

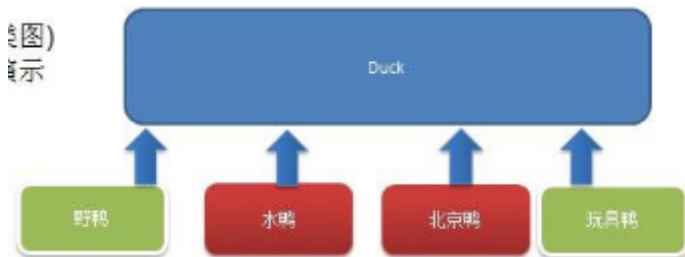
# 10.策略模式

## 编写鸭子项目，具体要求如下：

- 1) 有各种鸭子(比如 野鸭、北京鸭、水鸭等， 鸭子有各种行为，比如 叫、飞行等)
- 2) 显示鸭子的信息

## 传统方案解决鸭子问题的分析和代码实现

- 1) 传统的设计方案(类图)



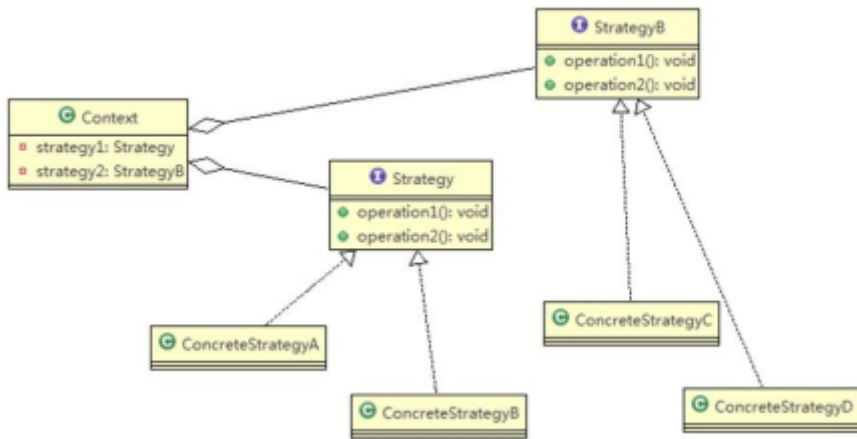
## 传统的方式实现的问题分析和解决方案

- 1) 其它鸭子，都继承了 Duck 类，所以 fly 让所有子类都会飞了，这是不正确的
- 2) 上面说的 1 的问题，其实是继承带来的问题：对类的局部改动，尤其超类的局部改动，会影响其他部分。会有溢出效应
- 3) 为了改进 1 问题，我们可以通过覆盖 fly 方法来解决 => 覆盖解决
- 4) 问题又来了，如果我们有一个玩具鸭子 ToyDuck, 这样就需要 ToyDuck 去覆盖 Duck 的所有实现的方法 => 解决思路 -》 策略模式 (strategy pattern)

## 策略模式基本介绍

- 1) 策略模式 (Strategy Pattern) 中，定义算法族 (策略组)，分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户
- 2) 这算法体现了几个设计原则，第一、把变化的代码从不变的代码中分离出来；第二、针对接口编程而不是具体类 (定义了策略接口)；第三、多用组合/聚合，少用继承 (客户通过组合方式使用策略)。

## 策略模式的原理类图



1. 象策略 (**Strategy**) 类: 定义了一个公共接口, 各种不同的算法以不同的方式实现这个接口, 环境角色使用这个接口调用不同的算法, 一般使用接口或抽象类实现。

2. 具体策略 (**Concrete Strategy**) 类: 实现了抽象策略定义的接口, 提供具体的算法实现。

3. 环境 (**Context**) 类: 持有一个策略类的引用, 最终给客户端调用。

说明: 从上图可以看到, 客户 context 有成员变量 strategy 或者其他的策略接口, 至于需要使用到哪个策略, 我们可以在构造器中指定

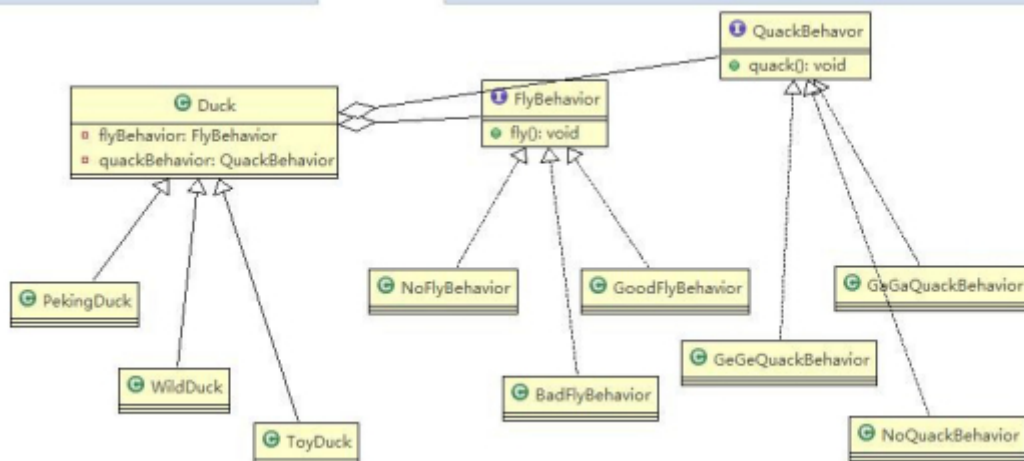
## 策略模式解决鸭子问题

### 1) 应用实例要求

编写程序完成前面的鸭子项目, 要求使用策略模式

### 2) 思路分析(类图)

策略模式: 分别封装行为接口, 实现算法族, 超类里放行为接口对象, 在子类里具体设定行为对象。原则就是: 分离变化部分, 封装接口, 基于接口编程各种功能。此模式让行为的变化独立于算法的使用者



```

public interface FlyBehavior {
    void fly(); // 子类具体实现
}
public class NoFlyBehavior implements FlyBehavior{
    @Override
    public void fly() {
        // TODO Auto-generated method stub
        System.out.println(" 不会飞翔 ");
    }
}
public class GoodFlyBehavior implements FlyBehavior {
    @Override
    public void fly() {
        // TODO Auto-generated method stub
        System.out.println(" 飞翔技术高超 ~~~");
    }
}
public abstract class Duck {
    //属性, 策略接口
    FlyBehavior flyBehavior;
    //其它属性<->策略接口
    QuackBehavior quackBehavior;

    public abstract void display();//显示鸭子信息

    public void quack() {
        System.out.println("鸭子嘎嘎叫~~");
    }

    public void swim() {
        System.out.println("鸭子会游泳~~");
    }

    public void fly() {
        //改进
        if(flyBehavior != null) {
            flyBehavior.fly();
        }
    }

    //动态的选择行为(算法)
    public void setFlyBehavior(FlyBehavior flyBehavior) {
        this.flyBehavior = flyBehavior;
    }

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

```

```
}  
}
```

## 策略模式在 JDK-Arrays 应用的源码分析

- 1) JDK 的 Arrays 的 Comparator 就使用了策略模式
- 2) 代码分析+Debug 源码+模式角色分析

策略模式在JDK-Arrays 应用的源码分析

1) JDK的 Arrays 的Comparator就使用了策略模式  
2) 代码分析+Debug源码+模式角色分析

实现

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

1

```
public class Strategy {  
    public static void main(String[] args) {  
        Integer[] data = { 9, 1, 2, 8, 4, 3 };  
        // 实现升序排序，返回-1放左边，1放右边，0保持不变  
        Comparator<Integer> comparator = new Comparator<Integer>() {  
            public int compare(Integer o1, Integer o2) {  
                if (o1 > o2) {  
                    return 1;  
                } else {  
                    return -1;  
                }  
            }  
        };  
        Arrays.sort(data, comparator);  
        System.out.println(Arrays.toString(data));  
    }  
}
```

2

```
public static<T> void sort(T[] a, Comparator<? super T> c) {  
    if (c == null) { // c==null, 就按照自己的方法排序  
        sort(a);  
    } else {  
        if (LegacyMergeSort.userRequested)  
            legacyMergeSort(a, c); //按照这个方法调用c  
        else  
            TimSort.sort(a, 0, a.length, c, null, 0, 0); //按照此方式调用比较器  
    }  
}
```

## 策略模式的注意事项和细节

- 1) 策略模式的关键是：分析项目中变化部分与不变部分
- 2) 策略模式的核心思想是：多用组合/聚合 少用继承；用行为类组合，而不是行为的继承。更有弹性
- 3) 体现了“对修改关闭，对扩展开放”原则，客户端增加行为不用修改原有代码，只要添加一种策略（或者行为）即可，避免了使用多重转移语句 (if..else if..else)
- 4) 提供了可以替换继承关系的办法：策略模式将算法封装在独立的 Strategy 类中使得你可以独立于其 Context 改变它，使它易于切换、易于理解、易于扩展
- 5) 需要注意的是：每添加一个策略就要增加一个类，当策略过多是会导致类数目庞大
- 6) 注意思考桥接模式和访问者模式