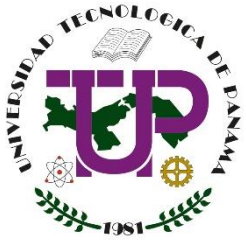


UNIVERSIDAD TECNOLÓGICA DE PANAMÁ

FACULTAD DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

DEPARTAMENTO DE PROGRAMACIÓN DE COMPUTADORAS

Actividad Inicial:

Presentación

Nombre de la Actividad:

Patrón Mediador y Patrón Observador

Integrantes:

Sué Acosta 8-1002-1727

Josty Crastz 3-752-2314

Grupo:

1LS242

Profesor:

Erick Agrazal

Asignatura:

Desarrollo de Software IX

Fecha de Entrega:

18 de abril de 2025

Introducción

En entorno de programación cada vez es más mantener o promover la creación de código o escritura de código mantenible, escalable y reutilizable. Los patrones de diseños surgen para ayudar a resolver estos problemas recurrentes en el desarrollo de programas o en la programación. Constituyéndose de un vocabulario más común entre desarrolladoras y siendo parte de un conjunto de buenas prácticas de programación.

Entre las variedades de patrones de diseño podemos mencionar, el patrón mediador y el patrón observador, están destacan como soluciones elegantes para gestionar las interacciones entre los componentes en un entorno de desarrollo. Ambos abordan aspectos fundamentales como la comunicación entre objetos, aunque de maneras diferentes.

En este trabajo estaremos profundizando mas acerca de cada uno de ellos, la mejor manera de utilizar, ejemplos de uso y la diferencia entre ellos. Comprender las fortalezas y limitaciones de estos patrones permite a los desarrolladores tomar decisiones informadas sobre su implementación, mejorando significativamente la calidad arquitectónica de sus soluciones software.

Índice

Contenido

¿Qué es un patrón de diseño?

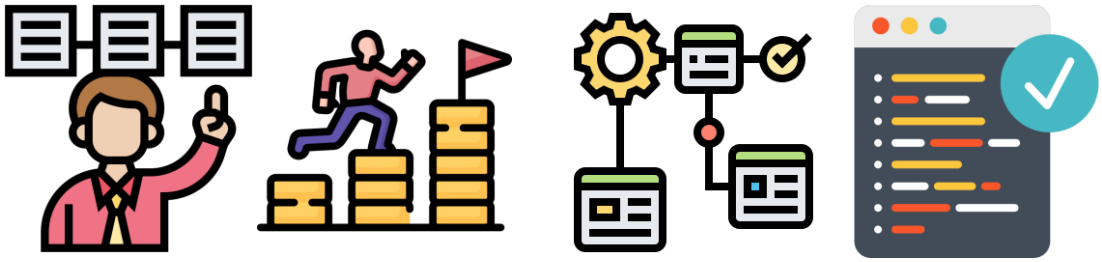
Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

¿En qué consiste el patrón?

La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos. Las secciones que suelen estar presentes en la descripción de un patrón:

- El **propósito** del patrón explica brevemente el problema y la solución.
- La **motivación** explica en más detalle el problema y la solución que brinda el patrón.
- La **estructura** de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
- El **ejemplo de código** en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.



Patrón Mediador

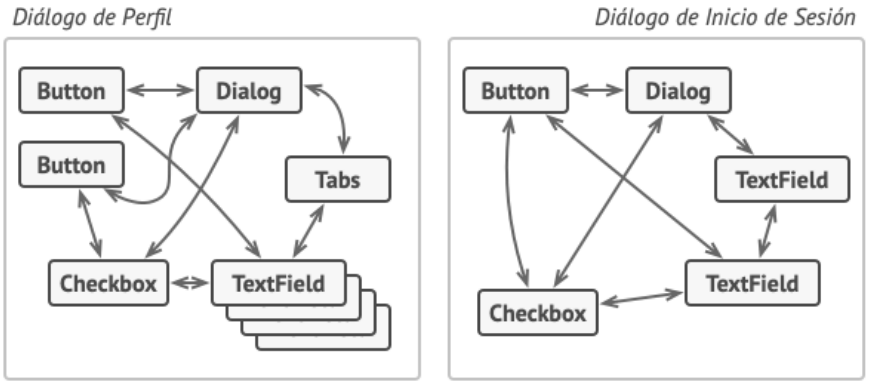
¿En qué consiste?

Es un patrón de diseño de comportamiento que permite reducir las dependencias caóticas entre objetos. Este patrón restringe la comunicación directa entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

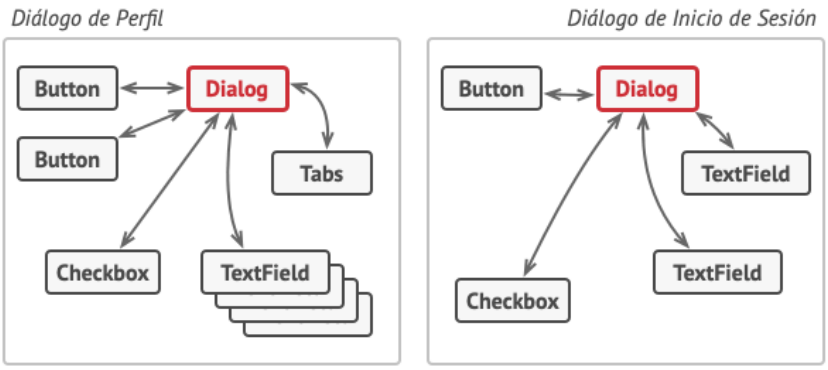
El patrón Mediator sugiere que detengas toda comunicación directa entre los componentes que quieres hacer independientes entre sí. En lugar de ello, estos componentes deberán colaborar indirectamente, invocando un objeto mediador especial que redireccione las llamadas a los componentes adecuados. Como resultado, los componentes dependen únicamente de una sola clase mediadora, en lugar de estar acoplados a decenas de sus colegas.

El patrón Mediator te permite encapsular una compleja red de relaciones entre varios objetos dentro de un único objeto mediador. Cuantas menos dependencias tenga una clase, más fácil es modificar, extender o reutilizar esa clase.

Ejemplo Sin Aplicar el Patrón Mediador:



Ejemplo de aplicación del Patrón Mediador:



¿Como funciona?

1. Definición del mediador: se crea un objeto que actuara como el punto central de comunicación.
2. Componentes que se comunican con el mediador: En lugar de comunicarse directamente los componentes envían mensajes o solicitudes al mediador.
3. El mediador maneja la comunicación: El mediador analiza los mensajes, iden6tifica que componentes deben involucrarse y luego realiza la acción necesaria, ya sea enviando mensajes, invocar métodos o coordinando acciones entre los componentes.
4. Desacoplamiento: al ser independiente de algunos componentes, los componentes se vuelven independientes y fáciles de modificar.

¿Cuándo se debe aplicar?

- Cuando resulte difícil cambiar algunas de las clases porque están fuertemente acopladas a un puñado de otras clases.
- Cuando no puedas reutilizar un componente en un programa diferente porque sea demasiado dependiente de otros componentes.
- Cuando te encuentres creando cientos de subclases de componente sólo para reutilizar un comportamiento básico en varios contextos.

Ejemplos de código y su aplicación:

Sin aplicación del patrón	Aplicación del patrón

// Comunicación directa entre componentes
class UserWithoutMediator {
  constructor(name) {
  this.name = name;
  this.colleagues = [];
  }
 	
  addColleague(colleague) {
  this.colleagues.push(colleague);
  }
 	
  sendMessage(message) {
  console.log(`${this.name} envía: ${message}`);
  // Cada usuario debe conocer a todos los demás y enviarles mensajes directamente
  this.colleagues.forEach(colleague => {
  colleague.receiveMessage(message, this.name);
  });
  }
  receiveMessage(message, senderName) {
  console.log(`${this.name} recibe de ${senderName}: ${message}`);
  }
}
	// El Mediador coordina la comunicación entre componentes
	class ChatMediator {
	  constructor () {
	  this.users = [];
	  }
	
	  addUser(user) {
	  this.users.push(user);
	  }
	
	  sendMessage(message, sender) {
	  // El mediador distribuye el mensaje a todos los usuarios excepto al remitente
	  this.users.forEach(user => {
	  if (user !== sender) {
	  user.receiveMessage(message, sender.name);
	  }
	  });
	  }
	}
	// Clase de Usuario que interactúa a través del mediador
	class User {

```
// Uso sin el patrón mediador
const alice = new UserWithoutMediator("Alice");
const bob = new UserWithoutMediator("Bob");
const charlie = new UserWithoutMediator("Charlie");

// Configuración manual de las relaciones entre todos los usuarios
alice.addColleague(bob);
alice.addColleague(charlie);

bob.addColleague(alice);
bob.addColleague(charlie);

charlie.addColleague(alice);
charlie.addColleague(bob);

// Envío de mensajes
alice.sendMessage("Hola a todos!");
bob.sendMessage("¡Hola Alice, hola Charlie!");
charlie.sendMessage("¡Qué bueno verlos!");
```

```
  constructor (name, mediator) {
    this.name = name;
    this.mediator = mediator;
    this.mediator.addUser(this);
  }

  sendMessage(message) {
    console.log(`${this.name} envía: ${message}`);
    this.mediator.sendMessage(message, this);
  }

  receiveMessage(message, senderName) {
    console.log(`${this.name} recibe de ${senderName}: ${message}`);
  }
}

// Uso del patrón mediador
const chatRoom = new ChatMediator();

const alice = new User("Alice", chatRoom);
const bob = new User("Bob", chatRoom);
const charlie = new User("Charlie", chatRoom);

alice.sendMessage("Hola a todos!");
bob.sendMessage("¡Hola Alice, hola Charlie!");
charlie.sendMessage("¡Qué bueno verlos!");
```

Comparación de las diferencias principales

Con el patrón Mediador:

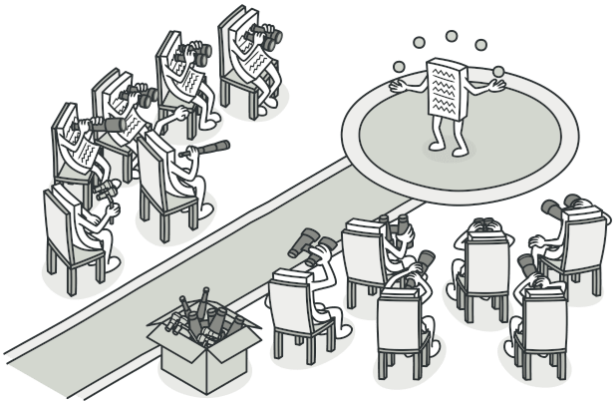
- Los componentes (usuarios) solo conocen al mediador, no entre sí
- La comunicación está centralizada a través del mediador
- Es más fácil añadir nuevos componentes sin modificar el código existente
- Las relaciones entre componentes están desacopladas
- Los componentes no necesitan tener código para gestionar a otros componentes

Sin el patrón Mediador:

- Cada componente debe conocer a todos los demás componentes
- La comunicación es directa entre componentes
- Añadir nuevos componentes requiere actualizar las referencias en todos los existentes
- Hay un alto acoplamiento entre componentes
- El código para gestionar las relaciones está duplicado en cada componente

Patrón Observador

Es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



¿Para qué se usa este patrón?

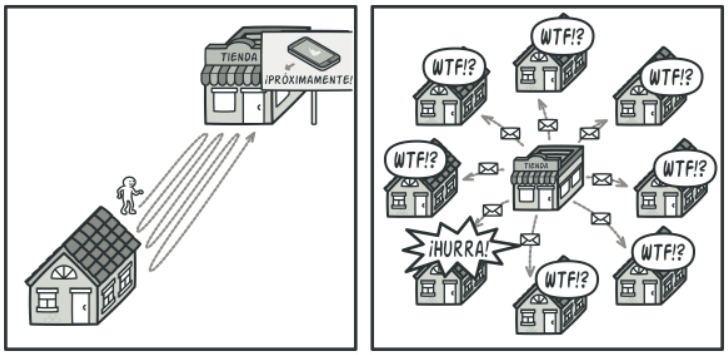
El patrón Observer es un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora.

Ejemplo de funcionalidad

Imagina que tienes dos tipos de objetos: un objeto `Cliente` y un objeto `Tienda`. El cliente está muy interesado en una marca particular de producto (digamos, un nuevo modelo de iPhone) que estará disponible en la tienda muy pronto.

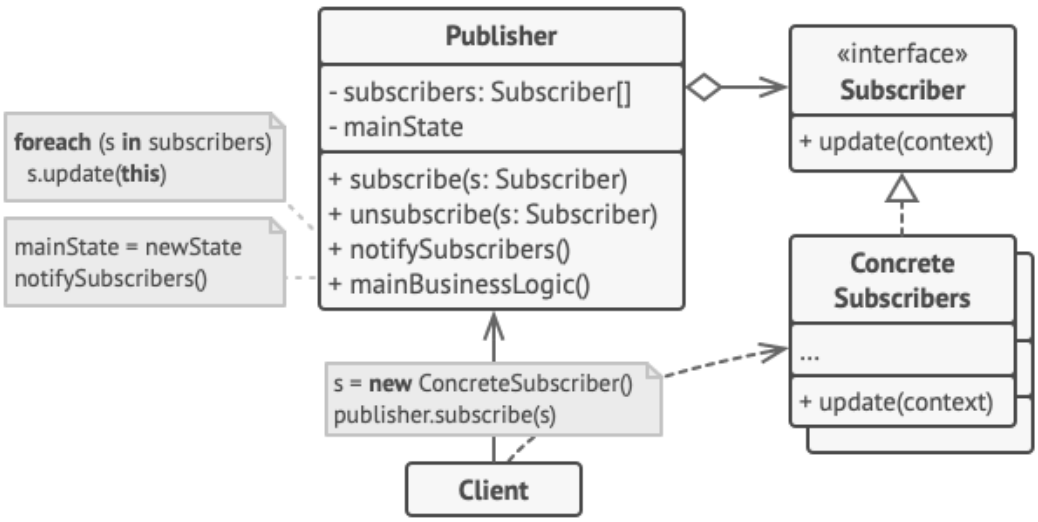
El cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero, mientras el producto está en camino, la mayoría de estos viajes serán en vano.

Por otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados en los nuevos productos.



¿Como se Estructura este Patrón?

a) El Notificador envía eventos de interés a otros objetos. Esos eventos ocurren cuando el notificador cambia su estado o ejecuta algunos comportamientos.
b) Cuando sucede un nuevo evento, el notificador recorre la lista de suscripción e invoca el método de notificación declarado en la interfaz suscriptora en cada objeto suscriptor.
c) Los Suscriptores Concretos realizan algunas acciones en respuesta a las notificaciones emitidas por el notificador.
d) Normalmente, los suscriptores necesitan cierta información contextual para manejar correctamente la actualización. Por este motivo, a menudo los notificadores pasan cierta información de contexto como argumentos del método de notificación.
e) El Cliente crea objetos tipo notificador y suscriptor por separado y después registra a los suscriptores para las actualizaciones del notificador.



Ejemplo de código de JavaScript

Código Sin Aplicar:	Código con patrón aplicado:

public class SistemaTemperatura {
 	
private double temperatura;
 	
private PanelControl panel;
 	
private ServicioAlerta servicio;
 	
private RegistroTemperatura registro;
 	
public SistemaTemperatura() {
 	
this.temperatura = 25.0;
 	
this.panel = new PanelControl();
 	
this.servicio = new ServicioAlerta();
 	
this.registro = new RegistroTemperatura();}
 	
public void setTemperatura(double nuevaTemperatura) {
 	
this.temperatura = nuevaTemperatura;
 	
panel.actualizarTemperatura(temperatura);
 	
servicio.verificarTemperatura(temperatura);
 	
registro.almacenarTemperatura(temperatura);}}
 	
class PanelControl {
 	
public void actualizarTemperatura(double temperatura) {
	interface Observado {
	
	void agregarObservador(Observador o);
	
	void eliminarObservador(Observador o);
	
	void notificarObservadores();}
	
	interface Observador {
	
	void actualizar (double temperatura);}
	
	public class SistemaTemperatura implements Observado {
	
	private double temperatura;
	
	private List<Observador> observadores;
	
	public SistemaTemperatura() {
	
	this.temperatura = 25.0;
	
	this.observadores = new ArrayList<>();}
	
	public void setTemperatura(double nuevaTemperatura) {
	
	this.temperatura = nuevaTemperatura;
	
	notificarObservadores();}
	
	public void agregarObservador(Observador o) {
	
	observadores.add(o);}
	
	public void eliminarObservador(Observador o) {
	
	observadores.remove(o);}
	
	public void notificarObservadores() {

```
        System.out.println("Panel de control: Temperatura actualizada a " + temperatura + "°C");}}



class ServicioAlerta {

    public void verificarTemperatura(double temperatura) {

        if (temperatura > 30) {

            System.out.println("ALERTA: Temperatura elevada detectada: " + temperatura + "°C");}}}



class RegistroTemperatura {

    public void almacenarTemperatura(double temperatura) {

        System.out.println("Registro: Guardando temperatura: " + temperatura + "°C");}}



public class Cliente {

    public static void main(String[] args) {

        SistemaTemperatura sistema = new SistemaTemperatura();

        sistema.setTemperatura(32.5);}}
```

```
        for (Observador observador : observadores) {

            observador.actualizar(temperatura); }}}

class PanelControl implements Observador {

    public void actualizar (double temperatura) {

        System.out.println("Panel de control: Temperatura actualizada a " + temperatura + "°C");}}



class ServicioAlerta implements Observador {

    public void actualizar (double temperatura) {

        if (temperatura > 30) {

            System.out.println("ALERTA: Temperatura elevada detectada: " + temperatura + "°C");}}}



class RegistroTemperatura implements Observador {

    @Override

    public void actualizar (double temperatura) {

        System.out.println("Registro: Guardando temperatura: " + temperatura + "°C");}}

public class Cliente {

    public static void main(String[] args) {

        SistemaTemperatura sistema = new SistemaTemperatura();

        PanelControl panel = new PanelControl();
```

	ServicioAlerta servicio = new ServicioAlerta();
	
	RegistroTemperatura registro = new RegistroTemperatura();
	
	sistema.agregarObservador(panel);
	
	sistema.agregarObservador(servicio);
	
	sistema.agregarObservador(registro);
	
	sistema.setTemperatura(32.5)
	
	sistema.eliminarObservador(registro);
	
	System.out.println("\nDespués de eliminar el observador de registro:");
	
	sistema.setTemperatura(33.8); }}

Comparación entre ambas implementaciones:

1. **Sin el patrón Observador**:
   - El Sistema Temperatura tiene referencias directas a todas sus dependencias
   - Cada vez que cambia la temperatura, se deben llamar explícitamente a todos los métodos de actualización
   - Alta dependencia y acoplamiento entre clases
   - Difícil de mantener si queremos agregar o eliminar componentes que necesiten conocer los cambios
2. **Con el patrón Observador**:
   - Se establece una relación uno-a-muchos entre el objeto observable y sus observadores
   - El objeto observable no necesita conocer detalles de sus observadores
   - Permite añadir/quitar observadores dinámicamente en tiempo de ejecución
   - Facilita cumplir con el principio de diseño "abierto/cerrado": el sistema está abierto a extensión, pero cerrado a modificación

Diferencias entre el patrón mediador y el patrón observador

Diferencias	Patrón Mediador	Patrón Observador
Método de Comunicación	La comunicación es indirecta a través de un intermediario	La comunicación es directa desde el sujeto a sus observadores.
Propósito Final	La gestión centralizada de interacciones complejas	La notificación de cambio s de estado.
La relación entre objetos	La Relación muchos-a-uno-a-muchos en este caso (muchos se comunican a través de un mediador)	La relación uno-a-muchos en este caso (una persona, muchos observadores)
Control de flujo	Cualquiera persona, componente u objeto puede iniciar la comunicación a través de un mediador.	La persona o sujeto inicia la comunicación.
Utilidad	Es útil cuando hay numerosas interacciones complejas entre componentes.	Se utilizan donde un cambio en un objeto debe reflejarse entre otros objetos sin conocer detalles específicos de ellos.
Estructura	Un mediador central y múltiples colegas, componentes que se comunican a través de él.	Es un objeto observable y múltiples observadores.
Caso de uso	Sistemas de control de tráfico aéreo, salas de chat, interfaces de usuario complejas.	Implementación de eventos, interfaces gráficas, sistemas de notificaciones.

Conclusión

Los patrones de diseños son principalmente soluciones probadas y reutilizables a problemas que nos ocurren en entornos de programación, ofreciendo una estandarización que mejora el mantenimiento y escalabilidad de los sistemas. Entre estos, el patrón mediador y el patrón observador representan dos enfoques fundamentales para gestionar las interacciones entre componentes de un sistema.

El patrón mediador establece a un objeto central que establece la comunicación entre múltiples objetos, reduciendo así el mensaje directo entre ellos. Esta centralización simplifica las interacciones que son complejas, facilitando la modificación de comportamiento sin alterar los demás componentes.

Por otra parte, el Patrón observador implementa un mecanismo donde los objetos observadores reciben notificaciones automáticas cuando cambia el estado del objeto.

Ambos patrones, aunque diferentes en su implementación, estructura y su caso de uso, comparten el objetivo fundamental de los patrones de diseño: promover código modular, reutilizable y entendible. Su aplicación adecuada requiere una compresión de sus beneficios como de sus limitaciones, para integrarlos de la manera adecuada.

Conclusiones Individuales

Sué Acosta

Considero que los patrones de diseño, son importantes ya que nos ayuda a mejorar más que la estética, mejora la implementación de código, el entendimiento del código y la comunicación entre los mismo objetos y clases del código. Es importante aprender implementación de estos patrones de diseño para su entendimiento en otros códigos y para la implementación de las mismos en nuestro propio código para tener una buena practica en programación.

Josty Crastz

Como podemos discutir durante el documento los patrones de diseño nos pueden ayudar es mejor tanto el código y el orden para una mejor funcionalidad de, pero cada patrón tiene sus ventajas y desventajas y tiene su manera de uso, no puede implementar un patrón de observación solo consultar de la base de datos sería algo incensario, si se sabe para que usar cada patrón. También de observación y de patrón medidor se puede sacar el mayor provecho.

Bibliografía

Harsh, K. (2023, febrero 14). *Guía Exhaustiva de Patrones de Diseño de JavaScript*. Kinsta®; Kinsta. https://kinsta.com/es/blog/patrones-de-diseno-javascript/

Mediator. (s/f). Refactoring.guru. Recuperado el 17 de abril de 2025, de https://refactoring.guru/es/design-patterns/mediator

(S/f-a). Flaticon.es. Recuperado el 17 de abril de 2025, de https://www.flaticon.es/icono-gratis/proposito_2967601?term=proposito&page=1&position=44&origin=search&related_id=2967601

(S/f-b). Flaticon.es. Recuperado el 17 de abril de 2025, de https://www.flaticon.es/icono-gratis/estructura-de-la-jerarquia_2857406?term=estructura&page=1&position=2&origin=search&related_id=2857406

(S/f-c). Flaticon.es. Recuperado el 17 de abril de 2025, de https://www.flaticon.es/icono-gratis/recompensa_3400793?term=motivacion&page=1&position=9&origin=search&related_id=3400793

(S/f-d). Flaticon.es. Recuperado el 17 de abril de 2025, de https://www.flaticon.es/icono-gratis/codigo_868786?term=codigo&page=1&position=8&origin=search&related_id=868786

(S/f-e). Refactoring.guru. Recuperado el 17 de abril de 2025, de https://refactoring.guru/images/patterns/content/observer/observer.png?id=6088e31e1b0d4a417506a66614dcf065

(S/f-f). Refactoring.guru. Recuperado el 17 de abril de 2025, de https://refactoring.guru/images/patterns/content/observer/observer-comic-1-es.png?id=27b9968d42e009f95c6d598b4029c22c

(S/f-g). Refactoring.guru. Recuperado el 17 de abril de 2025, de https://refactoring.guru/images/patterns/diagrams/observer/structure.png?id=365b7e2b8fbecc8948f34b9f8f16f33c