Mapeo Objeto-Relacional. Hibernate

Desfase objeto-relacional

El *desfase objeto-relacional* surge cuando en el desarrollo de una aplicación con un lenguaje orientado a objetos se hace uso de una base de datos relacional. Hay que tener en cuenta que esta situación se da porque tanto los lenguajes orientados a objetos como las bases de datos relacionales están ampliamente extendidas.

En cuanto al desfase, ocurre que en nuestra aplicación Java (como ejemplo de lenguaje Orientada a Objetos) tendremos, por ejemplo, la definición de una clase cualquiera con sus atributos y métodos:

```
public class Personaje {

  private int id;

  private String nombre;

  private String descripcion;

  private int vida;

  private int ataque;



  public Personaje(. . .) {

    . . .

  }



  // getters y setters

}
```

Mientras que en la base de datos tendremos una tabla cuyos campos se tendrán que corresponder con los atributos que hayamos definido anteriormente en esa clase. Puesto que son estructuras que no tienen nada que ver entre ellas, tenemos que hacer el mapeo manualmente, haciendo coincidir (a través de los getters o setters) cada uno de los atributos con cada uno de los campos (y vicerversa) cada vez que queramos leer o escribir un objeto desde y hacia la base de datos, respectivamente.

```
CREATE TABLE personajes (
```

```
  id INT PRIMARY KEY AUTO_INCREMENT;

  nombre VARCHAR(50) NOT NULL,

  descripcion VARCHAR(50),

  vida INT DEFAULT 10,

  ataque INT DEFAULT 10;

);
```

Eso hace que tengamos que estar continuamente descomponiendo los objetos para escribir la sentencia `SQL` para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos.

¿Qué es el mapeo objeto-relacional?

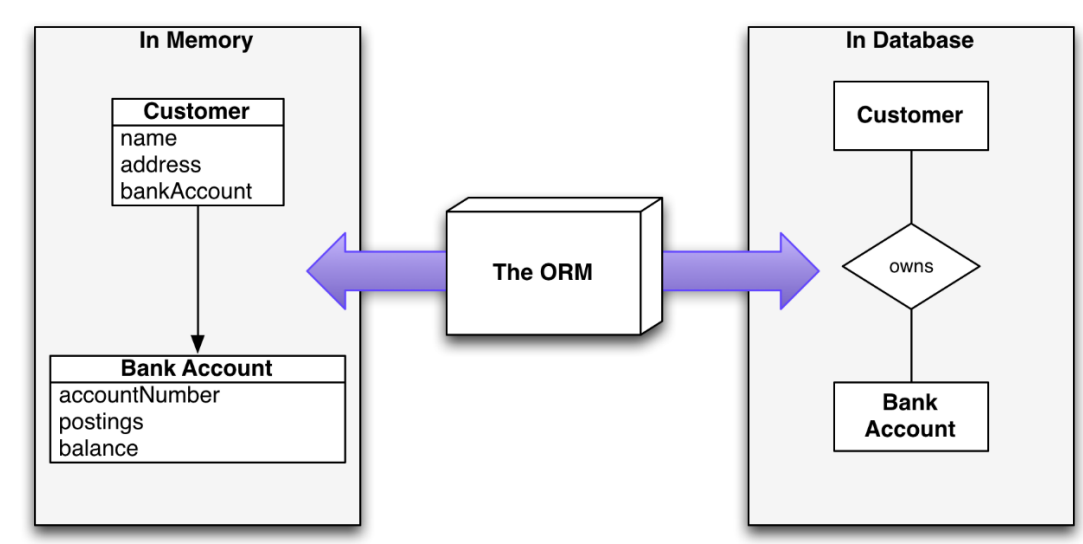

Figure 1: Mapeo Objeto-Relacional (ORM)

Por ejemplo, si trabajamos directamente con JDBC tendremos que descomponer el objeto para construir la sentencia `INSERT` del siguiente ejemplo

```
. . .

String sentenciaSql = "INSERT INTO personajes (nombre, descripcion, vida,
ataque)" +

  ") VALUES (?, ?, ?, ?)";

PreparedStatement sentencia = conexion.prepareStatement(sentenciaSql);
```

```
sentencia.setString(1, personaje.getNombre());

sentencia.setString(2, personaje.getDescripcion());

sentencia.setInt(3, personaje.getVida());

sentencia.setInt(4, personaje.getAtaque());

sentencia.executeUpdate();



if (sentencia != null)

  sentencia.close();

. . .
```

Si contamos con un framework como *Hibernate*, esta misma operación se traduce en unas pocas líneas de código en las que podemos trabajar directamente con el objeto Java, puesto que el framework realiza el mapeo en función de las anotaciones que hemos implementado a la hora de definir la clase, que le indican a éste con que tabla y campos de la misma se corresponde la clase y sus atributos, respectivamente.

```
@Entity

@Table(name="personajes")

public class Personaje {

  @Id // Marca el campo como la clave de la tabla

  @GeneratedValue(strategy = IDENTITY)

  @Column(name="id")

  private int id;

  @Column(name="nombre")

  private String nombre;

  @Column(name="descripcion")

  private String descripcion;

  @Column(name="vida")
```

```
  private int vida;

  @Column(name="ataque")

  private int ataque;



  public Personaje(. . .) {

    . . .

  }



  // getters y setters

}
```

Así, podemos simplemente establecer una sesión con la Base de Datos y enviarle el objeto, en este caso invocando al método `save` que se encarga de registrarlo en la Base de Datos donde convenga según sus propias anotaciones.

```
. . .

sesion = HibernateUtil.getCurrentSession();

sesion.beginTransaction();

sesion.save(personaje);

sesion.getTransaction().commit();

sesion.close();

. . .
```

Hibernate

En nuestro caso usaremos *Hibernate* como librería *ORM*, concretamente **Hibernate 5.2**. Habrá que tenerlo en cuenta puesto que algunas clases/métodos pueden variar entre diferentes versiones de este framework, especialmente a la hora de configurar (`hibernate.cfg.xml`) e implementar el gestor de sesiones (`HibernateUtil.java`)

Configuración

El fichero de configuración de hibernate `hibernate.cfg.xml` se debe crear directamente dentro de la carpeta `src` del proyecto y el propio framework *Hibernate* será el encargado de leerlo para obtener las sesiones que permitan conectar con la Base de Datos con el código que se implementa en el fichero `HibernateUtil.java` que se muestra justo después del primero.

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN"


"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

  <session-factory>

    <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</pro
perty>

    <property
name="hibernate.connection.url">jdbc:mysql://localhost/basededatos<
/property>

    <property
name="hibernate.connection.username">usuario</property>

    <property
name="hibernate.connection.password">contraseña</property>

    <property
name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prope
rty>

    <property name="hibernate.show_sql">true</property>

   </session-factory>

</hibernate-configuration>
```

HibernateUtil.java

```
public class HibernateUtil {
```

```
  private static SessionFactory sessionFactory;

  private static Session session;



  /**

   * Crea la factoria de sesiones

   */

  public static void buildSessionFactory() {



    Configuration configuration = new Configuration();

    configuration.configure();

    // Se registran las clases que hay que mapear con cada tabla de
la base de datos

    configuration.addAnnotatedClass(Clase1.class);

    configuration.addAnnotatedClass(Clase2.class);

    configuration.addAnnotatedClass(Clase3.class);

    . . .



    ServiceRegistry serviceRegistry = new
StandardServiceRegistryBuilder().applySettings(

      configuration.getProperties()).build();

    sessionFactory =
configuration.buildSessionFactory(serviceRegistry);

  }



  /**

   * Abre una nueva sesión
```

```
     */
    public static void openSession() {
      session = sessionFactory.openSession();
    }

    /**
     * Devuelve la sesión actual
     * @return
     */
    public static Session getCurrentSession() {

      if ((session == null) || (!session.isOpen()))
        openSession();

      return session;
    }

    /**
     * Cierra Hibernate
     */
    public static void closeSessionFactory() {

      if (session != null)
        session.close();
```

```
    if (sessionFactory != null)
      sessionFactory.close();
  }
}
```

Mapeo de entidades/relaciones con clases/atributos Java

En el caso de las entidades, se deben anotar tanto la propia clase como cada uno de los atributos (se puede hacer en el atributo o en su getter/setter) para indicar con qué tabla mapearla y cómo mapear los atributos con los campos que corresponda, respectivamente.

Las anotaciones que nos podemos encontrar para anotar una clase Java que debe ser mapeada con una tabla son:

- `@Entity` Indica que la clase es una tabla en la base de datos
- `@Table(name = "nombre_tabla", catalog = "nombre_base_datos")` Indica el nombre de la tabla y la base de datos a la que pertenece (este último parámetro no es necesario ya que esa información viene en el fichero de configuración)

En el caso de los atributos simples que deben ser mapeados con los campos de la tabla correspondiente:

- `@Id` Indica que un atributo es la clave
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` Indica que es un valor *autonumérico* (`PRIMARY KEY` en MySQL, por ejemplo)
- `@Column(name = "nombre_columna")` Se utiliza para indicar el nombre de la columna en la tabla donde debe ser mapeado el atributo

```
@Entity

@Table(name = "actor", catalog = "db_peliculas")

public class Actor {



  private Integer id;

  private String nombre;

  private Date fechaNacimiento;



  // Constructor/es
```

```
public Actor() { . . .}

. . .


@Id
@GeneratedValue(strategy = IDENTITY)
@Column(name = "id")
public Integer getId() {
  return this.id;
}


public void setId(Integer id) {
  this.id = id;
}


@Column(name = "nombre")
public String getNombre() {
  return this.nombre;
}


public void setNombre(String nombre) {
  this.nombre = nombre;
}


@Column(name = "fecha_nacimiento")
```

```
  public Date getFechaNacimiento() {
    return this.fechaNacimiento;
  }

  public void setFechaNacimiento(Date fechaNacimiento) {
    this.fechaNacimiento = fechaNacimiento;
  }

  . . .

  // El resto de métodos se implementan como siempre
  @Override
  public String toString() {
    return nombre;
  }

  . . .
}
```

Para el mapeo de las relaciones, además de crear el correspondiente objeto que permita mantener la relación entre las clases (de forma bidireccional), éstos atributos deben ser mapeados según convenga. Para todos los casos, en ambos casos se indicará el tipo de relación visto desde el lado correspondiente pero sólo se codificará la información de mapeo en uno de los lados:

- `@OneToOne` Indica que el objeto es parte de una relación 1-1
- `@ManyToOne` Indica que el objeto es parte de una relación N-1. En este caso el atributo sería el lado 1
- `@OneToMany` Indica que el objeto es parte de una relación 1-N. En este caso el atributo sería el lado N

- `@ManyToMany` Indica que el objeto es parte de una relación N-M. En este caso se indica la tabla que mantiene la referencia entre las tablas y los campos que hacen el papel de claves ajenas en la base de datos

En el otro lado de la relación indicaremos el tipo de relación acompañado de la anotación `@MappedBy` añadiendo el atributo de la otra clase donde se especifica toda la información sobre el mapeo

Relaciones 1-1

Figure 2: Relación 1 a 1

```
@Entity

@Table(name = "personajes")

public class Personaje {



  . . .

  private Arma arma;

  . . .

  @OneToOne(cascade = CascadeType.ALL)

  @PrimaryKeyJoinColumn

  public Arma getArma() { return arma; }

  . . .

}

@Entity

@Table(name = "armas")

public class Arma {



  . . .
```

```
  private Personaje personaje;

  . . .

  @OneToOne(cascade = CascadeType.ALL)

  @PrimaryKeyJoinColumn

  public Personaje getPersonaje() { return personaje; }

  . . .

}
```

Relaciones 1-N

Figure 3: Relación 1 a N

```
@Entity

@Table(name = "personajes")

public class Personaje {

  . . .

  private Arma arma;

  . . .

  @ManyToOne

  @JoinColumn(name="id_arma")

  public Arma getArma() { return arma; }

  . . .

}

@Entity
```

```
@Table(name = "armas")

public class Arma {

  . . .

  private List<Personaje> personajes;

  . . .

  @OneToMany(mappedBy = "arma", cascade = CascadeType.ALL)

  public List<Personaje> getPersonajes() { return personajes; }

  . . .
```

Relaciones N-M

Figure 4: Relación N a M

```
@Entity

@Table(name = "enemigos")

public class Enemigo {

  . . .

  private List<Arma> armas;

  . . .

  // Cuando se elimine un personaje se desvínculará el arma pero ésta no
se borrará (DETACH)

  @ManyToMany(cascade = CascadeType.DETACH)

  @JoinTable(name="enemigo_arma",

    joinColumns={@JoinColumn(name="id_enemigo")},

    inverseJoinColumns={@JoinColumn(name="id_arma")})
```

```
  public List<Arma> getArmas() { return armas; }

  . . .

}

@Entity

@Table(name = "armas")

public class Arma {

  . . .

  private List<Enemigo> enemigos;

  . . .

  @ManyToMany(cascade = CascadeType.DETACH, mappedBy = "armas")

  public List<Enemigo> getEnemigos() { return enemigos; }

  . . .

}
```

Operaciones sobre la Base de Datos

Registrar un objeto

Para registrar un nuevo objeto en la Base de Datos necesitamos haber creado previamente la clase y haberla mapeado correctamente con la tabla que le corresponda. Entonces, utilizando la clase `HibernateUtil` podremos obtener una sesión (conexión con la Base de Datos) para registrar ese objeto directamente en la Base de Datos de la siguiente forma.

```
. . .

UnaClase unObjeto = new UnaClase();

. . .

Session sesion = HibernateUtil.getCurrentSession();
```

```
sesion.beginTransaction();

sesion.save(unObjeto);

sesion.getTransaction().commit();

sesion.close();

. . .
```

Hay que tener en cuenta que entre el inicio y cierre de la transacción podemos realizar más de una operación y éstas se ejecutarán como tal. Es la forma correcta en el caso de que queramos registrar más de un objeto cuando éstos estén relacionados de alguna forma y dependan entre ellos. Un caso muy claro sería el del registro de un pedido junto con todas sus líneas de detalle puesto que no tendría sentido registrarlo sin los detalles, por lo que la forma más segura sería darlos de alta dentro de una misma transacción.

```
. . .

Session sesion = HibernateUtil.getCurrentSession();

sesion.beginTransaction();

sesion.save(unPedido);

for (DetallePedido detallePedido : detallesDelPedido)

  sesion.save(detallePedido);

sesion.getTransaction().commit();

sesion.close();

. . .
```

Modificar un objeto

En el caso de que queramos modificar un objeto, la operación se realiza de la misma forma que para el caso de registrar uno nuevo. Hibernate decide qué hacer (si registrar o modificar) comprobando si el objeto que se le envía tiene un valor válido para el campo `id`. Así, la única diferencia con el ejemplo anterior es que ahora dispondremos de un objeto que hemos obtenido previamente de la Base de Datos y al que hemos realizado algunas modificaciones (nunca el campo `id`).

```
. . .
```

```
Session sesion = HibernateUtil.getCurrentSession();

sesion.beginTransaction();

sesion.save(unObjeto);

sesion.getTransaction().commit();

sesion.close();

. . .
```

Eliminar un objeto

```
. . .

Session sesion = HibernateUtil.getCurrentSession();

sesion.beginTransaction();

sesion.delete(unObjeto);

sesion.getTransaction().commit();

sesion.close();

. . .
```

Búsquedas

Para el caso de las búsquedas se utiliza el lenguaje *HQL* (Hibernate Query Language), muy similar al lenguaje SQL que se usa en las base de datos relacionales, pero en este caso totalmente Orientado a Objetos puesto que en vez de trabajar con las tablas se trabaja con las clases y sus atributos directamente.

- Obtener un **objeto identificado por el id**

```
. . .

int id = . . .;

Cliente cliente = HibernateUtil.getCurrentSession().get(Cliente.class,
id);

. . .
```

- Obtener **todos los objetos** de una clase

```
. . .

Query query = HibernateUtil.getCurrentSession().createQuery("FROM
Cliente");

ArrayList<Cliente> clientes = (ArrayList<Cliente>) query.list();

. . .
```

- Obtener objetos de una clase **añadiendo algún criterio de búsqueda**

→ Si el criterio especificado nos **devuelve un solo objeto**:

```
. . .

String nombre = . . .;

. . .

Query query = HibernateUtil.getCurrentSession().

  createQuery("FROM Cliente c WHERE c.nombre = :nombre");

query.setParameter("nombre", nombre);

Cliente cliente = (Cliente) query.uniqueResult();

. . .
```

→ Si el criterio especificado nos **puede devolver más de un objeto**:

```
. . .

String ciudad = . . .;

. . .

Query query = HibernateUtil.getCurrentSession().

  createQuery("FROM Cliente c WHERE c.ciudad = :ciudad");

query.setParameter("ciudad", ciudad);

ArrayList<Cliente> clientes = (ArrayList<Cliente>) query.list();
```

```
. . .
```

- Obtener objetos de una clase utilizando las **relaciones entre clases**

```
. . .

Query query = HibernateUtil.getCurrentSession().

  createQuery("FROM DetallePedido dp WHERE dp.pedido.numeroPedido =
:numeroPedido");

query.setParameter("numeroPedido", numeroPedido);



ArrayList<DetallePedido> detalles = (ArrayList<DetallePedido>)
query.list();

. . .
```

- Y también es posible lanzar **consultas directamente en lenguaje SQL**, trabajando entonces directamente con las tablas y campos de la base de datos

```
. . .

SQLQuery sqlQuery = HibernateUtil.getCurrentSession().

  createSQLQuery("SELECT nombre, apellidos FROM clientes WHERE ciudad =
:ciudad");

query.setParameter("ciudad", ciudad);

List resultado = query.list();



for (Object objeto : resultado) {

  Map fila = (Map) objeto;

  String nombre = fila.get("nombre");

  String apellidos = fila.get("apellidos");

  . . .

}
```

```
. . .
```

Práctica 3.1

Objetivos

Desarrollar una aplicación que conecta con una Base de Datos Relacional utilizando Hibernate

Enunciado

Siguiendo el mismo diseño de la aplicación de las práctica 1.1 y 2.1, se deberá implementar una aplicación que conecte con una Base de Datos en MySQL, y utilizando Hibernate, según los requisitos que se enumeran a continuación

Requisitos (1 pto cada uno)

- La aplicación deberá conectar con una Base de Datos de forma transparente para el usuario, de forma que los datos de conexión puedan configurarse en un fichero a parte (fichero *properties*)
- El usuario tiene que poder dar de alta, modificar y eliminar datos de dos objetos relacionados entre sí
- Mostrar todos los registros en un `JList`
- Implementar un sistema de autenticación de usuarios para la aplicación
- Añadir alguna forma de búsqueda

Otras funcionalidades (1 pto cada una)

- Añadir una opción que permita exportar los datos de la aplicación (a JSON, XML, . . .)
- Añadir una opción que permita importar datos a la aplicación (JSON, XML, . . .)
- Realizar la aplicación creando y reutilizando algún componente propio
- Trabajar con, al menos, una relación N-M entre objetos de la aplicación
- Añadir soporte para multiusuario, implementando lo necesario para que varios usuarios simultáneos puedan trabajar con la aplicación sin que se produzcan problemas (por ejemplo, que dos usuarios estén modificando el mismo elemento)
- Utilizar la herramienta Git (y GitHub) durante todo el desarrollo de la aplicación. Utilizar el gestor de *Issues* para los problemas/fallos que vayan surgiendo
- Añadir una opción al usuario que permita recuperar el último elemento borrado
- Añadir una opción a la aplicación que permita eliminar todos los datos del programa