私塾在线《研磨设计模式》
——跟着CC学设计系列精品教程

本节课程概览

- 学习策略模式

一：初识策略模式

包括：定义、结构、参考实现

二：体会策略模式

包括：场景问题、不用模式的解决方案、使用模式的解决方案

三：理解策略模式

包括：认识策略模式、Context和Strategy的关系 、容错恢复机制、策略模式结合模板方法模式、策略模式的优缺点

四：思考策略模式

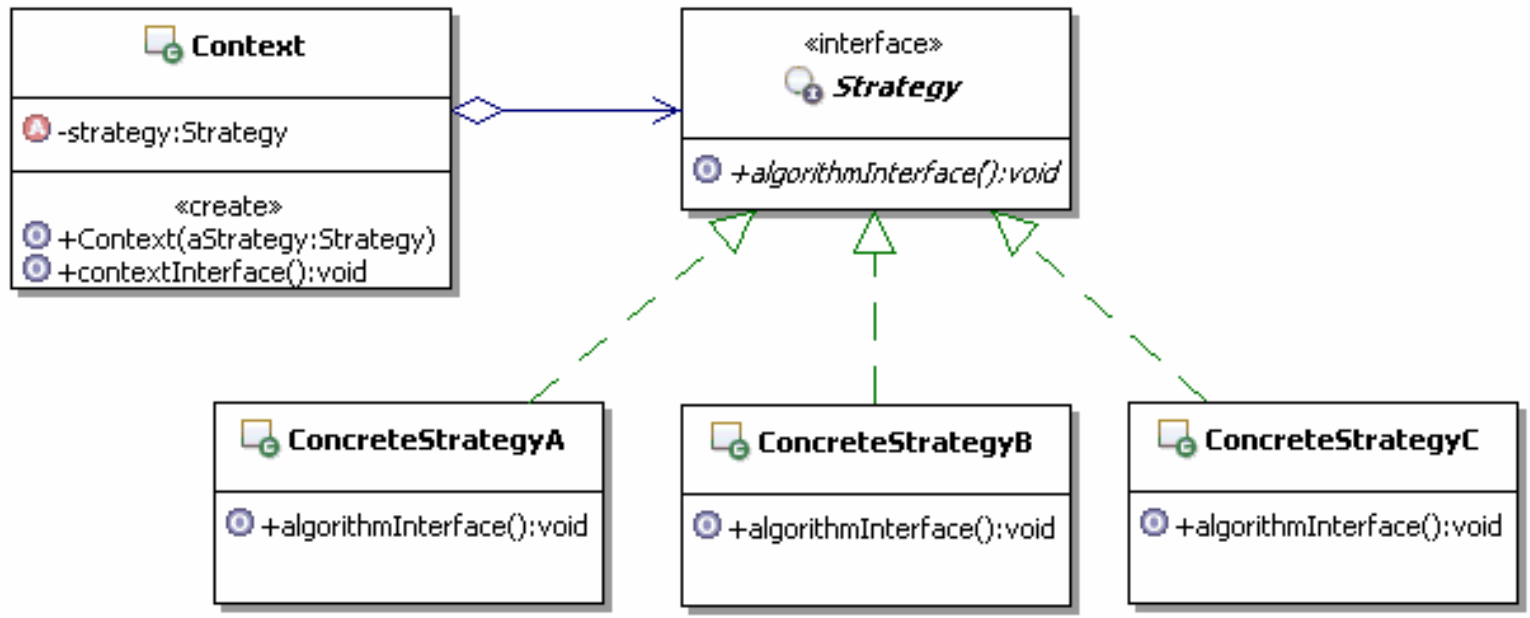
包括：策略模式的本质、对设计原则的体现、何时选用

初识策略模式

- 定义

  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 。

- 结构和说明



初识策略模式

Strategy：

策略接口，用来约束一系列具体的策略算法。Context使用这个接口来调用具体的策略实现定义的算法。

ConcreteStrategy：

具体的策略实现，也就是具体的算法实现。

Context：

上下文，负责和具体的策略类交互，通常上下文会持有一个真正的策略实现，上下文还可以让具体的策略类来获取上下文的数据，甚至让具体的策略类来回调上下文的方法。

体会策略模式

- 报价管理

向客户报价，对于销售部门的人来讲，这是一个非常重大、非常复杂的问题，对不同的客户要报不同的价格，比如：

1：对普通客户或者是新客户报的是全价
2：对老客户报的价格，根据客户年限，给予一定的折扣
3：对大客户报的价格，根据大客户的累计消费金额，给予一定的折扣
4：还要考虑客户购买的数量和金额，比如：虽然是新用户，但是一次购买的数量非常大，或者是总金额非常高，也会有一定的折扣
5：还有，报价人员的职务高低，也决定了他是否有权限对价格进行一定的浮动折扣
6：甚至在不同的阶段，对客户的报价也不同，一般情况是刚开始比较高，越接近成交阶段，报价越趋于合理。

总之，向客户报价是非常复杂的，因此在一些CRM（客户关系管理）的系统中，会有一个单独的报价管理模块，来处理复杂的报价功能。

体会策略模式

为了演示的简洁性，假定现在需要实现一个简化的报价管理，实现如下的功能：

（1）对普通客户或者是新客户报全价

（2）对老客户报的价格，统一折扣5%

（3）对大客户报的价格，统一折扣10%

该怎么实现呢？

体会策略模式

- 不用模式的解决方案

- 有何问题

上面的写法是很简单的，也很容易想，但是仔细想想，这样实现，问题可不小，比如：

第一个问题：价格类包含了所有计算报价的算法，使得价格类，尤其是报价这个方法比较庞杂，难以维护。

第二个问题：经常会有这样的需要，在不同的时候，要使用不同的计算方式。

那么到底应该如何实现，才能够让价格类中的计算报价的算法，能很容易的实现可维护、可扩展，又能动态的切换变化呢？

体会策略模式

- 使用模式来解决的思路

仔细分析上面的问题，先来把它抽象一下，各种计算报价的计算方式就好比是具体的算法，而使用这些计算方式来计算报价的程序，就相当于是使用算法的客户。

再分析上面的实现方式，为什么会造成那些问题，根本原因，就在于算法和使用算法的客户是耦合的，甚至是密不可分的，在上面实现中，具体的算法和使用算法的客户是同一个类里面的不同方法。

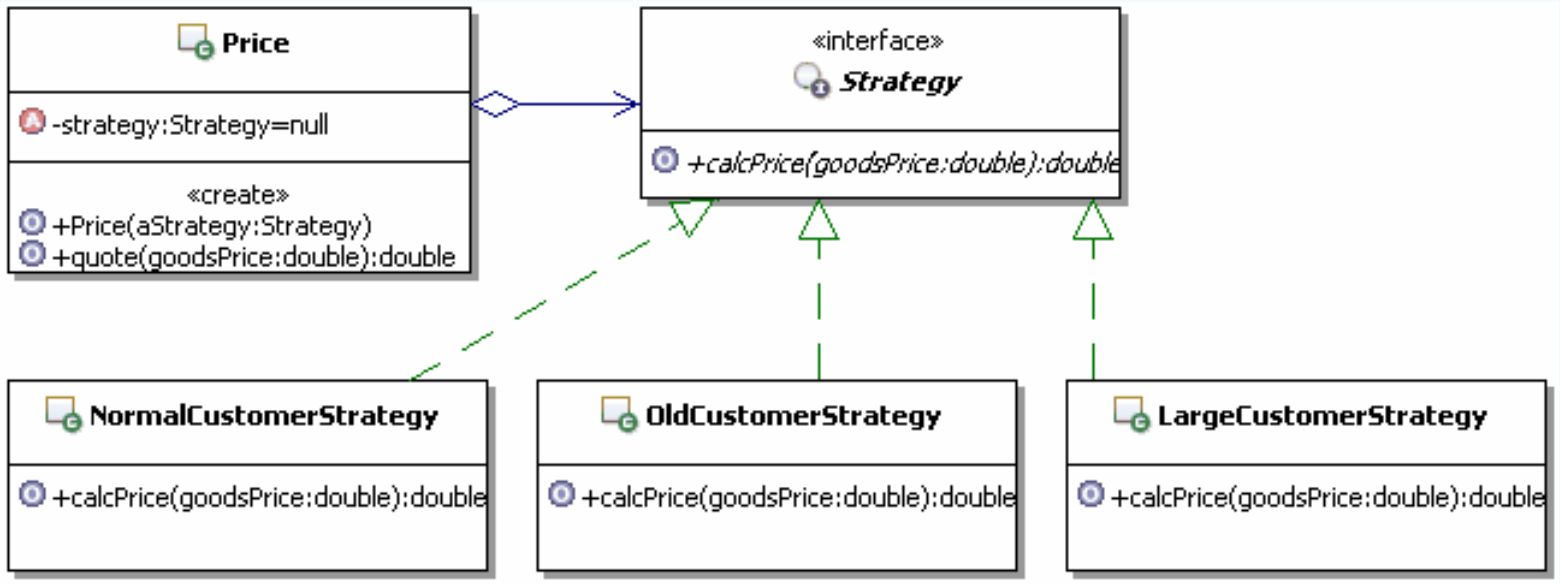
现在要解决那些问题，按照策略模式的方式，应该先把所有的计算方式独立出来做成单独的算法类。

然后引入上下文对象来实现具体的算法和直接使用算法的客户是分离的。

具体的算法和使用它的客户分离过后，使得算法可独立于使用它的客户而变化，并且能够动态的切换需要使用的算法，只要客户端动态的选择不同的算法，然后设置到上下文对象中去，实际调用的时候，就可以调用到不同的算法。

体会策略模式

- 使用模式的解决方案的类图



理解策略模式

n 认识策略模式

1：策略模式的功能

策略模式的功能是把具体的算法实现，从具体的业务处理里面独立出来，实现成为单独的算法类，从而形成一系列的算法，并让这些算法可以相互替换。

策略模式的重心不是如何来实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活、具有更好的维护性和扩展性

2：策略模式和if-else语句

看了前面的示例，很多朋友会发现，每个策略算法具体实现的功能，就是原来在if-else结构中的具体实现。没错，其实多个if-elseif语句表达的就是一个平等的功能结构，你要么执行if，要不你就执行else，或者是elseif，这个时候，if块里面的实现和else块里面的实现从运行地位上来讲就是平等的。

而策略模式就是把各个平等的具体实现封装到单独的策略实现类了，然后通过上下文来与具体的策略类进行交互。

因此多个if-else语句可以考虑使用策略模式

理解策略模式

3：算法的平等性

策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正是因为这个平等性，才能实现算法之间可以相互替换。

所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。

所以可以这样描述这一系列策略算法：**策略算法是相同行为的不同实现**。

4：谁来选择具体的策略算法

在策略模式中，可以在两个地方来进行具体策略的选择。

一个是在客户端，在使用上下文的时候，由客户端来选择具体的策略算法，然后把这个策略算法设置给上下文。前面的示例就是这种情况。

还有一个是客户端不管，由上下文来选择具体的策略算法，这个在后面讲容错恢复的时候给大家演示一下。

理解策略模式

5：Strategy的实现方式

在前面的示例中，Strategy都是使用的接口来定义的，这也是常见的实现方式。但是如果多个算法具有公共功能的话，可以把Strategy实现成为抽象类，然后把多个算法的公共功能实现到Strategy里面。

6：运行时策略的唯一性

运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态的在不同的策略实现中切换，但是同时只能使用一个。

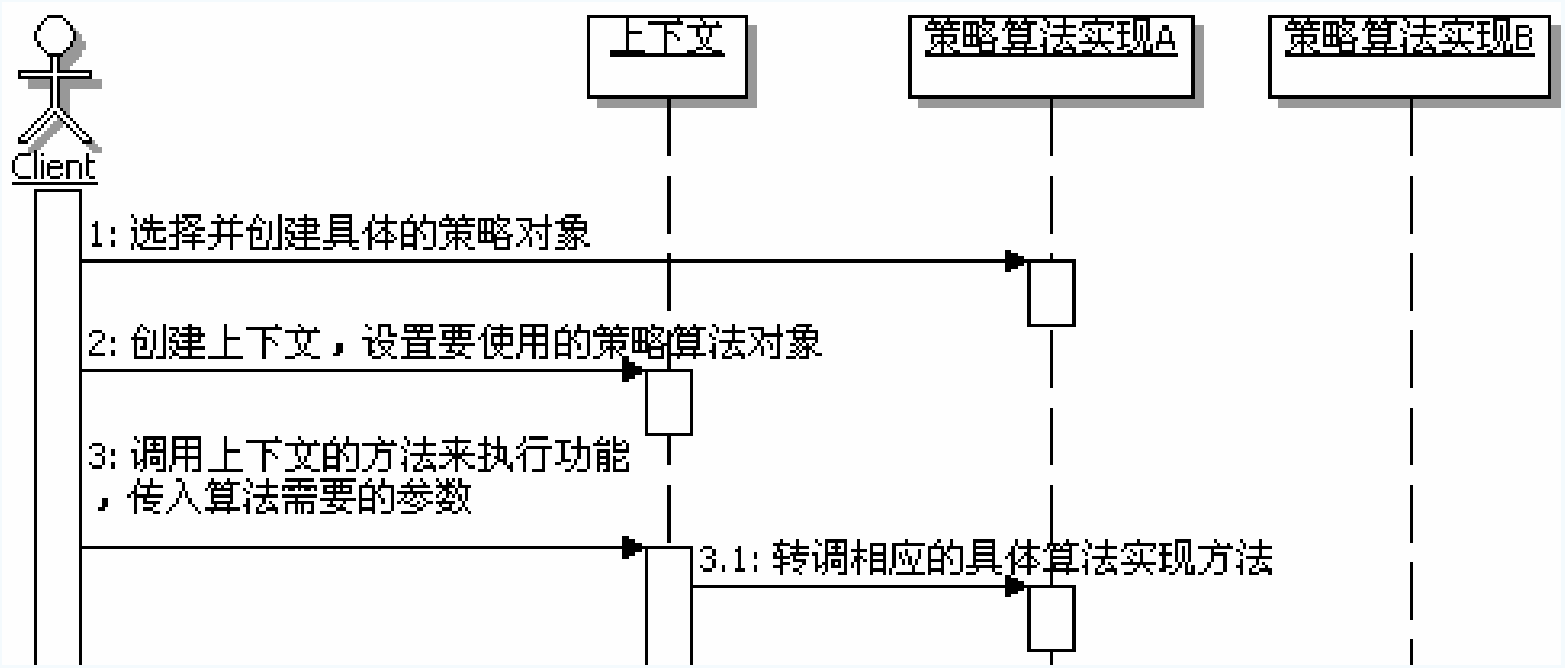
7：增加新的策略

在前面的示例里面，体会到了策略模式中切换算法的方便，但是增加一个新的算法会怎样呢？比如现在要实现如下的功能：对于公司的“战略合作客户”，统一8折。

其实很简单，策略模式可以让你很灵活的扩展新的算法。具体的做法是：先写一个策略算法类来实现新的要求，然后在客户端使用的时候指定使用新的策略算法类就可以了。

理解策略模式

- 策略模式调用顺序示意图



理解策略模式

- Context和Strategy的关系

在策略模式中，通常是上下文使用具体的策略实现对象，反过来，策略实现对象也可以从上下文获取所需要的数据，因此可以将上下文当参数传递给策略实现对象，这种情况下上下文和策略实现对象是紧密耦合的。

在这种情况下，上下文封装着具体策略对象进行算法运算所需要的数据，具体策略对象通过回调上下文的方法来获取这些数据。

甚至在某些情况下，策略实现对象还可以回调上下文的方法来实现一定的功能，这种使用场景下，上下文变相充当了多个策略算法实现的公共接口，在上下文定义的方法可以当做是所有或者是部分策略算法使用的公共功能。

但是请注意，由于所有的策略实现对象都实现同一个策略接口，传入同一个上下文，可能会造成传入的上下文数据的浪费，因为有的算法会使用这些数据，而有的算法不会使用，但是上下文和策略对象之间交互的开销是存在的了。

理解策略模式

- 工资支付的实现思路

随着公司的发展，会不断有新的工资支付方式出现，这就要求能方便的扩展；另外工资支付方式不是固定的，要求能很方便的切换具体的支付方式。

要实现这样的功能，策略模式是一个很好的选择。在实现这个功能的时候，不同的策略算法需要的数据是不一样，比如：现金支付就不需要银行帐号，而银行转账就需要帐号。这就导致在设计策略接口中的方法时，不太好确定参数的个数，而且，就算现在把所有的参数都列上了，今后扩展呢？难道再来修改策略接口吗？如果这样做，那无异于一场灾难，加入一个新策略，就需要修改接口，然后修改所有已有的实现，不疯掉才怪！那么到底如何实现，在今后扩展的时候才最方便呢？

解决方案之一，就是把上下文当做参数传递给策略对象，这样一来，如果要扩展新的策略实现，只需要扩展上下文就可以了，已有的实现不需要做任何的修改。

理解策略模式

现在有这么两种扩展的实现方式，到底使用哪一种呢？或者是哪种实现更好呢？下面来比较一下：

对于扩展上下文的方式：这样实现，所有策略的实现风格更统一，策略需要的数据都统一从上下文来获取，这样在使用方法上也很统一；另外，在上下文中添加新的数据，别的相应算法也可以用得上，可以视为公共的数据。但缺点也很明显，如果这些数据只有一个特定的算法来使用，那么这些数据有些浪费；另外每次添加新的算法都去扩展上下文，容易形成复杂的上下文对象层次，也未见得有必要。

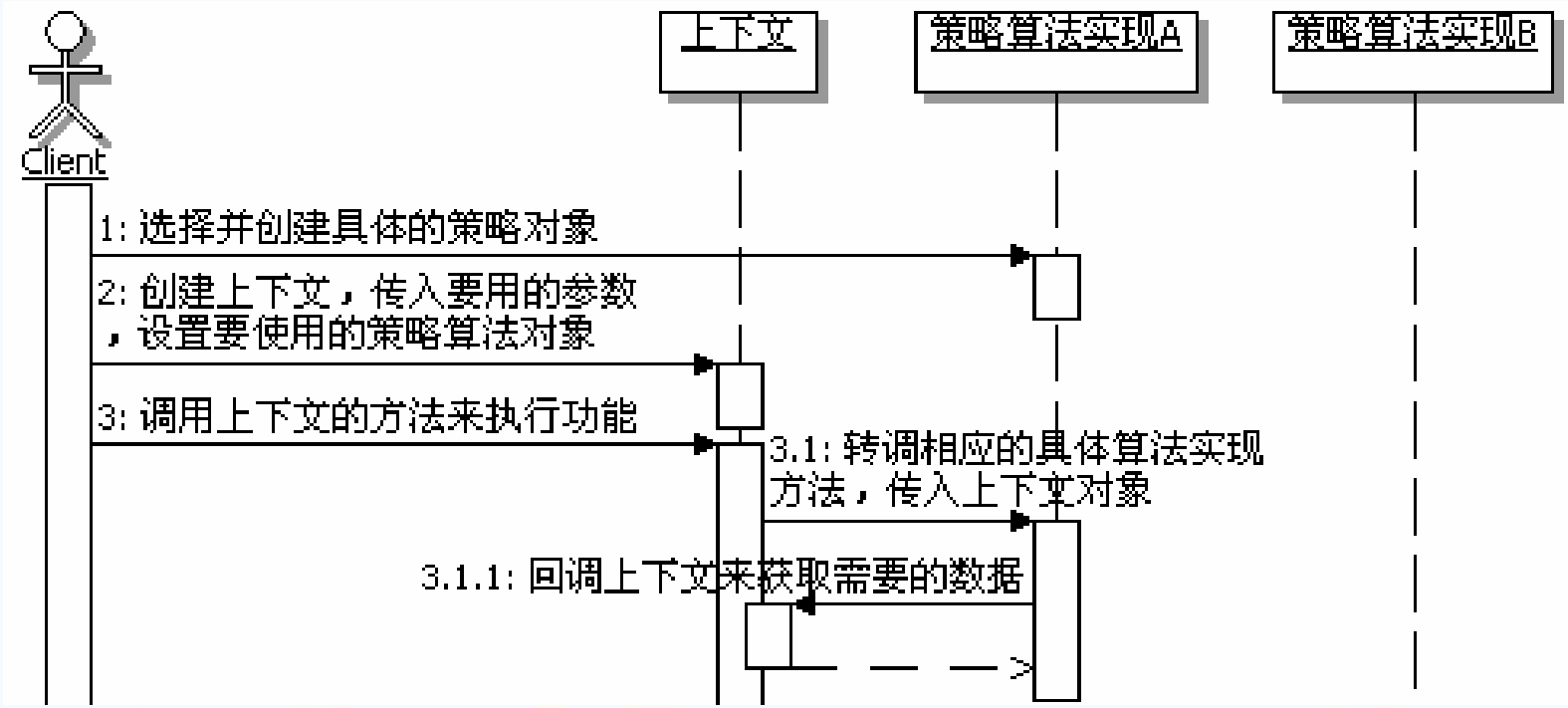
对于在策略算法的实现上添加自己需要的数据的方式：这样实现，比较好想，实现简单。但是缺点也很明显，跟其它策略实现的风格不一致，其它策略都是从上下文中来获取数据，而这个策略的实现一部分数据来自上下文，一部分数据来自自己，有些不统一；另外，这样一来，外部使用这些策略算法的时候也不一样了，不太好以一个统一的方式来动态切换策略算法。

两种实现各有优劣，至于如何选择，那就具体问题，具体的分析了。

理解策略模式

- 另一种策略模式调用顺序示意图

策略模式调用还有一种情况，就是把Context当做参数来传递给Strategy，也就是本例示范的这种方式，这个时候策略模式的调用顺序如图



理解策略模式

- 容错恢复机制

容错恢复机制是应用程序开发中非常常见的功能。什么是容错恢复呢？简单点说就是：程序运行的时候，正常情况下应该按照某种方式来做，如果按照某种方式来做发生错误的话，系统并不会崩溃，也不会就此不能继续向下运行了，而是有容忍出错的能力，不但能容忍程序运行出现错误，还提供出现错误后的备用方案，也就是恢复机制，来代替正常执行的功能，使程序继续向下运行。

举个实际点的例子吧，比如在一个系统中，所有对系统的操作都要有日志记录，而且这个日志还需要有管理界面，这种情况下通常会把日志记录在数据库里面，方便后续的管理，但是在记录日志到数据库的时候，可能会发生错误，比如暂时连不上数据库了，那就先记录在文件里面，然后在合适的时候把文件中的记录再转录到数据库中。

对于这样的功能，就可以采用策略模式，把日志记录到数据库和日志记录到文件当作两种记录日志的策略，然后在运行期间根据需要进行动态的切换

理解策略模式

- 策略模式结合模板方法模式

在实际应用策略模式的过程中，经常会出现一系列算法的实现上存在公共功能，甚至这一系列算法的实现步骤都是一样的，只是在某些局部步骤上有所不同，这个时候，就需要对策略模式进行些许的变化使用了。

对于一系列算法的实现上存在公共功能的情况，策略模式可以有如下三种实现方式：

1：在上下文当中实现公共功能，让所有具体的策略算法回调这些方法。

2：把策略的接口改成抽象类，然后在里面实现具体算法的公共功能。

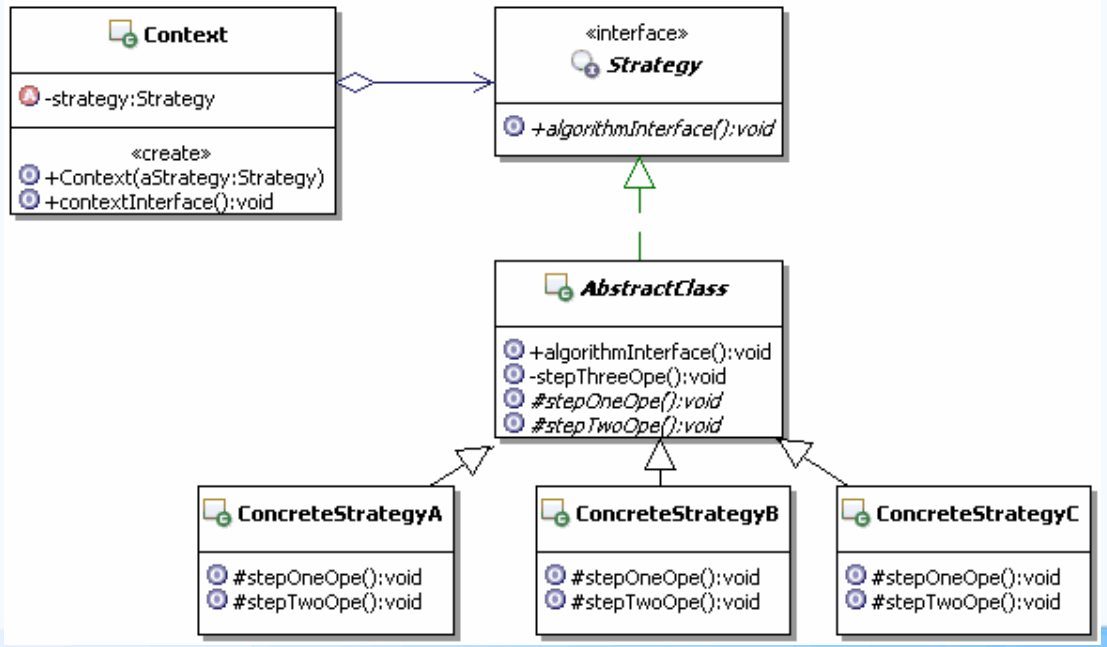
3：给所有的策略算法定义一个抽象的父类，让这个父类去实现策略的接口，然后在这个父类里面去实现公共的功能。

更进一步，如果这个时候发现“一系列算法的实现步骤都是一样的，只是在某些局部步骤上有所不同”的情况，那就可以在这个抽象类里面定义算法实现的骨架，然后让具体的策略算法去实现变化的部分。这样的一个结构自然就变成了策略模式来结合模板方法模式了，那个抽象类就成了模板方法模式的模板类。

理解策略模式

在前面我们讨论过模板方法模式来结合策略模式的方式，也就是主要的结构是模板方法模式，局部采用策略模式。而这里讨论的是策略模式来结合模板方法模式，也就是主要的结构是策略模式，局部实现上采用模板方法模式。通过这个示例也可以看出来，模式之间的结合是没有定势的，要具体问题具体分析。

此时策略模式结合模板方法模式的系统结构如下图



理解策略模式

- 策略模式的优缺点
  1：定义一系列算法
  2：避免多重条件语句
  3：更好的扩展性
  4：客户必须了解每种策略的不同
  5：增加了对象数目
  6：只适合扁平的算法结构

思考策略模式

- 策略模式的本质

  策略模式的本质是：**分离算法，选择实现**

- 对设计原则的体现

  从设计原则上来看，策略模式很好的体现了开-闭原则。策略模式通过把一系列可变的算法进行封装，并定义出合理的使用结构，使得在系统出现新算法的时候，能很容易的把新的算法加入到已有的系统中，而已有的实现不需要做任何修改。这在前面的示例中已经体现出来了，好好体会一下。

  从设计原则上来看，策略模式还很好的体现了里氏替换原则。策略模式是一个扁平结构，一系列的实现算法其实是兄弟关系，都是实现同一个接口或者继承的同一个父类。这样只要使用策略的客户保持面向抽象类型编程，就能够使用不同的策略的具体实现对象来配置它，从而实现一系列算法可以相互替换。

思考策略模式

- 何时选用策略模式

1：出现有许多相关的类，仅仅是行为有差别的情况，可以使用策略模式来使用多个行为中的一个来配置一个类的方法，实现算法动态切换

2：出现同一个算法，有很多不同的实现的情况，可以使用策略模式来把这些“不同的实现”实现成为一个算法的类层次

3：需要封装算法中，与算法相关的数据的情况，可以使用策略模式来避免暴露这些跟算法相关的数据结构

4：出现抽象一个定义了很多行为的类，并且是通过多个if-else语句来选择这些行为的情况，可以使用策略模式来代替这些条件语句