PATRONES DE DISEÑO

Josias Isaac Alvarenga Romero

Los patrones de diseño son soluciones reutilizables y probadas para problemas comunes en el desarrollo de software. Estos patrones no son código concreto, sino guías que ayudan a diseñar sistemas más flexibles, mantenibles y escalables. Se clasifican en tres categorías principales: creacionales, que se enfocan en la creación de objetos; estructurales, que tratan la composición de clases y objetos; y comportamentales, que se centran en la interacción y comunicación entre objetos.

Usar patrones de diseño facilita la comunicación entre desarrolladores y mejora la calidad del software. Dichos patrones se clasifican en:

PATRONES CREACIONALES:

Son el grupo de patrones de que se enfocan en el proceso de creación de objetos en un sistema de software. Su objetivo es abstraer el proceso de instanciación de objetos, lo que permite crear objetos de manera flexible y reutilizable. Esto es especialmente útil cuando un sistema necesita ser independiente de cómo se crean, componen y representan los objetos. Algunos de los principales patrones son:

Singleton:

Es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

- **Ejemplo de Uso:** Gestión de recursos compartidos, como conexiones de base de datos o configuraciones globales.

Factory Method:

Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

- **Ejemplo de Uso**: Creación de objetos que requieren procesos de inicialización complejos o diferentes tipos de objetos en función de ciertas condiciones.

Abstract Factory:

Es un patrón que proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Es un patrón de "fábrica de fábricas".

- **Ejemplo de Uso**: Sistemas que necesitan ser independientes de cómo se crean sus productos y necesitan crear familias de objetos relacionados.

Builder:

Es un patrón que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

- **Ejemplo de Uso**: Construcción de objetos complejos con múltiples partes o etapas, como un objeto que requiere varios pasos para configurarse completamente.

Prototype:

Nos permite crear nuevos objetos copiando una instancia existente, conocida como prototipo. Es útil cuando se necesita crear una copia de un objeto en lugar de instanciarlo directamente.

- **Ejemplo de Uso**: Sistemas que requieren crear nuevos objetos basados en un prototipo existente, especialmente cuando los costos de creación son altos.

```java
package refactoring_guru.singleton.example.non_thread_safe;

public final class Singleton {
    private static Singleton instance;
    public String value;

    private Singleton(String value) {
        // The following code emulates slow initialization.
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        this.value = value;
    }

    public static Singleton getInstance(String value) {
        if (instance == null) {
            instance = new Singleton(value);
        }
        return instance;
    }
}
```

```java
public class DemoSingleThread {
    public static void main(String[] args) {
        System.out.println("If you see the same value, then singleto
                "If you see different values, then 2 singletons were
                "RESULT:" + "\n");
        Singleton singleton = Singleton.getInstance("FOO");
        Singleton anotherSingleton = Singleton.getInstance("BAR");
        System.out.println(singleton.value);
        System.out.println(anotherSingleton.value);
    }
}
```

```
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (booo!!)

RESULT:

FOO
FOO
```

Ejemplo Patrón Singleton.

PATRONES ESTRUCTURALES:

Los patrones estructurales son el grupo de patrones de diseño que se centran en cómo las clases y los objetos se componen para formar estructuras más grandes y complejas. Su objetivo principal es facilitar la creación de relaciones entre entidades para garantizar que el sistema sea flexible. Algunos de ellos son:

Adapter:

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

- **Ejemplo de Uso**: Integrar clases que no fueron diseñadas para trabajar juntas o para usar una clase existente en un entorno que requiere una interfaz diferente.

Bridge:

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.

- **Ejemplo de Uso**: Cuando una abstracción y su implementación deben evolucionar independientemente, por ejemplo, en sistemas con múltiples plataformas.

Decorator:

Agrega responsabilidades adicionales a un objeto de manera dinámica sin modificar su estructura. Permite decorar un objeto con nuevas funcionalidades.

- **Ejemplo de Uso**: Cuando se necesita añadir funcionalidades a objetos individuales sin afectar a otros objetos de la misma clase.

```
// Digamos que tienes dos clases con interfaces compatibles:
// RoundHole (HoyoRedondo) y RoundPeg (PiezaRedonda).
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Devuelve el radio del agujero.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Devuelve el radio de la pieza.


// Pero hay una clase incompatible: SquarePeg (PiezaCuadrada).
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Devuelve la anchura de la pieza cuadrada.


// Una clase adaptadora te permite encajar piezas cuadradas en
// hoyos redondos. Extiende la clase RoundPeg para permitir a
// los objetos adaptadores actuar como piezas redondas.
class SquarePegAdapter extends RoundPeg is
    // En realidad, el adaptador contiene una instancia de la
    // clase SquarePeg.
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // El adaptador simula que es una pieza redonda con un
        // radio que pueda albergar la pieza cuadrada que el
        // adaptador envuelve.
        return peg.getWidth() * Math.sqrt(2) / 2
```

```
// En algún punto del código cliente.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // verdadero

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // esto no compila (tipos incompatibles)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // verdadero
hole.fits(large_sqpeg_adapter) // falso
```

Ejemplo Patrón Adapter.

PATRONES DE COMPORTAMIENTO:

Los patrones de comportamiento se enfocan en la forma en que los objetos interactúan y se comunican entre sí. Algunos de estos son:

Observer :

Define una dependencia uno a muchos entre objetos de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

- **Ejemplo de Uso**: Implementación de sistemas de eventos, como en interfaces gráficas o en patrones de suscripción/publicación.

Strategy :

Define una familia de algoritmos, encapsula cada uno y los hace intercambiables. Permite que un algoritmo sea seleccionado en tiempo de ejecución.

- **Ejemplo de Uso**: Situaciones en las que se necesita ejecutar diferentes algoritmos o comportamientos en función de una situación específica, como en motores de búsqueda o algoritmos de ordenación.

Command:

Encapsula una petición como un objeto, lo que permite parametrizar los clientes con diferentes solicitudes, hacer colas o registros de solicitudes y soportar operaciones que se pueden deshacer.

- **Ejemplo de Uso**: Implementación de sistemas de deshacer/rehacer, manejo de menús contextuales y comandos en interfaces gráficas.

```
// La interfaz estrategia declara operaciones comunes a todas
// las versiones soportadas de algún algoritmo. El contexto
// utiliza esta interfaz para invocar el algoritmo definido por
// las estrategias concretas.
interface Strategy is
    method execute(a, b)

// Las estrategias concretas implementan el algoritmo mientras
// siguen la interfaz estrategia base. La interfaz las hace
// intercambiables en el contexto.
class ConcreteStrategyAdd implements Strategy is
    method execute(a, b) is
        return a + b

class ConcreteStrategySubtract implements Strategy is
    method execute(a, b) is
        return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
        return a * b

// El contexto define la interfaz de interés para los clientes.
class Context is
    // El contexto mantiene una referencia a uno de los objetos
    // de estrategia. El contexto no conoce la clase concreta de
    // una estrategia. Debe trabajar con todas las estrategias a
    // través de la interfaz estrategia.
    private strategy: Strategy

    // Normalmente, el contexto acepta una estrategia a través
    // del constructor y también proporciona un setter
    // (modificador) para poder cambiar la estrategia durante el
    // tiempo de ejecución.
    method setStrategy(Strategy strategy) is
        this.strategy = strategy

    // El contexto delega parte del trabajo al objeto de
    // estrategia en lugar de implementar varias versiones del
    // algoritmo por su cuenta.
    method executeStrategy(int a, int b) is
        return strategy.execute(a, b)
```

```
// El código cliente elige una estrategia concreta y la pasa al
// contexto. El cliente debe conocer las diferencias entre
// estrategias para elegir la mejor opción.
class ExampleApplication is
    method main() is
        Create context object.

        Read first number.
        Read last number.
        Read the desired action from user input.

        if (action == addition) then
            context.setStrategy(new ConcreteStrategyAdd())

        if (action == subtraction) then
            context.setStrategy(new ConcreteStrategySubtract())

        if (action == multiplication) then
            context.setStrategy(new ConcreteStrategyMultiply())

        result = context.executeStrategy(First number, Second number)

        Print result.
```

Ejemplo Patrón Strategy.