Patrones de diseño

Derian Herrera, Julio Mejia, Randi Paredes , Marco Garcia y Alisson Chino

October 12, 2020

I. Resumen

En el siguiente articulo observaremos algunos de los patrones de diseño usados. Los diseñadores expertos no solucionan los problemas desde sus principios sino que reutilizan soluciones que anteriormente funcionaron. Aqui se encuentran los patrones de diseño que resuelven problemas especificos y hacen el diseño flexible y reusable.

II. Abstract

In the following article we will observe some of the design patterns used. Expert designers do not solve problems from the beginning instead they reuse solutions that previously worked. Here are the design patterns that solve specific problems and they make the design flexible and reusable.

III. Introduccion

Los patrones de diseño es un tema importante en el desarrollo de software actual, lo que busca es ayudar a los desarrolladores de software a resolver problemas comunes creando un lenguaje comun para comunicar ideas y experiencias acerca de problemas y soluciones. El usar patrones de diseño ayuda a tener un software de calidad. Según su proposito los patrones se pueden clasificar en tres: De creacion, proceso de creacion de objetos. De estructura, tratan composicion de clases y/o objetos. De comportamiento, se caracterizan en la forma que interactuan y reparten responsabilidades a sus clases y objetos.

IV. Desarrollo

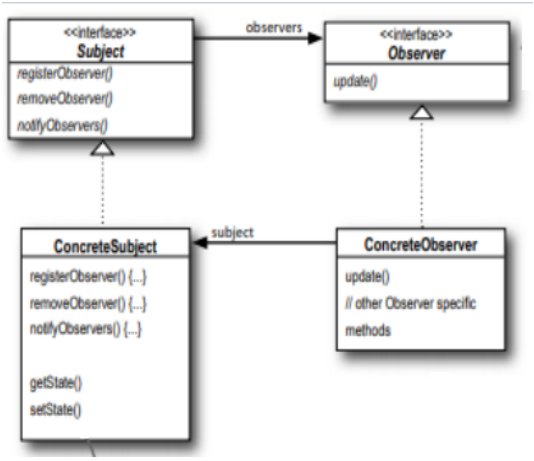
i. Patron de diseño observer

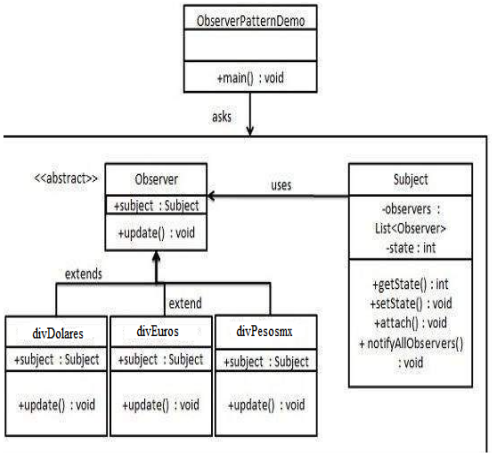
El patron de diseño observer es un patron comportamental,este patron define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que depende de él.

- **Descripcion**
  - Los objetos principales son "Subject" y "Observer"
  - La motivacion de este patron es la reutilizacion
  - Puede no haber relacion directa entre objetos
  - El tipo de interaccion es conocida como publicar-suscribir
  - El subject es publicador de notificaciones
  - Cualquier numero de observers puede suscribirse para recibir notificaiones

- **Componentes**
  Subject
  Cualquier numero de observers puede observar a subject
  Observer
  Define una interfaz de actualizacion para los objetos Observer que deben ser notificados de los cambios en el Subject
  Concrete Subject
  Almacena estados de interes para los objetos Concrete Observer.
  Envia notificaciones a sus observers cuando el estado cambia.
  Concrete Observer
  Mantiene referencia de objetos Concrete Subject.
  Almacena los estados que deben ser

consistentes con los subjects.
Implementa las observaciones del observer.





- **Ventajas**
  -Permite variar los sujetos y observadores indepedientemente.Se puede rehusar sujetos sin el rehuso de observadores y viceversa.
  -Permite agregar observadores sin modificar el sujeto o los observadores.
- **Desventajas**
  -No se especifica el receptor de una actualizacion.Se envia a todos los objetos interesados.
  -Actualizaciones inesperadas.Se podrian dar actualizaciones en casacada muy ineficientes.
- **Ejemplo**
  Crearemos un ejemplo en el cual al ingresar un monto en soles el programa nos dara la alerta segun el observador de cuando sera el cambio a la moneda de ese observador.

```java
package Divisas;


public abstract class Observador {

    protected Subject sujeto;
        public abstract void actualizar();


}
```

```java
package Divisas;

import java.util.ArrayList;
import java.util.List;


public class Subject {

    private List<Observador> observadores = new ArrayList<Observador>();
        private int estado;

        public int getEstado() {
                return estado;
        }

        public void setEstado(int estado) {
                this.estado = estado;
                notificarTodosObservadores();
        }

        public void agregar(Observador observador) {
                observadores.add(observador);
        }

        public void notificarTodosObservadores() {
                observadores.forEach(x -> x.actualizar());
        }
}
```

```java
package Divisas;


public class divEuros extends Observador{
    private double valorCambio = 0.24;

        public divEuros(Subject sujeto) {
                this.sujeto = sujeto;
                this.sujeto.agregar(this);
        }

        @Override
        public void actualizar() {
                System.out.println("EUROS: " + (sujeto.getEstado() * valorCambio));
        }
}
```

```java
package Divisas;


public class divPesosmx extends Observador{

private double valorCambio = 5.94;

        public divPesosmx(Subject sujeto) {
                this.sujeto = sujeto;
                this.sujeto.agregar(this);
        }

        @Override
        public void actualizar() {
                System.out.println("PESOS MEXICANOS: " + (sujeto.getEstado() * valorCambio));
        }

}
```

ii. Patron Facade

Proporcionar una interfaz unificada a un conjunto de interfaces en un subsistema. Facade define una interfaz de nivel superior que facilita el uso del subsistema.
Las fachadas se pueden utilizar no solo para crear una interfaz más simple en términos de llamadas a métodos, sino también para reducir el número de objetos que un cliente objeto debe tratar. Facade es un patrón de diseño de software que se usa comúnmente en la programación orientada a objetos. Facade es un objeto que sirve como una interfaz frontal que enmascara un código estructural o subyacente más complejo.
Los desarrolladores suelen utilizar el patrón de diseño Facade cuando un sistema es muy complejo o difícil de entender porque el sistema tiene muchas clases interdependientes o porque su código fuente no está disponible. Este patrón oculta las complejidades del sistema más grande y proporciona una interfaz más simple para el cliente. Por lo general, involucra una sola clase contenedora que contiene un conjunto de miembros requeridos por el cliente. Estos miembros acceden al sistema en nombre del cliente de fachada y ocultan los detalles de implementación.

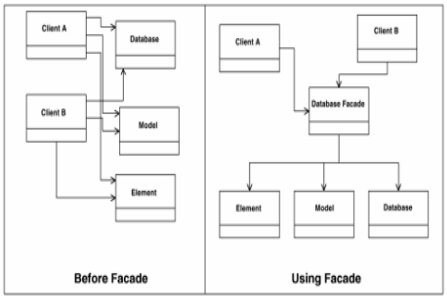
- **Caracteristicas**

  Intención Quiere simplificar cómo utilizar un sistema existente. Necesitas definir su propia interfaz.
  Problema Necesita utilizar solo un subconjunto de un sistema complejo. O necesita interactuar con el sistema de una manera particular.
  Solución Facade presenta una nueva interfaz para que la utilice el cliente del sistema existente.
  Participantes Facade presenta una nueva interfaz para que la utilice el cliente del sistema existente.
  Consecuencias Facade simplifica el uso del subsistema requerido. Sin embargo, debido a que la fachada no está completa, es posible que el cliente no disponga de determinadas funciones.
  Implementación Defina una nueva clase (o clases) que tenga la interfaz requerida.

- **Ventajas** Una de las ventajas de utilizar este patrón a la hora de controlar los permisos de los usuarios, es que cada usuario tiene un rol (administrador, invitado, usuario registrado, etc.) La principal ventaja del patrón Facade consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz / fachada, y los clientes pueden permanecer ajenos a ello. Además, y como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay tras dicha interfaz.
- **Desventajas** Como inconveniente, si se considera el caso de que varios clientes

necesiten acceder a subconjuntos diferentes de la función que proporcionan el sistema, podrían acabar usando sólo una pequeña parte de la fachada, por lo que sería conveniente utilizar varias fachadas más específicas en el lugar de una única global. Oculta a los clientes los componentes del subsistema, reduciendo así el número de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar.

- **Ejemplo**




```
package facade;


public class Arrancar {
    public Arrancar() {
    }
    // ------------------
    public void encenderContacto() {
        System.out.println("Introducimos la llave y le damos al encendido...");
    }
}
```

```
package facade;

public class ComprobarAsiento {
    public ComprobarAsiento() {
    }
    // ------------------
    public void comprobar() {
        System.out.println("Comprobamos y regulamos el asiento...");
    }
}
```

```
package facade;

public class ComprobarEspejos {
    public ComprobarEspejos() {
    }
    // ------------------
    public void comprobar() {
        System.out.println("Comprobamos y regulamos los espejos retrovisores...");
    }
}
```

```
package facade;

public class ComprobarLiquidos {
    public ComprobarLiquidos() {
    }
    // ------------------
    public void comprobar() {
        System.out.println("Comprobamos los liquidos de freno, agua, etc...");
    }
}
```

```
package facade;

public class Facade
{
    private ComprobarLiquidos liquidos;
    private ComprobarAsiento asiento;
    private ComprobarEspejos espejos;
    private Arrancar arrancar;
    // ------------------------------------
    public Facade()
    {
        this.liquidos  = new ComprobarLiquidos();
        this.asiento   = new ComprobarAsiento();
        this.espejos   = new ComprobarEspejos();
        this.arrancar = new Arrancar();
    }
    // ------------------------------------
    public void arrancarCoche()
    {
        liquidos.comprobar();
        asiento.comprobar();
        espejos.comprobar();
        arrancar.encenderContacto();
    }
}
```

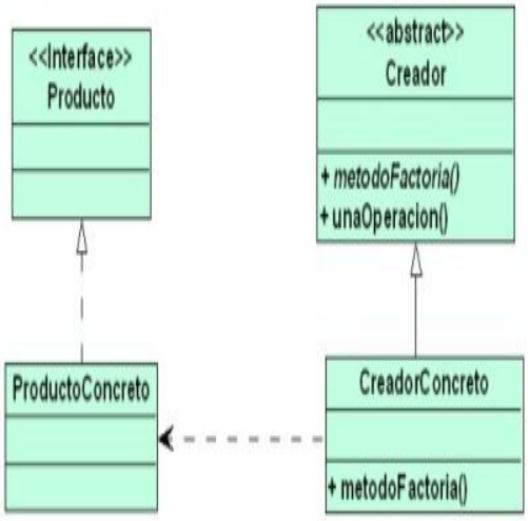
```
package facade;

public class Main {
//    public static void main(String[] args) {
//        ComprobarLiquidos liquidos = new ComprobarLiquidos();
//         liquidos.comprobar();
//        ComprobarAsiento asiento = new ComprobarAsiento();
//         asiento.comprobar();
//        ComprobarEspejos espejos = new ComprobarEspejos();
//         espejos.comprobar();
//        Arrancar arrancar = new Arrancar();
//         arrancar.encenderContacto();
//        System.out.println("\nProceso finalizado.");
//
//    }

    public static void main(String[] args)
    {
        Facade fachada = new Facade();
        fachada.arrancarCoche();
        System.out.println("\nProceso finalizado.");
    }

}
```

iii. Patrón de Diseño: FACTORY

El patrón de diseño Factory nos permite la creación de un subtipo determinado por medio de una clase de Factory, la cual oculta los detalles de creación del objeto. El objeto creado es enmascarado detrás de una interfaz común entre todos los objetos que pueden ser creado, con la finalidad de que estos pueden variar sin afectar la forma en que el cliente interactúa con ellos.

En un Factory es normal que pueda crear varios subtipos de una determinada interfaz y que todos los objetos concretos fabricados hagan una tarea similar, pero con detalles de implementación diferentes. La intención del Factory es tener una clase a la cual delegar la responsabilidad de la creación de los objetos, para que no sea el mismo programador el que decida que clase es la que instanciará, si no que delegará esta responsabilidad al Factory confiando en que este le regresará la clase adecuada para trabajar.

- **ESTRUCTURA**



Productos Abstractos son los que declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.

Productos Concretos son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto (silla/sofá) debe implementarse en todas las variantes dadas (victoriano/moderno).

Fábrica Abstracta en esa interfaz se declara un grupo de métodos para crear cada uno de los productos abstractos.

Fábricas Concretas implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea tan solo dichas variantes de los productos.

- **VENTAJAS**
  La ventaja de usar este patrón es que elimina la necesidad de instanciar los objetos de forma explicita que se utilizaran, también nos permitirá encapsular en las clases Factory la lógica de creación de los objetos, que incluso pueden ser mas complejas que realizar el (new). Es extensible ya que la arquitectura queda libre a desarrollos con nuevas clases que extienden a Factory y la familia de productos, también de esa manera responde al principio SOLID de open/close.

- **DESVENTAJAS**
  La desventaja es ya que Factory se usará para crear objetos que heredan una clase en común, puede que sea necesario mucho código repetitivo en cada una de las subclases. También al delegar funciones puede ser mas complejo encontrar en primera instancia la mecánica de funcionamiento del sistema. Al momento de querer añadir un nuevo producto, se necesita la implementación de la interfaz y todos sus métodos.

- **EJEMPLO**

- Creamos dos clases de tipo IArchivo así como la clase abstracta en la que se define el método de fabricación, y otra que hereda de ella y lo implementa.
- En el programa principal se crea una instancia de la clase que implementa el método de fabricación, el cual usaremos para crear y devolver los distintos tipos de objetos.

Main.java

```
package FactoryMethod1;

public class Main
{
    public static void main(String[] args)
    {
        CreadorAbstracto creator = new Creador();

        IArchivo audio = creator.crear( Creador.AUDIO );
        audio.reproducir();

        IArchivo video = creator.crear( Creador.VIDEO );
        video.reproducir();
    }
}
```

IArchivo.java

```
package FactoryMethod2;


public interface IArchivo

{

    public void reproducir();

}
```

Creador.java

```
package FactoryMethod2;

public class Creador
{
    public static final int AUDIO = 1;
    public static final int VIDEO = 2;

    // ----------------------------------------

    public Creador() {
    }

    // ----------------------------------------

    public static IArchivo getArchivo(int tipo)
    {
        IArchivo objeto;

        switch( tipo )
        {
            case AUDIO:
                objeto = new ArchivoAudio();
                break;
            case VIDEO:
                objeto = new ArchivoVideo();
                break;
            default:
                objeto = null;
        }

        return objeto;
    }
}
```

CreadorAbstracto.java

```
package FactoryMethod1;

public abstract class CreadorAbstracto
{
    public static final int AUDIO = 1;
    public static final int VIDEO = 2;

    // --------------------------------

    public abstract IArchivo crear(int tipo);
}
```

V. Conclusiones

- Conocer los patrones de diseño facilita la comprensión de los sistemas existentes.

- Las personas que aprenden programación orientada a objetos a menudo se quejan de que los sistemas con los que están trabajando usan la herencia de formas complicadas y que es difícil seguir el flujo de control. En gran parte, esto se debe a que no comprenden los patrones de diseño del sistema. Aprender estos patrones de diseño lo ayudará a comprender los sistemas orientados a objetos existentes.

- Estos patrones de diseño también pueden convertirlo en un mejor diseñador.

- Proporcionan soluciones a problemas comunes. Si trabaja con sistemas orientados a objetos el tiempo suficiente, probablemente aprenderá estos patrones de diseño por su cuenta. Aprender estos patrones ayudará a un novato a actuar más como un experto.

- Además, describir un sistema en términos de los patrones de diseño que utiliza hará que sea mucho más fácil de entender. De lo contrario, la gente tendrá

que realizar ingeniería inversa en el diseño para descubrir los patrones que utiliza

- Tener un vocabulario común significa que no tiene que describir todo el patrón de diseño; puede simplemente nombrarlo y esperar que su lector lo sepa. Un lector que no conozca los patrones tendrá que buscarlos al principio, pero eso sigue siendo más fácil que la ingeniería inversa.

VI. Recomendaciones

Como recomendacion usar un patron de diseño es algo que el programador debe elegir ya que cada patron se acomoda a soluciones a diferentes problemas , por eso existen distintos tipos de patrones los cuales el desarrollador puede usar y asi ahorrarse tiempo escencial para el desarrollo de software.

References

[1] Elisabeth Freeman, Kathy Sierra (2004). Head First design patterns, Sebastopol, CA: O'Reilly.

[2] Scott Millett, Nick Tune (2015). Patterns, Principles and Practices of Domain-Driven Design, vvvvvvvvv

[3] Bipin Joshi (2004). Beginning SOLID Principles and Design Patterns for ASP.NET Developers, Sebastopol, CA: O'Reilly.

[4] Holzner S., 2006. Design Patterns For Dummies. Hoboken, N.J.: Wiley.

[5] Oscar Belmonte , Carlos Granell, Maria del Carmen Erdozain.(2012). Desarrollo de proyectos informaticos con teconologia java. Universitat jaumen I