当我们掌握了Java的语法，当我们了解了面向对象的封装、继承、多态等特性，当我们可以用Swing、Servlet、JSP技术构建桌面以及Web应用，不意味着我们可以写出面向对象的程序，不意味着我们可以很好的实现代码复用，弹性维护，不意味着我们可以实现在维护、扩展基础上的代码复用。一把刀，可以使你制敌于无形而于江湖扬名，也可以只是一把利刃而使你切菜平静。Java，就是这把刀，它的威力取决于你使用的方式。当我们陷入无尽无止重复代码的泥沼，当我们面临牵一发而动全身的维护恶梦，你应该想起"设计模式"这个行动秘笈。面向对象的精义，看似平淡，其实要经过艰苦实践才能成功。而构造OO系统的隐含经验于是被前人搜集而成并冠以"设计模式"之名。我们应该在编码行动初始就携带以它。接下来，让我们步"四人组"先行者之后，用中国文字、用实际案例领略模式于我们代码焕然一新的改变：

设计模式解读之一：策略模式

1. 模式定义

把会变化的内容取出并封装起来，以便以后可以轻易地改动或扩充部分，而不影响不需要变化的其他部分；

2. 问题缘起

当涉及至代码维护时，为了复用目的而使用继承，结局并不完美。对父类的修改，会影响到子类型。在超类中增加的方法，会导致子类型有该方法，甚至连那些不该具备该方法的子类型也无法免除。示例，一个鸭子类型：

```
public abstract class Duck {
    //所有的鸭子均会叫以及游泳，所以父类中处理这部分代码
    public void quack() {
        System.out.println("Quack");
    }

    public void swim() {
        System.out.println("All ducks float, even decoys.");
    }

    //因为每种鸭子的外观是不同的，所以父类中该方法是抽象的，由子类型自己完成。
    public abstract void display();
}

public class MallardDuck extends Duck {
    //野鸭外观显示为绿头
    public void display() {
        System.out.println("Green head.");
    }
}

public class RedHeadDuck extends Duck {
```

```
    //红头鸭显示为红头
    public void display() {
        System.out.println("Red head.");
    }
}

public class RubberDuck extends Duck {
    //橡皮鸭叫声为吱吱叫，所以重写父类以改写行为
    public void quack() {
        System.out.println("Squeak");
    }

    //橡皮鸭显示为黄头
    public void display() {
        System.out.println("Yellow head.");
    }
}
```

上述代码，初始实现得非常好。现在我们如果给 Duck.java 中加入 fly()方法的话，那么在子类型中均有了该方法，于是我们看到了会飞的橡皮鸭子，你看过吗？当然，我们可以在子类中通过空实现重写该方法以解决该方法对于子类型的影响。但是父类中再增加其它的方法呢？

通过继承在父类中提供行为，会导致以下缺点：

a. 代码在多个子类中重复;
b. 运行时的行为不容易改变；
c. 改变会牵一发动全身，造成部分子类型不想要的改变；

好啦，还是刚才鸭子的例子，你也许想到使用接口，将飞的行为、叫的行为定义为接口，然后让 Duck 的各种子类型实现这些接口。这时侯代码类似于：

```
public abstract class Duck {
    //将变化的行为 fly() 以及 quake()从 Duck 类中分离出去定义形成接口，有需求的子类中自行去实现

    public void swim() {
        System.out.println("All ducks float, even decoys.");
    }

    public abstract void display();
}

//变化的 fly() 行为定义形成的接口
public interface FlyBehavior {
```

```
    void fly();
}

//变化的 quack() 行为定义形成的接口
public interface QuackBehavior {
    void quack();
}

//野鸭子会飞以及叫，所以实现接口  FlyBehavior, QuackBehavior
public class MallardDuck extends Duck implements FlyBehavior, QuackBehavior{
    public void display() {
        System.out.println("Green head.");
    }

    public void fly() {
        System.out.println("Fly.");
    }

    public void quack() {
        System.out.println("Quack.");
    }
}

//红头鸭子会飞以及叫，所以也实现接口  FlyBehavior, QuackBehavior
public class RedHeadDuck extends Duck implements FlyBehavior, QuackBehavior{
    public void display() {
        System.out.println("Red head.");
    }

    public void fly() {
        System.out.println("Fly.");
    }

    public void quack() {
        System.out.println("Quack.");
    }
}

//橡皮鸭不会飞，但会吱吱叫，所以只实现接口 QuackBehavior
public class RubberDuck extends Duck implements QuackBehavior{
    //橡皮鸭叫声为吱吱叫
    public void quack() {
        System.out.println("Squeak");
    }
```

```
    //橡皮鸭显示为黄头
    public void display() {
        System.out.println("Yellow head.");
    }
}
```

上述代码虽然解决了一部分问题，让子类型可以有选择地提供一些行为(例如 fly() 方法将不会出现在橡皮鸭中).但我们也看到，野鸭子 MallardDuck.java 和红头鸭子 RedHeadDuck.java 的一些相同行为代码不能得到重复使用。很大程度上这是从一个火坑跳到另一个火坑。

在一段程序之后，让我们从细节中跳出来，关注一些共性问题。不管使用什么语言，构建什么应用，在软件开发上，一直伴随着的不变的真理是：需要一直在变化。不管当初软件设计得多好，一段时间之后，总是需要成长与改变，否则软件就会死亡。

我们知道，继承在某种程度上可以实现代码重用，但是父类(例如鸭子类 Duck)的行为在子类型中是不断变化的，让所有子类型都有这些行为是不恰当的。我们可以将这些行为定义为接口，让 Duck 的各种子类型去实现，但接口不具有实现代码，所以实现接口无法达到代码复用。这意味着，当我们需要修改某个行为，必须往下追踪并在每一个定义此行为的类中修改它，一不小心，会造成新的错误。

设计原则：把应用中变化的地方独立出来，不要和那些不需要变化的代码混在一起。这样代码变化引起的不经意后果变少，系统变得更有弹性。

按照上述设计原则，我们重新审视之前的 Duck 代码。

1) 分开变化的内容和不变的内容

Duck 类中的行为 fly(), quack(), 每个子类型可能有自己特有的表现，这就是所谓的变化的内容。
Duck 类中的行为 swim() 每个子类型的表现均相同，这就是所谓不变的内容。

我们将变化的内容从 Duck()类中剥离出来单独定义形成接口以及一系列的实现类型。将变化的内容定义形成接口可实现变化内容和不变内容的剥离。其实现类型可实现变化内容的重用。这些实现类并非 Duck.java 的子类型，而是专门的一组实现类，称之为"行为类"。由行为类而不是 Duck.java 的子类型来实现接口。这样，才能保证变化的行为独立于不变的内容。于是我们有：

变化的内容：

```
//变化的 fly() 行为定义形成的接口
public interface FlyBehavior {
 void fly();
}

//变化的 fly() 行为的实现类之一
public class FlyWithWings implements FlyBehavior {
```

```
 public void fly() {
     System.out.println("I'm flying.");
 }
}

//变化的 fly() 行为的实现类之二
public class FlyNoWay implements FlyBehavior {
 public void fly() {
     System.out.println("I can't fly.");
 }
}

   ----------------------------------------------------------------

//变化的 quack() 行为定义形成的接口
public interface QuackBehavior {
 void quack();
}

//变化的 quack() 行为实现类之一
public class Quack implements QuackBehavior {
 public void quack() {
     System.out.println("Quack");
 }
}

//变化的 quack() 行为实现类之二
public class Squeak implements QuackBehavior {
 public void quack() {
     System.out.println("Squeak.");
 }
}

//变化的 quack() 行为实现类之三
public class MuteQuack implements QuackBehavior {
 public void quack() {
     System.out.println("<<Slience>>");
 }
}
```

通过以上设计，fly()行为以及 quack()行为已经和 Duck.java 没有什么关系，可以充分得到复用。而且我们很容易增加新的行为，既不影响现有的行为，也不影响 Duck.java。但是，大家可能有个疑问，就是在面向对象中行为不是体现为方法吗？为什么现在被定义形成类(例如 Squeak.java)？在 OO 中，类代表的"东西"一般是既有状态(实例变量）又有方法。只是在本例中碰巧"东西"是个行为。既使是行为，也有属性

及方法，例如飞行行为，也需要一些属性记录飞行的状态，如飞行高度、速度等。

2) 整合变化的内容和不变的内容

Duck.java 将 fly()以及 quack()的行为委托给行为类处理。

不变的内容：

```
public abstract class Duck {
    //将行为类声明为接口类型，降低对行为实现类型的依赖
 FlyBehavior flyBehavior;
 QuackBehavior quackBehavior;

 public void performFly() {
    //不自行处理 fly()行为，而是委拖给引用 flyBehavior 所指向的行为对象
    flyBehavior.fly();
 }

 public void performQuack() {
    quackBehavior.quack();
 }

 public void swim() {
    System.out.println("All ducks float, even decoys.");
 }

 public abstract void display();
}
```

Duck.java 不关心如何进行 fly()以及 quack(), 这些细节交由具体的行为类完成。

```
public class MallardDuck extends Duck{
 public MallardDuck() {
    flyBehavior=new FlyWithWings();
    quackBehavior=new Quack();
 }

 public void display() {
    System.out.println("Green head.");
 }
}
```

测试类：

```
public class DuckTest {
 public static void main(String[] args) {
     Duck duck=new MallardDuck();
     duck.performFly();
     duck.performQuack();
 }
}
```

在 Duck.java 子类型 MallardDuck.java 的构造方法中，直接实例化行为类型，在编译的时侯便指定具体行为类型。当然，我们可以：

1) 我们可以通过工厂模式或其它模式进一步解藕(可参考后续模式讲解);
2) 或做到在运行时动态地改变行为。

3) 动态设定行为

在父类 Duck.java 中增加设定行为类型的 setter 方法，接受行为类型对象的参数传入。为了降藕，行为参数被声明为接口类型。这样，既便在运行时，也可以通过调用这二个方法以改变行为。

```
public abstract class Duck {
 //在刚才 Duck.java 中加入以下二个方法。
 public void setFlyBehavior(FlyBehavior flyBehavior) {
     this.flyBehavior=flyBehavior;
 }

 public void setQuackBehavior(QuackBehavior quackBehavior) {
     this.quackBehavior=quackBehavior;
 }

 //其它方法同，省略...
}
```

测试类：

```
public class DuckTest {
 public static void main(String[] args) {
     Duck duck=new MallardDuck();
     duck.performFly();
     duck.performQuack();
     duck.setFlyBehavior(new FlyNoWay());
     duck.performFly();
 }
}
```

如果，我们要加上火箭助力的飞行行为，只需再新建 FlyBehavior.java 接口的实现类型。而子类型可通过调用 setQuackBehavior(...)方法动态改变。至此，在 Duck.java 增加新的行为给我们代码所带来的困绕已不复存在。

该是总结的时侯了，让我们从代码的水中浮出来，做一只在水面上自由游动的鸭子吧:

3. 解决方案

MallardDuck 继承 Duck 抽象类； ->不变的内容
FlyWithWings 实现 FlyBehavior 接口； ->变化的内容,行为或算法
在 Duck.java 提供 setter 方法以装配关系； ->动态设定行为

以上就是策略模式的实现三步曲。接下来，让我们透过步骤看本质:

1) 初始，我们通过继承实现行为的重用，导致了代码的维护问题。 ->继承, is a
2) 接着，我们将行为剥离成单独的类型并声明为不变内容的实例变量并通过 setter 方法以装配关系； ->组合, has a

继承，可以实现静态代码的复用；组合，可以实现代码的弹性维护；使用组合代替继承，可以使代码更好地适应软件开发完后的需求变化。

策略模式的本质：少用继承，多用组合