

patrones de diseño- Angel Gabriel Ortega Corzo

tenemos 4 principales patrones a tocar durante este laboratorio, comenzaremos dando una contextualizacion y luego un ejemplo de la vida real donde yo considero que se podria aplicar.

singleton

en clase vimos que es un patron creacional o anti patron que garantiza que una clase tenga una unica instancia global de acceso, un ejemplo de la vida real que yo creo que se podria aplicar es un calendario, desde donde accedamos siempre va a tener un mismo valor, en aplicaciones de tareas siempre es una practica añadir estos componentes.



ejemplo de codigo

```
package project.singleton;

public class Calendario {
    private static Calendario instance = new Calendario("Miercoles");
    public String dia;
    public int año=2024;

    private Calendario(String valor){
```

```

        this.dia=valor;
    }

    // y aqui se puede acceder a la instancia de la clase
    public static Calendario getInstance(){
        return instance;
    }
}

```

y luego podemos acceder a la clase main para finalmente poder demostrar la instancia unica:

```

package project.singleton;

public class main {
    public static void main(String[] args) {
        System.out.println(Calendario.getInstance().dia);
        System.out.println(Calendario.getInstance().año);
        Calendario.getInstance().año=2025;
        System.out.println(Calendario.getInstance().año);
        Calendario.getInstance().dia="Jueves";
        System.out.println(Calendario.getInstance().dia);
    }
}

```

obtendremos el siguiente output



```

C:\Users\Usuario\.jdk\corretto-21.0.4\bin\java.exe "-javaagent:C:\Program Files\
Miercoles
2024
2025
Jueves

Process finished with exit code 0

```

como podemos ver podemos modificar pero la instancia unica de la clase que creamos lo cual cumple con los requerimientos de singleton esto podria dar solucion cuando usamos clases globales, configuraciones de versiones, como podria ser el pom.xml en maven o los .sln en .net o el package.json en node que solo tienen una instancia unica a las dependencia y por medio de ellas accedemos a las dependencias donde las necesitemos en cualquier parte de nuestro proyecto.

factory

este patron es creacional al igual que el singleton, este proporciona un interfaz para crear objetos en un super clase. Con otras palabras, el Factory Method define un método de una clase base que será la fábrica que se utilice para crear objetos, pero las subclases que hereden pueden proporcionar su

propia implementación para crear objetos específicos. Un ejemplo podría ser tal vez una marca de moda que ofece distintos accesorios pero ellos pueden estar destinados a distintas partes del cuerpo, ya sea cabeza, cuello, tronco. etc



Ejemplo Codigo:

```
package project.Factory;

public interface Acessories {
    public void use();
}
```

clases que implementen la interfaz.

```
package project.Factory;

public class Gloves implements Acessories {
    @Override
    public void use() {
        System.out.println("Using gloves youre protecting your hands lil nigga");
    }
}
```

```
package project.Factory;

public class Hat implements Acessories {
    @Override
    public void use() {
        System.out.println("You are wearing a hat now u cant get sunburned");
    }
}
```

luego creamos una clase abstracta para definir un metodo por defecto para crear las clases, claramente el metodo debe ser del tipo de las clases:

```
package project.Factory;

public abstract class AcesoriesFactory {

    public abstract Acessories create();

}
```

```
package project.Factory;

public class GlovesFactory extends AcesoriesFactory {

    @Override
    public Acessories create() {
        return new Gloves();
    }

}
```

```
package project.Factory;

public class HatFactory extends AcesoriesFactory {

    @Override
    public Acessories create() {
        return new Hat();
    }

}
```

y finalmente podremos crear cada objeto.

```
package project.Factory;

public class Main {

    public static void main(String[] args) {
        AcesoriesFactory acesoriesFactory = new GlovesFactory();
        Acessories glove = acesoriesFactory.create();
        glove.use();

        AcesoriesFactory hatFactory = new HatFactory();
        Acessories hat = hatFactory.create();
        hat.use();

    }

}
```

y obtendremos la salida esperada en el metodo use que sobreescribimos en cada clase.

```
C:\Users\Usuario\.jdk\corretto-21.0.4\bin\java.exe "-javaagent:C:\Program
Using gloves youre protecting your hands lil nigga
You are wearing a hat now u cant get sunburned
```

este patron es una herramienta para manejar la creacion de objetos en aplicaciones donde los tipos de objetos a crear pueden variar. En lugar de codificar explicitamente la creacion de cada objeto, delegamos esta responsabilidad a las subclases especializadas, lo que nos permite mantener un codigo mas flexible y facil de extender.

Facade

ya dejando de lado las patrones de diseño creacionales pasaremos a los patrones estructurales, pasaremos a este patron, normalmente lo usamos para usar una interfaz unificada, tratamos de simplificar la iteracion, podemos conseguir que los clientes no se compliquen entendiendo la complejidad interna. Un ejemplo podria ser un sistema de ventas o tienda, manejar, usuarios, compras y ventas, entregas. Pero como vimos debemos hacer esto lo mas sencillo posible para que el usuario tenga esto tan sencillo como realizar unos cuantos pasos sencillos.



ejemplo de codigo:

```
package project.Facade;

public class Inventory {

    public boolean isDisponible(String item) {
        System.out.println("Comprobacion de la disponibilidad para {" + item
+ "}");
        return true;
    }
}
```

```
package project.Facade;

public class Payout {

    public void pay(int price) {
        System.out.println("Procesando Pago de {" + price + "}");
    }
}
```

```
package project.Facade;

public class Delivery {

    public void deliver(String product) {
        System.out.println("el envio esta realizado con la entrega de {" +
product + "}");
    }
}
```

apenas tenemos nuestros distintos servicios se puede proceder a implementarlos en un unico sistema o app

```
package project.Facade;

public class FacadeRequest {
    private Payout payout;
    private Inventory inventory;
    private Delivery delivery ;

    public FacadeRequest() {
        this.payout = new Payout();
        this.inventory = new Inventory();
        this.delivery = new Delivery();
    }
}
```

```

    }

    public void dorequest(String item, int quantity) {
        if (inventory.isDisponible(item)) {
            System.out.println("Producto disponible, preparando pago");
            payout.pay(quantity);
            delivery.deliver(item);
        }
        else {
            System.out.println("No hay producto disponible");
        }
    }
}

```

y como podemos ver el metodo dorequest hace una agrupacion de todos los distintos metodos necesarios que necesitamos para que el usuario tenga todo en un solo lugar.

```

package project.Facade;

public class Main {

    public static void main(String[] args) {
        FacadeRequest facadeRequest = new FacadeRequest();
        facadeRequest.dorequest("arroz", 10);
    }
}

```

ejemplo de la salida.

```

C:\Users\Usuario\.jdk\corretto-21.0.4\bin\java.exe "-javaage
Comprobacion de la disponibilidad para {arroz}
Producto disponible, preparando pago
Procesando Pago de {10}
el envio esta realizado con la entrega de {arroz}

```

como ya vimos este patron estructural puede ser bastante usado para las grandes compañías monolito complejas que al dia de hoy no se centran en solo tener un servicio a ofrecer sino tratar de poder amplificar sus servicios en todos los indoles como estamos viendo a meta trantando de seguir la vision de su creador para aproximarse al metaverso, todo junto en un solo lugar, o como roblox que tiene millones de experiencias en una unica plataforma, algo que estoy mas que seguro que en algun punto debieron usar facade.

Strategy

este sin duda es el que considero mas cercano al chess debido a que este esta situado a poder aplicar distintos algoritmos y poder seleccionar alguno en tiempo de ejecucion. Un ejemplo podria ser las conversiones de descuentos a cada usuario.



entonces comenzamos por una interfaz:

```
package project.Strategy;

public interface Discount {
    public double getDiscount(double price);
}
```

y asignamos los descuentos que vamos a realizar implementando la interfaz:

```
package project.Strategy;

public class TenDiscount implements Discount {
    @Override
    public double getDiscount(double price) {
        return price * 0.90; // 10% of discount applied
    }
}
```

```
package project.Strategy;

public class FiveDollarsDiscount implements Discount {
    public double getDiscount(double price) {
        return price - 5.0; // 5$ of discount applied
    }
}
```

```
package project.Strategy;

public class WithoutDiscount implements Discount {
    @Override
    public double getDiscount(double price) {
        return price; // no discount applied xDDDD
    }
}
```



```
}  
}
```

luego creamos una clase que haga uso de un atributo descuento del tipo de la interfaz ya mencionada

```
package project.Strategy;  
  
public class Product {  
    private String name;  
    private double price;  
    private Discount discount;  
  
    public Product( Discount discount) {  
        this.discount = discount;  
    }  
  
    public double CalculateDiscount(double price) {  
        return discount.getDiscount(price);  
    }  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public double getPrice() {  
        return price;  
    }  
  
    public void setPrice(double price) {  
        this.price = price;  
    }  
}
```

y hacemos uso de la misma en el main y tendremos acceso a todos y cada uno de los descuentos y el metodo get discount se encargara de el resto.

```
package project.Strategy;  
  
public class Main {  
    public static void main(String[] args) {  
        Product product = new Product(new TenDiscount());  
        product.setName("pan");  
    }  
}
```

```

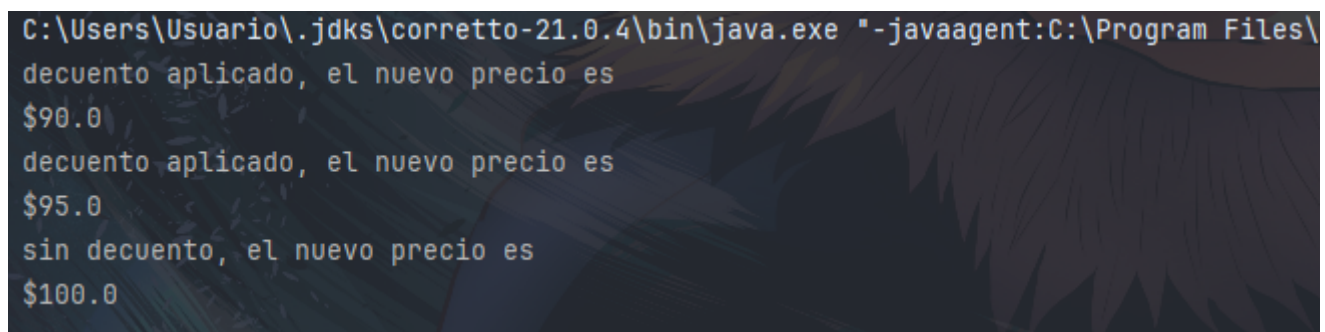
        product.setPrice(100);
        System.out.println("descuento aplicado, el nuevo precio es
\n$" + product.CalculateDiscount(100));

        product = new Product(new FiveDollarsDiscount());
        product.setName("pan");
        product.setPrice(100);
        System.out.println("descuento aplicado, el nuevo precio es
\n$" + product.CalculateDiscount(100));

        product = new Product(new WithoutDiscount());
        product.setName("pan");
        product.setPrice(100);
        System.out.println("sin descuento, el nuevo precio es
\n$" + product.CalculateDiscount(100));
    }
}

```

y obtendremos la siguiente salida:



```

C:\Users\Usuario\.jdk\corretto-21.0.4\bin\java.exe -javaagent:C:\Program Files\
descuento aplicado, el nuevo precio es
$90.0
descuento aplicado, el nuevo precio es
$95.0
sin descuento, el nuevo precio es
$100.0

```

esta interfaz ayuda a que podamos aplicar lo mismo con un distinto enfoque en este caso aplicando descuento, otro enfoque tambien podria ser en cuanto a las conversiones de moneda, las cuales son indispensables en plataformas de e-commerce y donde es vital para poder ampliar el enfoque de la expansion de la app, un patron sumamente util.

Redactado y presentado por:

angelgabrielortega