Decorator

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 behaviour.

Adding new responsibilities is permissible through inheritance. However, this solution is static, and therefore not flexible enough.

Decorator

A more flexible approach is to put the component in another object called a **decorator.** The Decorator is a structural pattern used to add new responsibilities to an object dynamically without extending functionality.

This pattern lets you dynamically change the behaviour of an object at runtime by wrapping it in an object of a decorator class.

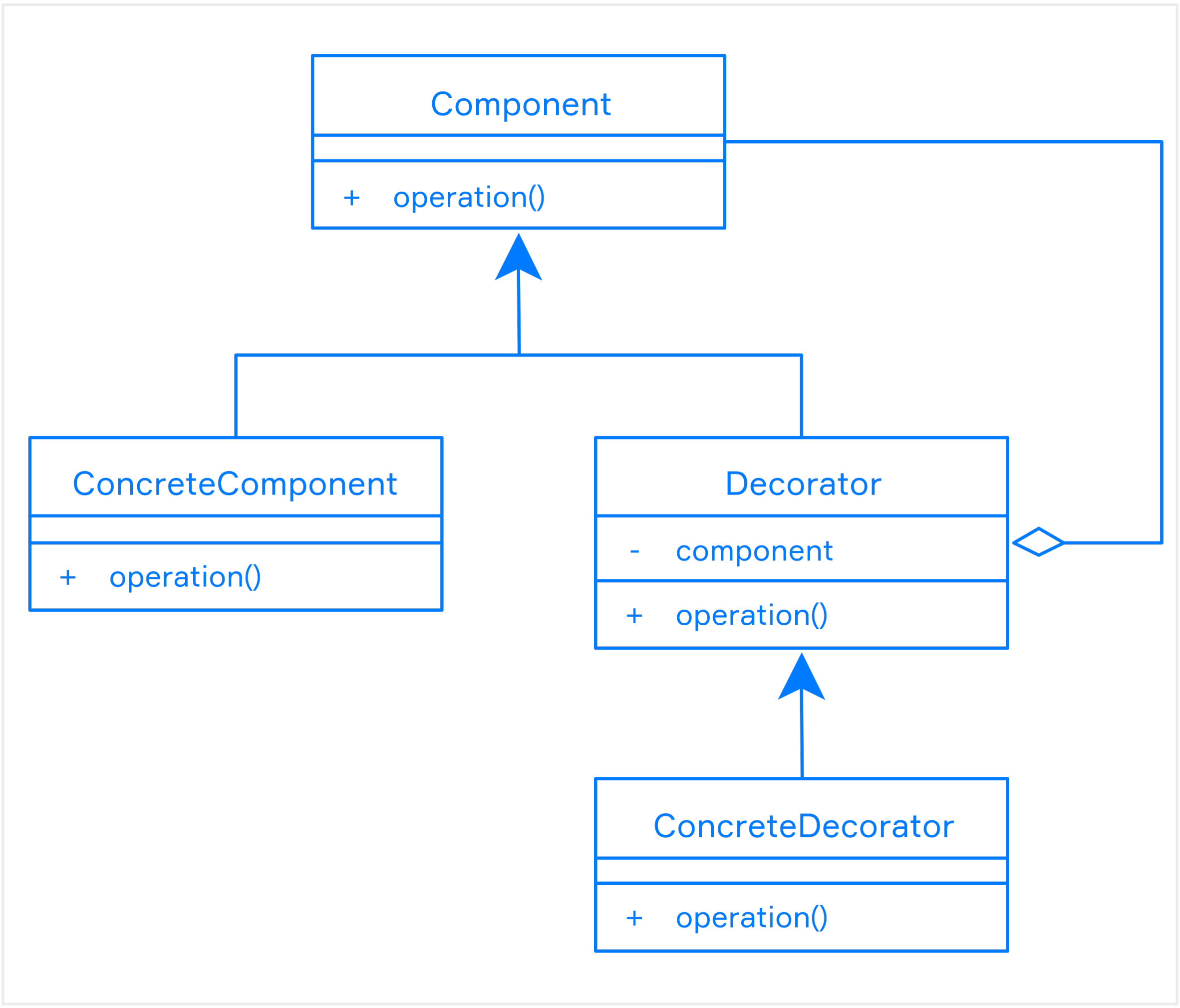
Decorators are used for adding some behaviour that is not part of the core functionality to all interface methods.

The decorator pattern perfectly suits 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 a reference to a Component and overrides component methods;
- *Concrete Decorator* extends the 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.

Practical 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:

```
public interface Developer {

    public String makeJob();
}
```

Next, we create a concrete developer:

```
public class JavaDeveloper implements Developer {

    public String makeJob() {
        return "Write Java Code.";
    }
}
```

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

```
public class DeveloperDecorator implements Developer {
   private Developer developer;

   public DeveloperDecorator(Developer developer) {
      this.developer = developer;
   }

   public String makeJob() {
      return developer.makeJob();
   }
}
```

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

```
public class SeniorJavaDeveloper extends DeveloperDecorator {

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


   public String makeCodeReview() {
      return "Make code review.";
   }

   public String makeJob() {
      return super.makeJob() + " " + makeCodeReview();
   }
}
```

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

```
public class JavaTeamLead extends DeveloperDecorator {

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

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

   public String makeJob() {
      return super.makeJob() + " " + sendWeekReport();
   }
}
```

Here is the final demo of the Decorator pattern:

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

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

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.