连接两地的交通枢纽——桥接模式

1.桥接模式介绍

桥接模式(Bridge Pattern)也称为桥梁模式，是结构型设计模式之一。在现实生活中大家都知道“桥梁”是连接河道两岸的主要交通枢纽，简而言之其作用就是连接河的两边，而我们的桥接模式与现实中的情况很相似，也是承担着连接“两边”的作用。

2.桥接模式的定义

将抽象部分与现实部分分离，使它们都可以独立地进行变化。

3.桥接模式的使用场景

从模式的定义中我们大致可以了解到，这里“桥梁”的作用其实就是连接“抽象部分”与“实现部分”，但是事实上，任何多维度变化类或者说多个树状类之间的耦合都可以使用桥接模式来实现解耦。

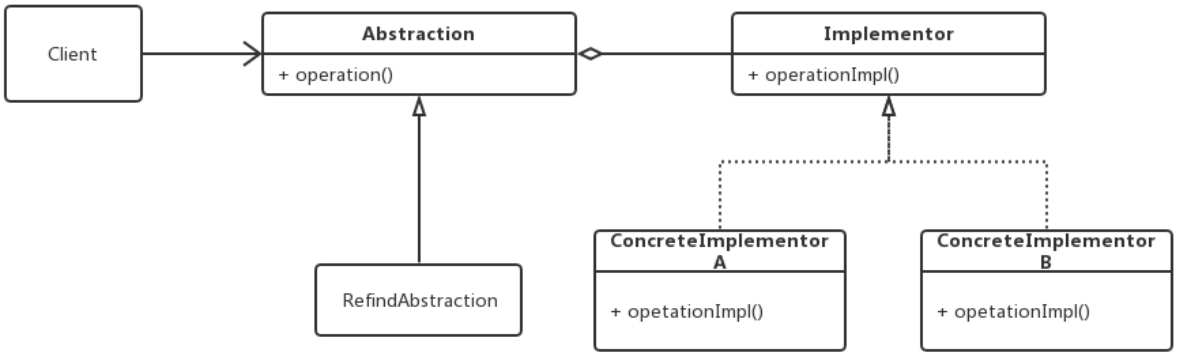
如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，可以通过桥接模式使它们在抽象层建立一个关联关系。

对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，也可以考虑使用桥接模式。

一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

4. 桥接模式的UML类图

UML类图如图：



实现部分的抽象接口：

```
public interface Implementor {
        /*
         * 实现抽象部分的具体方法
         */
        public void operatopImpl();
}

```

实现部分具体的实现一\二

```
public class ConcreteImplementorA implements Implementor {

        public void operatopImpl() {
                // 具体的实现类
        }

}

public class ConcreteImplementorB implements Implementor {

        public void operatopImpl() {
                // 具体的实现类
        }

}
```

抽象部分的实现

```
public abstract class Abstraction {

        private Implementor mImplementor;// 声明一个私有成员变量引用实现部分的对象

```

```
        /*
         * 通过实现不分对象的引用构造部分的对象
         *
         * @param implementor 实现部分对象的引用
         */
        public Abstraction(Implementor implementor) {
                this.mImplementor = implementor;
        }

        /*
         * 通过调用实现部分具体的方法实现具体的功能
         */
        public void operation() {
                mImplementor.operatopImpl();
        }
}
```

抽象部分的子类：

```
public class RefinedAbstraction extends Abstraction {

        public RefinedAbstraction(Implementor implementor) {
                super(implementor);
        }
        /*
         * 对父类抽象部分中的方法进行扩展
         */
        public void refinedOperation(){
                //对Abstraction中的方法进行扩展
        }

}
```

客户类：

```
public class Main {

        public static void main(String[] args) {
        //客户调用逻辑
    }
```

角色介绍：

- Abstraction：抽象部分。
  该类保持一个对实现部分对象的引用，抽象部分中的方法需要调用实现部分的对象来实现，该类一般为抽象类。
- RefinedAbstraction：优化的抽象部分。
  抽象部分的具体实现，该类一般是对抽象部分的方法进行完善和扩展。
- Implementor：实现部分。
  可以为接口或抽象类，其方法不一定要与抽象部分中的一致，一般情况下是由实现部分提供基本的操作，而抽象部分定义的则是基于实现部分这些基本操作的业务方法。

- ConcreteImplementorA/ConcreteImplementorB：实现部分的具体实现。
  完善实现部分中方法定义的具体逻辑。
- Client：客户类，客户端程序。

5.桥接模式的简单实现

现实生活中有很多桥接模式应用的影子，比如开关与具体的电器，开关的类型有多种，而电器也是各式各样，这两者是独立变化的且又有耦合。还有程序员天天面对的显示屏，对于显示屏来说它的尺寸与生产厂商之间也是一种二维关系，具体的尺寸与具体的厂商独立变化，而更贴近生活的例子就是我们在和咖啡时，咖啡一般分为4种。大杯加糖、大杯不加糖、大杯不加糖、小杯加糖、小杯不加糖，对一杯咖啡来说这说4中实质就两种变化，一是大杯小杯，二是加糖不加糖，不管怎样，我们都先来定义一个咖啡类。

```
public abstract class Coffee {

        protected CoffeeAdditives impl;

        public Coffee(CoffeeAdditives impl) {
                this.impl = impl;
        }

        /*
         * 咖啡具体是什么样的由子类决定
         */
        public abstract void makeCoffee();
}

```

Coffee类中保持了对CoffeeAddivities的引用，一便调用具体的实现。同样地，咖啡还分大杯小杯，定义两个子类继承与Coffee。

```
public class LargeCoffee extends Coffee{

        public LargeCoffee(CoffeeAdditives impl) {
                super(impl);
        }
        public void makeCoffee() {
                System.out.println("大杯的"+impl.addSomeString()+"咖啡");
        }


}

public class SmallCoffee extends Coffee{

        public SmallCoffee(CoffeeAdditives impl) {
                super(impl);
        }
        public void makeCoffee() {
                System.out.println("小杯的"+impl.addSomeString()+"咖啡");
        }

```

```
}
```

而对于加进咖啡中的糖，当然也可以选择不加，我们也用一个抽象类定义：

```
public abstract class CoffeeAdditives {
        /*
         * 具体要往咖啡里添加什么也是由子类实现
         *
         * @return 具体添加的东西
         */
        public abstract String addSomeString();
}
```

注意i，这里的CoffeeAdditives其实就是对英语上面我们UML类图中的实现部分，而Coffee则对应于抽象部分，模式定义中所谓的“抽象”与"实现"实质上对应的是两个独立变化的维度，因此，上文中我们也曾说过，任何多维度变化类或者说多个树状类之间耦合都可以使用桥接模式来实现解耦。在本例中，我们的Coffee类虽是一个抽象类，但是并非所谓的“抽象部分”，而CoffeeAddivitives类也并非一定就是“实现部分”,两者各自为一维度，独立变化仅此而已，所谓的”抽象与实现分离“更偏向于我们实际的程序开发，两者并不一定挂钩，这里其实就可以看到桥接模式的应用型其实很广泛，并不局限与程序设计。

CoffeeAdditives对应的两个子类：加糖与不加糖

```
public class Sugar extends CoffeeAdditives{

        public String addSomeString() {
                return "加糖";
        }

}

public class Ordinary extends CoffeeAdditives{

        public String addSomeString() {
                return "不加糖";
        }

}
```

不加糖我们以原味表示，最后来看客户端，将两者进行整合。

```
public class Main {

        public static void main(String[] args) {
                //原汁原味
                Ordinary implOrdinary =new Ordinary();

                //准备加糖
                Sugar implSugar=new Sugar();

                //大杯咖啡 原味
                LargeCoffee largeCoffeeOrdinary=new LargeCoffee(implOrdinary);

```

```
                largeCoffeeOrdinary.makeCoffee();

                //小杯咖啡 原味
                SmallCoffee smallCoffeeOrdinary=new SmallCoffee(implOrdinary);
                smallCoffeeOrdinary.makeCoffee();

                //大杯咖啡 加糖
                LargeCoffee largeCoffeeSugar=new LargeCoffee(implSugar);
                largeCoffeeSugar.makeCoffee();

                //小杯咖啡 加糖
                SmallCoffee smallCoffeeSugar=new SmallCoffee(implSugar);
                smallCoffeeSugar.makeCoffee();

        }

}
```

输出：

```
大杯的不加糖咖啡
小杯的不加糖咖啡
大杯的加糖咖啡
小杯的加糖咖啡

```

从本例我们可以看到，不管是Coffee变化了还是CoffeeAdditives变化了，其相对于对方而言都是独立的没有什么过多的交集，两者之间唯一的联系就是Coffee中保持的对CoffeeAdditives的引用，此乃两者之纽带，这就是桥接模式。

6.总结

桥接模式可以应用到许多开发中，但是它应用的却不多，一个很重要的原因是对于抽象与实现的分离的被我，是不是需要分离、如何分离？对设计者来说要有一个恰到好处的分寸。不管怎么说，桥接模式的有点我们无容置疑，分离抽象与实现、灵活的扩展以及对呵护来说透明的实现等。但是使用桥接模式也有一个不明显的缺点，就是不容易设计，对开发者来说要有一定的经验要求。因此，对桥接模式应用来说，理解很简单，设计却不容易。

xyzso1z
原创文章 80　获赞 40　访问量 2万+