Unidad 4

Correspondencia objeto-relacional

Acceso a datos

4

Índice

Introducción

Con el crecimiento de aplicaciones basadas en lenguajes de programación orientados a objetos y el auge de base de datos relacionales, empezaron a surgir distintos problemas para la persistencia de datos, debido a que los diseños orientados a objetos no se asemejan a los esquemas relacionales de las bases de datos.

Para intentar dar soluciones a estos problemas se plantearon diferentes planteamientos:

- Hacer uso de base de datos objetos. Permitiendo almacenar directamente objetos de los lenguajes en la base de datos.
- Hacer uso de base de datos objeto-relacional. Una extensión de las bases de datos objetos y mezclando el concepto relacional.
- Correspondencia objeto-relacional (ORM). Aplicar una capa intermedia entre la aplicación y cualquier tipo de base de datos, haciendo un mapeo de como son los objetos y los elementos de la base de datos.
- En esta unidad nos centraremos en estudiar cómo funciona la correspondencia objeto-relacional y como aplicarla.

Al finalizar esta unidad

- Conoceremos la correspondencia objeto-relacional (ORM).
- Aprenderemos a instalar hibernate.
- Aprenderemos a manejar hibernate para la persistencia de datos.
- Diseñaremos POJO para la correspondencia objeto-relacional con hibernate.
- Analizaremos el ciclo de vida de un objeto persistente en hibernate.

4.1. Correspondencia objeto-relacional

Los lenguajes de programación orientado a objetos pueden ser representado en un gráfico de objetos (colección de clases), mientras que en una base de datos relacional se representa en formato tabular tablas con filas y columnas.

Cuando se intenta dar persistencia a un objeto desde el lenguaje de programación a la base de datos, surgen diferencias entre ambos sistemas, no se representa igual un objeto que una tabla y dependiendo de la complejidad se puede romper el modelo relacional. ¿Sería fácil guardar un objeto complejo con múltiple herencia en una base de datos relacional? ¿Qué sucede con tipos de datos que no tienen correspondencia con la base de datos? Para dar solución a estos problemas, surge **la correspondencia objeto-relacional (ORM).**

Se puede definir la correspondencia objeto-relacional como el proceso para realizar un mapa que describa la relación entre una clase definida en el lenguaje de programación con una tabla en la base de datos.

Ventajas:

- Agiliza el desarrollo, evitando duplicidad de código, no es necesario crear operaciones CRUD por cada clase.
- Desarrollos más orientados a objetos
- Facilidad de gestión entre ambos sistemas
- Separa la aplicación del tipo de base de datos

Desventajas:

- Exceso de consumo de recursos del sistema.
- Para modelo de datos simples no tiene ningún beneficio
- No aporta mecanismos para optimización de consultas de datos complejas.

Para realizar este proceso de correspondencia existen diferentes framework que nos facilitan la tarea. Los más conocidos son **Hibernate** y **Ebeans**.

4.2. Hibernate

Es un framework disponible para Java que ofrece servicios y herramientas para aplicar la correspondencia objeto-relacional, con funcionalidades bastante potentes de consulta. Es el más usado y extendido del mercado.

La arquitectura de Hibernate incluye distintos elementos como son los objetos de persistencia, sesiones, transacciones, entre otros.

Hibernate usa el principio de reflexión en Java, permite el análisis y la modificación de los distintos atributos y características de las distintas clases en tiempo de ejecución.

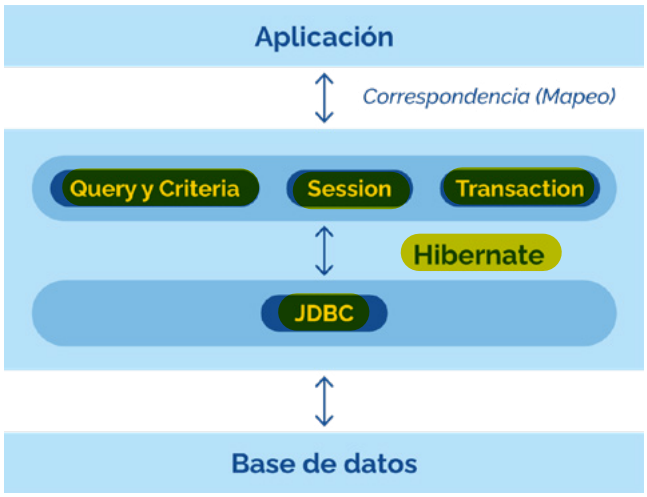

Imagen 1. Arquitectura Hibernate

La capa de Aplicación e Hibernate están unidas por los objetos de persistencia, debido a que es en esta parte donde se aplica la correspondencia objeto-relacional, dónde los datos fluyen y son mapeados desde los ficheros persistentes a la base de datos.

La capa de hibernate se encarga de realizar la conexión con la base de datos, gestionar las sesiones, transacciones como tener el mapa de la correspondencia objeto-relacional.

Hibernate tiene su propio lenguaje para manejar objetos persistentes muy similar a SQL denominado **HQL (Hibernate Query Language)**. A su vez, basándose en la api estándar de Java JPA para la persistencia de objetos, y haciendo uso de HQL, aporta un el lenguaje **JPQL (Java Persistence Query Language)**.

4.3. Descarga y uso de una versión reciente de Hibernate

Hibernate es un conjunto de librerías disponible que se pueden descargar desde su página oficial https://hibernate.org/ o descargar independientemente por separado. En la imagen siguiente se puede ver el listado de librerías necesarias para poder trabajar con hibernate.

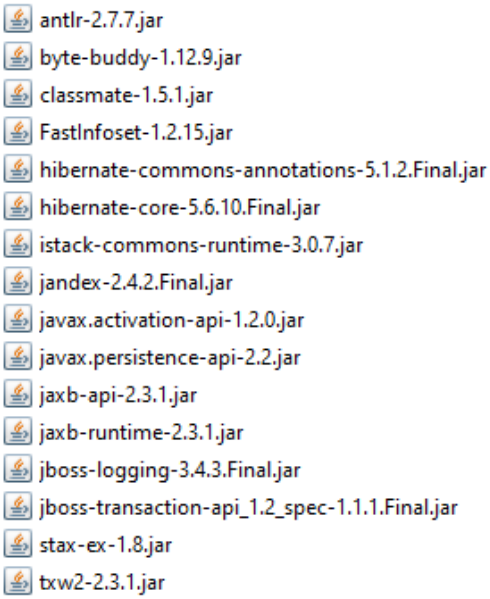

Imagen 2. Listado de librerías de hibernate

Hay que tener en cuenta los requerimientos necesarios para que pueda funcionar correctamente según la versión de hibernate que se use. Actualmente la versión estable es la 6.1 y requiere tener una versión de java 11 o superior.

Para empezar a trabajar con este framework, dependeremos el tipo de proyecto java que tengamos:

- Se pueden añadir directamente las librerías .jar de hibernate en el proyecto
- Si se trabaja con un gestor de dependencias como Maven, solo tenemos que añadir la dependencia de hibernate y ya se descargarán las librerías automáticamente y se añadirán al proyecto.

Existen otras herramientas que engloban un conjunto de frameworks para facilitar la instalación y puesta en marcha, sin tener que descargar uno a uno. **JBoss tools** es uno de los más importantes que contiene el framework hibernate a parte de otros. En esta unidad se instalará esta herramienta en eclipse. Dependiendo de la versión de eclipse puede que ya lleve instalado alguna herramienta.

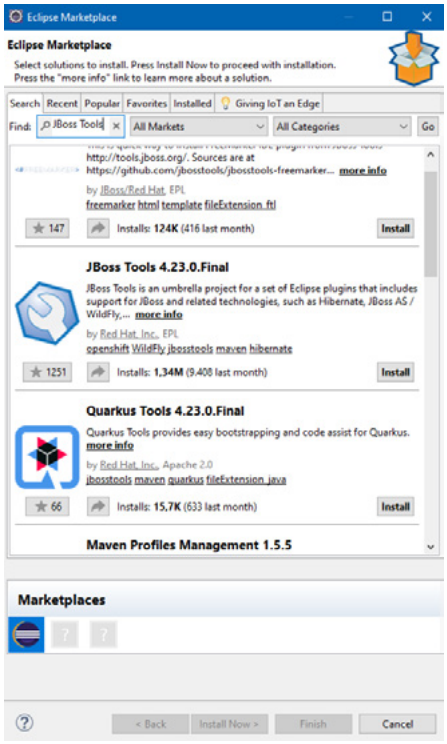

Imagen 3. EJBoss Tools

Una vez instalado en eclipse, dispondremos de la vista Hibernate, creación de ficheros de configuración, etc.

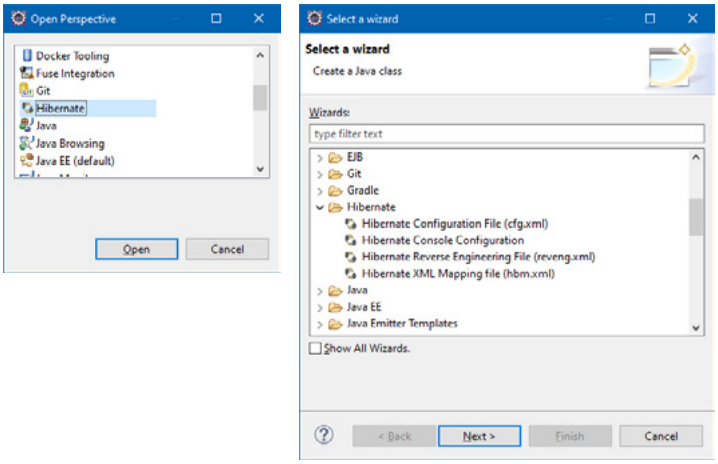

Imagen 4. Vista de hibernate y opciones de añadir ficheros

4.4. Correspondencia objeto-relacional a partir de las tablas

La correspondencia objeto-relacional (ORM) es la forma de mapear la estructura de la base de datos, conjunto de tablas y relaciones con los objetos en java usando hibernate.

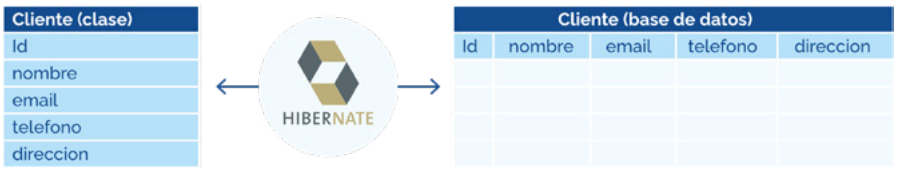

Imagen 5. Correspondencia entre una clase y una tabla de clientes

A partir de este punto veremos pasa a paso de la creación de una aplicación con conexión a una base de datos aplicando la correspondencia objeto-relacional con hibernate.

Se usará un esquema de base de datos de ventas para registrar clientes, productos y pedidos.

```
CREATE DATABASE ventas;
USE ventas;

CREATE TABLE IF NOT EXISTS clientes (
  id int(11) NOT NULL AUTO_INCREMENT,
  nombre varchar(100) NOT NULL,
  email varchar(100) NOT NULL,
  telefono varchar(15) NOT NULL,
  direccion text NOT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS productos (
  id int(11) NOT NULL AUTO_INCREMENT,
  nombre varchar(200) NOT NULL,
  descripcion text NOT NULL,
  precio float(10,2) NOT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS orden (
  id int(11) NOT NULL AUTO_INCREMENT,
  cliente_id int(11) NOT NULL,
  precio_total float(10,2) NOT NULL,
  PRIMARY KEY (id),
  FOREIGN KEY fk_orden_cliente(cliente_id)
    REFERENCES clientes (id) ON DELETE CASCADE ON UPDATE NO ACTION
);

CREATE TABLE IF NOT EXISTS orden_productos (
  id int(11) NOT NULL AUTO_INCREMENT,
  orden_id int(11) NOT NULL,
  producto_id int(11) NOT NULL,
  cantidad int(5) NOT NULL,
  PRIMARY KEY (id),
  FOREIGN KEY fk_orden_productos_orden_id(orden_id)
    REFERENCES orden (id) ON DELETE CASCADE ON UPDATE NO ACTION,
  FOREIGN KEY fk_orden_productos_producto_id(producto_id)
    REFERENCES productos (id) ON DELETE CASCADE ON UPDATE NO ACTION
);

CREATE USER 'usuarioVentas'@'localhost' IDENTIFIED BY 'usuarioVentas';
GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP,EXECUTE
    ON ventas.* TO 'usuarioVentas'@'localhost';
```

Imagen 5. Fichero SQL con el esquema de base de datos

4.4.1. Creación de la conexión con la base de datos

Lo primero es crear la conexión con la base de datos, necesitaremos tener disponible el driver de MySQL y el esquema de la base de datos creado. Vamos a usar el asistente de creación de conexiones de eclipse para verlo de forma gráfica y entender todo el proceso de una forma más simple. Este apartado también se puede hacer manualmente.

Los pasos a seguir son:

1. Añadir el driver de conectividad a MySQL. Desde *Preferences/Data Management/Connectivity/Driver Definitions* se indicará la versión del driver y en la pestaña *JAR List* añadiremos el driver de MySQL. *Si ya tuviéramos el driver definido, este paso no sería necesario.*

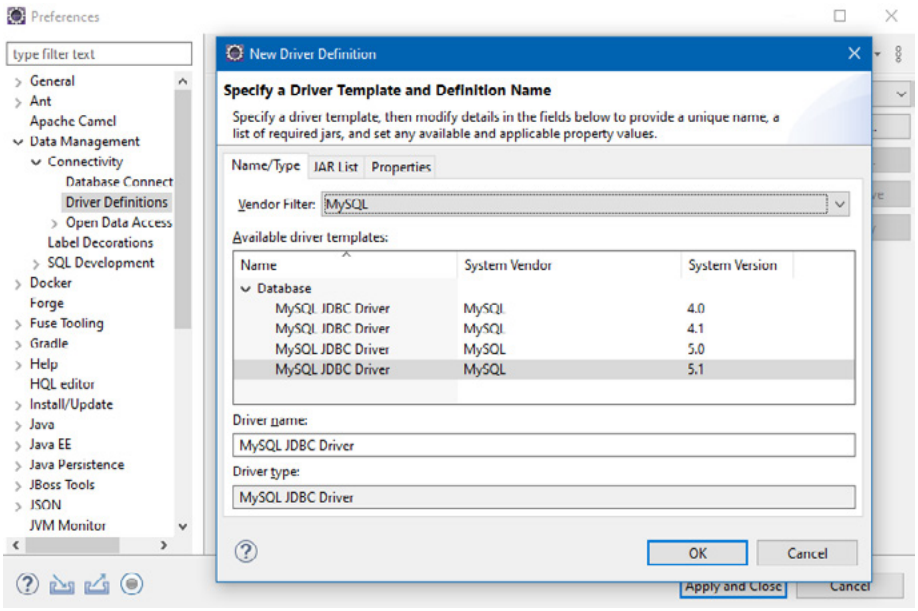

Imagen 6. Añadir el driver de MySQL al entorno de eclipse

2. Crear una conexión a la base de datos a partir de *añadir – Connection Profile – Connection Profile. Al añadir una nueva conexión podremos elegir el driver que hemos añadido en el paso anterior. Ya tan solo hay que indicar los datos de conexión, esquema, usuario y contraseña.*

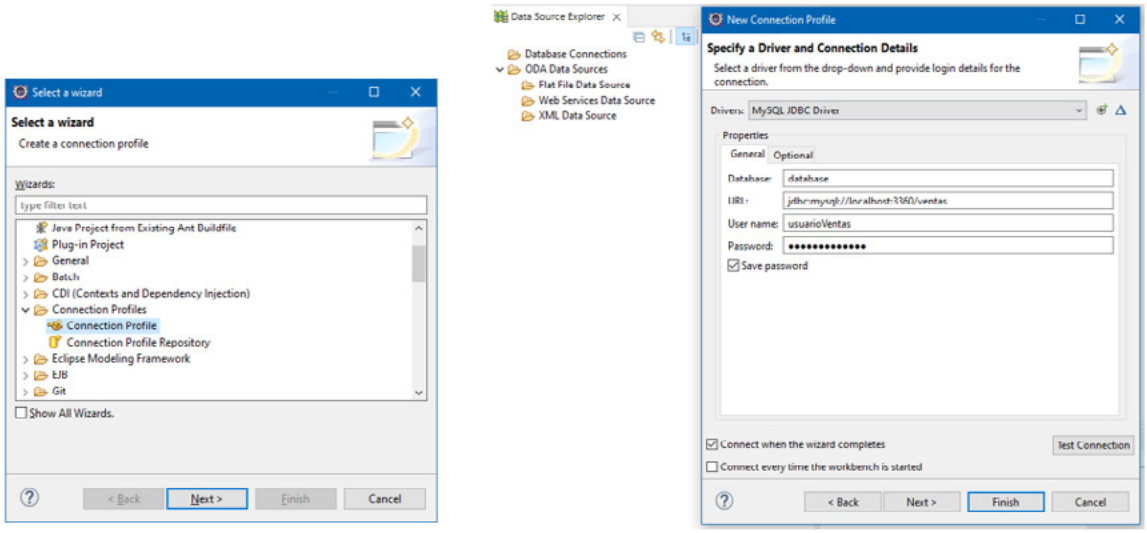

Imagen 7. Crear una conexión a la base de datos de MySQL

Si hemos realizado correctamente todos los pasos, es recomendable hacer un test de la conexión para verificar que conecta a la base de datos.

4.4.2. Creación del proyecto

Los proyectos se crean como hasta ahora, no hay que añadir ninguna configuración específica para indicar que es un proyecto con hibernate. Tan solo añadiremos las librerías de hibernate y el driver de conexión a la base de datos al proyecto.

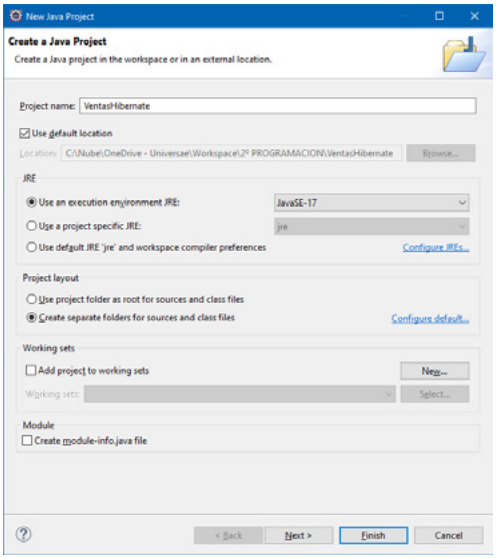

Imagen 8. Crear un proyecto Java Project

4.4.3. Fichero de configuración de Hibernate

El fichero principal de configuración es *hibernate.cfg.xml*. Este fichero está formado en xml y se especifica la conexión a la base de datos, la configuración de sesiones, los ficheros de mapeo, entre otras opciones.

Este fichero se puede crear a mano o con el asistente de eclipse desde añadir – *Hibernate - Configuration File (xfg.xml).* Nos solicitará los datos de conexión a la base de datos, la URL, el usuario y contraseña, adicionalmente se puede indicar el nombre de la sesión y el tipo de base de datos.

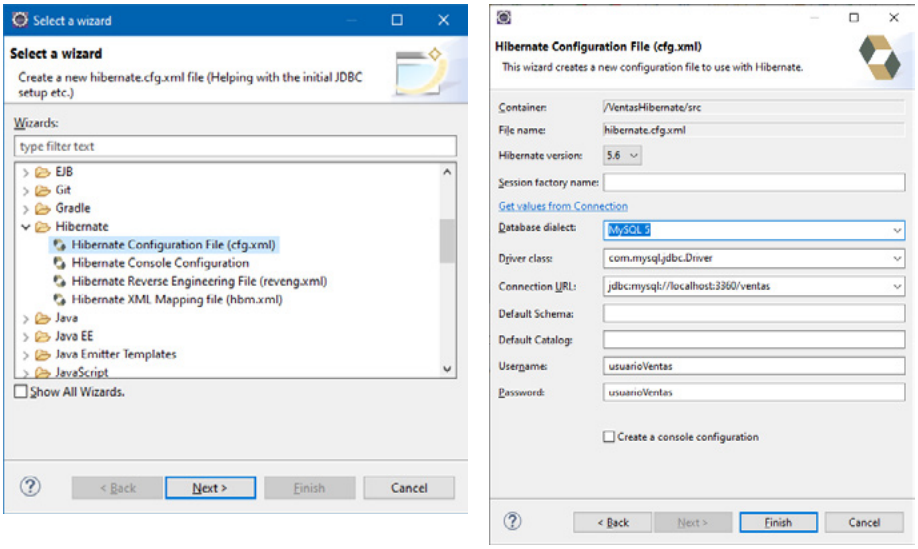

Imagen 9. Asistente creación hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.password">usuarioVentas</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3360/ventas</property>
        <property name="hibernate.connection.username">usuarioVentas</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>
    </session-factory>
</hibernate-configuration>
```

Imagen 10. Contenido hibernate.cfg.xml

Adicionalmente se tiene que crear una **consola de configuración de hibernate. Este punto no es necesario si cuando se creó el fichero de configuración se marcó la casilla para crear la consola automáticamente.**

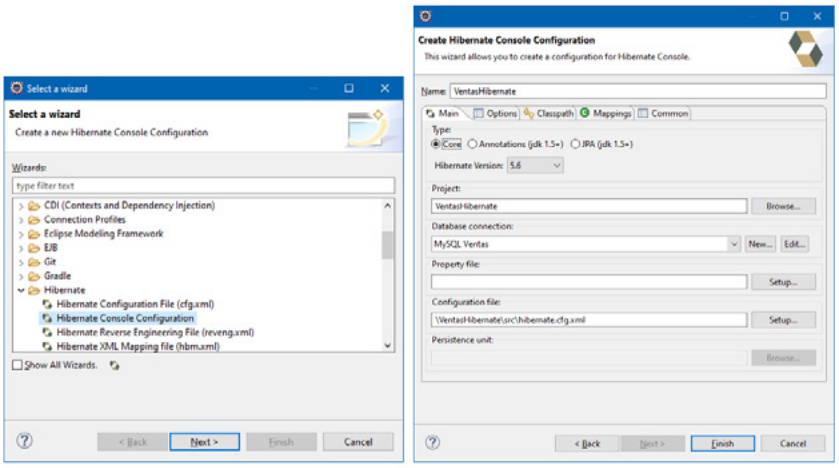

Imagen 11. Crear una consola de configuración de hibernate

4.4.4. Fichero de ingeniería inversa hibernate.reveng.xml

El fichero de ingeniería inversa define que tablas de la base de datos se van a usar para crear sus respectivas clases equivalentes en la aplicación.

Es de gran utilidad y ahorra bastante tiempo a los desarrolladores, solo es necesario disponer del esquema de la base de datos creado y con el fichero ***hibernate.reveng.xml*** podemos crear todas las clases que son necesarias para el ORM.

Este fichero se puede crear a mano o con el asistente de eclipse desde *añadir – Hibernate – Reverse Engineering File (reveng.xml).* El asistente no solicitará la consola de configuración y una vez seleccionada podremos ver el esquema de la base de datos. El siguiente paso es seleccionar aquellas tablas que se aplicará la ingeniería inversa.

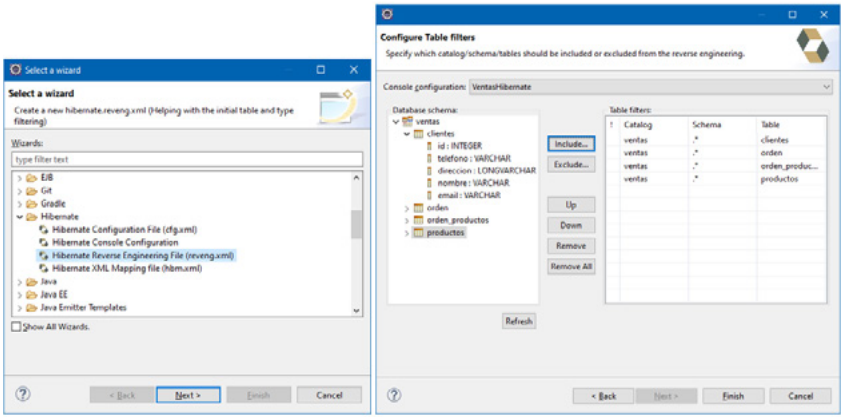

Imagen 12. Crear el fichero hibernate.reveng.xml

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

<hibernate-reverse-engineering>
  <table-filter match-catalog="ventas" match-name="clientes"/>
  <table-filter match-catalog="ventas" match-name="orden"/>
  <table-filter match-catalog="ventas" match-name="orden_productos"/>
  <table-filter match-catalog="ventas" match-name="productos"/>
</hibernate-reverse-engineering>
```

Imagen 13. Contenido del fichero hibernate.reveng.xml

Este fichero no es necesario para la configuración de hibernate. Podemos ubicarlo donde queramos e incluso renombrarlo.

4.4.5. POJO (clases) y ficheros de correspondencia

Las clases que se generan para el mapeo de la correspondencia objeto-relacional se le denominan **POJO (*plain old java file*).** Los POJOs son clases java simples, que contiene atributos según a la columna de la tabla que mapea y métodos get y set para consultar y modificar sus valores.

A partir del fichero *hibernate.reveng.xml* vamos a generar los POJOs:

1. Abrir la perspectiva de hibernate desde *Windows/Perspective*. Una vez abierta la vista, se habilitará un nuevo botón para arrancar aplicaciones, esta vez para hibernate.

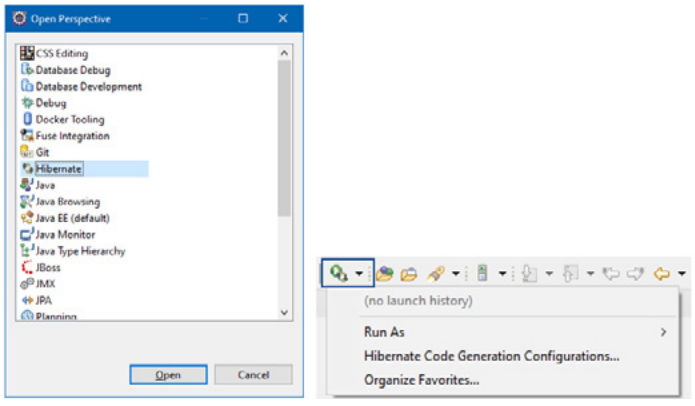

Imagen 14. Perspectiva y botón de arranque de hibernate

2. Ahora se configurará como se tienen que crear los POJOs. Se pulsará en *Hibernate Code Generation Configuration* y añadiremos una nueva configuración, indicando en la pestaña *Main*, la consola de configuración, la ruta de la carpeta src de nuestro proyecto, el nombre del paquete donde se ubicarán las clases y la ruta del fichero hibernate.reveng.xml. Y por último en la pestaña *Exporters* se seleccionará Generate EJB3 annotations, Domain code, Hibernate XML Mapping y Hibernate XML Configuration.

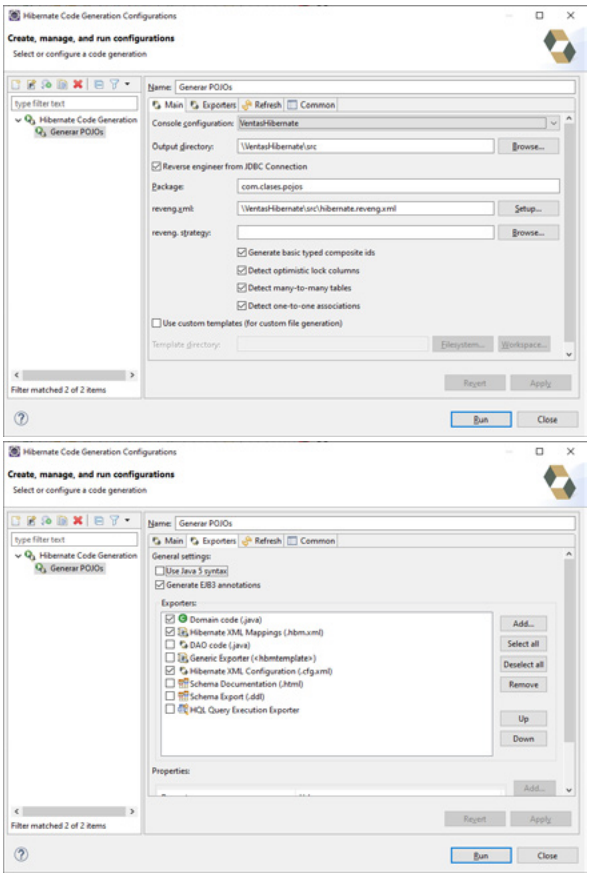

Imagen 15. Configuración de Hibernate Code Generation Configuration

3. Por último, solo quedará darle a Run para que empiece el proceso de ingeniería inversa. Una vez finalizado se actualizará el proyecto y aparecerán las clases y unos ficheros de mapeo .hbm.xml por cada clase.

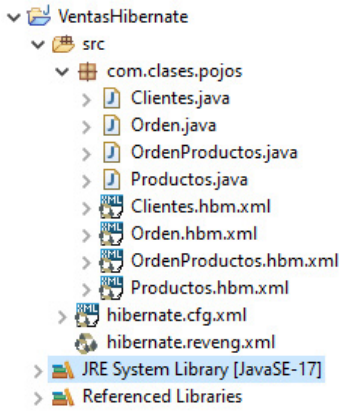

Imagen 16. Resultado de la ingeniería inversa

Al finalizar este proceso se puede comprobar que el fichero de configuración de hibernate ha sido actualizado, añadiendo nuevas propiedades, y una de ellas la ubicación donde se encuentran los ficheros .hbm.xml de mapeo.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.bytecode.use_reflection_optimizer">false</property>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.password">usuarioVentas</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/ventas</property>
        <property name="hibernate.connection.username">usuarioVentas</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

        <property name="hibernate.hbm2ddl.auto">none</property>
        <property name="hibernate.search.autoregister_listeners">true</property>
        <property name="hibernate.validator.apply_to_ddl">false</property>
        <property name="hibernate.show_sql">true</property>

        <mapping resource="com/clases/pojos/Productos.hbm.xml"/>
        <mapping resource="com/clases/pojos/Clientes.hbm.xml"/>
        <mapping resource="com/clases/pojos/Orden.hbm.xml"/>
        <mapping resource="com/clases/pojos/OrdenProductos.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Imagen 17. Contenido de hibernate.cfg.xml tras el proceso de ingeniería inversa

4.4.6. HibernateUtil.java

La clase *HibernateUtil.java* tienen código para facilitar la gestión de sesiones de hibernate y su puesta en marcha. No es una clase obligatoria y tampoco es requerida para la configuración de hibernate, pero si es recomendable hacer uso de ella para facilitar las conexiones con hibernate.

No existe un asistente para crear esta clase, debido a que es una recomendación y no está incluida en el estándar de hibernate, es por ello que desde eclipse se tenga que crear a mano. Se puede usar el código base para la versión de Hibernate 5.6 que aparece en la siguiente imagen.

```
public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            return new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Build SessionFactory failed :" + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void close() {
        getSessionFactory().close();
    }

}
```

Imagen 18. Contenido del fichero HibernateUtil

Para entender el código vamos a explicar que es lo que hace. Consulta el fichero de configuración de hibernate para obtener todas las propiedades definidas como, la conexión, usuario, contraseña, ficheros de mapeo, etc. Finalmente crea un objeto SessionFactory para su uso en toda la aplicación con la finalidad de gestionar las sesiones y conexiones a la base de datos.

4.5. Programa de ejemplo para persistencia de objetos con Hibernate

Llegado a este punto vamos a hacer un ejemplo para ver su funcionamiento. Se va a crear un nuevo cliente y producto, en la siguiente imagen se puede ver el código resultante.

```
public static void main(String[] args) {

    Clientes cliente = new Clientes();
    cliente.setNombre("Maria");
    cliente.setTelefono("600888999");
    cliente.setEmail("maria@email.com");
    cliente.setDireccion("C/ Garcilaso, nº 12, 8ªA, Madrid");

    Productos producto = new Productos();
    producto.setNombre("Camiseta");
    producto.setDescripcion("Camiseta de color verde y talla unica");
    producto.setPrecio(10.5F);

    Session session = null;

    try {
        session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();

        session.save(cliente);
        session.save(producto);

        session.getTransaction().commit();
        session.close();

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

    } finally {
        HibernateUtil.close();
    }

}
```

Imagen 19. Código ejemplo para insertar dos registros usando hibernate

Si todo ha ido bien desde la consola de eclipse se verá una gran cantidad de información de color rojo y dos líneas de color negro mostrando las dos sentencias de inserción en SQL. Se puede comprobar en la base de datos que se han creado los registros.

4.6. Ficheros hbm o de correspondencia de Hibernate

Hasta ahora se han generado automáticamente los ficheros de mapeo .hbm.xml facilitando el trabajo ya que solo era necesario tener el esquema de base de datos.

Ahora vamos a profundizar en este tipo de fichero. Son documentos xml que enlazan e identifican como se relaciona la clase con la tabla en la base de datos.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Generated 24 ago 2022 19:13:02 by Hibernate Tools 5.6.7.Final -->
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class catalog="ventas" name="com.clases.pojos.Clientes" optimistic-lock="none" table="clientes">
        <id name="id" type="java.lang.Integer">
            <column name="id"/>
            <generator class="identity"/>
        </id>
        <property name="nombre" type="string">
            <column length="100" name="nombre" not-null="true"/>
        </property>
        <property name="email" type="string">
            <column length="100" name="email" not-null="true"/>
        </property>
        <property name="telefono" type="string">
            <column length="15" name="telefono" not-null="true"/>
        </property>
        <property name="direccion" type="string">
            <column length="65535" name="direccion" not-null="true"/>
        </property>
        <set fetch="select" inverse="true" lazy="true" name="ordens" table="orden">
            <key>
                <column name="cliente_id" not-null="true"/>
            </key>
            <one-to-many class="com.clases.pojos.Orden"/>
        </set>
    </class>
</hibernate-mapping>
```

Imagen 20. Código del fichero de mapeo clientes.hbm.xml

4.6.1. Correspondencia para las clases y atributos de clase

El primer elemento que nos encontramos es **<hibernate-mapping>** obligatorio para indicar que es un fichero de mapeo de hibernate.

Otro elemento que va después de la apertura es **<class>**. Este elemento indica con el atributo *name* donde se encuentra la clase en java y con el atributo *table* indica que tabla de la base de datos le corresponde, en resumen, es la asociación entre la clase y la tabla de la base de datos.

Los siguientes elementos ya definirán los atributos de la clase y el campo que le corresponde en la base de datos. El primero de todo siempre será **<id>** que indica la clave primaria, los siguientes elementos serán **<property>** simples atributos de clase.

La estructura de **<id>** y **<property>** es similar. En la cabecera aparecen los atributos *name* y *type*, indicando como se llama y que tipo de datos aparece en la clase. En el cuerpo aparece el elemento **<column>** para especificar el nombre que corresponde a la columna de la base de datos, por otro lado, es posible que pueda aparecer otros atributos, como *not-null*, *length*... Estos atributos cogen las propiedades de restricción que hay definidas en la base de datos.

En las claves primarias suele aparecer el elemento **<generator>** que indica como se genera la clave primaria, que puede ser:

- **identity:** clave autogenerada.
- **assigned:** hay que asignarle un valor manualmente.
- **sequence:** cuando existe una secuencia de la base de datos para generar la clave. Se suele aplicar con base de datos oracle.
- **native:** se adapta y se comporta como identity o sequence.
- **foreign:** usa el identificador de otros objeto asociado. Para relaciones uno a uno.

4.6.2. Correspondencia para las relaciones

Igual que tenemos que indicar la asociación entre clase y tabla, atributos de la clase y los campos de la tabla en la base de datos, también es necesario especificar las relaciones que existen.

Relaciones uno a uno

Se usa el elemento **<one-to-one>** indicando el nombre del atributo de la clase y la ubicación de la clase a la que hace referencia.

Ejemplo: Una persona solo puede tener un animal, y un animal solo puede pertenecer a una persona.

Clase animal

```
<one-to-one name”persona_id” class=”paquete.Persona”>
```

Relaciones uno a muchos

Se usa el elemento **<many-to-one>** o **<one-to-many>** indicando en la cabecera el nombre del atributo de la clase, la ubicación de la clase a la que hace referencia y la forma de búsqueda con el atributo *fetch* tomando valores de select o join. Por último, en el cuerpo se indica con el elemento **<column>** el nombre y propiedades que tiene en la tabla de la base de datos.

Ejemplo: Un cliente puede tener más de una orden de pedido, pero una orden solo puede tener un cliente.

Clase cliente

```
<set name=”ordenes” table=”orden” fetch=”select”>
     <key>
         <column name=”cliente_id” not-null=”true”/>
     </key>
         <one-to-many class=”com.clases.pojos.Orden”/>
</set>
```

Para este tipo de relación, hibernate trata estas relaciones como un conjunto de elementos. Por eso, se puede ver que crea un tipo de objeto de colección set.

Relaciones de muchos a muchos

En hibernate este tipo de relación no se puede representar directamente, ya que, se puede descomponer representando dos relaciones uno a muchos. Aun así, existe el elemento **<many-to-many>**.

Ejemplo: Un producto puede tener más de una categoría. Y una categoría puede tener más de un producto.

Clase Producto

```
<set name="categorias" table="producto_categoria" fetch="select">
    <key>
        <column name="producto_id" not-null="true" />
    </key>
    <many-to-many entity-name="paquete.Categoria">
        <column name="categoria_id" not-null="true" />
    </many-to-many>
</set>
```

Clase Categoria

```
<set name="productos" table="producto_categoria" fetch="select">
    <key>
        <column name="categoria_id" not-null="true" />
    </key>
    <many-to-many entity-name="paquete.Producto">
        <column name="producto_id" not-null="true" />
    </many-to-many>
</set>
```

Lo que hace hibernate es simular una tabla intermedia (No crea una clase para la tabla intermedia), para hacerlo hay que usar el elemento <many-to-many> y un conjunto set en los dos ficheros de mapeo que intervienen en la relación. Es importante ver que el atributo *table* indica el nombre de la tabla intermedia que, si existe en la base de datos, pero que hibernate no creará objeto de ella.

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

Este tipo de relación en objetos persistentes es muy sencillo. Tenemos un objeto que interviene en la relación y otro que guarda su referencia en un atributo de la clase.

En la siguiente imagen podemos ver un ejemplo.

```
Session session = null;

try {
    session = HibernateUtil.getSessionFactory().openSession();
    session.beginTransaction();

    Animal animal = session.get(Animal.class, 1);

    Persona persona = new Persona();
    persona.setNombre("Juan");
    persona.setAnimal(animal);

    session.save(persona);

    session.getTransaction().commit();
    session.close();

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

} finally {
    HibernateUtil.close();
}
```

Imagen 21. Código manejo de objetos con relación uno a uno

Tenemos un caso de un animal que solo puede pertenecer a una persona. Se procede a consultar el animal con el método *.get(.class, id)* y con el objeto devuelto se asigna en el atributo del objeto persona para posteriormente guardarlo.

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

A diferencia de la relación uno a uno, aquí tendremos dos objetos que intervienen en la relación, un objeto tiene un atributo que guarda la referencia al objeto siguiente de la relación, y el otro objeto interviniente guarda una colección de objetos a los que hace referencia.

En la siguiente imagen podemos ver un ejemplo.

```
Session session = null;
try {
    session = HibernateUtil.getSessionFactory().openSession();
    session.beginTransaction();

    Clientes cliente = new Clientes();
    cliente.setNombre("Pablo");
    cliente.setDireccion("C. Miraflor, 77, 2º A, Barcelona");
    cliente.setTelefono("699888000");
    cliente.setEmail("pablo@email.com");

    session.save(cliente);

    Orden orden1 = new Orden();
    orden1.setClientes(cliente);
    orden1.setPrecioTotal(100);

    Orden orden2 = new Orden();
    orden2.setClientes(cliente);
    orden2.setPrecioTotal(150);

    session.save(orden1);
    session.save(orden2);

    session.refresh(cliente);

    Set<Orden> ordenes = cliente.getOrdens();

    for (Orden orden : ordenes) {
        System.out.println(orden.getId() + " " + orden.getClientes() + " "+ orden.getPrecioTotal());
    }

    session.getTransaction().commit();
    session.close();

} catch (Exception e) {
    e.printStackTrace(System.err);
    if(session != null)
        session.getTransaction().rollback();
} finally {
    HibernateUtil.close();
}
```

Imagen 22. Código manejo de objetos con relación uno a muchos

En este ejemplo tenemos un cliente que va a tener más de una línea de orden o pedido. Se crea el cliente y posteriormente dos órdenes, se relacionan añadiendo en el atributo clientes de la orden el cliente creado. Finalmente se obtiene la lista de órdenes a mostrar del cliente, este paso se puede realizar si hemos invocado al método `.refresh(cliente)`. En el siguiente punto veremos que hace.

4.9. Sesiones y estados de los objetos persistentes

Uno de los aspectos más importantes es conocer la gestión de sesiones y como se mantienen los objetos en hibernate y su actualización a la base de datos.

Cuando se crea, modifica o se borrar un objeto persistente en código no se traspasa automáticamente a hibernate, y aún menos a la base de datos, hay que indicar que hacer con ese objeto.

El primer paso es cargar los objetos a la sesión que nos facilita hibernate. La **sesión** contiene una conexión a la base de datos y gestiona las diferentes **transacciones** que se pueden hacer. Por último, se manda las instrucciones para que los objetos que están cargados en la sesión sean guardados, actualizados o eliminados en la base de datos.

Los objetos persistentes se pueden adoptar alguno de los siguientes estados:

- **Transitorio.** Es un objeto nuevo, no ha sido cargado en la sesión y tampoco existe en la base de datos. Puede ser que no se tenga la clave primaria de la base de datos, si esta es autogenerada.
- **Persistente o gestionado.** Es un objeto que contiene la clave primaria, está cargado en la sesión y también existe en la base de datos.
- **Separado.** Es un objeto que contiene la clave primaria y existe en la base de datos, pero se ha separado de la sesión, es decir, que se quitó de la sesión.
- **Eliminado.** Es un objeto que contiene la clave primaria, existe en la base de datos y está cargado en la sesión. Solo que está pendiente de que sea borrado de la base de datos.

En el siguiente esquema aparecen los diferentes estados y los métodos que cambian de un estado a otro.

Imagen 23. Diagrama del flujo de estados

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

Estos métodos permiten pasar un objeto del estado transitorio a persistente.

- `save()`. Guarda el objeto en sesión. Si el objeto tiene una clave primaria que se autogenera, una vez guardado en la base de datos, automáticamente lo carga. Este tipo de método se puede usar dentro o fuera de una transacción. **Ejemplo:** `Session.save(Objeto)`.
- `saveOrUpdate()`. Guarda o actualiza el objeto en sesión. Si el objeto tiene una clave primaria que se autogenera, una vez guardado o actualizado en la base de datos, automáticamente lo carga. **Ejemplo:** `Session.saveOrUpdate(Objeto)`. Este tipo de método se puede usar dentro o fuera de una transacción.
- `persist()`. Guarda el objeto en sesión. A diferencia de los demás este método no devuelve la clave primaria una vez guardado en la base de datos. Este método solo se puede usar en una transacción. **Ejemplo:** `Session.persist(Objeto)`.

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

Permite obtener objetos persistentes, su función es realizar una búsqueda.

- `get()`. Hay que indicarle la clase y el identificador. Si no lo encuentra devuelve null. **Ejemplo:** `Session.get(-Clase.class,identificador)`.
- `load()`. Hay que indicarle la clase y el identificador. Si no lo encuentra devuelve una excepción. **Ejemplo:** `Session.load(Clase.class,identificador)`.

4.9.3. De persistente a eliminado con delete()

Indica a la sesión que el objeto va a ser eliminado con el método `delete()`. El objeto a eliminar se le pasa como parámetro. **Ejemplo:** `Session.delete(Objeto)`.

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

Separa los objetos de la sesión.

- `evict()`. Quita el objeto de la sesión. **Ejemplo:** `Session.evict(Objecto)`.
- `close()`. Guarda todos los cambios de todos los objetos de la sesión a la base de datos y los separa. **Ejemplo:** `Session.close()`.
- `clear()`. Igual que `close()` pero sin guardar los cambios en la base de datos. **Ejemplo:** `Session.Clear()`.

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

Los objetos separados se pueden volver a cambiar de estado a persistente.

- `saveOrUpdate()`. Hace la misma función explicada en el punto anterior.
- `update()`. Carga de nuevo el objeto en sesion. Si el objeto ya está en estado persistente con la misma clave primaria, lanza una excepción **Ejemplo:** `Session.update(Objeto)`.
- `merge()`. Similar a `update()`. Si ya existe un objeto en estado persistente con la misma clave primaria lo actualiza atributo a atributo, si no, lo carga en sesión. **Ejemplo:** `Session.merge(Objeto)`.
- `lock()`. Carga el objeto en sesión y lo bloquea contra lectura, escritura... **Ejemplo:** `Session.lock(Objeto, tipo de bloqueo)`.

4.10. Lenguajes de consulta HQL y JPQL

HQL (Hibernate Query Language) es un lenguaje propio de hibernate, similar a SQL con sentencias de `select`, `insert`, `update` y `delete`. JPQL (Java Persistence Query Language) es un subconjunto de HQL. La única diferente entre ambos, es que SQL devuelve datos, en cambio HQL devuelve objetos.

En este punto, no se ve en detalle estos lenguajes, debido a que son muy parecidos a SQL. Y con tener conocimiento de SQL es suficiente para su comprensión.

4.10.1. La interfaz Query

Hibernate proporciona interfaz Query para crear consultas con HQL. Esta interfaz es muy parecida a PreparedStatement de JDBC.

La interfaz Query ofrece el método `.createQuery()` para poder construir cualquier tipo de sentencia.

A continuación, se muestra cómo hacer un `select`, `update` y `delete`.

Select

Se puede crear una consulta directa o parametrizable. Los parámetros se indican con dos puntos y se pasan con el método `.setParameter(parámetro, valor)`.

Los resultados se obtienen con `.getResultList()`.

```
Session session = null;

try {
    session = HibernateUtil.getSessionFactory().openSession();

    Query query = session.createQuery("FROM Clientes WHERE id = :id");

    query.setParameter("id", 1);

    List<Clientes> clientes = (List<Clientes>) query.getResultList();

    for (Clientes cliente : clientes) {
        System.out.print(cliente.getId() +" "+ cliente.getNombre() + " "+ cliente.getDireccion() + " "+cliente.getEmail()+"\n");
    }

    session.close();

} catch (Exception e) {
    e.printStackTrace(System.err);
} finally {
    HibernateUtil.close();
}
```

Imagen 24. Ejemplo de consulta con HQL

Como se puede ver en la imagen 24. Las sentencias HQL no es necesario llevar `SELECT`, obtienen un objeto cliente con todos sus atributos.

Update

La sentencia de actualización es idéntica a SQL. Finalmente se ejecuta con el método `executeUpdate()`. A diferencia de las sentencias `SELECT`, para la actualización es necesario el uso de transacciones.

```
Session session = null;

try {
    session = HibernateUtil.getSessionFactory().openSession();
    session.beginTransaction();

    Query query = session.createQuery("UPDATE Clientes SET nombre = 'Paola'  WHERE id = :id");

    query.setParameter("id", 2);

    int i = query.executeUpdate();
    System.out.println("Registros actualizados: "+ i);

    session.getTransaction().commit();

    session.close();

} catch (Exception e) {
    e.printStackTrace(System.err);
} finally {
    HibernateUtil.close();
}
```

Imagen 25. Ejemplo de actualización con HQL

Delete

La sentencia de borrado es idéntica a SQL. Finalmente se ejecuta con el método `executeUpdate()`. A diferencia de las sentencias `SELECT`, para la actualización es necesario el uso de transacciones.

```
Session session = null;

try {
    session = HibernateUtil.getSessionFactory().openSession();
    session.beginTransaction();

    Query query = session.createQuery("DELETE Clientes WHERE id = :id");

    query.setParameter("id", 4);

    int i = query.executeUpdate();

    System.out.println("Registros borrados: "+ i);

    session.getTransaction().commit();

    session.close();

} catch (Exception e) {
    e.printStackTrace(System.err);
} finally {
    HibernateUtil.close();
}
```

Imagen 26. Ejemplo de borrado con HQL

4.11. Correspondencia de la herencia

Cuando hablamos de herencia aplicada en código, tenemos varias clases relacionadas entre sí, que depende de la superclase o clase padre. Hibernate permite hacer la correspondencia de herencia, pero sucede un problema y es que, en base de datos no existe un modelo de datos que represente tal cual la herencia aplicada en código.

Los diagramas E/R ayudan a definir como se representa la herencia tanto en código y en la base de datos según las distintas formas de diseño disponibles.

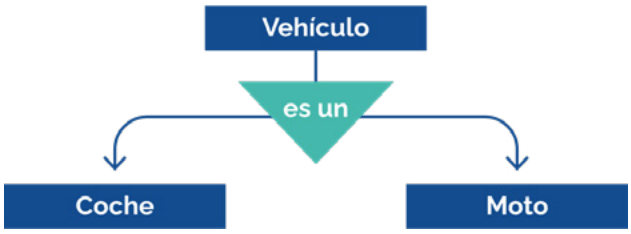

Imagen 27. Diagrama E/R para representar una relación de herencia

En los ficheros de mapeo hbm.xml tenemos los elementos `<subclass>` y `<joined-subclass>` para definir estructuras de herencia.

A continuación, vamos a ver unos ejemplos dependiendo de cómo se representa la herencia en la base de datos.

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

En la base de datos solo existe una sola tabla para representar la herencia. Esta tabla contiene todos los campos de la clase principal y subclase que componen la herencia. A parte se añade un campo nuevo para distinguir un tipo u otro de la herencia.

```
CREATE TABLE IF NOT EXISTS vehiculo (
    id int(10) NOT NULL AUTO_INCREMENT,
    marca varchar(100) NOT NULL,
    modelo varchar(100) NOT NULL,
    tipo char(10) not null,
    id_coche int(10),
    num_puertas int(2),
    id_moto int (10),
    num_plazas int(2),
    PRIMARY KEY (id),
    constraint check_tipos check(tipo='vehiculo'
                                  OR (tipo='coche' and not isnull(id_coche) and not isnull(num_puertas))
                                  OR (tipo='moto' and not isnull(id_moto) and not isnull(num_plazas)))
);
```

Imagen 28. Esquema SQL de la tabla vehiculo

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class catalog="herencia" name="com.herencia.Vehiculo" table="vehiculo" discriminator-value="vehiculo">
        <id name="id" type="java.lang.Integer">
            <column name="id"/>
            <generator class="identity"/>
        </id>
        <discriminator column="tipo" type="string"></discriminator>
        <property name="marca" type="string">
            <column length="100" name="marca" not-null="true"/>
        </property>
        <property name="modelo" type="string">
            <column length="100" name="modelo" not-null="true"/>
        </property>
        <subclass name="com.herencia.Coche" discriminator-value="coche">
            <property name="idCoche" type="int">
                <column length="10" name="id_coche"/>
            </property>
            <property name="numPuertas" type="int">
                <column length="2" name="num_puertas"/>
            </property>
        </subclass>
        <subclass name="com.herencia.Moto" discriminator-value="moto">
            <property name="idMoto" type="int">
                <column length="10" name="id_moto"/>
            </property>
            <property name="numPlazas" type="int">
                <column length="2" name="num_plazas"/>
            </property>
        </subclass>
    </class>
</hibernate-mapping>
```

Imagen 29. Código de mapeo vehiculo.hbm.xml con subclass

Como se puede ver en el fichero de mapeo de la imagen 29, existe la superclase vehículo, dos subclases coche y moto, y un campo tipo que se define con la etiqueta `<discriminator>`, este campo solo puede tomar tres valores, `"vehículo"`, `"coche"` y `"moto"` igual que en la base de datos y en el mapeo se indica mediante la propiedad `discriminator-value`.

4.11.2. Eliminación de la jerarquía (Una tabla por subclase)

Para representar esta herencia se crea una tabla por cada subclase en la base datos, con una peculiaridad, es que las tablas que representan las subclases su clave primaria será la clave de la tabla de la superclase y estarán relacionadas como clave foránea.

```sql
CREATE TABLE IF NOT EXISTS vehiculo (
    id_vehiculo int(10) NOT NULL AUTO_INCREMENT,
    marca varchar(100) NOT NULL,
    modelo varchar(100) NOT NULL,
    PRIMARY KEY (id_vehiculo)
);

CREATE TABLE IF NOT EXISTS coche (
    id_vehiculo int NOT NULL,
    id_coche int(10) NOT NULL,
    num_puertas int(2) NOT NULL,
    PRIMARY KEY (id_vehiculo),
    CONSTRAINT fk_coche_vehiculo FOREIGN KEY(id_vehiculo) REFERENCES vehiculo(id_vehiculo)
);

CREATE TABLE IF NOT EXISTS moto (
    id_vehiculo int NOT NULL,
    id_moto int(10) NOT NULL,
    num_plazas int(2) NOT NULL,
    PRIMARY KEY (id_vehiculo),
    CONSTRAINT fk_moto_vehiculo FOREIGN KEY(id_vehiculo) REFERENCES vehiculo(id_vehiculo)
);
```

Imagen 30. Esquema SQL de las tablas vehículo, coche y moto

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class catalog="herencia" name="com.herencia.Vehiculo" table="vehiculo" >
        <id name="idVehiculo" type="java.lang.Integer">
            <column name="id_vehiculo"/>
            <generator class="identity"/>
        </id>
        <property name="marca" type="string">
            <column length="100" name="marca" not-null="true"/>
        </property>
        <property name="modelo" type="string">
            <column length="100" name="modelo" not-null="true"/>
        </property>
        <joined-subclass name="com.herencia.Coche">
            <key column="id_vehiculo"/>
            <property name="idCoche" type="int">
                <column length="10" name="id_coche"/>
            </property>
            <property name="numPuertas" type="int">
                <column length="2" name="num_puertas"/>
            </property>
        </joined-subclass>
        <joined-subclass name="com.herencia.Moto">
            <key column="id_vehiculo"/>
            <property name="idMoto" type="int">
                <column length="10" name="id_moto"/>
            </property>
            <property name="numPlazas" type="int">
                <column length="2" name="num_plazas"/>
            </property>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

Imagen 31. Código de mapeo vehiculo.hbm.xml con joined-subclass

Este tipo de herencia es más sencillo que el anterior debido a que no hay que definir un campo de tipo con `discriminator`. Solo hay que cambiar `subclass` por `joined-subclass`.

4.12. Consultas con SQL

Aún que hibernate dispone de herramientas suficientes para realizar consultas a la base de datos con su propio lenguaje, existe la posibilidad de realizar consultas con SQL. Para ello se usa el método *.createNativeQuery(SQL).*

```java
Session session = null;

try {
    session = HibernateUtil.getSessionFactory().openSession();

    List<Object[]> query = session.createNativeQuery("SELECT * FROM Clientes").list();

    for (Object[] cliente : query) {
        System.out.print(cliente[0]+" ");
        System.out.print(cliente[1]+" ");
        System.out.print(cliente[2]+" ");
        System.out.println(cliente[3]);
    }

    session.close();

} catch (Exception e) {
    e.printStackTrace(System.err);
} finally {
    HibernateUtil.close();
}
```

Imagen 32. Ejemplo de código para hacer consultas en SQL

Es recomendable no usar SQL, ya existe HQL para ello, de esta manera no se pierde el potencial que tiene hibernate.

www.universae.com