

Diseño y Desarrollo de Sistemas de Información

Mapeo Objeto Relacional
(JPA - Hibernate)

Objetivos

- Conocer y entender la técnica de Mapeo Objeto Relacional
- Aprender a utilizar una implementación concreta de un software de Mapeo Objeto Relacional (ORM)
- Aprender a implementar operaciones CRUD en Java utilizando una herramienta de ORM

Mapeo Objeto Relacional. Conceptos y definiciones

- Un ORM es un modelo de programación que permite "mapear" las estructuras de una **base de datos relacional** (SQL Server, Oracle, MySQL, etc.) sobre una estructura lógica de entidades para simplificar y acelerar el desarrollo de nuestras aplicaciones
- Las estructuras de la base de datos relacional quedan vinculadas con las entidades lógicas o base de datos virtual definida en el ORM, de tal modo que las acciones **CRUD** (*Create, Read, Update, Delete*) se realizan, de forma indirecta, a través del ORM

- Las herramientas de ORM, además de generar código de forma automática, utilizan un lenguaje propio para realizar las consultas y gestionar la persistencia de los datos
- Los objetos o entidades de la base de datos virtual creada por el ORM pueden ser gestionados con lenguajes de propósito general
- La interacción con el SGBD se realizará mediante los métodos propios del ORM
- Interactuar directamente con las entidades de la base de datos virtual sin necesidad de generar código SQL puede generar importantes ventajas a la hora de acelerar el desarrollo o implementación de las aplicaciones

Ventajas e inconvenientes de los ORM

■ Ventajas

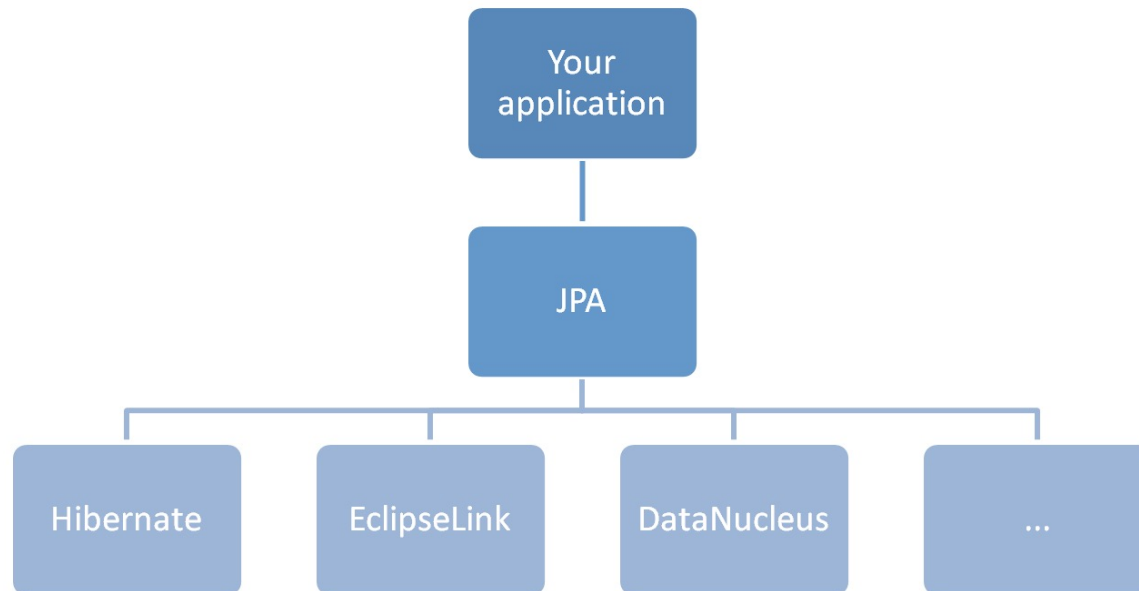
- No es necesario escribir "mucho" código SQL. Algunos programadores no lo dominan y, a veces, puede resultar complejo y propenso a errores
- Permite aumentar la reutilización del código y mejorar el mantenimiento del mismo
- Reduce el tiempo de desarrollo
- Mayor seguridad, evitando posibles ataques de inyección SQL y similares
- Están bien optimizados para las operaciones CRUD *insert*, *delete* y *update*, aunque para la recuperación (*select*) es mejor usar SQL nativo

■ Inconvenientes

- En entornos con gran carga puede reducir el rendimiento, ya que se está agregando una capa extra al sistema
- Su aprendizaje puede llegar a ser complejo
- Si las consultas son complejas, el ORM no garantiza una buena optimización, tal y como lo realizaría un desarrollador de forma nativa. De ahí que las implementaciones de ORM ofrecen extensiones para escribir consultas en SQL nativo

¿Qué es JPA? ¿Qué es Hibernate?

- **JPA** (*Java Persistence API*) es la especificación que define el funcionamiento de la persistencia de objetos en Java
- **Hibernate** es uno de los *frameworks* que implementa la especificación JPA. Hay otros como *EclipseLink*, *DataNucleus*, *TopLink*, etc.



Diferencia entre JDBC e Hibernate

```
...  
    ps = conexion.getConnection().prepareStatement("INSERT INTO MONITOR VALUES (?, ?, ?, ?, ?, ?, ?)");  
    ps.setString(1, monitor.getCodMonitor());  
    ps.setString(2, monitor.getNombre());  
    ps.setString(3, monitor.getDni());  
    ps.setString(4, monitor.getTelefono());  
    ps.setString(5, monitor.getCorreo());  
    ps.setString(6, monitor.getFechaEntrada());  
    ps.setString(7, monitor.getNick());  
  
    ps.executeUpdate();  
    ps.close();  
...
```

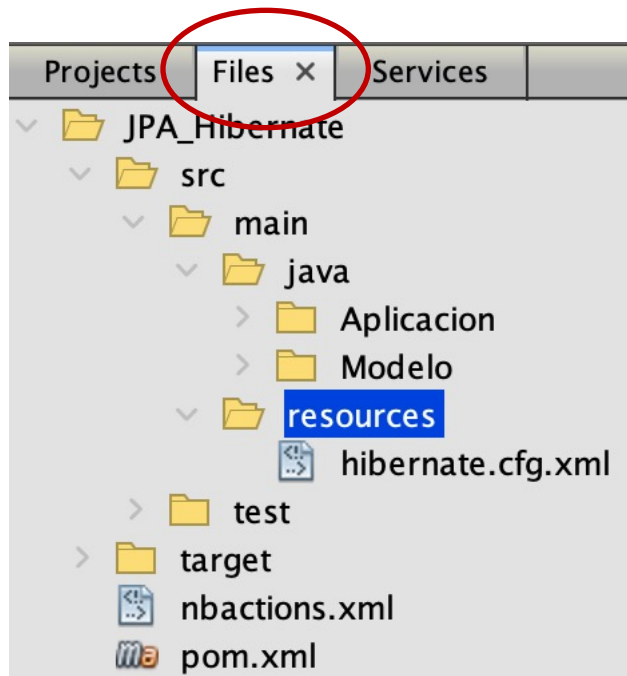
```
...  
    Session sesion = HibernateUtil.getSessionFactory().openSession();  
    sesion.beginTransaction();  
  
    sesion.save(monitor);  
  
    sesion.getTransaction().commit();  
  
    sesion.close();  
...
```

Dependencia MAVEN

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-tools-maven-plugin</artifactId>  
  <version>5.6.0.Final</version>  
  <type>maven-plugin</type>  
</dependency>
```


Fichero de configuración de Hibernate

- Hibernate utiliza un fichero de configuración, denominado **hibernate.cfg.xml**
- Este fichero debe situarse en la carpeta raíz del proyecto. Concretamente en **src/main/resources**



- Si no existe la carpeta *resources*, deberá crearse
- Una vez en la carpeta, con el botón derecho, seleccionar "Nuevo" + "Fichero xml"

Fichero de configuración de Hibernate

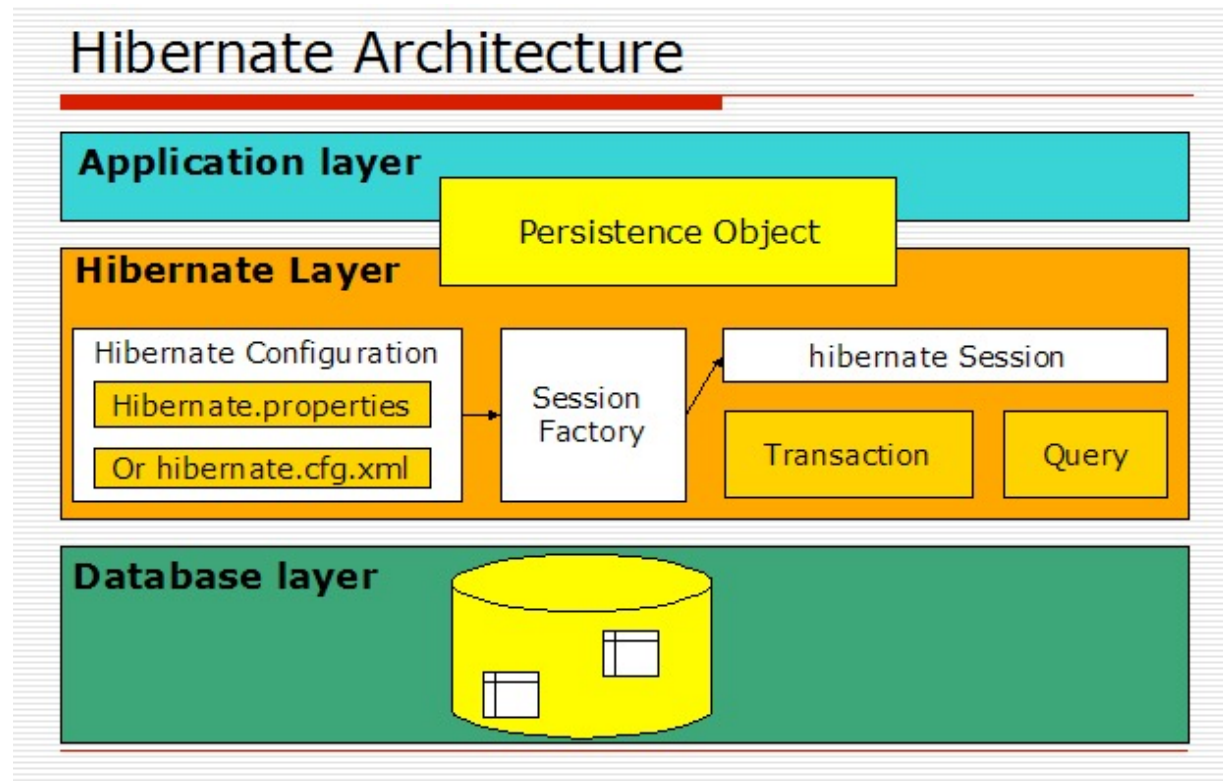
■ Ejemplo

```
<?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">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">oracle.jdbc.OracleDriver</property>
    <property name="hibernate.connection.url">jdbc:oracle:thin:@172.17.20.75:1521:rabida</property>
    <property name="hibernate.connection.username">usuario</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
    <property name="hibernate.show_sql">>true</property>
    <mapping class="Modelo.Monitor"/>
    <mapping class="Modelo.Actividad"/>
    <mapping class="Modelo.Socio"/>
  </session-factory>
</hibernate-configuration>
```

- Las etiquetas <mapping class> definen las clases que se mapean con las tablas de la base de datos
- Las tablas que surgen de las relaciones "muchos a muchos" no se mapean en la aplicación (su funcionamiento se verá más adelante)

Conexión y comunicación entre la aplicación y la base de datos

- Las clases que establecen la conexión y la comunicación entre la base de datos y la aplicación son *SessionFactory* y *Session*
- La clase *SessionFactory*, junto con sus métodos principales, suele crearse en la clase *HibernateUtil*, que situaremos en cualquier parte del proyecto



■ Ejemplo de *HibernateUtil.java*

```
public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
                .configure("hibernate.cfg.xml").build();
            Metadata metadata = new MetadataSources(serviceRegistry).getMetadataBuilder().build();
            return metadata.getSessionFactoryBuilder().build();

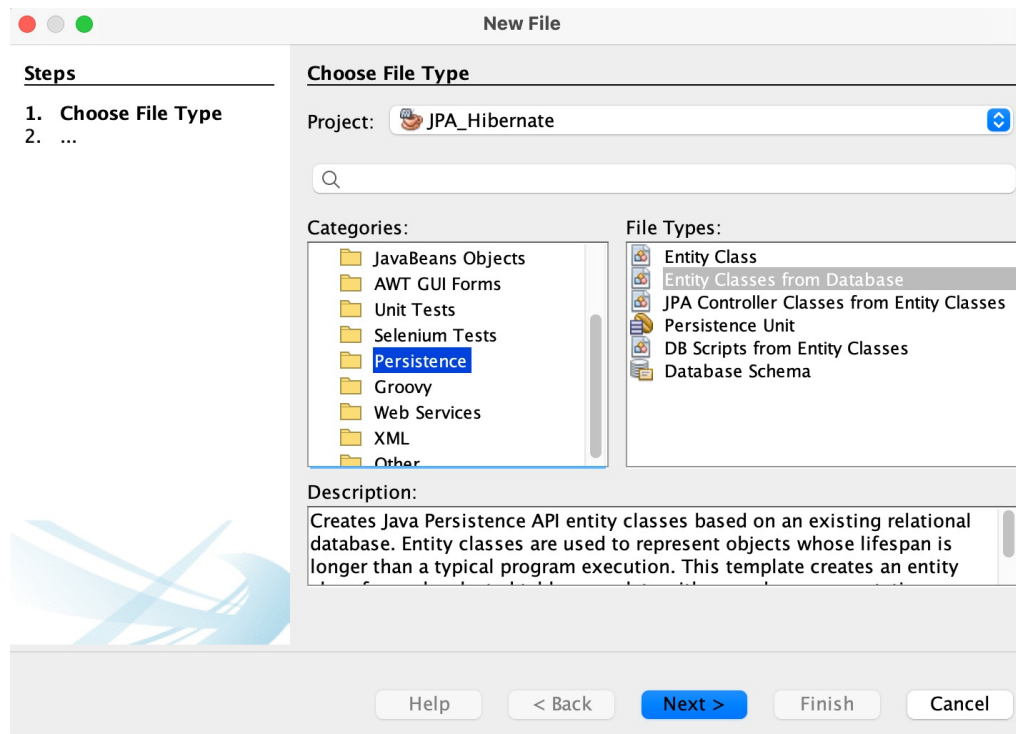
        } catch (Throwable ex) {
            System.err.println("Build SeesionFactory failed :"+ ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() { return sessionFactory; }

    public static void close() {
        if ((sessionFactory!=null) && (sessionFactory.isClosed()==false)) {
            sessionFactory.close();
            sessionFactory.close();
        }
    }
}
```

Creación de las clases derivadas de las tablas de la BD

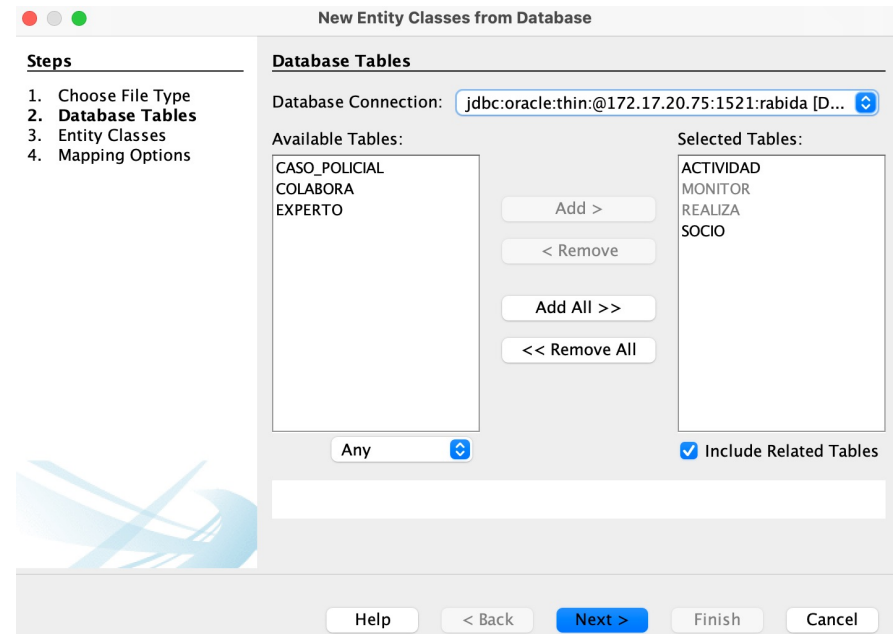
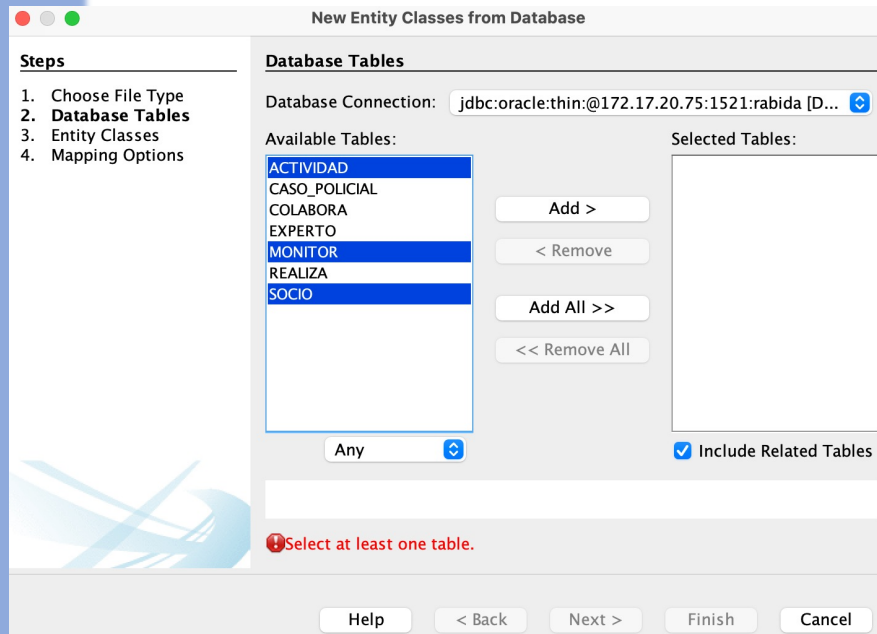
- El siguiente paso consiste en crear las clases sobre las que mapearemos las tablas de la base de datos
- Como casi siempre, se puede hacer de forma manual o con alguna utilidad existente. En este caso, nos apoyaremos en una funcionalidad de *NetBeans*
- La funcionalidad se llama **"Entity classes from databases"**, que se encuentra dentro del módulo **"Persistence"**



Creación de las clases derivadas de las tablas de la BD

IMPORTANTE

- En JPA/Hibernate **no se mapean las tablas "intermedias"** que surgen de las relaciones "muchos a muchos"



Creación de las clases derivadas de las tablas de la BD

Steps

1. Choose File Type
2. Database Tables
3. **Entity Classes**
4. Mapping Options

Entity Classes

Specify the names and the location of the entity classes.

Class Names:	Database Table	Class Name	Generation Type
	ACTIVIDAD	Actividad	New
	MONITOR	Monitor	New
	REALIZA	join table	New
	SOCIO	Socio	New

Project: PruebaHibernate

Location: Source Packages

Package: Modelo

☒ Generate Named Query Annotations for Persistent Fields

☐ Generate JAXB Annotations

☐ Generate MappedSuperclasses instead of Entities

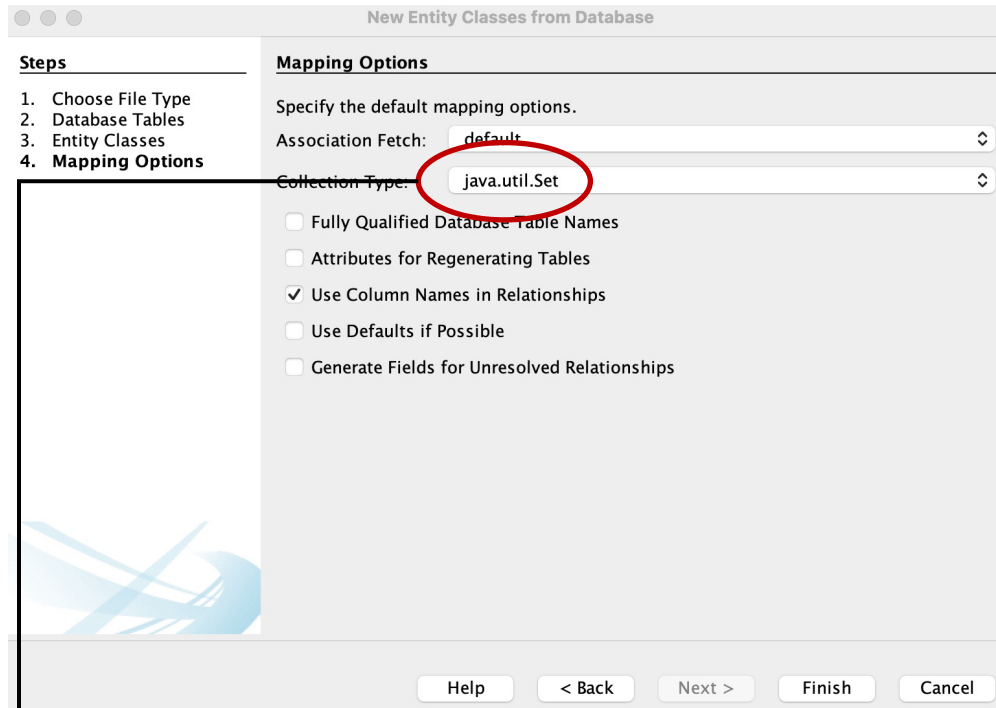
☐ Create Persistence Unit

⚠ The project does not have a persistence unit. You need a persistence unit to persist...

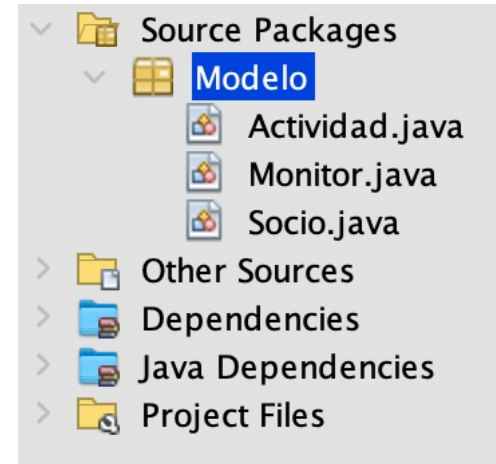
Help < Back Next > Finish Cancel

→ Deseleccionar "Create Persistence Util"

Creación de las clases derivadas de las tablas de la BD



Seleccionar Set como tipo de datos para almacenar las colecciones de los atributos



■ Clase Monitor (1/2)

```
@Entity
@Table(name = "MONITOR")
@NamedQueries({
    @NamedQuery(name = "Monitor.findAll", query = "SELECT m FROM Monitor m"), ... })
```

```
public class Monitor implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "CODMONITOR")
    private String codmonitor;
    @Basic(optional = false)
    @Column(name = "NOMBRE")
    private String nombre;
    @Basic(optional = false)
    @Column(name = "DNI")
    private String dni;
    @Column(name = "TELEFONO")
    private String telefono;
    @Column(name = "CORREO")
    private String correo;
    @Column(name = "FECHAENTRADA")
    private String fechaentrada;
    @Column(name = "NICK")
    private String nick;
```

```
@OneToMany(mappedBy = "monitorresponsable")
private Set<Actividad> actividadSet;
```

Es necesario inicializar esta estructura y, para mayor claridad, ponerle un nombre más significativo

```
@OneToMany(mappedBy = "monitorresponsable")
private Set<Actividad> actividadesResponsable = new HashSet<Actividad>();
```

■ Clase Monitor (2/2)

```
public Monitor() { }

public Monitor(String codmonitor) { this.codmonitor = codmonitor; }

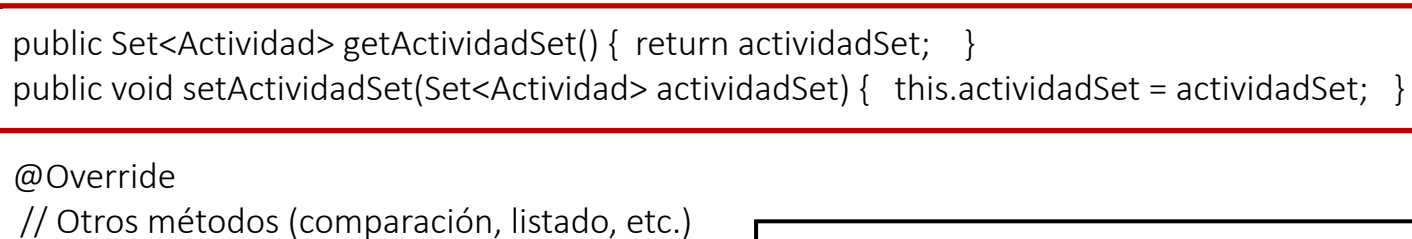
public Monitor(String codmonitor, String nombre, String dni) {
    this.codmonitor = codmonitor;
    this.nombre = nombre;
    this.dni = dni;
}

public String getCodmonitor() { return codmonitor; }
public void setCodmonitor(String codmonitor) { this.codmonitor = codmonitor; }

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

public Set<Actividad> getActividadSet() { return actividadSet; }
public void setActividadSet(Set<Actividad> actividadSet) { this.actividadSet = actividadSet; }

@Override
// Otros métodos (comparación, listado, etc.)
```



```
public Set<Actividad> getActividadesResponsable() { return actividadesResponsable; }
public void setActividadesResponsable(Set<Actividad> actividadesResponsable) { this.actividadesResponsable = actividadesResponsable; }
```

- Se puede comprobar que la herramienta ha generado, de forma automática, 3 constructores para la clase Experto:

```
public Monitor ()  
public Monitor (String codmonitor)  
public Monitor (String codmonitor, String nombre, String dni)
```

- Sin embargo, no ha generado el constructor con todos los atributos (ya que algunos no son obligatorios), por lo que tendremos que añadirlo manualmente:

```
public Monitor (String codmonitor, String nombre, String dni, String telefono, String correo,  
String fechaentrada, String nick)
```

■ Clase Actividad (1/2)

```
@Entity
@Table(name = "ACTIVIDAD")
@NamedQueries({
    @NamedQuery(name = "Actividad.findAll", query = "SELECT a FROM Actividad a"), ... })
```

```
public class Actividad implements Serializable {
    private static final long serialVersionUID = 1L;
```

```
@Id
```

```
@Basic(optional = false)
```

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

```
private String idactividad;
```

```
@Basic(optional = false)
```

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

```
private String nombre;
```

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

```
private String descripcion;
```

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

```
private BigInteger preciobasemes;
```

```
@JoinTable(name = "REALIZA", joinColumns = {
```

```
    @JoinColumn(name = "IDACTIVIDAD", referencedColumnName = "IDACTIVIDAD")}, inverseJoinColumns = {
```

```
    @JoinColumn(name = "NUMEROSOCIO", referencedColumnName = "NUMEROSOCIO")})
```

```
@ManyToMany
```

```
private Set<Socio> socioSet;
```

```
@JoinColumn(name = "MONITORRESPONSABLE", referencedColumnName = "CODMONITOR")
```

```
@ManyToOne
```

```
private Monitor monitorresponsable;
```

Es necesario inicializar esta estructura y, para mayor claridad, ponerle un nombre más significativo

```
@ManyToMany
```

```
private Set<Socio> socios = new HashSet<Socio>();
```

■ Clase Actividad (2/2)

```
public Actividad() { }

public Actividad(String idactividad) { this.idactividad = idactividad; }


public Actividad(String idactividad, String nombre) {
    this.idactividad = idactividad;
    this.nombre = nombre;
}

public String getIdactividad() { return idactividad; }
public void setIdactividad(String idactividad) { this.idactividad = idactividad; }

public Set<Socio> getSocioSet() { return socioSet; }
public void setSocioSet(Set<Socio> socioSet) { this.socioSet = socioSet; }

public Monitor getMonitorresponsable() { return monitorresponsable; }
public void setMonitorresponsable(Monitor monitorresponsable) {
    this.monitorresponsable = monitorresponsable;
}

@Override
// Otros métodos (comparación, listado, etc.)
```



```
public Set<Socio> getSocios() { return socios; }
public void setSocios(Set<Socio> socios) { this.socios = socios; }
```

■ Clase Socio (1/2)

@Entity

@Table(name = "SOCIO")

@NamedQueries({

@NamedQuery(name = "Socio.findAll", query = "SELECT s FROM Socio s"), ... })

public class Socio implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@Basic(optional = false)

@Column(name = "NUMEROSOCIO")

private String numerosocio;

@Basic(optional = false)

@Column(name = "NOMBRE")

private String nombre;

@Basic(optional = false)

@Column(name = "DNI")

private String dni;

@Column(name = "FECHANACIMIENTO")

private String fechanacimiento;

@Column(name = "TELEFONO")

private String telefono;

@Column(name = "CORREO")

private String correo;

@Column(name = "FECHAENTRADA")

private String fechaentrada;

@Basic(optional = false)

@Column(name = "CATEGORIA")

private String categoria;

@ManyToMany(mappedBy = "socioSet")

private Set<Actividad> actividadSet;

Es necesario inicializar esta estructura y, para mayor claridad, ponerle un nombre más significativo

@ManyToMany(mappedBy = "socios")

private Set<Actividad> actividades = new HashSet<Actividad>();

■ Clase Socio (2/2)

```
public Socio() { }

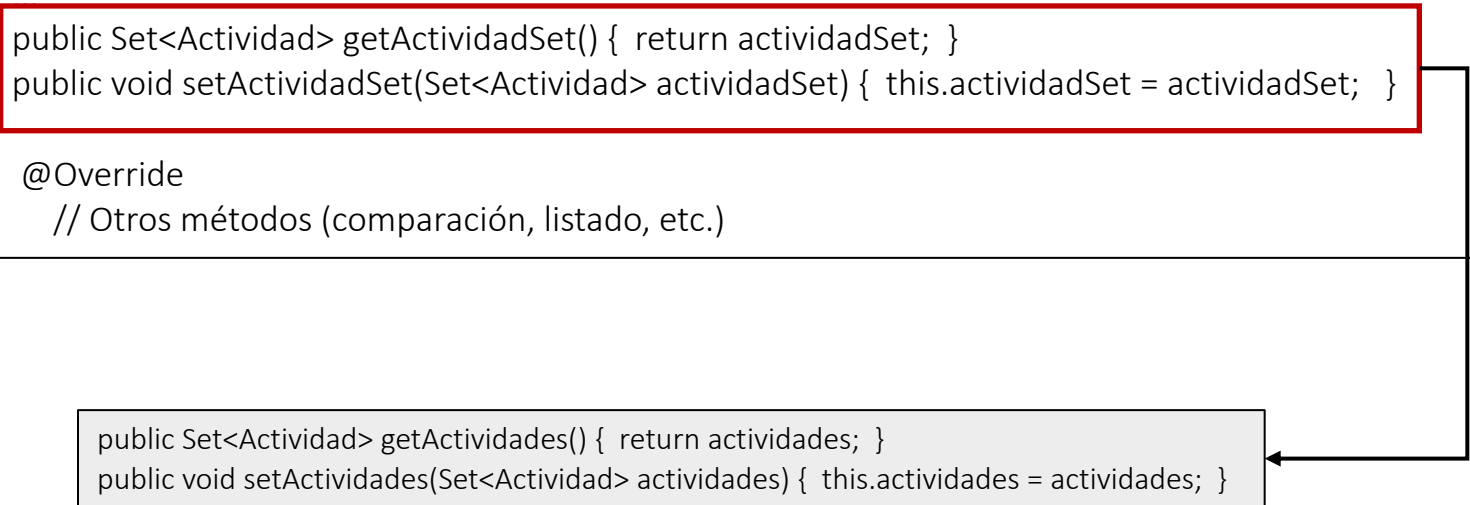
public Socio(String numerosocio) { this.numerosocio = numerosocio; }

public Socio(String numerosocio, String nombre, String dni, Character categoria) {
    this.numerosocio = numerosocio;
    this.nombre = nombre;
    this.dni = dni;
    this.categoria = categoria;
}

public String getNumerosocio() { return numerosocio; }
public void setNumerosocio(String numerosocio) { this.numerosocio = numerosocio; }

public Set<Actividad> getActividadSet() { return actividadSet; }
public void setActividadSet(Set<Actividad> actividadSet) { this.actividadSet = actividadSet; }

@Override
// Otros métodos (comparación, listado, etc.)
```



```
public Set<Actividad> getActividades() { return actividades; }
public void setActividades(Set<Actividad> actividades) { this.actividades = actividades; }
```

Migración del proyecto desde JDBC a Hibernate

Controlador Login

- Debe tener un atributo de clase **Session**, que será el encargado de establecer la conexión con la base de datos a través de Hibernate
- La función conectar(), que se invocará al pulsar un botón de la ventana de login establecerá la conexión y devolverá el objeto de tipo **Session**

```
public Session conectar() {
    String server = (String) (vLogin.jComboBoxServidores.getSelectedItem());
    String ip = vLogin.jTextFieldIP.getText();
    String service_bd = vLogin.jTextFieldService_BD.getText();
    String u = vLogin.jTextFieldUsuario.getText();
    String p = new String (vLogin.jPasswordField.getPassword());

    if (server == "MySQL") server = "mysql";
    else if (server == "Oracle") server = "oracle";

    if ("oracle".equals(server) && "172.17.20.75".equals(ip) && "rabida".equals(service_bd)
        && "usuario".equals(u) && "password".equals(p))
        sesion = HibernateUtil.getSessionFactory().openSession();

    return(sesion);
}
```


Migración del proyecto desde JDBC a Hibernate

Controlador Login

- Si la conexión es correcta, se crea un objeto de tipo Controlador y se le pasa por parámetro la conexión (en este caso le hemos llamado [sesion](#))

```
switch (e.getActionCommand()) {  
    case "Conectar":  
        sesion = conectar();  
        if (sesion != null) {  
            vMensaje.Mensaje("info", "Conexión correcta con Hibernate");  
            vLogin.dispose();  
            Controlador controlador = new Controlador(sesion);  
        }  
        else vMensaje.Mensaje("error", "Error en la conexión con Hibernate. "  
            + "Revise los valores de conexión");  
        break;
```

Operaciones CRUD con Hibernate

Consultas

- Para realizar consultas se puede utilizar el lenguaje **HQL** (*Hibernate Query Language*) o realizarlas directamente en lenguaje SQL ("nativas")
- Una de las ventajas de utilizar consultas nativas es que la migración entre JDBC y JPA/Hibernate es más sencilla
- En la medida de lo posible, mantendremos la sintaxis de la especificación de los métodos en los DAO y solo modificaremos la implementación

Operaciones CRUD con Hibernate

Consultas

```
public ArrayList<Monitor> listaMonitores() throws Exception {  
    Transaction transaction = sesion.beginTransaction();  
  
    Query consulta = sesion.createNativeQuery ("SELECT * FROM Monitor M", Monitor.class);  
    ArrayList<Monitor> monitores = (ArrayList<Monitor>) consulta.list();  
  
    transaction.commit();  
    return monitores;  
}
```

```
public ArrayList<Monitor> listaMonitores() throws Exception {  
    Transaction transaction = sesion.beginTransaction();  
  
    Query consulta = sesion.createQuery ("FROM Monitor M");  
    ArrayList<Monitor> monitores = (ArrayList<Monitor>) consulta.list();  
  
    transaction.commit();  
    return monitores;  
}
```

Operaciones CRUD con Hibernate

Consultas

- Para realizar consultas con parámetros, que impliquen JOIN o que devuelvan campos específicos, utilizaremos consultas nativas

```
public ArrayList<Object[]> listaNombreDNIMonitores() throws Exception {  
    Transaction transaction = sesion.beginTransaction();  
    Query consulta = sesion.createNativeQuery("SELECT nombre, dni FROM Monitor M");  
    ArrayList<Object[]> monitores = (ArrayList<Object[]>) consulta.list();  
  
    transaction.commit();  
    return monitores;  
}
```

Consulta que devuelve dos campos de la tabla MONITOR

```
public ArrayList<String> listaNombreMonitores() throws Exception {  
    Transaction transaction = sesion.beginTransaction();  
    Query consulta = sesion.createNativeQuery("SELECT nombre FROM Monitor M");  
    ArrayList<String> monitores = (ArrayList<String>) consulta.list();  
  
    transaction.commit();  
    return monitores;  
}
```

Consulta que devuelve un único campo de la tabla MONITOR

Operaciones CRUD con Hibernate

Consultas

```
public ArrayList<Monitor> listaMonitorPorLetra(String letra) throws Exception {  
    Transaction transaction = sesion.beginTransaction();  
    letra = letra + "%";  
    Query consulta = sesion.createNativeQuery("SELECT * FROM MONITOR M "  
        + "WHERE nombre LIKE :letra", Monitor.class).setParameter("letra", letra);  
    ArrayList<Monitor> monitores = (ArrayList<Monitor>) consulta.list();  
  
    transaction.commit();  
    return monitores;  
}
```

Consulta parametrizada

Operaciones CRUD con Hibernate

Inserción

- Para insertar una nueva tupla en una tabla es necesario crear, previamente, el objeto correspondiente. El método que inserta un objeto es **save()**

```
public void insertaMonitor(Monitor monitor) throws Exception {  
    Transaction transaction = sesion.beginTransaction();  
    sesion.save(monitor);  
    transaction.commit();  
}
```

Eliminación

- Para eliminar una tupla se utiliza el método **delete()**. El método **get()** recupera un objeto mediante su clave principal

```
public void eliminaMonitor(String codMonitor) throws Exception {  
    Transaction transaction = sesion.beginTransaction();  
    Monitor monitor = sesion.get(Monitor.class, codMonitor);  
    sesion.delete(monitor);  
    transaction.commit();  
}
```

Operaciones CRUD con Hibernate

Actualización

- La actualización de una tupla se realiza de la misma forma que la inserción de una nueva. Hibernate decide lo que debe hacer (insertar o actualizar) en función del valor de la clave principal del objeto que se envía

```
public void actualizaMonitor(Monitor monitor) throws Exception {  
    Transaction transaction = sesion.beginTransaction();  
    sesion.save(monitor);  
    transaction.commit();  
}
```

La única diferencia con la inserción es que ahora, el objeto "monitor" que se pasa por parámetro se ha obtenido, previamente, con el método **get()** de Hibernate, y se le han realizado algunas modificaciones mediante los métodos **set()** de la clase Monitor