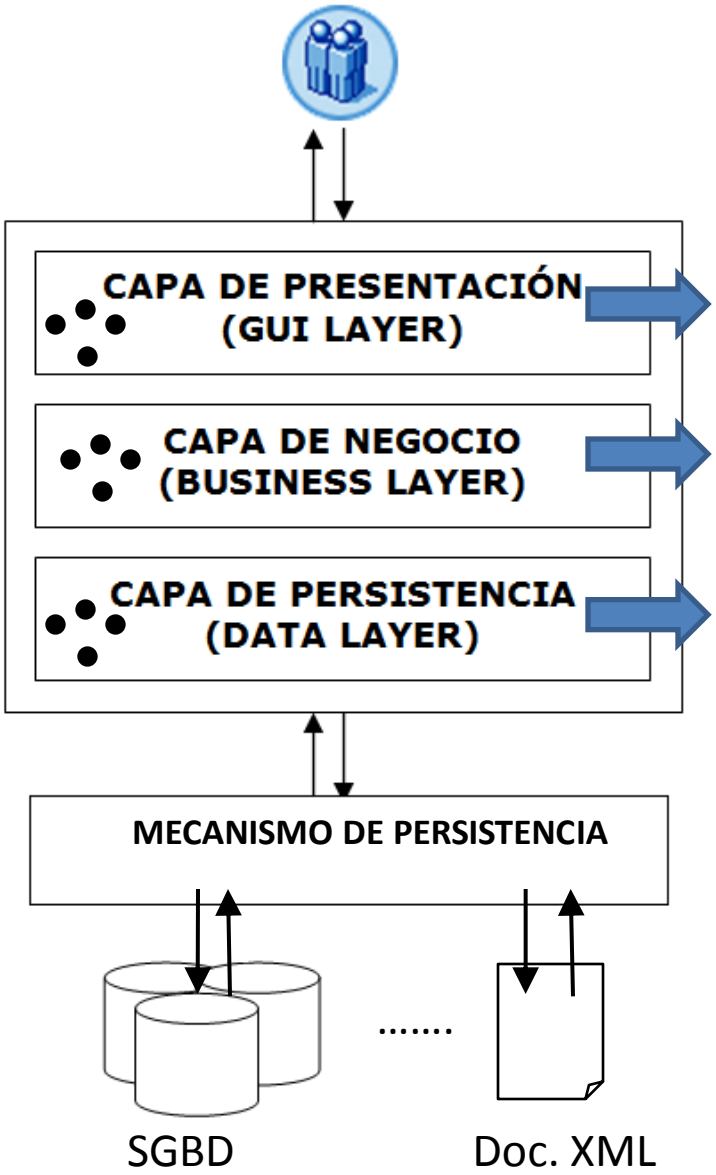
Diseño de la capa de persistencia o acceso a datos

Jordi Ariño Santos
(jordi.arino@pue.es)

Diseño de la capa de persistencia

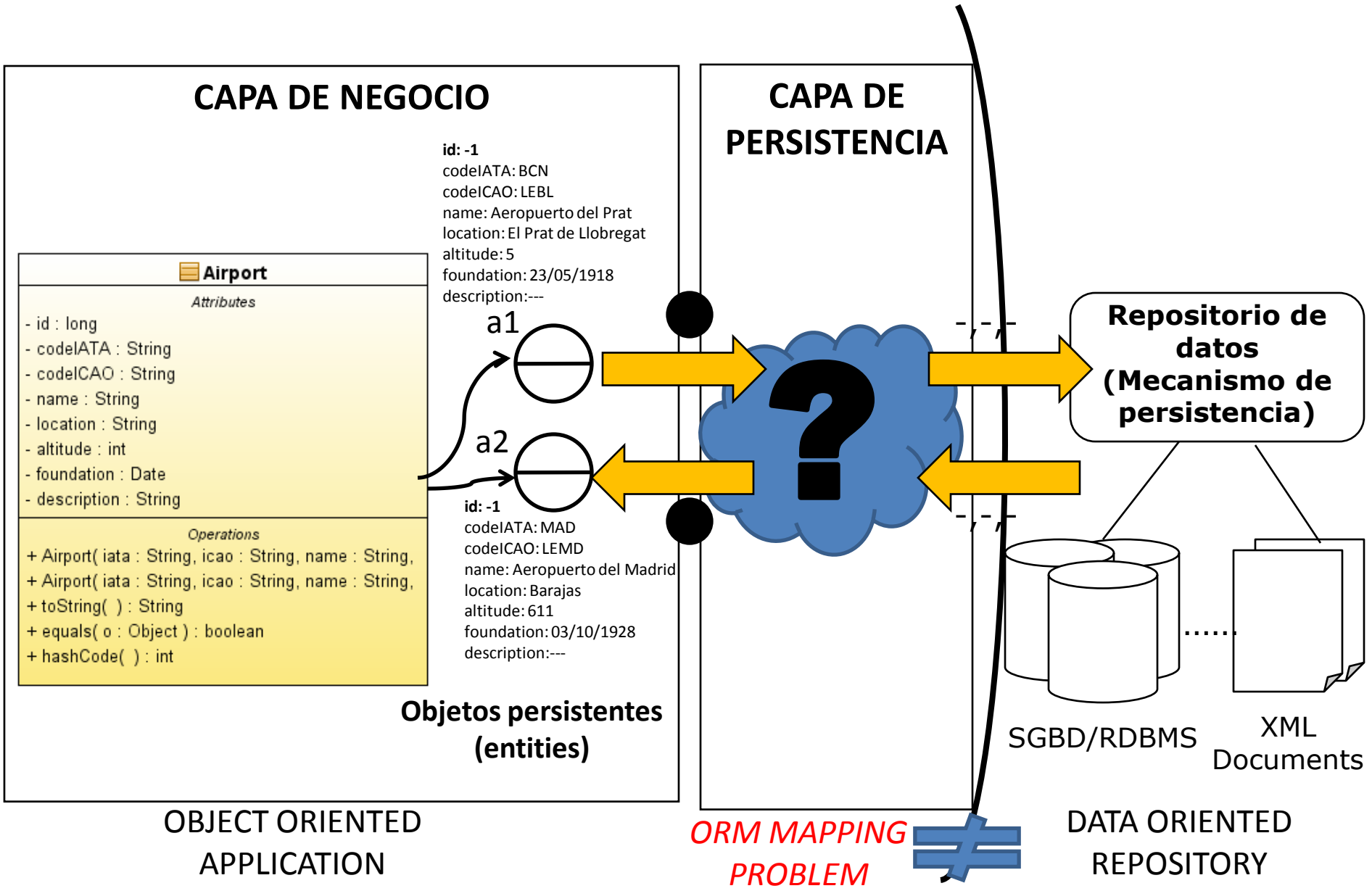


Clases que representan las **vistas** de una aplicación. Implementan la interfaz gráfica a mostrar al usuario, así como los mecanismos para interactuar con él

Clases que definen a los objetos que representan el **dominio** de una aplicación: es decir, los objetos que representan los conceptos/elementos que queremos representar en una aplicación.

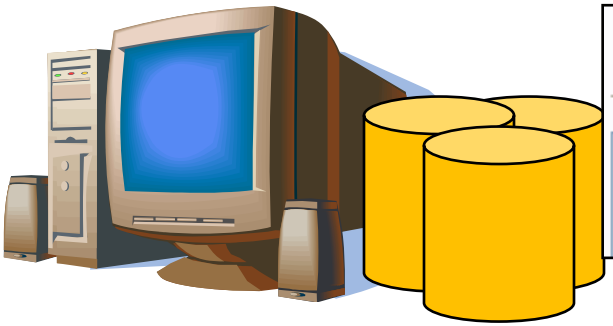
Clases que implementan la **persistencia** de los objetos de nuestra capa de negocio: cómo se van a almacenar esos objetos y como se van a recuperar posteriormente.

Diseño de la capa de persistencia



Diseño de la capa de persistencia

- **Diseño del mecanismo de persistencia**



Schema/Catalog: *MyVueling*

myvueling
All tables of the myvueling schema

Table Name	Engine
airports	InnoDB

MySQL Server 6.0 Alpha
>**Host**: *localhost*
>**Port**: *3306*
>**User**: *root*
>**Password**: *admin*

Column Name	Datatype	NOT NULL	AUTO INC
id	BIGINT(20)	✓	✓
codeIATA	CHAR(3)	✓	
codeICAO	CHAR(4)	✓	
name	VARCHAR(200)	✓	
location	VARCHAR(200)	✓	
altitude	INTEGER	✓	
foundation	DATETIME	✓	
description	TEXT	✓	

Diseño de la capa de persistencia

- **Problema:**
  Conseguir que los objetos de la **capa de negocio** se hagan persistentes (*"hibernen"*) en algún medio de persistencia (SGBD, documentos XML, ..) y poderlos recuperar más delante estos objetos de su hibernación.

- **Solución**: Usar un *patrón de diseño*
  A la hora de implementar la capa de persistencia de una aplicación, el **patrón de diseño DAO (Data Access Objects)** se presenta como una solución muy aceptada.
  - DAO forma parte de la especificación empresarial de Java (Java EE Platform).
  - Se puede utilizar en cualquier lenguaje de programación.
  - Publicado por Java BluePrints.
    http://java.sun.com/blueprints/patterns/DAO.html

Diseño de la capa de persistencia

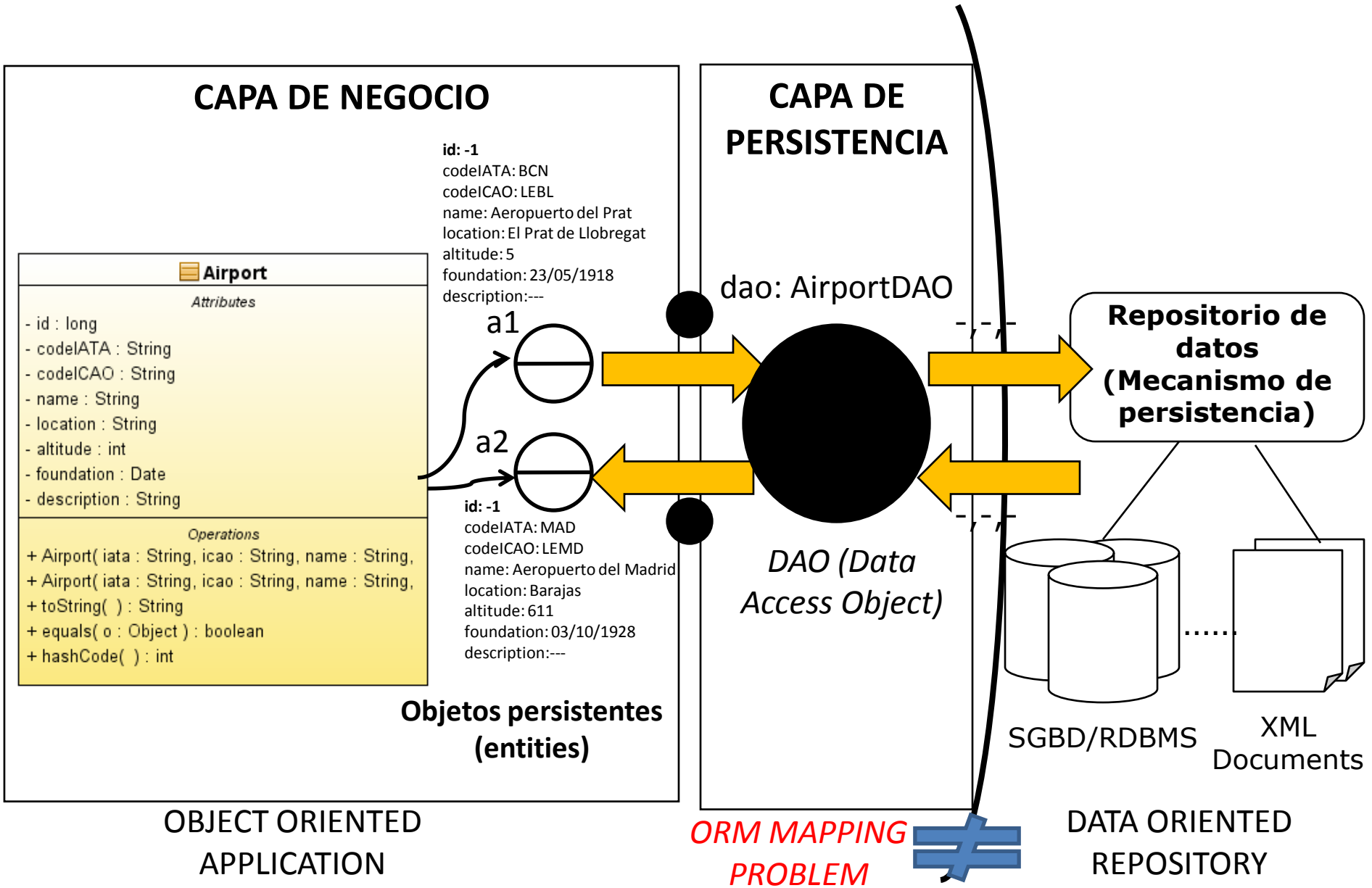
EL PATRON DE DISEÑO DAO

*Para cada clase de la capa de negocio de la que queremos que sus objetos sean persistentes (**objetos de transferencia**), tendremos un objeto de una clase propia de acceso a datos (**clase DAO**) que gestionará e implementará como se realizará esta persistencia*

- **Objetivos**:
  - Abstraer y encapsular todos los accesos al origen de datos en una capa aparte (capa de persistencia).

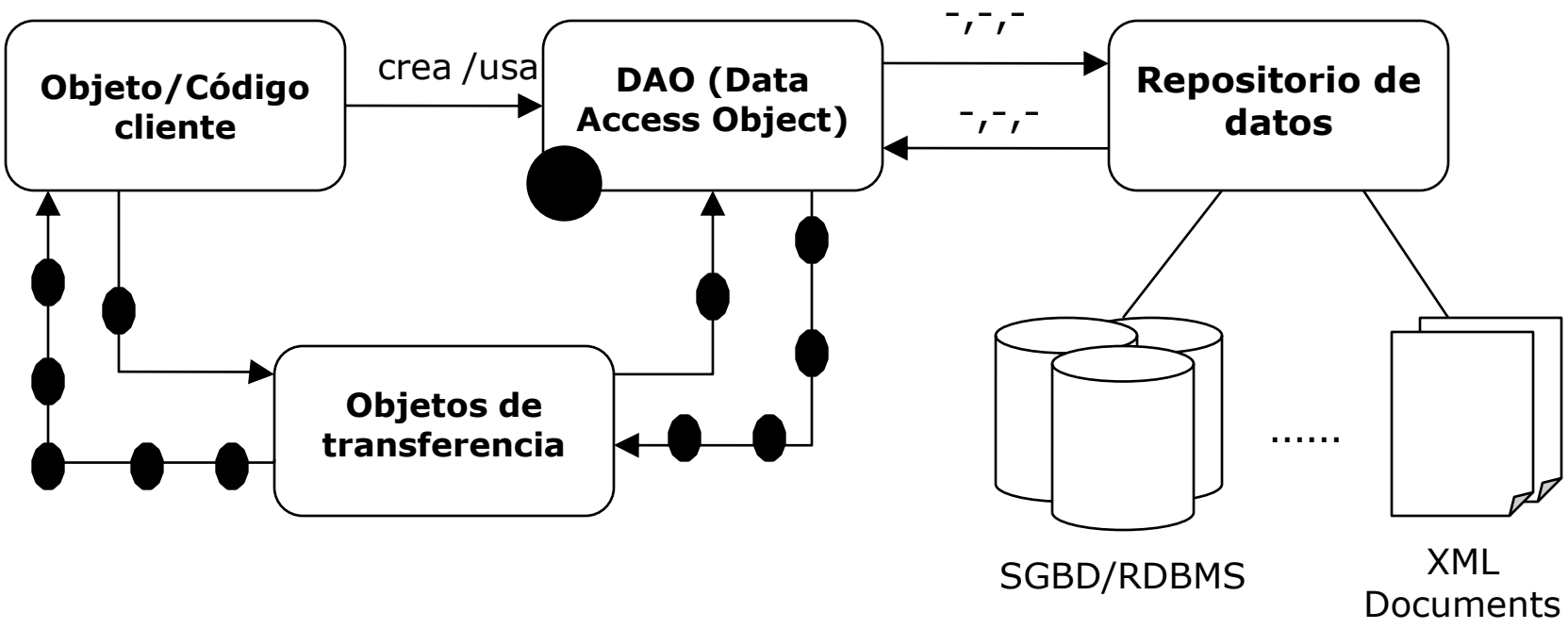
- **Ventajas**:
  - Se baja el nivel de acoplamiento entre clases reduciendo la complejidad de realizar cambios
  - Se aíslan las conexiones al origen de datos en una capa fácilmente identificable y mantenible
  - Acceso de manera transparente al repositorio de datos
  - DAO proporciona una interfaz única de acceso a los datos del repositorio, independientemente de cuál sea este.

Diseño de la capa de persistencia



Diseño de la capa de persistencia

- **El modelo de acceso a datos con el patrón DAO**



Diseño de la capa de persistencia

- **Repositorio de datos**
  - Representa una implementación de un repositorio de datos o mecanismo de persistencia.
- **Objeto o código cliente**
  - Representa el componente que desea acceder al repositorio de datos para hacer pesistentes objetos de nuestra lógica de negocio, recuperarlos, hacer búsquedas,…
- **DAO (Data Access Object)**
  - Principal elemento del patrón.
  - Abstrae los detalles de implementación del repositorio de datos al objeto de negocio (acceso transparente a los datos).
  - Representa el punto de enlace entre nuestra aplicación y el repositorio de datos.
  - Es el único componente de nuestra aplicación que podrá comunicarse con el repositorio de datos.
  - Se encargará de hacer persistentes a los objetos de la lógica de negocio, y de recuperar objetos que estuvieran almacenados en el repositorio.
- **Objetos de transferencia**
  - Objetos de la lógica de negocio que DAO va a hacer persistentes, o que va a recuperar del repositorio de datos.

Diseño de la capa de persistencia

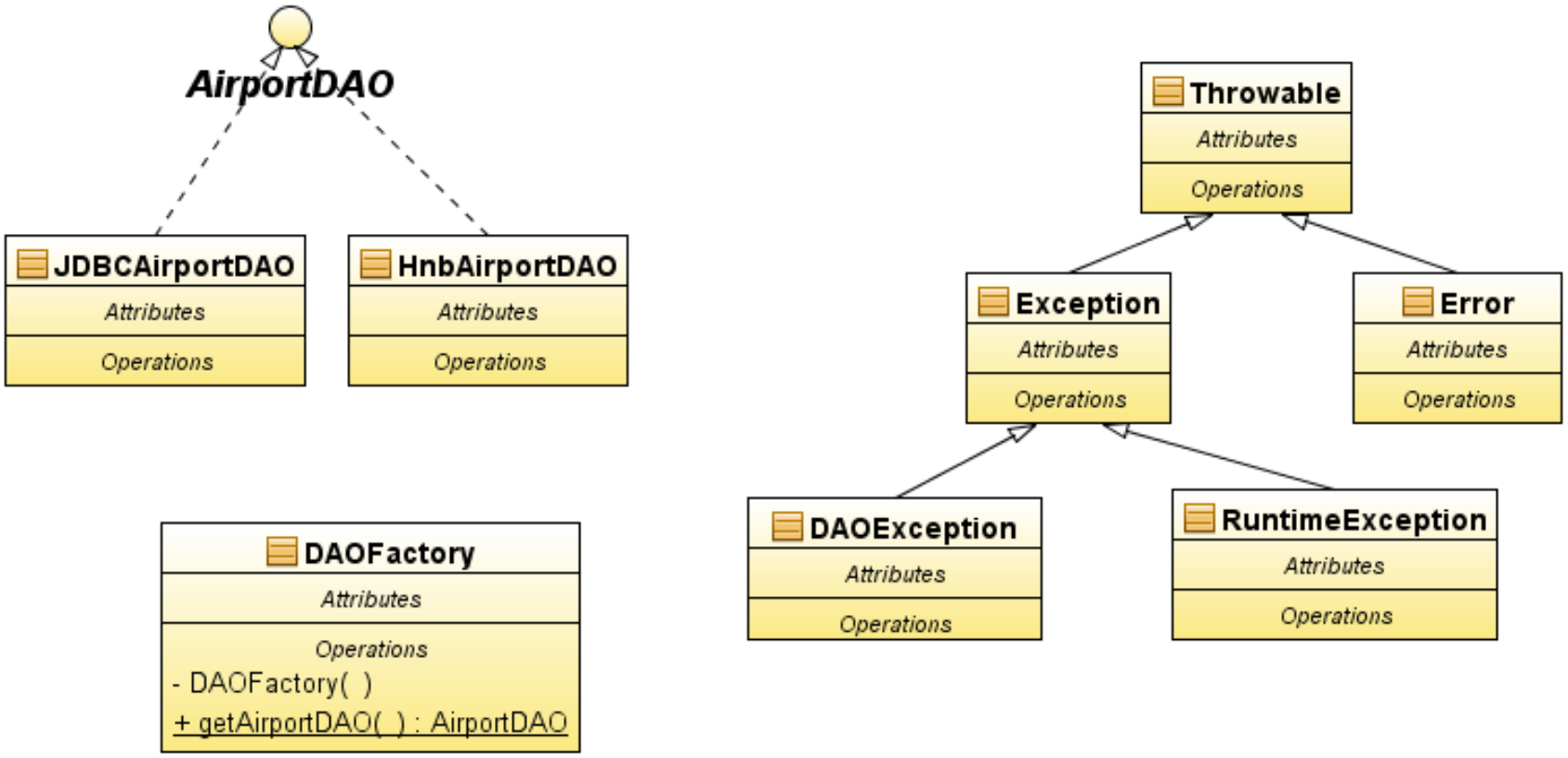
- **Implementación del patrón de diseño DAO**

"DAO proporciona una interfaz única de acceso al repositorio de persistencia e independiente de cual sea éste, permitiéndolo cambiar sin modificar el resto de la aplicación"

1. Ofrecer una interface que defina las operaciones de persistencia o API de persistencia por cada tipo de DAO que sea independiente del repositorio
2. Crear un tipo de excepción propia e independiente del mecanismo de persistencia que encapsule todas las situaciones erróneas que se puedan producir en los DAOs
3. Ofrecer una implementación específica de ese API de persistencia según el repositorio a utilizar
4. Implementar una factoría o fábrica de *DAOs* (patrón de diseño factory) que se encargue de centralizar la creación de los objetos DAO

Diseño de la capa de persistencia

- **Diagrama de clases UML**



Acceso a datos en Java (API JDBC)

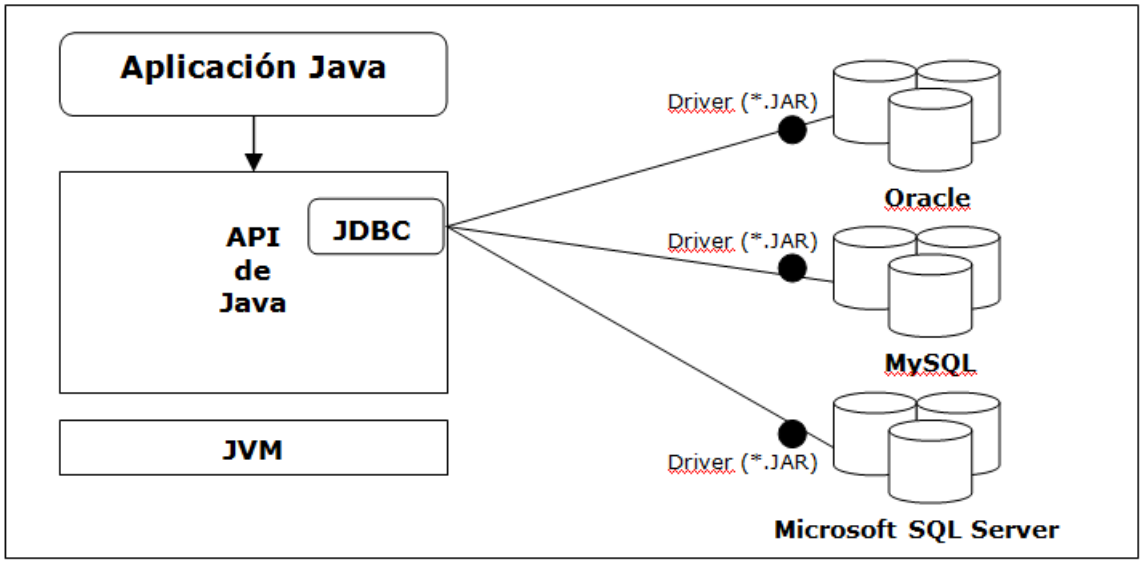
- **JDBC (Java DataBase Connectivity):**
  - Es un API que contiene las clases e interficies Java necesarias para establecer la comunicación entre una aplicación y un sistema gestor de base de datos
  - Es independiente del sistema gestor de base de datos (SGBD/RDBMS)
  - Proporciona una interfaz de programación única

Acceso a datos en Java (API JDBC)

- **JDBC (Java DataBase Connectivity):**
  - Esta independencia se debe al hecho de que está constituido básicamente por clases que no se encuentran implementadas (interficies)
    - Ofrecen una especificación que otra clase ha de implementar
  - Cada proveedor o fabricante de SGBD dispondrá de una implementación específica de las interficies que forman el API de JDBC
  - Estas clases específicas que implementan el API JDBC se agrupan en lo que se conoce como drivers.

Acceso a datos en Java (API JDBC)

- El API JDBC incluye:
  - Gestión de conexiones a bases de datos
  - Ejecución de sentencias SQL y procedimientos almacenados (stored procedures)
  - Procesamiento de resultados asociados a sentencias SQL
  - Soporte de transacciones



Acceso a datos en Java (API JDBC)

- **Componentes del API JDBC:**
  - Packages: *java.sql* y *javax.sql*
  - **DriverManager**
    - Representa el gestor de drivers (componente donde se registran los drivers que queremos utilizar en nuestra aplicación).
    - Es el componente encargado de abrir conexiones a las bases de datos

Acceso a datos en Java (API JDBC)

- **Componentes del API JDBC:**
  - **Connection**
    - Representa una conexión contra una base de datos específica
  - **Statement**
    - Representa una sentencia SQL sin información dinámica que queremos ejecutar contra una BD
  - **PreparedStatement**
    - Representa una sentencia SQL con información dinámica y que ha sido compilada en el SGBD antes de su ejecución

Acceso a datos en Java (API JDBC)

- **Componentes del API JDBC:**
  - **CallableStatement**
    - Representa una sentencia de invocación a un procedimiento almacenado residente en una BD
  - **ResultSet**
    - Representa a un iterador/cursor sobre el conjunto de resultados asociado a la ejecución de una sentencia SQL de consulta

Acceso a datos en Java (API JDBC)

- **Componentes del API JDBC:**
  - **ResultSetMetaData**
    - Permite obtener información acerca de un conjunto de resultados asociado a una consulta SQL
  - **DatabaseMetaData**
    - Permite obtener información acerca de una base de datos

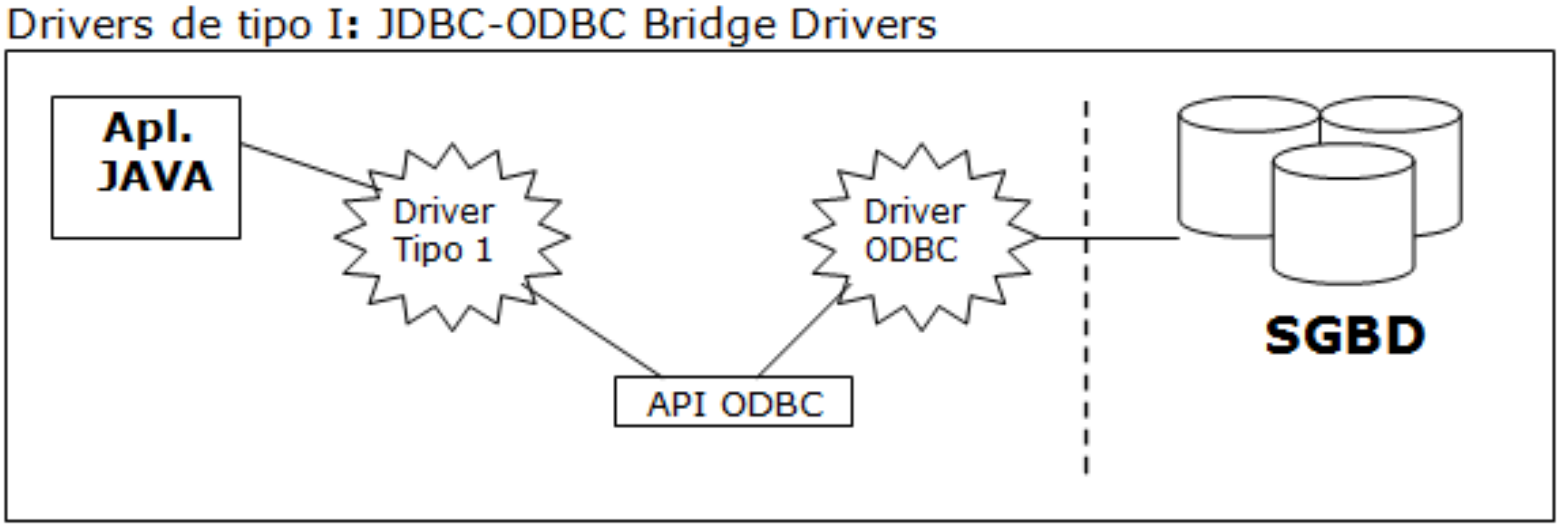
Acceso a datos en Java (API JDBC)

- **Drivers en JDBC:**
  - Un driver JDBC es una colección de clases que implementan de manera específica para un proveedor o fabricante de SGBD las interfícies de JDBC
  - Existen 4 categorías o tipos de drivers en Java:
    - Drivers de tipo I: JDBC-ODBC Bridge Drivers
    - Drivers de tipo II: Native-API Partly Java Drivers
    - Drivers de tipo III: Net-protocol All-Java Drivers
    - Drivers de tipo IV: Native-protocol All-Java Drivers
  - Puedes consultar una lista de todos los drivers JDBC ofrecidos por los distintos fabricantes de SGBD,

    http://developers.sun.com/product/jdbc/drivers

Acceso a datos en Java (API JDBC)

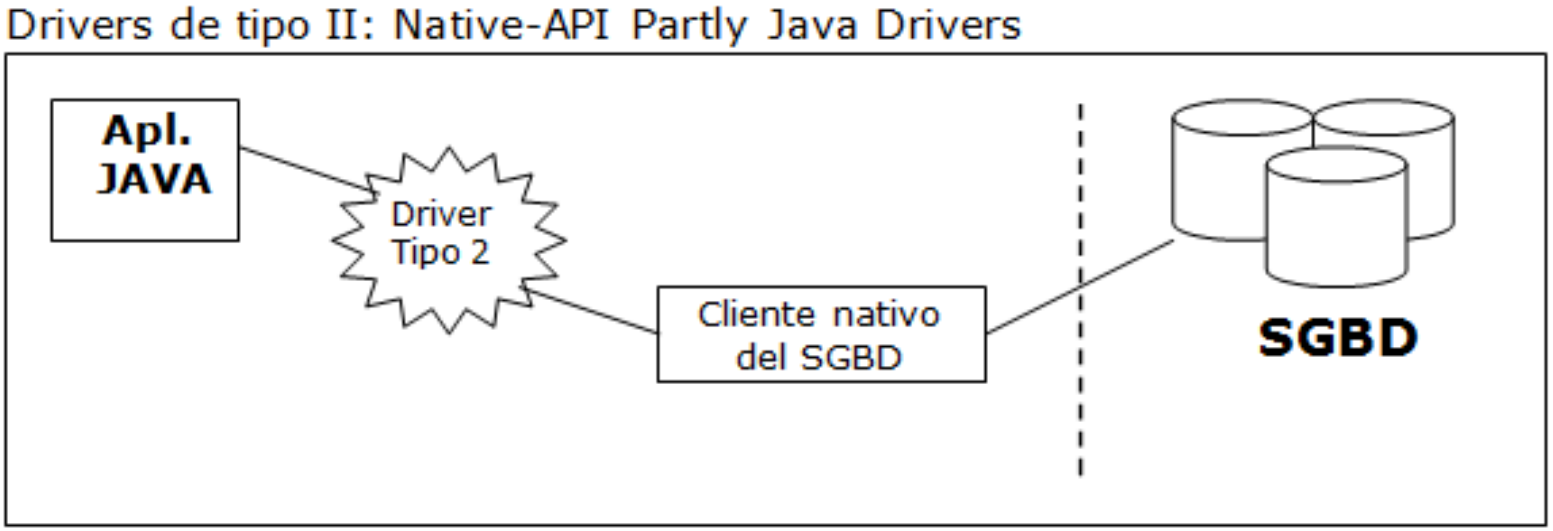
- **Drivers de tipo I**



- Incluido en el API de Java.
- Traduce invocaciones JDBC a invocaciones ODBC a través de las librerías ODBC del SO.
- Requiere de la instalación y configuración del cliente ODBC.
- Solución no portable (depende de librerías nativas).

Acceso a datos en Java (API JDBC)

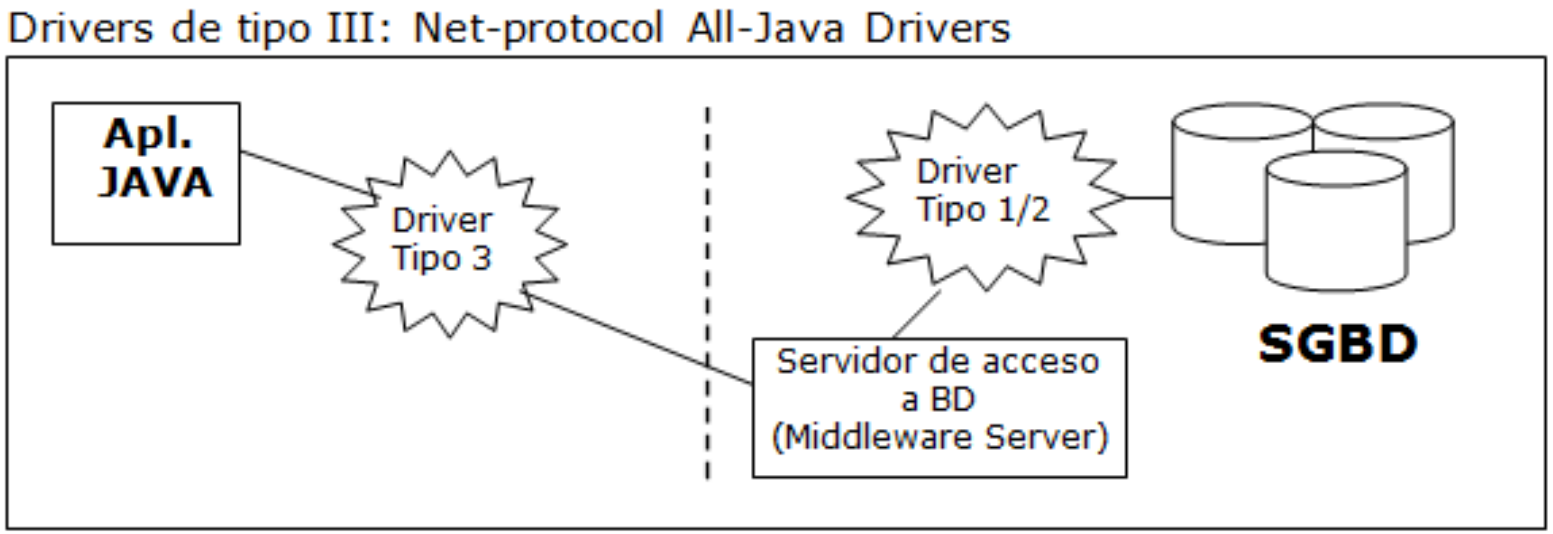
- **Drivers de tipo II**



- No incluido en el API de Java.
- Traduce invocaciones JDBC a invocaciones específicas de la API cliente del proveedor del SGBD.
- Requiere de la instalación y configuración del cliente del SGBD.
- Solución no portable (depende de librerías nativas).

Acceso a datos en Java (API JDBC)

- **Drivers de tipo III**

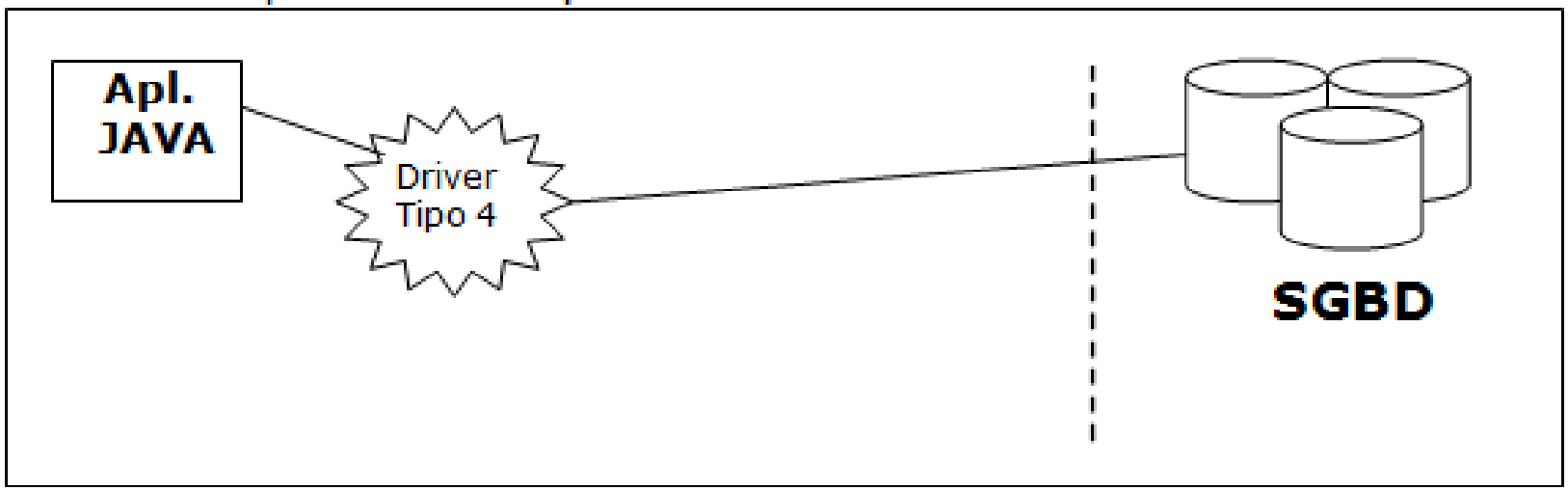


- No incluido en el API de Java.
- Conecta de manera remota mediante un protocolo independiente del SGBD (TPC/IP) con un listener o servidor de escucha del SGBD.
- Este servidor de escucha se comunica con el SGBD utlizando otro driver JDBC (tipo 1 ó 2)
- No requiere ninguna instalación previa en el cliente, pero sí en el servidor.

Acceso a datos en Java (API JDBC)

- **Drivers de tipo IV**

Drivers de tipo IV: Native-protocol All-Java Drivers



- No incluido en el API de Java.
- Conecta de manera remota mediante un protocolo de red independiente del SGBD (TCP/IP)
- No requiere ninguna instalación previa.
- Método más eficiente de acceso a BD.
- Solución altamente portable.

Acceso a datos en Java (API JDBC)

- **JDBC URL:**
  - Representa la cadena de conexión o URL (Uniform Resource Locator) a una base de datos específica
  - Es un identificador único que permite identificar de manera única a una base de datos
  - Es un identificador específico del tipo de driver y del fabricante del SGBD.

Acceso a datos en Java (API JDBC)

- **JDBC URL:**
  - Formato:

```
jdbc:<<subprotocol>>:<<database-locator>>
```

    - *<<subprotocol>>*
      Especifica el tipo driver a utilizar en la conexión
    - *<<database-locator>>*
      Indicador específico del driver para específicar de forma única la base de datos con la que queremos interactuar (suele incluir nombre del host, puerto, nombre de la base de datos...)

Acceso a datos en Java (API JDBC)

- **Soporte de transacciones en JDBC:**
  - JDBC soporta de manera nativa la gestión de transacciones
  - Una transacción SQL es un conjunto de sentencias SQL que deben ser ejecutadas como una unidad atómica
    - Si todas las sentencias SQL se ejecutan correctamente se hace *commit* (aceptar posibles cambios)
    - Si alguna sentencia SQL provoca error se hace *rollback* (deshacer posibles cambios)

Acceso a datos en Java (API JDBC)

- **Soporte de transacciones en JDBC:**
  - Por defecto, una conexión funciona en *modo autocommit* (commit implicito)
    - Cada vez que se ejecuta una sentencia SQL se abre y se cierra automáticamente una transacción, que sólo afecta a dicha sentencia y de la que se hace el commit de manera automática
  - Si queremos podemos activar el *modo transaccional*, desactivando el modo autocommit y gestionando de manera explícita las transacciones (*commit()* o *rollback()*)

Acceso a datos en Java (API JDBC)

- **Programación del API JDBC:**
  - Lo primero de todo:
    - Obtener el driver del proveedor o fabricante del SGBD al que nos queremos conectar.
  - Antes de realizar cualquier tipo de comunicación con el SGBD (y siempre una única vez):
    - Cargar en memoria las clases que componen el driver, y registrarlas en el gestor de drivers.

Acceso a datos en Java (API JDBC)

- **Programación del API JDBC:**
  - Por cada comunicación a realizar con el SGBD desde la clase:
    - Pedirle al gestor de drivers que nos abra una conexión a una BD indicada usando su identificador de conexión (JDBC URL).
    - Activar el modo transaccional (deshabilitar el modo autocommit).
    - Obtener y ejecutar las sentencias SQL deseadas, y procesar los posibles resultados.
    - Si todo ha ido bien, haremos el commit (aceptar posibles cambios producidos) de la transacción.
    - Si algo falla, haremos el rollback (deshacer posibles cambios producidos) de la transacción.
    - Finalmente, siempre cerraremos todos los recursos que hayan intervenido en la comunicación con el SGBD.

Acceso a datos en Java (API JDBC)

- **Datos de conexión al SGBD**

mysql.properties

```
# JDBC URL Connection properties
mysql.url = jdbc:mysql://localhost:3306/MyVueling
mysql.user = root
mysql.password = admin
```

Key	Value
mysql.url	jdbc:mysql://localhost:3306/MyVueling
mysql.user	root
mysql.password	admin

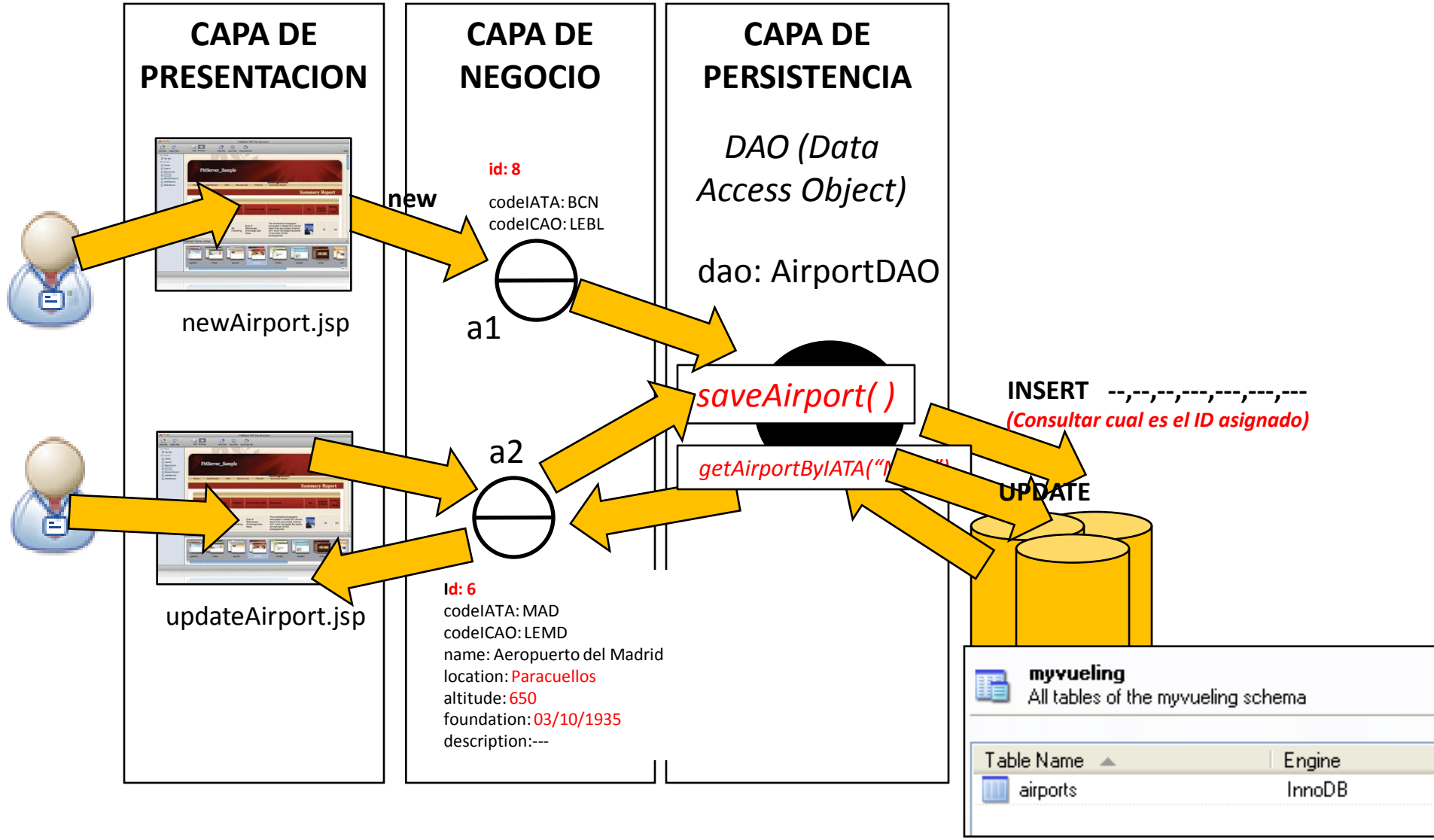
Acceso a datos en Java (API JDBC)

- **Registro del driver en los DAOs**

```
public class JDBCAirportDAO implements AirportDAO {

    static {
        //Registrar el driver en memoria
        try{
            Class.forName("com.mysql.jdbc.Driver");
        }
        catch (ClassNotFoundException ex) {
            //ERROR CRITICO (Finalizar ejecución de la JVM)
            System.exit(-1);
        }
    }
}
```

Diseño de la capa de persistencia



Diseño de la capa de persistencia

```
public void saveAirport(Airport a) throws DAOException{
    if (a == null) {
        throw new NullPointerException("No se admite la referencia nula como parámetro válido");
    }

    if (a.getId() == -1) {
        insertAirport(a);
    }
    else {
        updateAirport(a);
    }
}
```

Diseño de la capa de persistencia

```
private void insertAirport(Airport a) throws DAOException {
    //Recursos a utilizar en la comunicación con el sgbd
    Connection conn = null;
    PreparedStatement sentSQL = null;
    ResultSet reader = null;

    try
    {
        //Abrimos la conexión al SGBD mediante el gestor de drivers (DriverManager)
        conn = getConnection();

        //Activamos el modo transaccional
        conn.setAutoCommit(false);

        //Precompilamos y ejecutamos la sentencia SQL contra el SGBD
        sentSQL = conn.prepareStatement("INSERT INTO Airports(codeIATA, codeICAO, name, location, altitude,
        sentSQL.setString(1, a.getCodeIATA());
        sentSQL.setString(2, a.getCodeICAO());
        sentSQL.setString(3, a.getName());
        sentSQL.setString(4, a.getLocation());
        sentSQL.setInt(5, a.getAltitude());
        sentSQL.setDate(6, new java.sql.Date(a.getFoundation().getTime()));
        sentSQL.setString(7, a.getDescription());
        sentSQL.executeUpdate();
```

Diseño de la capa de persistencia

```
    sentSQL = conn.prepareStatement("SELECT @@IDENTITY");
    reader = sentSQL.executeQuery();
    if (reader.next()) {
        a.setId(reader.getLong(1));
    }
    //COMMIT (Aceptamos posibles cambios)
    conn.commit();
}
catch (SQLException ex)
{
    if (conn != null) {
        try {
            conn.rollback();
        }
        catch (SQLException ex1) {
            throw new DAOException(ex1.getMessage(), ex1);
        }
    }

    throw new DAOException(ex.getMessage(), ex);
}
catch (IOException ex)
{
    throw new DAOException(ex.getMessage(), ex);
}
```

Diseño de la capa de persistencia

```
    finally
    {
        try {
            if (reader != null) {
                reader.close();
            }
            if (sentSQL != null) {
                sentSQL.close();
            }
            if (conn != null) {
                conn.close();
            }
        }
        catch (SQLException ex) {
            throw new DAOException(ex.getMessage(), ex);
        }
    }
}
```