UCSM Esc. Prof. de Ingeniería de Sistemas

INFORME DE PRÁCTICAS

Diciembre - 2024

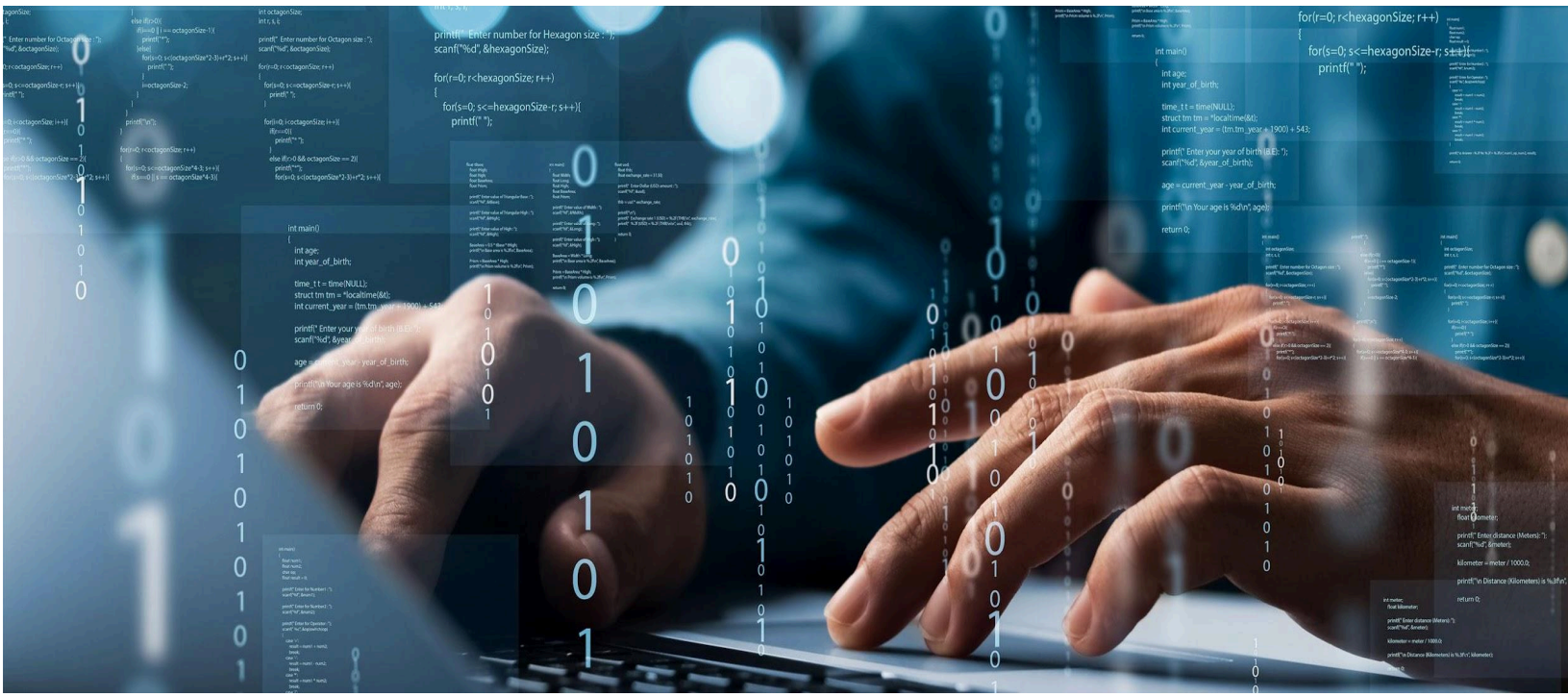
LENGUAJES DE PROGRAMACIÓN III

Práctica N° 11:

Patrones de Diseño: Observer, Strategy y Command

Elaborado por:

DIAZ ACOSTA KAHORI FERNANDA



GRUPO N° 05

LENGUAJES DE PROGRAMACIÓN

Presentado por:

2023242762 DIAZ ACOSTA KAHORI FERNANDA 100%

RECONOCIMIENTOS

El reconocimiento a los visionarios que dieron vida a Java. A James Gosling, el "padre de Java", y a su equipo en Sun Microsystems, quienes en la década de 1990 concibieron un lenguaje que revolucionaría la forma en que se interactúa con la tecnología. Su enfoque en la portabilidad, la seguridad y la facilidad de uso sentó las bases para un lenguaje que trascendería las plataformas y se convertiría en un estándar de la industria.

PALABRAS CLAVES

Patrones de diseño, Observer, Strategy, Command, Java, Programación orientada a objetos

ÍNDICE

1. ACTIVIDADES

ENLACE GITHUB:

https://github.com/KahoriDiazUCSM/Laboratorio_11

EXPERIENCIA DE PRÁCTICA 1:

Observer: Un sistema de notificaciones para múltiples usuarios

```
/**********************************************************************************
* NOMBRE : Observer.java
* DESCRIPCIÓN: Interfaz Observer -> Define el método update() para recibir
notificaciones
* de cambios en el estado de un sujeto observado.
**********************************************************************************/
interface Observer {
    void update(String message);
}
```

```
/**********************************************************************************
* NOMBRE : Usuario.java
* DESCRIPCIÓN: Clase Usuario -> Representa a un usuario que implementa la interfaz
Observer
* para recibir notificaciones.
**********************************************************************************/
class Usuario implements Observer {
    private String nombre;

    public Usuario(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public void update(String message) {
        System.out.println(nombre + " recibió la notificación: " + message);
    }
}
```

```
/*********************************************************************************
* NOMBRE : SistemaNotificaciones.java
* DESCRIPCIÓN: Clase SistemaNotificaciones -> Actúa como el sujeto observado en el
patrón Observer.
* Gestiona la lista de observadores y envía notificaciones a los mismos.
*********************************************************************************/
import java.util.ArrayList;
import java.util.List;

class SistemaNotificaciones {
    private List<Observer> observadores = new ArrayList<>();

    public void agregarObservador(Observer observador) {
        observadores.add(observador);
    }

    public void eliminarObservador(Observer observador) {
        observadores.remove(observador);
    }

    public void notificarTodos(String mensaje) {
        for (Observer observador : observadores) {
            observador.update(mensaje);
        }
    }
}
```

```
/*********************************************************************************
* NOMBRE : SistemaNotificacionesDemo.java
* DESCRIPCIÓN: Clase SistemaNotificacionesDemo -> Clase principal que demuestra el uso
del patrón Observer
* mediante la implementación de un sistema de notificaciones para múltiples usuarios.
*********************************************************************************/
public class SistemaNotificacionesDemo {
    public static void main(String[] args) {
        SistemaNotificaciones sistema = new SistemaNotificaciones();

        Usuario usuario1 = new Usuario("Juan");
        Usuario usuario2 = new Usuario("María");

        sistema.agregarObservador(usuario1);
        sistema.agregarObservador(usuario2);

        sistema.notificarTodos("Nueva promoción disponible!");

        sistema.eliminarObservador(usuario2);

        sistema.notificarTodos("Actualización de producto");
    }
}
```

```
EXPLICACIÓN:

Interfaz EstrategiaDescuento: Define el método aplicarDescuento(double precio) que es
implementado por todas las estrategias de descuento.

Clases Concretas de Estrategias:

SinDescuento: No aplica ningún descuento al precio base.

DescuentoFijo: Aplica un descuento fijo de $10.

DescuentoPorcentual: Aplica un 30% de descuento si la cantidad es 2 o más.

DescuentoPorcentualAcumulado: Aplica un 50% de descuento si la cantidad es 3 o más.


Clase Producto: Representa el producto al que se le aplicarán los descuentos.

Clase CalculadoraDePrecios: Permite cambiar dinámicamente la estrategia de descuento y
calcular el precio final.

Clase SistemaDescuentosDemo: Demuestra cómo seleccionar estrategias desde un menú
interactivo y calcular el precio final.
```

```
SALIDA:

Juan recibió la notificación: Nueva promoción disponible!
María recibió la notificación: Nueva promoción disponible!
María recibió la notificación: Actualización de producto
```

Strategy: Un sistema de cálculo con diferentes estrategias de promociones de productos.

```
/**********************************************************************************
* NOMBRE : EstrategiaDescuento.java
* DESCRIPCIÓN: Interfaz EstrategiaDescuento -> Define un método para aplicar diferentes
estrategias
* de descuento a un precio base.
**********************************************************************************/
interface EstrategiaDescuento {
```

```
    double aplicarDescuento(double precio);
}
```

```
/**********************************************************************************
* NOMBRE : SinDescuento.java
* DESCRIPCIÓN: Clase SinDescuento -> Implementa la estrategia que no aplica ningún
descuento
* al precio base.
**********************************************************************************/
class SinDescuento implements EstrategiaDescuento {
    @Override
    public double aplicarDescuento(double precio) {
        return precio;
    }
}
```

```
/**********************************************************************************
* NOMBRE : DescuentoFijo.java
* DESCRIPCIÓN: Clase DescuentoFijo -> Implementa una estrategia que aplica un descuento
fijo
* de $10 al precio base.
**********************************************************************************/
class DescuentoFijo implements EstrategiaDescuento {
    @Override
    public double aplicarDescuento(double precio) {
        return precio - 10;
    }
}
```

```
/**********************************************************************************
* NOMBRE : DescuentoPorcentual.java
* DESCRIPCIÓN: Clase DescuentoPorcentual -> Aplica un descuento del 30% si la cantidad
de
* productos es 2 o más. Implementa la interfaz EstrategiaDescuento.
**********************************************************************************/
class DescuentoPorcentual implements EstrategiaDescuento {
    private int cantidad;

    public DescuentoPorcentual(int cantidad) {
        this.cantidad = cantidad;
    }

    @Override
    public double aplicarDescuento(double precio) {
        if (cantidad >= 2) {
```

```
            return precio * 0.7;
        }
        return precio;
    }
}
```

```
/**********************************************************************************
* NOMBRE : DescuentoPorcentualAcumulado.java
* DESCRIPCIÓN: Clase DescuentoPorcentualAcumulado -> Aplica un descuento del 50% si la
cantidad
* de productos es 3 o más. Implementa la interfaz EstrategiaDescuento.
**********************************************************************************/
class DescuentoPorcentualAcumulado implements EstrategiaDescuento {
    private int cantidad;

    public DescuentoPorcentualAcumulado(int cantidad) {
        this.cantidad = cantidad;
    }

    @Override
    public double aplicarDescuento(double precio) {
        if (cantidad >= 3) {
            return precio * 0.5;
        }
        return precio;
    }
}
```

```
/**********************************************************************************
* NOMBRE : Producto.java
* DESCRIPCIÓN: Clase Producto -> Representa un producto con un nombre y un precio base.
**********************************************************************************/
class Producto {
    private String nombre;
    private double precio;

    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public double getPrecio() {
        return precio;
    }
}
```

```
/**********************************************************************************
* NOMBRE : CalculadoraDePrecios.java
* DESCRIPCIÓN: Clase CalculadoraDePrecios -> Permite calcular el precio final de un
producto
* utilizando una estrategia de descuento seleccionada.
**********************************************************************************/
class CalculadoraDePrecios {
    private EstrategiaDescuento estrategia;

    public void setEstrategia(EstrategiaDescuento estrategia) {
        this.estrategia = estrategia;
    }

    public double calcularPrecioFinal(Producto producto) {
        return estrategia.aplicarDescuento(producto.getPrecio());
    }
}
```

```
/**********************************************************************************
* NOMBRE : SistemaDescuentosDemo.java
* DESCRIPCIÓN: Clase SistemaDescuentosDemo -> Clase principal que demuestra el uso del
patrón Strategy
* para calcular el precio final de un producto utilizando diferentes estrategias de
descuento.
**********************************************************************************/
import java.util.Scanner;

public class SistemaDescuentosDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        CalculadoraDePrecios calculadora = new CalculadoraDePrecios();
        Producto producto = new Producto("Camiseta", 100);

        while (true) {
            System.out.println("\nSeleccione una estrategia de descuento:");
            System.out.println("1. Sin descuento");
            System.out.println("2. Descuento fijo ($10)");
            System.out.println("3. Descuento porcentual (30% para 2 o más productos)");
            System.out.println("4. Descuento porcentual acumulado (50% para 3 o más
productos)");
            System.out.println("5. Salir");

            int opcion = scanner.nextInt();

            if (opcion == 5) break;

            EstrategiaDescuento estrategia;
            switch (opcion) {
```

```
                case 1:
                    estrategia = new SinDescuento();
                    break;
                case 2:
                    estrategia = new DescuentoFijo();
                    break;
                case 3:
                    System.out.println("Ingrese la cantidad de productos:");
                    int cantidad = scanner.nextInt();
                    estrategia = new DescuentoPorcentual(cantidad);
                    break;
                case 4:
                    System.out.println("Ingrese la cantidad de productos:");
                    cantidad = scanner.nextInt();
                    estrategia = new DescuentoPorcentualAcumulado(cantidad);
                    break;
                default:
                    System.out.println("Opción no válida");
                    continue;
            }

            calculadora.setEstrategia(estrategia);
            double precioFinal = calculadora.calcularPrecioFinal(producto);
            System.out.println("Precio final: $" + precioFinal);
        }

        scanner.close();
    }
}
```

EXPLICACIÓN:

Interfaz Observer: Define el método update(String message) que será implementado por las clases observadoras para recibir las notificaciones.

Clase Usuario: Implementa la interfaz Observer y representa a cada usuario que desea recibir notificaciones.

Clase SistemaNotificaciones: Actúa como el sujeto observado. Gestiona una lista de observadores (usuarios) y notifica a todos cuando se produce un evento.

Clase SistemaNotificacionesDemo: Demuestra cómo agregar usuarios al sistema, enviar notificaciones y eliminar observadores dinámicamente.

```
SALIDA:
Seleccione una estrategia de descuento:
1. Sin descuento
2. Descuento fijo ($10)
3. Descuento porcentual (30% para 2 o más productos)
4. Descuento porcentual acumulado (50% para 3 o más productos)
5. Salir
3
Ingrese la cantidad de productos:
3
Precio final: $70.0

Seleccione una estrategia de descuento:
1. Sin descuento
2. Descuento fijo ($10)
3. Descuento porcentual (30% para 2 o más productos)
4. Descuento porcentual acumulado (50% para 3 o más productos)
5. Salir
2
Precio final: $90.0
```

Command: Un control remoto de un televisor con cinco (5) funcionalidades.

```
/**********************************************************************************
* NOMBRE : Command.java
* DESCRIPCIÓN: Interfaz Command -> Define el método execute() para encapsular una
acción como un objeto
**********************************************************************************/
interface Command {
    void execute();
}
```

```
/**********************************************************************************
* NOMBRE : Televisor .java
* DESCRIPCIÓN: Clase Televisor -> Representa un televisor con funciones básicas como
encender, apagar, cambiar de canal, y ajustar el volumen.

**********************************************************************************/
class Televisor {
    public void encender() {
        System.out.println("Televisor encendido");
    }
```

```
    public void apagar() {
        System.out.println("Televisor apagado");
    }

    public void cambiarCanal(int canal) {
        System.out.println("Cambiando al canal " + canal);
    }

    public void subirVolumen() {
        System.out.println("Subiendo volumen");
    }

    public void bajarVolumen() {
        System.out.println("Bajando volumen");
    }
}
```

```
/**********************************************************************************
* NOMBRE: EncenderCommand.java
* DESCRIPCIÓN: Clase EncenderCommand -> Comando para encapsular la acción de encender
el televisor.
**********************************************************************************/
class EncenderCommand implements Command {
    private Televisor tv;

    public EncenderCommand(Televisor tv) {
        this.tv = tv;
    }

    @Override
    public void execute() {
        tv.encender();
    }
}
```

```
/**********************************************************************************
* NOMBRE: ApagarCommand.java
* DESCRIPCIÓN: Clase ApagarCommand -> Comando para encapsular la acción de apagar el
televisor.
**********************************************************************************/
class ApagarCommand implements Command {
    private Televisor tv;

    public ApagarCommand(Televisor tv) {
        this.tv = tv;
    }
```

```
    @Override
    public void execute() {
        tv.apagar();
    }
}
```

```
/**********************************************************************************
* NOMBRE: CambiarCanalCommand.java
* DESCRIPCIÓN: Clase CambiarCanalCommand -> Comando para encapsular la acción de
cambiar el canal del televisor.
**********************************************************************************/
class CambiarCanalCommand implements Command {
    private Televisor tv;
    private int canal;

    public CambiarCanalCommand(Televisor tv, int canal) {
        this.tv = tv;
        this.canal = canal;
    }

    @Override
    public void execute() {
        tv.cambiarCanal(canal);
    }
}
```

```
/**********************************************************************************
 * NOMBRE: SubirVolumenCommand.java
 * DESCRIPCIÓN: Clase SubirVolumenCommand -> Comando que encapsula la acción de subir
el volumen del televisor, implementando la interfaz Command.

**********************************************************************************/
class SubirVolumenCommand implements Command {
    private Televisor tv;

    public SubirVolumenCommand(Televisor tv) {
        this.tv = tv;
    }

    @Override
    public void execute() {
        tv.subirVolumen();
    }
}
```

```
/*********************************************************************************
 * NOMBRE: BajarVolumenCommand.java
 * DESCRIPCIÓN: Clase BajarVolumenCommand -> Comando que encapsula la acción de bajar
el volumen del televisor, implementando la interfaz Command.
*********************************************************************************/
class BajarVolumenCommand implements Command {
    private Televisor tv;

    public BajarVolumenCommand(Televisor tv) {
        this.tv = tv;
    }

    @Override
    public void execute() {
        tv.bajarVolumen();
    }
}
```

```
/*********************************************************************************
 * NOMBRE: ControlRemoto.java
 * DESCRIPCIÓN: Clase ControlRemoto -> Representa un control remoto que puede asignar
comandos a botones específicos y ejecutarlos en función de la interacción del usuario.
*********************************************************************************/
class ControlRemoto {
    private Map<String, Command> comandos = new HashMap<>();

    public void setComando(String boton, Command comando) {
        comandos.put(boton, comando);
    }

    public void presionarBoton(String boton) {
        Command comando = comandos.get(boton);
        if (comando != null) {
            comando.execute();
        } else {
            System.out.println("Botón no configurado");
        }
    }
}
```

```
/**********************************************************************************
 * NOMBRE: CommandDemo.java
 * DESCRIPCIÓN: Clase CommandDemo -> Clase principal que demuestra el uso del patrón
Command implementado en el control remoto de un televisor con diferentes
funcionalidades.
**********************************************************************************/
public class ControlRemotoTVDemo {
    public static void main(String[] args) {
        Televisor tv = new Televisor();
        ControlRemoto control = new ControlRemoto();

        control.setComando("POWER", new EncenderCommand(tv));
        control.setComando("OFF", new ApagarCommand(tv));
        control.setComando("CANAL 1", new CambiarCanalCommand(tv, 1));
        control.setComando("SUBIR VOLUMEN", new SubirVolumenCommand(tv));
        control.setComando("BAJAR VOLUMEN", new BajarVolumenCommand(tv));

        control.presionarBoton("POWER");
        control.presionarBoton("CANAL 1");
        control.presionarBoton("SUBIR VOLUMEN");
        control.presionarBoton("BAJAR VOLUMEN");
        control.presionarBoton("OFF");
    }
}
```

EXPLICACIÓN:

Interfaz Command: Define el método execute() que será implementado por los comandos concretos.

Clases Concretas de Comandos:

EncenderCommand: Encapsula la acción de encender el televisor.

ApagarCommand: Encapsula la acción de apagar el televisor.

CambiarCanalCommand: Encapsula la acción de cambiar de canal.

SubirVolumenCommand y BajarVolumenCommand: Encapsulan las acciones de ajustar el volumen.

Clase Televisor: Contiene las funcionalidades básicas del televisor (encender, apagar, cambiar canal, subir y bajar volumen).

Clase ControlRemoto: Gestiona los comandos asignados a cada botón y los ejecuta cuando son presionados.

Clase CommandDemo: Demuestra cómo asignar comandos al control remoto y ejecutar las acciones.

```
SALIDA:
Televisor encendido
Cambiando al canal 1
Subiendo volumen
Bajando volumen
Televisor apagado
```

EXPERIENCIA DE PRÁCTICA 2:

Observer: Maneja las notificaciones dinámicas.

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

/*****************************************************************************
* NOMBRE : Observer.java
* DESCRIPCIÓN: Define el método update() para recibir notificaciones de cambios.
*****************************************************************************/
interface Observer {
    void update(String message);
}

/*****************************************************************************
* NOMBRE : Usuario.java
* DESCRIPCIÓN: Implementa Observer para recibir notificaciones sobre promociones.
*****************************************************************************/
class Usuario implements Observer {
    private String nombre;

    public Usuario(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public void update(String message) {
        System.out.println(nombre + " recibió la notificación: " + message);
    }
}

/*****************************************************************************
* NOMBRE : SistemaNotificaciones.java
* DESCRIPCIÓN: Gestiona observadores y envía notificaciones.
*****************************************************************************/
class SistemaNotificaciones {
    private List<Observer> observadores = new ArrayList<>();
```

```
    public void agregarObservador(Observer observador) {
        observadores.add(observador);
    }

    public void eliminarObservador(Observer observador) {
        observadores.remove(observador);
    }

    public void notificarTodos(String mensaje) {
        for (Observer observador : observadores) {
            observador.update(mensaje);
        }
    }
}
```

```
EXPLICACIÓN:
Interfaz Observer: Define el contrato para los objetos que desean recibir
notificaciones.
Clase Usuario: Representa un observador que implementa el método update para recibir
mensajes.
Clase SistemaNotificaciones: Gestiona la lista de observadores y les envía mensajes
automáticamente cuando ocurre un evento.
```

Strategy: Calcula descuentos dinámicos sobre los productos.

```
/*********************************************************************************
* NOMBRE : EstrategiaDescuento.java
* DESCRIPCIÓN: Define estrategias de descuento.
*********************************************************************************/
interface EstrategiaDescuento {
    double aplicarDescuento(double precio);
}

/*********************************************************************************
* NOMBRE : Descuentos.java
* DESCRIPCIÓN: Implementa varias estrategias de descuento.
*********************************************************************************/
class SinDescuento implements EstrategiaDescuento {
    @Override
    public double aplicarDescuento(double precio) {
        return precio;
    }
}

class DescuentoFijo implements EstrategiaDescuento {
    @Override
    public double aplicarDescuento(double precio) {
```

```
        return precio - 10;
    }
}

class DescuentoPorcentual implements EstrategiaDescuento {
    private int cantidad;

    public DescuentoPorcentual(int cantidad) {
        this.cantidad = cantidad;
    }

    @Override
    public double aplicarDescuento(double precio) {
        if (cantidad >= 2) {
            return precio * 0.7;
        }
        return precio;
    }
}

/**********************************************************************************
* NOMBRE : Producto.java
* DESCRIPCIÓN: Representa un producto con un precio base.
**********************************************************************************/
class Producto {
    private String nombre;
    private double precio;

    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public double getPrecio() {
        return precio;
    }

    public String getNombre() {
        return nombre;
    }
}
```

```
EXPLICACIÓN:
Interfaz EstrategiaDescuento: Define el contrato para aplicar diferentes tipos de
descuentos.

Clases de Descuento:
SinDescuento: No aplica descuentos.

DescuentoFijo: Aplica un descuento fijo de $10.

Clase Producto: Representa el producto al que se le aplicarán los descuentos.
```

Command: Gestiona la activación de notificaciones y aplicación de descuentos.

```
import java.util.HashMap;
import java.util.Map;

/*********************************************************************************
* NOMBRE : Command.java
* DESCRIPCIÓN: Define el método execute() para encapsular acciones.
*********************************************************************************/
interface Command {
    void execute();
}

/*********************************************************************************
* NOMBRE : Comandos.java
* DESCRIPCIÓN: Implementa comandos para encender notificaciones y aplicar descuentos.
*********************************************************************************/
class ActivarNotificacionesCommand implements Command {
    private SistemaNotificaciones sistema;
    private Usuario usuario;

    public ActivarNotificacionesCommand(SistemaNotificaciones sistema, Usuario usuario)
{
        this.sistema = sistema;
        this.usuario = usuario;
    }

    @Override
    public void execute() {
        sistema.agregarObservador(usuario);
        System.out.println(usuario + " ahora recibe notificaciones.");
    }
}

class AplicarDescuentoCommand implements Command {
    private EstrategiaDescuento estrategia;
    private Producto producto;

    public AplicarDescuentoCommand(EstrategiaDescuento estrategia, Producto producto) {
```

```
        this.estrategia = estrategia;
        this.producto = producto;
    }

    @Override
    public void execute() {
        double precioFinal = estrategia.aplicarDescuento(producto.getPrecio());
        System.out.println("El precio final de " + producto.getNombre() + " es $" +
precioFinal);
    }
}

/**********************************************************************************
* NOMBRE : ControlRemoto.java
* DESCRIPCIÓN: Asocia botones con comandos y ejecuta las acciones.
**********************************************************************************/
class ControlRemoto {
    private Map<String, Command> comandos = new HashMap<>();

    public void setComando(String boton, Command comando) {
        comandos.put(boton, comando);
    }

    public void presionarBoton(String boton) {
        Command comando = comandos.get(boton);
        if (comando != null) {
            comando.execute();
        } else {
            System.out.println("Botón no configurado");
        }
    }
}
```

```
EXPLICACIÓN:
Interfaz Command: Define el contrato para encapsular acciones como comandos.

Clases Concretas de Comandos:

ActivarNotificacionesCommand: Registra un usuario en el sistema de notificaciones
.
AplicarDescuentoCommand: Aplica una estrategia de descuento a un producto.

Clase ControlRemoto: Gestiona los comandos y los ejecuta al presionar un botón.
```

Prueba

```
/*********************************************************************************
* NOMBRE : AplicacionTienda.java
* DESCRIPCIÓN: Integra Observer, Strategy y Command para simular una tienda en línea.
*********************************************************************************/
public class AplicacionTienda {
    public static void main(String[] args) {
        // Sistema de Notificaciones
        SistemaNotificaciones sistemaNotificaciones = new SistemaNotificaciones();
        Usuario usuario1 = new Usuario("Juan");
        Usuario usuario2 = new Usuario("María");

        // Productos y estrategias de descuento
        Producto producto = new Producto("Laptop", 1000);
        EstrategiaDescuento sinDescuento = new SinDescuento();
        EstrategiaDescuento descuentoFijo = new DescuentoFijo();

        // Control remoto
        ControlRemoto control = new ControlRemoto();

        // Configuración de comandos
        control.setComando("NOTIFICAR JUAN", new
ActivarNotificacionesCommand(sistemaNotificaciones, usuario1));
        control.setComando("NOTIFICAR MARÍA", new
ActivarNotificacionesCommand(sistemaNotificaciones, usuario2));
        control.setComando("DESCUENTO FIJO", new AplicarDescuentoCommand(descuentoFijo,
producto));
        control.setComando("SIN DESCUENTO", new AplicarDescuentoCommand(sinDescuento,
producto));

        // Simulación
        control.presionarBoton("NOTIFICAR JUAN");
        sistemaNotificaciones.notificarTodos("¡Nueva oferta en laptops!");
        control.presionarBoton("DESCUENTO FIJO");
    }
}
```

```
EXPLICACIÓN:
Se registra a los usuarios para recibir notificaciones.
Se seleccionan y aplican estrategias de descuento a un producto.
Las acciones (activar notificaciones y aplicar descuentos) son gestionadas mediante un
control remoto, que encapsula la lógica de cada acción.
```

```
SALIDA

Juan ahora recibe notificaciones.
Juan recibió la notificación: ¡Nueva oferta en laptops!
El precio final de Laptop es $990.0
```

2. EJERCICIOS

ENLACE GITHUB:
https://github.com/KahoriDiazUCSM/Laboratorio_11/tree/main/EJERCICIOS

2.1 EJERCICIO 1

Sistema de Notificaciones (Observer):

- ❖ Diseña una aplicación que implemente el patrón Observer para un sistema de notificaciones:
- ❖ Usuarios registrados reciben notificaciones de eventos, como promociones o actualizaciones de productos.
- ❖ Crea clases Usuario y Notificación, donde cada usuario puede suscribirse y recibir notificaciones automáticamente.
- ❖ Agrega la funcionalidad para que los usuarios se puedan suscribir/de suscribir dinámicamente.

```
/*********************************************************************************
* NOMBRE : Observer.java
* DESCRIPCIÓN: Interfaz Observer -> Define el método update() para que las clases
suscriptoras
* reciban notificaciones automáticas de cambios en el estado del sujeto observado.
*********************************************************************************/
interface Observer {
void update(String mensaje); }
```

```
/*********************************************************************************
* NOMBRE : Notificacion.java
* DESCRIPCIÓN: Clase Notificacion -> Actúa como el sujeto observado. Permite suscribir,
* desuscribir y notificar a los observadores registrados.
*********************************************************************************/
class Notificacion {
```

```
    private List<Observer> observers = new ArrayList<>();

    // Agregar un suscriptor
    public void agregarSuscriptor(Observer observer) {
        observers.add(observer);
        System.out.println("Usuario suscrito.");
    }

    // Remover un suscriptor
    public void eliminarSuscriptor(Observer observer) {
        observers.remove(observer);
        System.out.println("Usuario desuscrito.");
    }

    // Notificar a todos los suscriptores
    public void notificar(String mensaje) {
        for (Observer observer : observers) {
            observer.update(mensaje);
        }
    }
}
```

```
/*****************************************************************************
* NOMBRE : Usuario.java
* DESCRIPCIÓN: Clase Usuario -> Representa un usuario que se puede suscribir al sistema
* de notificaciones y recibir mensajes automáticamente.
*****************************************************************************/
class Usuario implements Observer {
    private String nombre;

    public Usuario(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public void update(String mensaje) {
        System.out.println(nombre + " recibió la notificación: " + mensaje);
    }
}
```

```
/*****************************************************************************
* NOMBRE : SistemaNotificaciones.java
* DESCRIPCIÓN: Clase principal que implementa el sistema de notificaciones utilizando
* el patrón Observer. Permite probar las funcionalidades de suscripción, desuscripción
* y notificación dinámica.
*****************************************************************************/
public class SistemaNotificaciones {
```

```
    public static void main(String[] args) {
        // Crear el sujeto observado
        Notificacion notificacion = new Notificacion();

        // Crear observadores (usuarios)
        Usuario usuario1 = new Usuario("Sebastián");
        Usuario usuario2 = new Usuario("Joaquín");

        // Suscribir usuarios
        notificacion.agregarSuscriptor(usuario1);
        notificacion.agregarSuscriptor(usuario2);

        // Enviar notificación a todos los suscriptores
        System.out.println("\nEnviando notificación...");
        notificacion.notificar("Nueva promoción en tecnología!");

        // Desuscribir un usuario
        System.out.println("\nDesuscribiendo a Joaquín...");
        notificacion.eliminarSuscriptor(usuario2);

        // Enviar notificación después de la desuscripción
        System.out.println("\nEnviando notificación...");
        notificacion.notificar("Actualización en productos disponibles.");
    }
}
```

```
EXPLICACIÓN:
Interfaz Observer:
Define el contrato para las clases que actuen como observadores.
Contiene el método update(String mensaje) que se ejecuta cuando el sujeto notifica un
cambio.

Clase Notificacion:
Actúa como el sujeto observado.
Permite:
Agregar suscriptores con agregarSuscriptor.
Eliminar suscriptores con eliminarSuscriptor.
Notificar cambios a todos los suscriptores mediante notificar.

Clase Usuario:
Representa un usuario suscrito al sistema.
Implementa el método update para recibir mensajes cuando el sujeto notifica un cambio.

Clase Principal SistemaNotificaciones:
Simula el sistema de notificaciones.
Permite probar:
Registro de usuarios.
Notificaciones a todos los usuarios registrados.
Eliminación dinámica de usuarios del sistema.
```

```
SALIDA

Usuario suscrito.
Usuario suscrito.
Primera notificación:


Enviando notificación...
Sebastián recibió la notificación: Nueva promoción en tecnología!
Joaquín recibió la notificación: Nueva promoción en tecnología!

Desuscripción de Joaquín:

Desuscribiendo a Joaquín...
Usuario desuscrito.

Segunda notificación:

Enviando notificación...
Sebastián recibió la notificación: Actualización en productos disponibles.
```

2.2 EJERCICIO 2

Estrategias de Descuento (Strategy). Crea un sistema que utilice el patrón Strategy para calcular el precio final de un producto:

- ❖ Implementa tres estrategias de descuento (SinDescuento, DescuentoFijo, DescuentoPorcentual, DescuentoPorcentualAcumulado).
- ❖ DescuentoFijo → 10%
- ❖ DescuentoPorcentual → 2 productos iguales 30% de descuento.
- ❖ DescuentoPorcentualAcumulado → a partir de 3 productos descuento del 50% sobre el producto mas bajo de precio.
- ❖ Diseña una clase Producto y una clase CalculadoraDePrecios para aplicar el descuento seleccionado.
- ❖ Permite que el usuario elija la estrategia desde un menú interactivo.

```
/**********************************************************************************
* NOMBRE : EstrategiaDescuento.java
* DESCRIPCIÓN: Interfaz que define el método para calcular el precio final con
descuento.
**********************************************************************************/
public interface EstrategiaDescuento {
```

```
    double calcularPrecioFinal(List<Producto> productos);
}
```

```
/*****************************************************************************
* NOMBRE : SinDescuento.java
* DESCRIPCIÓN: Estrategia de descuento que no aplica ningún descuento.
*****************************************************************************/
public class SinDescuento implements EstrategiaDescuento {
    @Override
    public double calcularPrecioFinal(List<Producto> productos) {
        return productos.stream().mapToDouble(Producto::getPrecio).sum();
    }
}
```

```
/*****************************************************************************
* NOMBRE : DescuentoFijo.java
* DESCRIPCIÓN: Estrategia de descuento que aplica un 10% sobre el total.
*****************************************************************************/
public class DescuentoFijo implements EstrategiaDescuento {
    @Override
    public double calcularPrecioFinal(List<Producto> productos) {
        double total = productos.stream().mapToDouble(Producto::getPrecio).sum();
        return total * 0.90; // 10% de descuento
    }
}
```

```
/*****************************************************************************
* NOMBRE : DescuentoPorcentual.java
* DESCRIPCIÓN: Estrategia que aplica un 30% de descuento si hay al menos 2 productos
iguales.
*****************************************************************************/
public class DescuentoPorcentual implements EstrategiaDescuento {
    @Override
    public double calcularPrecioFinal(List<Producto> productos) {
        double total = 0;
        for (int i = 0; i < productos.size(); i++) {
            Producto actual = productos.get(i);
            long count = productos.stream().filter(p ->
p.getNombre().equals(actual.getNombre())).count();
            if (count >= 2) {
                total += actual.getPrecio() * 0.70; // 30% de descuento
                productos.remove(i); // Evita contar el producto dos veces
                i--; // Ajusta el índice
            } else {
                total += actual.getPrecio();
```

```
            }
        }
        return total;
    }
}
```

```
/*****************************************************************************
* NOMBRE : DescuentoPorcentualAcumulado.java
* DESCRIPCIÓN: Estrategia que aplica un 50% de descuento al producto más barato si hay
al menos 3 productos.
*****************************************************************************/
public class DescuentoPorcentualAcumulado implements EstrategiaDescuento {
    @Override
    public double calcularPrecioFinal(List<Producto> productos) {
        double total = productos.stream().mapToDouble(Producto::getPrecio).sum();
        if (productos.size() >= 3) {
            double masBarato =
productos.stream().mapToDouble(Producto::getPrecio).min().orElse(0);
            total -= masBarato * 0.50; // 50% de descuento en el más barato
        }
        return total;
    }
}
```

```
/*****************************************************************************
* NOMBRE : Producto.java
* DESCRIPCIÓN: Clase que representa un producto con nombre y precio.
*****************************************************************************/
public class Producto {
    private String nombre;
    private double precio;

    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public String getNombre() {
        return nombre;
    }

    public double getPrecio() {
        return precio;
    }

    @Override
    public String toString() {
        return nombre + " S/." + precio;
```

```
    }
}
```

```
/*********************************************************************************
* NOMBRE : CalculadoraDePrecios.java
* DESCRIPCIÓN: Clase que aplica una estrategia de descuento seleccionada para calcular
el precio final.
*********************************************************************************/
public class CalculadoraDePrecios {
    private EstrategiaDescuento estrategia;

    public void setEstrategia(EstrategiaDescuento estrategia) {
        this.estrategia = estrategia;
    }

    public double calcularPrecioFinal(List<Producto> productos) {
        if (estrategia == null) {
            throw new IllegalStateException("Estrategia de descuento no
seleccionada.");
        }
        return estrategia.calcularPrecioFinal(productos);
    }
}
```

```
/*********************************************************************************
* NOMBRE : SistemaDeDescuento.java
* DESCRIPCIÓN: Clase principal que permite seleccionar estrategias de descuento y
calcular el precio final.
*********************************************************************************/
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class SistemaDeDescuento {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<Producto> productos = new ArrayList<>();
        productos.add(new Producto("Laptop", 3000));
        productos.add(new Producto("Laptop", 3000));
        productos.add(new Producto("Mouse", 150));
        productos.add(new Producto("Teclado", 200));
        productos.add(new Producto("Auriculares", 350));

        CalculadoraDePrecios calculadora = new CalculadoraDePrecios();

        System.out.println("Lista de Productos:");
        productos.forEach(System.out::println);
```

```
        System.out.println("\n--- Elige una estrategia de descuento ---");
        System.out.println("1. Sin Descuento");
        System.out.println("2. Descuento Fijo (10%)");
        System.out.println("3. Descuento Porcentual (30% si hay 2 productos iguales)");
        System.out.println("4. Descuento Porcentual Acumulado (50% sobre el producto
más barato si hay al menos 3 productos)");

        int opcion = scanner.nextInt();
        switch (opcion) {
            case 1 -> calculadora.setEstrategia(new SinDescuento());
            case 2 -> calculadora.setEstrategia(new DescuentoFijo());
            case 3 -> calculadora.setEstrategia(new DescuentoPorcentual());
            case 4 -> calculadora.setEstrategia(new DescuentoPorcentualAcumulado());
            default -> {
                System.out.println("Opción no válida.");
                return;
            }
        }

        double precioFinal = calculadora.calcularPrecioFinal(new
ArrayList<>(productos));
        System.out.println("\n--- Precio Final: S/." + precioFinal);
    }
}
```

EXPLICACIÓN:

Clase Producto:

Representa un producto con atributos básicos:
Atributos: nombre y precio.
Métodos principales:
Constructor para inicializar los valores.
Métodos de acceso (getNombre, getPrecio).
Método toString para representar el producto como texto.

Interfaz EstrategiaDescuento:

Define un contrato para calcular precios con diferentes tipos de descuentos:
Contiene el método calcularPrecioFinal(List<Producto> productos).
Cada estrategia de descuento implementa esta interfaz.

Clases de Estrategias de Descuento:

SinDescuento: Retorna el precio total de los productos sin aplicar descuentos.
DescuentoFijo: Aplica un descuento fijo del 10% al total.
DescuentoPorcentual: Identifica productos repetidos y aplica un 30% de descuento a ellos.
DescuentoPorcentualAcumulado: Si hay 3 o más productos, aplica un 50% de descuento al más barato.
Clase CalculadoraDePrecios:
Encargada de gestionar las estrategias de descuento:
Método setEstrategia: Cambia la estrategia de descuento actual.
Método calcularPrecioFinal: Usa la estrategia configurada para calcular el precio total

```
de una lista de productos.
```

```
SALIDA

Lista de Productos:
Laptop S/.3000.0
Laptop S/.3000.0
Mouse S/.150.0
Teclado S/.200.0
Auriculares S/.350.0

--- Elige una estrategia de descuento ---
1. Sin Descuento
2. Descuento Fijo (10%)
3. Descuento Porcentual (30% si hay 2 productos iguales)
4. Descuento Porcentual Acumulado (50% sobre el producto más barato si hay al menos 3
productos)

--- Precio Final: S/.6075.0
```

2.3 EJERCICIO 3

Sistema de Control de Dispositivos (Command). Crea un sistema que utilice el patrón Command para controlar distintos dispositivos electrónicos de forma remota:

- ❖ Los dispositivos disponibles son: Luz, Ventilador y Aire Acondicionado.
- ❖ Crea una clase para cada dispositivo con métodos para encender y apagar.
- ❖ Crea comandos específicos para cada acción (encender y apagar) de los dispositivos.
- ❖ Implementa una clase ControlRemoto que permita almacenar y ejecutar comandos dinámicamente.
- ❖ Agrega la funcionalidad de deshacer la última acción realizada.

```
/**********************************************************************************
* NOMBRE: Command.java
* DESCRIPCIÓN: Define la interfaz Command para implementar comandos genéricos.
**********************************************************************************/
interface Command {
    void execute();
    void undo();
}
```

```
/*********************************************************************************
* NOMBRE: Luz.java
* DESCRIPCIÓN: Representa el dispositivo Luz con métodos para encender y apagar.
*********************************************************************************/
class Luz {
    public void encender() {
        System.out.println("Luz encendida");
    }

    public void apagar() {
        System.out.println("Luz apagada");
    }
}
```

```
/*********************************************************************************
* NOMBRE: Ventilador.java
* DESCRIPCIÓN: Representa el dispositivo Ventilador con métodos para encender y apagar.
*********************************************************************************/
class Ventilador {
    public void encender() {
        System.out.println("Ventilador encendido");
    }

    public void apagar() {
        System.out.println("Ventilador apagado");
    }
}
```

```
/*********************************************************************************
* NOMBRE: AireAcondicionado.java
* DESCRIPCIÓN: Representa el dispositivo Aire Acondicionado con métodos para encender y
apagar.
*********************************************************************************/
class AireAcondicionado {
    public void encender() {
        System.out.println("Aire acondicionado encendido");
    }

    public void apagar() {
        System.out.println("Aire acondicionado apagado");
    }
}
```

```
/*****************************************************************************
* NOMBRE: EncenderLuz.java
* DESCRIPCIÓN: Implementa Command para encender una Luz.
*****************************************************************************/
class EncenderLuz implements Command {
    private Luz luz;

    public EncenderLuz(Luz luz) {
        this.luz = luz;
    }

    @Override
    public void execute() {
        luz.encender();
    }

    @Override
    public void undo() {
        luz.apagar();
    }
}
```

```
/*****************************************************************************
* NOMBRE: ApagarLuz.java
* DESCRIPCIÓN: Implementa Command para apagar una Luz.
*****************************************************************************/
class ApagarLuz implements Command {
    private Luz luz;

    public ApagarLuz(Luz luz) {
        this.luz = luz;
    }

    @Override
    public void execute() {
        luz.apagar();
    }

    @Override
    public void undo() {
        luz.encender();
    }
}
```

```
/*****************************************************************************
* NOMBRE: EncenderVentilador.java
* DESCRIPCIÓN: Implementa Command para encender un Ventilador.
*****************************************************************************/
class EncenderVentilador implements Command {
    private Ventilador ventilador;

    public EncenderVentilador(Ventilador ventilador) {
        this.ventilador = ventilador;
    }

    @Override
    public void execute() {
        ventilador.encender();
    }

    @Override
    public void undo() {
        ventilador.apagar();
    }
}
```

```
/*****************************************************************************
* NOMBRE: ApagarVentilador.java
* DESCRIPCIÓN: Implementa Command para apagar un Ventilador.
*****************************************************************************/
class ApagarVentilador implements Command {
    private Ventilador ventilador;

    public ApagarVentilador(Ventilador ventilador) {
        this.ventilador = ventilador;
    }

    @Override
    public void execute() {
        ventilador.apagar();
    }

    @Override
    public void undo() {
        ventilador.encender();
    }
}
```

```
/*****************************************************************************
* NOMBRE: EncenderAire.java
* DESCRIPCIÓN: Implementa Command para encender un Aire Acondicionado.
*****************************************************************************/
class EncenderAire implements Command {
    private AireAcondicionado aire;

    public EncenderAire(AireAcondicionado aire) {
        this.aire = aire;
    }

    @Override
    public void execute() {
        aire.encender();
    }

    @Override
    public void undo() {
        aire.apagar();
    }
}
```

```
/*****************************************************************************
* NOMBRE: ApagarAire.java
* DESCRIPCIÓN: Implementa Command para apagar un Aire Acondicionado.
*****************************************************************************/
class ApagarAire implements Command {
    private AireAcondicionado aire;

    public ApagarAire(AireAcondicionado aire) {
        this.aire = aire;
    }

    @Override
    public void execute() {
        aire.apagar();
    }

    @Override
    public void undo() {
        aire.encender();
    }
}
```

```
/*********************************************************************************
* NOMBRE: ControlRemoto.java
* DESCRIPCIÓN: Maneja la configuración y ejecución de comandos.
*********************************************************************************/
class ControlRemoto {
    private Command ultimoComando;

    public void setCommand(Command command) {
        this.ultimoComando = command;
    }

    public void presionarBoton() {
        if (ultimoComando != null) {
            ultimoComando.execute();
        }
    }

    public void deshacerBoton() {
        if (ultimoComando != null) {
            ultimoComando.undo();
        }
    }
}
```

```
/*********************************************************************************
* NOMBRE: SistemaControlDispositivos.java
* DESCRIPCIÓN: Clase principal para probar el sistema de control de dispositivos.
*********************************************************************************/
public class SistemaControlDispositivos {
    public static void main(String[] args) {
        //dispositivos
        Luz luz = new Luz();
        Ventilador ventilador = new Ventilador();
        AireAcondicionado aire = new AireAcondicionado();

        // comandos
        Command encenderLuz = new EncenderLuz(luz);
        Command apagarLuz = new ApagarLuz(luz);
        Command encenderVentilador = new EncenderVentilador(ventilador);
        Command apagarVentilador = new ApagarVentilador(ventilador);
        Command encenderAire = new EncenderAire(aire);
        Command apagarAire = new ApagarAire(aire);

        // crear control remoto
        ControlRemoto control = new ControlRemoto();

        // -> Pruebas
        control.setCommand(encenderLuz);
        control.presionarBoton();   //Luz encendida
```

```
        control.deshacerBoton();    //Luz apagada

        control.setCommand(encenderVentilador);
        control.presionarBoton();   //Ventilador encendido
        control.deshacerBoton();    //Ventilador apagado

        control.setCommand(encenderAire);
        control.presionarBoton();   //Aire acondicionado encendido
        control.deshacerBoton();    //Aire acondicionado apagado
    }
}
```

EXPLICACIÓN:

Interfaz Command:
Define dos métodos: execute() para ejecutar la acción y undo() para deshacerla.

Clases Luz, Ventilador, AireAcondicionado:
Estas clases representan los dispositivos que se pueden encender o apagar.

Comandos concretos (como EncenderLuz, ApagarLuz, etc.):
Implementan la interfaz Command y encapsulan las acciones específicas para cada dispositivo.

EncenderLuz: Ejecuta el encendido de la luz y deshace apagándola.
ApagarLuz: Ejecuta el apagado de la luz y deshace encendiéndola.
-lo mismo para Ventilador y AireAcondicionado.
ControlRemoto:
Almacena el último comando ejecutado y permite ejecutar presionarBoton o deshacer

SALIDA

```
Luz encendida
Luz apagada
Ventilador encendido
Ventilador apagado
Aire acondicionado encendido
Aire acondicionado apagado
```

3. CUESTIONARIO

1. **¿Cuál es la ventaja principal de usar el patrón Observer en sistemas con múltiples dependencias?**

   La ventaja principal de usar el patrón Observer en sistemas con múltiples dependencias es que permite un desacoplamiento entre objetos que interactúan entre sí. El sujeto (objeto observado) no necesita conocer los detalles de sus observadores, lo que facilita añadir o eliminar observadores sin modificar el sujeto. Esto mejora la flexibilidad y escalabilidad del sistema.

2. **¿Qué problema resuelve el patrón Strategy en comparación con el uso de condicionales como if-else?**

   El patrón Strategy resuelve el problema de tener múltiples condicionales (if-else) para seleccionar diferentes comportamientos o algoritmos. En lugar de usar condicionales, encapsula cada algoritmo en una clase separada, permitiendo intercambiarlos dinámicamente. Esto mejora la mantenibilidad del código y facilita la adición de nuevos algoritmos sin modificar el código existente.

3. **Explica cómo el patrón Command ayuda a desacoplar la lógica de ejecución de las operaciones.**

   El patrón Command ayuda a desacoplar la lógica de ejecución de las operaciones al encapsular una solicitud como un objeto. Esto separa el objeto que invoca la operación (invocador) del objeto que sabe cómo llevarla a cabo (receptor). El invocador no necesita conocer los detalles de cómo se ejecuta la operación, solo necesita conocer la interfaz del comando.

4. **¿Qué métodos claves se deben implementar para crear un patrón Observer funcional en Java?**

   Para crear un patrón Observer funcional en Java, se deben implementar los siguientesmétodos clave:

   - En la interfaz Observer: update()

   - En la clase Subject: attach(), detach(), y notifyObservers()

5. **En el patrón Strategy, ¿Cómo se puede cambiar dinámicamente la estrategia utilizada?**

   En el patrón Strategy, se puede cambiar dinámicamente la estrategia utilizada mediante un método setter en el contexto. Por ejemplo, setStrategy (Strategy newStrategy). Esto permite al cliente cambiar la estrategia en tiempo de ejecución.

6. **Menciona tres casos de uso donde el patrón Command sería especialmente útil.**

   Tres casos de uso donde el patrón Command sería especialmente útil son:

   - Implementación de operaciones de deshacer/rehacer en editores de texto

   - Programación de tareas en sistemas de procesamiento por lotes

   - Implementación de menús y botones en interfaces gráficas de usuario

7. **¿Cómo puedes combinar los patrones Observer, Strategy y Command en una aplicación más compleja?**

   Se pueden combinar los patrones Observer, Strategy y Command en una aplicación más compleja de la siguiente manera:

   - Usar Observer para notificar cambios en el estado de la aplicación

   - Utilizar Strategy para implementar diferentes algoritmos o comportamientos

   - Emplear Command para encapsular acciones que pueden ser ejecutadas, deshechas o registradas

8. **¿Qué ventajas tiene encapsular algoritmos o acciones en clases independientes desde el punto de vista del diseño de software?**

   Encapsular algoritmos o acciones en clases independientes tiene varias ventajas desde el punto de vista del diseño de software:

   - Mejora la modularidad y la reutilización del código

   - Facilita la extensión y modificación de comportamientos sin afectar el código existente

   - Promueve el principio de responsabilidad única, haciendo que cada clase tenga una única razón para cambiar

9. **En el patrón Command, ¿cómo se implementaría la funcionalidad de "deshacer"?**

   En el patrón Command, la funcionalidad de "deshacer" se puede implementar añadiendo un método unexecute() o undo() a la interfaz Command. Cada comando concreto implementaría este método para revertir los efectos de su método execute(). Además, se mantendría una pila de comandos ejecutados para poder deshacer las operaciones en orden inverso.

10. Reflexiona sobre cómo los patrones de diseño promueven el principio de responsabilidad única.

Los patrones de diseño promueven el principio de responsabilidad única al encapsular comportamientos específicos en clases separadas. Cada patrón define roles claros para las clases participantes, asegurando que cada clase tenga una única razón para cambiar. Por ejemplo, en el patrón Strategy, cada estrategia concreta tiene la única responsabilidad de implementar un algoritmo específico. Esto mejora la cohesión de las clases y facilita el mantenimiento y la evolución del software.

4. BIBLIOGRAFÍA

Chuck's Academy. (s.f.). Patrón de diseño Observer.
https://www.chucksacademy.com/es/topic/javascript-design-patterns/observer-pattern

YouTube. (2020). Patrón Observer en JavaScript.
https://www.youtube.com/watch?v=cBdKQqIjXkk

YouTube. (2020). Patrón Observer en JavaScript.
https://www.youtube.com/watch?v=ZqkWeVxmQpc

JavaTutoriales. (2022). Patrón de diseño Strategy.
https://www.javatutoriales.com/2022/01/patron-de-diseno-strategy.html

García, D. (2014). Patrones de comportamiento II: Patrón Command.
https://danielggarcia.wordpress.com/2014/04/28/patrones-de-comportamiento-ii-patroncommand/

IONOS. (s.f.). ¿Qué es el patrón Observer?
https://www.ionos.com/es-us/digitalguide/paginas-web/desarrollo-web/que-es-el-patronobserver/

LaravelTip. (s.f.). Elimina sentencias if-else y switch con el patrón Estrategia.
https://www.laraveltip.com/elimina-sentencias-if-else-y-switch-con-el-patron-estrategia/

Welcome Developers. (s.f.). El patrón Command.
https://welcomedevelopers.es/patrones-diseno/el-patron-command/

KeepCoding. (2021). Patrón Observer y cómo se usa.
https://keepcoding.io/blog/patron-observer-y-como-se-usa/