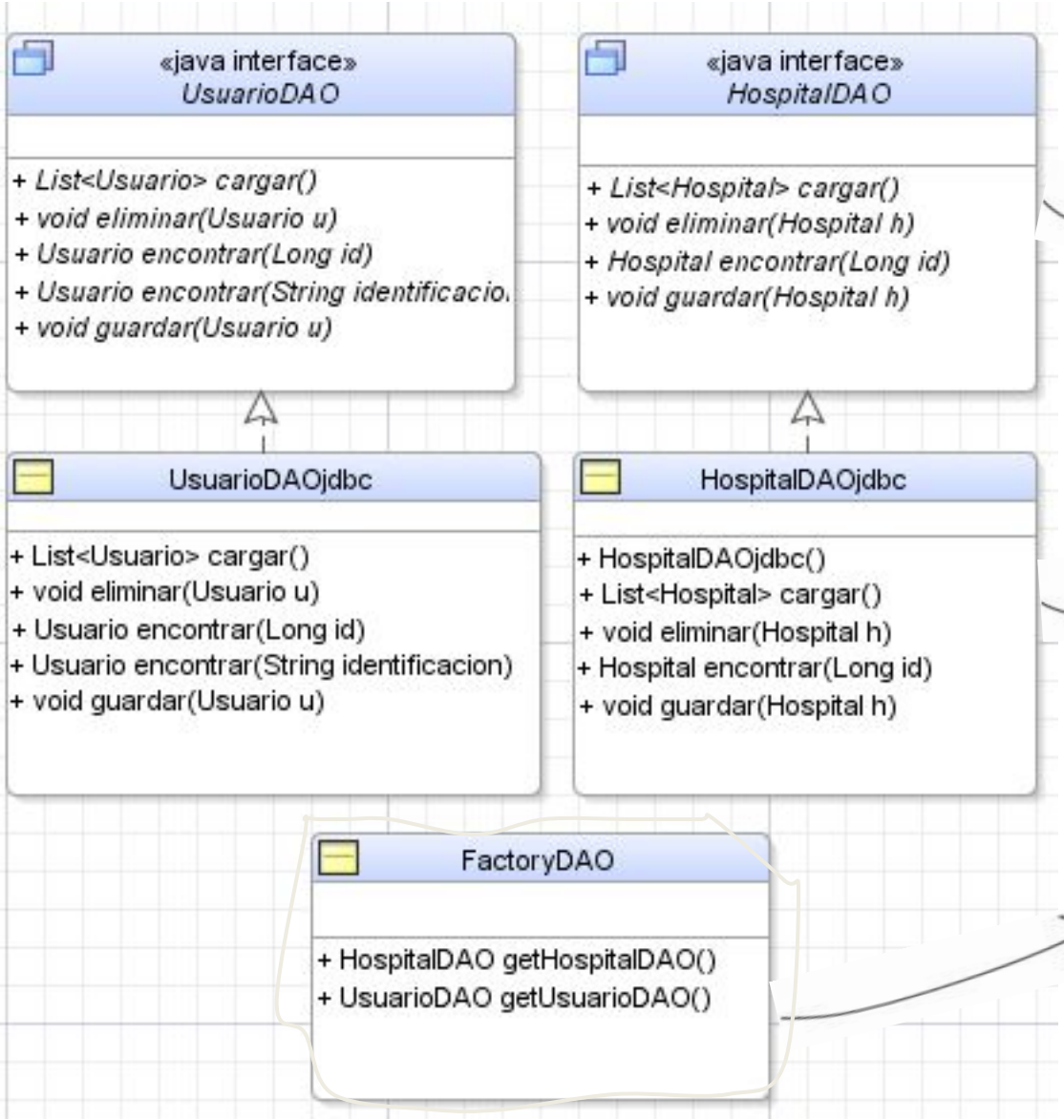
La API JDBC
Clases DAO (Data Access Object)

El patrón arquitectónico Data Access Object (DAO), permite separar la capa de negocios de lógica de acceso a datos, de tal forma que el DAO encapsula toda la lógica de acceso de datos del resto de la aplicación.

Implementar la lógica de acceso a datos en la capa de lógica de negocio puede hacer el código complejo y no extensible. Se recomienda siempre usar DAO para abstraer y encapsular todos los accesos a los datos.
El DAO maneja la conexión con las fuentes de datos para obtener y almacenar datos.

La API JDBC
Clases DAO

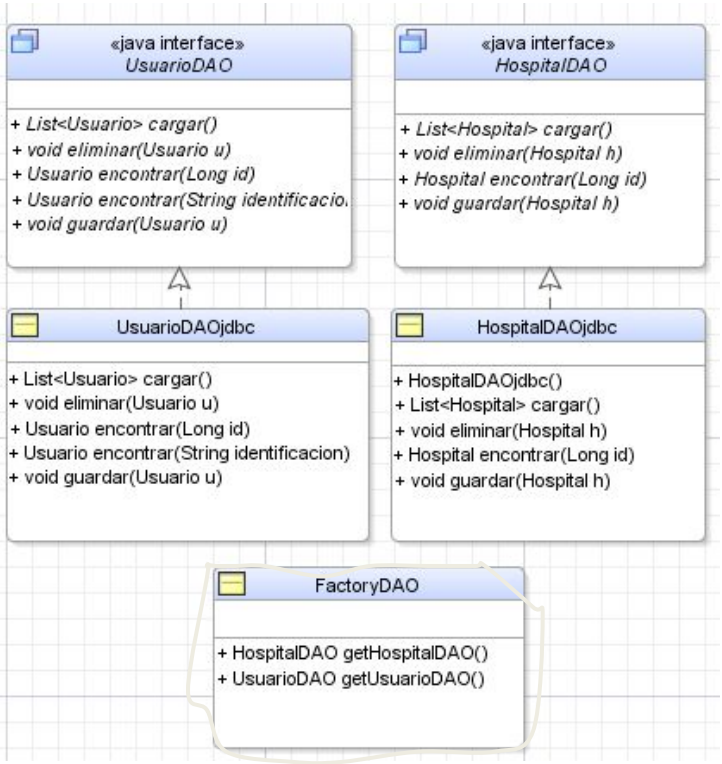


Las interfaces xxxDAO tienen operaciones comunes de acceso a datos.

Implementaciones de las interfaces xxxDAO usando JDBC

Esta clase crea objetos xxxDAO. Nos provee de objetos que implementan las distintas interfaces xxxDAO Estos objetos son usados para acceder a la capa de datos

La API JDBC
Clases DAO

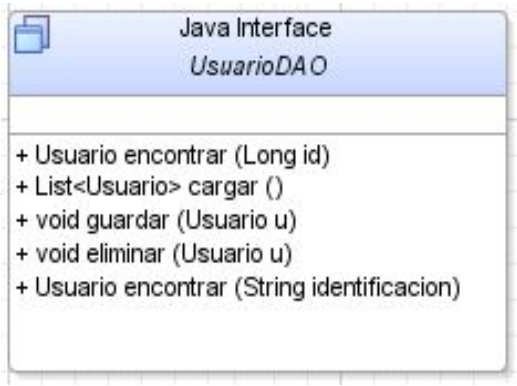



```
public class FactoryDAO {
    public static UsuarioDAO
getUsuarioDAO(){
    return new UsuarioDAOjdbc();
 }
 . . .
}
```

```
package dao.implJDBC;

public class UsuarioDAOjdbc implements UsuarioDAO {

public Usuario encontrar(String identificacion) {
  Usuario usuario = null;
  try{
    Connection con = MiConnection.getCon();
    Statement st = con.createStatement();
    ResultSet rs= st.executeQuery("Select * from Usuarios
            where u.identificacion='"+identificacion+"'");
     if (rs.next()==true) {
         usuario = new Usuario();
         usuario.setMatricula(rs.getInt(1));
         usuario.setApeynom(rs.getString(2));
         // más setters
     }
    rs.close();
    st.close();
    con.close();
  } catch (java.sql.SQLException e) {
    System.out.println("Error de SQL: "+e.getMessage());
  }
  return usuario;
 }

 public List<Usuario> cargar() {…}
 public void eliminar(Usuario u) {…}
 public Usuario encontrar(Long id) {…}
 public void guardar(Usuario u) {…}
}
```

La API JDBC
Clases DAO

Cuando se implemente la aplicación, se invocará al DAO correspondiente para acceder a los datos.




```
Usuario usr = new Usuario();
  . . .

UsuarioDAO uDAO = FactoryDAO.getUsuarioDAO();
Usuario u = uDAO.guardar(usr);
```

Hacer un singleton para el manejo de las conexiones?

El uso de un patrón Singleton para manejar conexiones a la base de datos depende del mecanismo que se utilice para obtener conexiones: `DataSource` o `DriverManager.getConnection`. El patrón Singleton se recomienda para `DataSource` porque la idea es tener un punto central de acceso que administre un conjunto de conexiones (pool de conexiones) a la base de datos. Esto permite reutilizar conexiones y manejar la administración de recursos de manera eficiente, lo que mejora el rendimiento y la escalabilidad de la aplicación.

Ventajas:

- Permite compartir el `DataSource` en toda la aplicación sin necesidad de instanciarlo varias veces.
- Mejora el rendimiento al re-utilizar conexiones en lugar de crear nuevas cada vez que se necesita una.
- Reduce la sobrecarga de recursos y mejora la administración de conexiones.

Hacer un singleton para la conexión?

¿Vale la pena hacer un Singleton con `DriverManager.getConnection`?

Si se utiliza `DriverManager.getConnection`, se abre una nueva conexión a la base de datos cada vez que se invoca al método. Esto puede ser costoso y afectar el rendimiento de una aplicación, ya que cada solicitud requiere el establecimiento de una nueva conexión, lo que implica un gasto considerable de recursos.

¿Cuándo usar un Singleton con `DriverManager.getConnection`? Un Singleton que gestione una única conexión compartida, es poco recomendado en la mayoría de los casos porque:

- No es una práctica segura, ni escalable para aplicaciones concurrentes o de alta carga, ya que varias partes de la aplicación podrían estar intentando acceder a la misma conexión al mismo tiempo.
- Es más propenso a fallos y problemas de bloqueo.

Conclusiones y Recomendación General

- **Para aplicaciones pequeñas o pruebas**: Si estás trabajando en un proyecto pequeño o de prueba, podrías usar `DriverManager.getConnection` sin un Singleton para simplificar la implementación. O podrias hacer un singleton para la única conexión.
- **Para aplicaciones de mediana a gran escala**: Siempre es mejor usar un `DataSource` con un pool de conexiones administrado por un Singleton. Esto te permite gestionar las conexiones de manera eficiente y soportar múltiples usuarios de forma simultánea.

La API JDBC
Singleton con una conexión

Este código obtiene una conexión a la base de datos y la guarda para ser utilizada desde cualquier lugar. Solo aca se necesitan conocer los datos del driver usuario, contraseña.

```
import java.sql.*;

public class MyConnection {
  private static Connection con = null;
  static {
    try {
        con = DriverManger.getConnection("jdbc:sqlite:test.db");
    } catch (java.sql.SQLException e) {
        System.out.println("Error de SQL: "+e.getMessage());
    }

   public static Connection getCon() {
     return con;
   }

   private MyConnection() {
   }
}
```

Hacer un singleton para la conexión?

Se recomienda usar un Singleton para manejar conexiones a la base de datos utilizando `DataSource`

```
public class MiDataSource {

 private static DataSource dataSource = null;
 static {
    try {
     dataSource=
         (DataSource) new InitialContext().lookup("java:comp/env/jdbc/wallet");
    } catch (NamingException e) {
         e.printStackTrace();
    }
 }
 public static DataSource getDataSource(){
    return dataSource;
 }

 private MiDataSource(){}
}
```