

# 策略模式

## 案例引入

- 描述: 根据客户所购买的商品的单价和数量, 向顾客收费
- 初步解决方案: 两个文本框来输入单价和数量, 一个确定按钮来算出每种商品的费用, 列表框来记录商品清单, 一个标签来记录总计, 一个重置按钮来重新开始

## 问题提出

- 描述: 商场要对商品搞活动, 所有商品打八折
- 解决1: 在price的时候乘上0.8
  - 问题: 活动结束后还要修改程序源代码
- 解决2: 实现确定好打折情况, 然后在switch里面判断, 开始出现代码冗余

## 问题进一步提出

- 描述: 继续增加一个满300送100的活动
- 解决: 利用简单工厂实现
  - 描述: 在打折的过程中, 打折是相同的, 不同的是打折的具体折数, 实现的方式不同, 如果针对不同的折再写不同的子类去继承, 那么代码也会冗余, 那么这个时候可以写一个打折的类, 只要传入不同的打折数就可以实现, 同时再写一个满减的类来计算满减

## 问题进一步提出

- 描述: 在原来工厂的基础之上需要增加一种促销手段, 满100积分10点, 积分到一定可以领取奖品
- 解决: 新建一个积分类, 然后再去实现父类的CashAccept方法。然后再收费对象工厂中新增加一个分支条件。
- 问题: 简单工厂解决了对对象的创建问题, 商场有时会更改打折额度和返利额度, 每次维护或者更改收费方式都要更改这个工厂。每次都要重新编译代码, 浪费开销。

## 策略模式

- 描述: 对算法的封装, 让算法之间可以相互替换, 但是不影响使用算法的客户。策略模式就是要封装变化点
- UML描述:
  - 一个Strategy策略类, 定义了所有算法的公共接口
  - 一个Context上下文类, 它持有一个Strategy对象的引用, 构造方法的参数是策略方式, 类中还有一个方法是获得他们共同实现父类的方法的结果
- 代码:
  - 服务端 2个类, 一个是Strategy, 一个是context
  - 客户端 switch来判断用哪个策略, 但是这样向客户端暴露了两个类, 同时也会有冗余

## 策略模式结合简单工厂

- 描述: 把客户端中写的switch放到context类的构造方法中, 然后再构造方法中判断并生成相应的对象

## 策略模式总结

- 所有的算法都是为了完成了相同的工作，只不过是完成的形式不同。
- 策略模式减少了算法类和算法使用类的耦合
- 简化单元测试，修改其中的算法不会影响其他的算法，易维护，易扩展，易复用
- 减少了代码的重复
- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道算法或行为的情况
- 由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观

page 48