Diseño de aplicaciones

Breve repaso al diseño multicapa

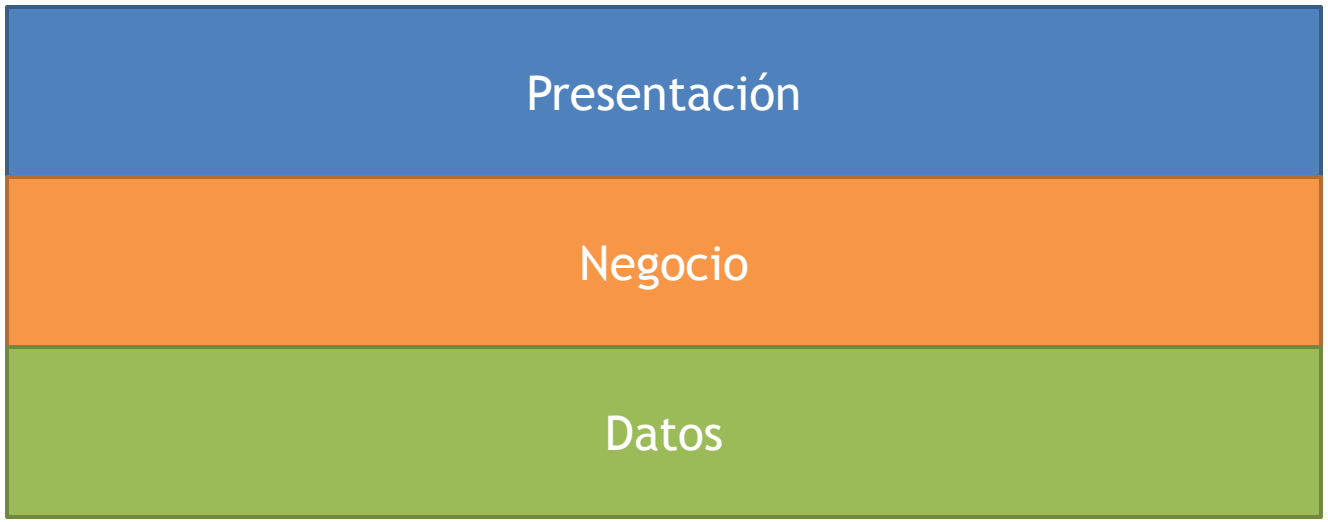
Diseño de una aplicación con Acceso a bbdd en Java

- Ya conocemos una tecnología para persistencia de datos: JDBC que aporta herramientas de acceso a una base de datos relacional.
- Pero hay más tipos de persistencias de datos: bbdd orientadas a objetos, bbdd documentales, bbdb distribuidas, bbdd no relacionales, incluso almacenaje en ficheros, así como tecnologías de acceso a estos datos como JDO, JPA, Mybasis.
- En una aplicación es posible cambiar de base de datos, o incluso de tecnología de acceso a datos según sean sus necesidades futuras o de forma dinámica según preferencias de usuario, pero la mayoría de la lógica de negocio se mantendrá.
- Se definen un patrón de diseño para abstraer la lógica de la aplicación de la fuente de datos.

Diseño de una aplicación con Acceso a bbdd en Java

- También existen varias formas de mostrar los datos e interactuar con el usuario (Presentación con Java Swing(ventanas) , los flujos de entrada y salida estándar (consola), aplicaciones web, aplicaciones móviles, interfaces adaptadas…)
- Una aplicación es posible que tenga la necesidad de adaptar su interfaz a varios tipos de dispositivos según sean sus necesidades futuras o incluso de forma dinámica según preferencias de usuario, pero la mayoría de la lógica de negocio se mantendrá.
- Se define un patrón de diseño para abstraer la lógica de la aplicación de la capa de presentación.

Diseño multicapa



Diseño multicapa

- Beneficios
  - Fomenta la reutilización de código
  - Facilita el mantenimiento
  - Posibilita distribuir el software entre distintas máquinas
  - Escalabilidad
  - Claridad

Diseño multicapa

- **POJOS** (Plain old Java Obecjt) o **Beans** o **DTO** (Data transporting objects)
- Objetos planos que solamente tienen atributos y métodos de acceso a estos atributos. Usados en las tres capas de una aplicación:
  - Presentación, muestra datos de los pojos.
  - Negocio: Trabaja con los pojos. Son el elemento clave en la capa de negocio
  - Datos: Almacena y recupera pojos.
- Ejemplo de pojo:

```
public class Pesona {
private String nombre
Private int dni
public int getDNI(){ return dni;}
public String getNombre(){return nombre;}
public setNombre(String n){nombre=n}
public setDNI(int dni){this.dni=dni}
}
```

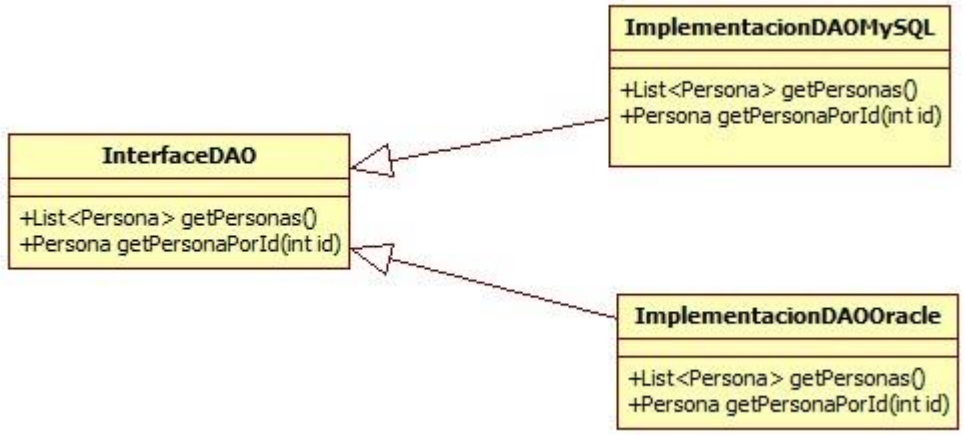
Diseño multicapa - Datos

- **Clases DAO** (Objetos de acceso a datos)
- Ofrecen métodos para persistir, modificar y obtener datos sobre un mismo pojo.(si trabajamos con JDBC son los únicos que tienen cogido SQL)
- Ejemplo de clase DAO, personaDAO

```
public class PersonaDAO {
public List<Persona> getPersonas();
public Persona getPersonaPorNombre (String nombre);
 ...
public void salvaPersona (Persona persona);
public void modificaPersona (Persona persona);
 ...
public void borraPersonaPorNombre (String nombre);
... }
```

Diseño multicapa - Datos

- El uso de interfaces DAO facilita la implementación de la persistencia de datos en distintas fuentes.
  - Por ejemplo en distintas bases de datos (con distintos drivers y algunos cambios en el tipo de datos y sentencias SLQ) como muestra la imagen
  - Con base de datos y ficheros
  - Con distintos ORM.



Diseño multicapa - Datos

- **Patron Singleton** para clases DAO con JDBC.
- Objetivo : evitar crear una nueva conexión a bbdd cada vez que se cree un objeto DAO. Los DAO pueden ser usados desde varias clases de negocio, incluso desde otros DAO. Con una misma instancia DAO se pueden realizar todas las acciones requeridas.

```
package persistencia;
import java.sql.*;
public
class Agente {
protected static Agente mInstancia=null;
protected Connection mBD;
protected Agente() throws Exception {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        String url="jdbc:odbc.JdbcOdbcDriver";
        mBD=DriverManager.getConnection(url);
}
public static Agente getAgente()throws Exception {
        if (mInstancia==null) {
        mInstancia=new Agente();
}
return mInstancia;
}
```

JDBC “Avanzado”

Víctor Custodio,

Crear Comando SQL- Statement

- Para Crear un statement:

```
Statement stmt = conn.createStatement();
```

- Ahora para que podamos utilizar las sentencias UPDATE, INSERT, DELETE,SELECT tenemos que utilizar los métodos:
  - **execute: Para ejecución de creación de tablas, eliminacion etc..**
  - **executeUpdate**: Retorna un número entero indicando la cantidad de registros afectados (UPDATE, INSERT,DELETE).
  - **executeQuery:** Devuelve un conjunto de resultados que se almacenan en un objeto ResultSet

- Ejemplos

```
ResultSet rs = s.executeQuery ("select * from persona");
cadSQL = "DELETE FROM persona WHERE nombreAutor='Pedro";
r = stmt.executeUpdate(cadSQL);
```

Crear Comando SQL- PreparedStatement

- Comando con introducción de valores de forma variable
  - Los parámetros de entrada se especifican por posición utilizando métodos setXXX(posición, valor);
  - Los valores se conservan entre ejecuciones.
  - Borrar parámetros: clearParameters()
- Para Crear un statement:

```
String sql = “SELECT * from persona where nombre=?”
PreparedStatement ps = conexion.prepareStatement(sql);
ps.setString(1,“Pedro”) //asigna  Pedro a la primera ?
rs = ps.executeQuery();
```

Crear Comando SQL- Callable

- CallableStatement es el modo estándar de llamar procedimientos almacenados con la sintaxis de escape SQL de procedimiento almacenado de API JDBC

Conexiones - Metadatos

- A partir de una Connection es posible obtener un conjunto de metadatos sobre la base de datos a la que se ha conectado mediante:

```
DatabaseMetaData meta = connection.getMetadata();
```

- Estos metadatos incluyen, entre otros muchos:
  - Nombre y versión de la base de datos
  - Capacidades y limitaciones
  - Listado de tablas

Conexiones - Metadatos

- Ejemplo de uso de metadatos para obtener un listado de tablas disponibles en un schema:

```
 dbmd=conexion.getMetaData();
  System.out.println("tipo de base de datos: " + dbmd.getDatabaseProductName());
  System.out.println("versión: " + dbmd.getDatabaseProductVersion());
  System.out.println("nombre del driver: " + dbmd.getDriverName());
  System.out.println("versión del driver: " + dbmd.getDriverVersion());
  System.out.println("nombre del usuario: " + dbmd.getUserName());
  System.out.println("url de conexión: " + dbmd.getURL());
  rs=dbmd.getTables(null,null,"%",null);
  System.out.println("estructura de la base de datos");
System.out.println("base de datos\tesquema\tnombre de tabla\ttipo de tabla");
while(rs.next())
{
      for (int i = 1; i <=4 ; i++)
      {
            System.out.print(rs.getString(i)+"\t");
      }
      System.out.println();
}
```

Ver: http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html#getTables(java.lang.String, java.lang.String, java.lang.String, java.lang.String[])

Transacciones

- Las transacciones son un mecanismo que permite al desarrollador agrupar sentencias en conjuntos, de forma que todas las ejecuciones de estas sentencias deben funcionar correctamente, ya que en caso contrario ninguna de ellas será efectiva.
- Los tres métodos que ofrece JDBC para gestionar transacciones son:
  - **Connection.commit**(): Envía al gestor de bases de datos el conjunto de sentencias incluidas en la transacción
  - **Connection.setSavePoint**(String nombre): Crea un punto al que volver en un posible rollback
  - **Connection.rollback(Savepoint sp)** / **Connection.rollback()**: Hace marcha atrás de las sentencias hasta un determinado savepoint dentro de la transacción o de la transacción completa

Transacciones - Ejemplo

- Ejemplo transacción bancaria de una cuenta a otra:
- Primero deshabilitar el autocommit.

```
conexion.setAutoCommit(auto);
```

- Realizar acciones dentro de un try

```
try{
Stm= conexión.preparedStatment("update cuentas set saldo=saldo + ? Where numero=?")
Stm.setDouble(1,cantidad*-1)
Stm.setString(2,cuentaDebito)
Stm.executeUpdate()
Imprimir cuenta debito()
Stm.setDouble(1,cantidad)
Stm.setString(2,cuentaCredito)
Stm.executeUpdate()
```

- Realizar acciones un commit tras las acciones

```
conexion.commit();
}
```

- Realizar un rollback su ha habido alguna excepción

```
catch(Exception e)
try{
Conexión.rollback();
}catch(SQLException e1){e1.printTrace();}
e.printTrace();
}
```