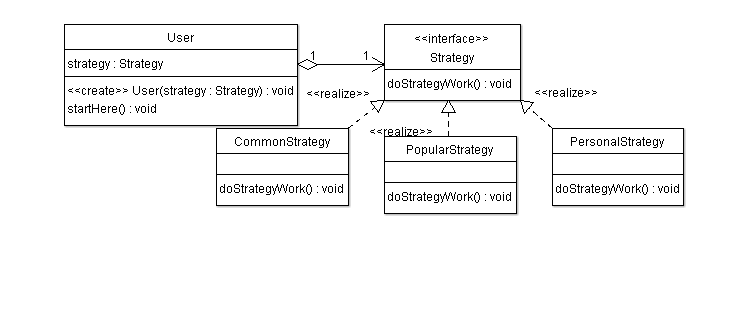
策略模式简介

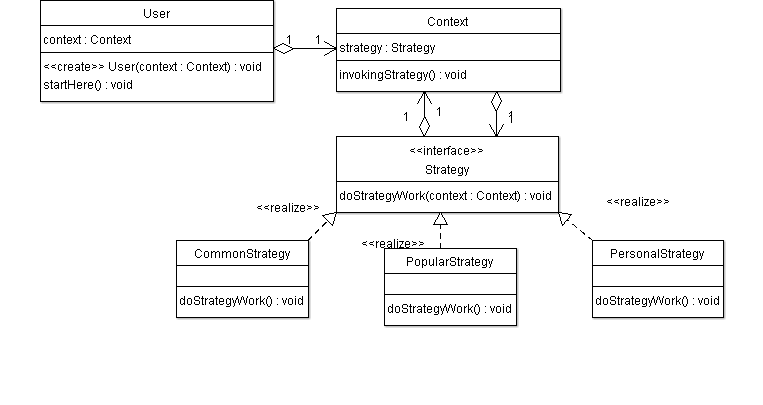
         策略模式一直程序开发中，最常用的模式之一；它的功能就是定义了一系列的算法，这些算法定义着公共的接口，所以它们之间可以相互替换。这使得我们在开发过程中，若有新的策略需要扩展时，程序变的很容易开发。下面是策略模式的结构示意图：



         从结构示意图中，我们能清楚的感受到，策略模式的巧妙之处就是将变化的东西（这里我们称之为算法）通过定义一个相同的公共的接口封装且隔离；以不同的策略实现相同的接口，使程序能够完成各种任务的同时，让上层的编程针对接口，实现不变，也不关心其调用的是哪种策略。

上下文的引入

         然而在实际开发中，我们的需求往往不会像结构示意图那样，如此简单、如此simple。实际较为常见的情况就是：上层的调用需要与接口之间有一定的交互。交互的可能是一些属性，或是一些方法。这样的交互往往会让接口变的难以调用；于是上下文的引入就是势在必行。将相关的属性或一些公共的方法封装到上下文中，让上下文去和接口进行复杂的交互。而上层的调用只需要跟上下文打交道就可以。下面有包含上下文的策略模式结构示意图：



         仔细看看两个结构示意图的变化，其实不外乎将变的东西和不变的东西进行封装与隔离，这样不变的东西能够尽可能的减小其变化，而变的东西更容易修改或扩充；其实这也正是软件设计的基本原则。

         透过示意图，我们上下文对象，将自己传递给接口，方便接口调取它的属性与方法，以完成交互。如此将前面提到的交互的行为提取出来，封装在自己内部，对上层提供统一调用方法。也正是将变化与不变封装且隔离。

延迟设计与内部类

         正如前面提到的，上下文中封装了上层调用类与接口间的交互；但这些交互有个前提，那就是它们能被抽象出来。另一些交互在我们的业务需求中无法被抽象出来。换句话说，这些交互，就是要延迟到上层调用内中来设计与实现的。在这种情况下：上层调用类与接口的某个策略的实现往往是紧耦合的、并且它们是一对一。

         要完成它们之间的这种交互，设计它们间通讯的方法，会让代码变杂乱，同时也增加了调用接口的难度。

         这时内部类的优势可以在此发挥：我们将接口的实现类放到上层调用类中，变成上层调用类的一个内部类；由于内部类可以任意调用其外部类的属性和方法的特性。它们能直接的无阻碍的通讯，完全不同设计多余的接口方法来完成交互；如此一来也正好符合它们一对一的关系。下面是我简单写的一个小Demo：

 策略接口：

/\*\*

\* 策略算法接口

\* [@author](http://my.oschina.net/arthor) Breath\_L

\*

\*/

public interface Strategy {

public void doStrategy(Context cont);

}

上下文类：

/\*\*

\* 上下文类

\* [@author](http://my.oschina.net/arthor) Breath\_L

\*

\*/

public class Context {

private String baseTxt;

private Strategy strategy;

public Context(String baseTxt, Strategy strategy){

this.baseTxt = baseTxt;

this.strategy = strategy;

}

/\*\*

\* 暴露给上层对象调用的方法

\*/

public void doStrategyNow(){

//公共逻辑...

System.out.println(baseTxt);

strategy.doStrategy(this);

//公共逻辑...

}

}

上层调用类：

public class TopGetter {

public Strategy in\_strategy;

public Context context;

public TopGetter(){

in\_strategy = new Strategy(){

@Override

public void doStrategy(Context cont) {

System.out.println(beDependent());

}

};

context = new Context("Hello World !", in\_strategy);

}

private String beDependent(){

return "We will rock you !";

}

public void startHere(){

context.doStrategyNow();

}

public static void main(String[] args){

TopGetter topGetter = new TopGetter();

topGetter.startHere();

}

}

         仔细思考上下文和内部类，它们算是策略模式在封装和隔离变与不变的东西时，的一个深入设计。万变不离其中的还是软件设计的基本原则。还是那句话：没有最完美的设计，只有最符合业务的实现。也许此刻的思考只是符合我最近的业务，不怕见笑，做一下记录与分享。