优秀借鉴

1. 装饰模式 – Graphic Design Patterns
2. 设计模式 | 装饰者模式及典型应用
3. 黑马程序员Java设计模式详解-装饰者模式概述

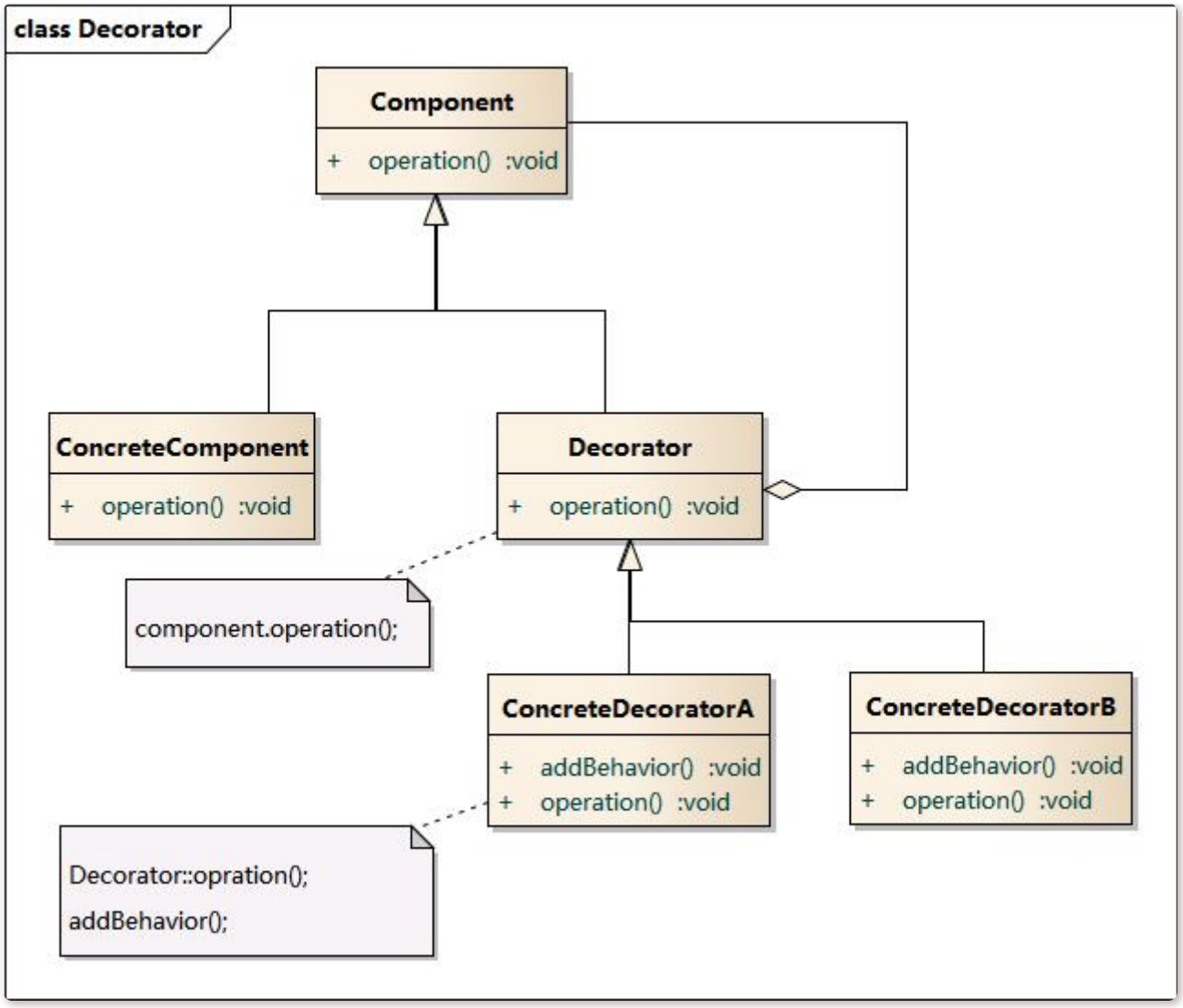
1、概述

装饰者模式(Decorator)是一种结构型设计模式，它允许你在**不改变**对象自身的基础上，**动态**地给一个对象**添加额外的功能。**该模式是通过创建一个包装对象来实现的，也就是用一个新的对象来包装真实的对象。这个装饰对象与原始对象拥有相同的接口，因此客户端无需更改代码即可使用装饰后的对象。

2、结构

在装饰者模式中，一般会涉及到下面四种角色：

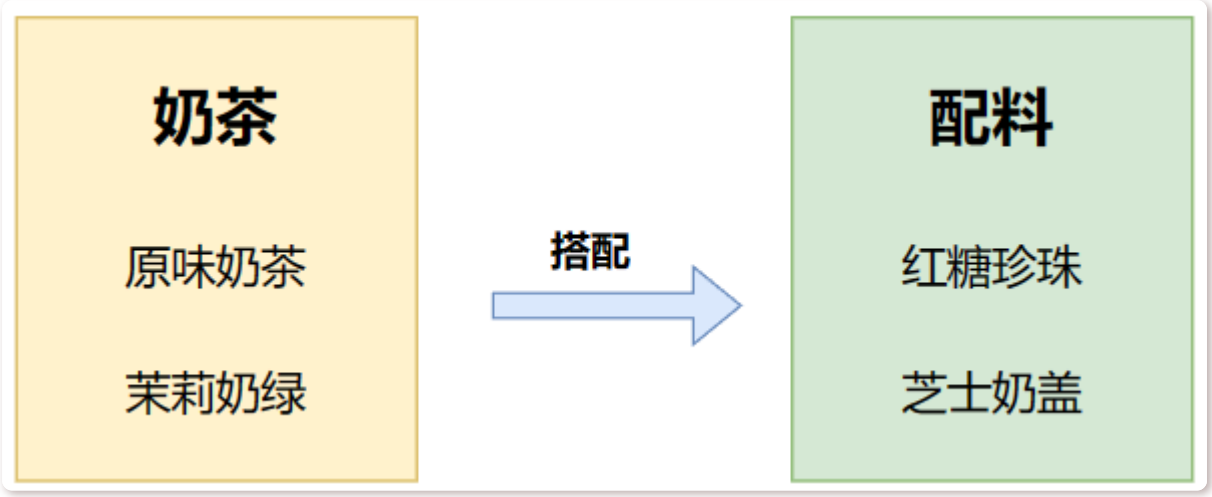
1. **Component（抽象构件）**：它是具体构件和抽象装饰类的共同父类，声明了在具体构件中实现的业务方法，它的引入可以使客户端以一致的方式处理未被装饰的对象以及装饰之后的对象，实现客户端的透明操作；
2. **ConcreteComponent（具体构件）**：它是抽象构件类的子类，用于定义具体的构件对象，实现了在抽象构件中声明的方法，装饰器可以给它增加额外的职责（方法）；
3. **Decorator（抽象装饰类）**：它也是抽象构件类的子类，用于给具体构件增加职责，但是具体职责在其子类中实现。它维护一个指向抽象构件对象的引用，通过该引用可以调用装饰之前构件对象的方法，并通过其子类扩展该方法，以达到装饰的目的；
4. **ConcreteDecorator（具体装饰类）**：它是抽象装饰类的子类，负责向构件添加新的职责。每一个具体装饰类都定义了一些新的行为，它可以调用在抽象装饰类中定义的方法，并可以增加新的方法用以扩充对象的行为。



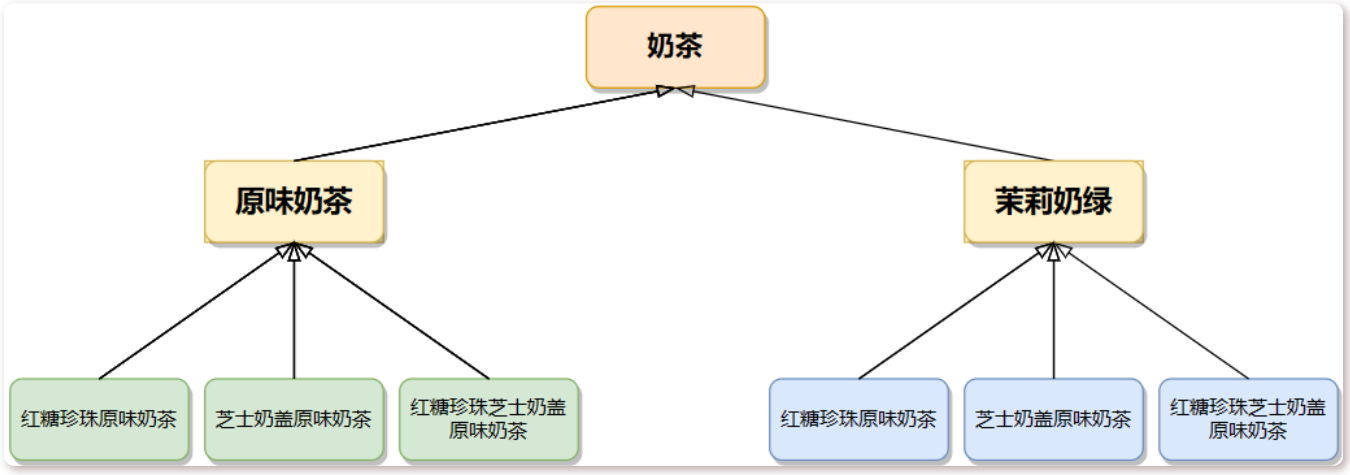
3、实现方式

3.1、案例引入

在生活中或多或少大家应该都点过奶茶，我们就以奶茶为例，假设“007奶茶店”中有原味奶茶和茉莉奶绿两种奶茶，而配料则是有红糖珍珠和芝士奶盖两种，每种奶茶都能添加不同的配料，且价格不同。



现在要求的是计算用户下单不同奶茶的价格，我们很直观的能够想象到把每种情况都列举出来即可，通过继承实现多种不同的搭配：



但是有个问题不知道大家有没有看出来，通过这种继承的方式，很容易产生**类爆炸**，种类少还好，一旦组合多起来那将是不可描述的一场类灾难(反正我画上面图的时候就挺累的)，这时我们就可以使用上这里说到的装饰者模式来进行优化。

3.2、实现步骤

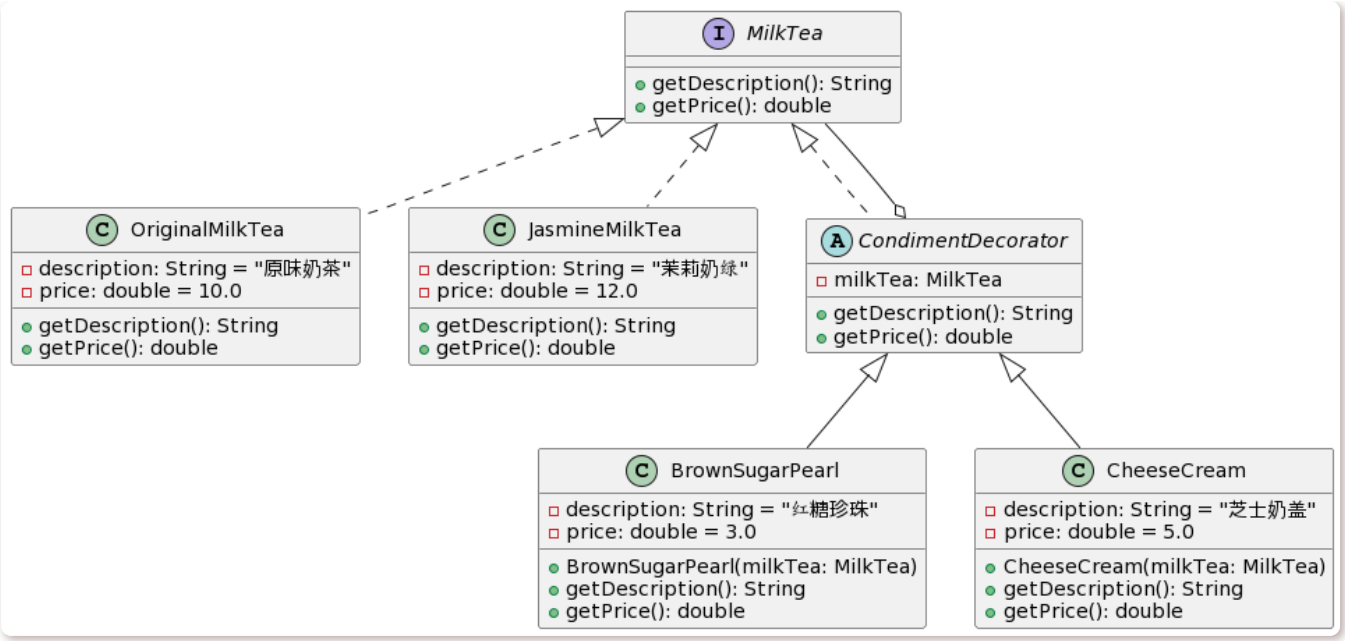
实现装饰者模式的步骤如下：

1. 定义一个**基础接口或抽象类**，作为所有具体组件和装饰者的公共接口；
2. 创建**具体的组件类**，实现基础接口或抽象类，并提供基础功能；
3. 创建一个**抽象的装饰者类**，它包含一个基础接口或抽象类类型的成员变量，并**实现基础接口或抽象类**。这个类通常是一个抽象类，因为它的目的是让子类来扩展装饰行为；
4. 创建**具体的装饰者类**，**继承自抽象的装饰者类**，重写基础方法并在方法执行前后添加自己的逻辑，还可以增加新的方法；
5. 在客户端代码中，使用具体的组件对象来声明一个基础接口或抽象类类型的变量，然后将装饰者对象赋值给该变量。由于装饰者对象也实现了基础接口或抽象类，所以可以通过该变量对被装饰对象进行操作。

3.3、案例实现

我们先来分析一下上面的角色担任：

- **奶茶**：对应装饰者模式中的**抽象构件**，是**具体构件**和**抽象装饰类**的**共同父类**；
- **原味奶茶和茉莉奶绿**：对应装饰者模式中的**具体构件**，装饰器可给它**增加额外的职责**；
- **配料**：对应装饰者模式中的**抽象装饰类**，为**抽象构件**奶茶的子类，用于**给具体构件增加职责**；
- **红糖珍珠和芝士奶盖**：对应装置者模式中的**具体装饰类**，负责给构件添加新的职责。



使用代码通过装饰者模式实现上述场景如下：

首先定义一个奶茶接口(当然，也可以是抽象类)：

```java
/**
 * 奶茶抽象类或接口(抽象构件)
 */
public interface MilkTea {
    String getDescription();
    double getPrice();
}
```

然后实现两种奶茶：

```java
/**
 * 原味奶茶(具体构件)
 */
@Data
public class OriginalMilkTea implements MilkTea {
    private final String description = "原味奶茶";
    private final double price = 10.0;

    @Override
    public String getDescription() {
        return description;
    }

    @Override
    public double getPrice() {
        return price;
    }
```

```
}

/**
 * 茉莉奶绿(具体构件)
 */
@Data
public class JasmineMilkTea implements MilkTea {
    private final String description = "茉莉奶绿";
    private final double price = 12.0;

    @Override
    public String getDescription() {
        return description;
    }

    @Override
    public double getPrice() {
        return price;
    }
}
```

然后定义一个抽象的配料类：

```
/**
 * 配料抽象类(抽象装饰器)
 */
@Data
public abstract class CondimentDecorator implements MilkTea {
    protected MilkTea milkTea;

    public CondimentDecorator(MilkTea milkTea) {
        this.milkTea = milkTea;
    }

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

    @Override
    public double getPrice() {
        return milkTea.getPrice();
    }
}
```

最后实现两个具体的配料类：

```
/**
```

```
 * 红糖珍珠配料(具体装饰器)
 */
@Data
public class BrownSugarPearl extends CondimentDecorator {
    private final String description = "红糖珍珠";
    private final double price = 3.0;

    public BrownSugarPearl(MilkTea milkTea) {
        super(milkTea);
    }

    @Override
    public String getDescription() {
        return milkTea.getDescription() + ", 加" + description;
    }

    @Override
    public double getPrice() {
        return milkTea.getPrice() + price;
    }
}

/**
 * 芝士奶盖配料(具体装饰器)
 */
@Data
public class CheeseCream extends CondimentDecorator {
    private final String description = "芝士奶盖";
    private final double price = 5.0;

    public CheeseCream(MilkTea milkTea) {
        super(milkTea);
    }

    @Override
    public String getDescription() {
        return milkTea.getDescription() + ", 加" + description;
    }

    @Override
    public double getPrice() {
        return milkTea.getPrice() + price;
    }
}
```

这样我们就可以使用装饰者模式来组合奶茶和配料了：

```
// 原味奶茶不加任何配料
MilkTea originalMilkTea = new OriginalMilkTea();
```

```
System.out.println(originalMilkTea.getDescription() + "价格：" +
originalMilkTea.getPrice());

// 茉莉奶绿搭配红糖珍珠
MilkTea jasmineMilkTea = new JasmineMilkTea();
jasmineMilkTea = new BrownSugarPearl(jasmineMilkTea);
System.out.println(jasmineMilkTea.getDescription() + "价格：" +
jasmineMilkTea.getPrice());

// 原味奶茶加芝士奶盖
MilkTea originalMilkTeaWithCheese = new OriginalMilkTea();
originalMilkTeaWithCheese = new
CheeseCream(originalMilkTeaWithCheese);
System.out.println(originalMilkTeaWithCheese.getDescription() + "价
格：" + originalMilkTeaWithCheese.getPrice());

// 茉莉奶绿满配
MilkTea jasmineMilkTea = new JasmineMilkTea();
jasmineMilkTea = new BrownSugarPearl(jasmineMilkTea);
jasmineMilkTea = new CheeseCream(jasmineMilkTea);
System.out.println(jasmineMilkTea.getDescription() + "价格：" +
jasmineMilkTea.getPrice());
```

输出结果：

```
原味奶茶价格：10.0
茉莉奶绿，加红糖珍珠价格：15.0
原味奶茶，加芝士奶盖价格：15.0
茉莉奶绿，加红糖珍珠，加芝士奶盖价格：20.0
```

其中第一杯奶茶没有添加任何配料，第二杯奶茶添加了红糖珍珠配料，第三杯奶茶添加了芝士奶盖配料，第四杯则是满配两个配料都添加了。

4、装饰者模式优缺点

装饰者模式是一种结构型设计模式，其主要**优点**有：

1. **动态扩展功能**：装饰者模式可以在运行时动态地添加、删除和修改对象的功能，从而实现对对象的动态扩展，避免了使用继承带来的静态局限性；
2. **单一职责原则**：装饰者模式将一个大类分为多个小类，每个小类只关注自己的功能实现，符合单一职责原则，使得代码更加清晰简洁；
3. **开放封闭原则**：通过装饰者模式，可以在不改变原有代码的情况下，增强、扩展对象的功能，符合开放封闭原则；
4. **可组合性**：装饰者模式中的装饰者可以任意组合，以增强对象的功能，形成不同的组合结果，具有很好的灵活性和可复用性。

其**缺点**包括：

1. **多层嵌套**：如果使用不当，装饰者模式会导致大量的嵌套和复杂度，使得代码难以维护和理解；
2. **具体组件与装饰者的耦合**：装饰者模式需要每个具体装饰者都依赖于一个具体组件，这种依赖关系可能会导致系统中出现大量的具体类，增加了系统的复杂度。

优点	缺点
动态扩展功能	多层嵌套
单一职责原则	具体组件与装饰者的耦合
开放封闭原则	
可组合性	

5、结构型模式对比

装饰者模式、代理模式和适配器模式都是常用的设计模式，它们之间有些许相似之处，但也存在一些区别。

5.1、装饰者模式和代理模式

装饰者模式和代理模式的联系：

1. 装饰者模式和代理模式都**委托被包装对象进行操作**。在代理模式中，代理对象控制着实际对象的访问，并根据需要对其进行更改或增强。而在装饰者模式中，装饰器对象对被装饰的对象进行了装饰，以增强它的功能；
2. 装饰者模式和代理模式都可以在运行时动态地增强和修改对象的行为。

装饰者模式和代理模式的区别：

1. 装饰者模式**侧重于在不改变已经存在的对象结构**的情况下，动态地将责任附加到对象上，以增强其功能；而代理模式则是**控制对对象的访问**；
2. 装饰者模式所实现的功能一般都是**增强性质**的，而代理模式则是**控制性质**的。

5.2、装饰者模式和适配器模式

适配器模式和装饰者模式的联系和区别：

1. 适配器模式**旨在将一个接口转换成另一个接口**，以便于不兼容的对象之间进行交互。而装饰者模式和代理模式并**不涉及接口转换**；
2. 适配器模式和装饰者模式都是结构型模式。适配器模式主要用于**解决接口不兼容的问题**，而装饰者模式则主要用于**为对象增加新的功能**；
3. 适配器模式和代理模式都能够**控制对对象的访问**，但是它们的目的不同。适配器模式关注**接口的转换**，代理模式关注**控制对对象的访问**。

6、应用场景

装饰者模式主要用于在不改变原有对象的结构和功能的情况下，动态地增加对象的功能。以下是一些使用装饰者模式的常见应用场景：

1. **动态地添加对象的职责**：通过装饰者模式，可以在运行时动态地为一个对象添加新的职责，而不需要修改它的代码或继承它；
2. **多个小对象进行组合**：使用装饰者模式可以将多个小对象组合成一个大对象，并且可以根据需要随意组合这些小对象，以形成不同的组合结果；
3. **需要扩展现有类的功能而又不能修改其源代码**：在一些开源库或第三方库中，由于源代码无法修改，但是又需要对其功能进行扩展，此时装饰者模式可以非常方便地实现这一需求；
4. **给已有的对象添加新的行为，而且这些行为还能够互相组合**：使用装饰者模式，可以很容易地给一个已有的对象添加新的行为，并且这些行为还能够互相组合，以形成更复杂的行为；
5. **避免继承带来的子类爆炸问题**：通过装饰者模式，可以避免使用继承带来的子类爆炸问题，从而使得系统更加灵活、可扩展。