Decorator

Patrón estructural (POO)

Fuente principal:
https://refactoring.guru/es/design-patterns/decorator

Idea clave

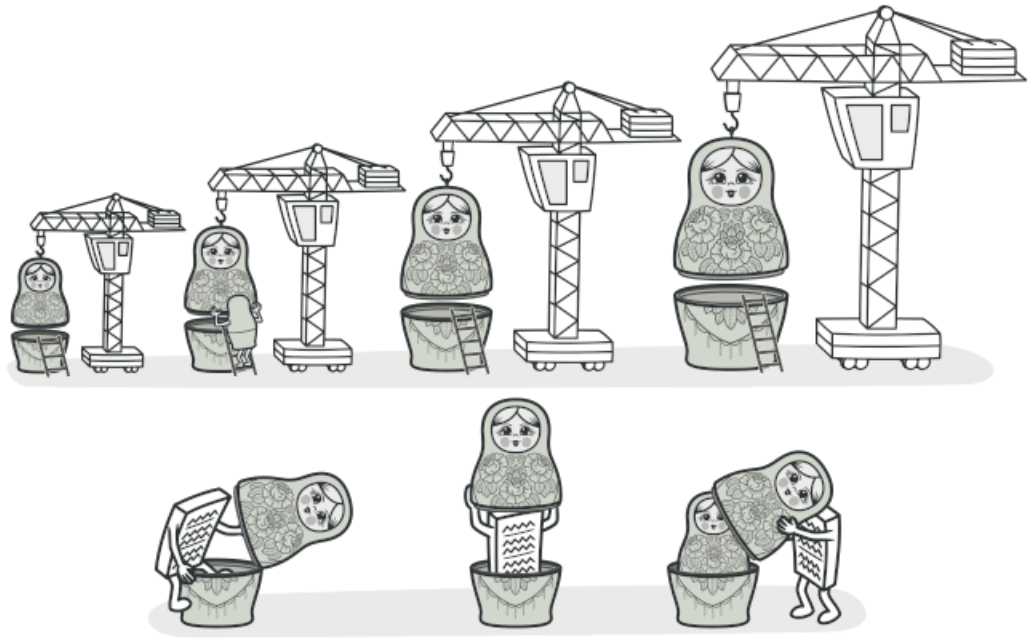
- Añade **responsabilidades** a un objeto **en tiempo de ejecución**
- Sin modificar la clase original
- Mediante **composición**: un objeto envuelve a otro y delega

Propósito

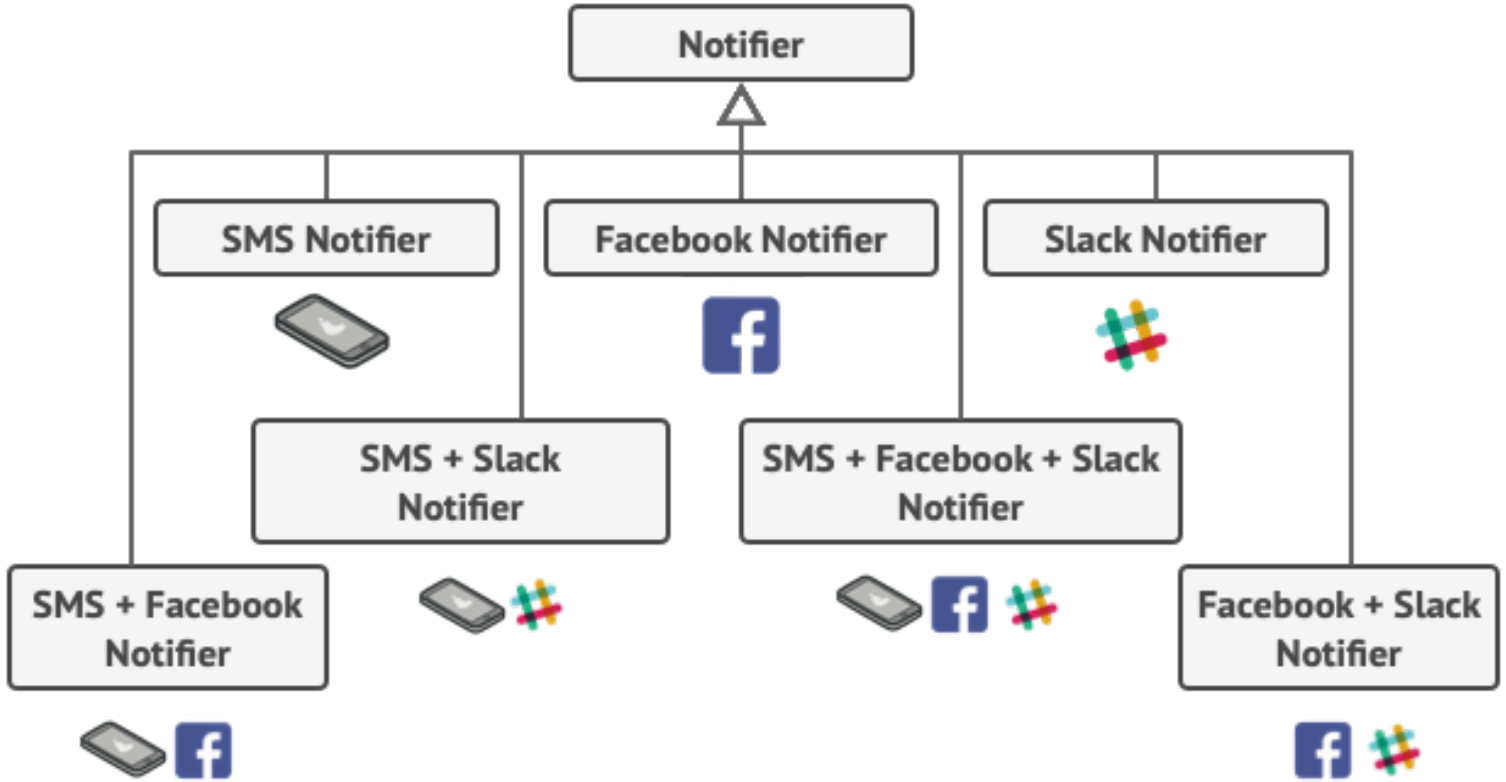
Decorator permite añadir funcionalidades a objetos colocando esos objetos dentro de **envoltorios** (decoradores) que implementan la **misma interfaz**.

- En vez de heredar y multiplicar subclases
- Compones capas:

```
DecoradorA(DecoradorB(Componente))
```



Imágenes extraídas de Refactoring.Guru



Problema

Cuando intentas añadir variantes por herencia:

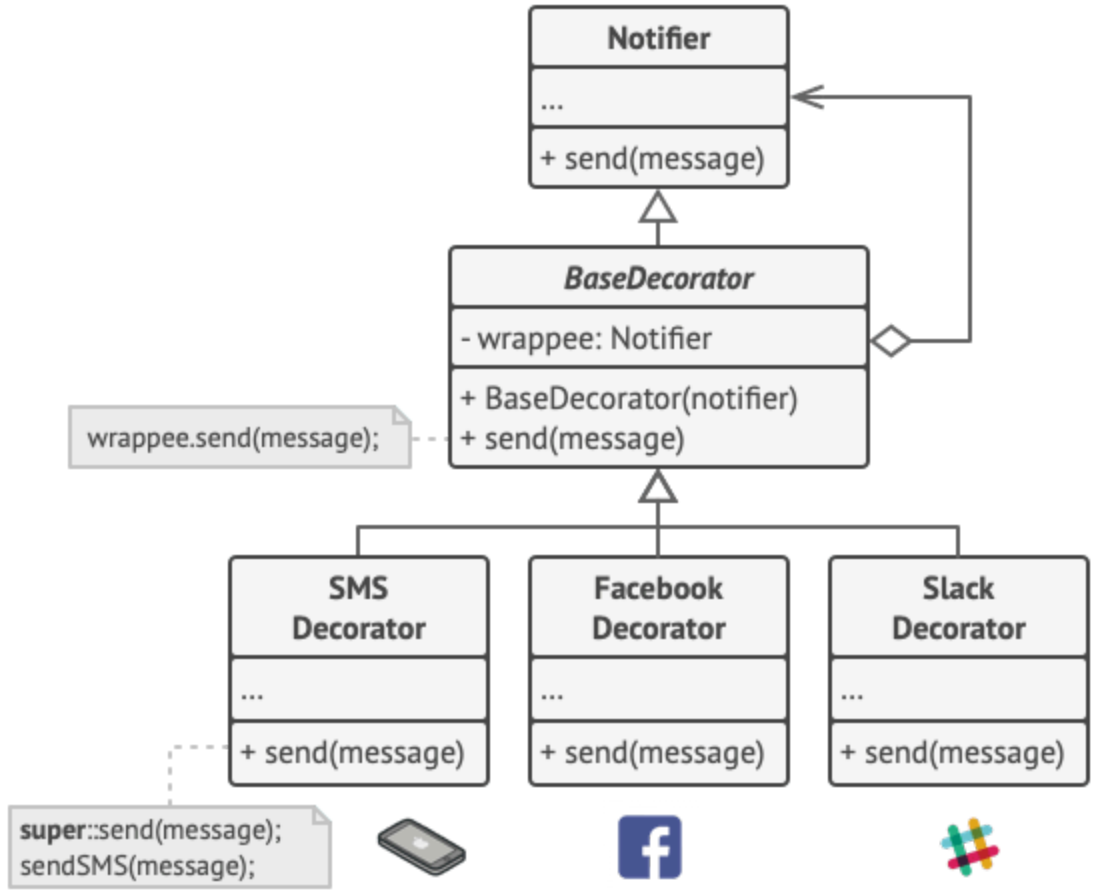
- Crecen las subclases (Email, SMS, Slack…)
- Y, peor aún, las **combinaciones** (Email+SMS, Email+Slack, …)
- Resultado: **explosión combinatoria** y código cliente rígido

Imágenes extraídas de Refactoring.Guru

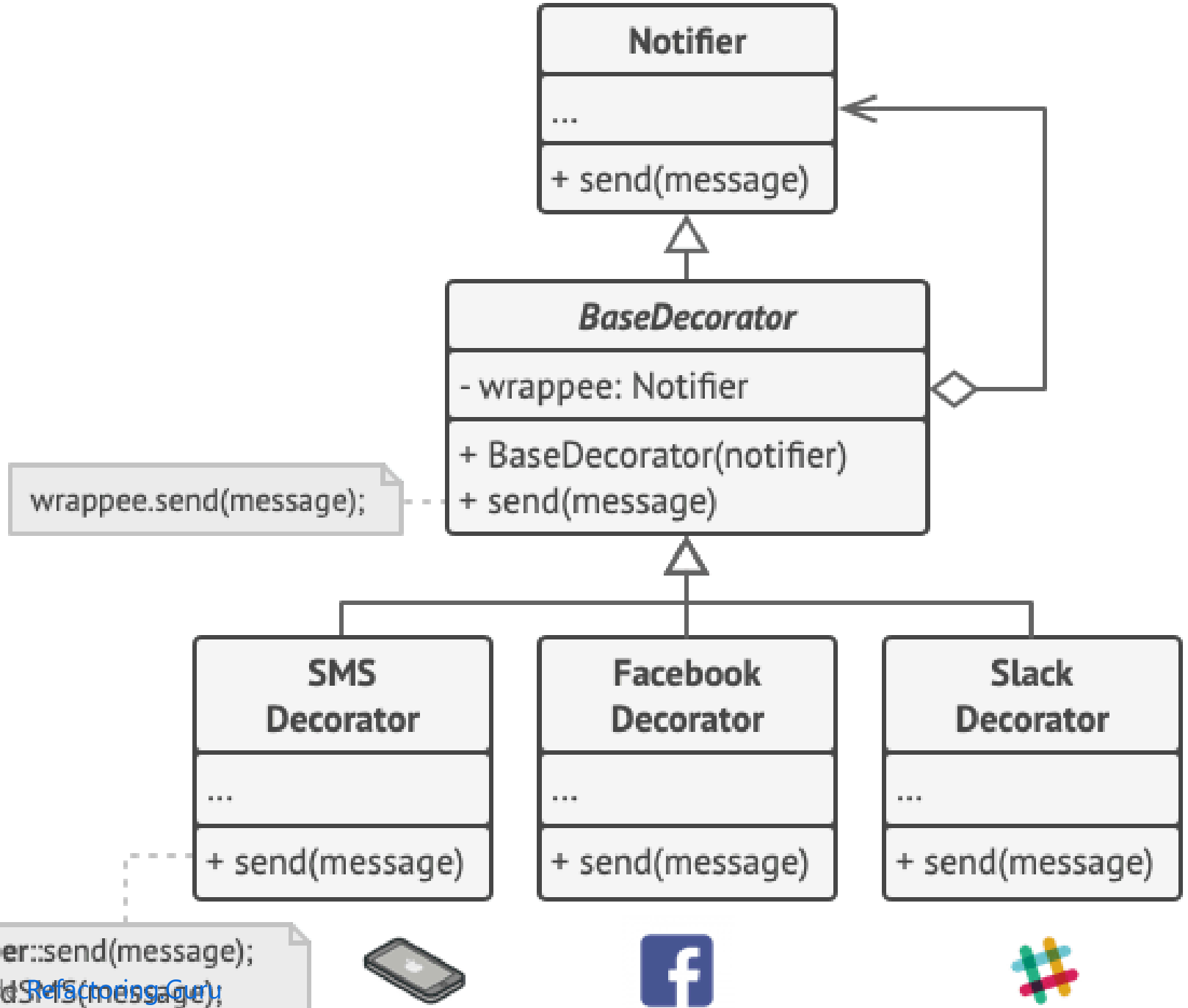
Solución

Usar composición (wrappers) para montar el comportamiento por capas:

- Todos implementan la **misma interfaz** que el componente
- Cada decorador **delegará** y añadirá trabajo **antes/después**
- Puedes cambiar la combinación en runtime (configuración)



Imágenes extraídas de Refactoring.Guru

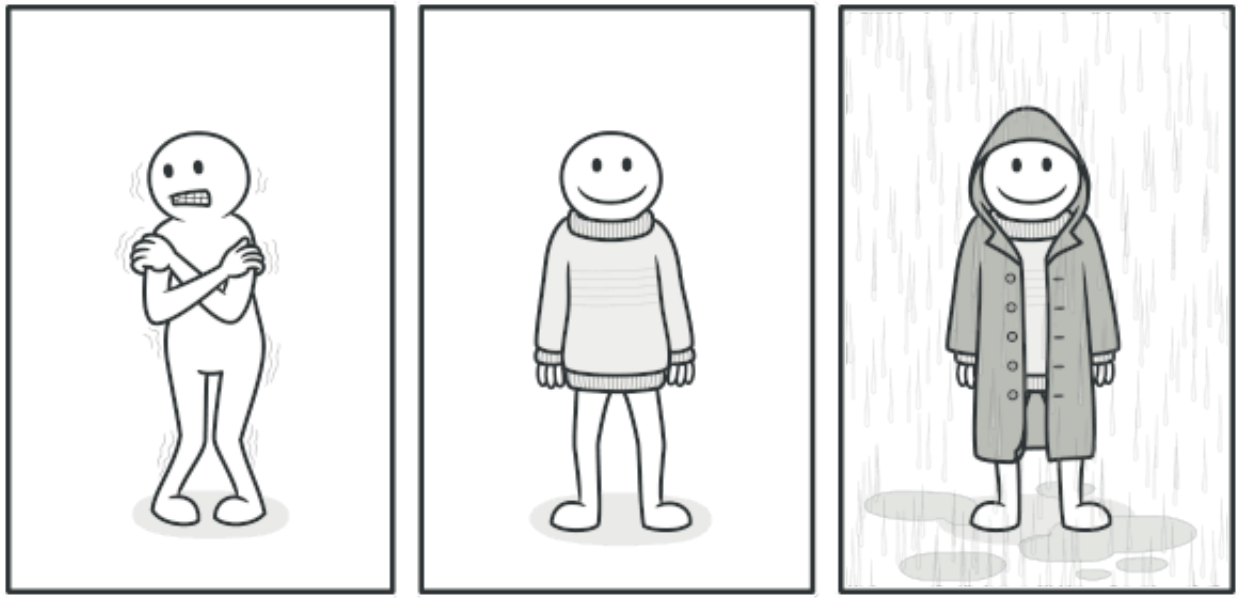


Imágenes extraídas de Refactoring Guru

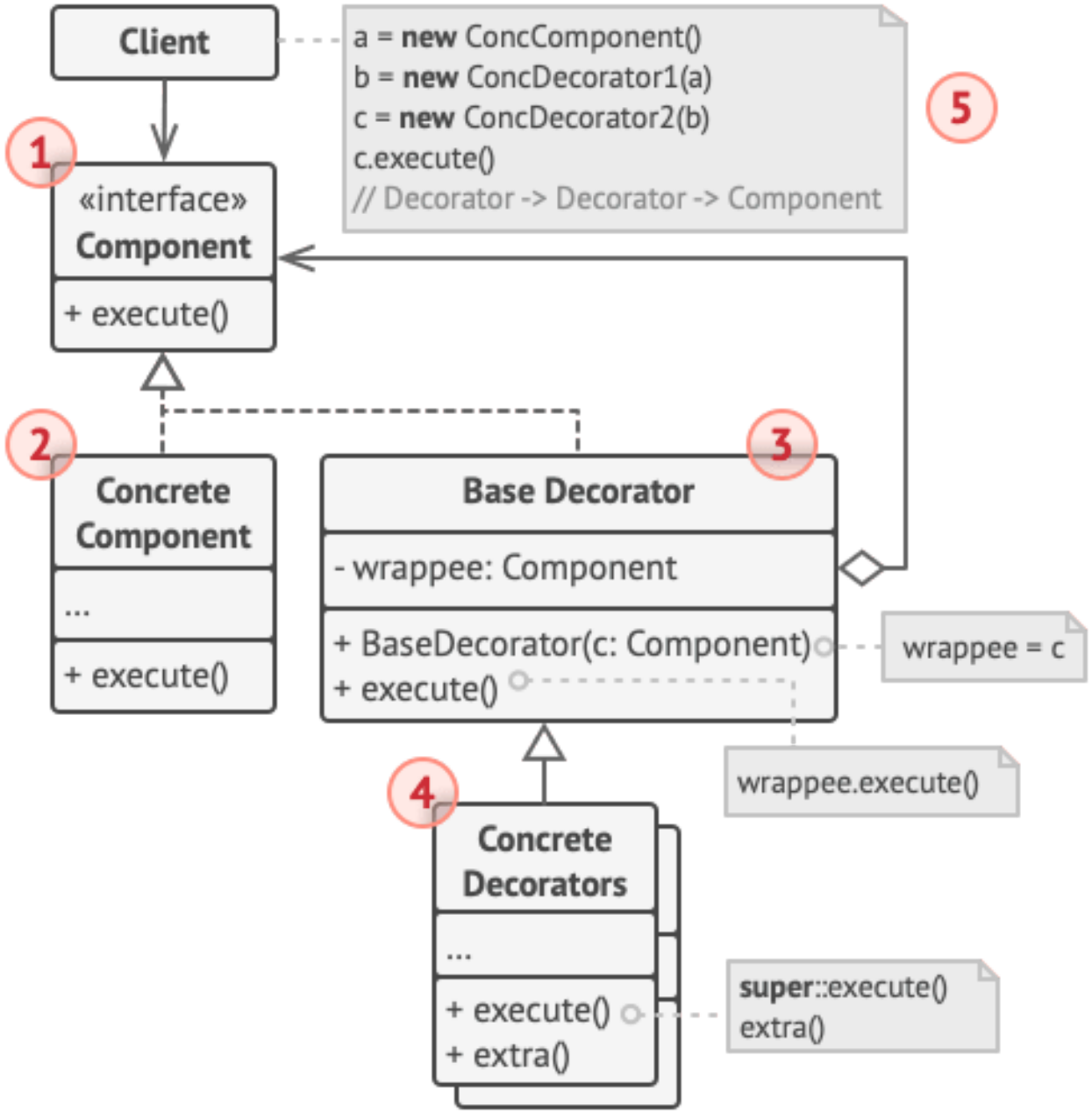
Analogía en el mundo real

Ropa como decoradores:

- “Persona” = componente
- “Jersey”, “Chaqueta”, “Impermeable” = decoradores
- Puedes poner/quitar capas según el contexto



Imágenes extraídas de Refactoring.Guru



Estructura

- **Componente**: interfaz común
- **Componente concreto**: comportamiento base
- **Decorador base**: referencia al componente y delegación
- **Decoradores concretos**: añaden responsabilidades

Imágenes extraídas de Refactoring.Guru

Código Java (puntos clave)

Basado en el ejemplo del paquete `es.uva.poo.decorator`.

- Compones decoradores **sin tocar** la clase del fichero
- El **orden importa** (comprimir→cifrar ≠ cifrar→comprimir)

```
FuenteDatos fuenteFichero = new FuenteDatosArchivo("decorator_demo.txt");

FuenteDatos fuenteDecorada = new DecoradorCifrado(
    new DecoradorCompresion(fuenteFichero),
    "clave-secreta"
);

fuenteDecorada.escribirDatos("Hola Decorator");
System.out.println(fuenteDecorada.leerDatos());
```

Aplicabilidad

Úsalo cuando:

- Quieras añadir/quitar funcionalidades **dinámicamente**
- Necesites combinar varias responsabilidades sin crear subclases
- La herencia sea mala opción (clases `final`, framework cerrado, etc.)

Evita usarlo si:

- La configuración inicial se vuelve demasiado compleja para el caso

Pros y contras

Pros:

- Amplías comportamiento sin crear nuevas subclases
- Añades/quitas responsabilidades en runtime
- Combinas comportamientos por composición
- Mejor SRP: cada “capa” en su clase

Contras:

- Quitar un wrapper concreto de una pila puede ser difícil
- El orden de decoradores puede afectar al resultado
- La configuración puede quedar “fea” (mucho `new Decorador(new Decorador(...))`)

Cierre

- Decorator = **misma interfaz + delegación + comportamiento extra**
- Una “pila” de decoradores construye la funcionalidad final
- En el ejemplo: `Cifrado(Compresión(Fichero))`