IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

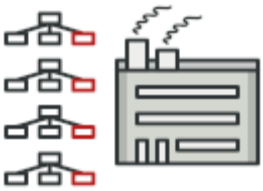
PATRONES DISEÑO

PATRONES CREACIONALES

- BUILDER O CONSTRUCTOR VIRTUAL

Permite abstraer el proceso de creación de un objeto y centralizarlo en un único punto.

- ABSTRACT FACTORY O FÁBRICA ABSTRACTA



Permite trabajar con objetos de distintas familias de forma trasparente. En ocasiones, en la creación de d interfaces, es preciso tratar con objetos de distintos tipos (botones, ventanas, etiquetas, etc) y una abstract factory facilita tanto el diseño como la implementación.

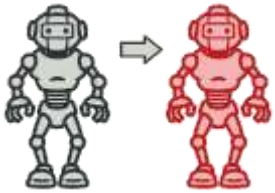
Por ejemplo conexión a distintas BD's

Muchos *frameworks* y bibliotecas lo utilizan para proporcionar una forma de extender y personalizar sus componentes estándar.

Aquí tienes algunos ejemplos de las principales bibliotecas de Java:

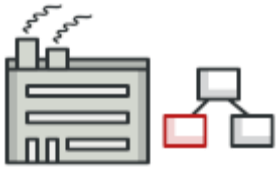
- `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`
- `javax.xml.xpath.XPathFactory#newInstance()`

- PROTOTYPE O PROTOTIPO



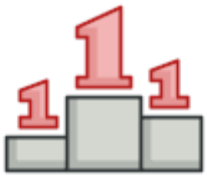
Servirá para crear nuevos objetos clonándolos de un objeto (instancia) ya existente.

- FACTORY METHOD O MÉTODO FABRICACIÓN



Se tendrá una clase constructora que determinará el subtipo de objeto que va a crearse. Se utiliza cuando la casuística o diversidad de subtipos de objetos es grande.

- SINGLETON



En el patrón de diseño Sigleton, se reduce o limita el número de objetos que ha de crearse a solo uno. Igual que las variables estáticas solamente existirá un objeto por clase.

Utilice este patrón cuando vayan a realizarse conexiones a bases de datos o a programar sockets. De esta forma, se centrará en el acceso al recurso y controlará el número de sesiones activas o conexiones.

Véase ejemplo de implementación. El código será el siguiente:

```
public class Singleton {
   private static Singleton instance =
   null;
   private Singleton() {
      // Este método existe solamente
      para evitar la instanciación.
   }
   // Se utiliza el método getInstance
   para controlar la instanciación.
   //solamente existirá un objeto
   único para la clase.
   public static Singleton
   getInstance() {
      if(instance == null) {
      instance = new Singleton();
   }
   return instance;
   }
   // Se definen los distintos métodos
   de la clase singleton
   protected static void dimeAlgo( ) {
   System.out.println("Método de la
   clase singleton");
   }
}
```

Como puede observarse, la clase tiene una referencia estática a la única instancia Singleton y la devuelve invocando al método getInstance().

El código de la clase PruebaSingleton será el siguiente:

```
public class PruebaSingleton {
   public static void main(Strine] args) {
   Singleton tmp = Singleton.getInstance( );
   tmp.dimeAlgo( );
   }
}
```

- OBJECT POOL O CONJUNTO DE OBJETOS

(Este patrón no pertenece al GoF)

Este patrón de diseño utiliza un conjunto de objetos previamente inicializados para su uso inmediato. Permite mayor eficiencia, puesto que no necesita ir creando y destruyendo objetos según necesidad. Este patrón busca la eficiencia y la fluidez en la ejecución.

El funcionamiento es sencillo, un cliente le pide un objeto pool y, cuando termina de ejecutarlo, lo devuelve y estará disponible para otro cliente que lo necesite.

Como puede observarse, no se crean (salvo la primera vez) ni se destruyen objetos, sino que se reciclan.

- PATRÓN MVC (MODELO VISTA CONTROLADOR)

  http://myfpschool.com/que-es-el-mvc-framework/

MVC o Model View Controller (Modelo Vista Controlador) no es un lenguaje de programación, es un patrón de arquitectura software para implementar aplicaciones. Este modelo divide las aplicaciones en tres partes diferentes (componentes) pero interconectadas para poder desarrollar los interfaces al usuario. Cada uno de esos tres componentes se encarga de un aspecto de la aplicación.

Veamos cada uno de estos componentes y cuales son sus características:

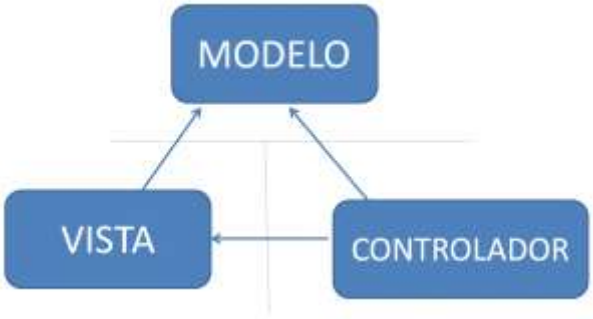
- **Modelo**: Contiene la lógica de la aplicación. En este componente residen los objetos y los datos. Es el encargado de tratar con los datos que van a ser transferidos entre el controlador y la vista.

- **Vista**: Contiene el interfaz de la aplicación. Es lo que el usuario ve y con lo que interacciona. En este componente tendremos cajas de texto, botones, listas, listas desplegables, pestañas, etc.

- **Controlador**: Hace de interfaz entre el modelo y la vista. Siempre tiene que haber un responsable que recoja los eventos que se generan en la vista y que manipule los datos utilizando el componente modelo. Se puede entender como un mensajero que envía y trae de vuelta datos.

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

Veamos una imagen que ilustra la relación entre estos tres componentes:



Existen muchos lenguajes que soportan la arquitectura Modelo-Vista-Controlador como por ejemplo Java, Swift, Objetive-C, ASP.NET, etc.

Microsoft

ASP.NET **ASP.NET**

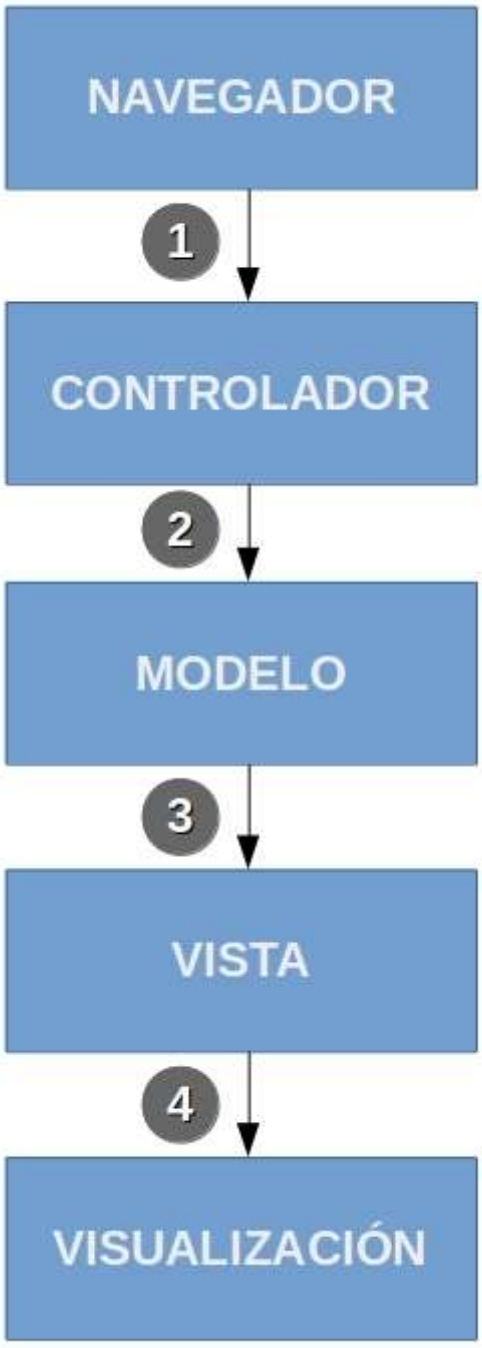
ASP soporta la arquitectura MVC. Los desarrolladores de ASP utilizan Visual Studio para crear sus aplicaciones.
Algunas de las características de ASP son:

- La ligereza de las aplicaciones lo cual no evita el poder desarrollar aplicaciones complejas.
- La posibilidad de añadir plugins con los cuales hacer la herramienta más eficiente y más adaptable a las necesidades del desarrollador.
- Soporta muchas funciones como en enrutamiento, la autorización y autenticación, enlazado de datos, páginas maestras, controles de usuarios, etc.

Diagrama de flujo del Modelo-Vista-Controlador

A continuación veremos cómo funciona el MVC de una forma gráfica:

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es



Paso 1. El navegador (chrome, firefox, safari, etc.) envía una petición a la aplicación MVC. Esta petición llega directamente al componente dedicado a gestionarla que es el controlador.

Paso 2. El controlador procesa la petición e interactúa con el modelo para resolverla. Tendrá que crear un modelo concreto instanciando objetos y otras variables.

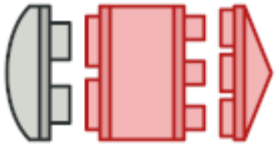
Paso 3. Este modelo se le pasa a la vista la cual procesa dicha petición.

Paso 4. La vista se encarga que el usuario reciba en el navegador la salida adecuada.

PATRONES ESTRUCTURALES

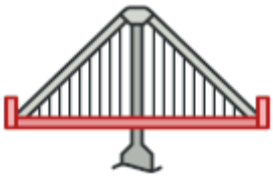
LOS ADAPTADORES O ADAPTER

Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.



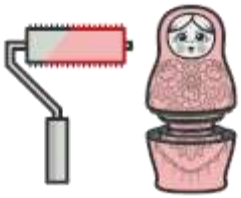
BRIDGE

Bridge es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.



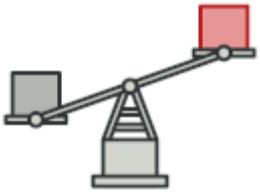
COMPOSITE

LOS DECORADORES O DECORATOR



FACADE

FLYWEIGHT



PROXY

Proxy es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

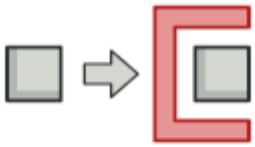
Puedes llevar a cabo una implementación diferida, es decir, crear este objeto sólo cuando sea realmente necesario. Todos los clientes del objeto tendrán que ejecutar algún código de inicialización diferida. Lamentablemente, esto seguramente generará una gran cantidad de código duplicado.

En un mundo ideal, querríamos meter este código directamente dentro de la clase de nuestro objeto, pero eso no siempre es posible. Por ejemplo, la clase puede ser parte de una biblioteca cerrada de un tercero.

El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

Pero, ¿cuál es la ventaja? Si necesitas ejecutar algo antes o después de la lógica primaria de la clase, el proxy te permite hacerlo sin cambiar esa clase. Ya que el proxy implementa la misma interfaz que la clase original, puede pasarse a cualquier cliente que espere un objeto de servicio real.

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es



PATRONES COMPORTAMIENTO

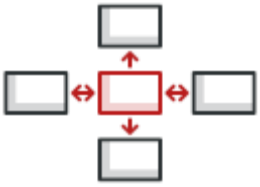
Los patrones de comportamiento son patrones que, como su nombre evidencia, tratan sobre la interacción entre clases y objetos, además de sus propios algoritmos.

Algunos muy utilizados son los siguientes:

- ITERATOR (ITERADOR)

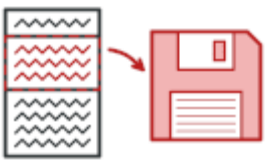
Permitirá al programador realizar recorridos por objetos de forma independiente al tipo de objeto que sea o como estén implementados.

- MEDIATOR (MEDIADOR)



Se define un objeto que realizará la mediación de la comunicación entre un conjunto de objetos.

- MEMENTO (RECUERDO)



Permite trasladar un conjunto de objetos o sistema a un estado anterior.

- TEMPLATE METHOD (PLANTILLA)

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

Se define el esqueleto de un algoritmo de tal manera que las subclases redefinirán ciertos pasos sin modificar su estructura.

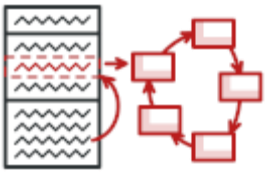
- VISITOR (VISITANTE)

Permitirá al programador definir nuevas operaciones sin modificar las clases sobre una jerarquía.

- COMMAND (ÓRDEN)

Permite encapsular una orden dentro de un objeto de tal manera que, cuando se ejecuta, no se necesita conocer su contenido.

- STATE (ESTADO)



Permite que un objeto modifique su comportamiento en el momento que cambie su estado interno.

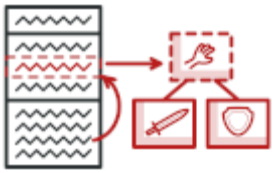
Caso concreto de implementación con un ejemplo. Desea implementarse una clase de tipo state para controlar el estado de un volcán. Cada vez que se cambie de estado, el monitor tiene que avisar de dicho cambio.

CHAIN OF RESPONSIBILITY

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

OBSERVER

STRATEGY



IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

PATRÓN DAO (DATA ACCESS OBJECT)

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 Bussines Objects u Objetos de negocios, de tal forma que el DAO encapsula toda la lógica de acceso de datos al resto de la aplicación.

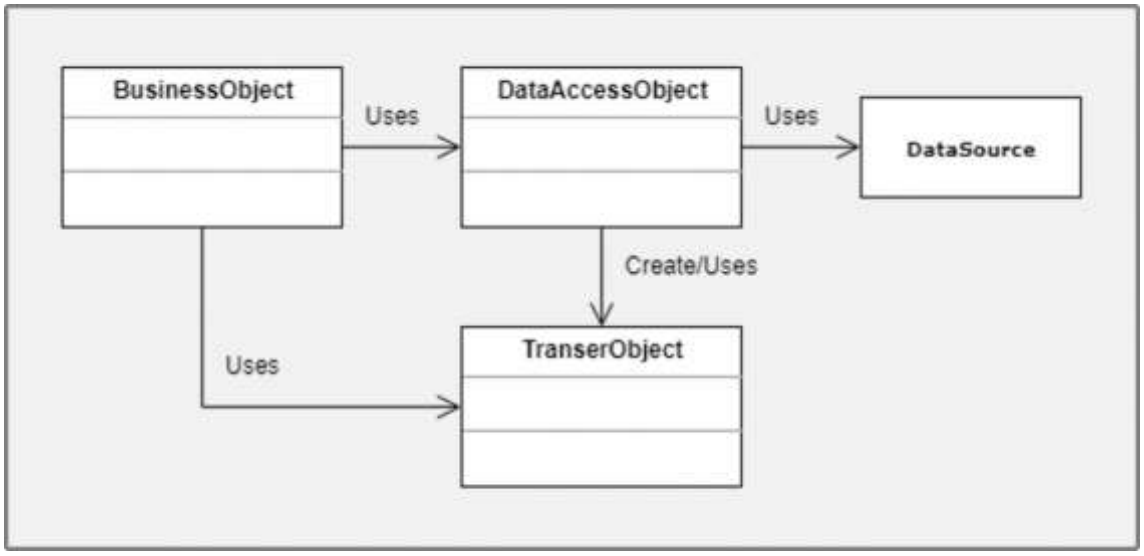
Problemática

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).

Solución

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.

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

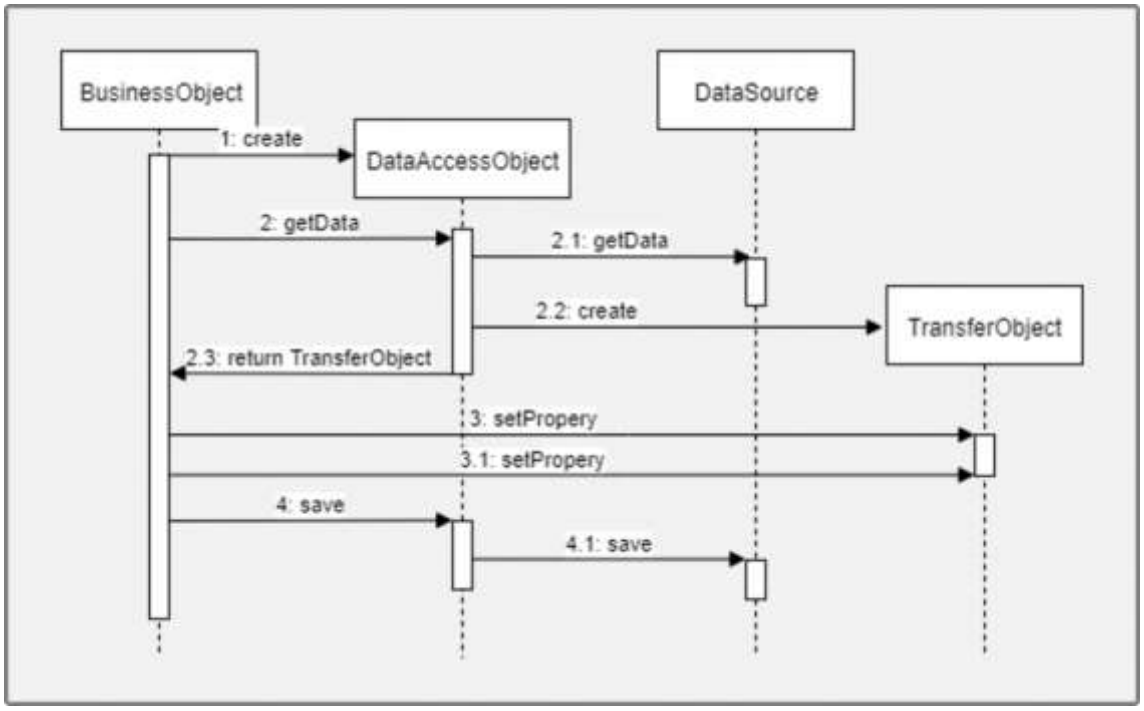


Los compones que conforman el patrón son:

- **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.

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es



El diagrama se interpreta de la siguiente manera:

1. El *BusinessObject* creo u obtiene una referencia al *DataAccessObject*.
2. 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* response con el *TransferObject* creado en los pasos anteriores.
3. El *BusinessObject* actualiza algún valor del *TransferObject*
    - Más actualizaciones
4. 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*.

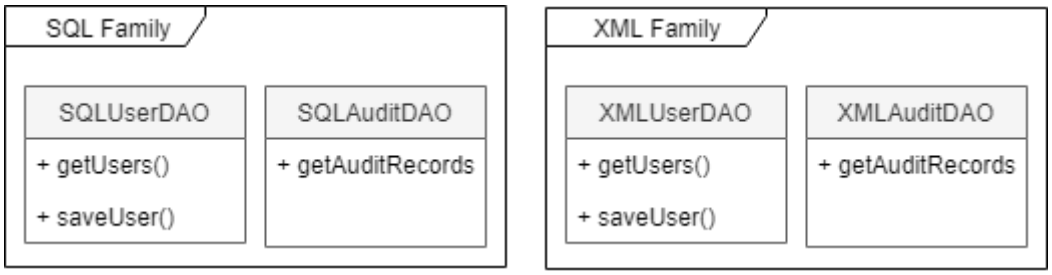
Hace unos días escribir sobre el patrón Data Transfer Object (DTO) por si quieres profundizar en el tema.

DAO y el patrón Abstract Factory

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

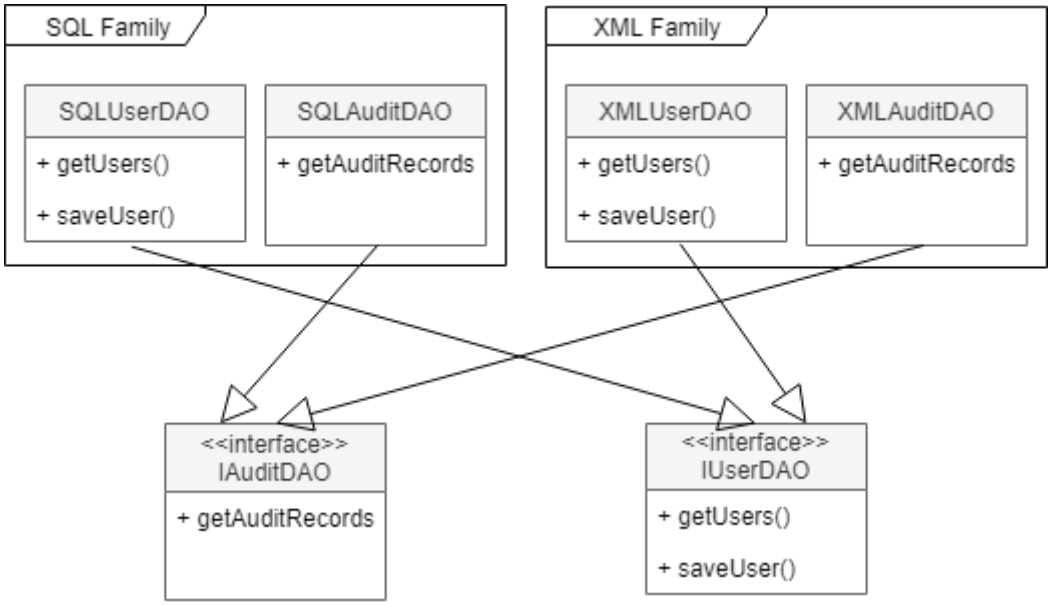
Mediante el patrón Abstract Factory podemos definir una serie de familias de clases que permitan conectarnos a las diferentes fuentes de datos. Para esto, examinaremos un sistema de autenticación de usuarios, el cual puede leer los usuarios en una base de datos o sobre un XML, adicional, el sistema generara registros de login que podrán ser utilizados para auditorias.

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



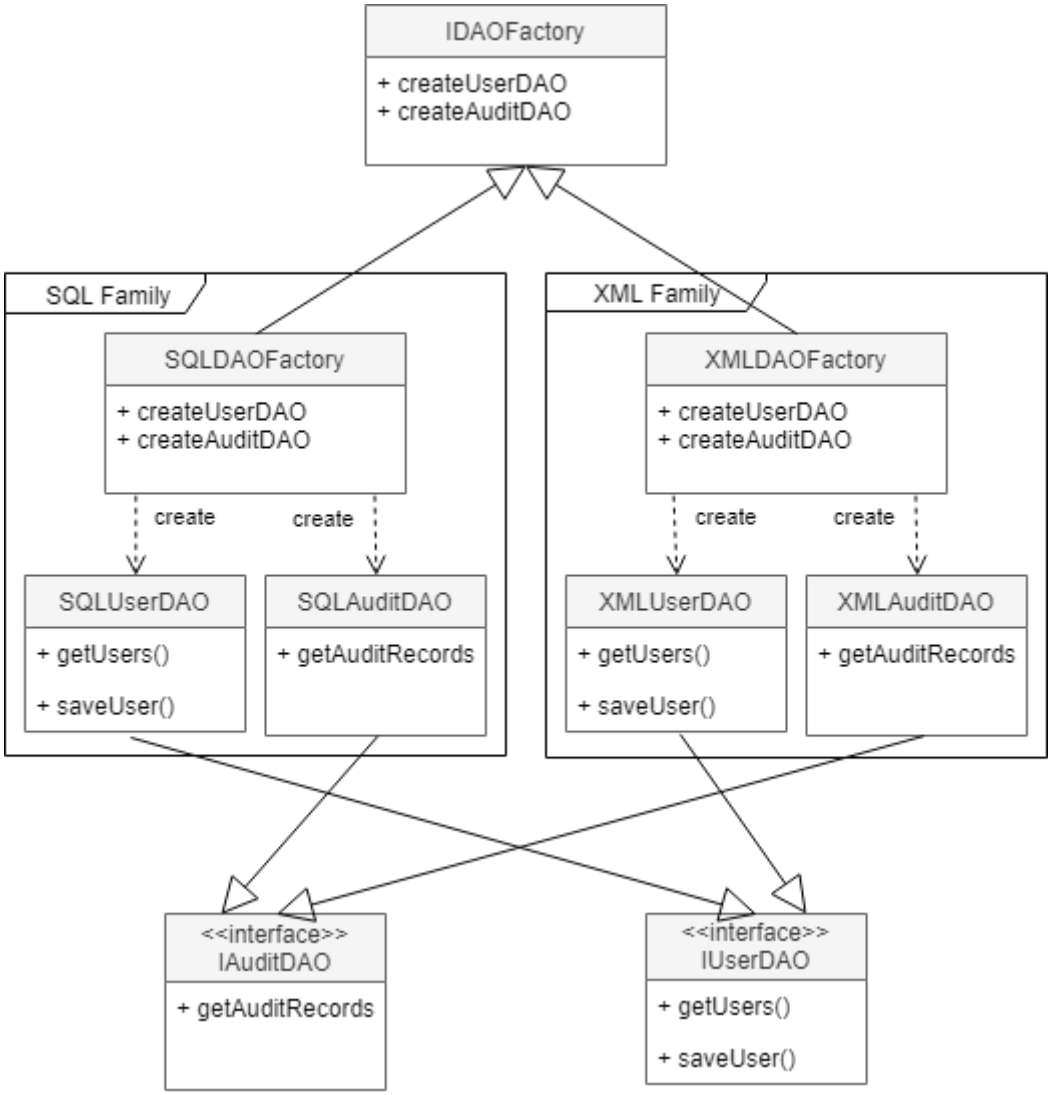
En la imagen anterior podemos apreciar dos familias de clases, con las cuales podemos obtener los Usuarios y los registros de auditoria, sin embargo, estas clases por separado no ayudan mucho, pues no implementan una misma interface que permita la variación entre ellas, por lo que el siguiente paso es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

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es



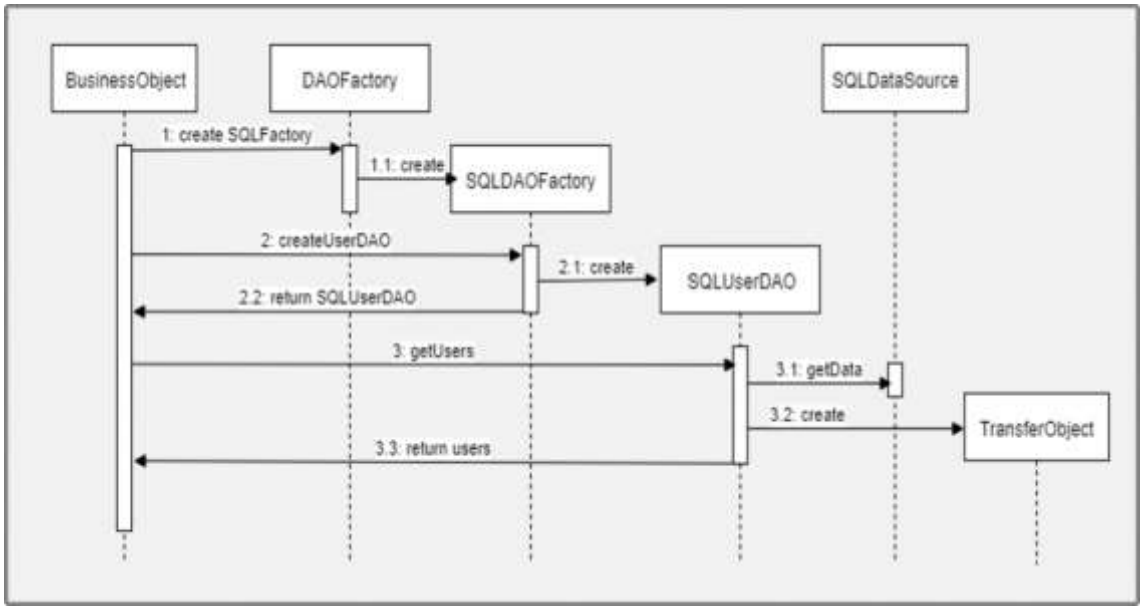
En este punto, los DAO ya implementan una interfaz común, lo que permite intercambiar la implementación sin afectar al Business Object. Sin embargo, ahora solo falta resolver la forma en que el Business Service obtendrá la familiar de interfaces, es por ello que deberemos crear un Factory para cada familia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

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es



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

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es



Analicemos como 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.

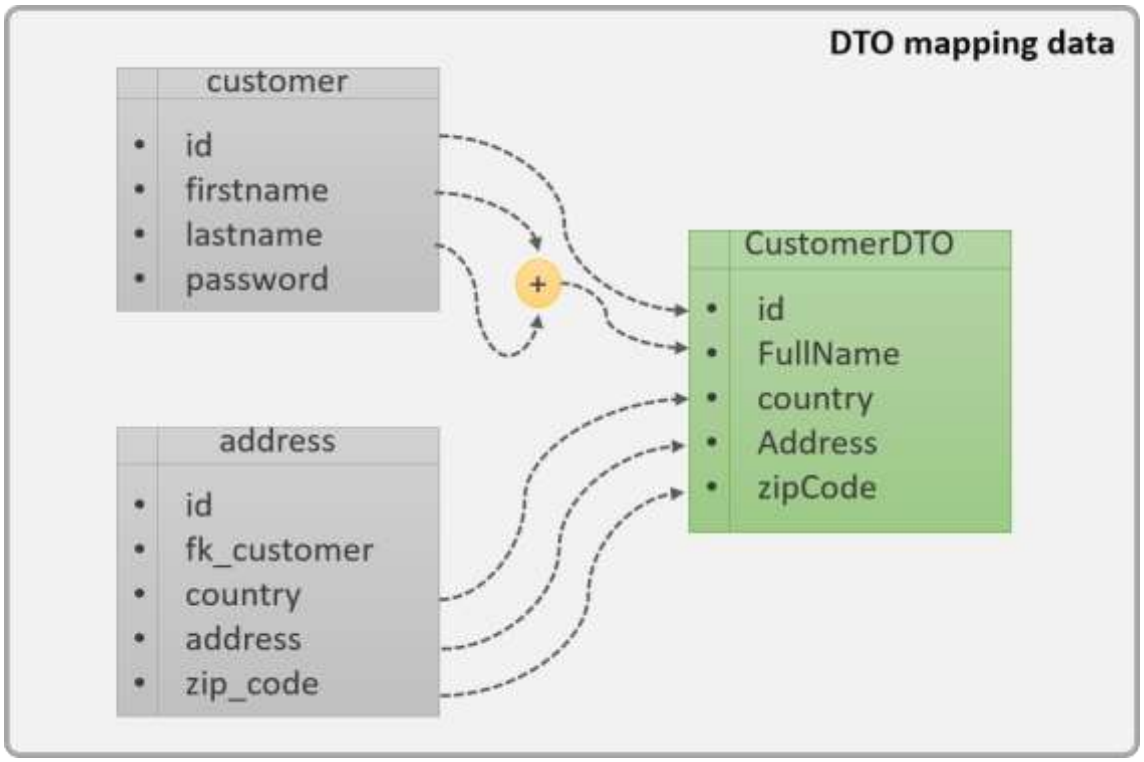
Conclusiones

El patrón DAO es sin lugar a duda, unos 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.

Data Transfer Object (DTO) – Patrón de diseño

Una de las problemáticas más comunes cuando desarrollamos aplicaciones, es diseñar la forma en que la información debe viajar desde la capa de servicios a las aplicaciones o capa de presentación, ya que muchas veces por desconocimiento o pereza, utilizamos las clases de entidades para retornar los datos, lo que ocasiona que retornemos más datos de los necesarios o incluso, tengamos que ir en más de una ocasión a la capa de servicios para recuperar los datos requeridos.

El patrón DTO tiene como finalidad de crear un objeto plano (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en **una sola invocación**, de tal forma que un DTO puede contener información de **múltiples fuentes** o tablas y concentrarlas en una única clase simple.



DTO mapping

En la imagen anterior podemos apreciar gráficamente como es que un DTO se conforma de una serie de atributos que puede o no, estar conformados por más de una fuente de datos. Para esto, el servidor obtiene la información de las tablas customer y address

(izquierda) y realiza un mapping con el DTO (derecha). Adicional, la información puede ser pasada de un lado intacta como es el caso del id , fullName , country , address y zipCode o ser una derivada de más de un campo, como es el caso del fullName , el cual es la unión del firstname y lastname .

Otra de las ventajas no tan claras en la imagen, es que nos permite omitir información que el usuario no requiere, como es el caso de password. No es solo que no lo requiere, sino que además podría ser una gran falla de seguridad está enviando los passwords, es por ello que en el DTO lo omitimos.

Características de un DTO

Si bien un DTO es simplemente un objeto plano, sí que tiene que cumplir algunas reglas para poder considerar que hemos creado un DTO correctamente implementado:

- **Solo lectura**: Dado que el objetivo de un DTO es utilizarlo como un objeto de transferencia entre el cliente y el servidor, es importante evitar tener operaciones de negocio o métodos que realicen cálculos sobre los datos, es por ello que solo deberemos de tener los métodos GET y SET de los respectivos atributos del DTO.
- **Serializable**: Es claro que, si los objetos tendrán que viajar por la red, deberán de poder ser serializables, pero no hablamos solamente de la clase en sí, sino que también todos los atributos que contenga el DTO deberán ser fácilmente serializables. Un error clásico en Java es, por ejemplo, crear atributos de tipo Date o Calendar para transmitir la fecha u hora, ya que estos no tienen una forma estándar para serializarse por ejemplo en Webservices o REST.

Entidades vs DTO

Un error muy frecuente entre programadores inexpertos es el hecho de utilizar las clases de Entidad para utilizarlos para la transmisión de datos entre el cliente y el servidor. Solo para entrar en contexto, las entidades son clases que representa al modelo de datos, o mapea directamente contra una tabla de la base de datos. Dicho esto, las entidades son clases que fueron diseñadas para mapear contra la base de datos, no para ser una vista para una pantalla o servicio determinado, lo que provoca que muchos de los campos no puedan ser serializables, no contengan todos los campos necesarios un servicio, ya sea que tengan de más o de menos.

El hecho de que las entidades no contengan todos los atributos necesarios o que no sean serializables trae otros problemas, como la necesidad de agregaras más atributos a las entidades con el único objetivo de poder cubrir los requerimientos de transferencia de datos, dejando de lado el verdadero propósito de la entidad, que es únicamente mapear contra la base de datos, lo que va llevando lentamente a ir creando una mezcla entre Entidad y DTO.

DTO en el mundo real

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

Para comprender mejor como es que se utilizan los DTO, vamos a realizar un análisis con un ejemplo de un servicio que recupera los datos todos los datos de los clientes. Para esto, veamos como quedarían las Entidades utilizando el API de JPA de Java.

Entidad Customer

```
package dtopattern;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="customers")
public class Customer {
	@Id
	@GeneratedValue(strategy= GenerationType.IDENTITY)
	private Long id;
	@Column(name="firstname")
	private String firstname;
	@Column(name="lastname")
	private String lastname;
	@Column(name="password")
	private String password;

	/** GET and SET */
}
```

Entidad Address

```
package dtopattern;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table(name="address")
public class Address {
	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private Long id;
	@ManyToMany()
	@JoinColumn(name="fk_customer")
	private Customer customer;
	@Column(name="country")
	private String country;
```

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es


```
        @Column(name="address")
        private String address;
        @Column(name="zipcode")
        private String zipCode;


        /** GET and SET */

}
```

Por otro lado, tenemos el DTO que contiene los datos del cliente (Customer) y su dirección (Address).

```
package dtopattern;

import java.io.Serializable;

public class CustomerDTO implements Serializable{

        private Long id;
        private String FullName;
        private String country;
        private String Address;
        private String zipCode;


        /** GET and SET */
}
```

Finalmente, veamos cómo quedaría un servicio que aproveche las ventajas del patrón DTO para transmitir los datos:

```
package dtopattern;

import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("customers")
public class CustomerService {

        @GET
        @PathParam("{customerId}")
        private Response findCustomer(@PathParam("customerId") Long
customerId) {
                Customer customer =
customerDAO.findCustomerById(customerId); //Entity
                Address address =
customerDAO.findAddressByCustomer(customerId); //Entity

                //Create dto
                CustomerDTO dto = new CustomerDTO();
```

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es


```
                dto.setAddress(address.getAddress());
                dto.setCountry(address.getCountry());
                dto.setZipCode(address.getZipCode());
                dto.setFullName(customer.getFirstname() + " " +
customer.getLastname());
                dto.setId(customer.getId());

                //Return DTO
                return Response.ok(dto,
MediaType.APPLICATION_JSON).build();
        }
}
```

El ejemplo que acabamos de ver, corresponde a una implementación de un servicio REST utilizando el API JAX-RS de Java, el cual indica que existe un servicio GET en la url */customers/{customerID}*, donde *{customerId}* corresponde al ID del cliente a buscar. En el servicio, realizamos la consulta del cliente y su dirección en dos pasos, para finalmente, mapear los datos de estas dos entidades en un simple DTO que será retornado.

Ponte a pensar, como le haríamos para obtener los datos del cliente y su dirección en una sola llamada al API REST sin ayuda de un DTO. Seguramente no podremos hacerlo en una sola llamada, si no que tendríamos que hacer dos consultas, una para recuperar el Cliente y otra para recuperar su dirección, o la otra alternativa y la peor de todas, sería modificar el Entidad Customer para agregar los campos que nos falte, a pesar de que la tabla no contenga esos campos.

Conclusiones

Como hemos podido demostrar, los DTO son un patrón muy efectivo para transmitir información entre un cliente y un servidor, pues permite crear estructuras de datos independientes de nuestro modelo de datos, lo que nos permite crear cuantas “vistas” sean necesarias de un conjunto de tablas u orígenes de datos. Además, nos permite controlar el formato, nombre y tipos de datos con los que transmitimos los datos para ajustarnos a un determinado requerimiento. Finalmente, si por alguna razón, el modelo de datos cambio (y con ello las entidades) el cliente no se afectará, pues seguirá recibiendo el mismo DTO.

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

Relaciones entre patrones

- Muchos diseños empiezan utilizando el Factory Method (menos complicado y más personalizable mediante las subclases) y evolucionan hacia
    - Abstract Factory, Prototype, o Builder (más flexibles, pero más complicados).
- Builder se enfoca en construir objetos complejos, paso a paso.
- Abstract Factory se especializa en crear familias de objetos relacionados.
    - *Abstract Factory* devuelve el producto inmediatamente, mientras que *Builder* te permite ejecutar algunos pasos adicionales de construcción antes de extraer el producto.
- Las clases del Abstract Factory a menudo se basan en un grupo de métodos de fábrica,
    - pero también puedes utilizar Prototype para escribir los métodos de estas clases.
- Abstract Factory puede servir como alternativa a Facade
    - cuando tan solo deseas esconder la forma en que se crean los objetos del subsistema a partir del código cliente.
- Puedes utilizar Abstract Factory junto a Bridge.
    - Este emparejamiento resulta útil cuando algunas abstracciones definidas por *Bridge* sólo pueden funcionar con implementaciones específicas. En este caso, *Abstract Factory* puede encapsular estas relaciones y esconder la complejidad al código cliente.
- Los patrones Abstract Factory, Builder y Prototype
    - pueden todos ellos implementarse como Singletons.

5 patrones de diseño asociados con conexiones de bases de datos

Etiquetas: Patrón de diseño MySQL Oracle SQL Server OS

Hace poco miré la información sobre patrones de diseño. De repente, pensé en 5 patrones de diseño a partir de la conexión de la base de datos. Luego los escribí y todos fueron alcanzables. Algunos métodos de implementación pueden ser prácticos. No tiene sentido en un entorno de producción. Aprendamos sobre los patrones de diseño.

La primera etapa es el modo de estrategia [b] [/ b]. Cuando nos conectamos a la base de datos, no es una base de datos. Por ejemplo, a veces es MySql, a veces es Oracle, y a veces cambia a SQL Server, lo que implica cambiar la base de datos. Podemos encapsular el algoritmo de conexión de la base de datos para que puedan ser reemplazados entre sí.

Primero definimos una interfaz de política para representar la conexión a la base de datos.

```
package strategy;

public interface Strategy {

        public void getConnDB();

}
```

Luego implementamos una clase de estrategia específica: la conexión de las tres bases de datos principales. Simplemente enfatizamos la implementación del patrón aquí. Por simplicidad, la operación específica de la conexión JDBC no está implementada. Lo mismo es cierto para lo siguiente. Mysql:

```
public class MysqlStrategy implements Strategy {
        public void getConnDB() {
                /*try {

        Class.forName("com.mysql.jdbc.Driver").newInstance();
                        String url =
"jdbc:mysql://localhost/myDB?user=root&password=123456&useUnicode=true
&characterEncoding=utf-8";
                        Connection connection =
DriverManager.getConnection(url);
                } catch (SQLException e) {
                        e.printStackTrace();
```

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es


```
                } catch (InstantiationException e) {
                        e.printStackTrace();
                } catch (IllegalAccessException e) {
                        e.printStackTrace();
                } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }*/
                System.out.println("connect MySQL");
        }
}
```

Oracle:

```
public class OracleStrategy implements Strategy {

        public void getConnDB(){

                System.out.println("connect  oracle");

        }

}
```

SQL Server:

```
public class SQLStrategy  implements Strategy{
        public void getConnDB(){
                        System.out.println("connect  SQL SERVER");
                }
        }
```

Escenarios de aplicación de estrategias para facilitar la selección dinámica de acciones específicas que se realizarán en tiempo de ejecución.

```
public class ClientContext {
        Strategy strategy;

        public ClientContext(Strategy strategy){

                this.strategy=strategy;
        }

        public void getConnDB(){
                strategy.getConnDB();
        }

}
```

Comencemos a probar:

```
public class StrategyTest {
        public static void main(String[] args) {
```

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es


```
                /**
                 * El modo de estrategia realiza la operación de
conexión a Oracle.
                 */
                ClientContext occ = new ClientContext(new
OracleStrategy());
                occ.getConnDB();
                /**
                 * El modo de estrategia realiza la operación de
conexión a Mysql.
                 */
                ClientContext mcc = new ClientContext(new
MysqlStrategy());
                mcc.getConnDB();
                /**
                 * El modo de estrategia implementa operaciones de
conexión a SQL Server.
                 */
                ClientContext scc = new ClientContext(new
SQLStrategy());
                scc.getConnDB();

        }

}
```

De esta manera, el algoritmo para cambiar dinámicamente las conexiones de la base de datos a través del modo de estrategia se completa básicamente. Si desea implementar operaciones en las bases de datos DB2, Sybase y PostgreSQL, solo necesita implementar la interfaz de la estrategia. Esto le permite extenderla arbitrariamente. Al mismo tiempo, está oculto para los clientes (clase StrategyTest) Los detalles de implementación de estrategias específicas (algoritmos) son completamente independientes entre sí. Logre completamente una alta cohesión y un bajo acoplamiento.

En este punto, si cambia repentinamente los requisitos, debe agregar algunas impresiones de registro antes de la conexión de la base de datos. De acuerdo con el enfoque tradicional, modificamos cada clase de implementación específica para agregar estas funciones, si hemos implementado muchas clases específicas antes. Esto no es diferente. No es una carga pequeña, también viola el principio [b] de apertura y cierre [/ b] (OCP).
Aquí puede pensar en el escenario de usar AOP para imprimir registros que se mencionan a menudo en el estudio de Spring AOP. AOP utiliza principalmente el modo proxy. Aquí también lo implementamos a través del modo proxy. Dado que la clase de estrategia abstracta es una interfaz, utilizamos la reflexión proporcionada en JAVA Proxy
[b] Modo agente [/ b] Implementación específica:

```
public class ProxyDB implements InvocationHandler {

        private Object target;

        public ProxyDB(Object target) {
                this.target = target;
        }
```

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es


```
        /**
         * Aquí está el método que queremos agregar para imprimir el
registro
         */
        public void printLog() {
                System.out.println ("---------------- Imprimir el
registro del punto de salida ----------");
        }

        /**
         * Procesador de negocios proxy
         */
        public Object invoke(Object proxy, Method method, Object[]
args)
                        throws Exception {
                printLog();
                return method.invoke(this.target, args);
        }

        public static void main(String[] args) {
                /**
                 *
                 * Aumente la impresión del registro antes de
conectarse a MySQL a través del modo proxy;
                 *
                 */
                MysqlStrategy ms = new MysqlStrategy();
                ProxyDB proxyCorps = new ProxyDB(ms);
                Strategy realObject = (Strategy)
Proxy.newProxyInstance(ms.getClass()
                                .getClassLoader(),
ms.getClass().getInterfaces(), proxyCorps);
                realObject.getConnDB();

                /**
                 * Agregar impresión de registro antes de conectarse a
Oracle a través del modo proxy;
                 *
                 */
                OracleStrategy os = new OracleStrategy();
                ProxyDB proxyCorps1 = new ProxyDB(os);
                Strategy realObject1 = (Strategy)
Proxy.newProxyInstance(os.getClass()
                                .getClassLoader(),
os.getClass().getInterfaces(), proxyCorps1);
                realObject1.getConnDB();

                /**
                 * Aumente la impresión del registro antes de
conectarse a SQL Server a través del modo proxy;
                 *
                 */
                SQLStrategy ss = new SQLStrategy();
                ProxyDB proxyCorps2 = new ProxyDB(ss);
                Strategy realObject2 = (Strategy)
Proxy.newProxyInstance(ss.getClass()
```

```
                                .getClassLoader(),
ss.getClass().getInterfaces(), proxyCorps2);
                realObject2.getConnDB();

        }

}
```

¿Es posible aumentar la función de registro solo a través de un proxy? En este momento, de repente pienso en un decorador, cuyo propósito original es agregar o extender funciones adicionales a la instancia de la clase. Puede cambiar dinámicamente el comportamiento de un método de objeto. También cumple con el Principios abiertos y cercanos en principios de diseño.
[b] Modo decorador [/ b]
(Decorator):

```
public class Decorator implements Strategy {

        private Strategy strategy;

        public Decorator(Strategy strategy) {
                this.strategy = strategy;
        }

        /**
         * Características adicionales,
         * Cambia dinámicamente el comportamiento del método del objeto
original a través del patrón decorador.
         */
        public void printLog() {
                System.out.println ("---------------- Imprima primero el
registro de puntos de salida ----------");
        }

        public void getConnDB() {
                printLog();
                strategy.getConnDB();
        }

        public static void main(String[] args) {

                /**
                 * Aumente la salida del registro antes de la conexión
de Oracle
                 */
                Decorator decorator = new Decorator(new
OracleStrategy());
                decorator.getConnDB();
                /**
                 * Aumente la salida del registro antes de la conexión
MySQL
                 */
                Decorator decorator1 = new Decorator(new
MysqlStrategy());
                decorator1.getConnDB();
```

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es


```
                /**
                 * Aumente la salida del registro antes de la conexión
MySQL
                 */
                Decorator decorator2 = new Decorator(new
SQLStrategy());
                decorator2.getConnDB();
        }
}
```

Ja, ja, aquí puede comparar la diferencia entre el modo proxy y el modo decorador.

A veces, en aras de la seguridad, no nos gusta exponer cada clase de conexión de base de datos específica al cliente. Reducir la dependencia del cliente en la clase de fondo concreta. Mejorar la independencia y la portabilidad del subsistema. Esto es lo que necesitamos para proporcionar un simple Interfaz, trato unificado con el mundo exterior En este momento, es hora de que nuestro modo de fachada salga al escenario.

[b] Modo de apariencia de fachada [/ b]:
Proporciona una interfaz unificada a un conjunto de interfaces en un subsistema. El patrón Fachada define una interfaz de nivel superior para hacer que el subsistema sea más fácil de usar. Al construir un sistema jerárquico, puede usar el patrón Fachada para definir la entrada de cada capa del sistema.Si las capas son interdependientes, puede limitarlas para comunicarse solo a través de la Fachada, lo que simplifica las capas y capas. Dependencias entre.
Al mismo tiempo, [b] La Ley de Dimiter (LCP) en los principios de diseño de software también se ha implementado. [/ B]

```
public class Facade {

        /**
         * Proporcionar una interfaz unificada a través de la operación
de conexión para tratar con el mundo exterior de manera unificada
         * Menor dependencia de clases de implementación concretas.
         *
         */
        public void getConn() {
                OracleStrategy os = new OracleStrategy();
                MysqlStrategy ms =  new MysqlStrategy();
                SQLStrategy ss =    new SQLStrategy();

                os.getConnDB();
                ms.getConnDB();
                ss.getConnDB();
        }

        public static void main(String[] args) {

                new Facade().getConn();
        }
}
```

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

De hecho, la conexión de la base de datos JDBC en JAVA también usa el modelo de fachada.

El último patrón que viene a la mente es el patrón de plantilla [b] [/ b]. No hay código para esto. Puede consultar el marco abstracto jdbc proporcionado por Spring. Su núcleo es la plantilla JDBC. Puede desviarse de la idea original, justo cuando Es una especie de pensamiento divergente.

Finalmente, resumámoslo brevemente. Principalmente implementa cinco modos: modo de estrategia, modo de agencia, modo de decorador, modo de fachada y modo de plantilla. Dos principios principales: principio abierto y cerrado y la ley de Dimit. En cuanto a los dos principios principales
Modo de estrategia: enfatiza principalmente la encapsulación y el cambio libre del algoritmo. Modo proxy: proporcione un proxy para que otros objetos controlen el acceso a este objeto. Patrón de decorador: agregue dinámicamente la funcionalidad de una clase para lograr una forma de extensión más flexible que la herencia.

Modo de fachada: el sistema proporciona una interfaz unificada y simple para el mundo exterior, lo que reduce el acoplamiento entre sistemas.

Patrón de plantilla: extraiga el comportamiento común del código para lograr el propósito de la reutilización.