软件体系结构作业 15

姓　名：洪伟鑫

专　业：软件工程

年　级：2022 级

学　号：37220222203612

2025 年 4 月 22 日

1、什么是透明装饰模式，什么是半透明装饰模式？请举例说明。

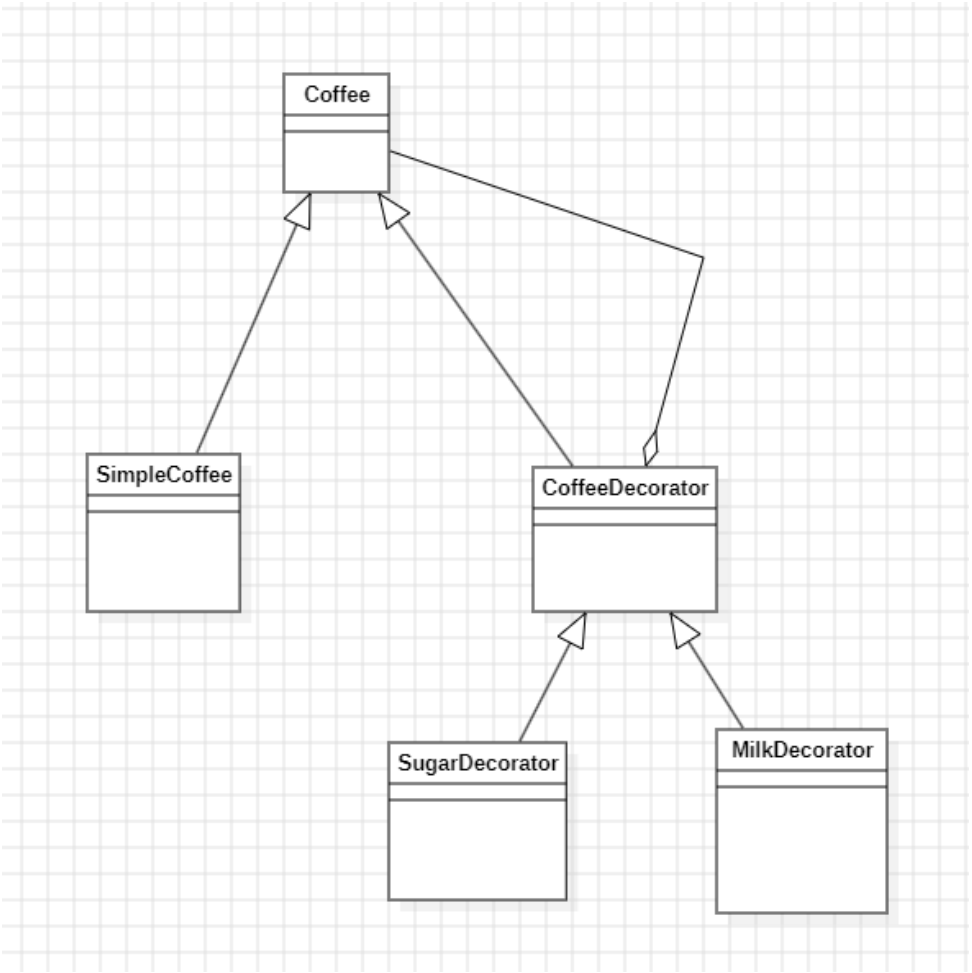
透明模式是一种结构型设计模式，旨在动态地向对象添加额外的职责。该模式的一个显著特点是其透明性：装饰器类实现了与被装饰对象相同的接口，这使得装饰器对象能够无缝地替代原始对象，客户端代码甚至察觉不到其差异。这种结构允许在运行时根据需要动态地添加或移除装饰器，为对象赋予新的行为。

在示例结构中，我们定义了一个 Coffee 接口来规范咖啡的基本行为（如获取价格和描述）。SimpleCoffee 类作为基础组件，实现了 Coffee 接口，代表一杯没有任何添加物的咖啡。

为了实现装饰功能，我们引入了抽象的 CoffeeDecorator 基类，它同样实现 Coffee 接口并持有一个 Coffee 对象的引用。具体的装饰器，如 MilkDecorator 和 SugarDecorator，继承自 CoffeeDecorator，它们在调用被包装对象（通过持有的引用）的原始方法前后，添加了各自的特定行为（例如，增加牛奶或糖的成本，并修改描述）。

运行 CoffeeShopDemo 的效果清晰地展示了该模式的工作方式：首先输出基础咖啡的价格和描述；接着，在基础咖啡上添加牛奶装饰器后，输出更新后的价格和描述；同样，在添加了糖装饰器后，也会看到相应的变化；最后，直接创建一个同时添加了牛奶和糖装饰器的咖啡实例，其价格和描述反映了所有装饰效果的累加。

装饰器模式的主要优势在于其高度的灵活性和可扩展性。它允许在不修改现有类代码的情况下动态地为对象添加新功能，完美地遵循了软件设计的开闭原则（对扩展开放，对修改关闭）。通过组合不同的装饰器，可以灵活地实现各种功能组合，有效避免了因功能排列组合而导致的子类数量爆炸性增长的问题。



代码：
普通的咖啡(基础组件，不是装饰器)：

```java
package CoffeeShop;

public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 10.0;
    }

    @Override
    public String getDescription() {
        return "简单咖啡";
    }
}
```

装饰器及具体实现：

```java
package CoffeeShop;

public abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    @Override
    public double getCost() {
        return coffee.getCost();
    }

    @Override
    public String getDescription() {
        return coffee.getDescription();
    }
}
```

```java
package CoffeeShop;

public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 1.0;
    }

    @Override
    public String getDescription() {
        return super.getDescription() + " + 糖";
    }
}
```

```java
package CoffeeShop;

public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 2.0;
    }

    @Override
    public String getDescription() {
        return super.getDescription() + " + 牛奶";
    }
}
```

在 Demo 中测试:

```java
package CoffeeShop;

public class CoffeeShopDemo {
    Run | Debug
    public static void main(String[] args) {
        // 创建基础咖啡
        Coffee coffee = new SimpleCoffee();
        System.out.println("基础咖啡: " + coffee.getDescription() + ", 价格: " + coffee.getCost());

        // 添加牛奶
        coffee = new MilkDecorator(coffee);
        System.out.println("加牛奶: " + coffee.getDescription() + ", 价格: " + coffee.getCost());

        // 添加糖
        coffee = new SugarDecorator(coffee);
        System.out.println("加糖: " + coffee.getDescription() + ", 价格: " + coffee.getCost());

        // 直接创建加牛奶和糖的咖啡
        Coffee coffeeWithMilkAndSugar = new SugarDecorator(new MilkDecorator(new SimpleCoffee()));
        System.out.println("直接创建加牛奶和糖的咖啡: " + coffeeWithMilkAndSugar.getDescription() +
                ", 价格: " + coffeeWithMilkAndSugar.getCost());
    }
}
```

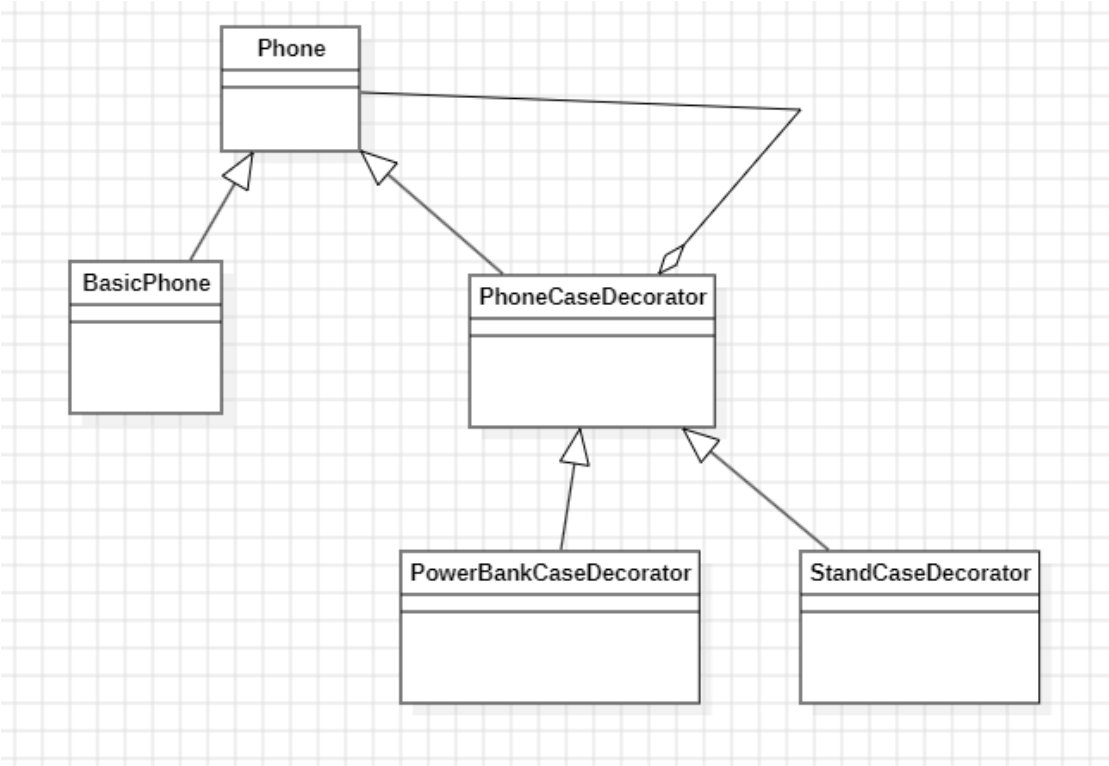
运行结果如下:

```
PS E:\大三资料\大三下课程资料\体系结构\PPT\Code\Decorator> javac -encoding UTF-8 CoffeeShop/*.java
PS E:\大三资料\大三下课程资料\体系结构\PPT\Code\Decorator> java CoffeeShop.CoffeeShopDemo
基础咖啡: 简单咖啡, 价格: 10.0
加牛奶: 简单咖啡 + 牛奶, 价格: 12.0
加糖: 简单咖啡 + 牛奶 + 糖, 价格: 13.0
直接创建加牛奶和糖的咖啡: 简单咖啡 + 牛奶 + 糖, 价格: 13.0
```

半透明模式

设定了一个 Phone 接口，它定义了如打电话和发短信等基本通信功能，由 BasicPhone 类实现这些基础能力。接着，我们引入了具体的装饰器，例如 StandCaseDecorator（支架手机壳）和 PowerBankCaseDecorator（充电宝手机壳）。这些装饰器类虽然也实现了 Phone 接口，从而保留了原有的打电话和发短信功能，但它们各自还添加了独特的新功能：StandCaseDecorator 增加了 useStand()（使用支架）和 adjustStandAngle()（调整支架角度）方法；而 PowerBankCaseDecorator 则增加了 chargePhone()（给手机充电）、chargePowerBank()（给充电宝充电）以及 getBatteryLevel()（查看电量）等方法。

这种模式被称为“半透明”的原因在于其部分失去了透明性。虽然客户端仍然可以将装饰后的对象视为原始的 Phone 类型来调用基础的打电话和发短信功能，但如果想要使用装饰器新增的特定功能（如使用支架或给手机充电），客户端代码就必须明确知道当前使用的是哪个具体的装饰器类型，并通常需要进行类型转换才能调用这些新增的方法。这与完全透明的装饰器模式（客户端无需关心具体装饰器类型即可使用所有功能）形成了对比。



这样只能使用接口中定义的方法

```
Phone phone = new StandCaseDecorator(new BasicPhone("iPhone"));
phone.call("10086");  // 可以
// phone.useStand();  // 不可以，因为 Phone 接口中没有这个方法
```

必须这样才能使用新增的方法

```
StandCaseDecorator standCase = new StandCaseDecorator(new BasicPhone("iPhone"));
standCase.useStand();           // 可以使用新增的方法
standCase.adjustStandAngle(45); // 可以使用新增的方法
```

代码:

```java
package PhoneCaseDecorator;

public interface Phone {
    void call(String number);

    void sendMessage(String number, String message);
}
```

```java
package PhoneCaseDecorator;

public abstract class PhoneCaseDecorator implements Phone {
    protected Phone phone;

    public PhoneCaseDecorator(Phone phone) {
        this.phone = phone;
    }

    @Override
    public void call(String number) {
        phone.call(number);
    }

    @Override
    public void sendMessage(String number, String message) {
        phone.sendMessage(number, message);
    }
}
```

抽象的 Decorator 只能基本的收发短信和打电话，不能充电或使用手机支架等

以充电为例:

```java
package PhoneCaseDecorator;

public class PowerBankCaseDecorator extends PhoneCaseDecorator {
    private int batteryLevel;

    public PowerBankCaseDecorator(Phone phone) {
        super(phone);
        this.batteryLevel = 100;
    }

    // 新增方法：给手机充电
    public void chargePhone() {
        if (batteryLevel > 0) {
            System.out.println("正在给手机充电，充电宝剩余电量：" + batteryLevel + "%");
            batteryLevel -= 20;
        } else {
            System.out.println(x:"充电宝电量不足，请先给充电宝充电");
        }
    }

    // 新增方法：给充电宝充电
    public void chargePowerBank() {
        batteryLevel = 100;
        System.out.println(x:"充电宝已充满电");
    }

    // 新增方法：查看充电宝电量
    public int getBatteryLevel() {
        return batteryLevel;
    }
}
```

```java
package PhoneCaseDecorator;

public class PhoneDemo {
    Run | Debug
    public static void main(String[] args) {
        // 创建一个基础手机
        Phone phone = new BasicPhone(model:"iPhone 14");
        System.out.println(x:"=== 使用基础手机 ===");
        phone.call(number:"10086");
        phone.sendMessage(number:"10086", message:"查询话费");

        // 使用支架手机壳
        System.out.println(x:"\n=== 使用支架手机壳 ===");
        StandCaseDecorator standCase = new StandCaseDecorator(phone);
        standCase.call(number:"10086"); // 原有功能
        standCase.useStand(); // 新增功能
        standCase.adjustStandAngle(angle:45); // 新增功能

        // 使用充电宝手机壳
        System.out.println(x:"\n=== 使用充电宝手机壳 ===");
        PowerBankCaseDecorator powerBankCase = new PowerBankCaseDecorator(phone);
        powerBankCase.sendMessage(number:"10086", message:"查询话费"); // 原有功能
        System.out.println("当前电量：" + powerBankCase.getBatteryLevel() + "%"); // 新增功能
        powerBankCase.chargePhone(); // 新增功能
        powerBankCase.chargePhone(); // 新增功能
        System.out.println("当前电量：" + powerBankCase.getBatteryLevel() + "%");
        powerBankCase.chargePowerBank(); // 新增功能
        System.out.println("充电后电量：" + powerBankCase.getBatteryLevel() + "%");

        // 组合使用多个装饰器
        System.out.println(x:"\n=== 组合使用支架和充电宝手机壳 ===");
        Phone superPhone = new PowerBankCaseDecorator(new StandCaseDecorator(phone));
        superPhone.call(number:"10086"); // 原有功能

        // 要使用特定装饰器的新功能，需要进行类型转换
        if (superPhone instanceof PowerBankCaseDecorator) {
            PowerBankCaseDecorator powerBank = (PowerBankCaseDecorator) superPhone;
            powerBank.chargePhone();
        }
    }
}
```

运行结果:

```
PS E:\大三资料\大三下课程资料\体系结构\PPT\Code\Decorator> java PhoneCaseDecorator.PhoneDemo
=== 使用基础手机 ===
iPhone 14 正在拨打电话: 10086
iPhone 14 正在发送短信到 10086: 查询话费

=== 使用支架手机壳 ===
iPhone 14 正在拨打电话: 10086
使用手机支架，可以解放双手观看视频
调整支架角度为: 45度

=== 使用充电宝手机壳 ===
iPhone 14 正在发送短信到 10086: 查询话费
当前电量: 100%
正在给手机充电，充电宝剩余电量: 100%
正在给手机充电，充电宝剩余电量: 80%
当前电量: 60%
充电宝已充满电
充电后电量: 100%

=== 组合使用支架和充电宝手机壳 ===
iPhone 14 正在拨打电话: 10086
正在给手机充电，充电宝剩余电量: 100%
```