链滴

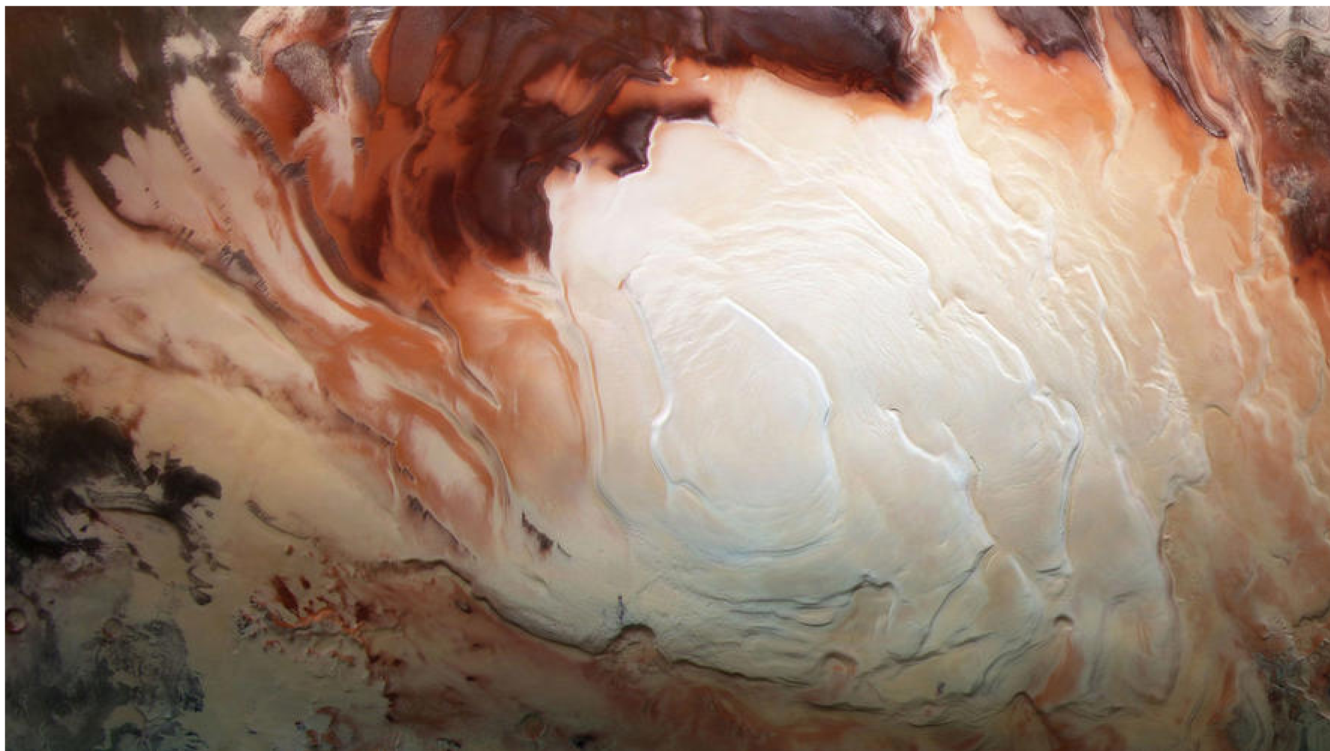
设计模式 |03 装饰者模式

作者：qq692310342

原文链接：https://ld246.com/article/1567487529738

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)



开头说几句

博主的博客地址：https://www.jeffcc.top/

推荐入门学习设计模式java版本的数据Head First 《设计模式》

什么是装饰者模式

专业定义：装饰者模式可以动态得讲责任附加到对象上，若要拓展功能，装饰者模式提供了比继承更有弹性的替代方案。

个人见解：装饰者模式重点在于装饰者，我们可以不修改对象的前提下，对对象进行功能的拓展，不过继承来实现行为的拓展，而是通过组合和委托来实现！以免出现继承泛滥以及一些其他的业务修改象代码而带来的bug。

设计原则

1. 类应该对拓展开放，对修改关闭。我们的目标是允许类容易拓展，再不修改代码的情况下，就可以配新的行为。

2. 并不是要求对类的所有部分都设计成开放-关闭原则，应该把注意力放在设计中最有可能改变的地，然后应用开放-关闭的原则，过多的设计会造成浪费，同时也会造成代码变得负责而难以理解，并太多益处。

设计方法

1. 装饰者和被装饰者具有相同的超类型；这里用到了继承的方式来实现同类型，但是并没有使用到继来拓展行为，所以这不违背多使用组合而少使用继承的原则；

2. 可以使用一个或者多个装饰者来装饰一个对象；

3. 既然装饰者和被装饰者都有相同的超类型，所以可以在任何需要原始数据类型的地方使用装饰对象

替；
4. 装饰者可以在所委托被装饰的行为之前或者之后，加上自己的行为，以达到某种特定的目的；
5. 对象可以在任何时候被装饰；

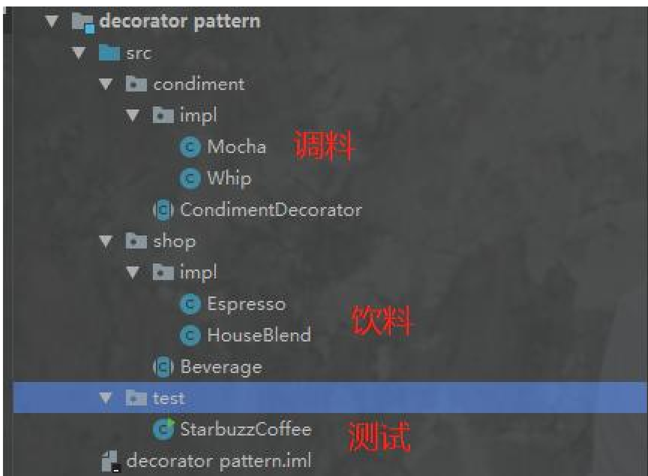
模式实例

实例背景

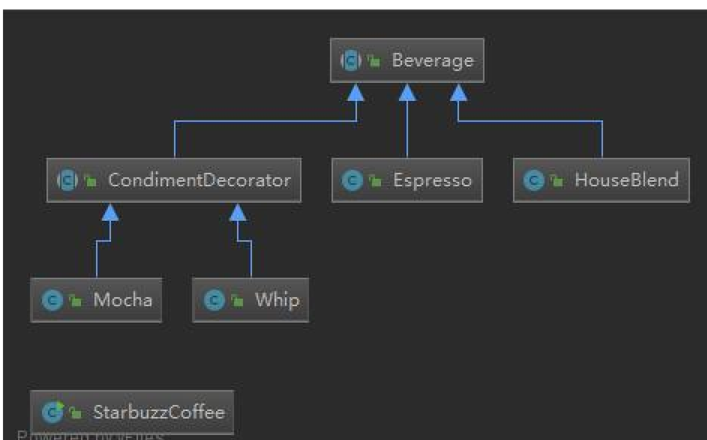
一家咖啡店需要设计一个订单系统，其中的订单价格和订单描述这一方面需要设计出一种优秀的模式，每款饮料都继承自Beverage，饮料配有配料以及本身的价格以及杯的大小的价格不同而有不同的定
。

代码实现

项目结构



项目类图



饮料抽象类

```
package shop;
```

```
/**
 * 咖啡店的饮料抽象类  所有饮料都要继承自它
 * 提供两个方法
 */
public abstract class Beverage {
    public String description = "Unkonw Beverage";

    public String getDescription() {
        return description;
    }

    /**
     * 计算价格的抽象方法
     * @return
     */
    public abstract double cost();
}
```

调料抽象类

```
package condiment;

import shop.Beverage;

/**
 *调料抽象类 所有调料的装饰者都要继承
 * 注意这里一定要继承Beverage！！！  否则无法进行嵌套的装饰
 */
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

饮料1 Espresso

```
package shop.impl;

import shop.Beverage;

/**
 * 饮料1 Espresso
 */
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    /**
     * 装饰者装饰价格1 饮料本身的价格
     * @return
```

```
     */
    @Override
    public double cost() {
        return 1.99;
    }
}
```

饮料2 HouseBlend

```
package shop.impl;

import shop.Beverage;

/**
 * 饮料2 HouseBlend
 */
public class HouseBlend extends Beverage {

    public HouseBlend() {
        description = "HouseBlend";
    }

    /**
     * 装饰者装饰价格2 HouseBlend饮料本身的价格
     * @return
     */
    @Override
    public double cost() {
        return 0.80;
    }
}
```

调料1 Mocha

```
package condiment.impl;

import condiment.CondimentDecorator;
import shop.Beverage;

/**
 * 调料1 Mocha
 */
public class Mocha extends CondimentDecorator {

//    定义要搭配的饮料类型
    public Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
```

```
    /**
     * 增加饮料的描述  加上配料
     * @return
     */
    @Override
    public String getDescription() {
        return beverage.getDescription()+"Mocha";
    }

    /**
     *装饰者装饰调料价格 在基础上加0.2
     * @return
     */
    public double cost(){
        return beverage.cost() + 0.20;
    }
}
```

调料2 Whip

```
package condiment.impl;

import condiment.CondimentDecorator;
import shop.Beverage;

/**
 * 调料2 Whip
 */
public class Whip extends CondimentDecorator {

//    定义要搭配的饮料类型
    public Beverage beverage;

    public Whip(Beverage beverage) {
        this.beverage = beverage;
    }

    /**
     * 增加饮料的描述  加上配料
     * @return
     */
    @Override
    public String getDescription() {
        return beverage.getDescription()+"Whip";
    }

    /**
     *装饰者装饰调料价格 在基础上加0.2
     * @return
     */
    public double cost(){
        return beverage.cost() + 0.50;
    }
```

```
}
```

咖啡店测试

```
package test;

import condiment.impl.Mocha;
import condiment.impl.Whip;
import shop.Beverage;
import shop.impl.Espresso;
import shop.impl.HouseBlend;

/**
 * 咖啡店的测试开业
 */
public class StarbuzzCoffee {
    public static void main(String[] args) {
//        定一杯 饮料 Espresso  不加任何调料
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()+" $"+beverage.cost());

//        再定一杯 Espresso
        Beverage beverage2 = new Espresso();
//        开始装饰 加上一个mocha配料 反复定义
        beverage2 = new Mocha(beverage2);
//        加上一个whip配料
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()+" $"+beverage2.cost());

//        再定一杯 HouseBlend
        Beverage beverage3 = new HouseBlend();
//        开始装饰 加上一个mocha配料 反复定义
        beverage3 = new Mocha(beverage3);
//        加上一个whip配料
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()+" $"+beverage3.cost());


    }
}
```

输出结果

```
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
Espresso $1.99  不加调料 原价1.99
EspressoMochaWhip $2.69  加了两个配料 1.99+0.5+0.2
HouseBlendMochaWhip $1.5  加两个配料 0.8+0.5+0.2

Process finished with exit code 0
```

原文链接：设计模式 |03 装饰者模式

分析

关键在于：装饰者CondimentDecorator一定要注意继承被装饰者的抽象类Beverage，这样才能够现不断得进行装饰。

现实中的装饰者 java I/O

Java世界中有太多的装饰者模式的设计了，java.io包中就有许多这样的装饰者；

FileInputStream就是一个被装饰的组件，提供最基本的io功能；

而BufferedInputStream是一个具体的装饰者，它加入两种行为：利用缓冲输入来改善性能，用一个radLine方法来增强了接口；

LineNumberInputStream也是一个具体的装饰者，它加上了计算行数的功能。

挖掘源码我们也可以发现： 这些io的装饰者都继承自同一个超类，这样使得io的装饰起来便捷了很多

装饰者模式的一个小缺点

利用装饰者模式造成的设计中有大量的小类，数量十分多，可能会造成使用此API的程序员的困扰。是我们理解了装饰者模式的工作原理了，就能够在以后的工作中容易的辨识出类是如何组织的，也就高效的进行开发了！

END

2019年9月3日13:11:51