Decorator Pattern

Tomás Felipe Melli

June 10, 2025

Índice

1 Intent

El patrón **Decorator** agrega responsabilidades adicionales a un objeto de forma dinámica. Es una alternativa flexible a la herencia para extender funcionalidad. También se conoce como **Wrapper**.

2 Motivation

Supongamos que tenemos una clase `TextView` que dibuja texto. Queremos poder agregarle funcionalidades como borde, scroll, sombra, etc., sin modificar su código ni crear muchas subclases como:

`BordeadoYConScrollTextView`, `ConSombraYConScrollTextView`, etc.

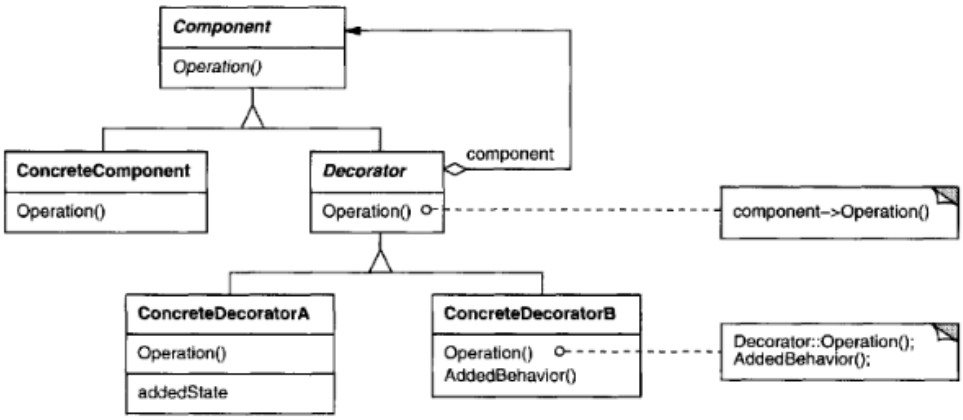
El patrón Decorator resuelve esto permitiendo componer dinámicamente decoraciones sobre un objeto base, sin alterar su clase original.

3 Applicability

Usar Decorator cuando:

- Querés agregar funcionalidades a objetos sin cambiar su clase.
- La herencia no es posible o adecuada.
- Necesitás combinar comportamientos de forma flexible.

4 Structure



5 Participants

- **Component**: Interfaz común para los objetos que pueden tener responsabilidades adicionales (`VisualComponent`).
- **ConcreteComponent**: Implementación básica del componente original (`TextView`).
- **Decorator**: Clase abstracta que mantiene una referencia al componente y define una interfaz compatible.
- **ConcreteDecorator**: Agrega responsabilidades concretas (`BorderDecorator`, `ScrollDecorator`).

6 Example

Contexto del problema

Tenemos un componente visual `TextView` que puede ser mostrado en una ventana. Queremos agregarle funcionalidades como bordes o scroll, sin modificar su clase ni usar herencia múltiple.

La solución es envolverlo con decoradores que agreguen las funcionalidades deseadas.

Componente base

```
class VisualComponent {
public:
    virtual void Draw() = 0;
    virtual ~VisualComponent() {}
};
```

Componente concreto

```
class TextView : public VisualComponent {
public:
    void Draw() override {
        // Dibuja el texto
    }
};
```

Clase Decorator

```
class Decorator : public VisualComponent {
protected:
    VisualComponent* component;

public:
    Decorator(VisualComponent* comp) : component(comp) {}

    void Draw() override {
        component->Draw(); // delega
    }
};
```

Decoradores concretos

```
class BorderDecorator : public Decorator {
private:
    int width;

public:
    BorderDecorator(VisualComponent* comp, int w)
        : Decorator(comp), width(w) {}

    void Draw() override {
        Decorator::Draw();  // dibuja el componente base
        DrawBorder(width);  // agrega borde
    }

    void DrawBorder(int w) {
        // C digo para dibujar borde
    }
};

class ScrollDecorator : public Decorator {
public:
    ScrollDecorator(VisualComponent* comp)
        : Decorator(comp) {}

    void Draw() override {
        Decorator::Draw();  // dibuja componente base
        DrawScroll();       // agrega scroll
    }

    void DrawScroll() {
        // C digo para scroll
    }
};
```

Resultado

Composición dinámica de decoradores:

```
TextView* textView = new TextView();

// Componemos decoradores
VisualComponent* component =
    new BorderDecorator(
        new ScrollDecorator(textView), 1);

component->Draw(); // Dibuja el TextView con scroll y borde
```