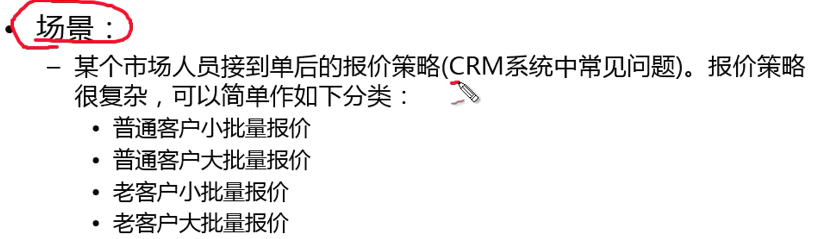
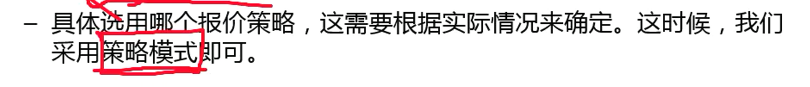
策略模式 Strategy

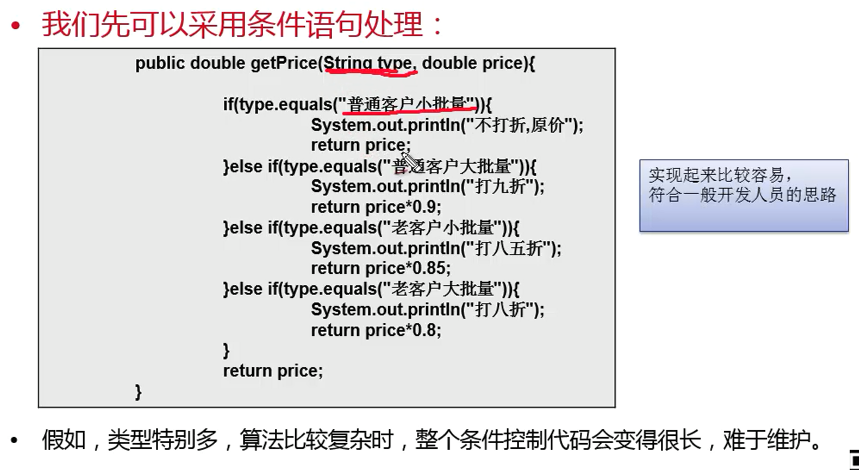


客户是有分类的 普通客户 和老客户的报价策略不一样



采用不同的策略 应对不同的类型

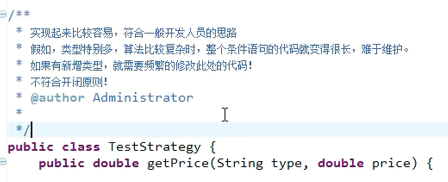
传统 不采用模式

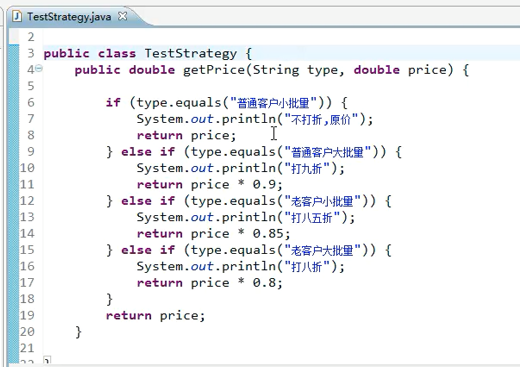
【当时那个责任链模式也是这样的 感觉也是一个请求 在责任链上跑 遇到合适的就。。。】

下一步扩展的时候 现在要增加新的客户类型 不得不频繁修改代码 违反了开闭原则

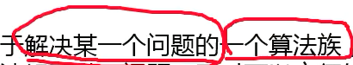
如果逻辑复杂的话 并且 每一部分的代码也很复杂

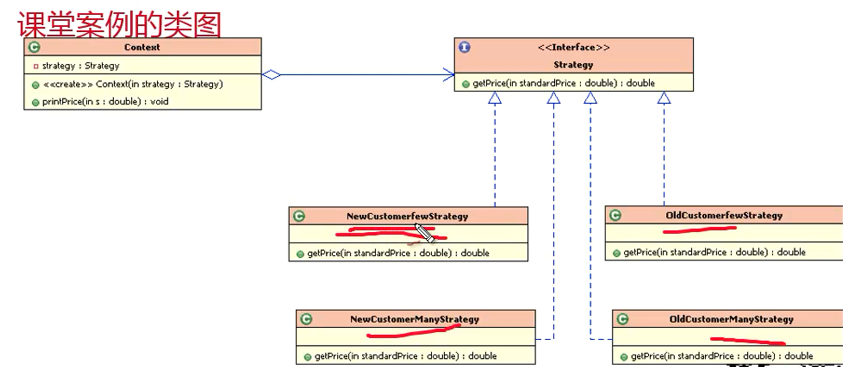
这样就是 需求的越来越多复杂 这个代码越来越难以维护





下面 采用策略模式来做这件事



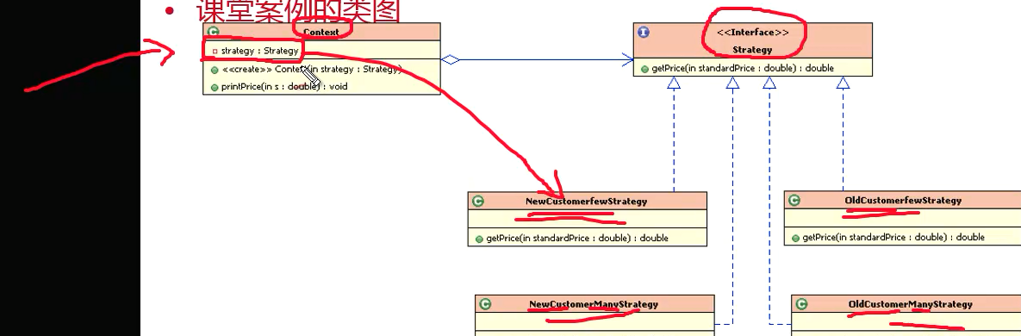


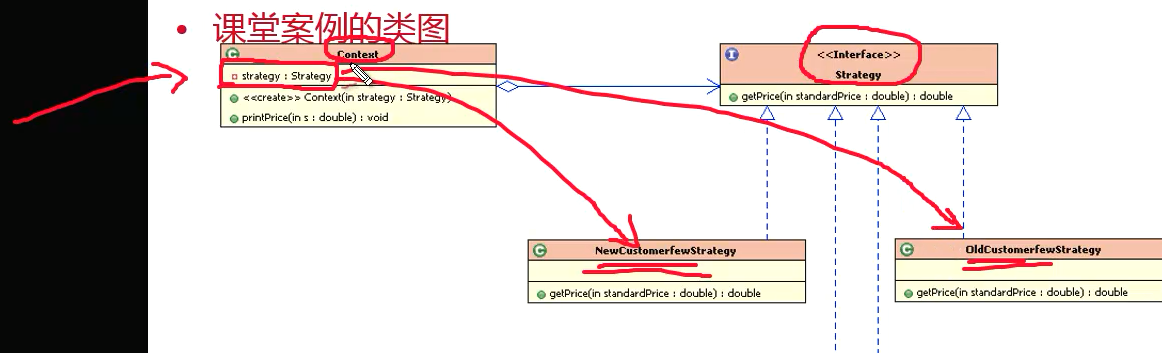
实现了同样的接口

新增加算法 及时加一个子类 -----

还有一个Context对象

客户端调用Context 就要看 strategy指向了哪个具体的实现类 就是那种策略



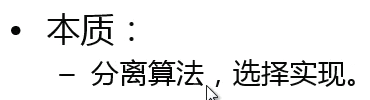


只要替换引用就可以了

这样 可以使用配置文件配置 然后使用反射

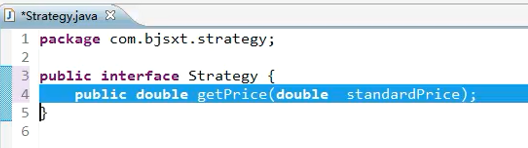
可以通过程序调用 也可以使用配置文件调用

===== 算法实现了分离



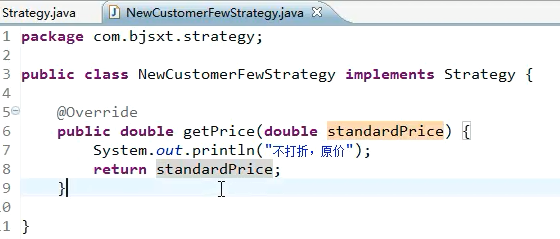
这么多的设计模式 都是把流程更容易扩展 适应后期更复杂的情况

定义一个接口Strategy

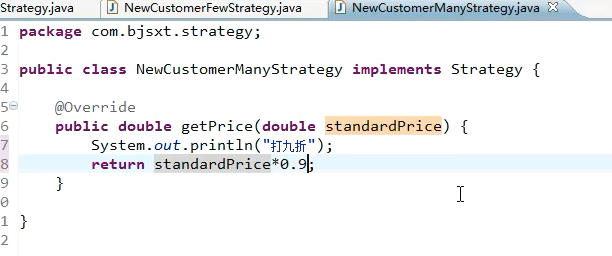


针对这个接口增加一个实现类

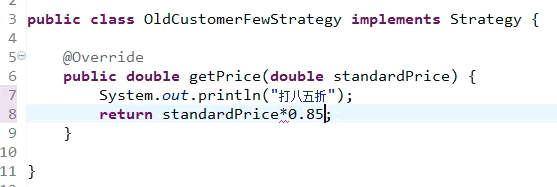
普通客户小批量购买



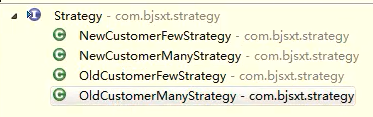
普通用户大批量



老客户 小批量



老客户 大批量

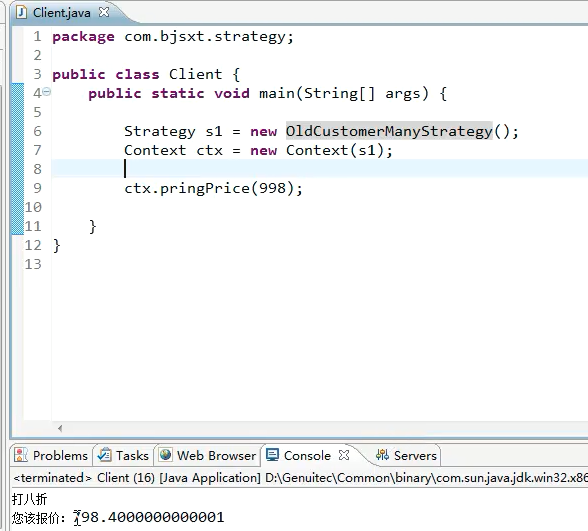
 --- 构成了算法簇

还有一个Context 用于管理算法

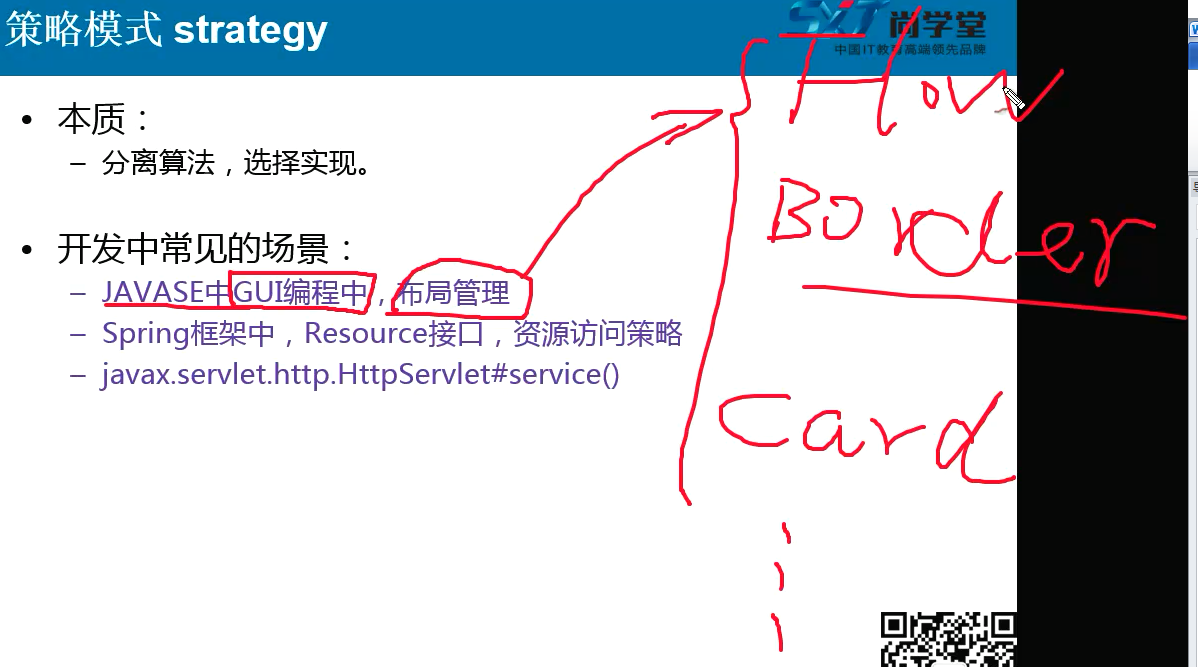


可以通过构造器来注入 或者setter注入 如果使用spring的依赖注入功能 还可以通过配置文件 动态注入不同的对象





【貌似 不能自动使用选择 没有职责连方便】



这个GUI的Flowlayout BorderlayOut 都有自己的布局算法

HttpServlet调用不同的子Servlet的service的时候是策略模式