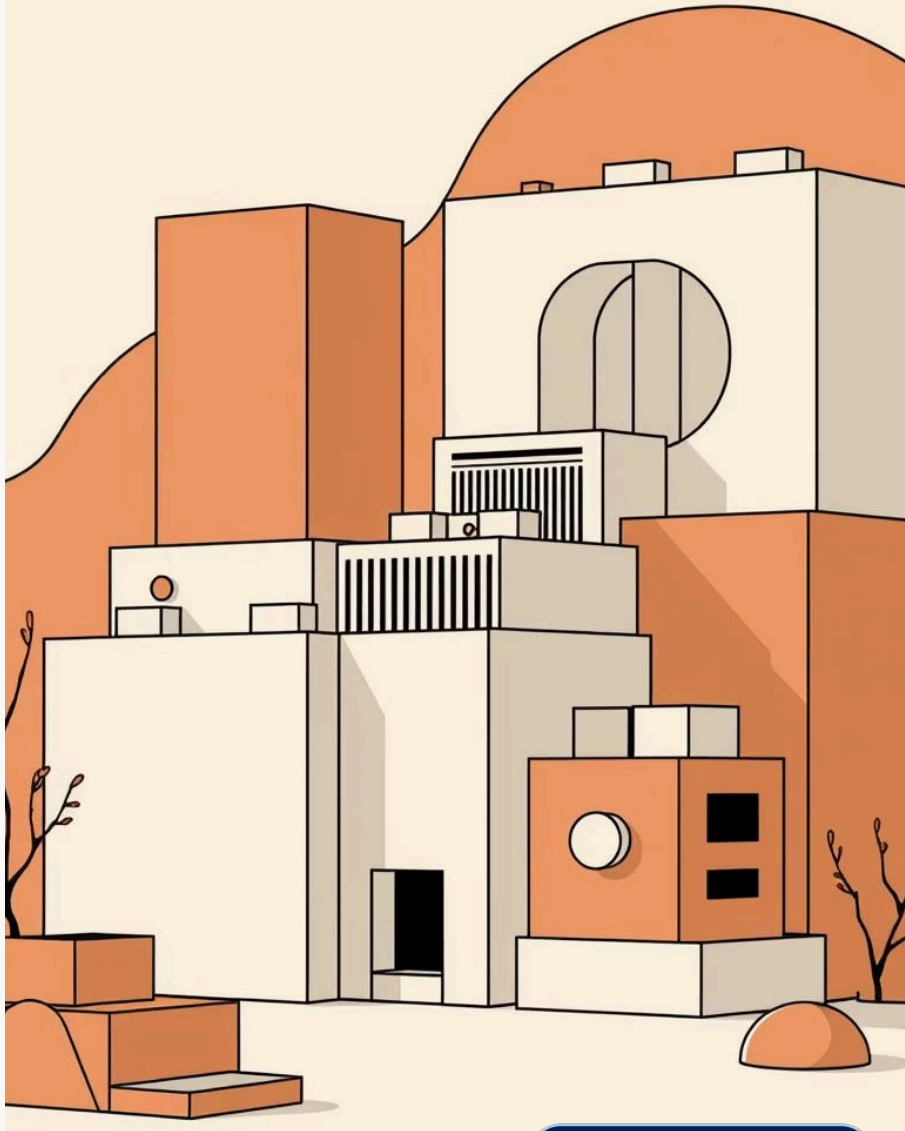
Patrón de Diseño Decorator: Añadiendo Comportamiento Dinámico

El patrón Decorator es un patrón de diseño estructural que permite añadir nuevas funcionalidades a objetos existentes sin modificar su código original. Esto se logra envolviendo los objetos con "decoradores" que implementan la misma interfaz. De esta forma, se facilita extender comportamientos de manera flexible y reutilizable en tiempo de ejecución, evitando la rigidez de la herencia tradicional.



¿Por qué usar Decorator? Beneficios Clave

Principio Abierto/Cerrado

Permite extender la funcionalidad de una clase sin modificar su código fuente original, cumpliendo con el principio "abierto para extensión, cerrado para modificación".

Flexibilidad Modular

Evita la creación de jerarquías de herencia complejas y la "explosión de subclases" que a menudo ocurre al intentar añadir comportamientos de forma rígida.

Composición Dinámica

Permite combinar múltiples decoradores para apilar y componer comportamientos en tiempo de ejecución, ofreciendo una gran flexibilidad.

Responsabilidad Única

Cada decorador se encarga de una única funcionalidad específica, lo que mejora el mantenimiento del código, su claridad y la reusabilidad de los componentes.

Ejemplo: Decorator aplicado a Componentes

Imaginemos una aplicación de procesamiento de texto donde necesitamos añadir funcionalidades como cifrado o compresión a un componente base. El patrón Decorator nos permite hacerlo de forma modular.

Componente Base: Notificador

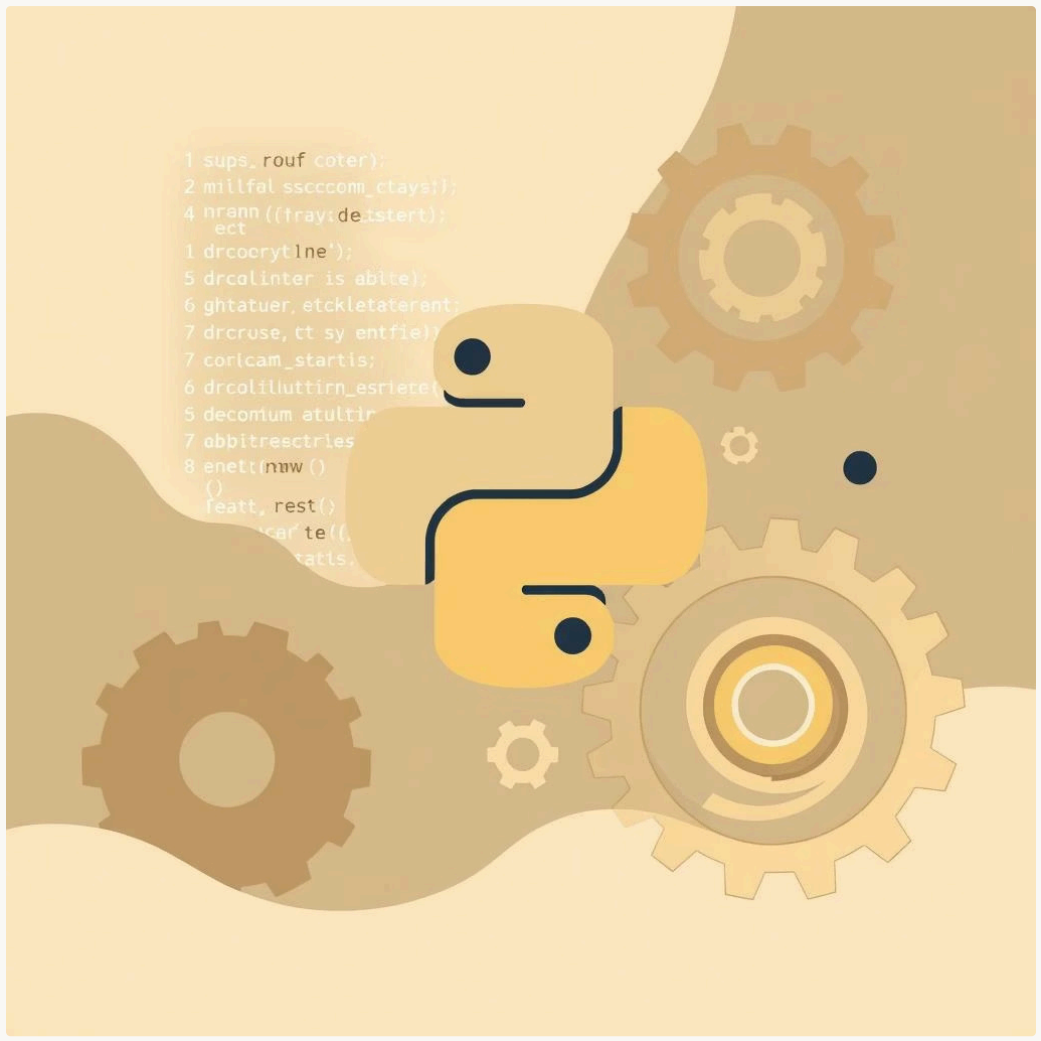
```
class Notificador:
    def enviar(self, mensaje):
        pass

class NotificadorEmail(Notificador):
    def enviar(self, mensaje):
        print(f"Enviando email: {mensaje}")
```

Decorador Concreto: NotificadorSMS

```
class NotificadorSMS(Notificador):
    def __init__(self, notificador):
        self._notificador = notificador

    def enviar(self, mensaje):
        self._notificador.enviar(mensaje)
        print(f"Enviando SMS: {mensaje}")
```



Uso del Decorator

```
notificador_base = NotificadorEmail()
# Añadimos SMS
notificador_sms = NotificadorSMS(notificador_base)
notificador_sms.enviar("¡Hola mundo!")

# Podemos añadir más capas fácilmente
#
NotificadorCifrado(NotificadorCompresion(NotificadorEmail()))
```

Este ejemplo simple demuestra cómo los decoradores "envuelven" el notificador base, añadiendo una nueva capa de funcionalidad (en este caso, enviar un SMS) sin alterar la clase original `NotificadorEmail`. Esto hace que nuestro sistema sea extensible y fácil de mantener.