Decorator

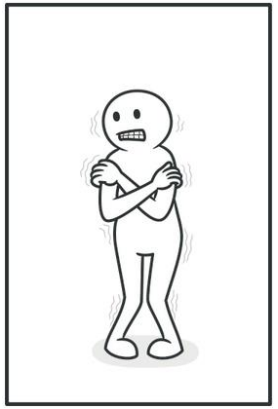
Shelton Lázio Agostinho - 115697

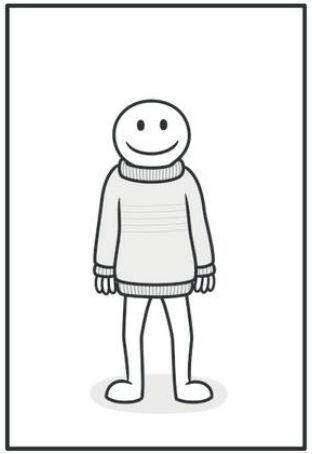
April 26th, 2024

When should we use this pattern?

This pattern is useful when we:

- need to add functionality to an object dynamically without changing its class structure.
- want to avoid inheritance to extend the functionality of an object.
- want to add functionality in a modular and reusable way.





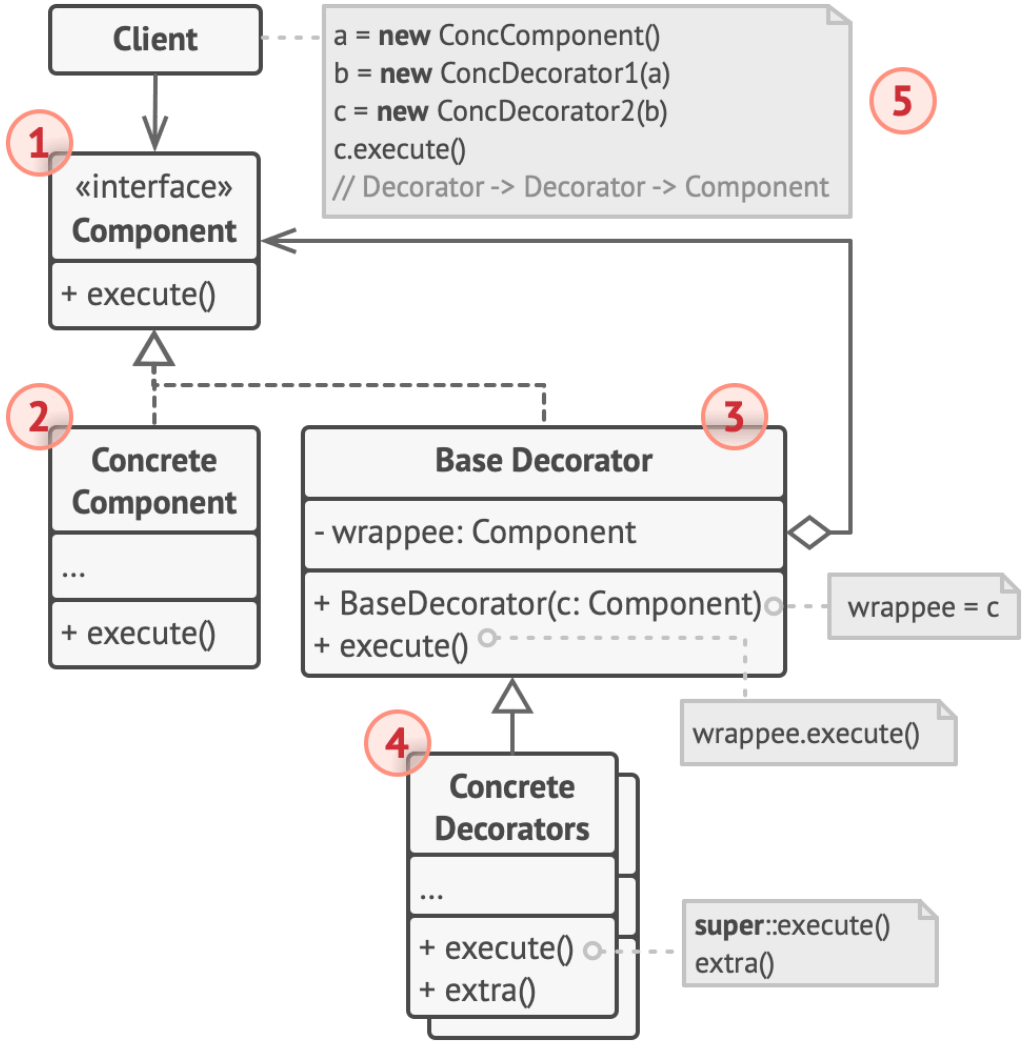
How to implement this pattern?

1. Create a common interface or abstract class for both the component and the decorators.
2. Create a concrete component class implementing the common interface.
3. Create concrete decorator classes also implementing the common interface, containing a reference to the component.

Decorator classes must forward requests to the component and can perform additional operations before/after forwarding.

(Base) Class Structure

1. .The Component declares the common interface for both wrappers and wrapped objects.
2. Concrete Component is a class of objects being wrapped.
3. Base Decorator class has a field for referencing a wrapped object.
4. Concrete Decorators define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator.
5. Client can wrap components in multiple layers of decorators, if it works with all objects via the component interface.



Code Example - Texto(1)

1. Component interface

```
public interface Texto {
        String getTexto();
}
```

2. Concrete Component

```
public class TextoSimples implements Texto {
        private String texto;
        public TextoSimples(String texto) {
                this.texto = texto;
        }
        @Override
        public String getTexto() {
                return texto;
        }
}
```

3. Base Decorator

```
public abstract class DecoradorTexto implements Texto {
        protected Texto textoDecorado;
        public DecoradorTexto(Texto textoDecorado) {
                this.textoDecorado = textoDecorado;
        }
        @Override
        public String getTexto() {
                return textoDecorado.getTexto();
        }
}
```

Code Example - Texto(2)

4. Concrete Decorators

```
public class Negrito extends DecoradorTexto {
        public Negrito(Texto textoDecorado) {
                super(textoDecorado);
        }
        @Override    public String getTexto() {
                return "**" + textoDecorado.getTexto() + "**";
        }
}

public class Italico extends DecoradorTexto {
        public Italico(Texto textoDecorado) {
                super(textoDecorado);
        }
        @Override
        public String getTexto() {
                return "*" + textoDecorado.getTexto() + "*";
        }
}
```

5. Cliente

```
public class AplicacaoNotas {
        public static void main(String[] args) {
                Texto texto = new TextoSimples("Olá, mundo!");
                System.out.println(texto.getTexto());
                texto = new Negrito(texto);
                System.out.println(texto.getTexto());
                texto = new Italico(texto);
                System.out.println(texto.getTexto());
        }
}
```