Theory: Decorator

🕐 20 minutes   0 / 5 problems solved

[Skip this topic]   [**Start practicing**]

§1. Design problem

Sometimes it is necessary to impose additional responsibilities on a separate object rather than the whole class. A library for building graphics in the user interface should be able to add a new property, say, frames or new behavior (for example, the ability to scroll to any element). Adding new responsibilities is permissible through inheritance. However, this solution is static, and therefore not flexible enough.
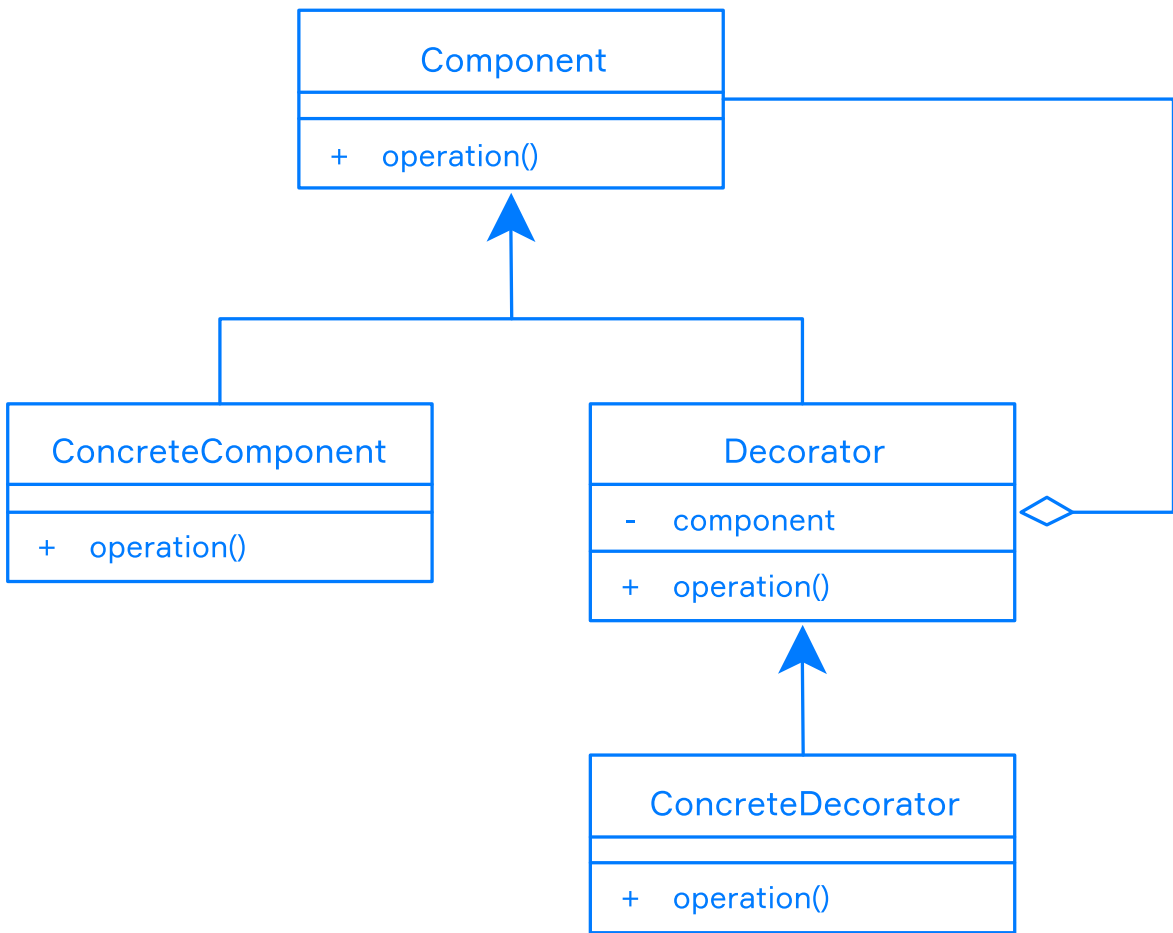
§2. Decorator

A more flexible approach is to put the component in another object called the **decorator.** The Decorator is a structural pattern used to add new responsibilities to an object dynamically without extending functionality. That lets you dynamically change the behavior of an object at runtime by wrapping it in an object of a decorator class.

Decorators are used for adding some behavior that is not part of the core functionality to all interface methods. Decorator pattern perfectly suits for the following tasks:

- caching the work results;
- measuring the execution time of methods;
- user access control.

The decorator pattern has the following components:

- *Component* is the interface for the objects that will get new responsibilities from the decorators;
- *Concrete Component* defines objects which implement the Component interface and will get new responsibilities from the concrete decorators;
- *Decorator* has reference to the Component and overridden component methods;
- *Concrete Decorator* extends Decorator class and adds new functions, properties or state without creating new classes;

The decorator pattern in JDK:

- Streams: java.io package;
- Collections: java.util package.

Current topic:

Decorator   ⋯

Topic depends on:

✓ The concept of patterns   ⋯
✗ Interface   ⋯

§3. Practice example

Let's consider a more specific example. Our components are software developers that have to make some job, that's why we create the Developer interface:

```
1    public interface Developer {
2
3        public String makeJob();
4    }
```

Next, we create a concrete developer:

```
1    public class JavaDeveloper implements Developer {
2
3        public String makeJob() {
4            return "Write Java Code.";
5        }
6    }
```

Now, describe the developer decorator to add functionality to our developers dynamically:

```
1    public class DeveloperDecorator implements Developer {
2        private Developer developer;
3
4        public DeveloperDecorator(Developer developer) {
5            this.developer = developer;
6        }
7
8        public String makeJob() {
9            return developer.makeJob();
1
0        }
1
1    }
```

The concrete decorator is the senior java developer who has an important additional responsibility: code review.

```
1    public class SeniorJavaDeveloper extends DeveloperDecorator {
2
3        public SeniorJavaDeveloper(Developer developer) {
4            super(developer);
5        }
6
7
8        public String makeCodeReview() {
9            return "Make code review.";
1
0        }
1
1
1
2        public String makeJob() {
1
3            return super.makeJob() + " " + makeCodeReview();
1
4        }
1
5    }
```

The second decorator is the team leader: being also a developer, the leader additionally has to communicate with customers and send weekly reports:

```java
public class JavaTeamLead extends DeveloperDecorator {

    public JavaTeamLead(Developer developer) {
        super(developer);
    }

    public String sendWeekReport() {
        return "Send week report to customers.";
    }


    public String makeJob() {

        return super.makeJob() + " " + sendWeekReport();

    }

}
```

Here is the final demo of the Decorator pattern:

```java
public class Task {

    public static void main(String[] args) {
        Developer developer = new JavaTeamLead(
                                new SeniorJavaDeveloper(
                                    new JavaDeveloper()));

        System.out.println(developer.makeJob());
    }

}
```

§4. Conclusion

The decorator pattern is applicable in the following cases:

- When you want to add new properties and functions to the object dynamically;
- When the extension of classes is superfluous.

🗐 Report a typo

99 users liked this theory. **21** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (15)   Hints (0)   Useful links (2)   Show discussion