第3章 装饰者模式

提出问题

> 问题描述：假设某科技公司设计了一款基础型机器人（BaseRobot）可应用于执行基本任务，随着行业应用深入需要对机器人功能进行扩展，例如需要支持执行搬运、射击和飞行等高级任务。基础机器人的实现如下：

```
public class BaseRobot {
    public void execute() {
        move();
    }
    public void move() {
        System.out.println("移动");
    }
}
```

功能扩展有两种直接的方式：增强和特化。增强方式是通过增加原类的特征（属性和方法）来增强原类功能以适应新场景，具体实现如下：

```
public class BaseRobot {
    enum CommandType{
        BASE, ATTACK, CARRIAGE, RECONNOITER
    }
    public void execute(CommandType type) {
        switch(type) {
            case BASE: move();break;
            case ATTACK: {move(); shoot()};break;
            case RECONNOITER: {move(); fly()};break;
            case CARRIAGE: {move(), carry()};break;
            default:break;
        }
    }
    public void move() {
        System.out.println("移动");
    }
    public void shoot() {
        System.out.println("射击");
    }
    public void fly() {
        System.out.println("飞行");
    }
    public void carry() {
        System.out.println("搬运");
    }
}
```

特化方式是通过对父类进行扩展，针对各类应用场景设计对应的子类，具体实现如下：

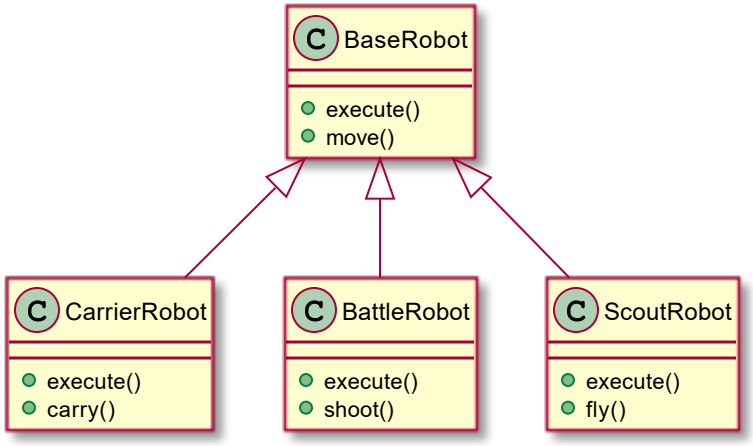
```
public class BaseRobot {
    public void execute() {
        move();
    }
    public void move() {
        System.out.println("移动");
    }
}

public class CarrierRobot extends BaseRobot {
    public void execute() {
        move();
        carry();
    }
    public void carry() {
        System.out.println("搬运");
    }
}

public class BattleRobot extends BaseRobot {
    public void execute() {
        move();
        shoot();
    }
    public void shoot() {
        System.out.println("射击");
    }
}

public class ScoutRobot extends BaseRobot {
    public void execute() {
        move();
        fly();
    }
    public void fly() {
        System.out.println("飞行");
    }
}
```

类结构如下：



> 思考：增强原类的方式与需求耦合度过高，不利于扩展，违背开闭原则；特化方式可能导致子类过多、复用效率较低等问题。两者还有一个共同的问题是程序可配置性较差，不能在程序运行的过程中动态配置功能。

模式名称

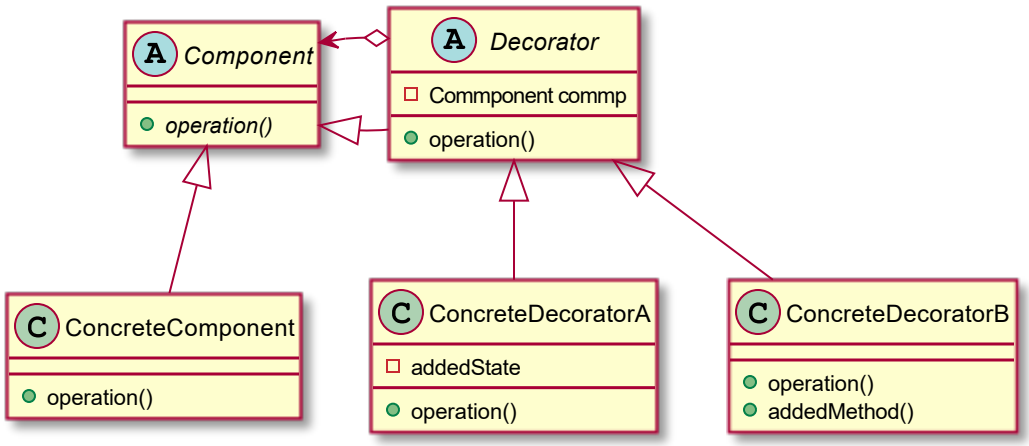
装饰者（Dcorator）或包装器（Wrapper）

设计意图

装饰者能动态地给对象添加职责（功能），相比通过类扩展增加功能的方式更灵活。

> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

设计结构

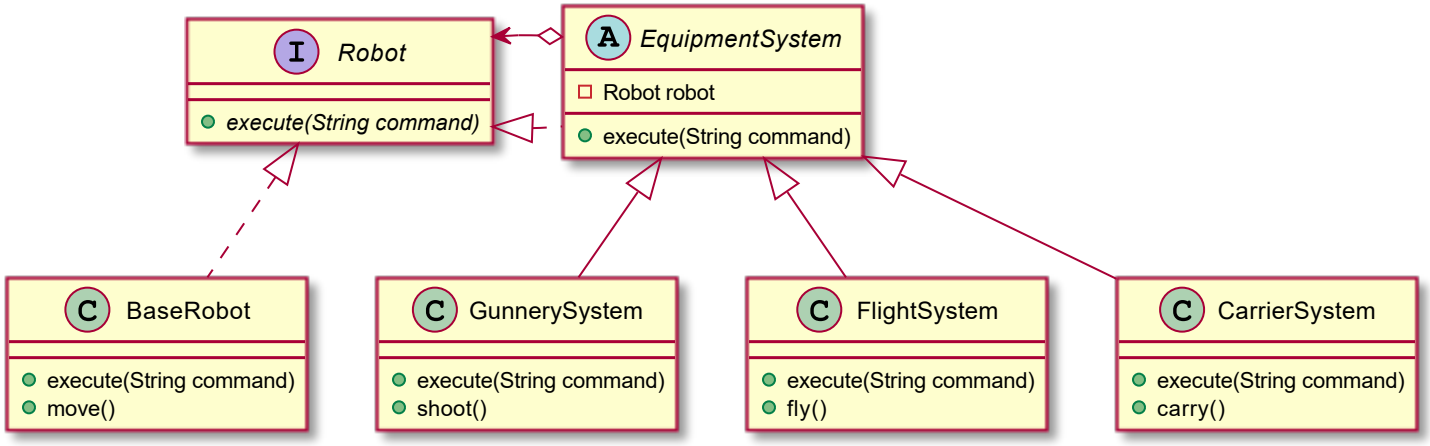


参与者：

- 抽象组件（Component）：接口或抽象类，主要用于定义可被动态扩展的功能接口。
- 具体组件（ConcreteComponent）：能被附加额外功能的组件，即被装饰组件。
- 抽象装饰器（Decorator）：包含一个组件引用，定义符合组件操作规范的接口。
- 具体装饰器（ConcreteDecorator）：具体装饰器，为组件扩展功能。

解决问题

> 设计思路：机器人的执行命令功能是要求动态扩展的，在不同业务场景执行命令所依赖的具体操作不同。因此，定义一个机器人接口对应于抽象组件，用于统一执行命令操作。基础型号机器人对应于具体组件，实现基础操作。根据现实经验，机器人一般通过配置不同的装备来增强功能，这里也为机器人基于装饰者模式设计一套装备系统。设计类图如下：



具体实现：

- 定义可动态扩展功能接口：

```
public interface Robot {
    // 机器人的基本功能，接收一个指令并执行
    public void execute(String command);
}
```

- 实现基础机器人。移动是机器人的基本功能，当传入不支持的命令时抛出异常：

```
public class BaseRobot implements Robot{
    @Override
    public void execute(String command) {
        if(command.equals("move")) {
            move();
        }else {
            throw new RuntimeException("Not Supported Command: " + command);
        }
    }
    public void move() {
        System.out.println("[完成移动]");
    }
}
```

- 设计装饰者抽象类，以实现对`execute()`功能的扩展：

```
public abstract class EquipmentSystem implements Robot{
    private Robot robot;
    public EquipmentSystem(Robot robot) {
        super();
        this.robot = robot;
    }
    @Override
    public void execute(String command) {
        robot.execute(command);
    }
    // 省略Setter和Getter方法
}
```

- 实现具体的扩展类。实现具体的扩展功能以及定义具体的命令执行逻辑，以武器装备为例：

```java
public class GunnerySystem extends EquipmentSystem {
    public GunnerySystem(Robot robot) {
        super(robot);
    }
    public void execute(String command) {
        if(command.equals("shoot")) {
            shot();
        }else {
            super.execute(command); // 调用被装饰的对象
        }
    }
    public void shot() {
        System.out.println("[完成射击]");
    }
}
```

- 其他具体装备类的实现方式类似。
- 测试装备系统：

```
public class Test {

    public static void main(String[] args) {
        Robot r = new BaseRobot();
        EquipmentSystem es = null;

        // 任务1：完成飞行侦察以及攻击特殊目标
        String[] task1 = {"fly", "move", "shoot"};
        es = new GunnerySystem(new FlightSystem(r));
        doTask(es, task1);


        // 任务2：完成飞行和运输
        String[] task2 = {"move", "fly", "carry"};
        es = new CarrierSystem(new FlightSystem(r));
        doTask(es, task2);


        // 任务3：完成飞行、运输和射击任务
        String[] task3 = {"move", "fly", "carry", "shoot"};
        es = new GunnerySystem(new CarrierSystem(new FlightSystem(r)));
        doTask(es, task3);

        // 不可能完成的任务：完成飞行与跳舞
        String[] task4 = {"fly", "dance"};
        es = new GunnerySystem(new CarrierSystem(new FlightSystem(r)));
        doTask(es, task4);
    }

    public static void doTask(EquipmentSystem es, String[] commands) {
        System.out.println("-------------开始执行任务--------------");
        for(String command:commands) {
            es.execute(command);
        }
    }
}
```

运行结果：

```
-------------开始执行任务--------------
[完成飞行]
[完成移动]
[完成射击]
-------------开始执行任务--------------
[完成移动]
[完成飞行]
[完成搬运]
-------------开始执行任务--------------
[完成移动]
[完成飞行]
[完成搬运]
[完成射击]
-------------开始执行任务--------------
[完成飞行]
Exception in thread "main" java.lang.RuntimeException: Not Supported Command: dance
```

效果与适用性