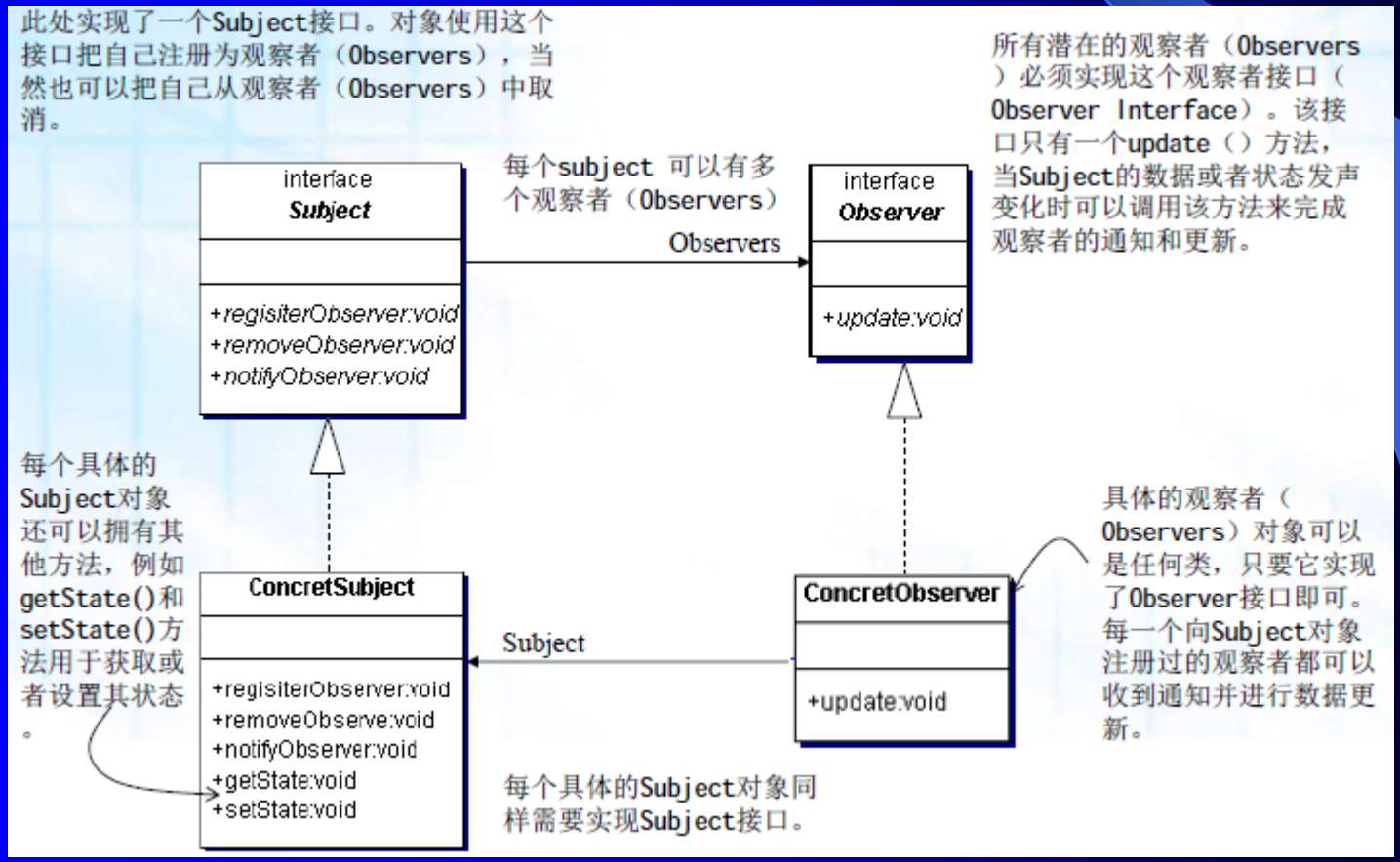
策略模式

内容回顾

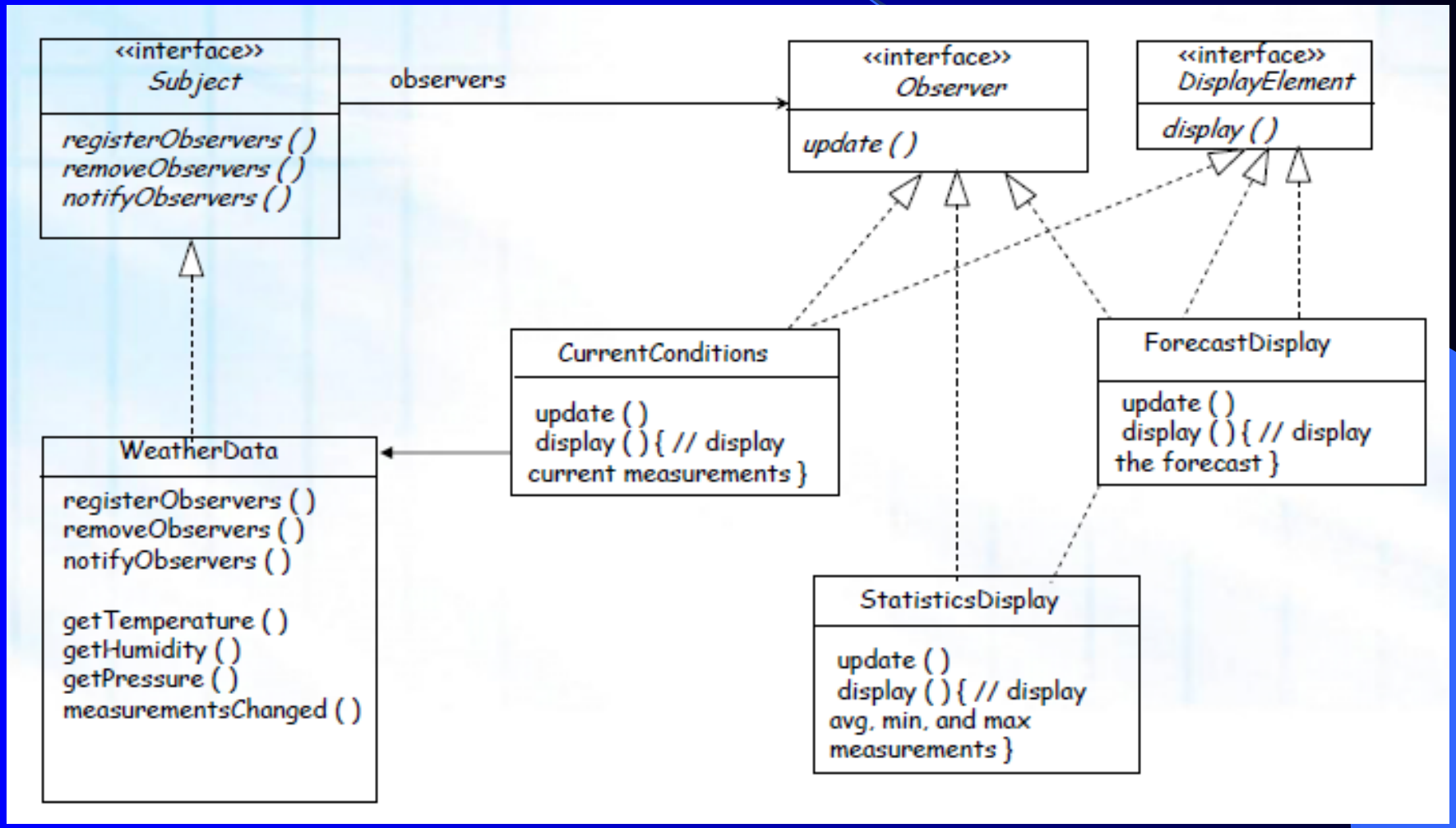
- Internet weatherstation
- Observer pattern



我们为什么要使用观察者模式？

- •Subject只需要知道有某个实现了确定的接口（指Observers接口）的观察者（Observer）就可以了。
- •我们可以在任何时候增加观察者（Observer）。
- •我们不需要修改Subject的代码来增加新类型的观察者（Observer）。
- •我们可以分别重用Subject或者Observer。
- •对于Subject和Observer的修改均不会影响对方。

Weather Station

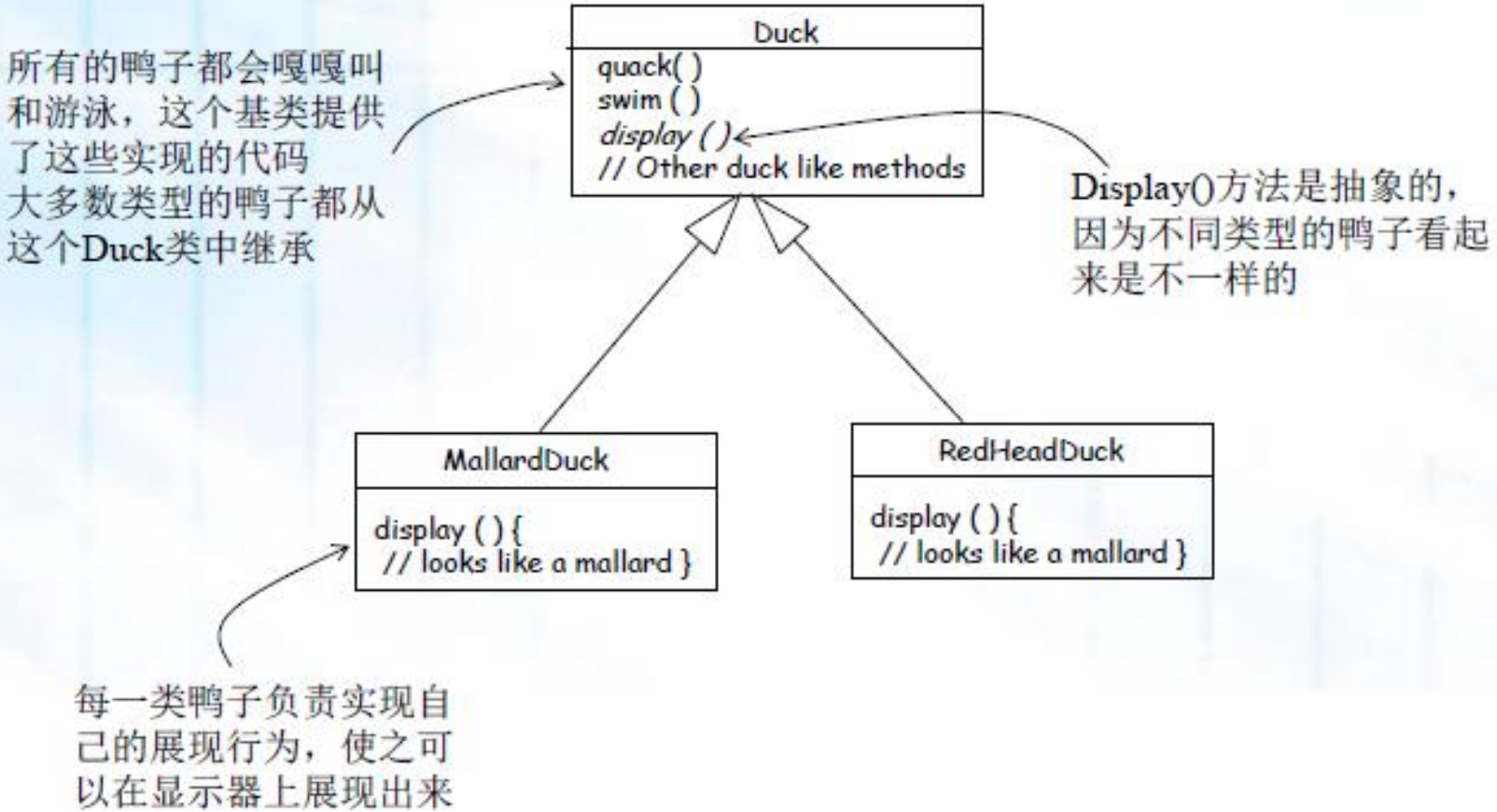


《仿真鸭》游戏

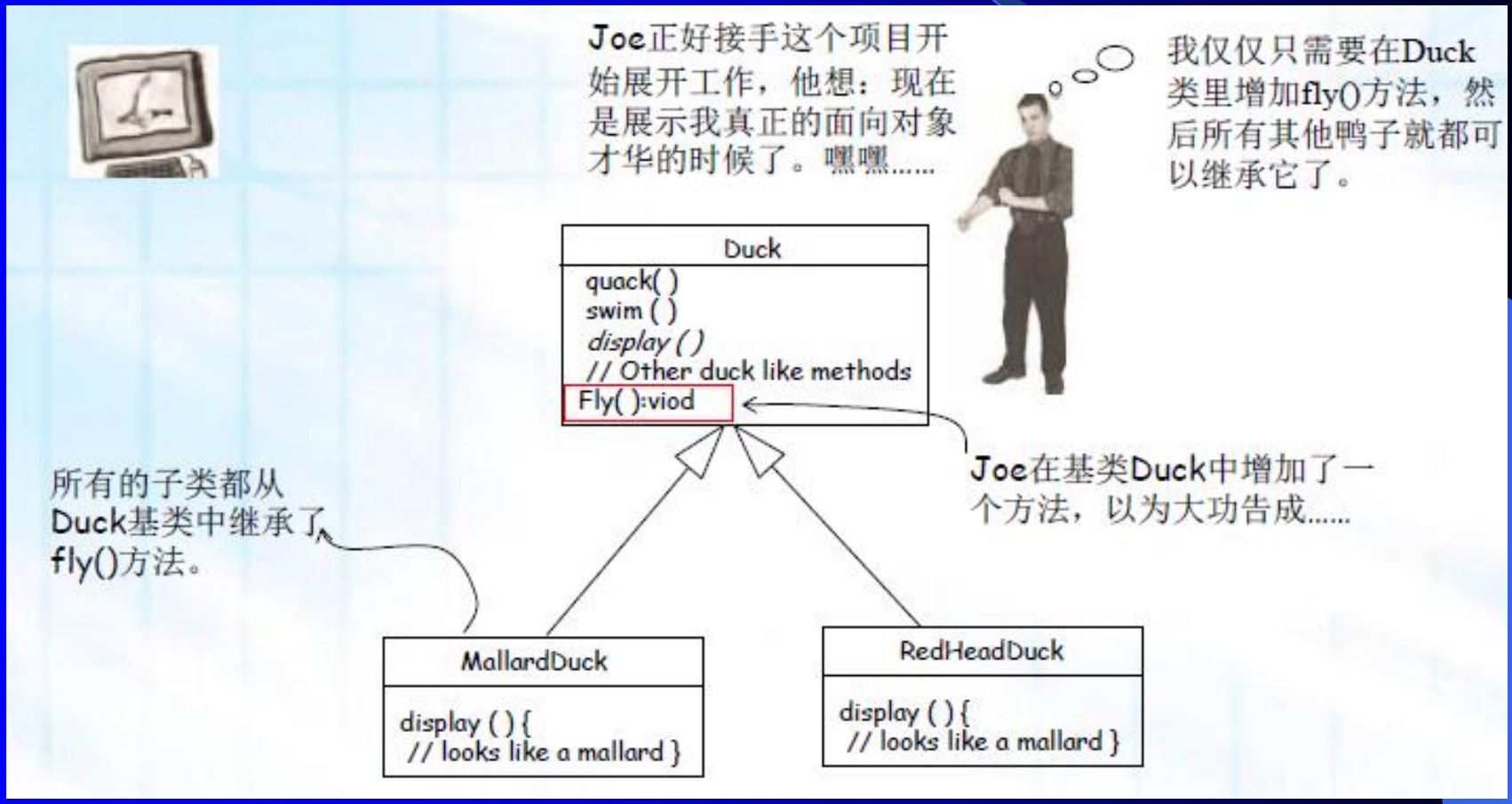
- 1，游戏中有两类鸭子，野鸭（Mallard Duck）和普通的鸭子（Red Head Duck），鸭子会游泳和嘎嘎叫（quack）
- 2，增加鸭子“飞”的功能
- 3，增加一个橡皮鸭（RubberDuck）
- 4，飞行行为发生变化（当你有100种类型的鸭子）

从一个例子开始：DuckSimulator

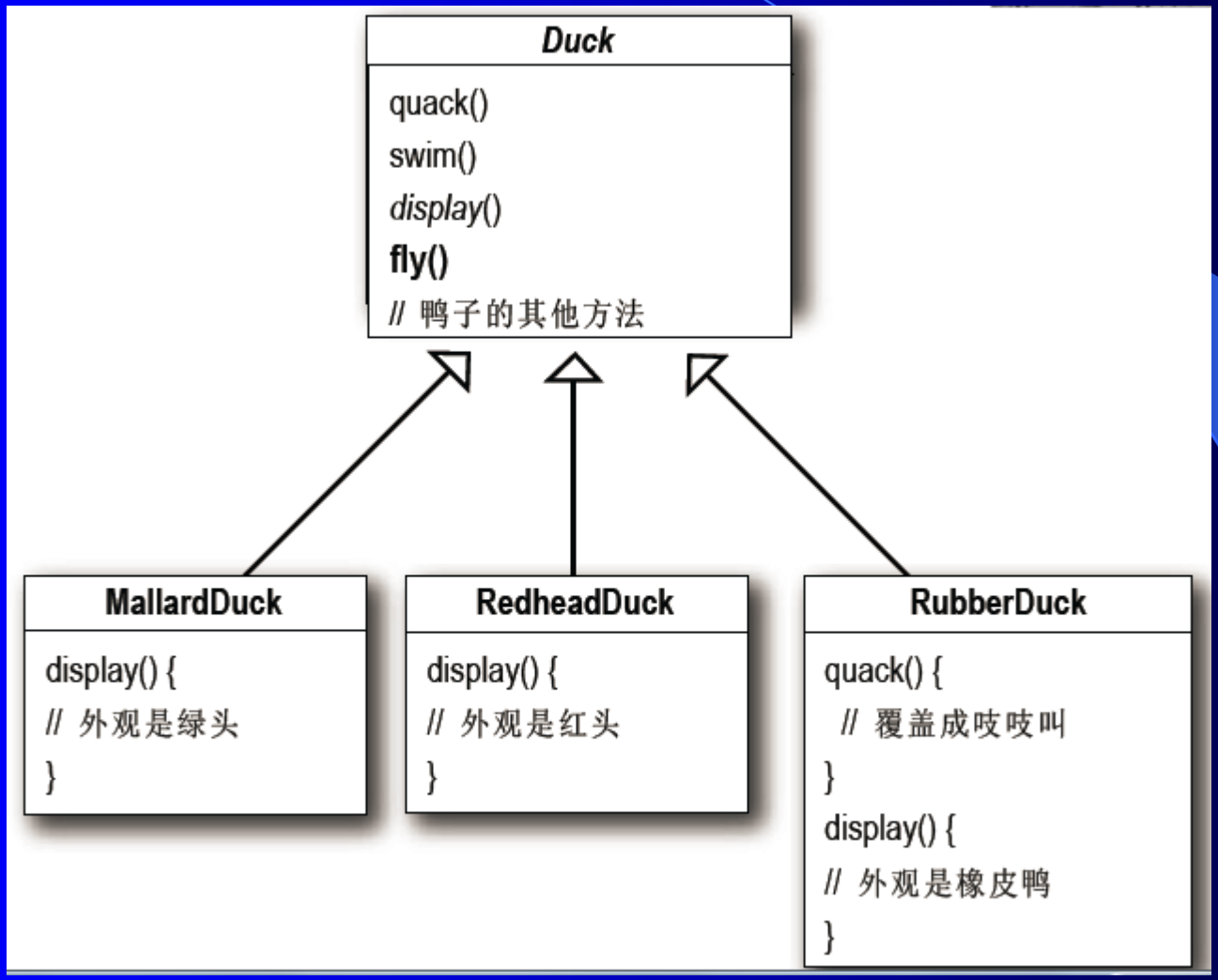
这是一个鸭池仿真游戏《仿真鸭》。这个游戏可以展现了所有种类鸭子的游泳和嘎嘎叫声。最初的系统设计者使用了标准的面向对象的技术并创建了“鸭”基类，使得其他种类的鸭可以继承。



客户要求为鸭子增加“飞”的功能



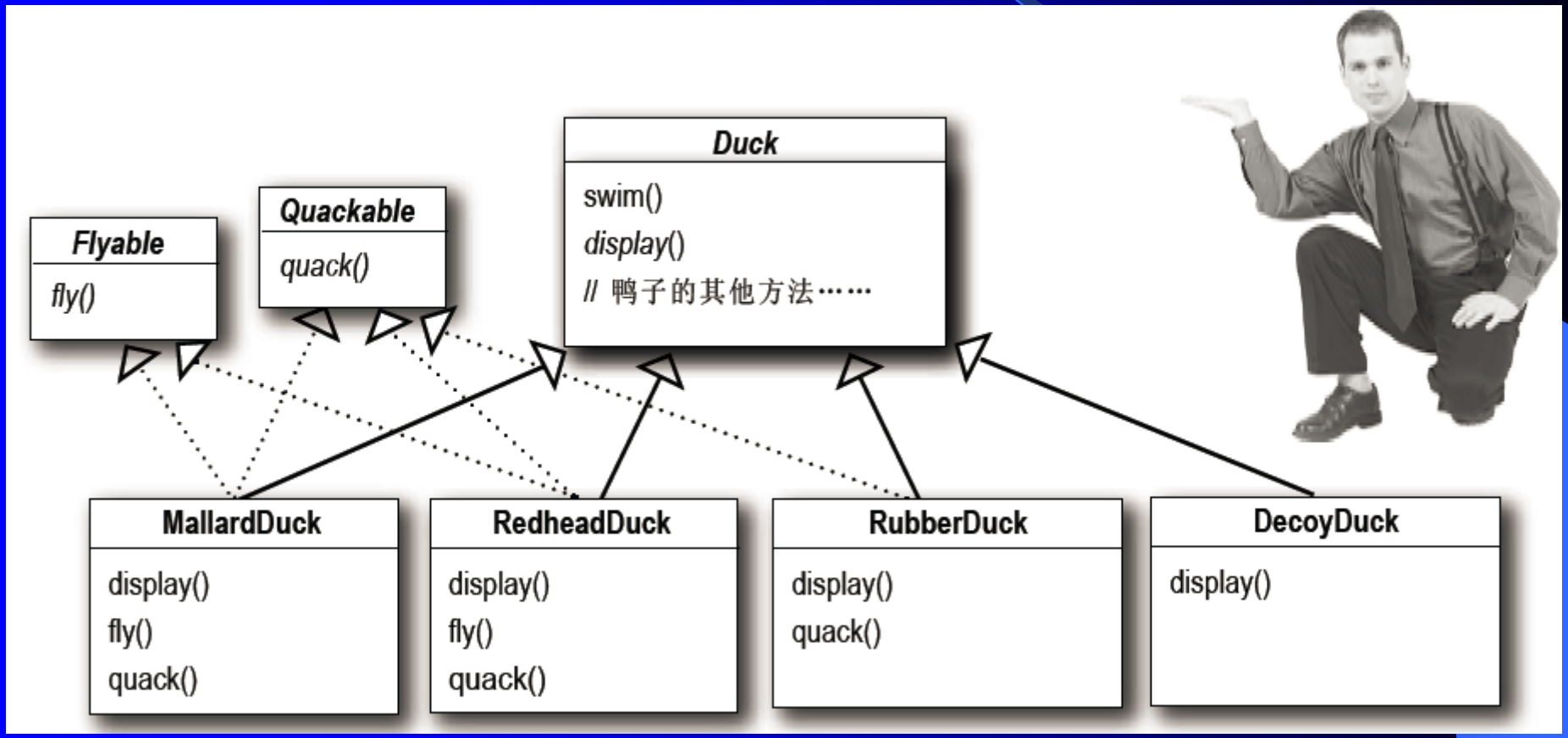
增加橡皮鸭


Duck
quack()
swim()
display()
fly()
// 鸭子的其他方法
MallardDuck
display() {
// 外观是绿头
}
RedheadDuck
display() {
// 外观是红头
}
RubberDuck
quack() {
// 覆盖成吱吱叫
}
display() {
// 外观是橡皮鸭
}

利用继承来提供Duck的行为，这会导致下列哪些缺点？（多选）

- A. 代码在子类间被复制
- B. 很难在运行时改变行为
- C. 我们不能让鸭子跳舞
- D. 很难得到所有鸭子的行为
- E. 鸭子不能在同一时间飞和呷呷叫
- F. 变动会无意间影响其他鸭子

怎么办？


Duck
swim()
display()
// 鸭子的其他方法……
Flyable
fly()
Quackable
quack()
MallardDuck
display()
fly()
quack()
RedheadDuck
display()
fly()
quack()
RubberDuck
display()
quack()
DecoyDuck
display()

我们似乎忘了些什么东西……

Design Principle

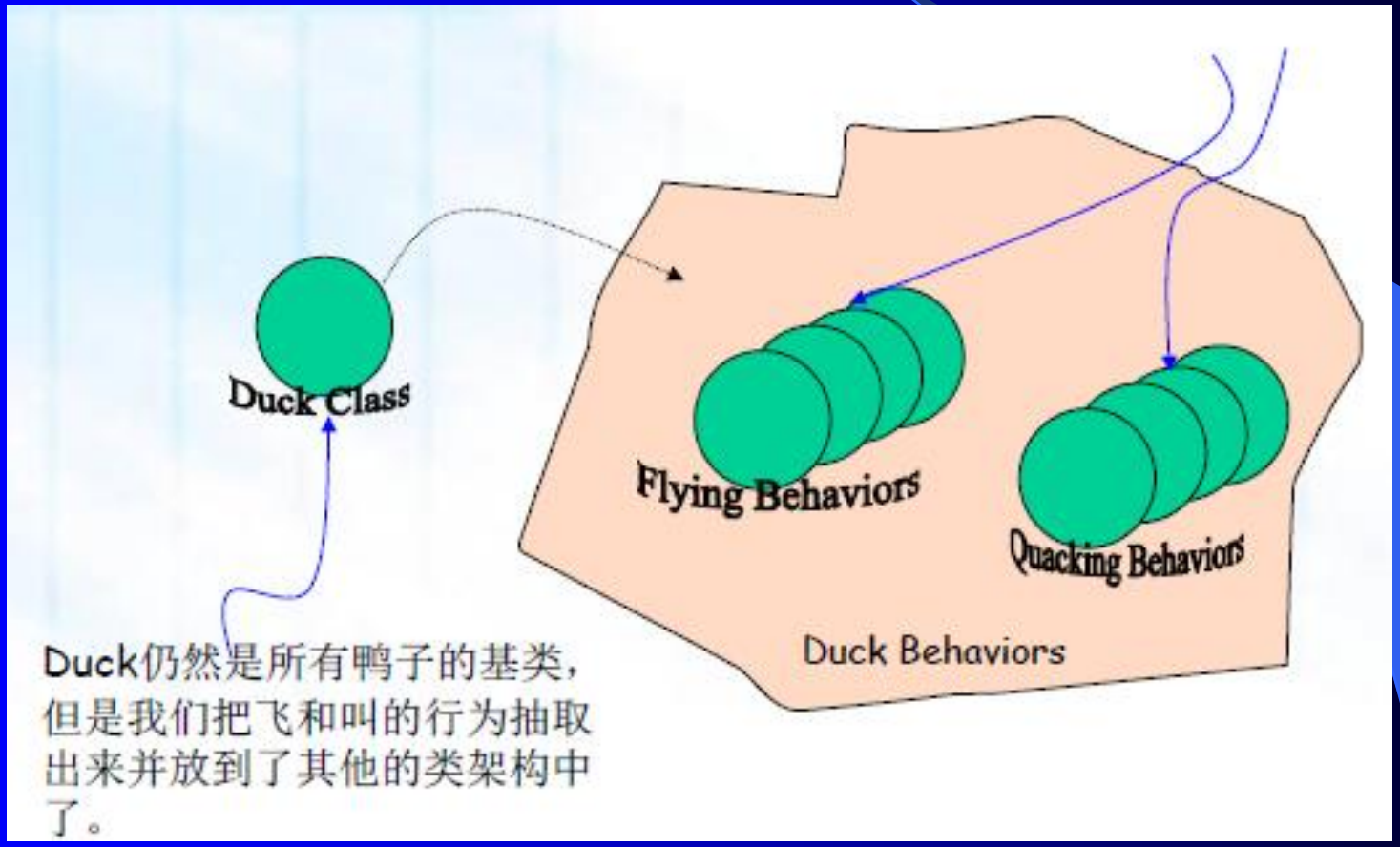
Identify the aspects of your application that vary and separate them from what stays the same.

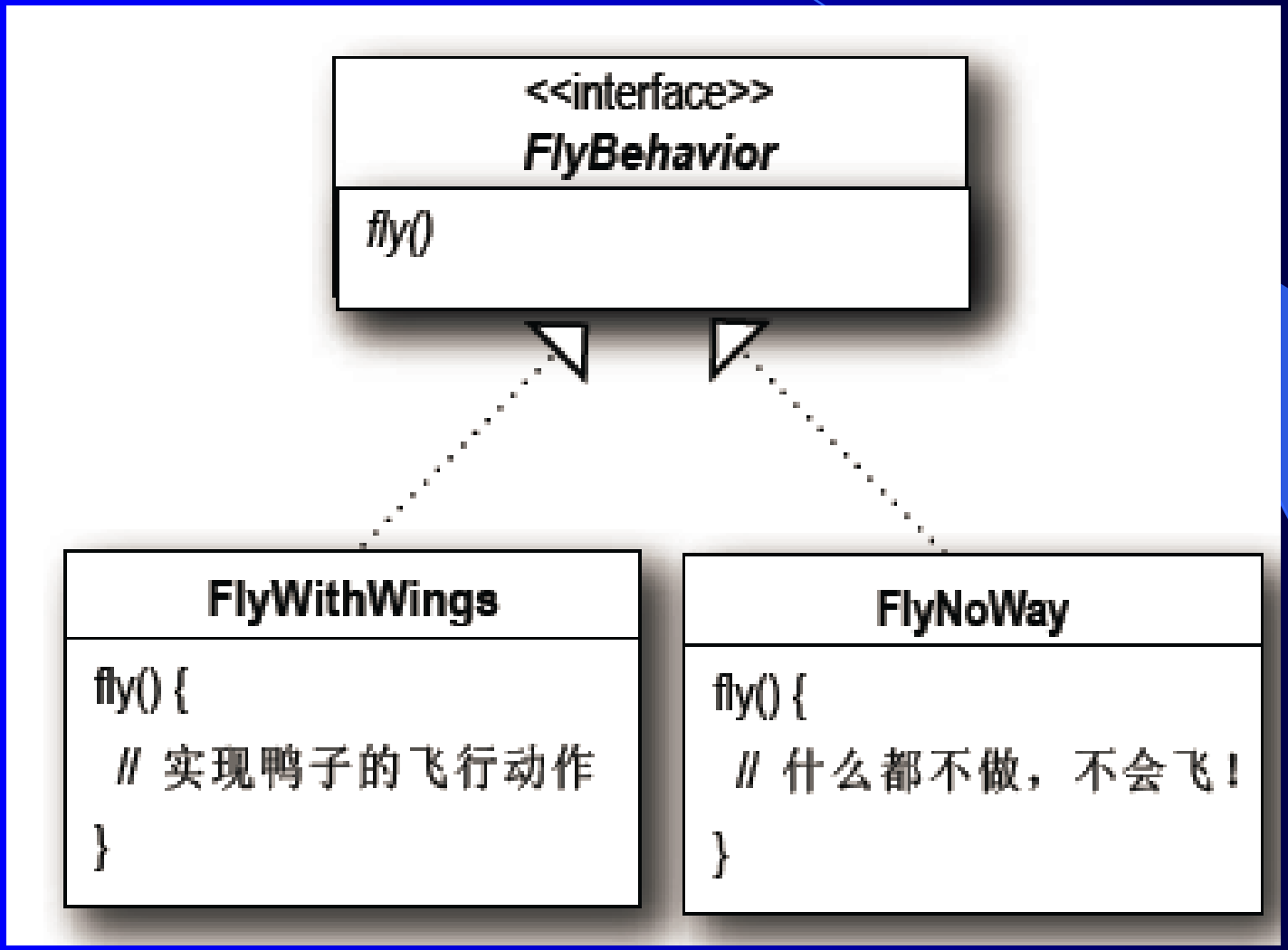
识别在程序中的那些多变的特征，并且把它们和稳定的特征分离开来

把那些变化比较快的特征封装起来，使得它不再影响其他代码的稳定性

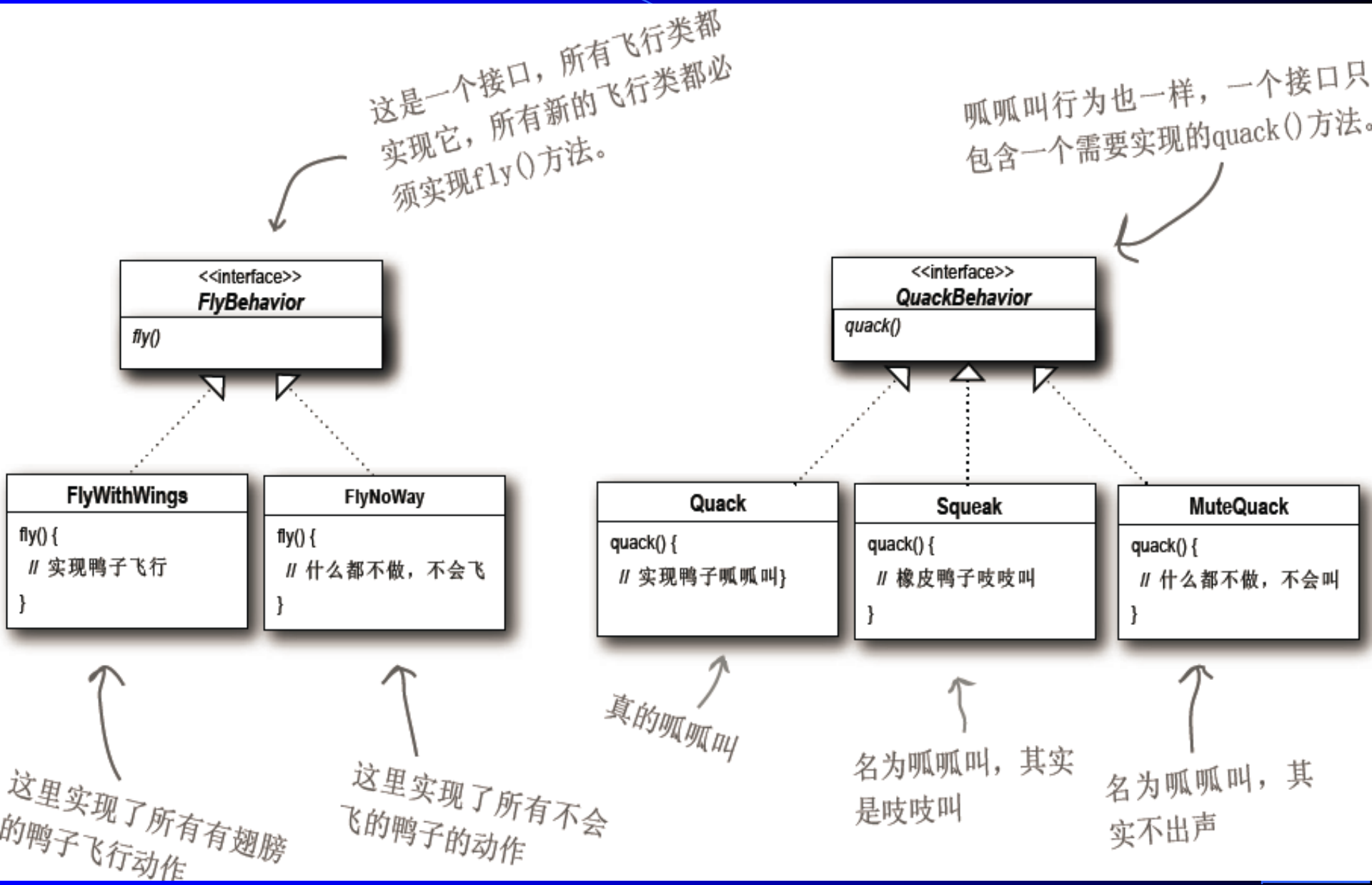
结果呢？代码变更时较少出现无意义的错误，而系统变得更加柔性。

识别可变的特征

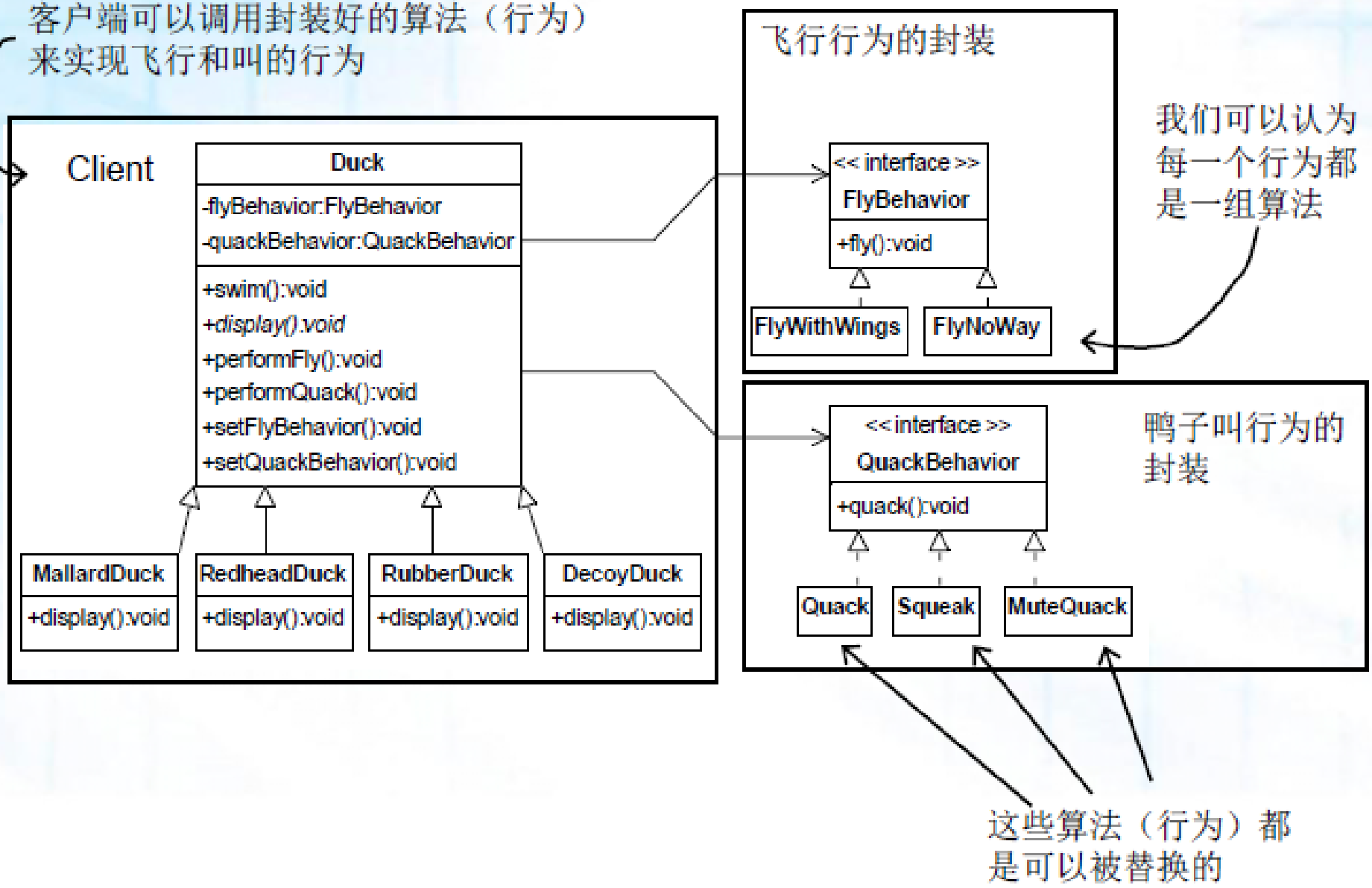

Duck Class
Flying Behaviors
Quacking Behaviors
Duck Behaviors
Duck仍然是所有鸭子的基类，
但是我们把飞和叫的行为抽取
出来并放到了其他的类架构中
了。


<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// 实现鸭子的飞行动作
}
FlyNoWay
fly() {
// 什么都不做，不会飞！
}

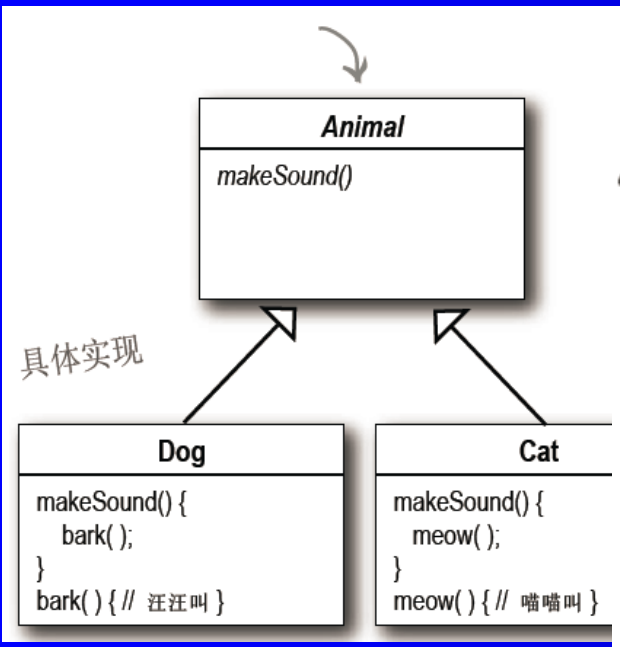
OK，我们来实现鸭子的行为



客户端可以调用封装好的算法（行为）来实现飞行和叫的行为



再来讨论一下



“针对实现编程”

```
Dog d = new Dog( );
d.bark( );
```

声明变量“d”为Dog类型（是Animal的具体实现），会造成我们必须针对具体实现编码。

但是，“针对接口/超类型编程”做法会如下：

```
Animal animal = new Dog( );
animal.makeSound( );
```

我们知道该对象是狗，但是我们现在利用animal进行多态的调用。

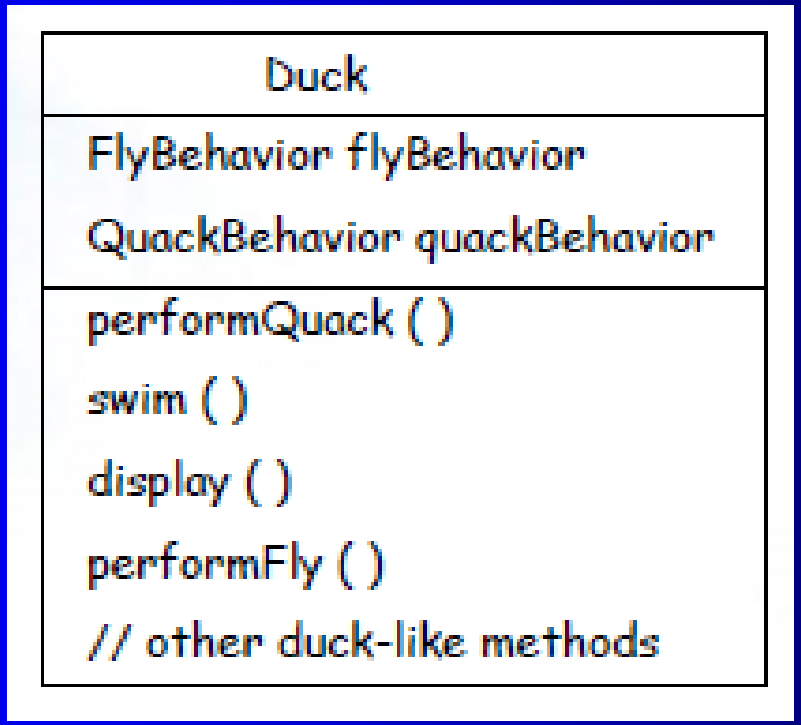
更棒的是，子类实例化的动作不再需要在代码中硬编码，例如new Dog()，而是“在运行时才指定具体实现的对象”。

```
a = getAnimal( );
a.makeSound( );
```

我们不知道实际的子类型是“什么”……我们只关心它知道如何正确地进行makeSound()的动作就够了。

接下来，我们来整合鸭子的行为

- 关键之处是**委托**，现在我们可以把飞行和嘎嘎叫的行为委托给已经定义这些方法的其他类—QuackBehavior 和FlyingBehavior



现在我们来实现 performQuack():

2

```
public class Duck {
        QuackBehavior quackBehavior;
        // more
        public void performQuack() {
                quackBehavior.quack();
        }
}
```

每个Duck都有一个被引用的变量用以实现 **QuackBehavior**接口

Duck对象把行为委托给了通过quackbehavior变量引用的对象

3 接着考虑 **flyBehavior** 和**quakcBehavior** 实例变量的设置.

```
public class MallardDuck extends Duck {
        public MallardDuck() {
                quackBehavior = new Quack();
                flyBehavior = new FlyWithWings();
        }
        public void display() {
                System.out.println("I'm a real Mallard duck");
        }
}
```

当MallardDuck实例化的时候，它的构造函数把它继承的quackBehavior实例变量初始化为一个Qucak类型的实例(一个QuackBehavior接口的具体实现类)

MallardDuck的构造函数把flyBehavior实例变量初始化为一个FlyWithWings类型的实例(一个FlyBehavior接口的具体实现类)

测试Duck的代码

1. 输入并编译下面的 Duck 类（Duck . Java）以及两页前的 MallarDuck 类（MallarDuck . Java）。

```
public abstract class Duck {
    FlyBehavior flyBehavior ;
    QuackBehavior quackBehavior ;
    public Duck () {
    }
      public abstract void display () ;

      public void performFly () {
          flyBehavior . Fly ();
        }
      public void performQuack () {
          quackBehavior . Quack () ;
      }
     public void swim ()  {
        System . out .println ( " All ducks float , even decoys ! " ) ;
     }
}
```

为行为接口类型声明两个引用变量，所有鸭子子类（在同一个package中）都继承它们。

委托给行为类

❷ 输入并编译FlyBehavior接口（FlyBehavior.java）与两个行为实现类（FlyWithWings.java与FlyNoWay.java）。

```
public interface FlyBehavior {
   public void fly();
}
```

所有飞行行为类必须实现的接口。

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("I'm flying!!");
   }
}
```

这是飞行行为的实现，给“真会”飞的鸭子用……

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("I can't fly");
   }
}
```

这是飞行行为的实现，给“不会”飞的鸭子用（包括橡皮鸭和诱饵鸭）。

❸ 输入并编译QuackBehavior接口（QuackBehavior.java）及其三个实现类（Quack.java、MuteQuack.java、Squeak.java）。

```
public interface QuackBehavior {
    public void quack();
}
```

```
public class Quack implements QuackBehavior {
    public void quack() {
        System.out.println("Quack");
    }
}
```

```
public class MuteQuack implements QuackBehavior {
    public void quack() {
        System.out.println("<< Silence >>");
    }
}
```

```
public class Squeak implements QuackBehavior {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

4 输入并编译测试类（MiniDuckSimulator.java）

```
public class MiniDuckSimulator {

    public static void main(String[] args) {

        Duck mallard = new MallardDuck();

        mallard.performQuack();

        mallard.performFly();

    }

}
```

这会调用MallardDuck继承来的performQuack()方法，进而委托给该对象的QuackBehavior对象处理（也就是说，调用继承来的quackBehavior引用对象的quack()）。

至于performFly()，也是一样的道理。

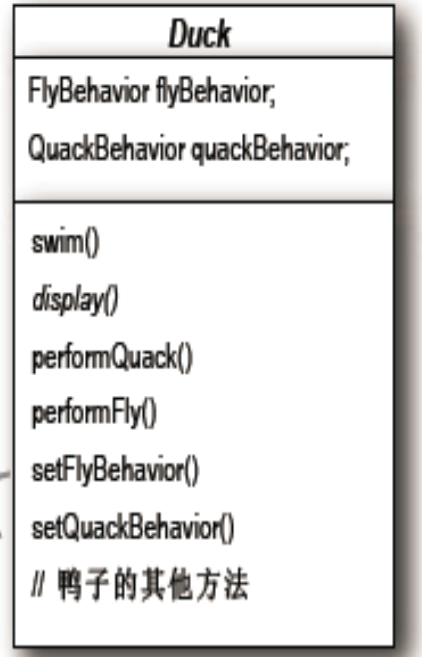
5 运行代码！

如何在运行时改变的鸭子？

动态设置Duck的行为

1. 在Duck类中，加入两个新方法：

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



从此以后，我们可以“随时”调用这两个方法改变鸭子的行为。

2 实现一个新的Duck类 (ModelDuck.java)

```
public class ModelDuck extends Duck {
        public ModelDuck() {
                flyBehavior = new FlyNoWay();
                quackBehavior = new Quack();
        }
        public void display() {
                System.out.println("I'm a model duck");
        }
}
```

3 实现一个 FlyBehavior 类 (FlyRocketPowered.java)

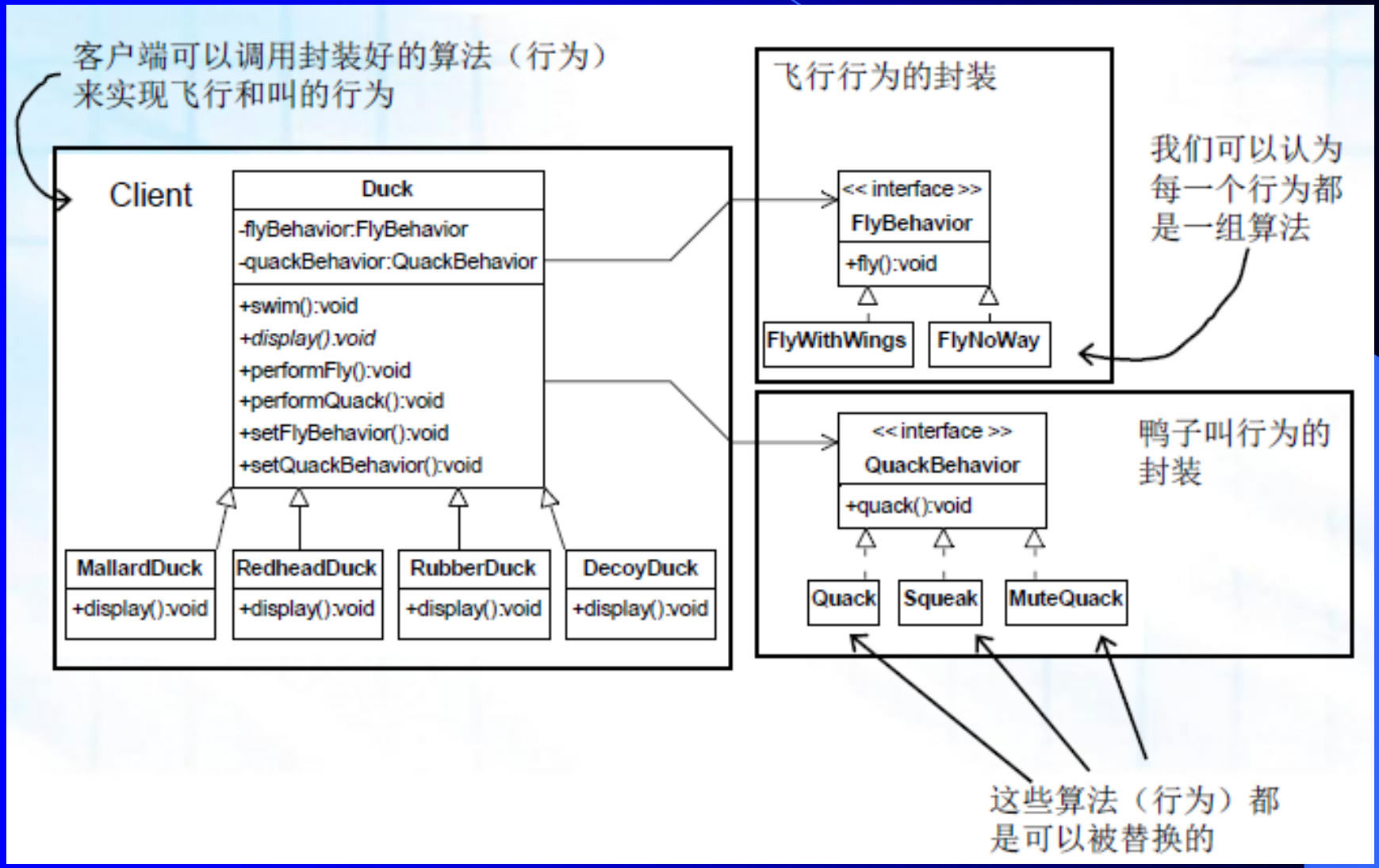
```
public class FlyRocketPowered implements FlyBehavior {
        public void fly() {
                System.out.println("I'm flying with a rocket")
        }
}
```

4 修改一下原来的测试类 (MiniDuckSImulator.Java), 增加ModelDuck, 并且让ModelDuck用FlyRocketPowered()方法来飞行

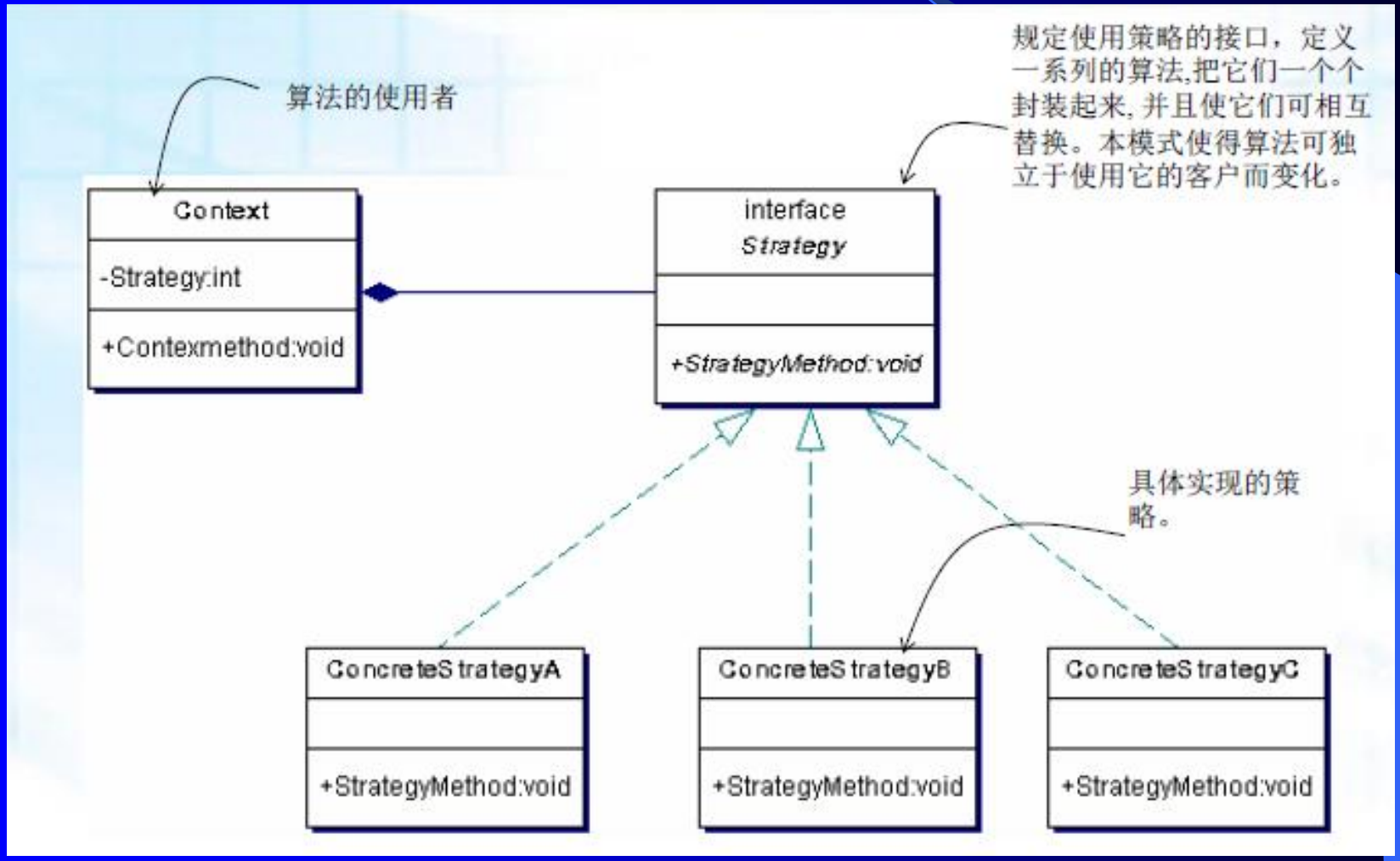
```
public class MiniDuckSimulator {
        public static void main(String[] args) {
                Duck mallard = new MallardDuck();
                mallard.performQuack();
                mallard.performFly();

                Duck model = new ModelDuck();
                model.performFly();
                model.setFlyBehavior(new FlyRocketPowered());
                model.performFly();
        }
}
```

before

本课程相关的类图


客户端可以调用封装好的算法（行为）来实现飞行和叫的行为
Client
Duck
-flyBehavior:FlyBehavior
-quackBehavior:QuackBehavior
+swim():void
+display():void
+performFly():void
+performQuack():void
+setFlyBehavior():void
+setQuackBehavior():void
MallardDuck
+display():void
RedheadDuck
+display():void
RubberDuck
+display():void
DecoyDuck
+display():void
飞行行为的封装
<< interface >>
FlyBehavior
+fly():void
FlyWithWings
FlyNoWay
我们可以认为每一个行为都是一组算法
<< interface >>
QuackBehavior
+quack():void
鸭子叫行为的封装
Quack
Squeak
MuteQuack
这些算法（行为）都是可以被替换的

策略模式相关的实体


规定使用策略的接口，定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。
算法的使用者
Context
-Strategy:int
+Contexmethod:void
interface
Strategy
+StrategyMethod:void
具体实现的策略。
ConcreteStrategyA
+StrategyMethod:void
ConcreteStrategyB
+StrategyMethod:void
ConcreteStrategyC
+StrategyMethod:void

练习1

- 数据处理程序可以对收到的数据进行排序，1，要求支持下列排序算法。2，未来可能支持新的排序算法。3，要求支持运行过程中选择不同的排序算法。4，给出main函数中的调用代码。
- 冒泡排序
- 选择排序
- 快排序
- 堆排序

练习2

- X公司需设计一款游戏软件，要求可以用多种方式显示人物的角色信息。人物的角色信息包括：身高Height，脸型Face，服饰Clothing，武器装备Weapon，武功等级Level。显示方式应包括：将人物角色信息列表显示TableView，将人物角色信息结构图显示FigureView，将人物角色信息动态显示(即玩家鼠标点击人物时显示该位置的相关信息)DynamicView。同一人物角色可以被多个玩家同时查看，角色信息可以在被查看时被修改。软件设计需保证多种显示方式下人物角色信息的一致性，即角色信息若被修改，所有显示方式都应察觉修改并更新信息。
- 假设你是X公司的首席软件架构师，请提出一种解决方案(不考虑View方法的实现细节)，给出main函数中的调用代码。