integer idSede = (Integer) s.save(sede);
```

En caso contrario, como es el caso con el valor `assigned` para el atributo `generator`, el programa debe asignar un valor al identificador, y la grabación en la base de datos no se realizará inmediatamente.

```
ORM.Empleado emp = new ORM.Empleado();
emp.setDni("78901234X");
emp.setNomEmp("NADALES");
emp.setDepartamento(depto);
s.save(emp);
```

Si se intenta hacer persistente un objeto transitorio creado con `new()` y ya existe un objeto persistente de la misma clase y con idéntico identificador, `save()` generará una excepción, dado que no puede haber dos objetos persistentes con el mismo identificador. De hecho, en alguna de las actividades propuestas anteriormente, se ha pedido verificar esto mismo, y gestionar la excepción que se producía.

El método `saveOrUpdate()` funciona igual que `save()` para un objeto transitorio creado con `new` si no hay ningún objeto persistente con el mismo identificador. Si lo hay, entonces modifica dicho objeto. Dicho de otra forma, `saveOrUpdate()` es a `save()` en Hibernate lo que `INSERT INTO … ON DUPLICATE KEY UPDATE` es a `INSERT INTO` en SQL.

```
ORM.Empleado emp = new ORM.Empleado();
emp.setDni("78901234X");
emp.setNomEmp("NADALES");
emp.setDepartamento(depto);
s.saveOrUpdate(emp);
```

Actividad propuesta 4.3

Modifica el primer programa de ejemplo para que, si ya existe un empleado con el DNI, le asigne un nombre determinado, y si no existe, cree un empleado con dicho nombre. Utiliza `saveOrUpdate()`.

Ya se ha comentado que `persist()` no devuelve un identificador y que, en el caso de que el identificador de la clase se genere automáticamente, en algunos casos se podría postergar la grabación del objeto en la base de datos. Esto puede mejorar el rendimiento si esta operación se realiza dentro de una transacción con muchas operaciones o que se prolongue mucho en el tiempo. Si no es el caso, no hay ninguna diferencia en la práctica entre utilizar `save()` y `persist()`, aparte de que con la primera se podrá disponer siempre inmediatamente de los identificadores generados automáticamente.

```
ORM.Sede sede = new ORM.Sede();
sede.setNomSede("PALENCIA");
s.persist(sede);
// sede.getIdSede() podría devolver null porque todavía no se haya grabado
// la sede en la base de datos. En cambio, con s.save() la sede se habría
// grabado ya en la base de datos y getIdSede() siempre devolvería un valor.
```

```
ORM.Empleado emp = new ORM.Empleado();
emp.setDni("78901234X");
emp.setNomEmp("NADALES");
emp.setDepartamento(depto);
s.persist(emp);
// emp.getDni() devolvería el DNI: "78901234X"
```

4.10.2. Obtención de un objeto persistente con `get()` y `load()`

`get()` y `load()` permiten obtener un objeto persistente e indicar su clase y el valor para su identificador único. Se diferencian en que, si no existe un objeto persistente de la clase indicada y con el identificador indicado, `get()` devuelve un valor `null`, mientras que `load()` lanza una excepción de tipo `ObjectNotFoundException`.

Hasta ahora no se había visto cómo recuperar un objeto persistente grabado en la base de datos. Más adelante se verá cómo obtener objetos persistentes sin necesidad de conocer su identificador único, sino especificando criterios de selección.

Existen versiones de ambos métodos con parámetros adicionales para especificar diversas opciones, tales como tipo de bloqueo sobre el objeto persistente, tiempo de espera máximo antes de desistir de conseguir el bloqueo, y otras más, pero no se explican aquí.

```
ORM.Empleado emp = s.get(ORM.Empleado.class, "78901234X");
if(emp==null) {
    System.out.println("No existe empleado");
}
else {
    System.out.println("El nombre del empleado es "+emp.getNomEmp());
}
```

Se suele utilizar `load()` cuando se sabe con seguridad que el objeto existe, o solo puede no existir en una situación anómala. El siguiente fragmento de código muestra un escenario de uso típico, en el que no se gestionan excepciones de tipo `ObjectNotFoundException`.

```
ORM.Sede sede = s.load(ORM.Sede.class, 12);
System.out.println("El nombre de la sede es " + sede.getNomSede());
```

4.10.3. De persistente a eliminado con `delete()`

Se puede borrar un objeto persistente con `delete()`.

```
ORM.Sede sede = s.get(ORM.Sede.class, 15);
if(sede != null) {
    s.delete(sede);
}
```

4.10.4. De persistente a separado con `evict()`, `close()` y `clear()`

Estos métodos permiten separar objetos de sesiones, es decir, pasarlos a estado separado o *detached*. Con `evict(Object obj)` se puede separar un objeto. Con `close()` y `clear()` se separan todos los objetos asociados a la sesión. Con `close()` se grabarán los cambios realizados sobre ellos en la base de datos, mientras que con `clear()` no se grabarán. Se pueden seguir consultando y modificando los datos de un objeto en estado separado, pero las modificaciones no se reflejarán en la base de datos, a menos que se vuelva a asociar el objeto a una sesión, que se puede hacer.

```
s.evict(sede); // Separa objeto sede de sesión
```

```
s.close(); // Graba todos los objetos asociados a la sesión y los separa
```

```
s.clear(); // Separa todos los objetos asociados a la sesión, sin grabarlos
```

4.10.5. De separado a persistente con `update()`, `saveOrUpdate()`, `lock()` y `merge()`

Un objeto separado puede volver a hacerse persistente con `lock()`. Al llamar a este método hay que indicar un modo de bloqueo. Hay unos cuantos modos de bloqueo.

Recurso web

www

Modos de bloqueo con Hibernate:

https://docs.jboss.org/hibernate/stable/orm/userguide/html_single/Hibernate_User_Guide.html#locking-LockMode

Se puede también volver a hacer persistente un objeto separado con `update()` y `saveOrUpdate()`. También con `merge()`, que no hace persistente el objeto que se le pasa, sino que actualiza los contenidos de otro objeto persistente con el mismo identificador en la sesión si existe, o crea uno nuevo si no existe. El resultado, en cualquier caso, es que los datos del objeto que se le pasa se incorporan a un objeto persistente asociado a la sesión, y se devuelve este objeto. El objeto que se le pasa a `merge()` sigue siendo un objeto separado y se puede descartar.

4.11. Lenguajes de consulta HQL y JPQL

HQL son las siglas de Hibernate Query Language. Es un lenguaje inspirado en SQL. Tiene sentencias `SELECT`, `INSERT`, `UPDATE` y `DELETE`, similares a las del lenguaje SQL.

JPQL es un subconjunto de HQL incluido en la especificación de JPA.

Recurso web

El lenguaje HQL es similar al lenguaje SQL. Para más información sobre la sintaxis de los lenguajes HQL y JPQL se puede consultar el siguiente enlace, con numerosos ejemplos, y muy asequible con unos conocimientos básicos de SQL (sección de la guía de usuario de Hibernate).

https://docs.jboss.org/hibernate/stable/orm/userguide/html_single/Hibernate_User_Guide.html#hql-statement-types

Los IDE NetBeans y Eclipse tienen intérpretes de HQL que permiten ejecutar directamente sentencias de HQL. En Netbeans, se accede a él pulsando con el botón derecho del ratón sobre el fichero de configuración `hibernate.cfg.xml` y seleccionando la opción "Run HQL Query". No es muy intuitivo, pero tiene su lógica que el intérprete de HQL se lance desde ahí, ya que en este fichero se define la conexión con la base de datos. En la imagen se puede ver el resultado de una sencilla consulta

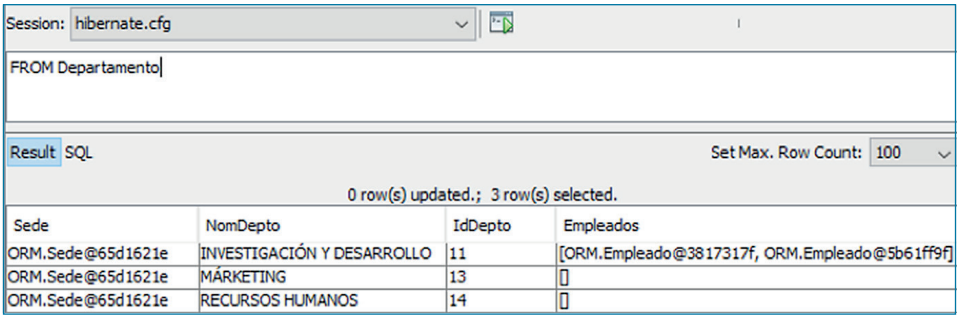


Figura 4.19
Resultado de una consulta en el intérprete de HQL de Netbeans

Tiene un aspecto muy similar al resultado de una consulta con SQL, pero con algunas particularidades. La columna Sede muestra objetos en lugar de datos de un tipo elemental. Cada objeto viene identificado por el nombre de su clase (`ORM.Sede`) y un identificador único. La columna Empleados muestra listas de empleados, vacías todas excepto para un departamento con dos empleados, que se muestran de la misma manera: con el nombre de su clase (`ORM.Empleado`) y un identificador único.

4.11.1. La interfaz `Query`

La API de Hibernate permite utilizar sentencias de HQL mediante la interfaz `Query`. Esto abre muchas posibilidades. Con lo visto hasta ahora solo se podía acceder a objetos persistentes mediante sus identificadores únicos. Pero hay que tener cuidado con las operaciones de actualización y borrado masivo.

TOMA NOTA

Conviene tener muy en cuenta una advertencia al principio de la sección de la guía de usuario de Hibernate dedicada a HQL y a JPQL, cuyo enlace se ha proporcionado antes (se traduce del inglés).

Hay que tener cuidado cuando se ejecuta una sentencia UPDATE o DELETE. "Debe actuarse con precaución cuando se ejecutan operaciones masivas de actualización o borrado porque pueden provocar inconsistencias entre la base de datos y las entidades dentro del contexto de persistencia activo. En general, las operaciones masivas de actualización y borrado solo deberían realizarse dentro de una transacción en un nuevo contexto de persistencia, o antes de recuperar o acceder a entidades cuyo estado pueda verse afectado por dichas operaciones".

— Sección 4.10 de la especificación JPA 2.0

La especificación JPA (http://download.oracle.com/otn-pub/jcp/persistence-2.0-fr-eval-oth-JSpec/persistence-2_0-final-spec.pdf) se refiere solo a JQPL. Pero se entiende que este aviso es de aplicación para HQL, dado que JPQL es un subconjunto suyo, y que está incluido en la documentación de Hibernate y en una sección dedicada conjuntamente a HQL y JPQL.

Existe una interfaz `Query` tanto en la API de Hibernate como en la de JPA. En este apartado se hablará solo de la API de Hibernate. Se puede obtener una instancia de `Query` a partir de una instancia de `Session`, mediante el método `createQuery(String sentenciaHQL)`.

RECUERDA

✓ `createQuery` devuelve una instancia de la interfaz `org.hibernate.query.Query`. No hay que confundirla con la interfaz `org.hibernate.Query`, que está obsoleta *(deprecated)* y que está previsto eliminar para la versión 6 de Hibernate. Hay que tener esto en cuenta cuando se consulte la documentación para `Query`, para hacerlo en el lugar correcto.

El funcionamiento de la interfaz `Query` es análogo al de `PreparedStatement` de JDBC. Una `Query` puede tener parámetros que se pueden identificar por nombre además de por posición. Por ejemplo: `SELECT nomDepto FROM Departamento WHERE idDepto=:dep`.

La interfaz `Query`, de la que se ha venido hablando hasta ahora, es, hablando con propiedad, `Query<R>`, es decir, una interfaz genérica con un parámetro de tipo `R`, que es el tipo de los objetos con los que trabaja la `Query`.

En el siguiente cuadro se incluyen algunos de sus principales métodos. Muchos devuelven la propia `Query`, de manera que se pueden encadenar varias llamadas al mismo método, lo que da como resultado un código muy compacto. Por ejemplo:

```
Query q = s.createQuery(
   "FROM Departamento WHERE idSede=:idSede AND nomDepto LIKE ':nomDepto'"
).
setParameter("idSede", 12).
setParameter("nomDepto", "%ADMIN%").
setReadOnly().
setFirstResult(10).setMaxResults(5);
List<ORM.Departamento> listaDep = q.getResultList();
```

CUADRO 4.1
Métodos de `Query <R>`

Método	Funcionalidad
`List<R> getResultList()`	Devuelve la lista de resultados de una consulta HQL.
`R getSingleResult()`	Devuelve un único resultado para una consulta. Este método puede lanzar varias excepciones, entre ellas `NoResultException` cuando la consulta no devuelve ningún resultado y `NonUniqueResultException` cuando devuelve más de uno.
`int executeUpdate()`	Ejecuta la sentencia de tipo `UPDATE` o `DELETE` y devuelve el número de objetos afectados.
`Query<R> setParameter(...)`	Asigna un valor a un parámetro de la sentencia. Este método tiene muchas variantes que se pueden consultar en los Javadocs de la API de Hibernate. En general, tienen un parámetro de tipo `Object` para especificar el valor.
`<P> Query <R> setParameterList(...)`	Este método tiene diversas variantes que permiten asignar a un parámetro una lista de valores. Es útil cuando se especifica en una sentencia de HQL una restricción del tipo `valor IN :listaValores`. La lista de valores se puede proporcionar en un *array* o en diversos tipos de colecciones.
`Query<R> setReadOnly(boolean readOnly)`	Especifica que los resultados recuperados serán solo para lectura, y no para modificación. Conviene usar este método si ese es el caso.

[.../...]

CUADRO 4.1 (CONT.)

`Query<R> setFirstResult(int posInicial)` `Query<R> setMaxResults(int maxResult)`	Especifican el primer resultado que recuperar de entre los obtenidos por la consulta, siendo cero el primero, y el número máximo de resultados que recuperar.

En el resto de esta sección se verán varios programas de ejemplo que utilizan distintos tipos de sentencias de HQL.

A) Sentencias SELECT

El siguiente programa muestra los datos de los departamentos cuyo nombre contenga una cadena de caracteres introducida por teclado. La consulta tiene un parámetro `patNombre` al que se le asigna valor con una llamada a `setParameter()` justo antes de lanzar la consulta. Como no se van a modificar los objetos persistentes recuperados, se utiliza `setReadOnly()`. La consulta se realiza con `getResultList()`, que devuelve la lista de resultados.

```
// Programa que recupera varios objetos persistentes con una consulta en HQL
package orm_query_consulta;

import java.util.List;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import org.hibernate.Session;
import org.hibernate.query.Query;

public class ORM_query_consulta {

    public static void main(String[] args) {

        try (Session s = HibernateUtil.getSessionFactory().openSession()) {

            System.out.println("Introducir texto para buscar por nombre: ");
            BufferedReader br =
                new BufferedReader(new InputStreamReader(System.in));
            String nomDepto = br.readLine();

            Query q = s.createQuery(
                "FROM Departamento WHERE nomDepto LIKE :patNombre"
            ).setParameter("patNombre", "%" + nomDepto + "%").
                setReadOnly(true);

            List<ORM.Departamento> listaDep =
                (List<ORM.Departamento>) q.getResultList();
            for (ORM.Departamento unDepto: listaDep) {
                System.out.println("Depto: [" + unDepto.getIdDepto() + "] (" +
                    unDepto.getNomDepto() + ")");
            }
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 4.4

Escribe un programa que utilice el método `setParameterList()` para mostrar los departamentos de varias sedes, y que muestre para cada uno el identificador y el nombre de la sede y del departamento. Es necesario consultar documentación acerca de la interfaz `Query` de la API de Hibernate.

Por supuesto, se podrían modificar los objetos obtenidos mediante una consulta de HQL y grabar los cambios realizados, lo que debería hacerse dentro de una transacción.

El siguiente programa muestra los datos del empleado con mayor sueldo, que recupera mediante una consulta con `getSingleResult()`. Si hubiera más de uno, o si no existieran datos profesionales para ningún empleado, se muestra un mensaje de error específico.

```
// Programa que recupera un único objeto persistente con getSingleResult()
package ORM_Query_consulta_getSingleResult;

import org.hibernate.Session;
import org.hibernate.query.Query;
import javax.persistence.NoResultException;
import org.hibernate.NonUniqueResultException;

public class ORM_Query_consulta_getSingleResult {

    public static void main(String[] args) {

        try(Session s = HibernateUtil.getSessionFactory().openSession()) {

            Query q = s.createQuery("FROM EmpleadoDatosProf dp WHERE
                dp.sueldoBrutoAnual>=ALL(SELECT sueldoBrutoAnual FROM
                EmpleadoDatosProf)").setReadOnly(true);

            ORM.EmpleadoDatosProf datosProf =
                (ORM.EmpleadoDatosProf) q.getSingleResult();

            System.out.println("Empleado [" + datosProf.getDni() + "] "
                + "(" + datosProf.getEmpleado().getNomEmp() + ") "
                + " de departamento: "
                + datosProf.getEmpleado().getDepartamento().getNomDepto()
                + " de sede: "
                + datosProf.getEmpleado().getDepartamento().getSede().
                    getNomSede()
                + ", con sueldo: " + datosProf.getSueldoBrutoAnual()
            );

            } catch (NoResultException e) {
                System.err.println("ERROR: No hay datos profesionales para
                    ningún empleado");
            } catch (NonUniqueResultException e) {
                System.err.println("ERROR: Hay más de un empleado con el salario
                    máximo");
        } catch (Exception e) {
                e.printStackTrace(System.err);
        }
    }
}
```

El siguiente programa de ejemplo obtiene objetos de tipo `Empleado` y los modifica. Si para un `Empleado` no existen datos profesionales, se crea un nuevo objeto de clase `EmpleadoDatosProf` y se asocian mutuamente ambos objetos, porque entre ellos hay una relación de uno a uno. Si existen, hay que modificar el objeto de clase `EmpleadoDatosProf` que ya existe. Como sueldo, se asigna una cantidad aleatoria entre 20.000 € y 80.000 €. El objeto de clase `EmpleadoDatosProf` se graba con `saveOrUpdate()` porque podría existir o no en la base de datos, y el de clase `Empleado`, con `update()`. Siempre que las modificaciones sobre los objetos se puedan realizar con una sentencia `UPDATE` de HQL, es preferible hacerlo así. Si no, habrá que recuperar los objetos con una sentencia `SELECT` de HQL y hacer las modificaciones en un bucle, como se hace en este programa.

```
// Programa que recupera objetos mediante una consulta de HQL y los modifica

package ORM_query_mas_modificacion;

import java.util.List;
import java.util.Random;
import java.math.BigDecimal;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.query.Query;

public class ORM_rel_uno_a_uno {

    public static void main(String[] args) {

        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {

            t = s.beginTransaction();
            Query q = s.createQuery("FROM Empleado");
            List<ORM.Empleado> listaEmp = (List<ORM.Empleado>)
                q.getResultList();
            Random rand = new Random();
            for (ORM.Empleado unEmp: listaEmp) {
                ORM.EmpleadoDatosProf datosProf = unEmp.getEmpleadoDatosProf();
                if (datosProf == null) {
                    datosProf = new ORM.EmpleadoDatosProf();
                    datosProf.setEmpleado(unEmp);
                    unEmp.setEmpleadoDatosProf(datosProf);
                }
                datosProf.setCategoria("A");
                datosProf.setSueldoBrutoAnual(
                    BigDecimal.valueOf(2000000 +
                        rand.nextInt(6000000)).movePointLeft(2)
                );
                s.saveOrUpdate(datosProf);
                s.update(unEmp);
            }

            t.commit();

        } catch (Exception e) {
            e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}
```

B) Sentencias UPDATE y DELETE

Se pueden ejecutar sentencias de tipo UPDATE de HQL con `executeUpdate()`. El siguiente programa cambia el nombre del departamento llamado Investigación y Desarrollo por I+D. Después, borra el departamento con nombre Actividades Paranormales. Además, escribe el número de objetos afectados por cada una de estas operaciones. Una característica de HQL es que el operador `LIKE` no distingue entre mayúsculas y minúsculas. Si hubiera algún empleado en uno de los departamentos que se quiere borrar, no se realizaría la operación. Por último, hay que insistir en que, como se indicó al principio, operaciones de este tipo que pueden realizar cambios con muchos objetos podrían dar como resultado inconsistencias si hubiera objetos afectados por ellas en el contexto de persistencia en el que se realizan. Y al igual que cuando se utiliza SQL, hay que extremar la precaución con este tipo de sentencias, y especificar la cláusula `WHERE` de manera apropiada para no modificar o borrar objetos que realmente no se quiere modificar o borrar, porque los resultados podrían ser catastróficos.

```
// Ejecución de sentencias UPDATE y DELETE de HQL con executeQuery()
package orm_executeUpdate;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.query.Query;

public cºlass orm_executeUpdate {

    public static void main(String[] args) {

        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();

            Query q = s.createQuery("UPDATE Departamento SET nomDepto='I+D'
                WHERE nomDepto='Investigación y Desarrollo'");
            int numObj = q.executeUpdate();
            System.out.println(numObj + " objetos cambiados.");

            q = s.createQuery(
                "DELETE Departamento WHERE nomDepto='ACTIVIDADES
                    PARANORMALES'");
            numObj = q.executeUpdate();
            System.out.println(numObj + " objetos borrados.");

            t.commit();
        } catch (Exception e) {
            e.printStackTrace(System.err);
            if (t != null) {
                t.rollback();
            }
        }
    }
}
```

4.12. Correspondencia de la herencia

Una característica fundamental de los lenguajes orientados a objetos es la relación de jerarquía entre clases. Cuando una clase se define como subclase de otra, hereda sus atributos y métodos.

Por esta razón, a este tipo de relación se le denomina también relación de herencia. En Java, una clase se declara como subclase de otra mediante la palabra clave `extends`.

La principal dificultad que plantea la herencia para la persistencia es la vinculación dinámica *(late binding)*. Esta consiste en que, si un objeto se define como perteneciente a una clase de la que existen subclases, solo se puede saber si ese objeto pertenece a la propia clase o a una de sus subclases en tiempo de ejecución, cuando se ejecuta el programa, y no antes, en tiempo de compilación. Cuando se quiera hacer persistente ese objeto, podría ser de esa misma clase o de una subclase suya. Por ejemplo, para esta sección se utiliza una superclase `Publicacion` y dos subclases suyas `Libro` y `Revista`. Si se declara un objeto de clase `Publicacion`, en el momento de almacenarlo en base de datos, podría ser un objeto de clase `Libro` o `Revista`, además de `Publicacion`, y en cada caso tendría un conjunto de atributos diferente. Tendrá los de `Publicación`, pero podría, además, tener los propios de `Libro` o de `Revista`.

A continuación, se muestra la definición de estas clases, que han sido escritas y no generadas automáticamente, y cumplen los requisitos para los POJO que impone Hibernate. Como es práctica habitual, los constructores de las subclases invocan el constructor de su superclase con `super()`.

```
// Fichero Publicacion.java
package ORM;
public class Publicacion implements java.io.Serializable {
    private Integer idPub;
    private String nomPub;
    public Publicacion() {
    }
    public Publicacion(String nomPub) {
        this.nomPub = nomPub;
    }
    public Integer getIdPub() {
        return this.idPub;
    }
    public void setIdPub(Integer idPub) {
        this.idPub = idPub;
    }
    public String getNomPub() {
        return this.nomPub;
    }
    public void setNomPub(String nomPub) {
        this.nomPub = nomPub;
    }
}

// Fichero Libro.java
package ORM;
public class Libro extends Publicacion implements java.io.Serializable {
    private String isbn;
```

```
    private String autor;

    public Libro() {
    }

    public Libro(String titulo, String isbn, String autor) {
        super(titulo);
        this.isbn = isbn;
        this.autor = autor;
    }

    public String getIsbn() {
        return this.isbn;
    }
    public void setIsbn(String isbn) {

        this.isbn = isbn;
    }

    public String getAutor() {
        return this.autor;
    }

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


// Fichero Revista.java
package ORM;
public class Revista extends Publicacion implements java.io.Serializable {

    private String issn;

    public Revista () {
    }

    public Revista(String nombre, String issn) {
        super(nombre);
        this.issn = issn;
    }

    public String getIssn() {
        return this.issn;
    }

    public void setIssn(String issn) {
        this.issn = issn;
    }
}
```

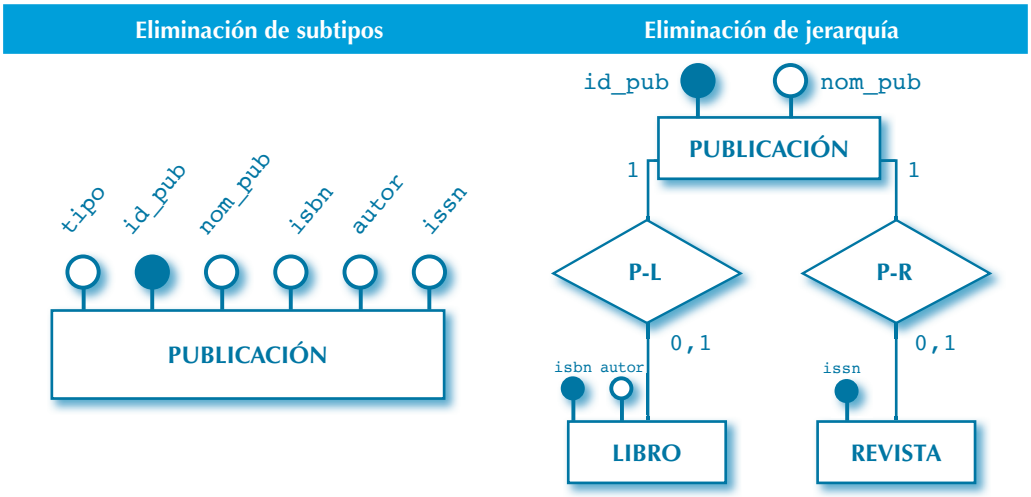
Al igual que sucede con las relaciones de muchos a muchos, las relaciones jerárquicas:

- No se pueden representar directamente en un esquema relacional.
- Se pueden representar en un diagrama E-R.
- Se pueden hacer corresponder con un esquema relacional con Hibernate.

Por tanto, se ilustrará un ejemplo de correspondencia de relaciones jerárquicas entre clases mediante un diagrama E-R.

Figura 4.20
Diagrama E-R con herencia

En el módulo Bases de Datos de primer curso se estudia la obtención de esquemas relacionales a partir de diagramas E-R. Las relaciones jerárquicas no se pueden representar en esquemas relacionales, pero existe un conjunto de posibles transformaciones que se pueden aplicar a los diagramas E-R para eliminarlas. No siempre se pueden aplicar todas. Cada una tiene ventajas e inconvenientes que hay que sopesar para tomar la decisión más adecuada para cada caso particular. No se entrará en más detalles aquí. Solo se explicarán dos posibles transformaciones y cómo se puede establecer la correspondencia mediante ficheros de correspondencia de Hibernate.



Los atributos de los subtipos pasan al supertipo. Las relaciones de los subtipos con otras entidades son ahora con el supertipo. Suele ser necesario, o al menos conveniente, introducir un nuevo atributo, llamado discriminante, para indicar el subtipo (aquí el discriminante introducido es `tipo`).

Se establecen nuevas relaciones de uno a uno entre supertipo y subtipos, que se traducen en el esquema relacional como claves foráneas.

Figura 4.21
Distintas formas de eliminar una relación de herencia en un diagrama E-R

En esencia es lo mismo hablar de entidades relacionadas por una relación jerárquica de generalización/especialización en un diagrama E-R que de una jerarquía de clases. Hibernate permite implementar fácilmente los planteamientos anteriores para la correspondencia entre una jerarquía de clases y un esquema relacional, que se puede obtener directamente del diagrama E-R que resulta de eliminar la jerarquía.

En los subapartados siguientes se mostrará cómo se puede establecer la correspondencia con Hibernate de cada una de las formas anteriores, y se indicará la definición de las tablas y los ficheros de correspondencia hbm.

El programa de ejemplo que se muestra a continuación funcionará sin cambios con cualquiera de los planteamientos anteriores para la correspondencia. Crea un objeto de cada una de las clases `Publicacion`, `Libro` y `Revista` que graba en la base de datos como persistentes. Para cada proyecto hay que indicar el nombre del paquete y de la clase apropiados, de acuerdo al nombre del proyecto.

```
// Programa que crea un objeto persistente de cada clase de una jerarquía

package (...);

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

public class (...) {

    public static void main(String[] args) {

        Transaction t = null;
        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            t = s.beginTransaction();

            ORM.Publicacion pub = new ORM.Publicacion("FOLLETO INDEFINIDO");
            s.save(pub);
            Integer idPub = pub.getIdPub();

            ORM.Libro libro = new ORM.Libro("CORRESPONDENCIA OBJETO-
                RELACIONAL", "9789901234567", "LOMAS, ANTONIO");
            s.save(libro);
            Integer idLibro = libro.getIdPub();

            ORM.Revista rev = new ORM.Revista("PERSISTENCIA DE OBJETOS",
                "6789012X05");
            s.save(rev);
            Integer idRev = rev.getIdPub();

            pub = (ORM.Publicacion) s.load(ORM.Publicacion.class, idPub);
            libro = (ORM.Libro) s.load(ORM.Libro.class, idLibro);
            rev = (ORM.Revista) s.load(ORM.Revista.class, idRev);

            System.out.println("Publicación [" + pub.getIdPub() + "]" + pub.
                getNomPub());
            System.out.println("Libro [" + libro.getIdPub() + "] (" + libro.
                getNomPub() + "; " + libro.getIsbn() + ";" + libro.getAutor() +
                ")");
            System.out.println("Revista [" + rev.getIdPub() + "] (" + rev.
                getNomPub() + "; " + rev.getIssn() + ")");

            t.commit();
        } catch (Exception e) {
        e.printStackTrace(System.err);
        if (t != null) {
            t.rollback();
        }
        }
    }
}
```

4.12.1. Eliminación de subtipos (una tabla para la jerarquía)

Se utiliza una sola tabla en la que se añaden todos los atributos del tipo y los subtipos y un campo discriminante llamado `tipo`. Todos los atributos de los subtipos deben admitir valores nulos

en la base de datos, pero, dependiendo del valor del discriminante, determinados atributos no deberían tenerlos. Para mayor seguridad, se añade una restricción adicional para controlar que `tipo` tiene un valor válido y que, para cada posible valor, no se admitan valores nulos en los atributos obligatorios para el subtipo correspondiente.

```
create table publicacion(
    id_pub integer auto_increment not null,
    nom_pub varchar(50) not null,
    tipo char(3) not null,
    isbn char(13),
    autor varchar(40) not null,
    issn char(10),
    primary key(id_pub),
    constraint check_subtipos check(tipo='pub' or (tipo='lib' and not
        isnull(isbn) and not isnull(autor)) or (tipo='rev' and not
        isnull(issn)))
);
```

El fichero de correspondencia `Publicacion.hbm.xml` sería el siguiente. Se especifica el campo discriminante con `<discriminator>`, y se especifica su valor para la superclase además de para las subclases. Las subclases se definen con `<subclass>`. Hibernate se encarga de asignar el valor apropiado a este campo, y de interpretarlo correctamente.

```
<?xml version="1.0" encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//
    EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ORM.Publicacion" table="publicacion" catalog="proyecto_orm"
        discriminator-value="pub">
        <id name="idPub" type="java.lang.Integer">
            <column name="id_pub"/>
            <generator class="identity"/>
        </id>
        <discriminator column="tipo" type="string"/>
        <property name="nomPub" type="string">
            <column name="nom_pub" length="50"/>
        </property>
        <subclass name="ORM.Libro" discriminator-value="lib">
            <property name="isbn" type="string">
                <column name="isbn" length="13"/>
                </property>
            <property name="autor" type="string">
            <column name="autor" length="40"/>
            </property>
        </subclass>
        <subclass name="ORM.Revista" discriminator-value="rev">
            <property name="issn" type="string">
                <column name="issn" length="10"/>
            </property>
        </subclass>
    </class>
</hibernate-mapping>
```

Debe añadirse la siguiente línea en el fichero de configuración `hibernate.cfg.xml`:

```
<mapping resource="ORM/Publicacion.hbm.xml"/>
```

El proyecto se puede crear en Netbeans de la siguiente forma:

1. Antes de nada, crear la tabla en la base de datos `proyecto_orm`.
2. Crear un proyecto para una aplicación de Java estándar (Java application).
3. Crear el fichero de configuración de Hibernate, `hibernate.cfg.xml`, utilizando el *wi*zard o asistente, para la conexión `conexion_ORM`.
4. Crear un paquete (*Java pack*age) con nombre ORM, e incluir en él (*New Java Class*…) los POJO `Publicacion.java, Libro.java` y `Revista.java`.
5. Crear el fichero de correspondencia `Publicacion.hbm.xml` en el paquete ORM, junto a `Publicacion.java`.Validarlo, pulsando con el botón derecho del ratón y seleccionando la op*ción* `Validate XML`.
6. Modificar el fichero de configuración hibernate.cfg.xml para añadir una línea `<mapping resource="ORM/Publicacion.hbm.xml"/>`. NetBeans no deja editarlo directamente, pero se puede añadir en la sección "Mappings", con "Add…", y especificando el valor `ORM/Publicacion.hbm.xml` para `resource`.
7. Crear un fichero `HibernateUtil.java` en el paquete que contiene el fichero creado por NetBeans para la clase principal (la que contiene el método `main()`). Este fichero debe ser igual que los que se han usado, con el mismo nombre, para todos los programas hasta ahora. Se puede copiar de un proyecto anterior.
8. Modificar el fichero para la clase principal, creado automáticamente por NetBeans, para incluir los contenidos del programa de ejemplo anterior.
9. Por último, eliminar del proyecto los ficheros JAR incluidos automáticamente para una versión antigua de Hibernate e incluir los de la nueva versión de Hibernate, y que se han venido incluyendo hasta ahora en todos los proyectos.

Una vez ejecutado el programa de ejemplo, asumiendo que la tabla `publicacion` estaba vacía inicialmente, sus contenidos serán:

id_pub	nom_pub	tipo	isbn	autor	issn
1	FOLLETO INDEFINIDO	pub	NULL	NULL	NULL
2	CORRESPONDENCIA OBJETO-RELACIONAL	lib	9789901234567	LOMAS, ANTONIO	NULL
3	PERSISTENCIA DE OBJETOS	rev	NULL	NULL	6789012X05

Debería, además, garantizarse la unicidad del ISBN para los libros y del ISSN para las revistas. Esto se podría hacer creando índices únicos para la tabla por estos campos:

```
CREATE UNIQUE INDEX i_publicaciones_isbn ON publicaciones(isbn);
CREATE UNIQUE INDEX i_publicaciones_issn ON publicaciones(issn);
```

4.12.2. Una tabla por subclase (eliminación de la jerarquía)

Se utiliza una tabla por clase, y las tablas para cada subclase tienen la misma clave primaria que la de la superclase, e incluyen una clave foránea hacia la tabla para la superclase. Es una solución muy flexible y elegante. Su único inconveniente, si acaso, es que se necesita una clave foránea en cada subclase, y para obtener los datos de un objeto de una subclase se necesita hacer una consulta en SQL relacionando las tablas para la subclase y la superclase.

```
create table publicacion(
    id_pub integer auto_increment not null,
    nom_pub varchar(50) not null,
    primary key(id_pub)
);
create table libro(
    id_pub integer not null,
    isbn char(13) not null,
    autor varchar(40),
    primary key(id_pub),
    constraint fk_libro_publicacion foreign key(id_pub) references
        publicacion(id_pub)
);
create table revista(
    id_pub integer not null,
    issn char(10) not null,
    primary key(id_pub),
    constraint fk_revista_publicacion foreign key(id_pub) references
        publicacion(id_pub)
);
```

En cuanto al fichero de correspondencia `Publicacion.hbm.xml`, las subclases se definen con `<joined-subclass>` en lugar de con `<subclass>`, y no hay campo discriminante. Por lo demás es prácticamente igual.

```
<?xml version="1.0" encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ORM.Publicacion" table="publicacion" catalog="proyecto_orm">
        <id name="idPub" type="java.lang.Integer">
            <column name="id_pub"/>
            <generator class="identity"/>
        </id>
        <property name="nomPub" type="string">
            <column name="nom_pub" length="50"/>
        </property>
        <joined-subclass name="ORM.Libro">
            <key column="id_pub"/>
            <property name="isbn" type="string">
                <column name="isbn" length="13"/>
            </property>
            <property name="autor" type="string">
                <column name="autor" length="40"/>
            </property>
```

```
        </joined-subclass>
            <joined-subclass name="ORM.Revista">
            <key column="id_pub"/>
            <property name="issn" type="string">
                <column name="issn" length="10"/>
            </property>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

Una vez ejecutado el programa de ejemplo, asumiendo que las tablas estaban vacías inicialmente, sus contenidos serán:

publicacion

id_pub	nom_pub
1	FOLLETO INDEFINIDO
2	CORRESPONDENCIA OBJETO-RELACIONAL
3	PERSISTENCIA DE OBJETOS

libro

id_pub	isbn	autor
2	9789901234567	LOMAS, ANTONIO

revista

id_pub	issn
3	6789012X05

De nuevo, debería garantizarse la unicidad del ISBN para los libros y del ISSN para las revistas. Esto se podría hacer creando índices únicos:

```
CREATE UNIQUE INDEX i_libros_isbn ON libros(isbn);
CREATE UNIQUE INDEX i_revistas_issn ON revistas (issn);
```

4.13. Consultas con SQL

Con Hibernate también se pueden realizar consultas directamente en SQL. Conviene, de todas formas, no abusar de esta posibilidad y recurrir a ella solo en casos muy justificados. Por ejemplo, cuando supone un aumento del rendimiento muy importante, cuando se requiere una ordenación particular de los resultados, o para consultas muy complejas que no es posible realizar en HQL, o si ya se dispone de la consulta en SQL y utilizarla ahorra tiempo.

El siguiente programa realiza una consulta en SQL y obtiene los resultados como una lista de *arrays* de objetos, que acto seguido muestra en pantalla. Hibernate ofrece muchas más posibilidades, algunas muy interesantes, pero no vale la pena entretenerse aquí en ellas.

```
// Programa que realiza una consulta en SQL y muestra los resultados
package ORM_Query_SQL;

import java.util.List;
import org.hibernate.Session;
```

```
public class ORM_Query_SQL {

    public static void main(String[] args) {

        try(Session s = HibernateUtil.getSessionFactory().openSession()) {
            List<Object[]> empleados = s.createNativeQuery("SELECT e.dni, e.id_
                depto, e.nom_emp, dp.categoria FROM empleado e LEFT OUTER JOIN
                empleado_datos_prof dp ON dp.dni=e.dni").list();
            for (Object[] objetos: empleados) {
                System.out.println("Empleado [dni:" + (String) objetos[0]
                    + ", id_depto: " + (Integer) objetos[1]
                    + ", nom_emp: " + (String) objetos[2]
                    + ", categoria: " + (String) objetos[3]
                    + "]");
            }
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Hibernate realiza la consulta de SQL mediante JDBC y consulta los metadatos del `ResultSet` obtenido para averiguar el tipo de cada columna. Para evitar que tenga que hacer esto, o simplemente para explicitar los nombres y los tipos de cada una de las columnas que se espera obtener, lo que redunda en un código de programa más legible y mantenible, se puede utilizar `addScalar()`. Con ello, la consulta se haría de esta forma:

```
import org.hibernate.type.IntegerType;
import org.hibernate.type.StringType;
(...)
List<Object[]> empleados = s.createNativeQuery("SELECT e.dni, e.id_depto,
    e.nom_emp, dp.categoria FROM empleado e LEFT OUTER JOIN empleado_datos_
    prof dp ON dp.dni=e.dni")
    .addScalar("dni", StringType.INSTANCE)
    .addScalar("id_depto", IntegerType.INSTANCE)
    .addScalar("nom_emp", StringType.INSTANCE)
    .addScalar("categoria", StringType.INSTANCE).list()
```

Resumen

- La persistencia de objetos en bases de datos relacionales plantea un conjunto de problemas que se conocen en conjunto como "desfase objeto relacional".
- ORM *(object-relational mapping)*, o "correspondencia objeto-relacional", consiste en el establecimiento de una correspondencia entre clases definidas en un lenguaje de programación orientado a objetos y tablas de una base de datos relacional, y en el uso de mecanismos para que las modificaciones sobre los objetos se registren en la base de datos y, a la inversa, para que se pueda recuperar en objetos la información registrada en la base de datos. La correspondencia objeto-relacional hace posible la persistencia de objetos en bases de datos puramente relacionales.

- Hibernate es un *framework* para ORM en lenguaje Java distribuido bajo licencia LGPL de GNU. Fue la primera herramienta para correspondencia objeto-relacional que logró amplia aceptación y, hoy en día, sigue siendo la más importante. Existe un *porting* de Hibernate para.NET, llamado NHibernate.
- Además de tener su propia API nativa, Hibernate es una implementación certificada de JPA, el estándar de Java para ORM, que es parte de la especificación Java EE.
- Hibernate permite establecer la correspondencia no solo para objetos individuales, sino también para las relaciones entre objetos, ya sean de uno a muchos, de uno a uno o de muchos a muchos. Las relaciones se reflejan en los objetos mediante atributos sencillos y mediante colecciones.
- Hibernate proporciona varios mecanismos alternativos para establecer la ORM para un conjunto de tablas relacionadas por la relación de herencia.
- Con Hibernate, la correspondencia se puede establecer mediante ficheros hbm (*Hibernate mapping files* o "ficheros de correspondencia de Hibernate") o mediante anotaciones de JPA.
- Una clase para la que se ha definido la correspondencia objeto-relacional es una clase persistente.
- El concepto de sesión es fundamental en Hibernate. Los objetos persistentes lo son en tanto en cuanto están asociados a una sesión. Hibernate sigue la pista de los cambios realizados sobre objetos persistentes asociados a una sesión mediante un gestor de entidades asociado a la sesión y, llegado el momento, se encarga de que esos cambios se reflejen en la base de datos, de manera que se mantenga siempre la consistencia entre el contenido de los objetos persistentes en memoria y en la base de datos. Una sesión y un gestor de entidades asociados constituyen un contexto de persistencia.
- Hibernate tiene su propio lenguaje para manejar objetos persistentes, HQL, similar a SQL, con la diferencia de que maneja objetos y sus propiedades en lugar de filas y columnas de tablas de bases de datos relacionales. JPQL es un subconjunto de HQL y parte del estándar JPA.

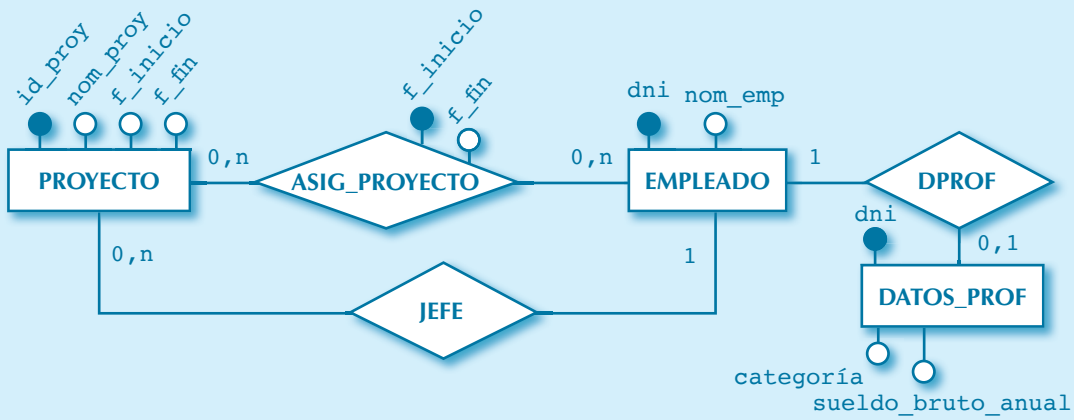
Ejercicios propuestos

Para la realización de estos ejercicios puede ser necesario consultar la documentación de la API de Java SE 8 (https://docs.oracle.com/javase/8/docs/api/) o de la de Hibernate (http://docs.jboss.org/hibernate/orm/5.3/javadocs/).

Para algunos de los ejercicios debe utilizarse un esquema relacional para representar los proyectos que desarrolla una empresa, sus empleados y las asignaciones de empleados a proyectos. Se da el diagrama entidad-relación para describir las entidades y las relaciones entre ellas, y las sentencias SQL necesarias para crear las tablas en MySQL. No hay que crear todas las tablas y clases desde el principio, sino a medida que vayan siendo necesarias para realizar las actividades. De esa manera, se podrá desarrollar la correspondencia objeto-relacional paso a paso, y centrar la atención en cada momento en cada aspecto que desarrollar.

Para desarrollar los componentes de la aplicación hay que utilizar los asistentes siempre que sea posible, y revisar los ficheros generados, y hacer cambios manualmente si es preciso.

Se puede utilizar otra base de datos relacional distinta de MySQL, preferiblemente Oracle.



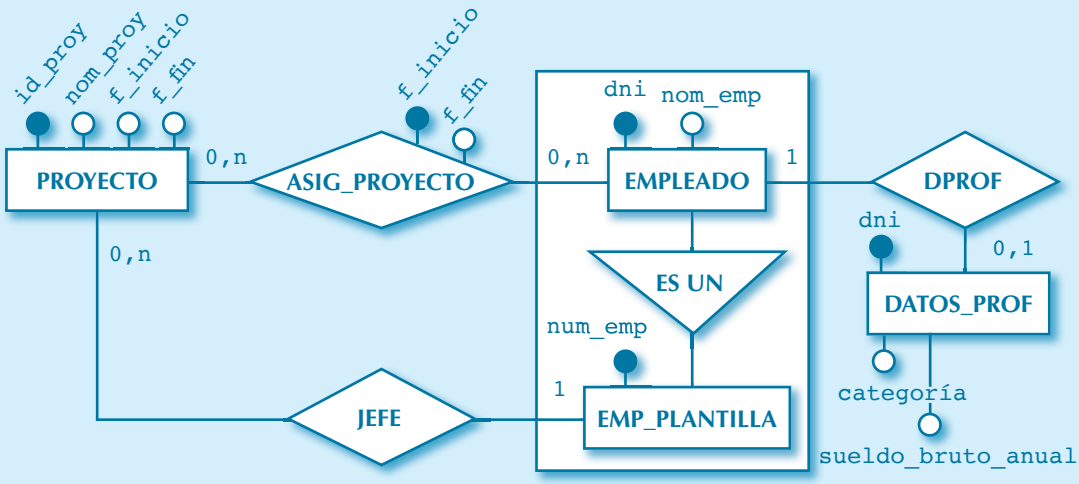
Precisiones adicionales:

- La fecha de inicio es obligatoria para proyectos y para asignaciones de empleados a proyectos. Si la fecha de fin es nula, significa que el proyecto no ha concluido. Si no es nula y está en el pasado, significa que el proyecto ha concluido.
- El campo `dni` puede representar también un NIE, es decir, un número de identificación para extranjeros. Por brevedad se le ha llamado sencillamente `dni`.

```
create table empleado(
    dni char(9) not null,
    nom_emp varchar(32) not null,
    primary key(dni)
);
create table proyecto(
    id_proy integer auto_increment not null,
    nom_proy varchar(32) not null,
    f_inicio date not null,
    f_fin date,
    dni_jefe_proy char(9) not null,
    primary key(id_proy),
    foreign key fk_proy_jefe(dni_jefe_proy) references
       empleado(dni)
);
create table asig_proyecto(
    dni_emp char(9),
    id_proy integer not null,
    f_inicio date not null,
    f_fin date,
    primary key(dni_emp, id_proy, f_inicio),
```

```
    foreign key f_asig_emp(dni_emp) references empleado(dni),
    foreign key f_asig_proy(id_proy) references proyecto(id_proy)
);
create table datos_prof(
    dni char(9) not null,
    categoria char(2) not null,
    sueldo_bruto_anual decimal(8,2),
    primary key(dni),
    foreign key fk_datosprof_empl(dni) references empleado(dni)
);
```

1. Crea una base de datos en un servidor de bases de datos para las siguientes actividades, llamada `orm_gestion_proyectos`. Crea un usuario sobre la misma base de datos, al que hay que otorgarle los permisos apropiados, como se ha hecho en ejemplos anteriores. Crea una conexión a la base de datos desde el IDE, y verifica que funciona correctamente.
2. Crea las tablas `proyecto` y `empleado`. Crea los POJO y los ficheros de correspondencia utilizando los asistentes que proporciona el IDE. Crea un programa de prueba que cree varios empleados y varios proyectos, y que asigne a cada proyecto su jefe de proyecto. Verifica que no se puede crear un proyecto si no está informado su jefe de proyecto. Verifica que no se puede crear un proyecto si no está informada su fecha de inicio.
3. Crea todo lo necesario para la correspondencia de la relación de muchos a muchos entre proyectos y empleados (incluyendo –y empezando por– la tabla `asig_proyecto`). Crea un programa de prueba que cree varios proyectos y varios empleados y asigne al menos a un empleado a varios proyectos y al menos a un proyecto varios empleados. Verifica que no se puede crear una asignación de un empleado a un proyecto si no se asigna una fecha de inicio.
4. Crea la tabla `datos_prof` y todo lo necesario para la correspondencia de uno a uno entre empleados y datos profesionales. Crea un programa de prueba que genere un nuevo empleado y le asigne sus datos profesionales, y a un empleado ya existente, identificado por su DNI.
5. Crea un programa que, para varias asignaciones cualesquiera de empleados a proyectos, les asigne como fecha de fin una fecha en el pasado (por ejemplo, el día anterior a la fecha actual).
6. Crea un programa que, utilizando una sentencia de HQL, muestre los detalles de todas las asignaciones de empleados a proyectos.
7. Lo mismo que en la actividad anterior, pero solo para asignaciones vigentes. Es decir, no se deben mostrar asignaciones cuya fecha de inicio esté en el futuro ni asignaciones cuya fecha de fin esté informada y esté en el pasado.
8. Considera la distinción de empleados en plantilla frente a empleados no en plantilla, que se contratan para proyectos determinados. Solo los empleados en plantilla pueden ser jefes de proyecto. El siguiente esquema E-R describe este cambio.



Se creará una nueva tabla `emp_plantilla` para los empleados en plantilla, incluyendo un campo `dni` y con una clave foránea por este campo hacia `empleado`. A los empleados en plantilla se les asigna un número de empleado que sirve como identificación para ellos, pero no será una tabla autogenerada. La relación entre empleados y empleados en plantilla es una relación jerárquica, que se implementa en Java con una nueva clase `EmpPlantilla` subclase de la actual `Empleado`. Se pide hacer una copia de la base de datos y una copia del proyecto para ORM desarrollado hasta ahora, que funcionará con la nueva base de datos, y desarrollar la actividad a partir de esta copia. Se pide hacer los cambios necesarios en la estructura de la base de datos, hacer las conversiones de datos necesarias, hacer los cambios necesarios en el proyecto, tanto en las clases como en los ficheros de correspondencia, y crear un programa de prueba en el método `main()` de una nueva clase.

El primer paso será hacer los cambios de estructura y la conversión de datos en la nueva base de datos. Todos los empleados existentes se considerarán empleados en plantilla. Una vez creada la nueva tabla `emp_plantilla`, se tiene que insertar en ella una fila por empleado, con una sentencia `INSERT INTO emp_plantilla(…) SELECT … FROM empleado`. La clave foránea de `proyecto` a `empleado` debe cambiarse a `emp_plantilla`. Para ello debe borrarse la antigua clave foránea y crear la nueva. Eso es todo en lo que respecta a la conversión de datos. Después habrá que crear la nueva clase `EmpPlantilla` como subordinada de `Empleado`, y la nueva correspondencia para la clase `Empleado` en conjunto con `EmpleadoPlantilla`.

El programa de prueba debe: crear nuevos empleados en plantilla y no en plantilla, crear proyectos nuevos y asignarles como jefe de proyecto empleados en plantilla tanto existentes como empleados creados en el mismo programa, y por último asignar empleados a los nuevos proyectos creados, tanto de plantilla como no de plantilla, y tanto existentes como creados en el mismo programa.

ACTIVIDADES DE AUTOEVALUACIÓN

1. El desfase objeto-relacional es:
   - ☐ *a)* El conjunto de dificultades conceptuales y técnicas que plantean las bases de datos de objetos.
   - ☐ *b)* La sobrecarga del sistema que supone la persistencia de objetos en bases de datos relacionales, que se evita con la persistencia en bases de datos de objetos.
   - ☐ *c)* El conjunto de dificultades que plantea la persistencia de objetos en bases de datos relacionales.
   - ☐ *d)* El conjunto de correspondencias que es necesario establecer entre clases y tablas de un esquema relacional para hacer posible la persistencia de objetos en bases de datos relacionales.

2. ORM es:
   - ☐ *a)* El conjunto de extensiones que se pueden añadir a una base de datos relacional para convertirla en una base de datos objeto-relacional, en las que es posible, con ciertas limitaciones, la persistencia de objetos.
   - ☐ *b)* Un conjunto de extensiones que permiten implementar una base de datos de objetos sobre una base de datos relacional.
   - ☐ *c)* Un estándar para persistencia de objetos en bases de datos relacionales.
   - ☐ *d)* La correspondencia objeto-relacional, que hace posible la persistencia de objetos en bases de datos puramente relacionales.

3. Hibernate es:
   - ☐ *a)* Una implementación de JPA (Java Persistence Architecture).
   - ☐ *b)* Un *framework* para ORM para el lenguaje Java.
   - ☐ *c)* Una base de datos objeto-relacional.
   - ☐ *d)* Nada de lo anterior.

4. Las tablas utilizadas para ORM con Hibernate:
   - ☐ *a)* No es necesario que tengan clave primaria.
   - ☐ *b)* Deben tener una clave primaria simple, es decir, consistente en una única columna.
   - ☐ *c)* Deben tener una clave en la que solo intervengan valores numéricos y autogenerados.
   - ☐ *d)* Deben tener clave primaria.

5. El fichero de configuración de Hibernate (`hibernate.cfg.xml`):
   - ☐ *a)* Permite establecer la correspondencia objeto-relacional para un conjunto de clases, independientemente de que para otras clases se pueda establecer mediante ficheros hbm.
   - ☐ *b)* Guarda todos los datos necesarios para establecer una conexión con el sistema gestor de bases de datos relacional en el que se van a almacenar los objetos persistentes.
   - ☐ *c)* Normalmente hace referencia a un fichero de ingeniería inversa, cuyo nombre suele ser `hibernate.reveng.xml`.
   - ☐ *d)* Debe almacenarse en la base de datos.

6. Cuando se hace `s.save(obj)`, siendo obj un objeto persistente:
   - ☐ *a)* Hibernate asigna siempre un identificador único automáticamente al objeto.
   - ☐ *b)* Hibernate asigna un identificador único al objeto solo en el caso en que se haya especificado para el atributo `generator` el valor `assigned`.
   - ☐ *c)* Hibernate asigna un identificador único al objeto solo si se ha especificado para el atributo `generator` el valor `identity`.
   - ☐ *d)* Hibernate asigna un identificador único al objeto si se ha especificado para el atributo `generator` el valor `identity`.

7. Las relaciones de uno a muchos entre dos clases:
   - ☐ *a)* Se representan mediante colecciones en las dos clases.
   - ☐ *b)* Se representan mediante una colección de tipo `HashSet` en una de las clases.
   - ☐ *c)* Se representan mediante un atributo sencillo en una clase y mediante una colección en la otra clase.
   - ☐ *d)* No se pueden representar directamente en Hibernate.

8. JPA es:
   - ☐ *a)* Un estándar para el que Hibernate incluye una implementación.
   - ☐ *b)* Un estándar de ORM para Java.
   - ☐ *c)* Un estándar que incluye anotaciones en clases de Java para ORM y también el lenguaje JPQL.
   - ☐ *d)* Todas las opciones anteriores son correctas.

9. Un objeto de una clase persistente puede no tener un identificador único asignado:
   - ☐ *a)* Si está en estado transitorio (*transient*).
   - ☐ *b)* Si está en estado gestionado (*managed*).
   - ☐ *c)* Justo después de pasar a estado separado (*detached*).
   - ☐ *d)* Justo después de pasar a estado eliminado (*removed*).

10. Para establecer la correspondencia objeto-relacional para varias clases entre las cuales hay una relación de herencia:
    - ☐ *a)* Es necesario introducir un campo adicional que funcione como discriminante.
    - ☐ *b)* Es necesario crear una tabla en la base de datos para todas y cada una de las clases relacionadas mediante la relación de herencia.
    - ☐ *c)* Si no se crean tablas para las subclases, es necesario crear uno o varios índices únicos en la tabla para la superclase, para evitar duplicidades.
    - ☐ *d)* Ninguna de las respuestas anteriores es válida.

SOLUCIONES:

1. a b **c** d	**5.** a **b** c d		
2. a b c **d**	**6.** a b c **d**	**9.** **a** b c d	
3. a **b** c d	**7.** a b **c** d	**10.** a b c **d**	
4. a b c **d**	**8.** a b c **d**		