Semana 10: Patrones Estructurales – Composite y Decorator

## Patrón Composite

Permite componer objetos en estructuras de árbol para representar jerarquías parte-todo.

Trata objetos individuales y compuestos de forma uniforme.

Muy útil para estructuras recursivas como menús, archivos, organización, etc.

## Cuándo usar:

* Cuando quieras tratar objetos individuales y grupos de manera uniforme.
* Cuando estás modelando estructuras jerárquicas.

## Antipatrón:

Usarlo cuando no se necesita una jerarquía compleja puede sobrecomplicar el sistema.

Si las operaciones no son homogéneas entre nodos y hojas.

## Patrón Decorator

Añade funcionalidad a un objeto en tiempo de ejecución sin modificar su estructura.

Alternativa flexible a la herencia para extender funcionalidades.

## Cuándo usar:

* Cuando quieras añadir funcionalidades sin alterar clases existentes.
* Cuando necesites múltiples combinaciones de responsabilidades.

## Antipatrón:

Exceso de decoradores puede dificultar el mantenimiento y la depuración.

Decoradores anidados sin orden claro pueden generar errores inesperados.

# **🧪 2. Ejercicios con Historia de Usuario**

# **✳️ Ejercicio 1: Composite en un Sistema de Gestión de Archivos**

# **🧑‍💼 Historia de Usuario:**

# Como administrador de servidores, necesito representar carpetas y archivos en una estructura jerárquica para recorrer y gestionar todo el sistema de almacenamiento de forma uniforme.

# **💻 Código Java:**

interface ArchivoComponent {

void mostrar();

}

class Archivo implements ArchivoComponent {

private String nombre;

public Archivo(String nombre) {

this.nombre = nombre;

}

public void mostrar() {

System.out.println("Archivo: " + nombre);

}

}

class Carpeta implements ArchivoComponent {

private String nombre;

private List<ArchivoComponent> elementos = new ArrayList<>();

public Carpeta(String nombre) {

this.nombre = nombre;

}

public void agregar(ArchivoComponent componente) {

elementos.add(componente);

}

public void mostrar() {

System.out.println("Carpeta: " + nombre);

for (ArchivoComponent c : elementos) {

c.mostrar();

}

}

}

# **✳️ Ejercicio 2: Composite en Menú de Aplicación Web**

# **🧑‍💼 Historia de Usuario:**

# Como diseñador de interfaz, quiero crear menús que puedan tener submenús y opciones, y que puedan recorrerse uniformemente sin importar su nivel.

# **💻 Código Java:**

interface ElementoMenu {  
 void mostrar();  
}  
  
class Opcion implements ElementoMenu {  
 private String nombre;  
  
 public Opcion(String nombre) {  
 this.nombre = nombre;  
 }  
  
 public void mostrar() {  
 System.out.println("Opción: " + nombre);  
 }  
}  
  
class Menu implements ElementoMenu {  
 private String nombre;  
 private List<ElementoMenu> elementos = new ArrayList<>();  
  
 public Menu(String nombre) {  
 this.nombre = nombre;  
 }  
  
 public void agregar(ElementoMenu elemento) {  
 elementos.add(elemento);  
 }  
  
 public void mostrar() {  
 System.out.println("Menú: " + nombre);  
 for (ElementoMenu e : elementos) {  
 e.mostrar();  
 }  
 }  
}

# **✳️ Ejercicio 3: Decorator para Notificaciones de Usuario**

# **🧑‍💼 Historia de Usuario:**

# Como responsable de comunicación, quiero que las notificaciones puedan enviarse por email, SMS y notificaciones push sin modificar el código base.

# **💻 Código Java:**

interface Notificacion {  
 void enviar(String mensaje);  
}  
  
class NotificacionBasica implements Notificacion {  
 public void enviar(String mensaje) {  
 System.out.println("Notificación básica: " + mensaje);  
 }  
}  
  
abstract class NotificacionDecorator implements Notificacion {  
 protected Notificacion wrappee;  
  
 public NotificacionDecorator(Notificacion n) {  
 this.wrappee = n;  
 }  
  
 public void enviar(String mensaje) {  
 wrappee.enviar(mensaje);  
 }  
}  
  
class NotificacionEmail extends NotificacionDecorator {  
 public NotificacionEmail(Notificacion n) {  
 super(n);  
 }  
  
 public void enviar(String mensaje) {  
 super.enviar(mensaje);  
 System.out.println("Enviando email: " + mensaje);  
 }  
}  
  
class NotificacionSMS extends NotificacionDecorator {  
 public NotificacionSMS(Notificacion n) {  
 super(n);  
 }  
  
 public void enviar(String mensaje) {  
 super.enviar(mensaje);  
 System.out.println("Enviando SMS: " + mensaje);  
 }  
}

# **✳️ Ejercicio 4: Decorator en Sistema de Café Personalizado**

# **🧑‍💼 Historia de Usuario:**

# Como barista digital en una app de pedidos, necesito permitir a los usuarios personalizar su café con azúcar, leche, sabores, sin modificar la clase base.

# **💻 Código Java:**

interface Cafe {  
 String descripcion();  
 double costo();  
}  
  
class CafeBasico implements Cafe {  
 public String descripcion() { return "Café básico"; }  
 public double costo() { return 5.0; }  
}  
  
abstract class CafeDecorator implements Cafe {  
 protected Cafe cafe;  
  
 public CafeDecorator(Cafe cafe) {  
 this.cafe = cafe;  
 }  
  
 public String descripcion() {  
 return cafe.descripcion();  
 }  
  
 public double costo() {  
 return cafe.costo();  
 }  
}  
  
class ConLeche extends CafeDecorator {  
 public ConLeche(Cafe cafe) {  
 super(cafe);  
 }  
  
 public String descripcion() {  
 return super.descripcion() + " + Leche";  
 }  
  
 public double costo() {  
 return super.costo() + 1.5;  
 }  
}  
  
class ConAzucar extends CafeDecorator {  
 public ConAzucar(Cafe cafe) {  
 super(cafe);  
 }  
  
 public String descripcion() {  
 return super.descripcion() + " + Azúcar";  
 }  
  
 public double costo() {  
 return super.costo() + 0.5;  
 }  
}

# **🧩 3. Comparativa y Recomendaciones**

|  |  |  |
| --- | --- | --- |
| Patrón | Útil cuando… | Evitar cuando… |
| Composite | Tienes estructuras jerárquicas homogéneas | No hay jerarquía o elementos no tienen operaciones comunes |
| Decorator | Necesitas añadir funcionalidades en tiempo de ejecución | Requieres muchas decoraciones que generan complejidad |