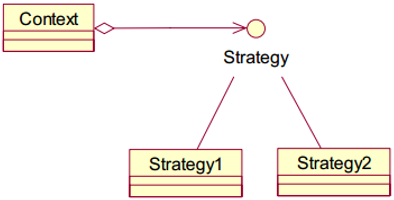
* 设计模式 – 策略模式（strategy）：

针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。

策略模式使得算法可以在不影响到客户端的情况下发生变化。策略模式把行为和环境分开。环境类负责维持和查询行为类，各种算法在具体的策略类中提供。

由于算法和环境独立开来，算法的增减，修改都不会影响到环境和客户端。



**策略模式角色：**

1) **算法使用环境(Context)角色**：算法被引用到这里和一些其它的与环境有关的操作一起来完成任务。

2) **抽象策略(Strategy)角色**：规定了所有具体策略角色所需的接口。在java它通常由接口或者抽象类来实现。

3) **具体策略(Concrete Strategy)角色**：实现了抽象策略角色定义的接口。

* 设计模式 – **策略模式应用场景：**

1、 多个类只区别在表现行为（算法）不同，可以使用Strategy模式，在运行时动态选择具体要执行的行为。

2、 需要在不同情况下使用不同的策略(算法)，或者策略还可能在未来用其它方式来实现。

3、 对客户隐藏具体策略(算法)的实现细节，彼此完全独立。

* 设计模式 – **策略模式优点**：

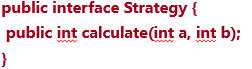
1、  提供了一种替代继承的方法，而且既保持了继承的优点(代码重用)还比继承更灵活(算法独立，可以任意扩展)。

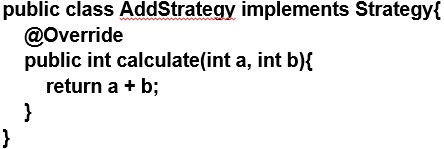
2、  避免程序中使用多重条件转移语句，使系统更灵活，并易于扩展。

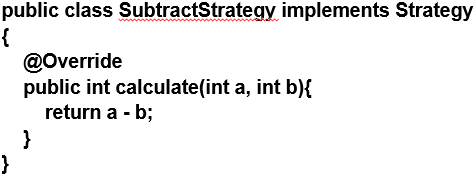
* 设计模式 – **策略模式缺点**：

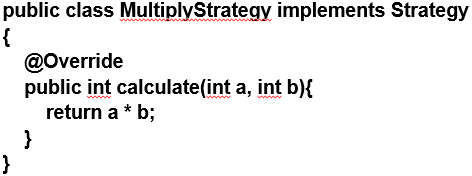
客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道所有的算法或行为的情况。

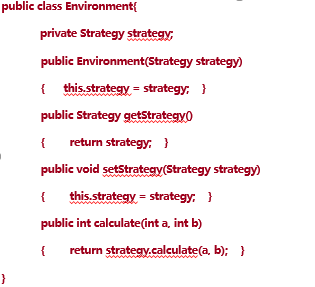
* 设计模式 – **策略模式实例：两个数字的加减乘除运算**

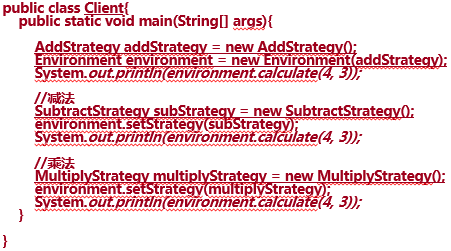












比如，可排序的Set 即TreeSet，它通过实现Compareble接口这个规则，重写其 中的compareTo()方法，就是一种策略模式。