Universidad de las Fuerzas Armadas-ESPE

INTRODUCCIÓN A PATRONES DE DISEÑO

Programación Orientada a Objetos

INTEGRANTES:

·Eduardo Anibal Mejia Catillo

·Bryan Roger Camacho Ramirez

·Israel Fernando Portilla Santamaria

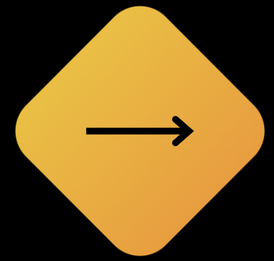
·Jenifer Andrea Castro Cuevas

·Julio Emerson Coro Pineida

·Alexis Sebastian Murminacho Cabascango

NRC: 1323

GRUPO: 9



INTRODUCCION

Los patrones de diseño son soluciones reutilizables para problemas recurrentes en el desarrollo de software. No se trata de fragmentos de código específicos, sino de enfoques estructurados que sirven como guías para mejorar la organización, mantenibilidad y escalabilidad del software.

Estos patrones funcionan como plantillas que pueden aplicarse en distintos contextos de la programación orientada a objetos, proporcionando un marco probado para abordar desafíos comunes en el diseño de sistemas. A continuación, se presentan algunos conceptos clave sobre los patrones de diseño y su importancia en la ingeniería de software.

¿Qué es un Design Pattern?

Los patrones de diseño (design patterns) son como planos de construcción prediseñados, que puedes modificar para adaptar mejor la resolución de un problema recurrente en tu código. Lo que diferencia los patrones de diseño de funciones y bibliotecas es que no puedes simplemente copiarlos directamente en tu programa, ya que no son un fragmento de código, sino un concepto que sirve como solución.

Por tanto, para implementar un patrón de diseño, debes seguir el concepto de los patrones elegidos (entre todos los existentes) y adaptarlo al problema que quieres resolver. Esto dependerá de las características del proyecto. De esta forma hemos creado una implementación que se ajusta exactamente a las necesidades de nuestra aplicación.

TIPOS DE PATRONES

1. Patrones Creacionales

Estos patrones se enfocan en la creación de objetos, proporcionando maneras de instanciarlos de manera más adecuada para la situación.

- Singleton: Asegura que una clase tenga una única instancia.
- Factory Method: Define una interfaz para crear objetos, delegando la instanciación a subclases.
- Abstract Factory: Provee una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
- Builder: Separa la construcción de un objeto complejo de su representación, permitiendo crear diferentes representaciones con el mismo proceso de construcción.
- Prototype: Permite crear nuevos objetos copiando una instancia existente.

2. Patrones Estructurales

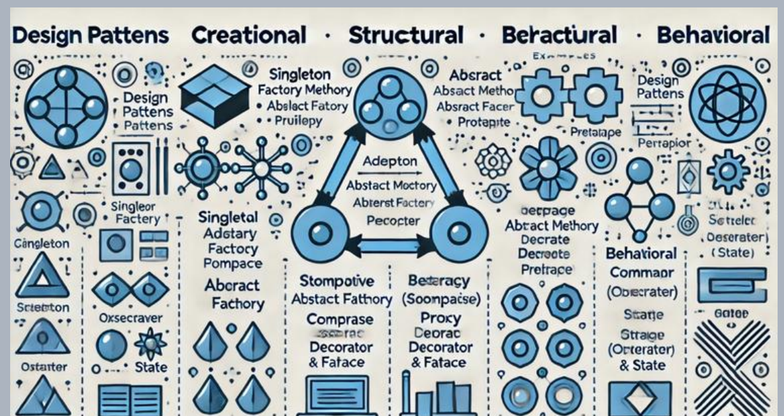
Estos patrones tratan sobre la composición de clases y objetos para formar estructuras más grandes.

- Adapter: Permite que clases con interfaces incompatibles trabajen juntas.
- Bridge: Separa una abstracción de su implementación, permitiendo cambiarlas independientemente.
- Composite: Permite tratar objetos individuales y composiciones de objetos de manera uniforme.
- Decorator: Añade responsabilidades adicionales a un objeto de manera dinámica.
- Facade: Proporciona una interfaz simplificada para un conjunto de interfaces de un subsistema.
- Flyweight: Reduce la sobrecarga de memoria compartiendo partes comunes de estado entre múltiples objetos.
- Proxy: Proporciona un sustituto o marcador de posición para otro objeto para controlar el acceso a él.

3. Patrones de Comportamiento

Estos patrones se centran en la interacción y comunicación entre objetos.

- Chain of Responsibility: Pasa una solicitud a lo largo de una cadena de manejadores.
- Command: Encapsula una solicitud como un objeto, permitiendo parametrizar clientes con solicitudes.
- Interpreter: Define una representación para la gramática de un lenguaje y un intérprete que usa la representación para interpretar oraciones en el lenguaje.
- Iterator: Proporciona un modo de acceder a los elementos de un objeto agregado secuencialmente sin exponer su representación subyacente.
- Mediator: Define un objeto que encapsula cómo los objetos interactúan.
- Memento: Permite capturar y externalizar el estado interno de un objeto para que luego se pueda restaurar.
- Observer: Define una dependencia uno-a-muchos entre objetos, de modo que cuando uno cambia de estado, todos sus dependientes son notificados.
- State: Permite a un objeto alterar su comportamiento cuando su estado interno cambia.
- Strategy: Define una familia de algoritmos, encapsulándolos y permitiéndoles intercambiarlos fácilmente.
- Template Method: Define el esqueleto de un algoritmo en un método, diferiendo algunos pasos a las subclases.
- Visitor: Permite definir nuevas operaciones en elementos de una estructura de objetos sin cambiar las clases de los elementos.



Puntos clave

1. PATRONES CREACIONALES: SE ENFOCAN EN LA INSTANCIACIÓN DE OBJETOS, ASEGURANDO QUE SEAN CREADOS DE MANERA CONTROLADA Y EFICIENTE. EJEMPLOS: SINGLETON, FACTORY METHOD, BUILDER.

2. PATRONES ESTRUCTURALES: FACILITAN LA COMPOSICIÓN DE CLASES Y OBJETOS, PROMOVIENDO ESTRUCTURAS FLEXIBLES Y REUTILIZABLES. EJEMPLOS: ADAPTER, COMPOSITE, DECORATOR

3. PATRONES DE COMPORTAMIENTO: SE CENTRAN EN LA INTERACCIÓN Y COMUNICACIÓN ENTRE OBJETOS PARA MEJORAR LA FLEXIBILIDAD DEL CÓDIGO. EJEMPLOS: OBSERVER, STRATEGY, COMMAND.

PATRONES DE ESTRUCTURA

Los patrones de estructura en Programación Orientada a Objetos (POO) son diseños que organizan las clases y objetos para formar estructuras más grandes y flexibles. Su objetivo es facilitar la reutilización del código, mejorar la modularidad y simplificar el mantenimiento del software.

PATRONES DE ESTRUCTURA MÁS COMUNES

*** FACADE**

Simplifica un sistema complejo con una interfaz única y fácil de usar.

*** ADAPTER**

Permite que dos clases incompatibles trabajen juntas.

*** DECORATOR**

Agrega funcionalidades a un objeto sin modificar su estructura original.

*** COMPOSITE**

Permite trabajar con grupos de objetos como si fueran uno solo.

*** BRIDGE**

Separa la abstracción de la implementación, facilitando cambios y mantenimiento.

*** PROXY**

Actúa como un intermediario para controlar el acceso a un objeto.

EJEMPLO FACADE (FACHADA)

Producción de Salchichas

SalchichaFacade.j... | FabricaSalchichas....

```java
// 1: Moler la carne
class Molino {
    void molerCarne() {
        System.out.println(" La carne ha sido molida.");
    }}
// 2: Mezclar los ingredientes
class Mezcladora {
    void mezclarIngredientes() {
        System.out.println(" Los ingredientes han sido mezclados.");
    }}
// 3: Embutir las salchichas
class Embutidora {
    void embutir() {
        System.out.println(" Las salchichas han sido embutidas.");
    }}
// 4: Cocinar las salchichas
class Horno {
    void cocinar() {
        System.out.println(" Las salchichas han sido cocinadas.");
    }}
// 5: Empacar las salchichas
class Empacadora {
    void empacar() {
        System.out.println(" Las salchichas han sido empacadas y están listas para la venta.");
    }}
// FACADE: Clase que simplifica el proceso de producción de salchichas
class SalchichaFacade {
    Molino molino;
    Mezcladora mezcladora;
    Embutidora embutidora;
    Horno horno;
    Empacadora empacadora;
// Constructor
    public SalchichaFacade() {
        molino = new Molino();
        mezcladora = new Mezcladora();
        embutidora = new Embutidora();
        horno = new Horno();
        empacadora = new Empacadora();
    }
// Método unificado para producir salchichas
    void producirSalchicha() {
        System.out.println("Iniciando la producción de salchichas...");
        molino.molerCarne();
        mezcladora.mezclarIngredientes();
        embutidora.embutir();
        horno.cocinar();
        empacadora.empacar();
        System.out.println("Ciclo de producción terminado");
    }}
```

SalchichaFacade.j... | FabricaSalchichas....

```java
// Clase principal
public class FabricaSalchichas {
    public static void main(String[] args) {
        // En lugar de llamar a cada clase por separado, usamos la fachada
        SalchichaFacade fabrica = new SalchichaFacade();
        fabrica.producirSalchicha();
    }
}
```

```
Iniciando la producción de salchichas...
La carne ha sido molida.
Los ingredientes han sido mezclados.
Las salchichas han sido embutidas.
Las salchichas han sido cocinadas.
Las salchichas han sido empacadas y están listas para la venta.
Ciclo de produccion terminado


..Program finished with exit code 0
Press ENTER to exit console.
```

PATRONES DE COMPORTAMIENTO

Los patrones de comportamiento están diseñados para gestionar la interacción entre objetos y la distribución de responsabilidades dentro de un sistema. Su objetivo es definir estructuras de comunicación y colaboración que sean tanto eficientes como adaptables, lo que permite desarrollar un software más modular y flexible ante modificaciones. Asimismo, estos patrones favorecen la escalabilidad y el mantenimiento del sistema, asegurando su evolución de manera sostenible a lo largo del tiempo.

PATRONES DE ESTRUCTURA MÁS COMUNES

-OBSERVER

EFINIR CÓMO LOS OBJETOS DEBEN REACCIONAR A CAMBIOS EN UN SUJETO (SUBJECT)

CLASE SUBJECT

LA CLASE SUBJECT PERTENECE AL PATRÓN OBSERVER Y ACTÚA COMO EL OBJETO OBSERVADO. MANTIENE UNA LISTA DE OBSERVADORES Y LES NOTIFICA CUANDO OCURRE UN CAMBIO EN SU ESTADO.

WheatherStation

NOTIFICANDO A LOS OBSERVADORES (WEATHEROBSERVER)

CLASE PRINCIPAL

EN EL MÉTODO MAIN DE "PRINCIPAL", SE CREAN INSTANCIAS DE WEATHERSTATION Y WEATHEROBSERVER. LUEGO, EL OBSERVADOR SE REGISTRA Y, AL ACTUALIZAR EL CLIMA CON SETWEATHER(), RECIBE UNA NOTIFICACIÓN Y MUESTRA EL CAMBIO.

Ejemplo

```java
public class SimpleSubject implements Subject {
    private List<Observer> observers;
    private int value = 0;

    public SimpleSubject() {
        observers = new ArrayList<Observer>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(value);
        }
    }

    public void setValue(int value) {
        this.value = value;
        notifyObservers();
    }
```

```java
public class SimpleObserver implements Observer {
    private int value;
    private Subject simpleSubject;

    public SimpleObserver(Subject simpleSubject) {
        this.simpleSubject = simpleSubject;
        simpleSubject.registerObserver(this);
    }

    public void update(int value) {
        this.value = value;
        display();
    }

    public void display() {
        System.out.println("Value: " + value);
    }
}
```

```java
public class Test {

    public static void main(String[] args) {
      SimpleSubject simpleSubject = new SimpleSubject();

      SimpleObserver simpleObserver = new SimpleObserver(simpleSubject);

      simpleSubject.setValue(80);
     }
}
```

Gracias