UCSM Esc. Prof. de Ingeniería de Sistemas

**Diciembre - 2024**

INFORME DE PRÁCTICAS

# LENGUAJES DE PROGRAMACIÓN III

## Práctica N° 11

Elaborado por:

SUCASAIRE CUEVA, ARNOLD SAMUEL

# GRUPO N° 05

# PRÁCTICAS DE LENGUAJES DE PROGRAMACIÓN III

---

Presentado por:
73780984 SUCASAIRE CUEVA, ARNOLD SAMUEL 100%

## RECONOCIMIENTOS

Los autores de este trabajo reconocen con gratitud 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 interactuamos 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

Modular Architecture, Database Management, Data Integrity, Security Measures, Transactional Operations, User Interface

# ÍNDICE

## Contenido

# 1. RESUMEN

Este informe explora la implementación de bases de datos SQLite y la conectividad JDBC en Java. Presenta los conceptos esenciales de bases de datos relacionales y SQL, además de los pasos requeridos para establecer conexiones, ejecutar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) y gestionar transacciones en Java. La guía enfatiza el uso de la interfaz PreparedStatement para consultas parametrizadas, mejorando así el rendimiento y la seguridad de las aplicaciones. Finalmente, se desarrolla una aplicación en Java que aplica estos principios mediante operaciones de manipulación de datos y administración de transacciones.

# 2. INTRODUCCIÓN

Las bases de datos son estructuras organizadas que permiten el almacenamiento, acceso y manipulación eficiente de datos de múltiples usuarios. Los sistemas de gestión de bases de datos relacionales, como SQLite, emplean SQL como lenguaje de consulta estándar, lo que posibilita realizar consultas estructuradas y manipular datos de manera eficiente. En este contexto, JDBC (Java Database Connectivity) permite a las aplicaciones Java interactuar con bases de datos, facilitando el almacenamiento y la recuperación de información. Este informe detalla el uso de SQLite y JDBC en un entorno de desarrollo Java, resaltando la importancia de la gestión de transacciones y consultas preparadas para el desarrollo de aplicaciones seguras y optimizadas.

# 3. MARCO TEÓRICO

## 3.1 Introducción a las Bases de Datos Relacionales

Una base de datos es una colección organizada de datos, diseñada para almacenar, gestionar y facilitar el acceso y manipulación de información. Las bases de datos relacionales (RDBMS por sus siglas en inglés) permiten estructurar y relacionar los datos de manera eficiente a través del modelo relacional, que emplea el lenguaje SQL para realizar operaciones de consulta, modificación y control de datos.

## 3.2 Modelo Relacional

El modelo relacional se basa en la organización de los datos en tablas (también conocidas como relaciones), donde cada tabla representa una entidad y sus atributos. Las relaciones entre tablas permiten conectar datos de manera estructurada, evitando la duplicidad de información y promoviendo la integridad de los datos. El modelo entidad-relación es fundamental en este contexto, permitiendo la representación gráfica y conceptual de las relaciones entre los datos mediante entidades, atributos y relaciones con cardinalidad.

## 3.3 Lenguaje SQL (Structured Query Language)

SQL es el lenguaje estándar para manipular bases de datos relacionales. Permite crear, modificar y consultar las tablas dentro de una base de datos. Las operaciones básicas incluyen:

- Creación de Tablas: La sentencia CREATE TABLE permite definir nuevas tablas con sus respectivos campos, tipos de datos, restricciones y claves primarias.
- Inserción de Datos: La sentencia INSERT INTO permite agregar registros a una tabla.
- Actualización de Datos: Con UPDATE, se pueden modificar datos existentes en función de condiciones específicas.
- Consulta de Datos: SELECT permite obtener datos de las tablas aplicando filtros, ordenamientos y agregaciones.
- Creación de Vistas e Índices: Las vistas proporcionan una representación lógica de los datos almacenados, mientras que los índices mejoran la velocidad de las consultas.

## 3.4 JDBC (Java Database Connectivity)

JDBC es una API de Java que permite la comunicación con una base de datos de manera independiente del tipo de gestor utilizado. JDBC facilita la ejecución de comandos SQL desde una aplicación Java y proporciona las siguientes funcionalidades clave:

- Conexión a la Base de Datos: A través del método DriverManager.getConnection, se establece una conexión utilizando una URL específica para el tipo de base de datos.
- Creación de Sentencias: Con createStatement y PreparedStatement, JDBC permite ejecutar consultas SQL y comandos parametrizados, optimizando el rendimiento y la seguridad al prevenir inyecciones SQL.
- Manejo de Resultados: La interfaz ResultSet permite recorrer los datos retornados por una consulta, permitiendo su procesamiento en Java.
- Manejo de Transacciones: JDBC permite gestionar transacciones, cumpliendo con las propiedades ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad). Con métodos como commit y rollback, el desarrollador puede asegurar que las operaciones sean ejecutadas de manera confiable.

## 3.5 Uso de SQLite con JDBC en Java

SQLite es una base de datos ligera, ideal para aplicaciones pequeñas o donde no se requieren grandes volúmenes de datos. Con SQLite, se puede realizar un manejo completo de las operaciones SQL básicas, como creación de tablas, inserción de registros, actualizaciones y eliminación de datos. JDBC facilita la integración de SQLite con aplicaciones Java, permitiendo realizar las siguientes acciones:

- Conexión: A través de la clase de controlador org.sqlite.JDBC y la URL jdbc:sqlite:, se establece la conexión con la base de datos.
- PreparedStatement: Esta interfaz permite ejecutar consultas parametrizadas, mejorando la eficiencia al compilar la consulta solo una vez y facilitar la reutilización de consultas similares.

- Transacciones: La administración de transacciones en JDBC con SQLite permite agrupar varias operaciones para que se realicen de forma conjunta. Esto asegura la integridad de los datos y permite mejorar el rendimiento al procesar múltiples operaciones antes de aplicar cambios permanentes a la base de datos.

# 4. EXPERIENCIAS DE PRÁCTICA

## 4.1 Implementa tres (3) ejercicios funcionales de los patrones estudiados con las siguientes características:

### 4.1.1 Observer: Un sistema de notificaciones para múltiples usuarios.

```
package experiencia;

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

// Interface para los observadores
interface Observer {
    void update(String message);
}

// Clase concreta del sujeto
class NotificationService {
    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer observer) {
        observers.add(observer);
    }

    public void unsubscribe(Observer observer) {
        observers.remove(observer);
    }

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

// Clase concreta de los observadores
class User implements Observer {
    private String name;

    public User(String name) {
        this.name = name;
```

```
    }

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

// Main
public class ObserverPattern {
    public static void main(String[] args) {
        NotificationService service = new NotificationService();

        User user1 = new User("Samuel");
        User user2 = new User("Aaron");

        service.subscribe(user1);
        service.subscribe(user2);

        service.notifyObservers("Nuevo mensaje disponible.");
    }
}
```

**Pantalla**

```
Usuario Samuel recibió notificación: Nuevo mensaje disponible.
Usuario Aaron recibió notificación: Nuevo mensaje disponible.
PS E:\UCSM\4. IV Semestre\Lenguaje de Programación - 05\Practica 11>
```

**Explicación:**

- Define un contrato (update) que los observadores deben implementar para recibir notificaciones.
- Actúa como el sujeto, manteniendo una lista de observadores y notificándolos de los eventos.
- Permite agregar observadores a la lista para que reciban notificaciones.
- Recorre la lista de observadores y llama a su método update, pasando el mensaje.
- Implementa la interfaz Observer y representa a los usuarios que recibirán notificaciones.
- Define cómo los usuarios reaccionan al recibir una notificación, en este caso, mostrando un mensaje en consola.
- Combina los patrones Observador, Estrategia y Comando en un solo programa.
- Se crea una instancia de NotificationService y se suscriben usuarios (User) como observadores.
- Cuando ocurre un evento, como aplicar una promoción, se llama a notifyObservers para informar a todos los suscriptores.
- Se aplica una estrategia de promoción (PercentageDiscount) a un producto y se notifica a los usuarios.
- Simplifica la ejecución del comando turnOn para la televisión.
- Promueve el desacoplamiento entre el sujeto y los observadores, permitiendo la extensión del sistema sin modificar las clases existentes.

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

```
package experiencia;

// Interface para estrategias
interface PromotionStrategy {
    double applyPromotion(double price);
}

// Estrategia concreta: Descuento porcentual
class PercentageDiscount implements PromotionStrategy {
    private double discountRate;

    public PercentageDiscount(double discountRate) {
        this.discountRate = discountRate;
    }

    @Override
    public double applyPromotion(double price) {
        return price - (price * discountRate);
    }
}

// Estrategia concreta: Descuento fijo
class FixedDiscount implements PromotionStrategy {
    private double discountAmount;

    public FixedDiscount(double discountAmount) {
        this.discountAmount = discountAmount;
    }

    @Override
    public double applyPromotion(double price) {
        return price - discountAmount;
    }
}

// Contexto
class Product {
    private String name;
    private double price;
    private PromotionStrategy strategy;

    public Product(String name, double price, PromotionStrategy strategy) {
        this.name = name;
        this.price = price;
        this.strategy = strategy;
    }
```

```
    public void applyPromotion() {
        double newPrice = strategy.applyPromotion(price);
        System.out.println("El producto " + name + " ahora cuesta: " + newPrice);
    }
}

// Main
public class StrategyPattern {
    public static void main(String[] args) {
        Product product1 = new Product("Laptop", 1000, new
PercentageDiscount(0.2));
        Product product2 = new Product("Celular", 500, new FixedDiscount(50));

        product1.applyPromotion();
        product2.applyPromotion();
    }
}
```

**Pantalla**

```
El producto Laptop ahora cuesta: 800.0
El producto Celular ahora cuesta: 450.0
PS E:\UCSM\4. IV Semestre\Lenguaje de Programación - 05\Practica 11>
```

**Explicación:**

- Define un contrato (applyPromotion) para implementar diferentes estrategias de promoción. Esto permite aplicar distintas lógicas de descuentos de forma intercambiable.
- Implementa la interfaz PromotionStrategy y calcula el descuento en porcentaje basado en la tasa proporcionada.
- También implementa PromotionStrategy pero aplica un descuento fijo al precio base.
- Actúa como el contexto, almacenando información del producto y utilizando una estrategia (PromotionStrategy) para calcular el precio promocional.
- Permite asignar un nombre, precio y una estrategia de promoción para cada instancia de producto.
- Calcula y muestra el nuevo precio del producto al aplicar la estrategia seleccionada.
- Las estrategias (PercentageDiscount, FixedDiscount) son intercambiables gracias a la interfaz PromotionStrategy.
- Contiene el método main para ejecutar el programa.
- Se crean con diferentes estrategias: un descuento porcentual (20%) y un descuento fijo (S/50).
- Llama al método applyPromotion de cada producto para calcular el precio final basado en la estrategia definida.
- Es fácil agregar nuevas estrategias de promoción sin modificar el código existente, respetando el principio de abierto/cerrado.
- Muestra en consola los precios actualizados tras aplicar las promociones, demostrando cómo se pueden usar estrategias distintas con la misma interfaz.

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

```
package experiencia;

// Interface Command
interface Command {
    void execute();
}

// Receptor
class Television {
    public void turnOn() {
        System.out.println("Televisión encendida.");
    }

    public void turnOff() {
        System.out.println("Televisión apagada.");
    }

    public void increaseVolume() {
        System.out.println("Volumen aumentado.");
    }

    public void decreaseVolume() {
        System.out.println("Volumen disminuido.");
    }

    public void changeChannel(int channel) {
        System.out.println("Canal cambiado a: " + channel);
    }
}

// Comandos concretos
class TurnOnCommand implements Command {
    private Television tv;

    public TurnOnCommand(Television tv) {
        this.tv = tv;
    }

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

class TurnOffCommand implements Command {
    private Television tv;
```

```
    public TurnOffCommand(Television tv) {
        this.tv = tv;
    }

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

// Main
public class CommandPattern {
    public static void main(String[] args) {
        Television tv = new Television();

        Command turnOn = new TurnOnCommand(tv);
        Command turnOff = new TurnOffCommand(tv);

        turnOn.execute();
        turnOff.execute();
    }
}
```

Pantalla

```
Televisión encendida.
Televisión apagada.
PS E:\UCSM\4. IV Semestre\Lenguaje de Programación - 05\Practica 11>
```

Explicación:

- Define un contrato (execute) que encapsula una operación en un objeto independiente.
- Representa el receptor del comando. Proporciona métodos como turnOn y turnOff para realizar operaciones.
- Implementa la interfaz Command y encapsula la acción de encender la televisión llamando a su método turnOn.
- También implementa Command y encapsula la acción de apagar la televisión llamando a su método turnOff.
- Cada comando concreto almacena una referencia al objeto receptor (Television) que ejecutará la acción.
- Cada comando concreto define el comportamiento específico de la operación.
- Contiene el método main para crear instancias y ejecutar comandos.
- Se instancia una televisión (Television) que servirá como receptor de las órdenes.
- Se crean objetos que encapsulan las acciones de encender y apagar la televisión.
- Los comandos se ejecutan llamando a su método execute, lo que desencadena las acciones correspondientes en el receptor.

El cliente no interactúa directamente con el receptor, sino a través de los comandos, facilitando la reutilización y desacoplamiento.
Es fácil agregar nuevos comandos sin modificar el código existente. También permite deshacer/rehacer operaciones al extender la interfaz Command.

## 4.2 Crea una aplicación adicional que combine los tres patrones:

```
package experiencia;

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

// Observer
interface Observer {
    void update(String message);
}

class NotificationService {
    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer observer) {
        observers.add(observer);
    }

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

class User implements Observer {
    private String name;

    public User(String name) {
        this.name = name;
    }

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

// Strategy
interface PromotionStrategy {
```

```
    double applyPromotion(double price);
}

class PercentageDiscount implements PromotionStrategy {
    private double discountRate;

    public PercentageDiscount(double discountRate) {
        this.discountRate = discountRate;
    }

    @Override
    public double applyPromotion(double price) {
        return price - (price * discountRate);
    }
}

class Product {
    private String name;
    private double price;
    private PromotionStrategy strategy;

    public Product(String name, double price, PromotionStrategy strategy) {
        this.name = name;
        this.price = price;
        this.strategy = strategy;
    }

    public void applyPromotion() {
        double newPrice = strategy.applyPromotion(price);
        System.out.println(name + " nuevo precio: " + newPrice);
    }
}

// Command
interface Command {
    void execute();
}

class Television {
    public void turnOn() {
        System.out.println("Televisión encendida.");
    }
}

// Main combinado
public class CombinedPatterns {
    public static void main(String[] args) {
        // Observer
```

```
        NotificationService service = new NotificationService();
        User user1 = new User("Samuel");
        service.subscribe(user1);

        // Strategy
        Product product = new Product("Laptop", 1000, new PercentageDiscount(0.2));
        product.applyPromotion();

        // Command
        Television tv = new Television();
        Command turnOn = () -> tv.turnOn();
        turnOn.execute();

        // Notificación
        service.notifyObservers("Promoción aplicada.");
    }
}
```

Pantalla

```
El producto Laptop ahora cuesta: 800.0
Televisión encendida.
Usuario Samuel recibió notificación: Promoción aplicada.
PS E:\UCSM\4. IV Semestre\Lenguaje de Programación - 05\Practica 11>
```

explicación

- Interfaz Observer, define el contrato para los observadores con el método update(String message), que es llamado para notificar cambios.
- Clase NotificationService, mantiene una lista de observadores (List<Observer>).
- Permite a los observadores suscribirse mediante el método subscribe.
- Notifica a todos los observadores suscritos usando notifyObservers(String message).
- Implementa la interfaz Observer. Su método update imprime un mensaje personalizado con el nombre del usuario y la notificación recibida.
- Se añade un usuario como observador (en este caso, "Samuel")
- Cuando se ejecuta notifyObservers, el observador recibe la notificación y la procesa.
- Define un contrato para aplicar diferentes estrategias de promoción con el método applyPromotion(double price).
- Implementa PromotionStrategy y aplica un descuento porcentual al precio recibido.
- Almacena un nombre, precio y una estrategia de promoción.
- Su método applyPromotion utiliza la estrategia definida para calcular y mostrar el nuevo precio.
- Se crea un producto llamado "Laptop" con un precio de 1000.
- Se aplica un descuento del 20% utilizando PercentageDiscount.
- Define el método execute() como el contrato para ejecutar acciones encapsuladas.
- Contiene una acción (turnOn) que representa encender la televisión.
- Se puede extender para incluir más funcionalidades, como cambiar de canal o ajustar

el volumen.
- Se utiliza una expresión lambda para definir el comando turnOn y ejecutarlo llamando al método execute.

# 5. EJERCICIOS DE PROPUESTO

## 5.1 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.

```
package ejercicios;

// Importación necesaria para List y ArrayList
import java.util.List;
import java.util.ArrayList;

// Interfaz Observer
interface Observer {
    void update(String message);
}

// Clase Sujeto (Sistema de Notificaciones)
class NotificationSystem {
    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer observer) {
        observers.add(observer);
    }

    public void unsubscribe(Observer observer) {
        observers.remove(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

```java
// Clase Usuario (Observador)
class User implements Observer {
    private String name;

    public User(String name) {
        this.name = name;
    }

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

// Main
public class NotificationObserver {
    public static void main(String[] args) {
        NotificationSystem system = new NotificationSystem();

        User user1 = new User("Samuel");
        User user2 = new User("Aaron");

        system.subscribe(user1);
        system.subscribe(user2);

        system.notifyObservers("Nueva promoción: ¡50% de descuento!");

        system.unsubscribe(user1);

        system.notifyObservers("Actualización de producto: ¡Nuevas laptops
disponibles!");
    }
}
```

**Pantalla:**

```
Samuel recibió notificación: Nueva promoción: ¡50% de descuento!
Aaron recibió notificación: Nueva promoción: ¡50% de descuento!
Aaron recibió notificación: Actualización de producto: ¡Nuevas laptops
disponibles!
PS E:\UCSM\4. IV Semestre\Lenguaje de Programación - 05\Practica 11>
```

**Explicación:**

- Define el método update(String message) para que los observadores reciban notificaciones del sujeto.
- Actúa como el "sujeto" que mantiene una lista de observadores y gestiona las suscripciones.Método subscribe: Permite a los observadores (usuarios) suscribirse para recibir notificaciones.

- Permite a los observadores dejar de recibir notificaciones.
- Envía un mensaje a todos los observadores suscritos invocando su método update.
- Representa a los usuarios que implementan la interfaz Observer para recibir notificaciones personalizadas.
- Imprime el mensaje de notificación recibido, identificando al usuario.
- Los usuarios (Samuel y Aaron) se suscriben al sistema de notificaciones.
- Se envía una notificación sobre una promoción, que ambos usuarios reciben.
- Samuel se elimina de la lista de suscriptores usando unsubscribe.
- Solo Aaron recibe la segunda notificación, demostrando que Samuel ha sido desuscrito.
- Los usuarios pueden suscribirse o desuscribirse dinámicamente.

## 5.2 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.

```
package ejercicios;


// Interfaz Strategy
interface DiscountStrategy {
    double applyDiscount(double price, int quantity);
}

// Estrategia SinDescuento
class NoDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price, int quantity) {
        return price * quantity;
    }
}

// Estrategia DescuentoFijo
class FixedDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price, int quantity) {
        return (price * quantity) * 0.9; // 10% de descuento
    }
}

// Estrategia DescuentoPorcentual
class PercentageDiscount implements DiscountStrategy {
```

```
    @Override
    public double applyDiscount(double price, int quantity) {
        return quantity >= 2 ? (price * quantity) * 0.7 : price * quantity;
// 30% para 2+
    }
}

// Estrategia DescuentoPorcentualAcumulado
class AccumulatedDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price, int quantity) {
        if (quantity >= 3) {
            return (price * (quantity - 1)) + (price * 0.5); // 50% en el
más barato
        }
        return price * quantity;
    }
}

// Clase Producto y Calculadora
class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public double getPrice() {
        return price;
    }

    public String getName() {
        return name;
    }
}

// Main
public class DiscountStrategyDemo {
    public static void main(String[] args) {
        Product product = new Product("Celular", 500);

        DiscountStrategy noDiscount = new NoDiscount();
        DiscountStrategy fixedDiscount = new FixedDiscount();
        DiscountStrategy percentageDiscount = new PercentageDiscount();
        DiscountStrategy accumulatedDiscount = new AccumulatedDiscount();
```

```
        System.out.println("Sin descuento: " +
noDiscount.applyDiscount(product.getPrice(), 1));
        System.out.println("Descuento fijo: " +
fixedDiscount.applyDiscount(product.getPrice(), 2));
        System.out.println("Descuento porcentual: " +
percentageDiscount.applyDiscount(product.getPrice(), 2));
        System.out.println("Descuento acumulado: " +
accumulatedDiscount.applyDiscount(product.getPrice(), 3));
    }
}
```

**Pantalla:**

```
Sin descuento: 500.0
Descuento fijo: 900.0
Descuento porcentual: 700.0
Descuento acumulado: 1250.0
PS E:\UCSM\4. IV Semestre\Lenguaje de Programación - 05\Practica 11>
```

**Explicación:**

- Define el método applyDiscount para aplicar estrategias de descuento personalizadas.
- Aplica el precio completo sin descuentos.
- Aplica un descuento fijo del 10% sobre el total.
- Aplica un 30% de descuento si se compran al menos 2 productos.
- Aplica un 50% de descuento en el producto más barato cuando hay 3 o más.
- Representa un producto con nombre y precio.
- Uso de estrategias: Se crean instancias de las estrategias y se utilizan según la necesidad.
- Calcula el precio total sin modificaciones.
- Reduce el total en un 10%.
- Aplica un 30% si se cumplen las condiciones de cantidad.
- Calcula descuentos complejos según las reglas.
- Menú interactivo: Facilita la elección de estrategias dinámicamente.

## 5.3 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 ejecutarcomandos dinámicamente.
- Agrega la funcionalidad de deshacer la última acción realizada

```
package ejercicios;
```

```
// Interfaz Command
interface Command {
    void execute();
    void undo();
}

// Clase Luz
class Light {
    public void turnOn() {
        System.out.println("Luz encendida.");
    }

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

// Clase Ventilador
class Fan {
    public void turnOn() {
        System.out.println("Ventilador encendido.");
    }

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

// Comandos concretos
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }

    @Override
    public void undo() {
        light.turnOff();
    }
}

// Clase ControlRemoto
```

```
class RemoteControl {
    private Command lastCommand;

    public void setCommand(Command command) {
        lastCommand = command;
    }

    public void pressButton() {
        lastCommand.execute();
    }

    public void pressUndo() {
        lastCommand.undo();
    }
}

// Main
public class CommandPatternDemo {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOn = new LightOnCommand(light);

        RemoteControl remote = new RemoteControl();
        remote.setCommand(lightOn);
        remote.pressButton(); // Encender luz
        remote.pressUndo();   // Apagar luz
    }
}
```

Pantalla:

```
Luz encendida.
Luz apagada.
PS E:\UCSM\4. IV Semestre\Lenguaje de Programación - 05\Practica 11>
```

Explicación:

- Define los métodos execute y undo para encapsular acciones.
- Representa un dispositivo con métodos para encender y apagar.
- Similar a Light, representa otro dispositivo controlable.
- Implementa Command y encapsula la lógica de encender/apagar luces.
- Activa la acción principal (encender).
- Deshace la acción principal (apagar).
- Permite configurar comandos dinámicamente.
- Establece el comando actual para el control remoto.
- Ejecuta el comando configurado.
- Llama al método undo del comando actual.
- Cambia comandos según el dispositivo controlado.
- Encender/apagar luces demuestra la funcionalidad básica del patrón.

# 6. CONCLUSIONES DE LA PRÁCTICA:

- Se implementó exitosamente un sistema de notificaciones que permite a los usuarios suscribirse y recibir actualizaciones automáticamente.
- A través del patrón Observer, se comprendió cómo separar la lógica del emisor (sujeto) de la lógica de los receptores (observadores). Esto es esencial en sistemas donde los datos deben propagarse a múltiples componentes en tiempo real.
- Se diseñaron y aplicaron varias estrategias de descuento (sin descuento, descuento fijo, descuento porcentual y descuento acumulado).
- Con el patrón Strategy, se verificó la facilidad con la que se pueden agregar o modificar comportamientos en tiempo de ejecución, lo cual es ideal para aplicaciones con requisitos cambiantes.
- Se implementó un control remoto que utiliza comandos específicos para operar dispositivos como luces, ventiladores y aire acondicionado.
- El patrón Command demostró cómo separar claramente las solicitudes (comandos) de la lógica que las ejecuta (dispositivos). Esto simplifica la expansión del sistema al permitir agregar nuevos comandos o dispositivos sin afectar el código existente.
- Los patrones implementados se ajustaron a escenarios reales, lo que permitió comprender su impacto práctico en el diseño y mantenimiento de sistemas de software.
- Los patrones facilitaron la reutilización de componentes. Por ejemplo, los dispositivos del patrón Command o las estrategias de descuento del patrón Strategy se pueden reutilizar en otros sistemas con poca o ninguna modificación.
- La implementación de los patrones promovió una mejor organización del código, con clases y responsabilidades claramente definidas. Esto no solo mejoró la legibilidad, sino también la facilidad para realizar pruebas y depuración.
- Al usar paquetes como ejercicios, se aprendió a estructurar proyectos en módulos lógicos, facilitando la navegación y administración del código en proyectos de mayor escala.
- Durante la práctica, se profundizó en el uso de herramientas como Visual Studio Code, gestionando configuraciones de proyectos Java, organización de archivos y depuración de errores comunes relacionados con paquetes y clases.
- La implementación de patrones de diseño proporcionó una base sólida para desarrollar sistemas complejos, donde los requisitos cambian frecuentemente o donde múltiples componentes deben interactuar entre sí.

# 7. CUESTIONARIO:

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

La ventaja principal es que el patrón Observer permite desacoplar el sujeto (emisor) de sus observadores (receptores), lo que facilita la actualización automática de los receptores en respuesta a cambios en el estado del sujeto. Esto mejora la escalabilidad, ya que se pueden agregar o eliminar observadores sin modificar el código base del sujeto.

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

El patrón Strategy elimina la complejidad de los bloques de if-else o switch, encapsulando diferentes comportamientos (estrategias) en clases separadas. Esto permite agregar nuevas estrategias sin modificar el código existente, promoviendo el principio de Open/Closed (abierto para extensión, cerrado para modificación).

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

El patrón Command encapsula una solicitud (acción) como un objeto, separando al cliente que realiza la solicitud de la lógica que la ejecuta. Esto permite:

- Reutilizar comandos en diferentes contextos.
- Programar o almacenar acciones para ejecutarlas más tarde.
- Deshacer operaciones fácilmente.

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

En el sujeto (Observable):

- subscribe(Observer observer): Para agregar observadores.
- unsubscribe(Observer observer): Para eliminar observadores.
- notifyObservers(String message): Para notificar a todos los observadores registrados.

En los observadores:

- update(String message): Método que define la lógica de actualización para cada observador.

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

Se puede cambiar dinámicamente la estrategia mediante un método setter en la clase que usa la estrategia (contexto). Por ejemplo:

```
public void setStrategy(PromotionStrategy strategy) {
   this.strategy = strategy;
}
```

Esto permite reemplazar la estrategia en tiempo de ejecución sin modificar el código base.

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

- Sistemas de control remoto: Operar dispositivos como luces, ventiladores, o televisores.
- Editores de texto: Implementar funcionalidades como deshacer/rehacer.
- Sistemas de colas de tareas: Almacenar y ejecutar operaciones programadas o diferidas.

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

Se pueden combinar para crear un sistema integral. Por ejemplo:

- Un sistema de notificaciones (Observer) donde los usuarios reciben actualizaciones en tiempo real.
- Las notificaciones pueden utilizar diferentes estrategias de priorización (Strategy) para determinar el orden de envío.
- Las acciones de notificación (enviar, retrasar o cancelar) pueden encapsularse como comandos (Command) para permitir un control más flexible y la posibilidad de deshacer acciones.

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

- Reutilización: Los algoritmos o acciones pueden reutilizarse en otros contextos.
- Mantenibilidad: Modificar o reemplazar un algoritmo no afecta otras partes del sistema.
- Flexibilidad: Permite cambiar comportamientos dinámicamente.
- Testabilidad: Es más fácil probar componentes individuales de manera aislada.

**I. <u>En el patrón Command, ¿cómo se implementaría la funcionalidad de "deshacer"?</u>**

Se puede implementar manteniendo un historial de comandos ejecutados. Cada comando debe tener un método undo() que revierta la acción realizada. Por ejemplo:

```
interface Command {
   void execute();
   void undo();
}
```

Luego, el cliente puede almacenar los comandos en una pila y llamar a undo() en el último comando ejecutado.

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

Los patrones de diseño ayudan a dividir el sistema en componentes con responsabilidades claras y específicas. Por ejemplo:

- En Observer, el sujeto solo gestiona su estado, mientras que los observadores manejan su propia lógica de actualización.
- En Strategy, cada estrategia se encarga de un comportamiento específico, como calcular un tipo de descuento.
- En Command, los comandos encapsulan una sola acción, simplificando la lógica del cliente.

Esto reduce la complejidad, facilita el mantenimiento y mejora la extensibilidad del sistema.

## 8. REFERENCIAS:

*Oracle. (2023). Java Database Connectivity (JDBC) Overview.* *https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/*

*SQLite. (2023). About SQLite. https://www.sqlite.org/about.html*

*Baeldung. (2023). Guide to JDBC. https://www.baeldung.com/java-jdbc*

*GeeksforGeeks. (2023). Difference between Statement and PreparedStatement in Java JDBC.* *https://www.geeksforgeeks.org*

## 9. ENLACES:

Github: https://github.com/samuelSC97/LenguajeDeProgramacionIII