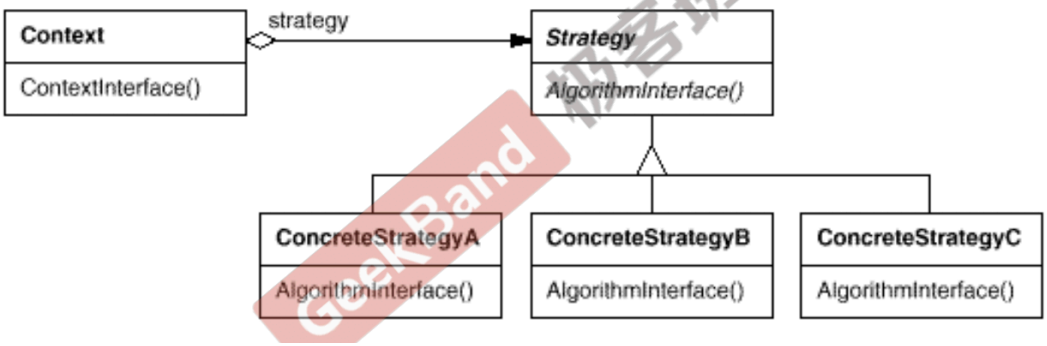
定义:一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换(变化)。策略模式让算法可独立于使用它的客户程序(稳定)而独立变化(扩展,子类化)。

要点总结:Strategy及其子类为组件提供了一系列可重用的算法，从而可以使得类型在运行时方便地根据需要在各个算法之间进行切换。Strategy模式提供了用条件判断语句以外的另一种选择，消除条件判断语句，就是在解耦合。含有许多条件判断语句的代码通常都需要Strategy模式。如果Strategy对象没有实例变量，那么各个上下文可以共享同一个Strategy对象，从而节省对象开销。



角色:

抽象策略角色（Strategy）： 抽象策略类。

具体策略角色(ConcreteStrategy)：封装了继续相关的算法和行为。

环境角色(Context)：持有一个策略类的引用，最终给客户端调用。

优点:

1、 使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护。

2、 策略模式让你可以动态的改变对象的行为，动态修改策略

缺点:

1、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。

2、类过多---策略模式造成很多的策略类，每个具体策略类都会产生

一个新类。

意图：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。

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

优点： 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。

缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

使用场景： 1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。 2、一个系统需要动态地在几种算法中选择一种。 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

注意事项：如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。