Decorator Pattern

Michal Moravik, SD20w2

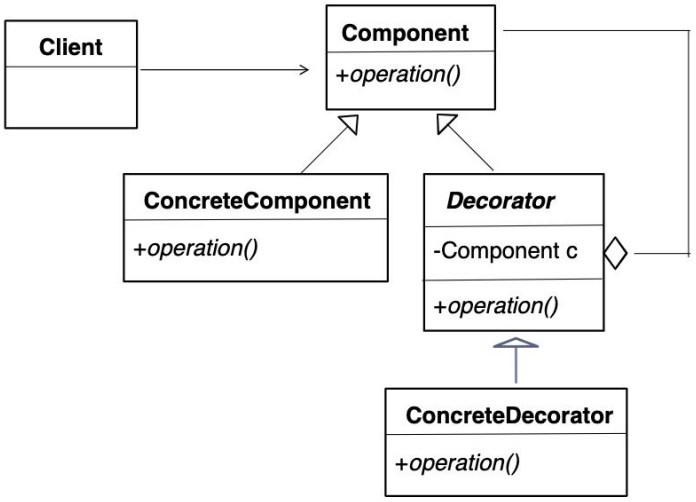
Name: Decorator Pattern

Category: Structural

Intent: Attach new (or bonus) behaviours to objects by placing these objects inside a wrapper object (decorator) that contains the behaviour.

Motivation:
Imagine a situation in which you sell drinks online. A drink can be, let's say, Whisky Sour. Whisky Sour, like the majority of cocktails, have some sugar and sour juice (lime/lemon) inside. But there is also an option for a customer to buy a sourer cocktail or less sugary one. If you would have to add all of these options as new classes which extend each drink (or cocktail), you would end up with a lot of unnecessary classes. In order to reduce the number of classes, you can have these options (sourer or less sugary cocktail, …) as classes and then apply them on any cocktail you want (as long as it implements the same interface), without exceptions.

UML:



Implementation:

```
public interface IDrink {
    int sugarLevel();
    int sourLevel();
}
```

```java
public class WhiskySour implements IDrink {

    @Override
    public int sugarLevel() { return 10; }

    @Override
    public int sourLevel() { return 15; }
}
```

```java
public abstract class DrinkDecorator implements IDrink {
    private IDrink drink;

    public DrinkDecorator(IDrink drink) { this.drink = drink; }

    @Override
    public int sugarLevel() { return drink.sugarLevel(); }

    @Override
    public int sourLevel() { return drink.sourLevel(); }
}
```

```java
public class MoreSourDrinkDecorator extends DrinkDecorator {

    public MoreSourDrinkDecorator(IDrink drink) { super(drink); }

    @Override
    public int sourLevel() { return super.sourLevel() + 10; }
}
```

```java
public class LessSugarDrinkDecorator extends DrinkDecorator {

    public LessSugarDrinkDecorator(IDrink drink) {
        super(drink);
    }

    @Override
    public int sugarLevel() { return super.sugarLevel() - 5; }
}
```

```java
public class Client {
    public static void main(String[] args) {
        WhiskySour whiskySour = new WhiskySour();
        System.out.println(whiskySour.sugarLevel());
        System.out.println(whiskySour.sourLevel());

        System.out.println("*** DECORATED BELOW ***");

        LessSugarDrinkDecorator lessSugarWhiskySour= new LessSugarDrinkDecorator(whiskySour);
        MoreSourDrinkDecorator moreSourWhiskySour = new MoreSourDrinkDecorator(lessSugarWhiskySour);
        System.out.println(moreSourWhiskySour.sugarLevel());
        System.out.println(moreSourWhiskySour.sourLevel());
    }
}
```

```
10
15
*** DECORATED BELOW ***
5
25
```

Consequences:

Pros:

- Extend behaviours without making new subclasses
- You can add/remove responsibilities from an object at runtime
- Combine several behaviours by wrapping an object into multiple decorators (see implementation in client's code)

Cons:

- If you wrap an object on many places, it is hard to remove the wrapper later on
- You need to know combinations and order in which different decorators can be applied
- It is harder to maintain code because of many similar-looking objects

Known uses:

- Decoupling behaviours from implementations
- When subclassing used to extend behaviours of an object is impractical

Related patterns:

- ...