策略模式

1.策略模式（Strategy patten）定义了算法族（defines a family of algorithms），分别封装起来（encapsulates each algorithm），让它们之间可以互相替换（makes the algorithms interchangeable within that family），此模式让算法的变化独立于使用算法的客户（Strategy lets the algorithm vary independently from clients that use it.）。

2.举一个例子，现在有一个 Car 类，有两个功能，刹车 brake和加速 accelerate，我们就刹车来说，至少存在普通刹车和使用ABS防抱死刹车，不同的车型有不同的刹车方式，有的是普通刹车有的是防抱死刹车方式，比如常见的小轿车车型就是普通刹车，而SUV车型就是ABS防抱死刹车系统。

3.对于以上的例子，我们最普通的方式，就是先写一个超类Car类，然后定义抽象方法brake（），然后让子类自己去实现brake（）方法，通过多态来实现不同的刹车行为。但是这种方式有以下几个问题：

- 需要在每一个子类中写具体的实现行为，如果有许多个子类车型，则工作量会很大，而其中只应用到了普通刹车和ABS防抱死刹车方式，无疑有很多的子类的刹车方式是相同的，但是这些刹车行为的具体实现方式却要写到每个子类中，所以造成了代码的大量重复。

```
public class 宝马小轿车 extends Car {

    public void applyBrake() {
        System.out.println("我是小轿车，所以使用了普通的刹车方式！");
    }
}



public class 丰田小轿车 extends Car {

    public void applyBrake() {
        System.out.println("我是小轿车，所以使用了普通的刹车方式！");
    }
}
```

如上图，红色圈出部分的代码就是重复的，如果有一百个小轿车，我们则需要重复一百次这样的代码。

- 除非你追踪到每一个类的刹车具体实现里面去看代码，要不你很难确定哪个车使用了什么刹车行为。
- 如果某一个车型的程序，想要在运行时改变自己的刹车行为，从普通刹车改变到ABS防抱死刹车，是很不容易的。

4.针对这些缺点，我们使用了策略模式来解决这个问题，代码如下：

```
/* Encapsulated family of Algorithms
 * Interface and its implementations
 */
public interface IBrakeBehavior {
 public void brake();
}
public class BrakeWithABS implements IBrakeBehavior {
 public void brake() {
```

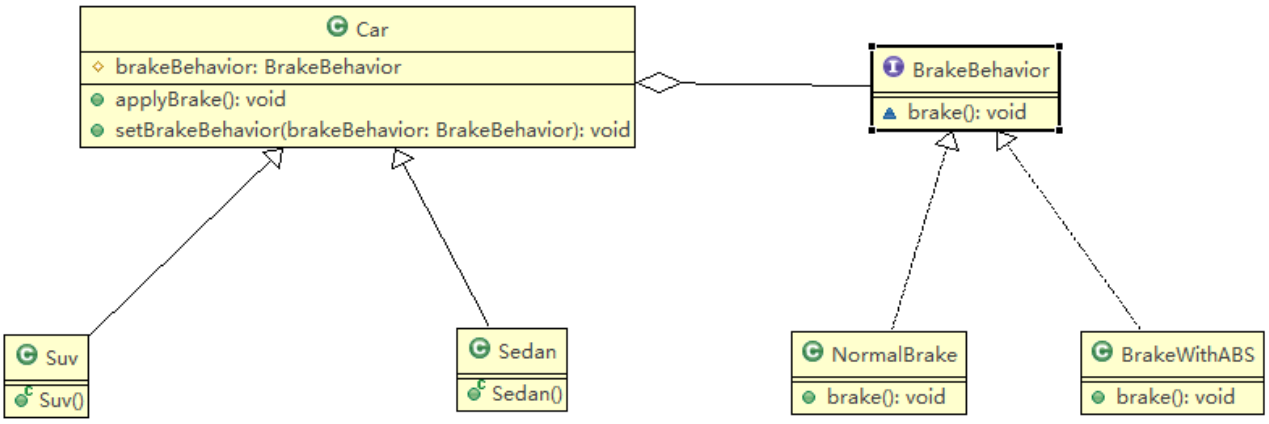
```
 System.out.println("Brake with ABS applied");
 }
}
public class Brake implements IBrakeBehavior {
 public void brake() {
 System.out.println("Simple Brake applied");
 }
}
/* Client that can use the algorithms above interchangeably */
public abstract class Car {
 protected IBrakeBehavior brakeBehavior;
public void applyBrake() {
 brakeBehavior.brake();
 }
public void setBrakeBehavior(final IBrakeBehavior brakeType) {
 this.brakeBehavior = brakeType;
 }
}
/* Client 1 uses one algorithm (Brake) in the constructor */
public class Sedan extends Car {
 public Sedan() {
 this.brakeBehavior = new Brake();
 }
}
/* Client 2 uses another algorithm (BrakeWithABS) in the constructor */
public class SUV extends Car {
 public SUV() {
 this.brakeBehavior = new BrakeWithABS();
 }
}
/* Using the Car example */
public class CarExample {
 public static void main(final String[] arguments) {
 Car sedanCar = new Sedan();
 sedanCar.applyBrake(); // This will invoke class "Brake"
 Car suvCar = new SUV();
 suvCar.applyBrake(); // This will invoke class "BrakeWithABS"
// set brake behavior dynamically
 suvCar.setBrakeBehavior( new Brake() );
 suvCar.applyBrake(); // This will invoke class "Brake"
 }
}
```

将刹车行为封装成一个接口，从Car类之中解耦出来，再定义具体的实现类来实现刹车行为，比如普通刹车和ABS防抱死刹车。再在Car类中定义一个刹车行为的引用IBrakeBehavior，再给定setBrakeBehavior()方法来实现刹车行为的切换。然后在定义车型子类的时候，在构造器之中初始化刹车行为，在必要的时候，该车型可以通过setBrakeBehavior()方法来切换自己的刹车行为，比如从普通刹车切换到ABS防抱死刹车，也就是说，可以在运行时（run-time）而不仅只是在编译时来定义自己的刹车行为。

将刹车行为从Car类之中分离开后，通过组合而不是继承的方式应用到每一个子类车型之中，免除了在每一个子类车型之中写具体的刹车实现，大大地减少了工作量以及提高了代码的复用，以后如果有其他的刹车方式，也可以通过实现

IBrakeBehavior接口加入到程序之中来，体现了可扩展性。通过查看构造器和set方法也能很快知道当前车型使用了什么刹车方式。

5.UML图



其中，Car为用户，BrakeBehavior为算法族。