Semana 12: Patrones de Comportamiento

# ¿Qué son los Patrones de Comportamiento?

Los **patrones de comportamiento** son un conjunto de soluciones de diseño que abordan **cómo interactúan los objetos entre sí** en un sistema orientado a objetos.  
Se centran en **la distribución de responsabilidades, comunicación, flujo de control y colaboración entre componentes**, promoviendo el bajo acoplamiento, alta cohesión y reutilización del comportamiento.

## ¿Por qué son importantes?

1. **Desacoplan lógica y responsabilidades**  
   Permiten que objetos cooperen sin depender unos de otros directamente, haciendo que los sistemas sean **más flexibles y mantenibles**.
2. **Controlan el flujo de ejecución**  
   Facilitan la implementación de **sistemas complejos con múltiples pasos, decisiones y estados**, como flujos de aprobación, motores de reglas o procesos de negocio.
3. **Simplifican la extensibilidad**  
   Muchos patrones de comportamiento están diseñados para **permitir nuevas operaciones o estados** sin modificar el código existente, siguiendo el **principio abierto/cerrado (OCP)**.
4. **Fomentan buenas prácticas SOLID**  
   Aplican principios como **responsabilidad única (SRP)**, **inversión de dependencias (DIP)** y **abierto/cerrado (OCP)**.

# Patrónes

Strategy

Observer

Command

Chain of Responsibility

State

Mediator

Memento

Template Method

Visitor

## PATRÓN 1: Strategy

El patrón **Strategy** permite definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables. El algoritmo puede variar independientemente del cliente que lo utiliza.

Este patrón sigue el principio de **abierto/cerrado**: puedes agregar nuevos comportamientos sin modificar el código existente. Es muy útil cuando hay múltiples formas de hacer algo y el algoritmo concreto puede cambiar en tiempo de ejecución.

**🛠 ¿Cuándo usarlo?**

* Cuando tienes múltiples variantes de un algoritmo (ordenamiento, cálculo, validación, etc.)
* Cuando quieres evitar múltiples condicionales if-else o switch-case.
* Cuando quieras permitir la selección dinámica del comportamiento en tiempo de ejecución.

**⚠️ Antipatrones**

* Incluir estrategias que son demasiado simples y no justifican la sobrecarga del patrón.
* Definir tantas estrategias que terminan generando complejidad innecesaria (sobreingeniería).
* No separar adecuadamente la estrategia de su contexto.

**🧑‍💼 Historia de Usuario**

**Contexto: Aplicación de cálculo de impuestos en un ERP de una empresa logística.**

Camila es una ingeniera de software encargada del módulo de facturación. Su sistema debe calcular el impuesto según el país del cliente: si el país es **Colombia**, se aplica el 19% de IVA; si es **Argentina**, se calcula el 21% de IVA; en **Ecuador**, no se aplica impuesto.

Inicialmente, Camila implementó una serie de condicionales que seleccionaban el algoritmo de cálculo dependiendo del país. Sin embargo, con cada cambio de política fiscal, debía modificar esta clase, violando el principio abierto/cerrado.

Decidió aplicar el patrón **Strategy**: creó una interfaz EstrategiaImpuesto con un método calcular(double monto) y definió implementaciones específicas para cada país.

Gracias a esto, pudo registrar estrategias nuevas en tiempo de ejecución y cumplir con los requerimientos del equipo de compliance fiscal sin modificar el código base.

**💻 Código Java**

// Estrategia

interface EstrategiaImpuesto {

double calcular(double monto);

}

// Implementación para Colombia

class ImpuestoColombia implements EstrategiaImpuesto {

public double calcular(double monto) {

return monto \* 0.19;

}

}

// Implementación para Argentina

class ImpuestoArgentina implements EstrategiaImpuesto {

public double calcular(double monto) {

return monto \* 0.21;

}

}

// Implementación para Ecuador

class ImpuestoEcuador implements EstrategiaImpuesto {

public double calcular(double monto) {

return 0.0;

}

}

// Contexto

class CalculadoraImpuesto {

private EstrategiaImpuesto estrategia;

public void setEstrategia(EstrategiaImpuesto estrategia) {

this.estrategia = estrategia;

}

public double calcularImpuesto(double monto) {

return estrategia.calcular(monto);

}

}

// Cliente

public class Main {

public static void main(String[] args) {

CalculadoraImpuesto calculadora = new CalculadoraImpuesto();

calculadora.setEstrategia(new ImpuestoColombia());

System.out.println("Colombia: " + calculadora.calcularImpuesto(100000));

calculadora.setEstrategia(new ImpuestoArgentina());

System.out.println("Argentina: " + calculadora.calcularImpuesto(100000));

calculadora.setEstrategia(new ImpuestoEcuador());

System.out.println("Ecuador: " + calculadora.calcularImpuesto(100000));

}

}

## PATRÓN 2: Observer

El patrón **Observer** define una dependencia uno-a-muchos entre objetos de manera que cuando uno cambia de estado, todos sus **observadores** son notificados automáticamente. Es ideal para implementar sistemas de **eventos** o **suscripción**.

Este patrón promueve un **bajo acoplamiento** entre el sujeto y sus observadores. El sujeto no necesita saber cuántos observadores existen ni cómo reaccionan.

**🛠 ¿Cuándo usarlo?**

* Cuando múltiples objetos necesitan reaccionar a cambios en otro objeto sin acoplarse directamente.
* Cuando implementas un sistema de **notificaciones**, **eventos**, o **logs**.
* Cuando quieres permitir extensiones dinámicas (plugins, hooks, listeners).

**⚠️ Antipatrones**

* Notificar observadores innecesariamente incluso si no hay cambios relevantes.
* No implementar mecanismos de desuscripción, generando fugas de memoria.
* Hacer que el observador dependa de estados concretos del sujeto, rompiendo la abstracción.

**🧑‍💼 Historia de Usuario**

**Contexto: Sistema de Monitoreo de Sensores en una Planta Industrial**

En una planta de producción de alimentos, Julia es responsable de monitorear sensores de temperatura ubicados en diferentes zonas de almacenamiento. Cada vez que un sensor supera los 8 °C, deben activarse múltiples acciones:

* Notificar al gerente por correo electrónico.
* Registrar el evento en el sistema de alertas.
* Activar una orden de verificación técnica.

Antes, todo esto estaba embebido en una sola clase de Sensor. El código era difícil de mantener y no podía ampliarse fácilmente.

Julia decidió aplicar el patrón **Observer**: el **Sensor** se convierte en el "sujeto observable", y cada uno de los módulos anteriores se convierte en un "observador" que se suscribe al sensor.

Con esta solución, ahora puede agregar o quitar observadores sin tocar el código del sensor. Incluso el equipo de TI implementó nuevos observadores para reportes en Power BI y simuladores.

**💻 Código Java**

// Interfaz del observador

interface Observador {

void actualizar(double temperatura);

}

// Sujeto (Sensor)

class Sensor {

private List<Observador> observadores = new ArrayList<>();

private double temperatura;

public void agregarObservador(Observador o) {

observadores.add(o);

}

public void removerObservador(Observador o) {

observadores.remove(o);

}

public void setTemperatura(double temperatura) {

this.temperatura = temperatura;

notificarObservadores();

}

private void notificarObservadores() {

for (Observador o : observadores) {

o.actualizar(temperatura);

}

}

}

// Observador 1: Notificación por correo

class NotificadorEmail implements Observador {

public void actualizar(double temperatura) {

if (temperatura > 8.0) {

System.out.println("Email enviado: temperatura alta -> " + temperatura + "°C");

}

}

}

// Observador 2: Registro en sistema

class RegistroEvento implements Observador {

public void actualizar(double temperatura) {

if (temperatura > 8.0) {

System.out.println("Evento registrado en base de datos -> " + temperatura + "°C");

}

}

}

// Observador 3: Orden de verificación

class OrdenTecnica implements Observador {

public void actualizar(double temperatura) {

if (temperatura > 8.0) {

System.out.println("Orden de revisión técnica generada.");

}

}

}

// Cliente

public class Main {

public static void main(String[] args) {

Sensor sensor = new Sensor();

sensor.agregarObservador(new NotificadorEmail());

sensor.agregarObservador(new RegistroEvento());

sensor.agregarObservador(new OrdenTecnica());

sensor.setTemperatura(6.5); // no se notifica

sensor.setTemperatura(8.5); // activa todos los observadores

}

}

## PATRÓN 3: Command

El patrón **Command** encapsula una solicitud como un objeto, permitiendo parametrizar clientes con diferentes solicitudes, colas de solicitudes y soporte para operaciones reversibles (undo/redo). Se utiliza cuando se desea desacoplar el emisor de una acción del objeto que la ejecuta.

Este patrón es ideal para aplicaciones que requieren historial de acciones, botones configurables o ejecución de tareas encoladas.

**🛠 ¿Cuándo usarlo?**

* Cuando necesitas **desacoplar** el objeto que envía una orden del que la ejecuta.
* Cuando quieres soportar operaciones **undo/redo**.
* Para **encolar o registrar comandos**, como en tareas programadas, acciones de usuario o editores.

**⚠️ Antipatrones**

* Crear demasiadas clases de comandos triviales puede generar sobreingeniería.
* No proveer una estructura común para ejecutar, deshacer o rehacer operaciones.
* Acoplar demasiado al invocador con comandos específicos.

**🧑‍💼 Historia de Usuario**

**Contexto: Panel de control en una aplicación de automatización de oficinas**

Santiago es desarrollador en una startup de domótica que permite controlar luces, cortinas y aire acondicionado desde una app móvil. El cliente desea que desde un panel central se puedan **configurar botones** que envíen órdenes a distintos dispositivos.

Antes, Santiago implementó lógica en botones como if (tipo.equals("LUZ")) { encenderLuz(); }. Este diseño no era extensible. Cada vez que se agregaba un nuevo dispositivo, debía modificar código existente, violando el principio de **cerrado para modificación**.

Aplicó el patrón **Command**: creó una interfaz Comando con el método ejecutar(). Cada acción (encender luz, subir cortina, activar aire) se encapsuló en su propia clase de comando.

Los botones del panel ahora reciben comandos configurables en tiempo de ejecución. Esto permitió agregar nuevas funciones, crear colas de tareas y hasta implementar funciones de deshacer.

**💻 Código Java**

// Interfaz comando

interface Comando {

void ejecutar();

}

// Receptor 1: Luz

class Luz {

public void encender() {

System.out.println("Luz encendida.");

}

}

// Receptor 2: Cortina

class Cortina {

public void subir() {

System.out.println("Cortina subida.");

}

}

// Comando concreto: Encender Luz

class ComandoEncenderLuz implements Comando {

private Luz luz;

public ComandoEncenderLuz(Luz luz) {

this.luz = luz;

}

public void ejecutar() {

luz.encender();

}

}

// Comando concreto: Subir Cortina

class ComandoSubirCortina implements Comando {

private Cortina cortina;

public ComandoSubirCortina(Cortina cortina) {

this.cortina = cortina;

}

public void ejecutar() {

cortina.subir();

}

}

// Invocador: Botón configurable

class Boton {

private Comando comando;

public Boton(Comando comando) {

this.comando = comando;

}

public void presionar() {

comando.ejecutar();

}

}

// Cliente

public class Main {

public static void main(String[] args) {

Luz luz = new Luz();

Cortina cortina = new Cortina();

Comando cmdLuz = new ComandoEncenderLuz(luz);

Comando cmdCortina = new ComandoSubirCortina(cortina);

Boton boton1 = new Boton(cmdLuz);

Boton boton2 = new Boton(cmdCortina);

boton1.presionar(); // Luz encendida

boton2.presionar(); // Cortina subida

}

}

## PATRÓN 4: Chain of Responsibility (Cadena de Responsabilidad)

El patrón **Chain of Responsibility** permite pasar una solicitud a lo largo de una **cadena de manejadores** hasta que uno de ellos la procese. Cada manejador decide si procesa la solicitud o la pasa al siguiente.

Esto **desacopla el emisor de la solicitud de su receptor**, y permite construir sistemas flexibles y escalables donde múltiples componentes pueden procesar peticiones condicionalmente.

**🛠 ¿Cuándo usarlo?**

* Cuando varios objetos pueden manejar una solicitud, pero no sabes cuál exactamente.
* Cuando deseas evitar acoplamiento fuerte entre emisor y receptor.
* Cuando quieres construir flujos dinámicos de procesamiento (middleware, validadores, aprobaciones).

**⚠️ Antipatrones**

* Cadena demasiado larga puede afectar el rendimiento.
* No implementar un manejador final que maneje el caso por defecto puede provocar errores silenciosos.
* No documentar el orden o lógica de encadenamiento puede dificultar el mantenimiento.

**🧑‍💼 Historia de Usuario**

**Contexto: Sistema de aprobación de solicitudes en una empresa multinacional**

María es analista de TI en una empresa multinacional que implementa un nuevo sistema de aprobación de compras. Cada solicitud de compra debe ser evaluada por distintos roles según su monto:

* Menor a $500: supervisor directo.
* Entre $500 y $5000: jefe de departamento.
* Mayor a $5000: dirección financiera.

Inicialmente, el flujo estaba codificado como una gran estructura if-else, lo que hacía difícil su modificación. María decide aplicar el patrón **Chain of Responsibility**.

Define una interfaz Aprobador con un método setSiguiente() y procesarSolicitud(). Cada clase representa un rol y maneja solo su caso.

Ahora, la cadena se configura dinámicamente, y nuevos niveles de aprobación pueden agregarse fácilmente sin modificar la lógica principal. Incluso se pueden registrar logs de cada paso en la cadena para auditoría.

**💻 Código Java**

// Interfaz manejador

abstract class Aprobador {

protected Aprobador siguiente;

public void setSiguiente(Aprobador siguiente) {

this.siguiente = siguiente;

}

public abstract void procesarSolicitud(double monto);

}

// Supervisor

class Supervisor extends Aprobador {

public void procesarSolicitud(double monto) {

if (monto <= 500) {

System.out.println("Aprobado por Supervisor.");

} else if (siguiente != null) {

siguiente.procesarSolicitud(monto);

}

}

}

// Jefe de Departamento

class JefeDepartamento extends Aprobador {

public void procesarSolicitud(double monto) {

if (monto <= 5000) {

System.out.println("Aprobado por Jefe de Departamento.");

} else if (siguiente != null) {

siguiente.procesarSolicitud(monto);

}

}

}

// Director Financiero

class DirectorFinanciero extends Aprobador {

public void procesarSolicitud(double monto) {

if (monto > 5000) {

System.out.println("Aprobado por Director Financiero.");

} else if (siguiente != null) {

siguiente.procesarSolicitud(monto);

}

}

}

// Cliente

public class Main {

public static void main(String[] args) {

Aprobador supervisor = new Supervisor();

Aprobador jefe = new JefeDepartamento();

Aprobador director = new DirectorFinanciero();

supervisor.setSiguiente(jefe);

jefe.setSiguiente(director);

supervisor.procesarSolicitud(300); // Supervisor

supervisor.procesarSolicitud(3000); // Jefe de Departamento

supervisor.procesarSolicitud(15000); // Director Financiero

}

}

## PATRÓN 5: State (Estado)

El patrón **State** permite a un objeto cambiar su comportamiento cuando cambia su estado interno, como si el objeto **cambiara de clase en tiempo de ejecución**. Se utiliza para evitar grandes estructuras if-else o switch basadas en estados.

Este patrón encapsula los distintos estados y comportamientos en clases separadas, facilitando la extensión y mantenimiento del código.

**🛠 ¿Cuándo usarlo?**

* Cuando un objeto cambia de comportamiento dependiendo de su estado.
* Cuando hay muchos condicionales if/else para manejar estados internos.
* Cuando necesitas que el objeto maneje transiciones entre estados sin exponer su lógica.

**⚠️ Antipatrones**

* Usar el patrón si los estados son triviales o no afectan el comportamiento.
* Colocar la lógica de transición fuera del contexto, violando el principio de encapsulamiento.
* No implementar interfaz común para los estados, lo que rompe la consistencia del contexto.

**🧑‍💼 Historia de Usuario**

**Contexto: Cajero automático con múltiples estados de operación**

Andrés trabaja en el equipo de desarrollo de una fintech que quiere simular un **cajero automático** en software. Este cajero debe tener comportamientos distintos dependiendo del estado actual:

* Cuando **sin tarjeta**, debe pedir que se inserte una.
* Cuando **con tarjeta**, debe solicitar PIN.
* Cuando **autenticado**, debe permitir operaciones.

La implementación inicial se basaba en muchos condicionales, lo que dificultaba las pruebas unitarias y el mantenimiento. Andrés decide usar el patrón **State**.

Define una interfaz EstadoCajero y crea clases como SinTarjeta, ConTarjeta y Autenticado, cada una encapsulando la lógica específica.

El objeto Cajero mantiene una referencia al estado actual y delega en él el comportamiento. El sistema se volvió más legible, cada estado es fácilmente extensible, y ahora se pueden agregar nuevos estados (por ejemplo, Bloqueado) sin alterar el código existente.

**💻 Código Java**

// Interfaz de Estado

interface EstadoCajero {

void insertarTarjeta();

void ingresarPin(int pin);

void retirarDinero(double monto);

}

// Estado: sin tarjeta

class SinTarjeta implements EstadoCajero {

private Cajero cajero;

public SinTarjeta(Cajero cajero) {

this.cajero = cajero;

}

public void insertarTarjeta() {

System.out.println("Tarjeta insertada.");

cajero.setEstado(new ConTarjeta(cajero));

}

public void ingresarPin(int pin) {

System.out.println("Primero inserte una tarjeta.");

}

public void retirarDinero(double monto) {

System.out.println("Primero inserte una tarjeta.");

}

}

// Estado: con tarjeta (esperando PIN)

class ConTarjeta implements EstadoCajero {

private Cajero cajero;

public ConTarjeta(Cajero cajero) {

this.cajero = cajero;

}

public void insertarTarjeta() {

System.out.println("Ya hay una tarjeta insertada.");

}

public void ingresarPin(int pin) {

if (pin == 1234) {

System.out.println("PIN correcto.");

cajero.setEstado(new Autenticado(cajero));

} else {

System.out.println("PIN incorrecto.");

}

}

public void retirarDinero(double monto) {

System.out.println("Debe ingresar su PIN primero.");

}

}

// Estado: autenticado

class Autenticado implements EstadoCajero {

private Cajero cajero;

public Autenticado(Cajero cajero) {

this.cajero = cajero;

}

public void insertarTarjeta() {

System.out.println("Ya autenticado. No necesita reiniciar.");

}

public void ingresarPin(int pin) {

System.out.println("Ya autenticado.");

}

public void retirarDinero(double monto) {

System.out.println("Retirando $" + monto);

cajero.setEstado(new SinTarjeta(cajero));

}

}

// Contexto: Cajero

class Cajero {

private EstadoCajero estado;

public Cajero() {

this.estado = new SinTarjeta(this);

}

public void setEstado(EstadoCajero estado) {

this.estado = estado;

}

public void insertarTarjeta() {

estado.insertarTarjeta();

}

public void ingresarPin(int pin) {

estado.ingresarPin(pin);

}

public void retirarDinero(double monto) {

estado.retirarDinero(monto);

}

}

// Cliente

public class Main {

public static void main(String[] args) {

Cajero cajero = new Cajero();

cajero.insertarTarjeta();

cajero.ingresarPin(1111);

cajero.ingresarPin(1234);

cajero.retirarDinero(200);

}

}

## PATRÓN 6: Mediator (Mediador)

El patrón **Mediator** define un objeto que encapsula cómo interactúan un conjunto de objetos. Este patrón promueve un **acoplamiento débil** al evitar que los objetos se referencien directamente entre sí, facilitando la escalabilidad y el mantenimiento.

El mediador **centraliza la comunicación** y coordinación entre componentes, haciendo que estos solo se comuniquen con él.

**🛠 ¿Cuándo usarlo?**

* Cuando tienes múltiples objetos interdependientes que interactúan entre sí de forma compleja.
* Cuando deseas simplificar la comunicación entre objetos.
* En sistemas GUI (interfaces gráficas) con múltiples componentes que interactúan.

**⚠️ Antipatrones**

* Convertir al Mediador en un "Dios del sistema", asumiendo demasiadas responsabilidades.
* Usarlo cuando hay poca interacción entre componentes (introducir complejidad innecesaria).
* No definir una interfaz clara para el mediador, acoplando componentes directamente a una implementación.

**🧑‍💼 Historia de Usuario**

**Contexto: Sistema de mensajería entre componentes en una torre de control de vuelos**

Clara es ingeniera de software en un sistema de gestión aérea. Está diseñando un módulo donde **aviones, controladores de vuelo y personal de pista** deben intercambiar mensajes para coordinar aterrizajes.

Inicialmente, cada clase conocía las otras y se comunicaba directamente, creando una red compleja y propensa a errores. Cualquier cambio en una clase obligaba a modificar muchas otras.

Clara decide implementar el patrón **Mediator**. Define una interfaz TorreControl, que centraliza toda la coordinación. Los objetos Avion, Controlador y Pista solo se comunican a través del mediador.

Esto permite que nuevos componentes puedan integrarse (como un sistema automático de alertas meteorológicas) sin alterar las clases existentes. Además, el código es más fácil de probar y mantener.

**💻 Código Java**

// Interfaz del Mediador

interface TorreControl {

void enviarMensaje(String mensaje, Participante emisor);

void registrarParticipante(Participante p);

}

// Participantes abstractos

abstract class Participante {

protected TorreControl torre;

protected String nombre;

public Participante(String nombre, TorreControl torre) {

this.nombre = nombre;

this.torre = torre;

torre.registrarParticipante(this);

}

public abstract void recibir(String mensaje);

public void enviar(String mensaje) {

torre.enviarMensaje(mensaje, this);

}

public String getNombre() {

return nombre;

}

}

// Participante concreto: Avión

class Avion extends Participante {

public Avion(String nombre, TorreControl torre) {

super(nombre, torre);

}

public void recibir(String mensaje) {

System.out.println("Avión " + nombre + " recibe: " + mensaje);

}

}

// Participante concreto: Controlador

class Controlador extends Participante {

public Controlador(String nombre, TorreControl torre) {

super(nombre, torre);

}

public void recibir(String mensaje) {

System.out.println("Controlador " + nombre + " recibe: " + mensaje);

}

}

// Participante concreto: Pista

class Pista extends Participante {

public Pista(String nombre, TorreControl torre) {

super(nombre, torre);

}

public void recibir(String mensaje) {

System.out.println("Pista " + nombre + " recibe: " + mensaje);

}

}

// Mediador concreto

class TorreDeControl implements TorreControl {

private List<Participante> participantes = new ArrayList<>();

public void registrarParticipante(Participante p) {

participantes.add(p);

}

public void enviarMensaje(String mensaje, Participante emisor) {

for (Participante p : participantes) {

if (p != emisor) {

p.recibir(mensaje + " (de " + emisor.getNombre() + ")");

}

}

}

}

// Cliente

public class Main {

public static void main(String[] args) {

TorreControl torre = new TorreDeControl();

Participante avion1 = new Avion("LATAM 123", torre);

Participante controlador = new Controlador("ATC Norte", torre);

Participante pista = new Pista("Pista A", torre);

avion1.enviar("Solicito autorización para aterrizar");

controlador.enviar("Autorizado a aterrizar");

}

}

## PATRÓN 7: Memento (Recuerdo)

El patrón **Memento** permite capturar y externalizar el estado interno de un objeto sin violar el encapsulamiento, para que pueda ser restaurado más tarde. Es ideal para implementar funcionalidades de **deshacer (undo)** y **rehacer (redo)**.

El patrón se compone de tres partes:

* **Originador**: el objeto cuyo estado queremos guardar.
* **Memento**: el objeto que representa el estado guardado.
* **Caretaker**: el encargado de almacenar/restaurar los mementos.

**🛠 ¿Cuándo usarlo?**

* Cuando necesitas permitir a los usuarios **revertir acciones**.
* Cuando el objeto tiene un estado complejo que cambia frecuentemente.
* Cuando deseas mantener el encapsulamiento pero ofrecer respaldo/restauración de estado.

**⚠️ Antipatrones**

* Guardar demasiado estado innecesariamente en el memento (afecta el rendimiento).
* Permitir que el caretaker modifique el memento (rompe el encapsulamiento).
* No limitar la cantidad de estados almacenados (puede consumir demasiada memoria).

**🧑‍💼 Historia de Usuario**

**Contexto: Editor de texto con funcionalidad de deshacer**

Daniela trabaja como desarrolladora frontend en una app de edición de notas en línea. Los usuarios pueden escribir texto, y la app guarda automáticamente cada versión del texto cada cierto tiempo o antes de una operación destructiva.

En un inicio, Daniela intentó guardar copias completas del objeto en listas manualmente, pero esto era propenso a errores y difícil de mantener. Para resolverlo de forma limpia, aplicó el patrón **Memento**.

Creó una clase EditorTexto que actúa como el **Originador**, una clase Memento que guarda el estado del texto, y una clase Historial como **Caretaker**, que maneja una pila de estados.

Ahora, el editor puede hacer undo() para regresar al último estado sin violar la encapsulación. La solución es limpia, testable, y fácil de extender para soporte de redo().

**💻 Código Java**

// Originador

class EditorTexto {

private String contenido = "";

public void escribir(String texto) {

contenido += texto;

}

public String getContenido() {

return contenido;

}

public Memento guardar() {

return new Memento(contenido);

}

public void restaurar(Memento memento) {

this.contenido = memento.getEstado();

}

// Memento interno

public static class Memento {

private final String estado;

private Memento(String estado) {

this.estado = estado;

}

private String getEstado() {

return estado;

}

}

}

// Caretaker

class Historial {

private Stack<EditorTexto.Memento> pila = new Stack<>();

public void guardar(EditorTexto.Memento m) {

pila.push(m);

}

public EditorTexto.Memento deshacer() {

if (!pila.isEmpty()) {

return pila.pop();

}

return null;

}

}

// Cliente

public class Main {

public static void main(String[] args) {

EditorTexto editor = new EditorTexto();

Historial historial = new Historial();

editor.escribir("Hola");

historial.guardar(editor.guardar());

editor.escribir(" mundo");

historial.guardar(editor.guardar());

editor.escribir(" cruel");

System.out.println("Actual: " + editor.getContenido()); // Hola mundo cruel

editor.restaurar(historial.deshacer());

System.out.println("Undo 1: " + editor.getContenido()); // Hola mundo

editor.restaurar(historial.deshacer());

System.out.println("Undo 2: " + editor.getContenido()); // Hola

}

}

## PATRÓN 8: Template Method

El patrón **Template Method** define el **esqueleto de un algoritmo** en una operación, dejando algunos pasos a las subclases. Permite redefinir ciertos pasos del algoritmo sin cambiar su estructura general.

Este patrón promueve la reutilización de código base y favorece el principio de **inversión de control**, permitiendo que las subclases decidan el comportamiento específico en algunos pasos del proceso.

**🛠 ¿Cuándo usarlo?**

* Cuando tienes algoritmos similares con pasos que varían ligeramente.
* Cuando deseas evitar duplicación de lógica y mantener una estructura fija.
* Cuando deseas controlar la secuencia de pasos en un proceso.

**⚠️ Antipatrones**

* Dejar demasiados métodos abstractos, rompiendo la cohesión del algoritmo.
* Romper el principio de responsabilidad única si el método plantilla hace demasiado.
* Usar Template Method sin necesidad real, introduciendo rigidez.

**🧑‍💼 Historia de Usuario**

**Contexto: Generación de reportes PDF y Excel en una plataforma contable**

Alejandro trabaja en el backend de una plataforma contable SaaS que genera reportes para los usuarios. Los reportes pueden exportarse en **PDF** o **Excel**, y aunque comparten varios pasos (recolección de datos, encabezado, pie de página), difieren en la forma en que presentan los datos y el formato.

Inicialmente, se implementaron dos clases separadas que compartían mucha lógica duplicada. Alejandro decidió usar el patrón **Template Method**.

Creó una clase abstracta GeneradorReporte con el método generar() que incluye pasos comunes como obtenerDatos(), generarEncabezado(), generarContenido() y generarPie(). Los métodos generarContenido() y formatear() se dejaron como abstractos.

Las clases ReportePDF y ReporteExcel extienden GeneradorReporte e implementan las diferencias específicas. Ahora, agregar un nuevo formato de exportación solo requiere implementar dos métodos, manteniendo la estructura central intacta.

**💻 Código Java**

// Clase abstracta con el método plantilla

abstract class GeneradorReporte {

public final void generar() {

obtenerDatos();

generarEncabezado();

generarContenido(); // método abstracto

generarPie();

formatear();

}

protected void obtenerDatos() {

System.out.println("Obteniendo datos de la base...");

}

protected void generarEncabezado() {

System.out.println("Generando encabezado del reporte.");

}

protected void generarPie() {

System.out.println("Agregando pie de página.");

}

protected abstract void generarContenido();

protected abstract void formatear();

}

// Implementación para PDF

class ReportePDF extends GeneradorReporte {

protected void generarContenido() {

System.out.println("Escribiendo contenido en PDF.");

}

protected void formatear() {

System.out.println("Formateando como documento PDF.");

}

}

// Implementación para Excel

class ReporteExcel extends GeneradorReporte {

protected void generarContenido() {

System.out.println("Escribiendo contenido en celdas Excel.");

}

protected void formatear() {

System.out.println("Formateando como hoja Excel.");

}

}

// Cliente

public class Main {

public static void main(String[] args) {

GeneradorReporte pdf = new ReportePDF();

GeneradorReporte excel = new ReporteExcel();

System.out.println("==== Generando PDF ====");

pdf.generar();

System.out.println("\n==== Generando Excel ====");

excel.generar();

}

}

## PATRÓN 9: Visitor (Visitante)

El patrón **Visitor** permite definir una operación a realizar sobre una estructura de objetos sin cambiar las clases sobre las que opera. Permite **agregar nuevas funcionalidades** a objetos sin modificar su código, respetando el principio de **abierto/cerrado**.

Se utiliza cuando se necesita realizar múltiples operaciones sobre un conjunto de objetos heterogéneos.

**🛠 ¿Cuándo usarlo?**

* Cuando tienes una estructura de objetos sobre la que se deben aplicar múltiples operaciones sin modificar sus clases.
* Cuando deseas **desacoplar** operaciones de los objetos que las reciben.
* Cuando necesitas agregar comportamientos nuevos frecuentemente.

**⚠️ Antipatrones**

* Romper el encapsulamiento accediendo a detalles internos innecesarios.
* Aplicarlo cuando las operaciones cambian más frecuentemente que la estructura de objetos (mejor usar Strategy).
* Crear jerarquías muy complejas de visitantes si hay demasiados tipos de elementos.

**🧑‍💼 Historia de Usuario**

**Contexto: Sistema de contabilidad que genera diferentes reportes sobre activos**

Francisco trabaja en un sistema contable que maneja diferentes tipos de activos: **Inmuebles**, **Vehículos**, **Equipos**. El sistema debe generar reportes fiscales y financieros sobre estos activos.

Cada tipo de activo tiene reglas distintas para el cálculo de depreciación, valor residual, e impacto contable. Además, los reportes pueden cambiar con la normativa legal.

En lugar de modificar las clases Activo, Inmueble, Vehículo, cada vez que se agrega una operación nueva, Francisco implementa el patrón **Visitor**. Define una interfaz ActivoVisitor con métodos visitar(Inmueble), visitar(Vehiculo), etc.

Las operaciones como ReporteFiscalVisitor, ReporteFinancieroVisitor implementan esa interfaz y aplican lógica según el tipo de activo. Esto permitió agregar nuevas operaciones sin modificar las clases base, y separar completamente la lógica contable de la lógica de presentación.

**💻 Código Java**

// Interfaz del Visitor

interface ActivoVisitor {

void visitar(Inmueble i);

void visitar(Vehiculo v);

}

// Interfaz del elemento visitable

interface Activo {

void aceptar(ActivoVisitor visitor);

}

// Elemento 1

class Inmueble implements Activo {

public void aceptar(ActivoVisitor visitor) {

visitor.visitar(this);

}

public double calcularValorFiscal() {

return 100000;

}

}

// Elemento 2

class Vehiculo implements Activo {

public void aceptar(ActivoVisitor visitor) {

visitor.visitar(this);

}

public double calcularValorFiscal() {

return 35000;

}

}

// Visitor concreto: Reporte Fiscal

class ReporteFiscalVisitor implements ActivoVisitor {

public void visitar(Inmueble i) {

System.out.println("Inmueble - Valor fiscal: $" + i.calcularValorFiscal());

}

public void visitar(Vehiculo v) {

System.out.println("Vehículo - Valor fiscal: $" + v.calcularValorFiscal());

}

}

// Cliente

public class Main {

public static void main(String[] args) {

List<Activo> activos = new ArrayList<>();

activos.add(new Inmueble());

activos.add(new Vehiculo());

ActivoVisitor reporte = new ReporteFiscalVisitor();

for (Activo a : activos) {

a.aceptar(reporte);

}

}

}

## ¿Cuándo utilizarlos?

|  |  |  |
| --- | --- | --- |
| Situación | Ejemplo | Patrón recomendado |
| Necesitas seleccionar un algoritmo en tiempo de ejecución | Validación fiscal, cálculo de precios | Strategy |
| Quieres responder a cambios de estado o eventos | Paneles, sensores, notificaciones | Observer |
| Necesitas ejecutar, deshacer o guardar comandos | Acciones de usuario, controles remotos | Command, Memento |
| Manejas una cadena de validaciones o filtros | Procesos de aprobación, validaciones | Chain of Responsibility |
| Deseas encapsular estados internos que afectan comportamiento | Flujo de pedidos, ciclos de vida | State |
| Tienes múltiples objetos que interactúan entre sí | Componentes de UI, módulos acoplados | Mediator |
| Requieres aplicar operaciones a muchos objetos distintos | Analítica, reporting, recorridos | Visitor |
| Deseas estandarizar pasos de un proceso dejando algunos abiertos | Generación de reportes, procesos | Template Method |

## Aspectos clave para su uso efectivo

1. **Identificar el comportamiento que varía**
   * Los patrones de comportamiento son útiles cuando el **cómo se hace algo que cambia con frecuencia**.
2. **Separar el "qué se hace" del "quién lo hace"**
   * Ejemplo: Command separa el emisor de la acción del ejecutor real.
3. **Evitar condicionales repetitivos**
   * Si usas mucho if/else, puedes necesitar Strategy, State o Chain of Responsibility.
4. **Considerar la trazabilidad y reversibilidad**
   * Si necesitas deshacer acciones (undo), los patrones Memento y Command son clave.
5. **No aplicar por aplicar**
   * El uso innecesario de patrones puede **complicar sistemas simples**. Evalúa si el beneficio supera la complejidad agregada.

## Conclusión

Los patrones de comportamiento son una poderosa herramienta para **modelar la lógica dinámica y colaborativa** de sistemas complejos.  
Aplicados correctamente, permiten construir **sistemas flexibles, robustos, modulares y fáciles de mantener**, adaptables a entornos en constante cambio.

Más que soluciones rígidas, los patrones de comportamiento son **formas probadas de pensar** que ayudan a estructurar interacciones, tomar decisiones y evolucionar software de manera disciplinada.