# 策略模式 Strategy Pattern

使用组合，而非继承：策略模式定义了算法族，分别封装起来，让它们之间可以相互替换。让算法的变化独立于使用算法的客户。

以电商网站为例。所有商品都继承自商品类 Commodity，因为它们有一些共同的特性，继承可以使那些共同的特性代码得到复用。现在 618 搞活动了，所有商品都要打折，但不过打折方法不一样，水果生鲜是一种打折方法，比如满 50 减 10 块，满 100 减 30；家电是一种打折方法，比如满 2000 减100，满 4000 减 400；图书是一种打折方法，日用品是一种打折方法，四个门类的商品有四套不同的打折算法。

（为了达到面相接口编程，必须使用继承。）

如果让所有商品都继承父类 Commodity 中的 discount() 方法，那么这个方法就只能定义为抽象的，不提供实现，每种商品都要提供自己的打折算法，比如苹果、香蕉、梨、西瓜使用同一套打折算法，冰箱、彩电、洗衣机使用一套打折算法。这样做的缺点在于无法做到算法的复用，不仅在开发的时候麻烦，将来维护算法的时候也麻烦——假如某个算法需要调整，就必须在每一个使用该算法的类中修改相关的代码，可能需要修改成千上万个地方。

不使用继承使用接口也一样，接口也不提供具体实现，效果跟使用继承一样。

要解决这个问题，必须将打折算法从商品定义中分离出去，定义成单独的算法类，再把算法类组合到商品类中，作为商品类的一个实例变量，商品的打折行为是委托给这些算法类来处理的：

1. 首先定义一个 Discount 接口，该接口只有一个方法 performDiscount()。
2. 再定义四个打折类，都实现 Discount 接口，并且提供 performDiscount() 方法的具体实现。
3. 最后在父类 Commodity 中声明一个 Discount 类型的实例变量，这样子类也都继承了这个实例变量。
4. 每次定义一个子类的时候，在这个子类的构造方法中为这个实例变量选择实际的算法类。但这不意味着实际的打折行为已经被绑死在子类中。因为如果父类 Commodity 中为这个实例变量提供了公共的 set 方法，那么将来可以在运行时任意改变子类的打折算法。（游戏中，角色换武器时也需要在运行时动态改变行为。）

这样做的好处是，算法代码可以复用——算法的定义仅限于四个算法类，维护也仅限于这四个地方。另外，如果希望可以在运行时改变某商品打折算法，比如结算时发现你领了券，可以使用更大的打折力度，那么只需要在父类 Commodity 中为 Discount 类型的实例变量提供一个公共的 set 方法，结算时使用 set 方法更换某个商品的打折算法即可。

再举一个例子：Character（角色）是抽象类，由具体的角色来继承。具体的角色包括国王（King）、皇后（Queen）、骑士（Knight）等。而武器（Weapon）是接口，由具体的武器来继承。所有实际的角色和武器都是具体类。任何角色如果想换武器，可以调用 setWeapon() 方法，此方法定义在 Character 超类中。在打斗过程中，会调用到目前武器的 useWeapon() 方法，攻击其他角色。

## 设计模式

* 模式不是代码，而是针对设计问题的通用解决方案。
* 模式不是被发明，而是被发现。
* 大多数模式和原则，都着眼于软件变化。
* 大多数模式都允许系统局部改变独立于其他部分。

## 设计原则

**建立弹性的设计，可以维护，可以应对变化**

| 封装变化 |

找出应用中可能需要变化的地方，把它们独立出来（封装）。

如果每次新的需求一来，都会使某方面的代码发生变化，那么这部分代码就需要被独立出来。独立出来的代码可以轻易地改动或扩充，不会影响程序中不需要变化的部分。

所有模式都提供了一套方法，让系统中的某部分改变不影响其他部分。

| 针对接口编程，不针对实现编程 |

针对接口编程的真正意思是针对超类编程。而不是针对 interface 编程。针对接口编程的关键在于利用多态，程序执行时会根据实际状况执行到真正的行为。更明确的说，就是变量的声明类型应该超类型，比如一个抽象类或者一个接口。声明时不用考虑执行时的真正对象。

| 多用组合，少用继承 |

使用组合建立系统具有很大的弹性，不仅可以将算法族封装成类，更可以在运行时动态改变类的行为。

## UML 图例

继承

实现

有一个

# 模拟鸭子应用

版本一，继承父类

VI 
Decoy Duc k.java 
Duck.java 
MallardDuckjava 
Read Head Duck.java 
RubberDuck.java 

版本二，实现接口

be havior 
Fly.java 
Quack.java 
Decoy Duc k.java 
Duck.java 
MallardDuckjava 
Red Head Duck.java 
RubberDuck.java 

版本三，策略模式，使用组合

be havior 
Fly8ehaviorjava 
Quack8ehaviorjava 
behaviorlmpl 
Fa keQuackjava 
FlyNoWayjava 
Fly RocketPowered java 
FlyWithWingsjava 
MuteQuackjava 
Quack.java 
Sq u ea k.ja va 
ducks 
u:] DecoyDuckjava 
Duck.java 
u:] MallardDuckjava 
u:] ModelDuckjava 
u:] RedHeadDuckjava 
u:] RubberDuckjava 
u:] MiniDuckSimulatorjava 

父类 Duck

package com.feng.dp.strategy.v3.ducks;

import com.feng.dp.strategy.v3.behavior.FlyBehavior;

import com.feng.dp.strategy.v3.behavior.QuackBehavior;

public abstract class Duck {

FlyBehavior flyBehavior;

QuackBehavior quackBehavior;

public Duck() {

}

// 可以动态改变飞行行为

public void setFlyBehavior(FlyBehavior fb) {

flyBehavior = fb;

}

// 可以动态改变叫声

public void setQuackBehavior(QuackBehavior qb) {

quackBehavior = qb;

}

abstract void display();

public void performFly() {

flyBehavior.fly();

}

public void performQuack() {

quackBehavior.quack();

}

public void swim() {

System.out.println("All ducks float, even decoys!");

}

}

行为“飞”的接口 FlyBehavior

public interface FlyBehavior {

void fly();

}

行为“飞”的接口的实现 FlyNoWay

import com.feng.dp.strategy.v3.behavior.FlyBehavior;

public class FlyNoWay implements FlyBehavior {

@Override

public void fly() {

System.out.println("I can't fly");

}

}

行为“飞”的接口的实现 FlyWithWings

import com.feng.dp.strategy.v3.behavior.FlyBehavior;

public class FlyWithWings implements FlyBehavior {

@Override

public void fly() {

System.out.println("I'm flying with wings!");

}

}

行为“飞”的接口的实现 FlyRocketPowered

import com.feng.dp.strategy.v3.behavior.FlyBehavior;

public class FlyRocketPowered implements FlyBehavior {

@Override

public void fly() {

System.out.println("I'm flying with a rocket");

}

}

行为“叫”的接口 QuackBehavior

public interface QuackBehavior {

void quack();

}

行为“叫”的接口的实现 Quack

import com.feng.dp.strategy.v3.behavior.QuackBehavior;

public class Quack implements QuackBehavior {

@Override

public void quack() {

System.out.println("Quack! Quack! Quack!");

}

}

行为“叫”的接口的实现 Squeak

import com.feng.dp.strategy.v3.behavior.QuackBehavior;

public class Squeak implements QuackBehavior {

@Override

public void quack() {

System.out.println("Squeak! Squeak! Squeak!");

}

}

行为“叫”的接口的实现 MuteQuack

import com.feng.dp.strategy.v3.behavior.QuackBehavior;

public class MuteQuack implements QuackBehavior{

@Override

public void quack() {

System.out.println("<< Silence >>");

}

}

鸭子 MallardDuck

import com.feng.dp.strategy.v3.behavior.FlyBehavior;

import com.feng.dp.strategy.v3.behavior.QuackBehavior;

import com.feng.dp.strategy.v3.behaviorImpl.FlyWithWings;

import com.feng.dp.strategy.v3.behaviorImpl.Quack;

public class MallardDuck extends Duck {

public MallardDuck() {

this.flyBehavior = new FlyWithWings();

this.quackBehavior = new Quack();

}

@Override

public void display() {

System.out.println("I've got a green head!");

}

}

鸭子 RedHeadDuck

import com.feng.dp.strategy.v3.behavior.FlyBehavior;

import com.feng.dp.strategy.v3.behavior.QuackBehavior;

import com.feng.dp.strategy.v3.behaviorImpl.FlyWithWings;

import com.feng.dp.strategy.v3.behaviorImpl.Quack;

public class RedHeadDuck extends Duck {

public RedHeadDuck() {

this.flyBehavior = new FlyWithWings();

this.quackBehavior = new Quack();

}

@Override

public void display() {

System.out.println("I've got a red head!");

}

}

鸭子 RubberDuck

import com.feng.dp.strategy.v3.behavior.FlyBehavior;

import com.feng.dp.strategy.v3.behavior.QuackBehavior;

import com.feng.dp.strategy.v3.behaviorImpl.FlyNoWay;

import com.feng.dp.strategy.v3.behaviorImpl.Squeak;

public class RubberDuck extends Duck {

public RubberDuck() {

this.flyBehavior = new FlyNoWay();

this.quackBehavior = new Squeak();

}

@Override

public void display() {

System.out.println("My whole body is yellow!");

}

}

鸭子 DecoyDuck

import com.feng.dp.strategy.v3.behavior.FlyBehavior;

import com.feng.dp.strategy.v3.behavior.QuackBehavior;

import com.feng.dp.strategy.v3.behaviorImpl.FlyNoWay;

import com.feng.dp.strategy.v3.behaviorImpl.MuteQuack;

public class DecoyDuck extends Duck {

public DecoyDuck() {

this.flyBehavior = new FlyNoWay();

this.quackBehavior = new MuteQuack();

}

@Override

public void display() {

System.out.println("I'm wooden!");

}

}

鸭子 ModelDuck

import com.feng.dp.strategy.v3.behaviorImpl.FlyNoWay;

import com.feng.dp.strategy.v3.behaviorImpl.Quack;

public class ModelDuck extends Duck {

public ModelDuck() {

flyBehavior = new FlyNoWay();

quackBehavior = new Quack();

}

@Override

public void display() {

System.out.println("I'm a model duck");

}

}

| 测试程序 |

import com.feng.dp.strategy.v3.behaviorImpl.FlyRocketPowered;

import com.feng.dp.strategy.v3.ducks.DecoyDuck;

import com.feng.dp.strategy.v3.ducks.Duck;

import com.feng.dp.strategy.v3.ducks.MallardDuck;

import com.feng.dp.strategy.v3.ducks.ModelDuck;

import com.feng.dp.strategy.v3.ducks.RubberDuck;

public class MiniDuckSimulator {

public static void main(String[] args) {

MallardDuck  mallard = new MallardDuck();

RubberDuck   rubberDuckie = new RubberDuck();

DecoyDuck    decoy = new DecoyDuck();

Duck model = new ModelDuck();

mallard.performQuack();

rubberDuckie.performQuack();

decoy.performQuack();

model.performFly();

model.setFlyBehavior(new FlyRocketPowered()); // 动态改变飞行行为

model.performFly();

}

}

输出：

Quack! Quack! Quack!

Squeak! Squeak! Squeak!

<< Silence >>

I can't fly

I'm flying with a rocket