



Argentina programa 4.0

Clase 3

Contenido

Patrón de diseño DAO	2
Creación de interfaces DAO. Implementación de la interfaz dao.	7
Implementación en JAVA	18

Patrón de diseño DAO

Prácticamente todas las aplicaciones de hoy en día, requiere acceso al menos a una fuente de datos, dichas fuentes son por lo general base de datos relacionales, por lo que muchas veces no tenemos problema en acceder a los datos, sin embargo, hay ocasiones en las que necesitamos tener más de una fuente de datos o la fuente de datos que tenemos puede variar, lo que nos obligaría a refactorizar gran parte del código. Para esto, tenemos el patrón Arquitectónico *Data Access Object* (DAO), el cual permite separar la lógica de acceso a datos de los *Business Objects* u Objetos de negocios, de tal forma que el DAO encapsula toda la lógica de acceso de datos al resto de la aplicación.

Patrón de objeto de acceso a datos

El patrón de objeto de acceso a datos o patrón DAO se utiliza para separar las operaciones o la API de acceso a datos de bajo nivel de los servicios comerciales de alto nivel.

La idea de este patrón es sencilla. En primer lugar, debemos hacer las clases que representan nuestros datos. Por ejemplo, podemos hacer una clase Persona con los datos de la persona y los métodos set() y get() correspondientes.

Luego hacemos una interface. Esta interface tiene que tener los métodos necesarios para obtener y almacenar Personas. Esta interface no debe tener nada que la relacione con una base de datos ni cualquier otra cosa específica del medio de almacenamiento que vayamos a usar, es decir, ningún parámetro debería ser una Connection, ni un nombre de fichero, etc.

Por ejemplo, puede ser algo así

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

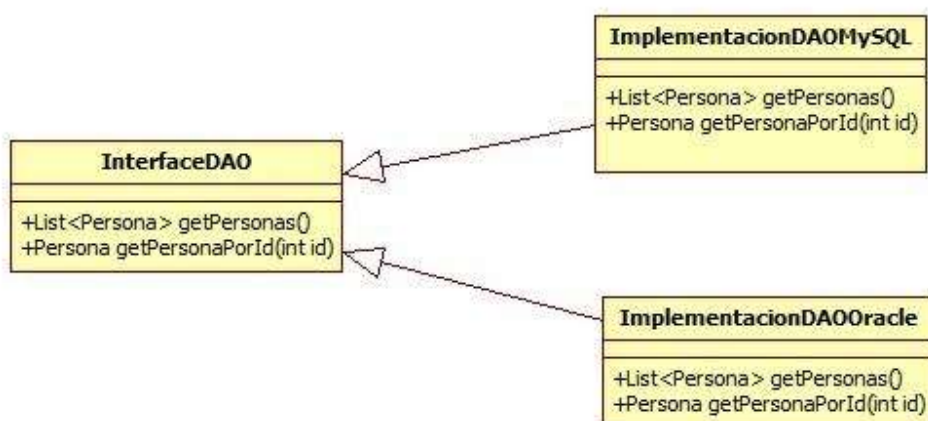
```
...  
public void borraPersonaPorNombre (String nombre);  
...  
}
```

y todos los métodos con todas las variantes que necesitemos en nuestra aplicación.

Con esto deberíamos construir nuestra aplicación, usando la clase *Persona* y usando la *InterfaceDAO* para obtener y modificar Personas.

Aparte, hacemos la implementación de la *InterfaceDAO*, ya contra una base de datos concreta o usando una herramienta determinada. Al usar nuestra aplicación la *InterfaceDAO*, podremos pasarle cualquier implementación que queramos o incluso cambiar una por otra en un momento dado.

Por ejemplo, podemos hacer implementaciones según usemos MySQL, Oracle, etc



Los siguientes son los participantes en el patrón de objetos de acceso a datos.

Interfaz de objeto de acceso a datos

: Define las operaciones estándar que serializarán en un objeto modelo.

Clase concreta de objeto de acceso a datos

: Implementa la interfaz anterior.

Esta clase es responsable de obtener datos de una fuente de datos que puede ser una base de datos/xml o cualquier otro mecanismo de almacenamiento.

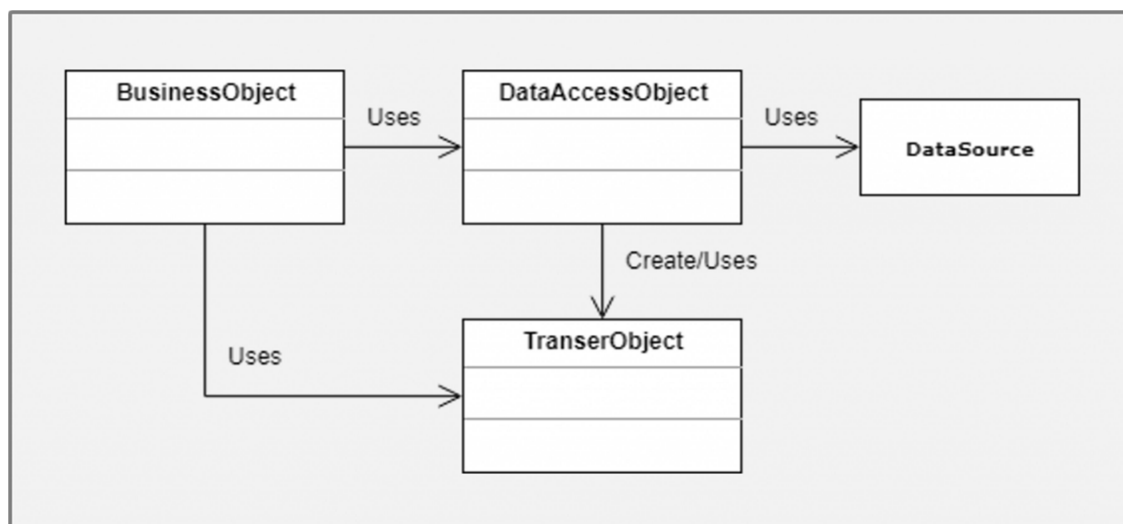
Objeto modelo u objeto de valor

: Es un POJO (Plain Old Java Object o antiguo objeto plano de java) simple que contiene métodos get/set para almacenar datos recuperados mediante la clase DAO.

Ahora para aplicar el patrón vamos a ver la siguiente problemática:

Una de las grandes problemáticas al momento de acceder a los datos, es que la implementación y formato de la información puede variar según la fuente de los datos. Implementar la lógica de acceso a datos en la capa de lógica de negocio puedes ser un gran problema, pues tendríamos que lidiar con la lógica de negocio en sí, más la implementación para acceder a los datos, adicional, si tenemos múltiples fuentes de datos o estas pueden variar, tendríamos que implementar las diferentes lógicas para acceder las diferentes fuentes de datos, como podrían ser: bases de datos relacionales, No SQL, XML, archivos planos, Webservices, etc).

La solución sería: Dado lo anterior, el patrón DAO propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.

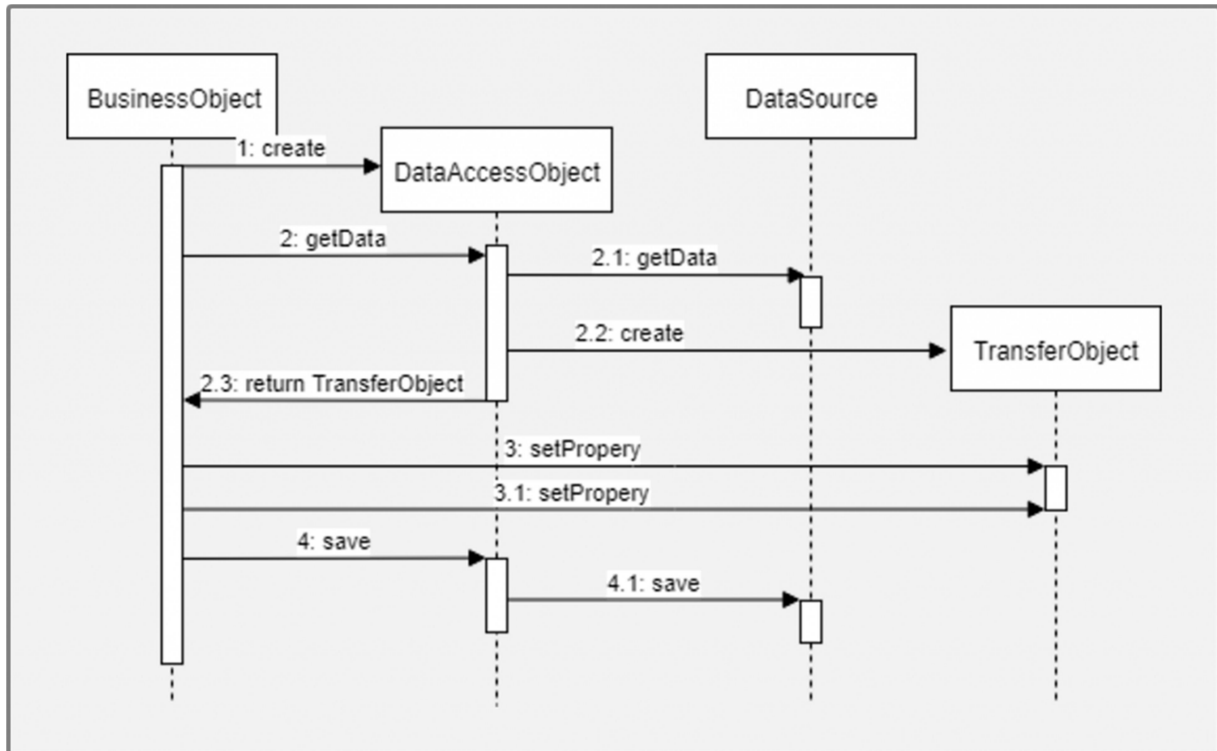




Los componentes que conforman el patrón son:

- **BusinessObject**: Representa un objeto con la lógica de negocio.
- **DataAccessObject**: Representa una capa de acceso a datos que oculta la fuente y los detalles técnicos para recuperar los datos.
- **TransferObject**: Este es un objeto plano que implementa el patrón Data Transfer Object (DTO), el cual sirve para transmitir la información entre el DAO y el Business Service.
- **DataSource**: Representa de forma abstracta la fuente de datos, la cual puede ser una base de datos, Webservices, LDAP, archivos de texto, etc.

El siguiente diagrama muestra mejor la forma en que funciona el patrón, pues muestra de forma secuencial la forma en que se ejecutaría el patrón.



El diagrama se interpreta de la siguiente manera:

- El *BusinessObject* crea u obtiene una referencia al *DataAccessObject*.
- El *BusinessObject* solicita información al *DataAccessObject*
- El *DataAccessObject* solicita la información al *DataSource*
- El *DataAccessObject* crea una instancia del *TransferObject* con los datos recuperados del *DataSource*
- El *DataAccessObject* responde con el *TransferObject* creado en los pasos anteriores.
- El *BusinessObject* actualiza algún valor del *TransferObject*
- Más actualizaciones
- El *BusinessObject* solicita el guardado de los datos actualizados al *DataAccessObject*.
- El *DataAccessObject* guarda los datos en el *DataSource*.



Como hemos podido ver, el `BusinessService` no se preocupa de donde vengan los datos ni cómo deben de ser guardados en el `DataSource`, el solo se preocupa por conocer el `TransferObject`. Un error común al implementar este patrón es no utilizar `TransferObject` y en su lugar, regresar los objetos que regresan las mismas API's de las fuentes de datos, ya que esto obliga al `BusinessService` tener una dependencia con estas librerías, además, si la fuente de datos cambia, también cambiarán estas clases, lo que provocaría una afectación directa al `BusinessService`.

Creación de interfaces DAO. Implementación de la interfaz dao.

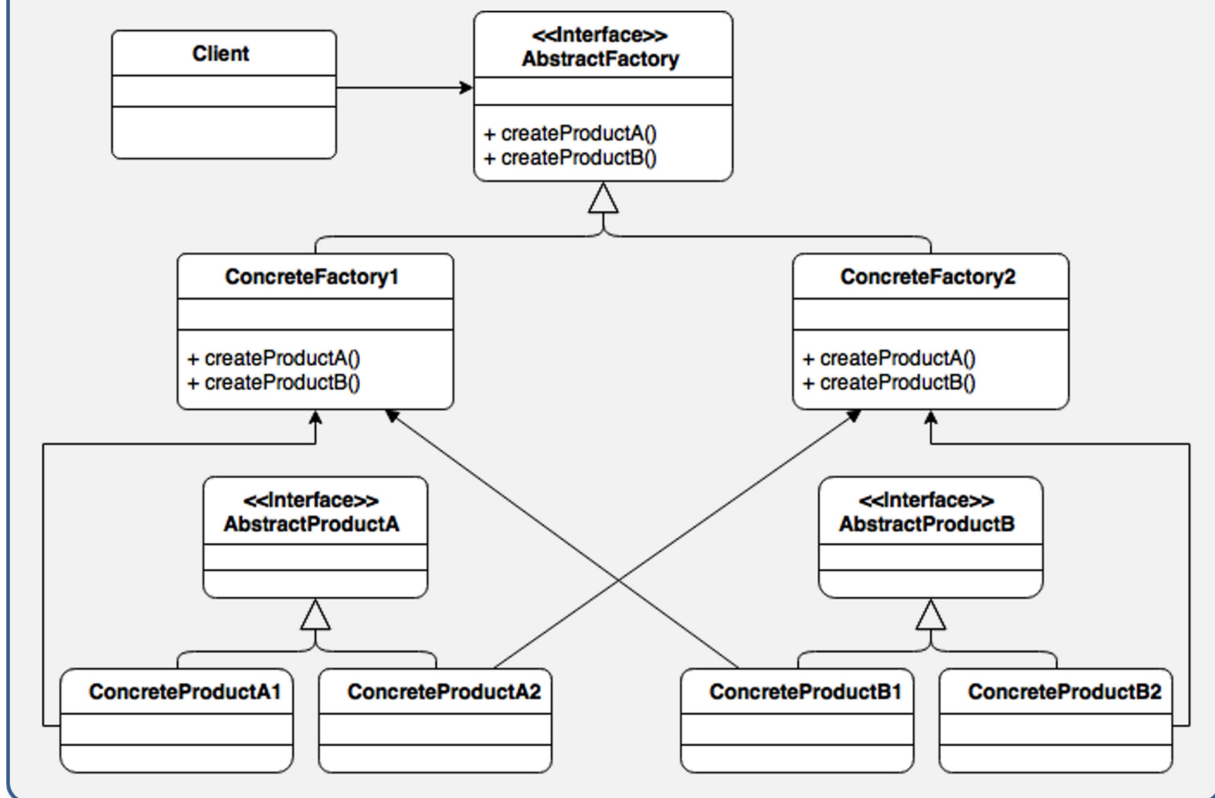
Aquí vemos una aplicación del DAO y el patrón Abstract Factory

Hasta este punto solo hemos analizado cómo trabajaríamos si solo tuviéramos una fuente de datos, sin embargo, existen ocasiones donde requerimos obtener datos de más de una fuente, y es allí donde entra el patrón de diseño `Abstract Factory`.

Para repasar el **Abstract Factory** Patrón de diseño creacional

El patrón de diseño `Abstract Factory` busca agrupar un conjunto de clases que tiene un funcionamiento en común llamadas familias, las cuales son creadas mediante un `Factory`, este patrón es especialmente útil cuando requerimos tener ciertas familias de clases para resolver un problema, sin embargo, puede que se requieran crear implementaciones paralelas de estas clases para resolver el mismo problema pero con una implementación distinta.

Abstract Factory – Class diagram



Estructura del patrón de diseño Abstract Factory.

La estructura de Abstract Factory puede resultar muy enredosa ya que tiene muchos componentes y éstos aparentemente se mezclan entre sí. Para comprender mejor cómo funciona este patrón explicaremos cada componente:

- **Client**: Representa la persona o evento que dispara la ejecución del patrón.
- **AbstractProduct (A, B)**: Interfaces que definen la estructura de los objetos para crear familias.
- **ConcreteProduct (A, B)**: Clases que heredan de AbstractProduct con el fin de implementar familias de objetos concretos.

- **ConcreteFactory:** Representan las fábricas concretas que servirán para crear las instancias de todas las clases de la familia. En esta clase debe existir un método para crear cada una de las clases de la familia.
- **AbstractFactory:** Define la estructura de las fábricas y deben proporcionar un método para cada clase de la familia.

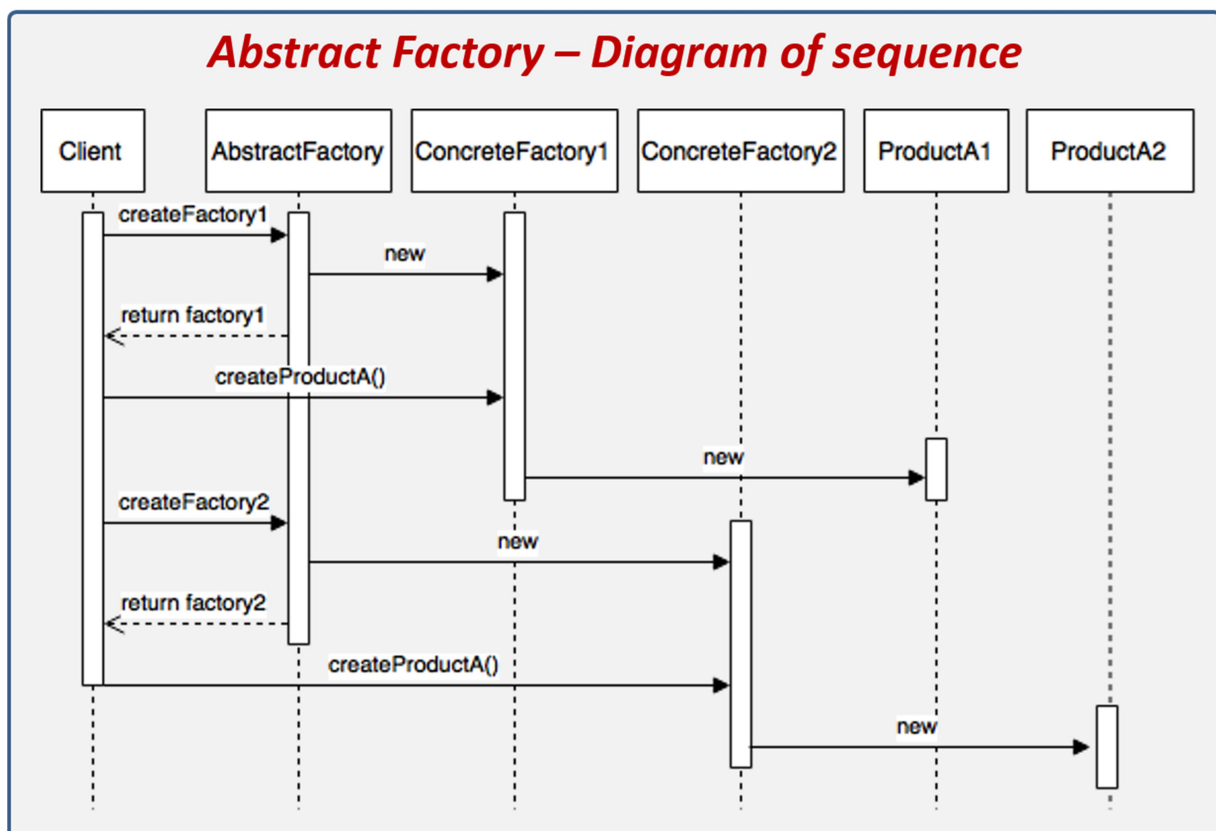


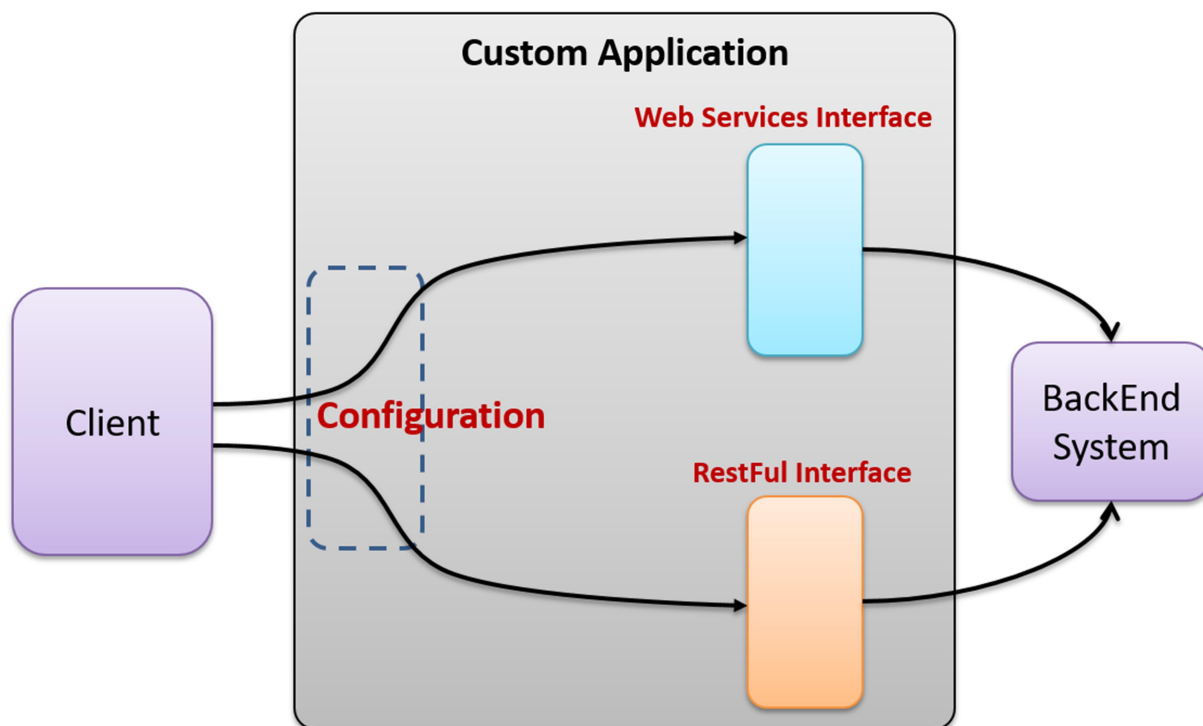
Diagrama de secuencia del patrón Abstract Factory.

1. El cliente solicita la creación del ConcreteFactory1 al AbstractFactory.
2. El AbstractFactory crea una instancia del ConcreteFactory1 y la regresa.
3. El cliente le solicita al ConcreteFactory1 la creación de un ProductA.
4. El ConcreteFactory1 crea una instancia del ProductA1 el cual es parte de la familia1 y lo regresa.
5. El cliente esta vez solicita la creación del ConcreteFactory2 al AbstractFactory.

6. El AbstractFactory crea una instancia del ConcreteFactory2 .
7. El cliente le solicita al ConcreteFactory2 la creación de un ProductA .
8. El ConcreteFactory2 crea una instancia del ProductA2 el cual es parte de la familia2 y lo devuelve.

Ejemplo del mundo real

Mediante la implementación del patrón de diseño Abstract Factory desarrollaremos una aplicación que permite la comunicación con el Back End mediante Web Services y servicios REST, de tal forma que nuestro cliente pueda elegir mediante configuración el tipo de comunicación que se utilizará.

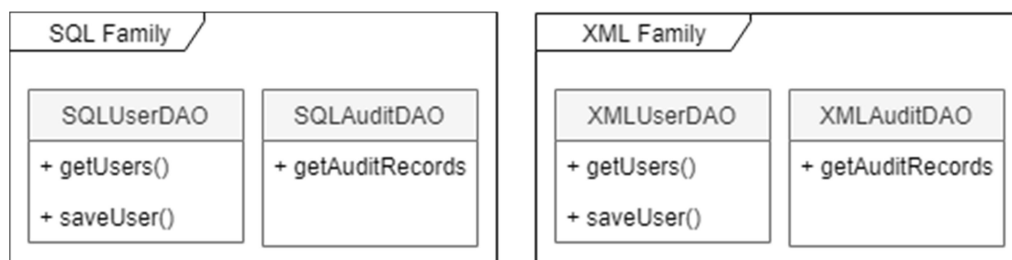


Descubre cómo el patrón Abstract Factory nos ayuda a resolver este problema.

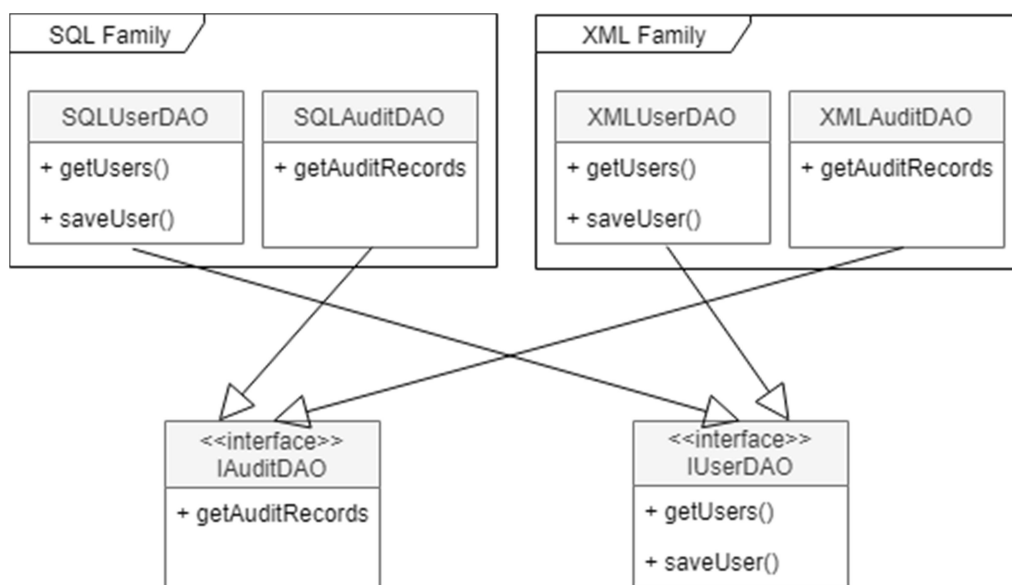
Continuando mediante el patrón Abstract Factory podemos definir una serie de familias de clases que permitan conectarnos a las diferentes fuentes de datos. Para esto, examinaremos un sistema de autenticación de usuarios, el cual puede leer los usuarios

en una base de datos o sobre un XML, adicional, el sistema generará registros de login que podrán ser utilizados para auditorías.

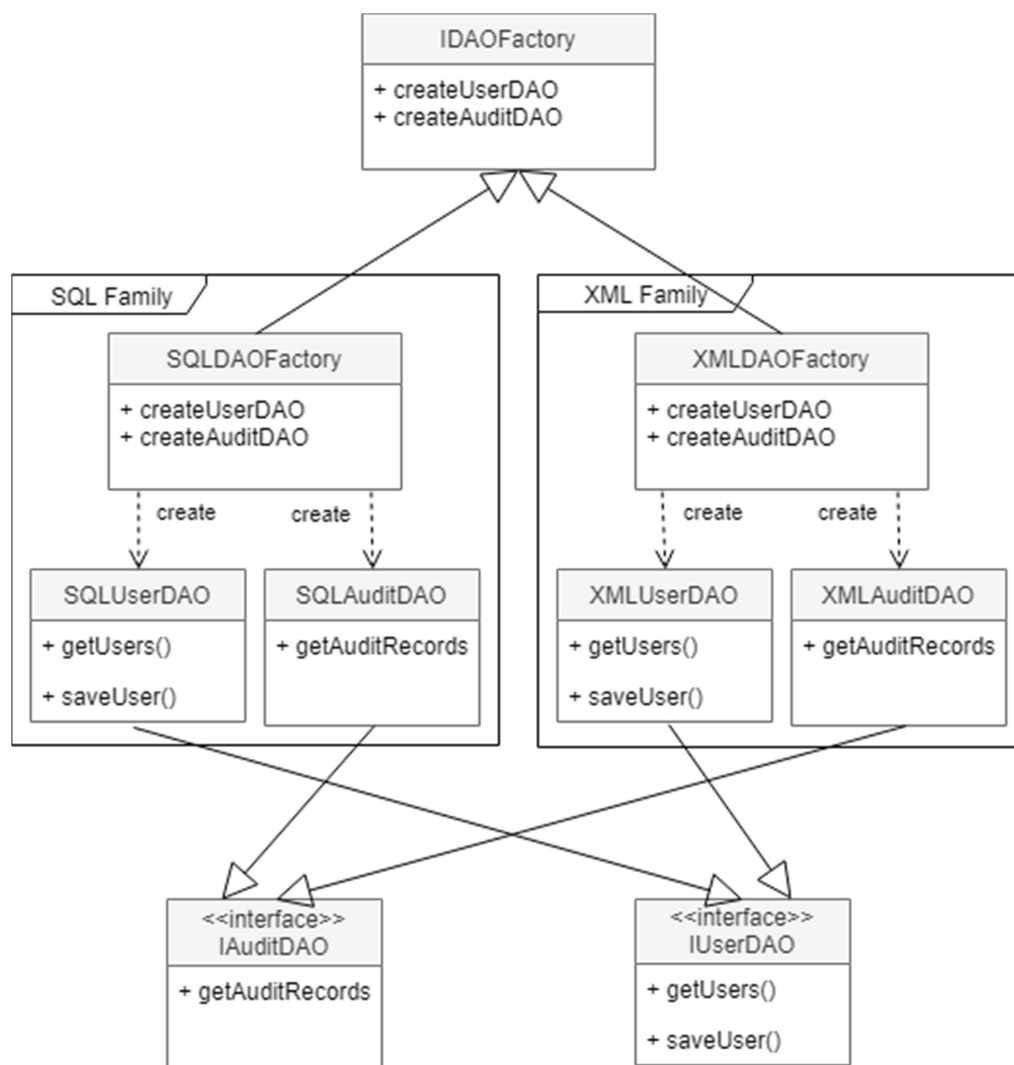
Lo primero sería implementar las clases para acceder de las dos fuentes:



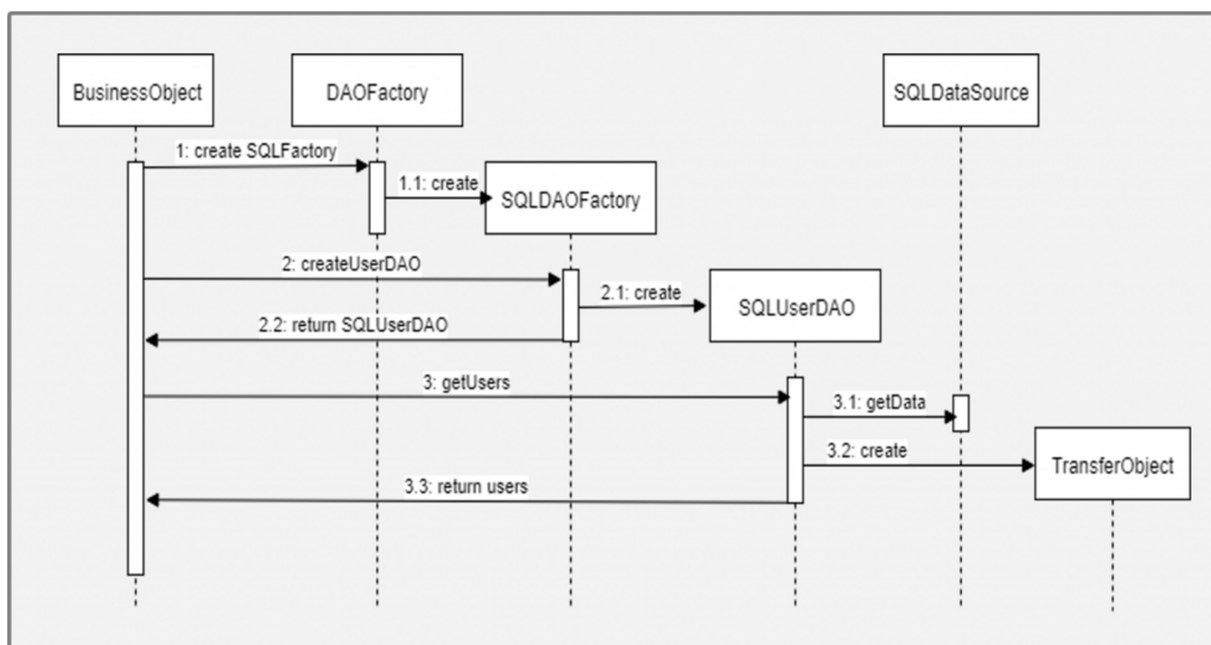
En la imagen anterior podemos apreciar dos familias de clases, con las cuales podemos obtener los Usuarios y los registros de auditoría, sin embargo, estas clases por separado no ayudan mucho, pues no implementan una misma interface que permita la variación entre ellas, por lo que el siguiente paso es crear estas interfaces:



En este punto, los DAO ya implementan una interfaz común, lo que permite intercambiar la implementación sin afectar al Business Object. Sin embargo, ahora solo falta resolver la forma en que el Business Service obtendrá la familiar de interfaces, es por ello que deberemos crear un Factory para cada familia de interfaces:



En esta nueva configuración, podemos ver que tenemos un Factory para cada familia, y los dos factories implementan una interfaz en común, adicional, tenemos la interface IDAOFactory necesaria para que el factory de cada familia implementen una interfaz en común.




Analicemos cómo quedaría la secuencia de ejecución

1. El BusinessObject solicita la creación de un DAOFactory para SQL
 - El DAOFactory crea una instancia de la clase SQLDAOFactory y la retorna
2. El BusinessObject solicita al SQLDAOFactory la creación del SQLUserDAO para interactuar con los usuarios.
 - El SQLDAOFactory crea una nueva instancia del SQLUserDAO
 - El SQLDAOFactory retorna la instancia creada del SQLUserDAO
3. El BusinessObject solicita el listado de todos los usuarios registrados al SQLUserDAO
 - El SQLUserDAO recupera los usuarios del SQLDataSource
 - El SQLUserDAO crea un TransferObject con los datos recuperados del paso anterior.
 - El SQLUserDAO retorna el TransferObject creado en el paso anterior.

Adicional a los pasos que hemos listado aquí, podríamos solicitar al SQLDAOFactory la creación del SQLAuditDAO o incluso, solicitar al DAOFactory la creación del XMLFactory para interactuar con la fuente de datos en XML.

Podemos concluir que el patrón DAO es sin lugar a duda, uno de los más utilizados en la actualidad, ya que es fácil de implementar y proporciona claros beneficios, incluso, si solo tenemos una fuente de datos y esta no cambia, pues permite separar por completo la lógica de acceso a datos en una capa separada y así solo nos preocupamos por la lógica de negocio sin preocuparnos de donde viene los datos o los detalles técnicos para consultarlos o actualizarlos.

Suponiendo la siguiente tabla y que le hemos creado un DTO



inquilinos	
idinquilinos	INT
dni	VARCHAR(8)
nombres	VARCHAR(150)
paterno	VARCHAR(150)
materno	VARCHAR(150)
telefono	VARCHAR(40)
correo	VARCHAR(200)
deuda	DECIMAL(6,2)
fecha_ingreso	DATE

CÓDIGO SQL

El código resultante será el siguiente:

```
1. CREATE TABLE IF NOT EXISTS inquilinos (  
2.     idinquilinos INT NOT NULL AUTO_INCREMENT,  
3.     dni VARCHAR(8) NOT NULL,  
4.     nombres VARCHAR(150) NOT NULL,  
5.     paterno VARCHAR(150) NOT NULL,
```

```
6.      materno VARCHAR(150) NOT NULL,  
7.      telefono VARCHAR(40) NULL,  
8.      correo VARCHAR(200) NULL,  
9.      deuda DECIMAL(10,2) NOT NULL,  
10.     fecha_ingreso DATE NOT NULL,  
11.     PRIMARY KEY (idinquilinos),  
12.     UNIQUE INDEX dni_UNIQUE (dni ASC))  
13.     ENGINE = InnoDB  
14.     DEFAULT CHARACTER SET = utf8;
```

- El campo "idinquilinos" sera "AUTO_INCREMENT", lo que significa que iniciará en 1 y cada registro que se inserte generará automáticamente un nuevo valor sin que necesitemos especificarlo. Al ser un dato numérico simple y no un código se obtiene mayor velocidad en consultas a la base de datos.
- El campo "idinquilinos" sera "PRIMARY KEY", lo que implica que será la clave primaria es decir cada registro tendrá un único identificador que es este campo y nos servirá para realizar búsquedas o emplearlo como referencias para claves foráneas a otras tablas.
- El campo "dni" es de tipo "UNIQUE", lo que implica que no se podrá repetir este valor en ningún registro de esta tabla.
- ENGINE = InnoDB, nos garantiza que podremos utilizar claves foráneas y soporte del "commit" y "rollback"
- DEFAULT CHARACTER SET = utf8; nos permite ingresar caracteres especiales como la "ñ" o vocales tildadas a nuestros registros.

La interfaz "DaoInquilinos.java" se colocará dentro de un paquete llamado "dao", el código para dar un mantenimiento básico a la tabla "inquilinos" será

```
1. package dao;  
2. import java.util.List;  
3. import dto.Inquilinos;  
4.  
5. public interface DaoInquilinos {  
6.     public List <Inquilinos> inquilinosSel();
```



```
7. public Inquilinos inquilinosGet(Integer id);  
8. public String inquilinosIns(Inquilinos inquilino);  
9. public String inquilinosUpd(Inquilinos inquilino);  
10. public String inquilinosDel(List<Integer> ids);  
11. public String getMessage();  
12. }
```

Ahora debemos implementar cada una de esas funciones en un archivo llamado "DaoInquilinosImpl.java" que colocaremos en un paquete llamado "impl" dentro del paquete "dao", para ello empleamos la palabra reservada "implements" que indica que es la implementación del "DaoInquilinos". El código será:

```
1. package dao.impl;  
2.  
3. import biblioteca.ConectaBD;  
4. import dao.DaoInquilinos;  
5. import dto.Inquilinos;  
6. import java.sql.Connection;  
7. import java.sql.PreparedStatement;  
8. import java.sql.ResultSet;  
9. import java.sql.SQLException;  
10. import java.time.LocalDate;  
11. import java.util.ArrayList;  
12. import java.util.List;  
13.  
14. public class DaoInquilinosImpl implements DaoInquilinos {  
15.
```

```
16. private final ConectaBD conectaDb;
17. private String mensaje;
18.
19. public DaoInquilinosImpl() {
20.     this.conectaDb = new ConectaBD();
21. }
22.
23. @Override
24. public List<Inquilinos> inquilinosSel() {
25.     List<Inquilinos> list = null;
26.     StringBuilder sql = new StringBuilder();
27.     sql.append("SELECT ")
28.         .append("idinquilinos,")
29.         .append("dni,")
30.         .append("nombres,")
31.         .append("paterno,")
32.         .append("materno,")
33.         .append("telefono,")
34.         .append("correo,")
35.         .append("deuda,")
36.         .append("fecha_ingreso")
37.         .append(" FROM inquilinos");
38.     try (Connection cn = conectaDb.conexionDB()) {
39.         PreparedStatement ps = cn.prepareStatement(sql.toString());
40.         ResultSet rs = ps.executeQuery();
41.         list = new ArrayList<>();
42.         while (rs.next()) {
```

```
43.      Inquilinos inquilinos = new Inquilinos();
44.      inquilinos.setIdinquilinos(rs.getInt(1));
45.      inquilinos.setDni(rs.getString(2));
46.      inquilinos.setNombres(rs.getString(3));
47.      inquilinos.setPaterno(rs.getString(4));
48.      inquilinos.setMaterno(rs.getString(5));
49.      inquilinos.setTelefono(rs.getString(6));
50.      inquilinos.setCorreo(rs.getString(7));
51.      inquilinos.setDeuda(rs.getDouble(8));
52.      inquilinos.setFecha_ingreso(LocalDate.parse(rs.getString(9)));
53.      list.add(inquilinos);
54.  }
55.  } catch (SQLException e) {
56.      mensaje = e.getMessage();
57.  }
58.  return list;
59.  }
60.
61.  @Override
62.  public Inquilinos inquilinosGet(Integer id) {
63.      Inquilinos inquilino = new Inquilinos();
64.      StringBuilder sql = new StringBuilder();
65.      sql.append("SELECT ")
66.          .append("idinquilinos,")
67.          .append("dni,")
68.          .append("nombres,")
69.          .append("paterno,")
```

```
70.         .append("materno,")
71.         .append("telefono,")
72.         .append("correo,")
73.         .append("deuda,")
74.         .append("fecha_ingreso")
75.         .append(" FROM inquilinos WHERE idinquilinos = ?");
76.     try (Connection cn = conectaDb.conexionDB()) {
77.         PreparedStatement ps = cn.prepareStatement(sql.toString());
78.         ps.setInt(1, id);
79.         try (ResultSet rs = ps.executeQuery()) {
80.             if (rs.next()) {
81.                 inquilino.setIدينquilinos(rs.getInt(1));
82.                 inquilino.setDni(rs.getString(2));
83.                 inquilino.setNombres(rs.getString(3));
84.                 inquilino.setPaterno(rs.getString(4));
85.                 inquilino.setMaterno(rs.getString(5));
86.                 inquilino.setTelefono(rs.getString(6));
87.                 inquilino.setCorreo(rs.getString(7));
88.                 inquilino.setDeuda(rs.getDouble(8));
89.                 inquilino.setFecha_ingreso(LocalDate.parse(rs.getString(9)));
90.             } else {
91.                 inquilino = null;
92.             }
93.         } catch (SQLException e) {
94.             mensaje = e.getMessage();
95.         }
96.     } catch (SQLException e) {
```

```
97.         mensaje = e.getMessage();
98.     }
99.     return inquilino;
100. }
101.
102.     @Override
103.     public String inquilinosIns(Inquilinos inquilino) {
104.         StringBuilder sql = new StringBuilder();
105.         sql.append("INSERT INTO inquilinos( ")
106.             .append("dni,")
107.             .append("nombres,")
108.             .append("paterno,")
109.             .append("materno,")
110.             .append("telefono,")
111.             .append("correo,")
112.             .append("deuda,")
113.             .append("fecha_ingreso")
114.             .append(") VALUES (?,?,?,?,?,?,?,?) ");
115.         try (Connection cn = conectaDb.conexionDB()) {
116.             PreparedStatement ps = cn.prepareStatement(sql.toString());
117.             ps.setString(1, inquilino.getDni());
118.             ps.setString(2, inquilino.getNombres());
119.             ps.setString(3, inquilino.getPaterno());
120.             ps.setString(4, inquilino.getMaterno());
121.             ps.setString(5, inquilino.getTelefono());
122.             ps.setString(6, inquilino.getCorreo());
123.             ps.setDouble(7, inquilino.getDeuda());
```

```
124.         ps.setString(8, inquilino.getFecha_ingreso().toString());
125.         int ctos = ps.executeUpdate();
126.         if (ctos == 0) {
127.             mensaje = "cero filas insertadas";
128.         }
129.     } catch (SQLException e) {
130.         mensaje = e.getMessage();
131.     }
132.     return mensaje;
133. }
134.
135. @Override
136. public String inquilinosDel(List<Integer> ids) {
137.     StringBuilder sql = new StringBuilder();
138.     sql.append("DELETE FROM inquilinos WHERE ")
139.         .append("idinquilinos = ? ");
140.     try (Connection cn = conectaDb.conexionDB()) {
141.         PreparedStatement ps = cn.prepareStatement(sql.toString());
142.         cn.setAutoCommit(false);
143.         boolean ok = true;
144.         for (int id = 0; id < ids.size(); id++) {
145.             ps.setInt(1, ids.get(id));
146.             int ctos = ps.executeUpdate();
147.             if (ctos == 0) {
148.                 ok = false;
149.                 mensaje = "ID: " + ids.get(id) + " no existe";
150.             }
```

```
151.         }  
152.         if (ok) {  
153.             cn.commit();  
154.         } else {  
155.             cn.rollback();  
156.         }  
157.         cn.setAutoCommit(true);  
158.     } catch (SQLException e) {  
159.         mensaje = e.getMessage();  
160.     }  
161.     return mensaje;  
162. }  
163.  
164. @Override  
165. public String inquilinosUpd(Inquilinos inquilino) {  
166.     StringBuilder sql = new StringBuilder();  
167.     sql.append("UPDATE inquilinos SET ")  
168.         .append("dni = ?,")  
169.         .append("nombres = ?,")  
170.         .append("paterno = ?,")  
171.         .append("materno = ?,")  
172.         .append("telefono = ?,")  
173.         .append("correo = ?,")  
174.         .append("deuda = ?,")  
175.         .append("fecha_ingreso = ? ");  
176.     sql.append("WHERE idinquilinos = ? ");  
177.     try (Connection cn = conectaDb.conexionDB()) {
```

```
178.         PreparedStatement ps = cn.prepareStatement(sql.toString());
179.         ps.setString(1, inquilino.getDni());
180.         ps.setString(2, inquilino.getNombres());
181.         ps.setString(3, inquilino.getPaterno());
182.         ps.setString(4, inquilino.getMaterno());
183.         ps.setString(5, inquilino.getTelefono());
184.         ps.setString(6, inquilino.getCorreo());
185.         ps.setDouble(7, inquilino.getDeuda());
186.         ps.setString(8, inquilino.getFecha_ingreso().toString());
187.         ps.setInt(9, inquilino.getIdinquilinos());
188.         int ctos = ps.executeUpdate();
189.         if (ctos == 0) {
190.             mensaje = "No se pudo actualizar";
191.         }
192.     } catch (SQLException e) {
193.         mensaje = e.getMessage();
194.     }
195.     return mensaje;
196. }
197.
198. @Override
199. public String getMessage() {
200.     return mensaje;
201. }
202. }
```

Teniendo la siguiente estructura de carpetas para un proyecto de NetBeans:

