Decorator Pattern

Cristopher Hernández Calderón - C13632
Esteban Rojas Carranza - C06816

¿Qué es Decorator?

Patrón de diseño el cual permite agregar funcionalidades por medio de objetos encapsuladores y sin utilizar herencia.

Wrapper.

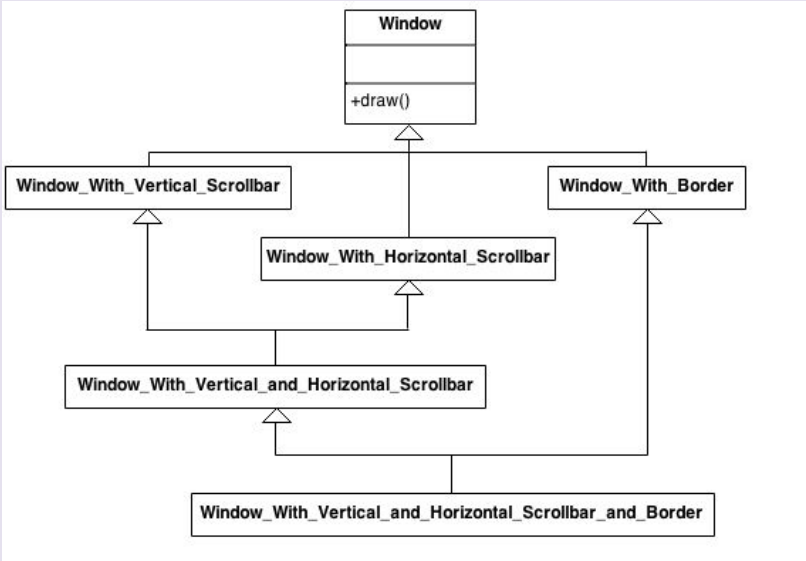
Problema

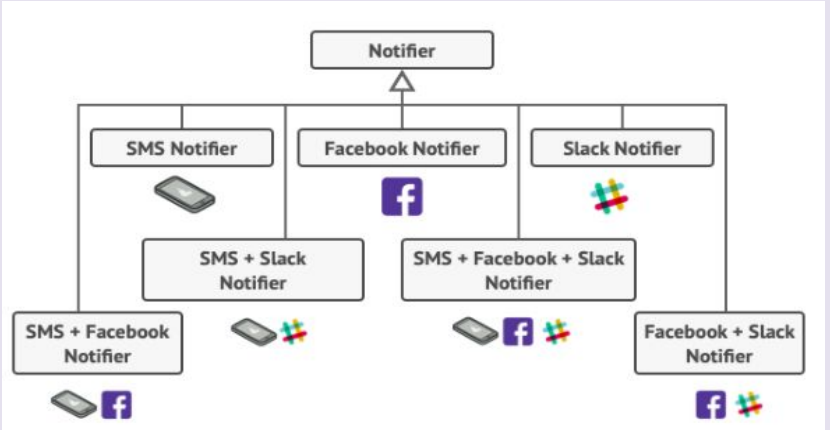
Agregar funcionalidades

Cuando se requiere agregar muchas características a un objeto.

Mezcla de funcionalidades

Cuando se necesitan muchas combinaciones de funcionalidades.


Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Horizontal_Scrollbar
Window_With_Border
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border


Notifier
SMS Notifier
Facebook Notifier
Slack Notifier
SMS + Slack Notifier
SMS + Facebook + Slack Notifier
SMS + Facebook Notifier
Facebook + Slack Notifier

Solución

Se crea un componente base a las que se le agregan decoradores que también se comportan como componentes.

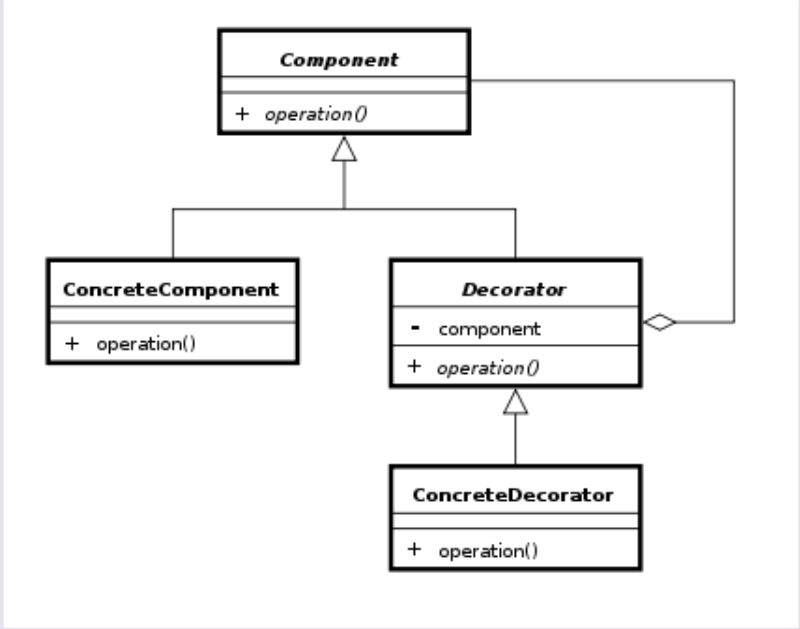
Se solucionan distintos problemas tales como:

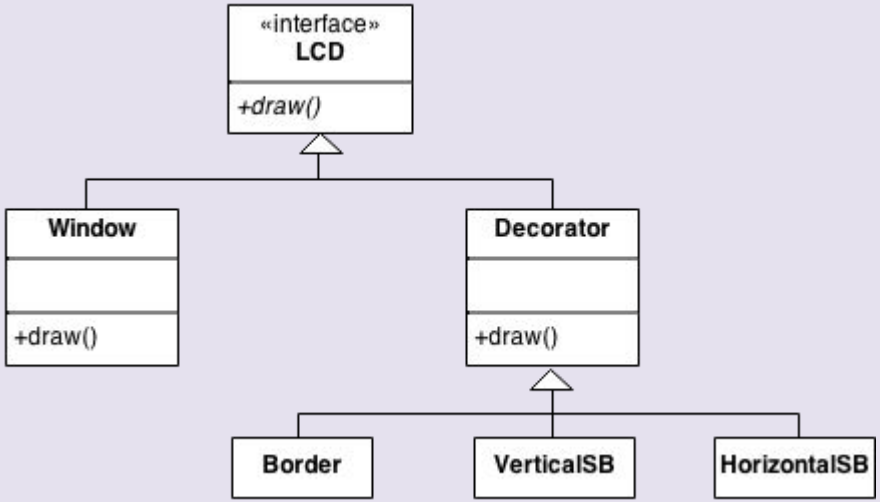
- La herencia es estática.
- Las clases sólo pueden tener una clase padre.

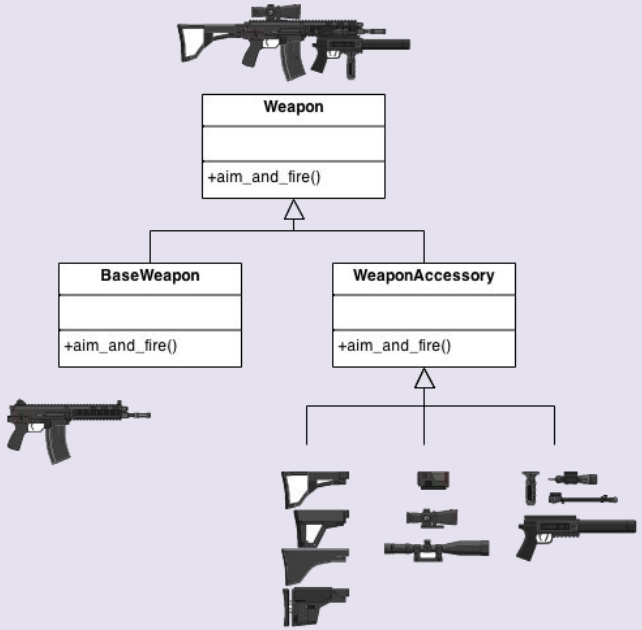
Estructura del patrón

Los componentes del patrón son:

1. Componente (interfaz): Es la interfaz común entre el objeto siendo decorado y las decoraciones.
2. Componente concreto: Es el objeto base, el cual va a ser modificado o se le van a agregar las decoraciones conforme se requiera.
3. Decorator: Es una clase abstracta que define lo mínimo para ser un decorador de la que heredan los decoradores.
4. Decorator concreto: Son los decoradores concretos que heredan de la clase Decorator y agregan las características o funcionalidades.

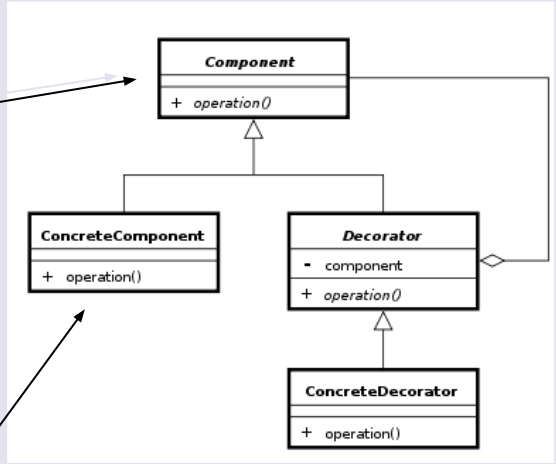


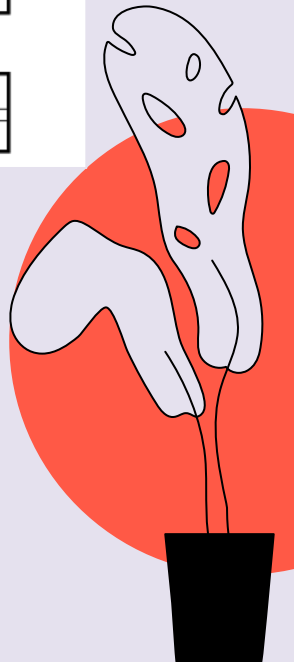

«interface»
LCD
+draw()
Window
+draw()
Decorator
+draw()
Border
VerticalSB
HorizontalSB

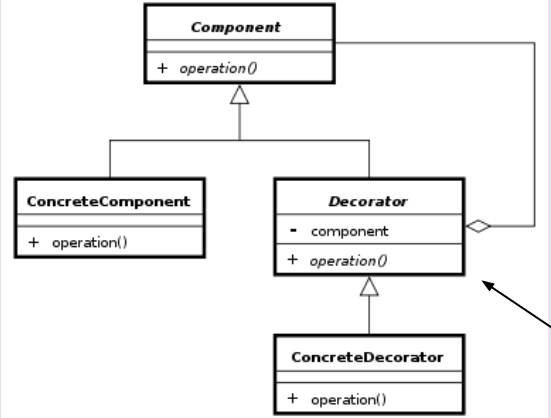

Weapon
+aim_and_fire()
BaseWeapon
+aim_and_fire()
WeaponAccessory
+aim_and_fire()

Implementación en código

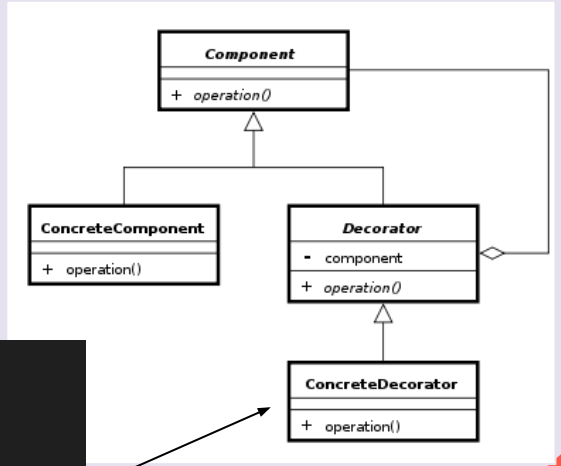
```python
class Component():

    def operation(self) -> str:
        pass

class ConcreteComponent(Component):


    def operation(self) -> str:
        return "ConcreteComponent"
```



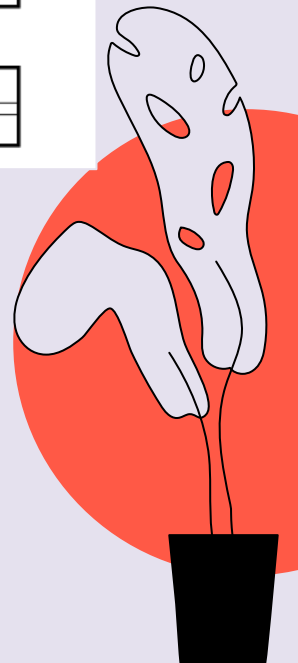


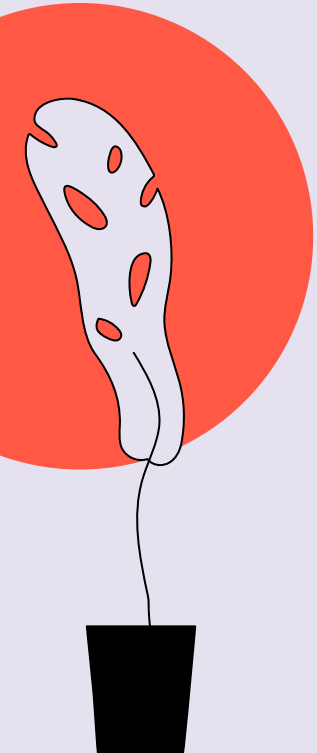



```python
class Decorator(Component):

    _component: Component = None

    def __init__(self, component: Component) -> None:
        self._component = component

    @property
    def component(self) -> Component:

        return self._component

    def operation(self) -> str:
        return self._component.operation()
```



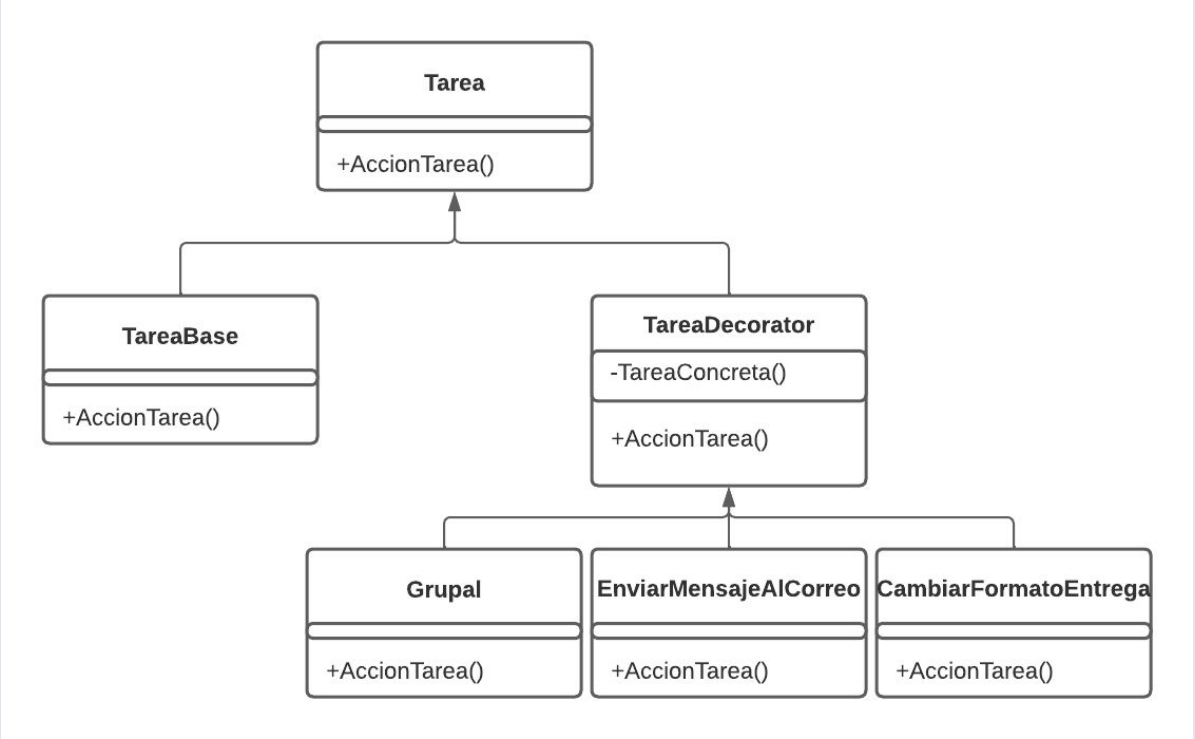

```python
class ConcreteDecoratorA(Decorator):

    def operation(self) -> str:
        return f"ConcreteDecoratorA({self.component.operation()})"

class ConcreteDecoratorB(Decorator):

    def operation(self) -> str:
        return f"ConcreteDecoratorB({self.component.operation()})"
```





```python
if __name__ == "__main__":
    simple = ConcreteComponent()
    print("Client: I've got a simple component:")
    print("\n")

    decorator1 = ConcreteDecoratorA(simple)
    decorator2 = ConcreteDecoratorB(decorator1)
    print("Client: Now I've got a decorated component:")
```

Ejemplo en mediación



Ventajas

Se puede modificar el resultado de un objeto sin crear una subclase nueva

Cumple con el principio de responsabilidad única

Se puede crear o quitar tareas de un objeto durante el tiempo de ejecución

Desventajas

Al crear muchos decoradores, el código se puede sobre complicar en vez de hacer más sencillo

Crear casos de prueba para todos los diferentes decoradores puede llegar a ser muy extenso

Como los decoradores pueden funcionar sin editar el código fuente, muchas veces se desconoce que está haciendo realmente un decorador

Consejos al utilizar

1

Implementar en la interfaz componente todos los métodos, por si no son decorados

2

Si son pocos decoradores, no vale la pena generar nuevas clases

3

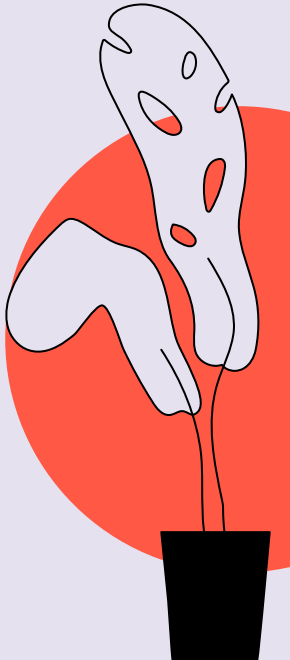
En algunos casos es importante el orden en el que se agreguen los decoradores

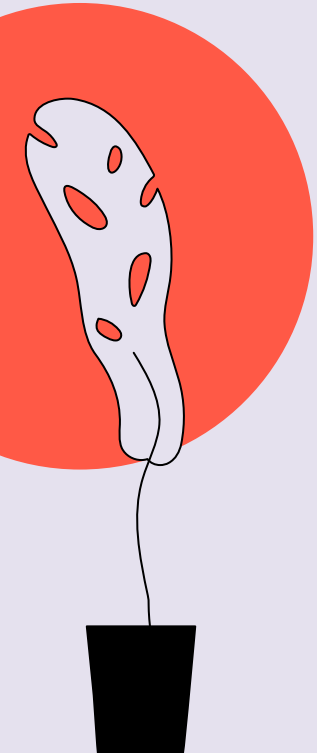
Patrones relacionados

- Adapter
- Strategy
- Composite
- Proxy

Actividad

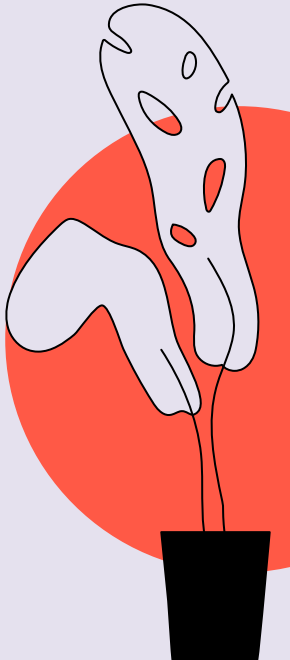
```python
from abc import ABC, abstractmethod

class Enemy(ABC):

    @abstractmethod
    def damageReceived(self):
        pass

class BaseEnemy(Enemy):

    def damageReceived(self):
        return 1000
```

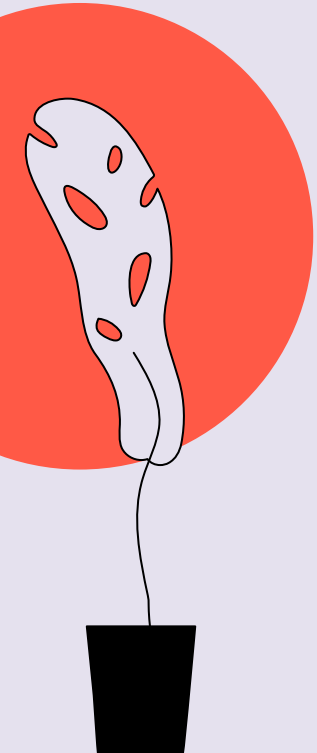




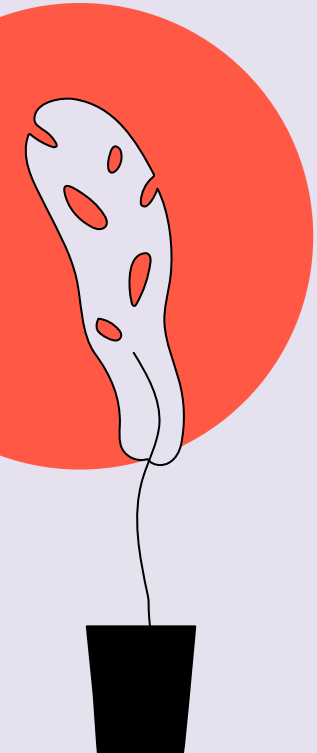
```python
class EnemywithArmour(EnemyDecorator):

    def damageReceived(self):
        return self.enemy.damageReceived() / 2

class EnemywithHelmet(EnemyDecorator):

    def damageReceived(self):
        return self.enemy.damageReceived() - 200
```

```python
if __name__ == "__main__":
    enemy = BaseEnemy()
    enemyArmour = EnemywithArmour(enemy)
    enemyHelmet = EnemywithHelmet(enemyArmour)
    print()
    print(enemyHelmet.damageReceived())
    print()

if __name__ == "__main__":
    enemy = BaseEnemy()
    enemyHelmet = EnemywithHelmet(enemy)
    enemyArmour = EnemywithArmour(enemyHelmet)
    print()
    print(enemyArmour.damageReceived())
    print()
```

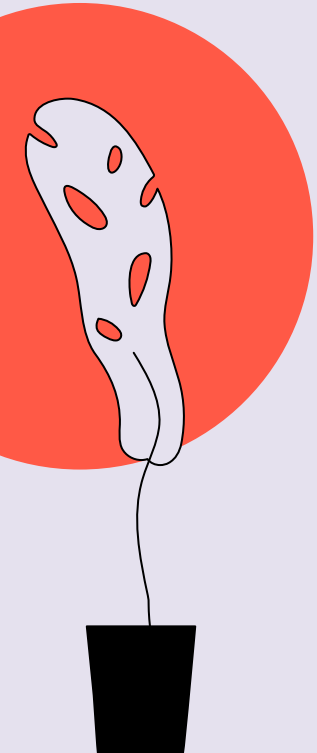




1. ¿Cuántas clases habría que implementar sin el patrón Decorator?



2. Implemente una clase de EnemywithBoots() que reduzca el daño en 250 y una clase EnemySpecial() que reduzca el daño en ⅔ si el daño es mayor a 600 y ¾ si es menor o igual a 600.



3. Genere un main que haga que el daño provocado sea de 250.

Gracias!