Decorator

Estefani valverde
Angelica Harmon
Amanda Montero

¿Qué es un patrón de diseño Decorator?

Decorator es un patrón de diseño estructural que le permite adjuntar nuevos comportamientos a los objetos colocando estos objetos dentro de objetos envolventes especiales que contienen los comportamientos.

¿Para qué se utiliza?

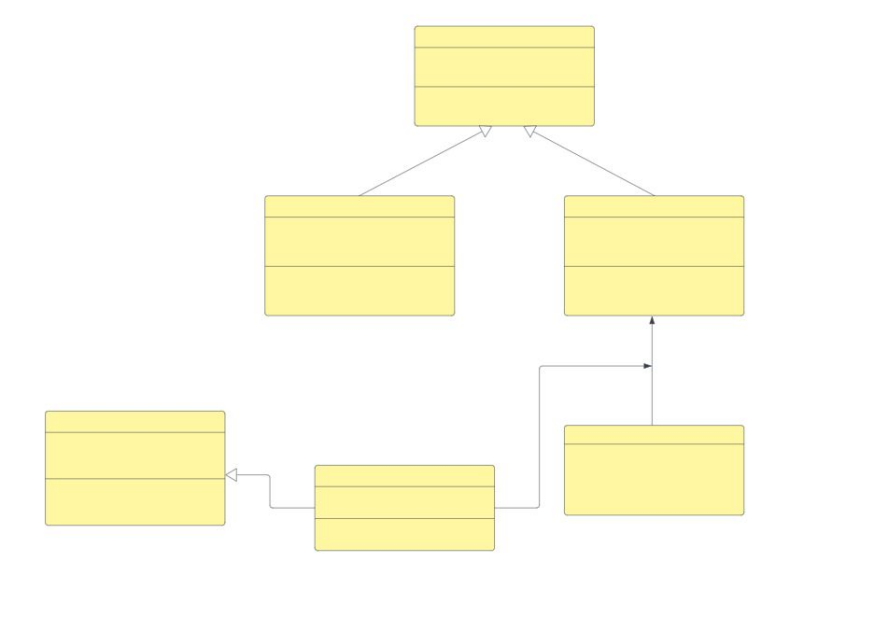
El patrón de diseño del decorador se utiliza para cambiar la funcionalidad de un objeto durante el tiempo de ejecución. Otras instancias de la misma clase no se verán afectadas, por lo tanto, cada objeto tendrá su comportamiento cambiado. El patrón de diseño Decorator es un patrón de diseño estructural que utiliza clases abstractas o interfaces con composición para implementar.

¿Cómo implementarlo?

- Crear una interfaz.
- Crear clases concretas que implementen la misma interfaz que las clases abstractas.
- Cree una clase decorator abstracta que implementa la misma interfaz que la anterior.
- Cree una clase de decorador concreto que extienda la clase de decorador abstracto mencionada anteriormente.
- Ahora puede decorar objetos de interfaz con la clase de decorador de concreto que generó anteriormente.
- Finalmente, verifique dos veces la salida.

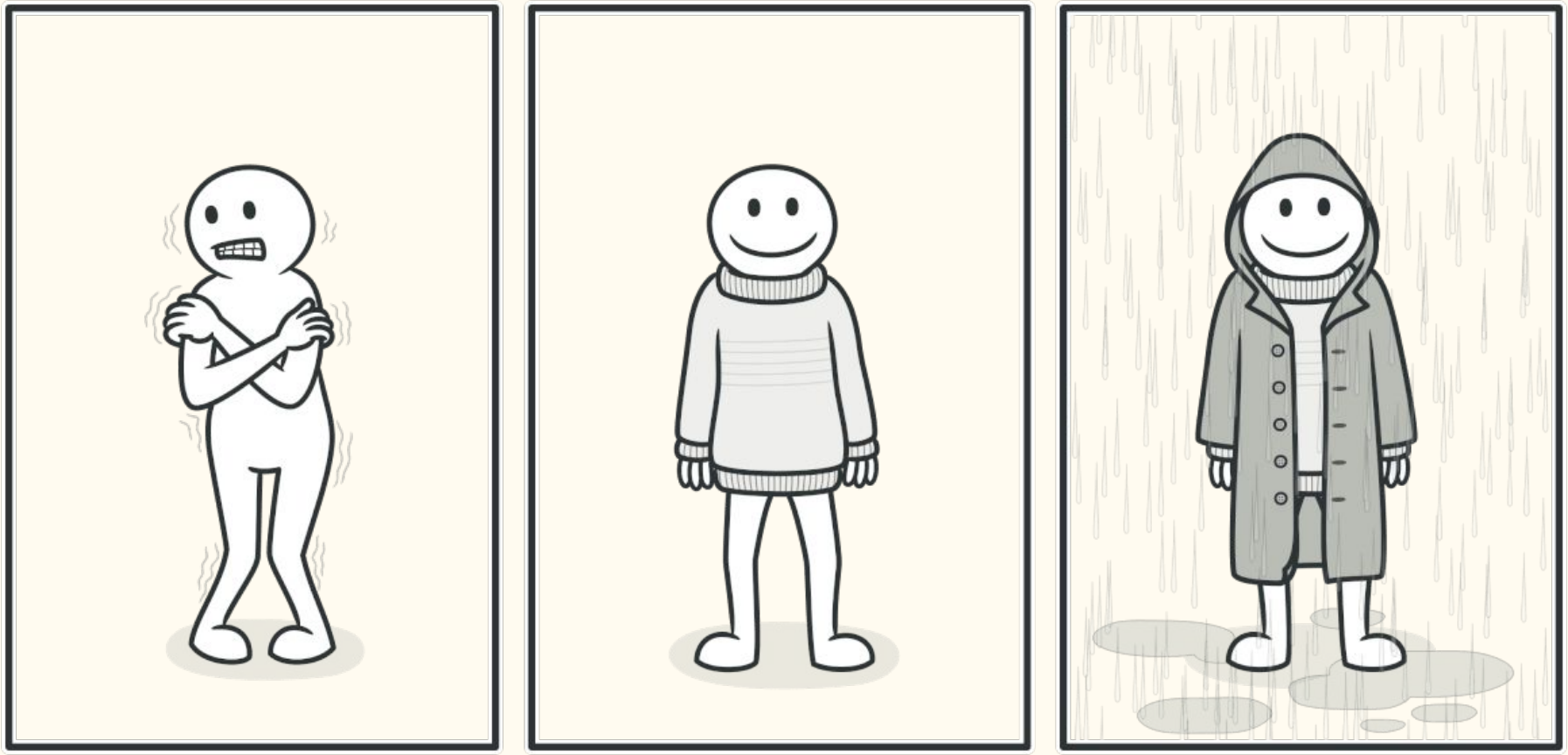
Diagrama de clases

Para comprender mejor el patrón de diseño del decorador se realiza un diagrama de clases que describe los diversos componentes y sus relaciones.



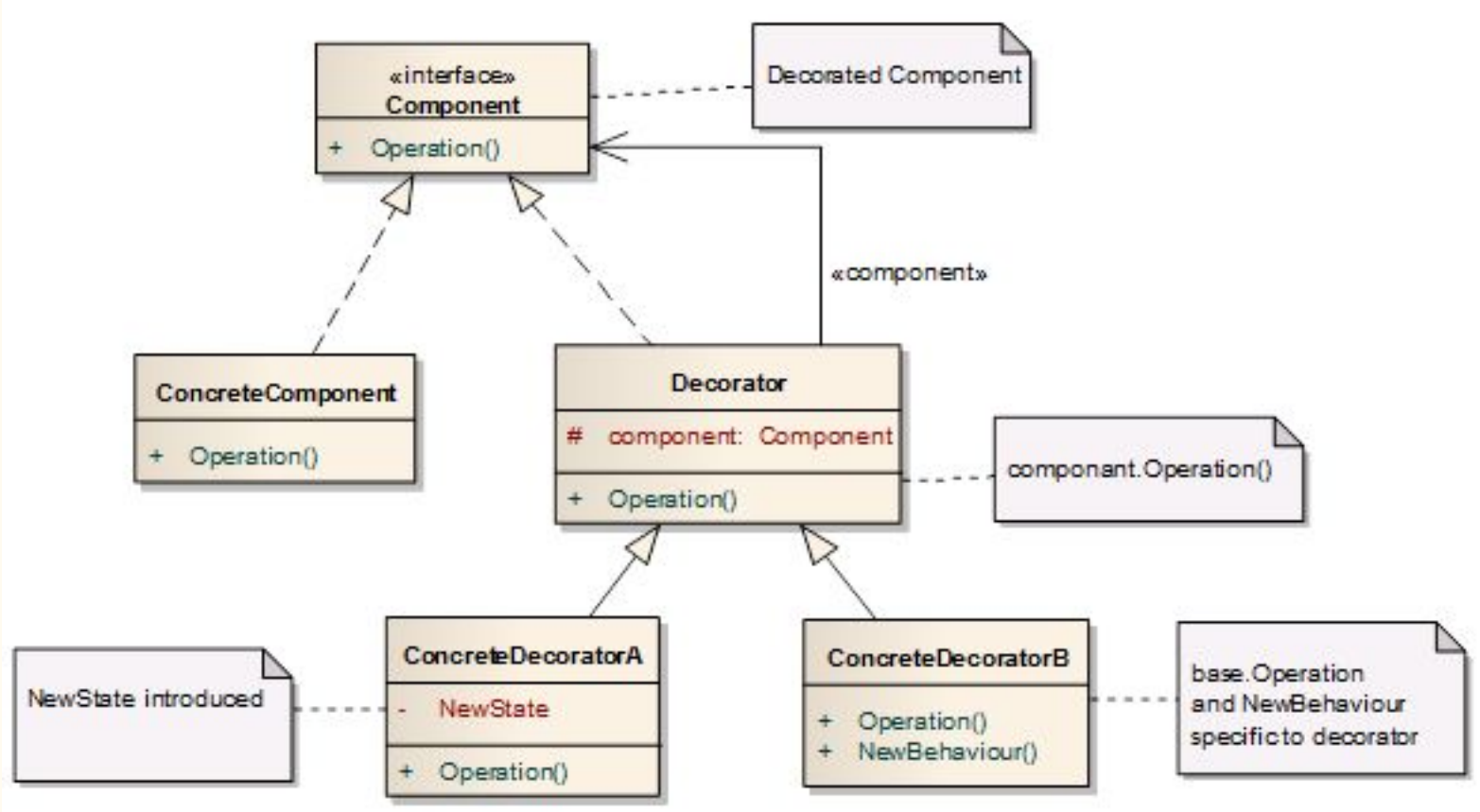
Ventajas

- **Flexibilidad:** El patrón decorador permite agregar comportamiento a un objeto individual, en lugar de a una clase entera.
- **Reutilizabilidad:** Los decoradores se pueden reutilizar para agregar comportamiento a múltiples objetos, en lugar de tener que crear una nueva subclase para cada combinación de comportamientos.
- **Estructura clara del código:** Usar el patrón decorador hace que quede claro qué comportamiento se agrega en cada nivel, lo que hace que el código sea más legible y fácil de mantener.
- **Principio Abierto/Cerrado:** Sigue el principio Abierto/Cerrado, que es uno de los principios SOLID, las clases son abiertas para extensión pero cerradas para modificación. Esto significa que los decoradores se pueden usar para agregar nueva funcionalidad a una clase existente, sin tener que modificar la clase en sí.



Desventajas

- **Complejidad:** Puede agregar una gran complejidad al código base, especialmente cuando se usan múltiples decoradores en combinación.
- **Rendimiento:** Envolver un objeto en múltiples decoradores puede agregar sobrecarga y disminuir el rendimiento, ya que cada decorador agrega su propio conjunto de métodos y campos.
- **Dificultad para entender:** Cuando se añaden muchos decoradores a un objeto, puede ser difícil entender cómo exactamente se compuso el objeto y cuáles son las responsabilidades exactas del objeto.


«interface»
Component
+ Operation()
Decorated Component
«component»
ConcreteComponent
+ Operation()
Decorator
component: Component
+ Operation()
componant.Operation()
NewState introduced
ConcreteDecoratorA
- NewState
+ Operation()
ConcreteDecoratorB
+ Operation()
+ NewBehaviour()
base.Operation
and NewBehaviour
specific to decorator

¿Cuándo usar?

- Para asignar responsabilidades a objetos de una forma transparente y dinámica sin alterar otros objetos.
- Queremos en el futuro darle un deber de objeto que tal vez desee cambiar.
- Cuando la subclasificación ya no es viable para ampliar la funcionalidad.
- Necesitamos agregar métodos adicionales a los objetos en tiempo de ejecución sin alterar el código que los usa.
- Usando herencia es inconveniente o imposible extender el comportamiento de un objeto.

Bibliografía

https://refactoring.guru/design-patterns/decorator

https://www.javatpoint.com/decorator-pattern

https://www.javadevjournal.com/java-design-patterns/decorator-design-pattern/