TEMA 12. ANÁLISIS E IMPLEMENTACIÓN DE BD ORIENTADAS A OBJETOS

Índice

Bases de Datos Orientadas a Objetos. Sistemas Gestores de Bases de Datos Orientados a Objetos

Las bases de datos orientadas a objetos surgen por la necesidad que tienen las aplicaciones orientadas a objetos de almacenar objetos en bases de datos. Hasta su aparición las bases de datos que se utilizaban para almacenar objetos eran las bases de datos relacionales. El hecho de almacenar objetos usando sistemas gestores de bases de datos relaciones implicaba una disminución del rendimiento de las aplicaciones orientadas a objetos ya que para almacenar un objeto en una base de datos había que convertirlo a una representación relacional y para recuperar un objeto había que transformar los datos leídos en un objeto.

En una base de datos orientada a objetos, la información se almacena en objetos.

Para poder gestionar bases de datos orientadas a objetos es necesario usar un **sistema gestor de base de datos orientado a objetos** (**ODBMS, Object Database Management System**). Un sistema gestor de bases de datos orientado a objetos almacena los objetos en la base de datos como objetos de un lenguaje de programación orientado a objetos.

Las bases de datos orientadas a objetos se diseñan para facilitar el trabajo con lenguajes de programación orientados a objetos como Java, C#, Visual Basic.NET y C++. Los sistemas gestores de base de datos orientados a objetos usan exactamente el mismo modelo que estos lenguajes de programación.

Un sistema gestor de base de datos orientado a objetos extiende los lenguajes con datos persistentes de forma transparente, control de concurrencia, recuperación de datos, consultas asociativas y otras capacidades.

Los sistemas gestores de base de datos orientados a objetos son una buena elección para aquellos sistemas que necesitan un buen rendimiento en la manipulación de objetos complejos.

Los sistemas gestores de base de datos orientados a objetos hacen que los costes de desarrollo sean más bajos y que aumente el rendimiento cuando se usan objetos gracias a que almacenan la información en disco como objetos directamente lo que permite que las aplicaciones orientadas a objetos recuperen los datos directamente, sin necesidad de conversión, de la base de datos.

Persistencia de Objetos

Cuando trabajamos con aplicaciones orientadas a objetos surge la necesidad de almacenar objetos con el objetivo de recuperarlos posteriormente. A la característica de almacenar un objeto completamente para poderlo recuperar más tarde también completamente se le conoce con el nombre de **persistencia de objetos**.

Cuando trabajamos con ficheros vimos que podíamos almacenar objetos en ficheros y recuperarlos más tarde usando el mecanismo de **serialización**.

Cuando en lugar de almacenar los objetos en un fichero los queremos almacenar en una base de datos tenemos que usar un **sistema gestor de bases de datos orientado a objetos** para realizar todas las operaciones necesarias para crear y manipular una base de datos orientada a objetos.

ObjectDB

ObjectDB es un **sistema gestor de bases de datos orientado a objetos para Java**. Se puede utilizar en modo local y en modo cliente-servidor.

ObjectDB al estar adaptado a Java permite el uso de una de las dos API estándar de Java **JPA** o **JDO**. Ambas APIs están incorporadas en ObjectDB, por lo que no es necesario ningún software adicional.

La **documentación oficial** se encuentra en el enlace https://www.objectdb.com/java/jpa

Para instalar ObjectDB basta con descargar la última versión disponible desde el sitio web http://www.objectdb.com/. Para ir directamente a la sección de descargas podemos ir a https://www.objectdb.com/download

Una vez descargada la última versión, tenemos que descomprimir el archivo con ObjectDB (en nuestro caso objectdb-2.8.x) en **c:\objectdb**.

Una vez copiados los datos, podemos **iniciar el servidor** ejecutando el archivo **c:\objectdb\bin\objectdb.jar**.

Si el servidor está funcionando aparece su **icono** en el **área de notificaciones**.

Las **bases de datos** se guardan localmente en **c:\objectdb\db**

Si queremos **administrar ObjectDB en modo gráfico** ejecutamos el archivo **c:\objectdb\bin\explorer.jar** que abre el **ObjectDB Explorer**.

Para acceder a las **bases de datos** ObjectDB que están **en la carpeta de instalación de objectdb** en el **equipo local** debemos usar **$objectdb** en la sentencia del createEntityManagerFactory. Por ejemplo, para acceder a la base de datos de nombre personajes.odb que está en la carpeta db del equipo local escribimos

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("$objectdb/db/personajes.odb");
```

Salvo cuando estamos empezando no es conveniente guardar todas las bases de datos en la carpeta de instalación de objectdb en el equipo local. Es mejor guardar cada **base de datos dentro de su proyecto** correspondiente por ejemplo dentro de una carpeta de nombre db de ese proyecto. Para acceder a las **bases de datos** ObjectDB que están en el **proyecto** usar **objectdb:** en la sentencia del createEntityManagerFactory. Por ejemplo, para acceder a la base de datos de nombre personajes.odb que está en la carpeta db del proyecto escribimos

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("objectdb:db/personajes.odb");
```

ObjectDB en Eclipse. Crear un Nuevo Proyecto

Una de las ventajas de **ObjectDB** es que permite una integración fácil y total con Eclipse.

En Eclipse, para que un proyecto Java pueda trabajar con bases de datos ObjectDB basta con **añadir al Build Path** del proyecto Java la librería **objectdb.jar** que se encuentra en **c:\objectdb\bin\.**

Por ejemplo, si queremos **crear un nuevo proyecto Java** de nombre ObjectDB en Eclipse vamos a **File → New → Java Project**. En **Project Name:** escribimos **ObjectDB** y pulsamos Next.

En la siguiente ventana hacemos clic sobre la pestaña **Libraries**., seleccionamos **Classpath** (si no seleccionamos Classpath y seleccionamos Modulepath da problemas) y hacemos clic en **Add External JARs...**

Navegamos por el sistema para seleccionar **c:\objectdb\bin\objectdb.jar**. Al seleccionarlo aparece en la pestaña Libraries.

Para finalizar la creación del proyecto Java pulsamos el botón Finish.

ObjectDB. Crear una Clase Java que Represente una Entidad en ObjectDB

Para almacenar objetos en una base de datos ObjectDB es necesario crear una clase Java por cada entidad de la base de datos.

Una entidad es una clase Java que permite la serialización de objetos y que incluye un dato que actuará como clave primaria dentro de la base de datos.

Por ejemplo, vamos a crear un **paquete** de nombre **personajes** dentro del proyecto anterior. Dentro de ese paquete vamos a crear una **clase Java** de nombre **Personaje**. Para hacer que la clase Java defina una entidad debemos importar **java.io.Serializable** y **javax.persistence.***, escribir antes de la definición de la clase **@Entity** y escribir antes del campo que va a ser la clave primaria **@Id** (podemos añadir **@GeneratedValue** para que el id lo genere de manera automática).

Para definir la clase Java Personaje que representa una entidad de la base de datos ObjectDB escribimos lo siguiente

```
import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Personaje implements Serializable {

 private static final long serialVersionUID = 20230323L;
 @Id @GeneratedValue
 private long idPersonaje;

 private String nombre;
 private int nivel;

 public Personaje() {
  nombre = "SinNombre";
  nivel = 1;
 }

 public Personaje(String no, int ni) {
  this.nombre = no;
  this.nivel = ni;
 }

public Personaje(Personaje p) {
 this.nombre = p.nombre;
 this.nivel = p.nivel;
}

public long getIdPersonaje() {
 return idPersonaje;
}

public void setIdPersonaje(long idPersonaje) {
```

```
 this.idPersonaje = idPersonaje;
}

public String getNombre() {
 return nombre;
}

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

public int getNivel() {
 return nivel;
}

public void setNivel(int nivel) {
 this.nivel = nivel;
}

 @Override
 public String toString() {
  return ("Nombre: " + this.nombre + " Nivel: " + this.nivel);
 }
}
```

ObjectDB. Trabajar con Bases de Datos ObjectDB

Para trabajar con una base de datos ObjectDB es necesario crear una clase Java que incluya una función **main** dentro de la cual se escribirán las instrucciones que queramos ejecutar sobre la base de datos ObjectDB.

Para trabajar con una base de datos ObjectDB que contenga objetos de la clase Personaje vamos a crear la clase Java **PersonajeMain**. Dentro de la clase vamos a explicar el funcionamiento de cada una de las instrucciones. El contenido de la clase es el siguiente

```
import javax.persistence.*;
import java.util.*;

public class PersonajeMain {
 public static void main(String[] args) {
  // Se conecta a la base de datos
  // crea una base de datos si todavia no existe
  EntityManagerFactory emf = Persistence.createEntityManagerFactory("$objectdb/db/personajes.odb");
  EntityManager em = emf.createEntityManager();

  // Creo 10 Objetos de tipo Personaje en la base de datos
  em.getTransaction().begin();
  String nombre;
  int nivel;
  for (int i = 0; i < 10; i++) {
   // creo los datos de un Personaje
   nombre = "P"+ i;
   nivel = i;
   // creo un nuevo Personaje
   Personaje p = new Personaje(nombre, nivel);
   // inserto el Personaje en la Base de Datos
   em.persist(p);
  }
 // Guardo los cambios en la Base de Datos
 em.getTransaction().commit();

 // Muestro el numero de objetos de tipo Personaje en la base de datos
```

```
 Query q1 = em.createQuery("SELECT COUNT(p) FROM Personaje p");
 System.out.println("Total de Personajes: " + q1.getSingleResult());

 // Muestro el nivel medio de los Personajes de la base de datos
 Query q2 = em.createQuery("SELECT AVG(p.nivel) FROM Personaje p");
 System.out.println("Nivel Medio: " + q2.getSingleResult());

 // Muestro todos los objetos de tipo Personaje de la base de datos
 TypedQuery<Personaje> tq1 =
 em.createQuery("SELECT p FROM Personaje p", Personaje.class);
 List<Personaje> results = tq1.getResultList();
 for (Personaje p : results) {
  System.out.println(p);
 }

 // Muestro todos los objetos de tipo Personaje de la base de datos con nivel 5
 TypedQuery<Personaje> tq2 =
 em.createQuery("SELECT p FROM Personaje p WHERE p.nivel = 5", Personaje.class);
 List<Personaje> results2 = tq2.getResultList();
 for (Personaje p : results2) {
  System.out.println(p);
 }

 // Borro todos los Personaje con id > 9
 Query q3 = em.createQuery("DELETE FROM Personaje p WHERE p.idPersonaje > 9");
 em.getTransaction().begin();
 int registrosafectados;
 registrosafectados = q3.executeUpdate();
 em.getTransaction().commit();
 if(registrosafectados > 0) {
  System.out.println("Se han Borrado "+ registrosafectados + " objetos de la clase Personaje con
idPersonaje > 9");
 }
 else {
  System.out.println("No se ha Borrado ningun Personaje con idPersonaje > 9");
 }
 // Actualizo a nombre 'N-1' y  nivel -1 todos los Personaje con id > 5
 Query q4 = em.createQuery("UPDATE Personaje p SET nombre = 'N-1', nivel = -1 WHERE p.idPersonaje >
5");
 em.getTransaction().begin();
 int registrosafectados = q4.executeUpdate();
 em.getTransaction().commit();
 if(registrosafectados > 0) {
  System.out.println("Se han Actualizado "+ registrosafectados + " objetos de la clase Personaje con
idPersonaje > 5");
 }
 else {
  System.out.println("No se ha Actualizado ningun Personaje con idPersonaje > 5");
 }
 // Cierro la conexion con la base de datos
 em.close();
 emf.close();
 }
}
```

Cuando se ejecutan algunas consultas como DELETE o UPDATE es muy importante comprobar cuantos registros han sido modificados ya que las consultas de modificación o borrado se pueden ejecutar correctamente pero no afectar a ningún registro (porque no existe un registro con esos datos).

ObjectDB en Maven.

Para eliminar los problemas que nos puede ocasionar el hecho de importar el jar de ObjectDB a nuestros proyectos Java en Eclipse, podemos crear en Eclipse un proyecto Maven e incluir la dependencia ObjectDB.

Por ejemplo, para realizar lo mismo que en el apartado anterior con la Base de Datos Orientada a Objetos de Personajes, vamos a crear un proyecto Maven de nombre **ObjectDBMaven** en Eclipse con Group Id **org.apache.maven.archetypes** y como Artifact Id **maven-archetypes-quickstart**.

Copiamos la clase **Personaje** dentro de src/main/java y cambiamos el código de la clase Java de nombre **App**. por el de la clase **PersonajeMain** del ejemplo anterior.

Podemos consultar las dependencias Maven de ObjectDB en https://www.objectdb.com/download

En nuestro caso tenemos que modificar el archivo pom.xml de nuestro proyecto Maven de la siguiente manera

```
…
<repositories>
 <repository>
  <id>objectdb</id>
  <name>ObjectDB Repository</name>
  <url>https://m2.objectdb.com</url>
 </repository>
</repositories>
…
<dependency>
 <groupId>com.objectdb</groupId>
 <artifactId>objectdb</artifactId>
 <version>2.8.7</version>
</dependency>
<dependency>
 <groupId>org.eclipse.persistence</groupId>
 <artifactId>javax.persistence</artifactId>
 <version>2.1.0</version>
</dependency>
<dependency>
 <groupId>javax.transaction</groupId>
 <artifactId>jta</artifactId>
 <version>1.1</version>
</dependency>
```

Ejecutamos la aplicación como Aplicación Java y comprobamos el resultado. Inicialmente la base de datos está vacía por lo que debemos añadir datos de prueba usando, por ejemplo, la clase Java PersonajeMainCrear10.

Si queremos crear un fichero **JAR ejecutable que incluya las dependencias** necesarias para poder conectarse a una base de datos ObjectDB debemos incluir, o modificar si ya existe, en el **pom.xml** de nuestro proyecto Maven el plugin siguiente

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-assembly-plugin</artifactId>
<executions>
        <execution>
                <phase>package</phase>
                <goals>
                        <goal>single</goal>
                </goals>
                <configuration>
                        <archive>
                        <manifest>
                <mainClass>com.fptxurdinaga.MavenVentanaJTableObjetos.App</mainClass>
```

```
                        </manifest>
                        </archive>
                        <descriptorRefs>
                                <descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                </configuration>
        </execution>
</executions>
</plugin>
```

En el ejemplo anterior la clase principal del proyecto Maven se llama **MavenVentanaJTableObjetos**.
El plugin maven-jar-plugin lo podemos quitar ya que ahora estamos usando el maven-assembly-plugin para generar el JAR.
Ejecutamos el proyecto Maven con un clean install en Run Configurations (marcando opcionalmente Skip Tests) y, si todo ha ido bien, en la carpeta target del proyecto aparecerá el archivo JAR **MavenVentanaJTableObjetos-0.0.1-SNAPSHOT-jar-with-dependencies.jar** que podemos ejecutar directamente.

Aunque el **JAR** incluye las dependencias **no incluye** el fichero que contiene la **base de datos ObjectDB**. Si queremos que al ejecutar el JAR se cree una base de datos nueva no hace falta que hagamos nada.

Sin embargo, si queremos que al ejecutar el JAR se **carguen los datos de una base de datos ObjectDB que ya existe** tendremos que **copiar el archivo** que contiene la base de datos ObjectDB en la ruta que le hayamos indicado en la aplicación.

Por ejemplo, si hemos creado la base de datos ObjectDB de nombre **personajes.odb** dentro de la **carpeta db** de nuestra aplicación tendremos que crear una carpeta de nombre db en el directorio donde esté el archivo JAR y copiar dentro de la carpeta db el fichero personajes.odb

Ejercicios de Bases de Datos Orientadas a Objetos.

1. Crea la clase Java **PersonajeMainCrear10** que crea 10 objetos de la clase **Personaje** en la la base de datos ObjectDB **personajes.odb** que se encuentra en el **directorio db del proyecto**.

2. Crea la clase Java **VentanaJTableObjetos** que modifica la clase **VentanaJTableActualizar** y carga en un JTable los datos de la tabla **Personaje** de la base de datos **ObjectDB personajes.odb**. En la parte superior hay una JLabel de nombre lblNombre con el texto Nombre, un JtTxtField de nombre txtNombre y con texto “N0”, una JLabel de nombre lblNivel con el texto Nivel, y un JTextField de nombre txtNivel y con texto “0”. En la parte inferior hay un JButton de nombre **btnInsertar** con el texto Insertar que al hacer clic sobre él inserta un nuevo registro con los datos de los campos, un JButton de nombre **btnBorrar** con el texto Borrar que al hacer clic sobre él borra los registros seleccionados en la tabla, si es que hay algún registro seleccionado, un JButton de nombre **btnActualizar** con el texto Actualizar en la parte inferior que al hacer clic sobre él modifica el valor de los registros seleccionados en la tabla, si es que hay algún registro seleccionado, con el valor **de los campos**, y un botón de nombre btnSalir que al pulsar sobre él se saldrá de la aplicación. Los datos de se deben de actualizar también en la tabla para reflejar los cambios.

Crear una Entidad de una Clase Java que Hereda de Otra

Cuando diseñamos el diagrama de las Entidades de la Base de Datos tenemos que elegir entre varias estrategias a la hora de crear Entidades cuando existen clases que heredan de otras.

Una de las estrategias consiste en definir la clase base en vez de como @Entity como **@MappedSuperclass** y la clase derivada como **@Entity**.

Por ejemplo, si la clase base es Persona y la clase derivada es Alumno la definición de las clases sería

```
@MappedSuperclass
public class Persona{
}
```

```
@Entity
public class Alumno extends Persona{
```

```
}
```

Crear una Entidad con una Propiedad cuyo tipo es una Clase Java

En algunas situaciones a la hora de definir una entidad nos vamos a encontrar con que una de las propiedades tiene como tipo una clase java.

Por ejemplo, al definir la clase Persona nos encontramos con que la propiedad **fechanacimiento** es del tipo **Fecha**.

En estos casos, para que la definición de la entidad se pueda realizar correctamente la clase que se usa, Fecha en este caso, tiene que ser definida como **@Embeddable**.

```
@Embeddable
public class Fecha{
}
```

```
@MappedSuperclass
public class Persona{
 @Id
 private String dni;
 private String nombre;
 private String apellidos;
 private Fecha fechanacimiento;
}
```

Cuando los objetos a actualizar tienen alguna propiedad que es a su vez un objeto no se pueden utilizar los datos usando sentencias UPDATE sino que hay que buscar el objeto a modificar dentro del EntityManager, modificar las propiedades del objeto que nos interesen, y guardar los datos modificados haciendo commit.

Por ejemplo, para modificar los datos de un objeto de la clase Alumno que tiene como dni “11111111A” con los datos de unos campos de texto escribimos

```
String dni = "11111111A";
// lo cargo desde la base de datos
Alumno a = em.find(Alumno.class, dni);
// actualizo sus datos
String nombre = this.txtNombre.getText();
a.setNombre(nombre);
String apellidos = this.txtApellidos.getText();
a.setApellidos(apellidos);
int dia = Integer.parseInt(this.txtDia.getText());
int mes = Integer.parseInt(this.txtMes.getText());
int año = Integer.parseInt(this.txtAño.getText());
Fecha fn = new Fecha(dia,mes,año);
a.setFechanacimiento(fn);
String grupo = (String) this.cmbGrupos.getSelectedItem();
a.setGrupo(grupo);
// guardo los cambios de la actualización
em.getTransaction().commit();
```

Ejercicios de Entidades de Clases Java que Heredan de Otras

3. Crea un paquete de nombre **alumnos** dentro del proyecto Java de nombre ObjectDB.

4. Crea dentro del paquete alumnos la clase Java **AlumnoMainCrear10** que crea 10 objetos de la clase Alumno en la la base de datos ObjectDB bdalumnos.odb que se encuentra en el directorio db del proyecto.

5. Crea dentro del paquete alumnos la clase Java **VentanaJTableObjetosAlumnos** que modifica la clase VentanaJTableObjetos para que el tratamiento se haga ahora con objetos de la clase Alumno. Los objetos de la clase Alumno se guardarán en la base de datos ObjectDB bdalumnos.odb que se encuentra en el directorio db del proyecto.

6. Crea dentro del paquete alumnos la clase **Java AsignaturaMainCrear10** que crea 10 objetos de la clase Asignatura en la la base de datos ObjectDB bdalumnos.odb.

7. Crea dentro del paquete alumnos la clase Java **AsignaturaMainCrear1DW3** que los objetos de la clase Asignatura para las asignaturas de 1DW3 en la la base de datos ObjectDB bdalumnos.odb.

8. Crea la clase Java **VentanaJTableObjetosAsignaturas** que modifica la clase VentanaJTableObjetosAlumnos para que el tratamiento se haga ahora con objetos de la clase Asignatura.

Crear Relaciones entre Entidades

En algunas situaciones es necesario definir relaciones entre entidades, por ejemplo, la relación entre un Alumno y sus Asignaturas que da lugar a sus Calificaciones.

En estos casos para simular la relación debemos incluir en las dos clases principales, Alumno y Asignatura en este caso, una propiedad de tipo List del tipo de la clase de la relación, Calificacion en este caso. Esta propiedad de tipo List deberá contar con una anotación que indique el tipo de relación entre la entidad principal, Alumno, y la tabla de la relación Calificacion.

```
@OneToMany(mappedBy = "alumno", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Calificacion> calificaciones;
```

Lo mismo pasará con la clase Asignatura

```
@OneToMany(mappedBy = "asignatura", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Calificacion> calificaciones;
```

Además, en las clases principales Alumno y Asignatura hay que añadir un método de nombre **agregarCalificacion** que permita añadir una nueva Calificacion tanto a Alumno como a Asignatura.

```
public void agregarCalificacion(Calificacion c){
 this.calificaciones.add(c);
}
```

El contenido de la clase Calificacion sería

```
@Entity
@Table(name = "calificaciones")
public class Calificacion {
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "alumno_dni")
  private Alumno alumno;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "asignatura_codasignatura")
  private Asignatura asignatura;

  @Column(name = "nota")
  private double nota;

  public Calificacion() {

  }

  public Calificacion(Alumno alumno, Asignatura asignatura, double nota) {
     this.alumno = alumno;
     this.asignatura = asignatura;
     this.nota = nota;
  }
}
```

Cada objeto de la clase Calificación incluye un objeto de la clase Alumno y otro de la clase Asignatura.

Para añadir una nueva calificación a la base de datos es necesario tener primero los datos del Alumno y de la Asignatura de esa calificación.

Una vez que tenemos los datos del Alumno y la Asignatura creamos la Calificacion.

Una vez creada la Calificacion se la tenemos que añadir tanto al Alumno como a la Asignatura usando el método agregarCalificacion.

Para finalizar guardamos la Calificacion en la base de datos.

Podemos comprobar si todo ha ido bien con el Explorer de ObjectDB.

Por ejemplo, si queremos guardar los datos de una Calificacion c11 con los datos del Alumno a1 y la asignatura as1 escribimos

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("objectdb:db/bdalumnos.odb");
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

// creo un nuevo Alumno
Alumno a1 = new Alumno(new Persona("D1","N1","A1",new Fecha()),"1DW3");
// inserto el Alumno en la Base de Datos
em.persist(a1);

// creo una nueva Asignatura
Asignatura as1 = new Asignatura("AS01","Asignatura 1","Prueba Asignatura 1");
// inserto la Asignatura en la Base de Datos
em.persist(as1);

// creo una nueva Calificacion con los datos del Alumno a1 y la Asignatura as1
Calificacion c11 = new Calificacion(a1,as1,6.0);
// inserto la calificacion en el Alumno a1
a1.agregarCalificacion(c11);
// inserto la calificacion en la Asignatura as1
as1.agregarCalificacion(c11);

// inserto la Calificacion en la Base de Datos
em.persist(c11);

// Guardo los cambios en la Base de Datos
em.getTransaction().commit();

// Cierro la conexion con la base de datos
em.close();
emf.close();
```

Ejercicios de Relaciones entre Entidades

9. Crea la clase Java **CalificacionMainCrear** que crea unos objetos de la clase Calificacion en la la base de datos **ObjectDB bdalumnos.odb**. Después de crearlos abre el Explorer del ObjectDB para comprobar que se hayan creado con éxito.

10. Crea la clase Java **VentanaJTableObjetosCalificaciones** que modifica la clase **VentanaJTableObjetosAlumnos** para trabajar ahora con objetos de la clase Calificacion. Los objetos de la clase Alumno se cargarán en una ComboBox de nombre cmbAlumnos y los objetos de la clase Asignatura se cargarán en una ComboBox de nombre cmbAsignaturas. Si no hay alumnos o asignaturas no se podrá añadir ninguna Calificacion. Actualizar solo permite actualizar la nota.

11. Crea la clase Java **VentanaJTableObjetosCalificacionesCodigos** que modifica la clase VentanaJTableObjetosCalificaciones añadiendo una ComboBox para el dni de los alumnos y otra ComboBox para el código de las Asignaturas. Estas ComboBox serán visibles mientras que la ComboBox cmbAlumnos y la ComboBox cmbAsignaturas estarán ahora ocultas. Al seleccionar una Calificacion en la tabla se seleccionarán los datos de esa Califacion en las ComboBox.

12. Crea la clase Java **BDObjetosAlumnosCompleto** que modifica la clase **BDAlumnosCompleto** para que el tratamiento se haga ahora con objetos de las clases Alumno, Asignatura, y Calificacion.

Crear Entidades usando Ingeniería Inversa

Existen herramientas que permiten crear automáticamente las Entidades usando Ingeniería Inversa. Para ello se conectan a la base de datos MySQL y, en función de los datos almacenados en el diccionario de datos de la base de datos, generan las Entidades.

Una de las herramientas de Ingeniería Inversa más utilizadas es Hibernate Tools.
https://tools.jboss.org/features/hibernate.html

Hibernate Tools se puede integrar en Eclipse para realizar todo el proceso de Ingeniería Inversa. Podemos encontrar un ejemplo de como hacerlo en el enlace
https://www.codejava.net/frameworks/hibernate/java-hibernate-reverse-engineering-tutorial-with-eclipse-and-mysql

Aunque al principio pueda parecer útil, la generación de Entidades usando Ingeniería Inversa no funciona de modo ideal y corregir los fallos o defectos que se producen lleva más tiempo que generar por separado la base de datos MySQL y las Entidades.

Por ello, recomiendo no usar la Ingeniería Inversa y generar por separado la base de datos MySQL y las Entidades.