PATRONES CREACIONALES

Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y reutilización de código existente.

FACTORY METHOD

Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán. Es útil para que el código que crea objetos esté desacoplado de las clases concretas que se instancian.

Características principales del patrón:

- Abstracción de la creación de objetos: Delegamos la creación del objeto a una subclase o implementación específica.
- Polimorfismo: La clase base define un método abstracto para crear objetos, y las subclases concretas proporcionan la implementación.
- Extensibilidad: Agregar nuevas clases de productos no requiere cambiar el código existente, solo hay que implementar nuevas fábricas.

```
// Producto: Interfaz común para todos los vehículos
interface Vehiculo {
    void conducir();
}

// Productos concretos: Auto y Moto
class Auto implements Vehiculo {
    @Override
    public void conducir() {
        System.out.println("Conduciendo un auto");
    }
}

class Moto implements Vehiculo {
    @Override
    public void conducir() {
        System.out.println("Conduciendo una moto");
    }
}

// Creador: Define el método abstracto para crear vehículos
abstract class FabricaVehiculo {
    abstract Vehiculo crearVehiculo();
}

// Fábricas concretas: Implementan el método de creación
class FabricaAuto extends FabricaVehiculo {
    @Override
    Vehiculo crearVehiculo() {
```

```
        return new Auto();
    }
}

class FabricaMoto extends FabricaVehiculo {
    @Override
    Vehiculo crearVehiculo() {
        return new Moto();
    }
}

// Cliente: Usa la fábrica sin conocer los detalles del producto concreto
public class Main {
    public static void main(String[] args) {
        FabricaVehiculo fabrica = new FabricaAuto();
        Vehiculo vehiculo = fabrica.crearVehiculo();
        vehiculo.conducir(); // Salida: Conduciendo un auto

        fabrica = new FabricaMoto();
        vehiculo = fabrica.crearVehiculo();
        vehiculo.conducir(); // Salida: Conduciendo una moto
    }
}
```

El creador basicamente lo único que tiene es un conjunto de **factory methods (metodos creadores)** y cada método creador es implementado de manera concreta por una subclase.

Cuándo usarlo:

- Cuando necesitas que la clase que usa un objeto no dependa de cómo se crea.
- Cuando tienes varias implementaciones o derivaciones posibles de un producto.

ABSTRACT FACTORY

Permite producir familias de objetos relacionados o dependientes sin especificar sus clases concretas. Mientras que el **Factory Method** se enfoca en la creación de un único tipo de producto, **Abstract Factory** trata con múltiples familias de productos relacionados.

```
// Producto 1: Botón
interface Boton {
    void render();
}

class BotonWindows implements Boton {
    @Override
    public void render() {
        System.out.println("Renderizando botón estilo Windows");
```

```
    }
}

class BotonMac implements Boton {
    @Override
    public void render() {
        System.out.println("Renderizando botón estilo Mac");
    }
}

// Producto 2: Ventana
interface Ventana {
    void mostrar();
}

class VentanaWindows implements Ventana {
    @Override
    public void mostrar() {
        System.out.println("Mostrando ventana estilo Windows");
    }
}

class VentanaMac implements Ventana {
    @Override
    public void mostrar() {
        System.out.println("Mostrando ventana estilo Mac");
    }
}

// Abstract Factory: Define la familia de productos
interface GUIFactory {
    Boton crearBoton();
    Ventana crearVentana();
}

// Fábricas concretas para cada sistema operativo
class WindowsFactory implements GUIFactory {
    @Override
    public Boton crearBoton() {
        return new BotonWindows();
    }

    @Override
    public Ventana crearVentana() {
        return new VentanaWindows();
    }
```

```java
}

class MacFactory implements GUIFactory {
    @Override
    public Boton crearBoton() {
        return new BotonMac();
    }

    @Override
    public Ventana crearVentana() {
        return new VentanaMac();
    }
}

// Cliente: Usa la fábrica abstracta para crear productos
public class Main {
    public static void main(String[] args) {
        GUIFactory factory = new WindowsFactory(); // Cambia a MacFactory para Mac
        Boton boton = factory.crearBoton();
        Ventana ventana = factory.crearVentana();

        boton.render();   // Salida: Renderizando botón estilo Windows
        ventana.mostrar(); // Salida: Mostrando ventana estilo Windows
    }
}
```

Cuándo usar Abstract Factory

- Cuando tienes múltiples familias de objetos relacionados y necesitas asegurarte de que se usen juntos.
- Cuando necesitas garantizar que las implementaciones de productos sean consistentes en diferentes contextos (como estilos visuales).

Resumen de uso: Factory Method vs Abstract Factory
Factory Method: Útil para casos simples donde solo necesitas un tipo de producto.
Abstract Factory: Ideal para casos más complejos donde necesitas crear familias de objetos relacionados.

BUILDER PATTERN

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

El patrón Builder sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados constructores, para construir objetos complejos paso a paso. El patrón Builder no permite a otros objetos acceder al producto mientras se construye.

El patrón organiza la construcción de objetos en una serie de pasos (construirParedes, construirPuerta, etc.) y para crear un objeto, se ejecuta una serie de estos pasos en un objeto constructor. Lo importante es que no necesitas invocar todos los pasos. Puedes invocar sólo aquellos que sean necesarios para producir una configuración particular de un objeto.

Características principales del patrón Builder

- **Separación de construcción y representación**: El proceso de creación de un objeto está separado de su estructura final.
- **Construcción paso a paso:** Puedes construir un objeto al agregar partes gradualmente, y el orden de las partes es flexible.
- **Reutilización:** Permite reutilizar el mismo proceso de construcción para crear diferentes tipos de representaciones.

Ejemplo de codigo de Pizza

SINGLETON

Nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Características principales del patrón Singleton

- **Una sola instancia**: La clase asegura que se crea exactamente un objeto, y este es accesible en toda la aplicación.
- **Control de acceso global:** Ofrece un método o mecanismo estático para obtener esa instancia, en vez de tener que usar, por ejemplo, variables globales.

```
public class Singleton {
    // Instancia única, inicializada de manera estática
    private static Singleton instancia;

    // Constructor privado para evitar instanciación externa
    private Singleton() {
        System.out.println("Instancia Singleton creada.");
    }

    // Método público para obtener la instancia única
    public static Singleton getInstancia() {
        if (instancia == null) {
            instancia = new Singleton();
        }
        return instancia;
    }

    public void hacerAlgo() {
        System.out.println("Método de Singleton invocado.");
```

```
    }
}

// Cliente: Uso del Singleton
public class Main {
    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstancia();
        Singleton singleton2 = Singleton.getInstancia();

        singleton1.hacerAlgo(); // Salida: Método de Singleton invocado.

        // Comprobación de que ambas referencias apuntan a la misma instancia
        System.out.println(singleton1 == singleton2); // Salida: true
    }
}
```

Cuándo usar Singleton

- Cuando necesitas exactamente una instancia de una clase (por ejemplo, gestores de configuración o conexión a bases de datos).
- Cuando quieres controlar el acceso a un recurso compartido.

PATRONES ESTRUCTURALES

Los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras

ADAPTER

Permite la colaboración entre objetos con interfaces incompatibles.

El Adapter se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

Un adaptador envuelve uno de los objetos para esconder la complejidad de la conversión que tiene lugar tras bambalinas. El objeto envuelto ni siquiera es consciente de la existencia del adaptador. Por ejemplo, puedes envolver un objeto que opera con metros y kilómetros con un adaptador que convierte todos los datos al sistema anglosajón, es decir, pies y millas.

Los adaptadores no solo convierten datos a varios formatos, sino que también ayudan a objetos con distintas interfaces a colaborar. Funciona así:

- El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
- Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos

del adaptador.

- Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

Ejemplo: Imagina que tienes un dispositivo que necesita conectarse a un enchufe europeo, pero estás en un país donde los enchufes son estadounidenses. Usaremos un adaptador para resolver este problema.

```
// Interfaz esperada por el cliente
interface EnchufeEuropeo {
    void conectarEnEuropeo();
}

// Clase existente con una interfaz incompatible
class EnchufeAmericano {
    public void conectarEnAmericano() {
        System.out.println("Conectado a un enchufe americano.");
    }
}

// Adaptador: Traduce la interfaz de EnchufeAmericano a la de EnchufeEuropeo
class AdaptadorEnchufe implements EnchufeEuropeo {
    private EnchufeAmericano enchufeAmericano;

    // El adaptador envuelve al objeto existente
    public AdaptadorEnchufe(EnchufeAmericano enchufeAmericano) {
        this.enchufeAmericano = enchufeAmericano;
    }

    @Override
    public void conectarEnEuropeo() {
        // Traduce la llamada a la interfaz compatible
        enchufeAmericano.conectarEnAmericano();
    }
}

// Cliente: Usa la interfaz esperada
public class Main {
    public static void main(String[] args) {
        // Clase existente incompatible
        EnchufeAmericano enchufeAmericano = new EnchufeAmericano();

        // Adaptador para hacerla compatible con EnchufeEuropeo
        EnchufeEuropeo adaptador = new AdaptadorEnchufe(enchufeAmericano);

        // Uso del adaptador
```

```
        adaptador.conectarEnEuropeo(); // Salida: Conectado a un enchufe americano.
    }
}
```

Ventajas del Adapter

- Flexibilidad: Permite que clases existentes con interfaces incompatibles trabajen juntas.
- Reutilización de código: Evita modificar clases existentes, reduciendo riesgos.
- Desacoplamiento: Aísla el código cliente de las clases que tienen interfaces no compatibles.

Cuándo usar Adapter

- Cuando tienes una clase existente cuya interfaz no se ajusta a lo que necesitas.
- Cuando quieres integrar una librería de terceros con tu sistema sin modificar su código.
- Cuando necesitas un sistema que trabaje con múltiples formatos o protocolos diferentes.

DECORATOR

te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Cómo funciona:

- Encapsulación: Cada decorador añade una funcionalidad adicional antes o después de delegar la llamada al objeto base.
- Flexibilidad: Los decoradores pueden apilarse, permitiendo añadir múltiples comportamientos combinados.
- Interfaz uniforme: Tanto el objeto base como los decoradores implementan la misma interfaz, garantizando que el cliente no note diferencias entre ellos.

Ejemplo 1:

Problema: Una clase como Notificador debe soportar múltiples tipos de notificaciones (correo, SMS, Slack, etc.) y permitir combinarlas. Usar herencia para manejar todas las combinaciones posibles genera una explosión de subclases, dificultando el mantenimiento y la escalabilidad.

Solución: Usar composición en lugar de herencia. Cada tipo de notificación (correo, SMS, Slack) se implementa como un decorador que envuelve al objeto base (Notificador). Esto

permite combinar decoradores dinámicamente para crear configuraciones complejas, sin necesidad de crear múltiples subclases.

Ejemplo de notificaciones:

- Notificador envía correos electrónicos.
- Decoradores como NotificadorSMS o NotificadorSlack añaden soporte para SMS o mensajes por Slack.
- Al envolver el objeto base con decoradores, puedes enviar notificaciones simultáneamente por varios canales, como correo y SMS.

Ejemplo 2: Decorador de Ventana

```
// Componente base
interface Ventana {
    void dibujar(); // Método principal
}

// Clase concreta: Ventana básica
class VentanaBasica implements Ventana {
    @Override
    public void dibujar() {
        System.out.println("Dibujando una ventana básica");
    }
}

// Decorador abstracto
abstract class DecoradorVentana implements Ventana {
    protected Ventana ventanaDecorada;

    public DecoradorVentana(Ventana ventanaDecorada) {
        this.ventanaDecorada = ventanaDecorada;
    }

    @Override
    public void dibujar() {
        ventanaDecorada.dibujar();
    }
}

// Decorador concreto: Borde
class BordeDecorador extends DecoradorVentana {
    public BordeDecorador(Ventana ventanaDecorada) {
        super(ventanaDecorada);
    }
```

```
    @Override
    public void dibujar() {
        super.dibujar();
        dibujarBorde();
    }

    private void dibujarBorde() {
        System.out.println("Dibujando un borde a la ventana");
    }
}

// Decorador concreto: Barra de desplazamiento
class BarraDesplazamientoDecorador extends DecoradorVentana {
    public BarraDesplazamientoDecorador(Ventana ventanaDecorada) {
        super(ventanaDecorada);
    }

    @Override
    public void dibujar() {
        super.dibujar();
        dibujarBarraDesplazamiento();
    }

    private void dibujarBarraDesplazamiento() {
        System.out.println("Añadiendo una barra de desplazamiento");
    }
}

// Cliente
public class Main {
    public static void main(String[] args) {
        Ventana ventana = new VentanaBasica();

        // Decoramos con un borde
        ventana = new BordeDecorador(ventana);

        // Añadimos una barra de desplazamiento
        ventana = new BarraDesplazamientoDecorador(ventana);

        ventana.dibujar();
        // Salida:
        // Dibujando una ventana básica
        // Dibujando un borde a la ventana
        // Añadiendo una barra de desplazamiento
    }
```

```
}
```

COMPOSITE

Permite tratar objetos individuales y grupos de objetos de manera uniforme. Representa jerarquías de objetos donde los nodos compuestos (contenedores) y los objetos individuales (hojas) comparten la misma interfaz.

Es útil cuando el modelo central de una aplicación puede representarse como una jerarquía.

Ejemplo 1

Problema: Modelar objetos como Productos y Cajas que pueden contener otros objetos, formando una estructura anidada. Calcular el precio total requiere recorrer esta jerarquía, lo que puede complicarse si no se maneja bien.

Solución: Definir una interfaz común para todos los objetos (Productos y Cajas).

- Para Productos: devuelve directamente su precio.
- Para Cajas: recorre sus elementos (Productos o Cajas) y suma sus precios, manejando recursivamente la jerarquía.

Ventajas:

- Permite trabajar con objetos simples y compuestos sin distinguir sus clases concretas.
- Facilita operaciones recursivas (como cálculos) sobre todos los niveles del árbol.

Estructura:

- Hoja: Elementos simples (sin subelementos, ej. Producto).
- Contenedor: Elementos complejos (pueden contener hojas u otros contenedores, ej. Caja).
- Cliente: Interactúa con todos los elementos a través de la misma interfaz, sin importar su tipo.

Ejemplo: Sistema de archivos

Un sistema de archivos puede contener carpetas (compuestos) y archivos (hojas). Ambos comparten la misma interfaz para mostrar información.

```
// Componente base
interface Componente {
    void mostrar();
}

// Hoja: Archivo
class Archivo implements Componente {
```

```
    private String nombre;

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

    @Override
    public void mostrar() {
        System.out.println("Archivo: " + nombre);
    }
}

// Compuesto: Carpeta
class Carpeta implements Componente {
    private String nombre;
    private List<Componente> hijos = new ArrayList<>();

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

    public void agregar(Componente componente) {
        hijos.add(componente);
    }

    public void eliminar(Componente componente) {
        hijos.remove(componente);
    }

    @Override
    public void mostrar() {
        System.out.println("Carpeta: " + nombre);
        for (Componente hijo : hijos) {
            hijo.mostrar();
        }
    }
}

// Cliente
public class Main {
    public static void main(String[] args) {
        Componente archivo1 = new Archivo("documento.txt");
        Componente archivo2 = new Archivo("imagen.png");

        Carpeta carpeta1 = new Carpeta("Mis Documentos");
        carpeta1.agregar(archivo1);
```

```
        carpeta1.agregar(archivo2);

        Componente archivo3 = new Archivo("video.mp4");
        Carpeta carpeta2 = new Carpeta("Media");
        carpeta2.agregar(archivo3);

        Carpeta root = new Carpeta("Root");
        root.agregar(carpeta1);
        root.agregar(carpeta2);

        root.mostrar();
        // Salida:
        // Carpeta: Root
        // Carpeta: Mis Documentos
        // Archivo: documento.txt
        // Archivo: imagen.png
        // Carpeta: Media
        // Archivo: video.mp4
    }
}
```

Diferencias entre Decorator y Composite

Aspecto	Decorator	Composite
Propósito	Añadir dinámicamente responsabilidades a un objeto.	Representar jerarquías de objetos (parte-todo).
Relación	Uno-a-uno: Cada decorador envuelve a un objeto.	Uno-a-muchos: Cada compuesto contiene múltiples hijos.
Uso común	Modificar o extender comportamientos.	Modelar estructuras jerárquicas complejas.
Ejemplo común	Añadir estilos o funcionalidades a un componente GUI.	Árboles, sistemas de archivos, menús.

PATRONES DE COMPORTAMIENTO

Los patrones de comportamiento tratan con algoritmos y la asignación de responsabilidades entre objetos.

STRATEGY

Te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

Cuándo usarlo:

- Cuando tienes múltiples maneras de realizar una operación y quieres que sean intercambiables.
- Cuando necesitas evitar condicionales repetitivos para seleccionar algoritmos.

El patrón Strategy sugiere que tomes esa clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas estrategias.

La clase original, llamada contexto, debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia vinculado en lugar de ejecutarlo por su cuenta.

La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea. En lugar de eso, el cliente pasa la estrategia deseada a la clase contexto. De hecho, la clase contexto no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta forma, el contexto se vuelve independiente de las estrategias concretas, así que puedes añadir nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.

```
interface PaymentStrategy {
    void pay(double amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Pagando " + amount + " con tarjeta de
crédito");
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Pagando " + amount + " con PayPal");
    }
}

class PaymentContext {
    private PaymentStrategy strategy;

    public PaymentContext(PaymentStrategy strategy) {
        this.strategy = strategy;
    }
```

```
    public void executePayment(double amount) {
        strategy.pay(amount);
    }
}

// Uso
PaymentContext context = new PaymentContext(new PayPalPayment());
context.executePayment(100.0); // Pagando 100.0 con PayPal
```

STATE

El patrón State permite a un objeto cambiar su comportamiento cuando cambia su estado interno. A diferencia de Strategy, este cambio es manejado automáticamente por el propio objeto.

Cuándo usarlo:

- Cuando el comportamiento de un objeto depende de su estado interno.
- Cuando necesitas evitar condicionales grandes para manejar estados.

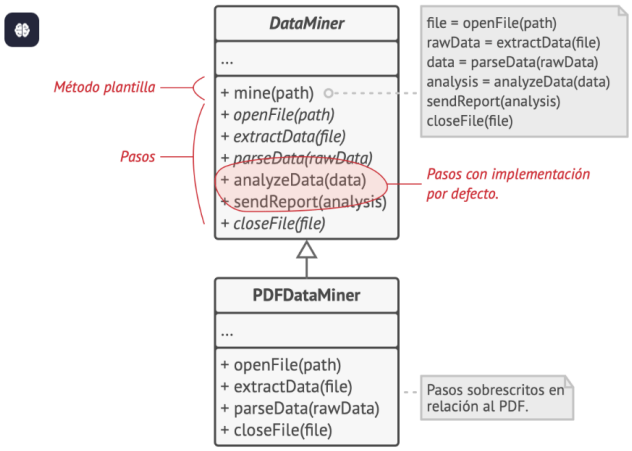
Ejemplo: Una máquina expendedora tiene diferentes comportamientos dependiendo de su estado: "Esperando moneda", "Esperando selección", "Entregando producto", etc.

```
interface State {
    void handle();
}

class IdleState implements State {
    public void handle() {
        System.out.println("Máquina en espera de moneda.");
    }
}

class ProcessingState implements State {
    public void handle() {
        System.out.println("Máquina procesando la selección.");
    }
}

class VendingMachine {
    private State state;
```

```
    public void setState(State state) {
        this.state = state;
    }

    public void request() {
        state.handle();
    }
}

// Uso
VendingMachine machine = new VendingMachine();
machine.setState(new IdleState());
machine.request(); // Máquina en espera de moneda.

machine.setState(new ProcessingState());
machine.request(); // Máquina procesando la selección.
```

TEMPLATE METHOD

Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.



El método plantilla divide el algoritmo en pasos, permitiendo a las subclases sobrescribir estos pasos pero no el método en sí.

El patrón Template Method sugiere que dividas un algoritmo en una serie de pasos, conviertas estos pasos en métodos y coloques una serie de llamadas a esos métodos dentro de un único método plantilla. Los pasos pueden ser abstractos, o contar con una implementación por defecto. Para utilizar el algoritmo, el cliente debe aportar su propia subclase, implementar todos los pasos abstractos y sobrescribir algunos de los opcionales si es necesario (pero no el propio método plantilla).

Cuándo usarlo:

- Cuando varias clases comparten la misma estructura de algoritmo, pero algunos pasos específicos varían.

- Cuando deseas evitar la duplicación de código al implementar algoritmos similares.

```
abstract class Beverage {
    // Método plantilla
    public final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    private void boilWater() {
        System.out.println("Hirviendo agua");
    }

    private void pourInCup() {
        System.out.println("Sirviendo en la taza");
    }

    // Pasos abstractos para personalización
    protected abstract void brew();
    protected abstract void addCondiments();
}

class Tea extends Beverage {
    protected void brew() {
        System.out.println("Remojando el té");
    }

    protected void addCondiments() {
        System.out.println("Añadiendo limón");
    }
}

class Coffee extends Beverage {
    protected void brew() {
        System.out.println("Preparando el café");
    }

    protected void addCondiments() {
        System.out.println("Añadiendo azúcar y leche");
    }
}

// Uso
Beverage tea = new Tea();
tea.prepareRecipe(); // Hirviendo agua, Remojando el té, etc.
```

```
Beverage coffee = new Coffee();
coffee.prepareRecipe(); // Hirviendo agua, Preparando el café, etc.
```

COMPOSITE

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

Cuándo usarlo:

- Cuando necesitas parametrizar objetos con operaciones.
- Cuando deseas implementar una funcionalidad de "deshacer".
- Cuando quieres almacenar operaciones para ejecutarlas más tarde.

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

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

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

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

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

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

class TurnOffLightCommand implements Command {
```

```
    private Light light;

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

    public void execute() {
        light.turnOff();
    }
}

// Invocador
class RemoteControl {
    private Command command;

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

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

// Uso
Light light = new Light();
Command turnOn = new TurnOnLightCommand(light);
Command turnOff = new TurnOffLightCommand(light);

RemoteControl remote = new RemoteControl();
remote.setCommand(turnOn);
remote.pressButton(); // Luz encendida

remote.setCommand(turnOff);
remote.pressButton(); // Luz apagada
```

VISITOR

Te permite separar algoritmos de los objetos sobre los que operan. Es útil cuando se necesita realizar diferentes operaciones sobre una estructura de objetos sin modificar esos objetos directamente.

Cuándo usarlo

- Cuando tienes una estructura de objetos compleja (como una jerarquía de clases) y deseas realizar operaciones sobre ella sin modificar las clases de los objetos.
- Cuando se necesita agregar nuevas operaciones a una estructura sin cambiar la

estructura misma.

El patrón Visitor sugiere que coloques el nuevo comportamiento en una clase separada llamada visitante, en lugar de intentar integrarlo dentro de clases existentes. El objeto que originalmente tenía que realizar el comportamiento se pasa ahora a uno de los métodos del visitante como argumento, de modo que el método accede a toda la información necesaria contenida dentro del objeto.

Ejemplo: Imagina un sistema de procesamiento de documentos donde tienes diferentes tipos de elementos (como párrafos, imágenes y tablas), y deseas realizar diferentes operaciones (como imprimir, exportar a HTML, o calcular el costo de impresión). En lugar de modificar las clases de los elementos cada vez que agregas una nueva operación, puedes crear un visitante para cada operación.

```
// Interfaz Visitor
interface Visitor {
    void visit(ConcreteElementA elementA);
    void visit(ConcreteElementB elementB);
}

// Elemento abstracto
interface Element {
    void accept(Visitor visitor);
}

// Elemento concreto A
class ConcreteElementA implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    public void operationA() {
        System.out.println("Operación A");
    }
}

// Elemento concreto B
class ConcreteElementB implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    public void operationB() {
        System.out.println("Operación B");
    }
}

// Visitor concreto 1
```

```java
class ConcreteVisitor1 implements Visitor {
    public void visit(ConcreteElementA elementA) {
        System.out.println("Visitor 1 operando sobre Elemento A");
        elementA.operationA();
    }

    public void visit(ConcreteElementB elementB) {
        System.out.println("Visitor 1 operando sobre Elemento B");
        elementB.operationB();
    }
}

// Visitor concreto 2
class ConcreteVisitor2 implements Visitor {
    public void visit(ConcreteElementA elementA) {
        System.out.println("Visitor 2 operando sobre Elemento A");
    }

    public void visit(ConcreteElementB elementB) {
        System.out.println("Visitor 2 operando sobre Elemento B");
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        ConcreteElementA elementA = new ConcreteElementA();
        ConcreteElementB elementB = new ConcreteElementB();

        Visitor visitor1 = new ConcreteVisitor1();
        Visitor visitor2 = new ConcreteVisitor2();

        // Aplicando visitor 1
        elementA.accept(visitor1);  // Visitor 1 operando sobre Elemento A
        elementB.accept(visitor1);  // Visitor 1 operando sobre Elemento B

        // Aplicando visitor 2
        elementA.accept(visitor2);  // Visitor 2 operando sobre Elemento A
        elementB.accept(visitor2);  // Visitor 2 operando sobre Elemento B
    }
}
```

INTERPRETER

Es un patrón de comportamiento que se utiliza para interpretar un lenguaje específico o una gramática. Este patrón define una representación de la gramática de un lenguaje y

proporciona un mecanismo para interpretar las expresiones dentro de este lenguaje.

Es útil cuando necesitas analizar o interpretar un conjunto de reglas gramaticales y realizar operaciones con base en esas reglas, como en el caso de lenguajes de programación, expresiones regulares, o consultas.

Cuándo usarlo:

- Cuando necesitas interpretar un lenguaje específico que tiene una gramática predefinida.
- Cuando las reglas del lenguaje pueden representarse como una jerarquía de objetos.
- Cuando el sistema necesita ser extendido para soportar nuevas reglas o cambios en la gramática sin modificar demasiado el código.

Ejemplo: Imagina que tienes un lenguaje sencillo que puede evaluar expresiones matemáticas como 3 + 5 * 2.

```
// Contexto que mantiene el valor actual de las variables o estado
class Context {
    private Map<String, Integer> variables = new HashMap<>();

    public void setVariable(String name, Integer value) {
        variables.put(name, value);
    }

    public Integer getVariable(String name) {
        return variables.get(name);
    }
}

// Expresión abstracta
interface Expression {
    int interpret(Context context);
}

// Expresión terminal (número)
class NumberExpression implements Expression {
    private int number;

    public NumberExpression(int number) {
        this.number = number;
    }

    public int interpret(Context context) {
        return number;
    }
}

// Expresión no terminal (operación binaria)
```

```
class AddExpression implements Expression {
    private Expression left;
    private Expression right;

    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    public int interpret(Context context) {
        return left.interpret(context) + right.interpret(context);
    }
}

// Expresión no terminal (operación de multiplicación)
class MultiplyExpression implements Expression {
    private Expression left;
    private Expression right;

    public MultiplyExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    public int interpret(Context context) {
        return left.interpret(context) * right.interpret(context);
    }
}

// Cliente que evalúa la expresión
public class InterpreterPatternExample {
    public static void main(String[] args) {
        // Definir la expresión: 3 + 5 * 2
        Expression expression = new AddExpression(
            new NumberExpression(3),
            new MultiplyExpression(
                new NumberExpression(5),
                new NumberExpression(2)
            )
        );

        // Evaluar la expresión
        Context context = new Context();
        System.out.println("Resultado: " + expression.interpret(context));  //
Resultado: 13
    }
```

```
}
```

CHAIN OF RESPONSIBILITY

Te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

Este patrón se utiliza cuando una solicitud puede ser manejada por uno de varios objetos, pero no sabes qué objeto manejará la solicitud hasta que se recorra la cadena de responsables.

Cuándo usarlo:

- Cuando se necesita pasar solicitudes a través de varios objetos y dejar que uno de ellos maneje la solicitud.
- Cuando se necesita permitir que múltiples objetos manejen una solicitud de forma flexible, sin que el cliente sepa qué objeto manejará la solicitud.
- Cuando la solicitud puede ser procesada por diferentes objetos, pero no quieres acoplar la solicitud a un objeto específico.

Ejemplo: Imagina que tienes un sistema de soporte técnico donde las solicitudes de los clientes se manejan por diferentes niveles de soporte (soporte básico, intermedio y avanzado).

```
// Manejador abstracto
abstract class SupportHandler {
    protected SupportHandler nextHandler;

    public void setNextHandler(SupportHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public abstract void handleRequest(String request);
}

// Manejador concreto (Soporte básico)
class BasicSupportHandler extends SupportHandler {
    public void handleRequest(String request) {
        if (request.equals("basic issue")) {
            System.out.println("Basic support handling request: " + request);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

// Manejador concreto (Soporte intermedio)
class IntermediateSupportHandler extends SupportHandler {
```

```
    public void handleRequest(String request) {
        if (request.equals("intermediate issue")) {
            System.out.println("Intermediate support handling request: " + request);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

// Manejador concreto (Soporte avanzado)
class AdvancedSupportHandler extends SupportHandler {
    public void handleRequest(String request) {
        if (request.equals("advanced issue")) {
            System.out.println("Advanced support handling request: " + request);
        } else {
            System.out.println("No handler for this request");
        }
    }
}

// Cliente
public class ChainOfResponsibilityExample {
    public static void main(String[] args) {
        SupportHandler basicSupport = new BasicSupportHandler();
        SupportHandler intermediateSupport = new IntermediateSupportHandler();
        SupportHandler advancedSupport = new AdvancedSupportHandler();

        // Configurar la cadena
        basicSupport.setNextHandler(intermediateSupport);
        intermediateSupport.setNextHandler(advancedSupport);

        // Enviar solicitudes
        basicSupport.handleRequest("basic issue");        // Basic support handling
request: basic issue
        basicSupport.handleRequest("intermediate issue");  // Intermediate support
handling request: intermediate issue
        basicSupport.handleRequest("advanced issue");      // Advanced support handling
request: advanced issue
        basicSupport.handleRequest("unknown issue");       // No handler for this request
    }
}
```