## 2018年真题

**1.请写出至少三个 OO 设计原则，并说明在状态模式中是如何体现的**。

符合单一指责原则、开闭原则、迪米特法则（最少知道法则）

一个类一个责任原则（SRP）

类应该对扩展开放，对修改关闭。（开闭原则）

最少知识原则，只和你的密友谈话（迪米特法则）

状态模式首先分析出有几个状态，然后提供一个状态接口，所有的状态都要实现这个接口，将每个状态的行为局部化到自己的类中，让每个状态“对修改关闭”，让context“对扩展开放”，将状态封装成独立的类，并将动作委托到代表当前状态的对象行为会随着内部状态而改变。

**2.求写出宏命令的概念，并用代码实现宏命令。**

宏命令：利用命令模式和命令数组来执行一堆命令而不是一个命令。



**6.什么是设计模式？ 什么是符合模式？**

**模式 是再某情境下，针对某问题的某种解决方案。**

**情景：不断出现的情况。**

**问题：某情境下达到的目标/约束。**

**解决方案：一个通用的设计，用来解决约束，达到目标。**

**复合模式结合两个或以上的模式，组成一个解决方案，解决一再发生的一般性问题。**

**8.MVC 如何体现设计模式？**

* **策略。**

视图和控制器实现了经典的策略模式：视图是一个对象，可以被调整使用不同的策略，而控制器提供了策略，视图只关心系统中的可视部分，对于任何界面行为，都委托给控制器处理。控制器—算法，

* **组合**

显示包含窗口、面板、按钮等。视图时GUI组件的组合顶层组件包含其他组件，直到叶子节点。

* **观察者**

模型是被观察者。只要模型的状态改变，注册到模型上的视图或者控制器都会接收到通知。

11.写一个双向适配器的代码，并画出类图。以 Cat 会捉老鼠，Dog 会 bark为例，实现 Cat bark，Dog catch。

## 真题：

1. **Please name at least three OO principles and explain how they are applied in the Strategy pattern. (15 marks)**

* **封装变化**：策略模式将代码中经常变化的部分封装成了一个一个算法类。
* **多用组合少用继承**：策略模式使用组合而不是继承来扩展算法，就是说如果增加算法，这需要增加算法类，而不是通过继承
* **针对接口编程，不针对实现编程（依赖倒转原则）：**策略模式使用策略的超类型，而不是具体的策略类
* **开闭原则**。策略模式对增加新的算法开放。
* **单一职责原则**：策略模式中的算法，只有一种功能就是算法。

1. **How are design pattern different from class libraies?**

答：库和框架提供了我们某些特定的实现，让我们可以轻易的引用，但是这并不算是设计模式，有些时候库和框架本身会用到设计模式；设计模式比库的等级更高，设计模式告诉我们如何组织类和对象以解决某种问题。

1. **Write a class “LowerCaseInputStream” Which is a Java I/O decorator that convert all uppercase characters to lowercase in the input stream. In other words, if we read in “I Know the Decorator Pattern threfore I RULE!” then your decorator converts this to “i know the decorator pattern threfore I rule!”**

public class LowerCaseInputStream extends FilterInputStream {  
 protected LowerCaseInputStream(InputStream in) {  
 super(in);  
 }  
 @Override  
 public int read() throws IOException {  
 int c = super.read();  
 return c == -1 ? c : Character.toLowerCase((char)c);  
 }  
 @Override  
 public int read(byte[] b, int offset, int len) throws IOException {  
 int result = super.read(b, offset, len);  
 for (int i = 0; i < b.length; i++) {  
 b[i] = (byte) Character.toLowerCase(b[i]);  
 }  
 return result;  
 }  
 public static void main(String[] args) throws IOException {  
 LowerCaseInputStream a = new LowerCaseInputStream(System.*in*);  
 int c;  
 while ((c = a.read()) >= 0){  
 System.*out*.print((char) c);  
 }  
 a.close();  
 }  
}

1. **What is a MacroCommand in the Command Pattern? Please write codes for a general MacroCommand class.**



### 适配器VS装饰者VS外观

* 适配器将一个对象包装起来以改变其接口
* 装饰者将一个对象包装起来以增加新的行为和责任
* 外观将一群对象包装起来以简化其接口

### 外观与适配器的差异？

外观和适配器可以包装许多类，但是外观的意图是简化接口,让一个子系统更加容易使用，而适配器的意图是将接口转换成不同接口。

### 对比简单工厂和工厂模式

简单工厂把全部事情在一个地方都处理完了，然而工厂方法确实创建一个框架，让子类决定要如何实现。简单工厂只是将对象的创建封装起来，不具备工厂方法的弹性，不能变更正在创建的产品。

### 对比策略模式和状态模式

状态模式和策略模式的类图一摸一样。

区别：

状态模式将一群行为封装在状态对象中，context的行为可以随时委托到状态中的一个。随着时间的流逝，当前的状态在**状态集合**中游走变化，以反映context内部的状态。

策略模式，客户通常主动指定context所要组合的策略对象是哪一个。虽然策略模式让我们更加有弹性，能够在运行时改变策略，但是对于某个context对象来说，通常**只有一个最恰当的策略对象**。

把策略模式当成除了继承以外的方案，可以通过组合不同的对象改变类的行为。

把状态模式当成不用再context中放置许多条件判断的代替方案。将行为包装进状态的对象中，可以通过在context内简单的改变状态对象来改变context的行为。

## OO原则

* 封装变化；
* 多用组合少用继承；（合成复用原则）
* 针对接口编程，不针对实现编程；（依赖倒转原则）
* 为交互对象之间松耦合设计而努力；
* 类应该对扩展开放，对修改关闭；（开闭原则）
* 依赖抽象，不是依赖具体类；
* 只和朋友交谈；（迪米特法则）
* **别来找我，我会找你**；在这个原则之下，允许低层组件将自己挂钩到系统之上，但是高层组件会决定什么时候和怎样使用这些底层组件。（**工厂方法、观察者**、**模板方法**）都采用了**好莱坞原则**）
* 类应该只有一个改变的理由。（单一职责原则）

## 软件设计原则

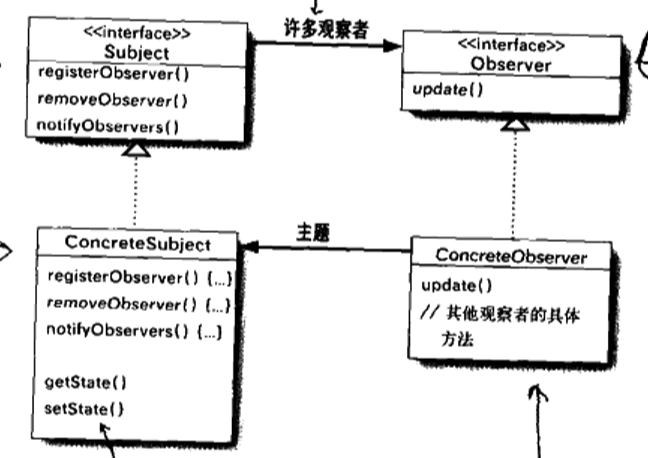
* **单一职责原则**（Single responsibility principle）： 一个类，只有一个引起它变化的原因，只有一个职责。
* **开闭原则**： 一个软件实体如类、 模块和函数应该对扩展开放，对修改关闭。模块应尽量在不修改原（是“原”，指原来的代码）代码的情况下进行扩展。
* **里氏替换原则**（ Liskov Substitution Principle）： 任何基类可以出现的地方，子类一定可以出现。 LSP 是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。
* **依赖倒置原则**（ Dependency Inversion Principle）： 程序要依赖于抽象接口，不要依赖于具体实现。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。 面向接口编程，而不是面向实现编程。 即要面向抽象编程，而不是面向具体编程。高层模块依赖高层模块，细节依赖抽象。
* **接口隔离原则**： 客户端不应该依赖它不需要的接口， 类间的依赖关系应该建立在最小的接口上。
* **合成复用原则**（Composite/Aggregate Reuse Principle）： 合成/聚合复用原则就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；新的对象通过向这些对象的委派达到复用已有功能的目的。它的设计原则是： 要尽量使用合成/聚合，尽量不要使用继承。
* **迪米特法则**（ Law of Demeter、 Least Knowledge Principle）： 一个对象应当对其他对象有尽可能少的了解， 不和陌生人说话。

## 观察者模式（组合）

类似于出版者（主题）+订阅者（观察者）=观察者模式

**定义**：定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

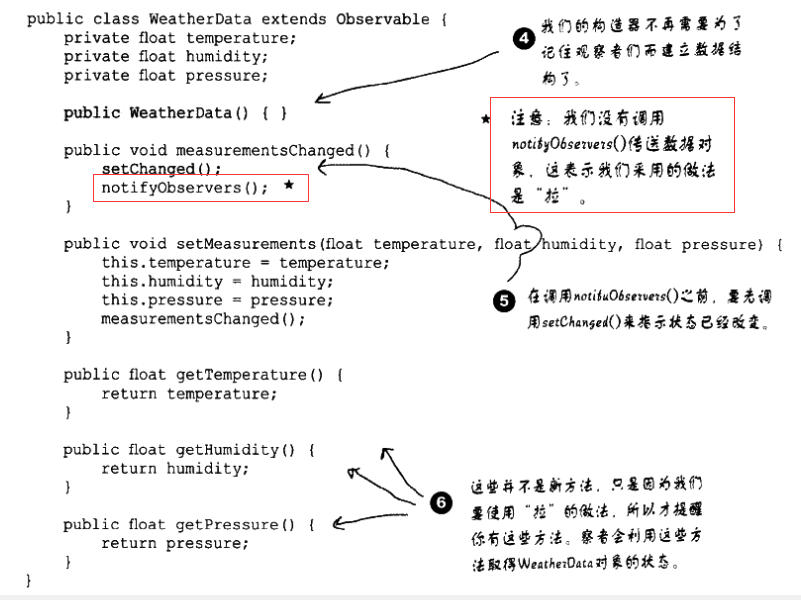
类图：

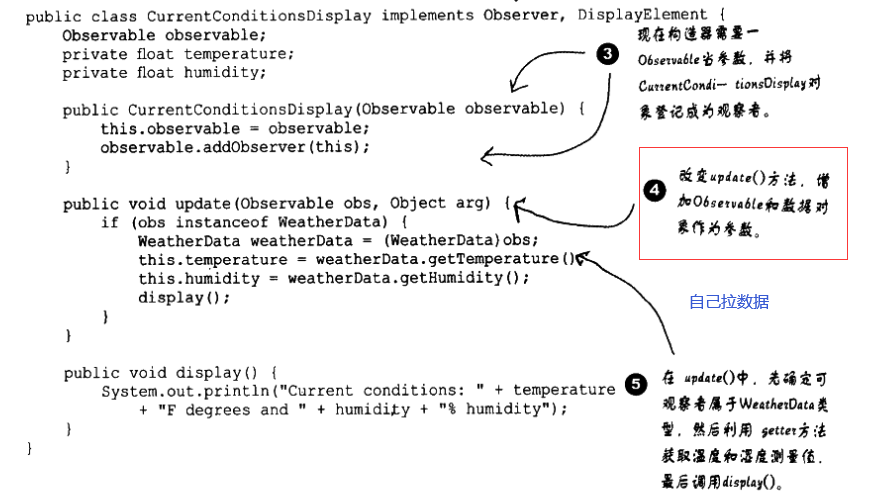


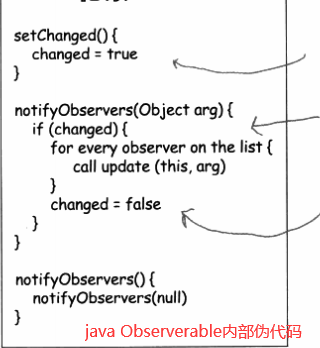
注意：

* ConcreateObserver需要保存 一个对主题的引用，这样以后如果想取消注册的时候会比较方便。
* 使用该模式可以从被观察者出推（push）或（pull）数据（然而，推的方式被认为更正确）。

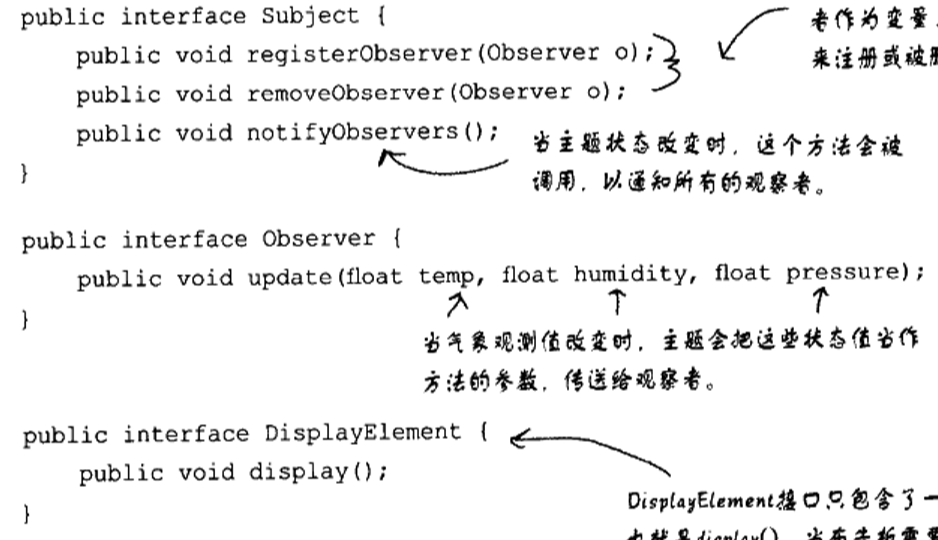
拉模式举例：



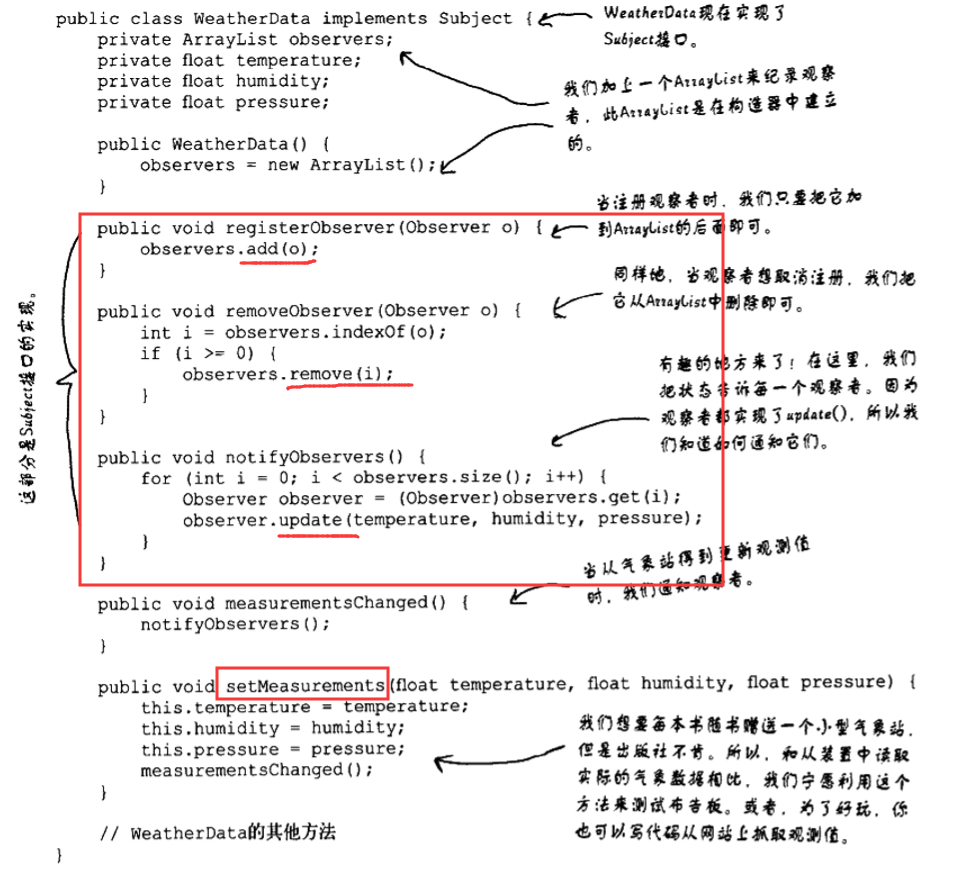




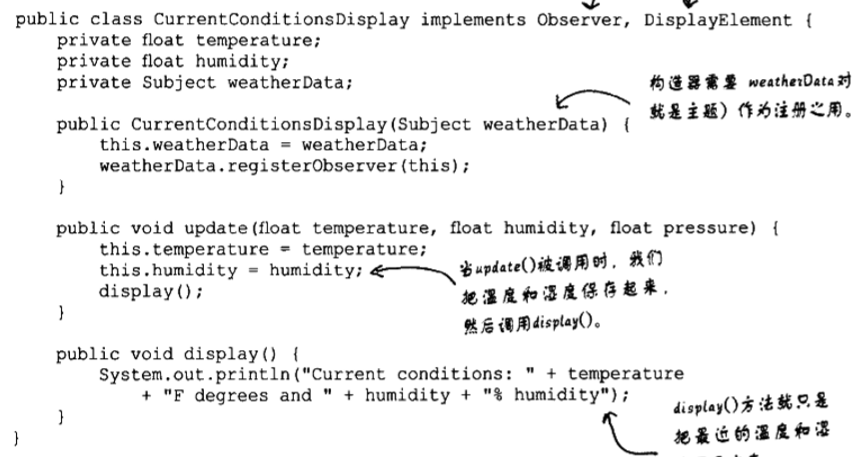
推模式举例：



2）主题：



3）布告板：



简单工厂方法、工厂模式和抽象工厂模式

New代表要实例化对象，就意味着要针对实现编程，但是我们需要针对接口编程，对于总要创建对象的情况，该怎么办？

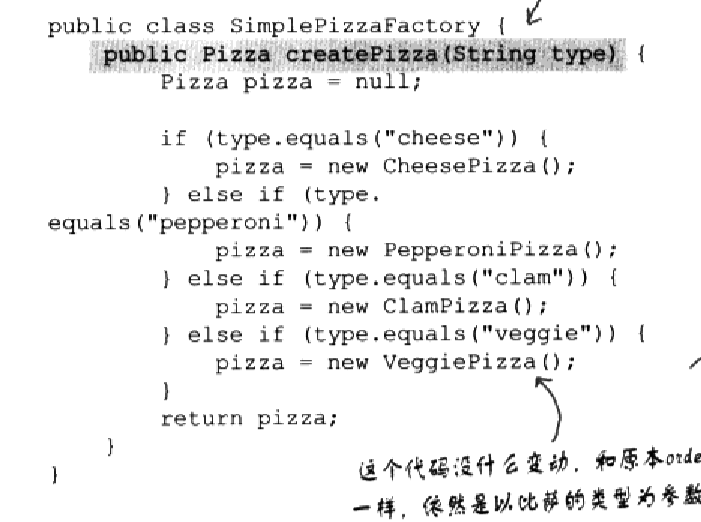
将变化封装成为工厂。

## 简单工厂方法

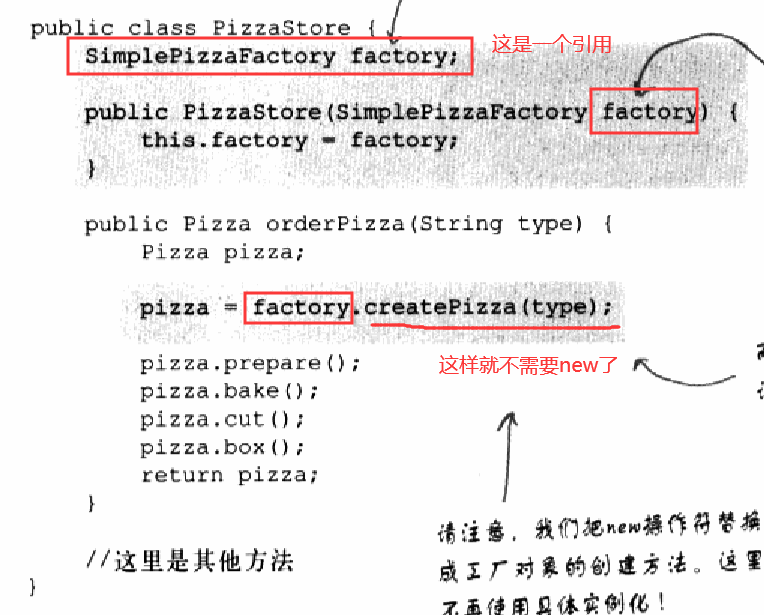
这是一种方法而不是模式，它利用了静态方法，所以又叫作静态工厂，不需要使用创建对象的方法来实例化对象，它将很多变化封装在一起成为一个静态类。但是缺点是不能通过继承来改变创建方法的行为。

例子：披萨工厂需要new很多新种类的披萨

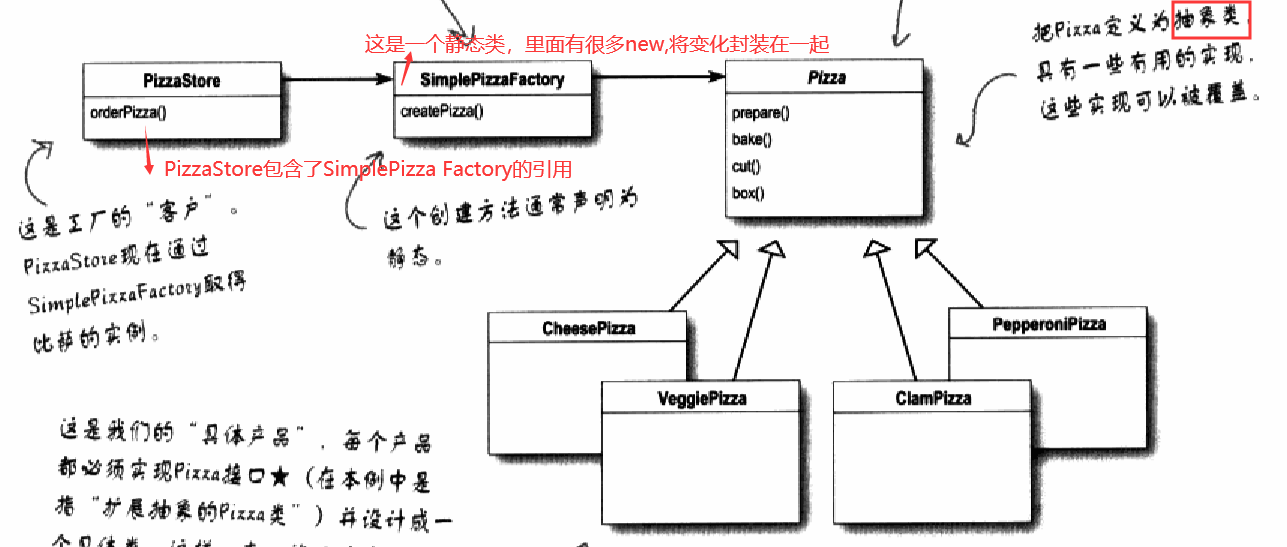
1）建一个简单披萨工厂：将所有的new,放在这个里面，将变化封装起来。下面点单时需要new的时候直接调用createPizza(type)就可以。



2）重写PazzaStore类



简单工厂类图：



## 工厂模式

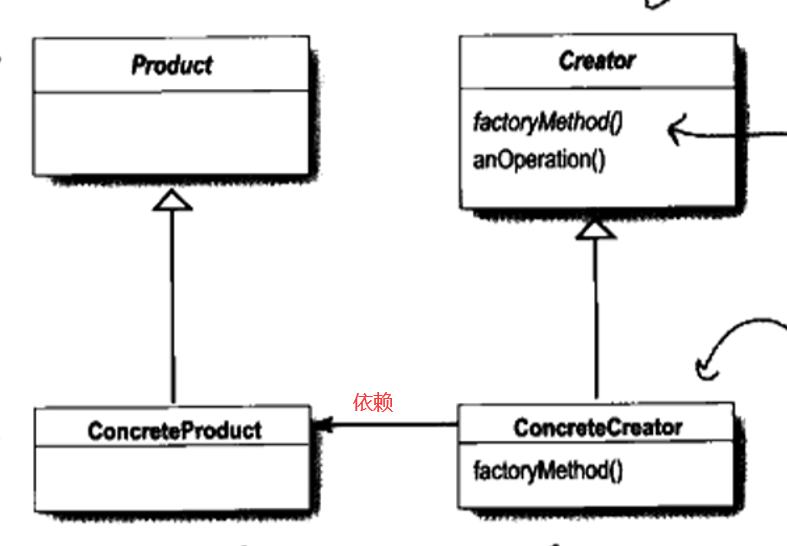
**定义：**定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。把实例化推迟到子类中。

工厂方法模式，是对简单工厂模式的扩展，其在简单工厂基础上对工厂类进行抽象，使得工厂类可以被扩展，实现了工厂类的“开闭原则”。在工厂方法模式中，创建具体产品的过程交由具体工厂实现，一种具体工厂只负责生产一种具体产品，也符合“单一职责原则”。具体产品类与客户端的依赖减少，使得客户端不再依赖具体产品，而依赖抽象产品接口，减少系统依赖，符合“迪米特法则”，由于在客户端中只需要依赖抽象，则符合了“里氏替换原则”，由于抽象工厂只依赖抽象产品，符合了“依赖倒置原则”，高层模块只依赖抽象，细节依赖抽象。

**缺点：**

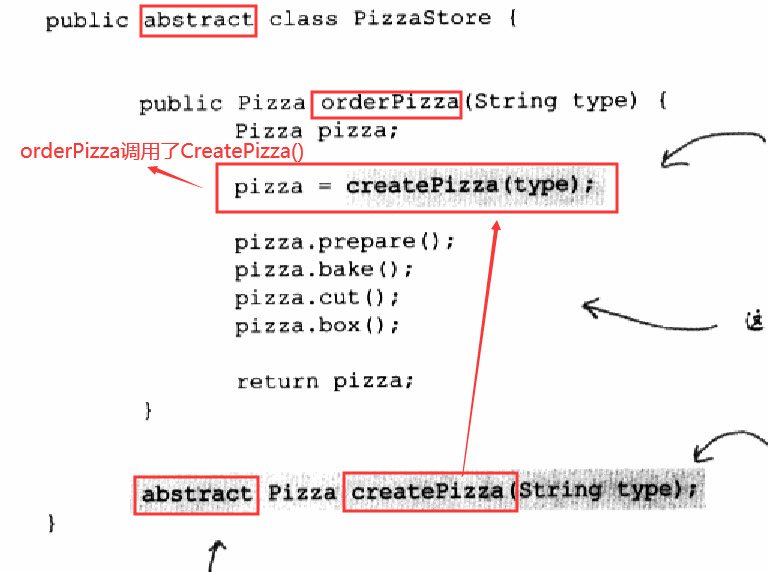
当系统产品庞大时，会产生很多产品类和工厂类，增加系统复杂程度；增加了一层抽象层，增加了系统抽象程度和实现难度。当产品和工厂减少时，模式将退化成简单工厂模式

类图：

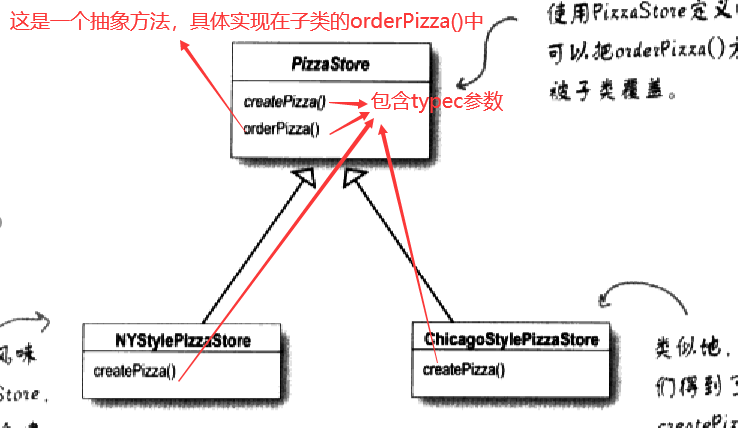


**例子：**加盟披萨店，需要有各地的风味披萨

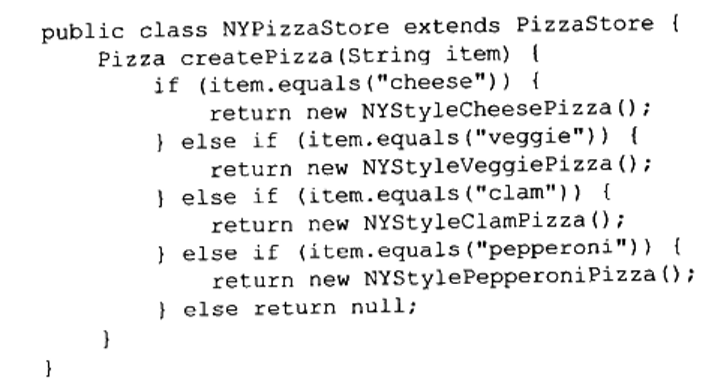
代码：在抽象工厂类中，抽象方法CreatePizza(type)是空的,具体的实现在各地的工厂中。



类图：



子类中的具体实现：



## 抽象工厂模式

**定义：**提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。

**设计原则6：**要依赖抽象，不要依赖具体类。（依赖倒置原则）DIP

底层的实现依赖高层的抽象

优点：

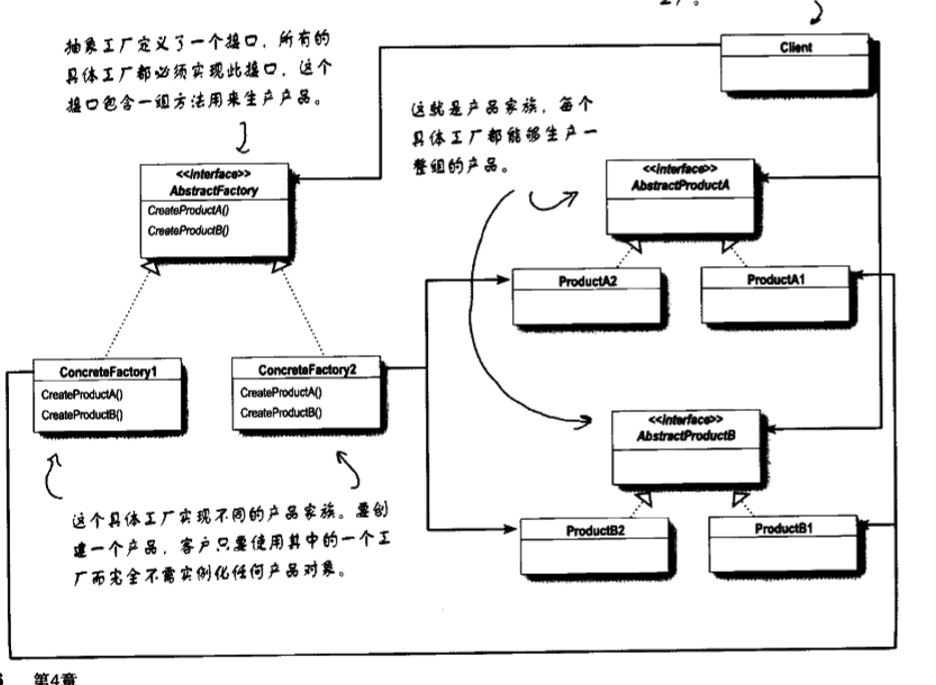
隔离了具体类的生产过程，客户不知道需要创建哪些对象（减少了客户端对产品的依赖）；能够保证相互依赖的一组对象始终属于同一个产品族；增加新的产品和工厂很简单，符合“开闭原则”。

缺点：

当需要新增产品的产品族结构时，将需要修改所有工厂类，破坏了“开闭原则”。

在抽象工厂模式中，产品族的概念是使用抽象工厂的关键，产品族是指一个对象需要依赖一组相关的其他对象。

类图：



## 对比工厂和抽象工厂：

1）都负责创建对象，将客户从具体类型中解耦，提高弹性。

2）工厂用的是继承，通过子类创建对象，子类决定具体类型，但是只能创建一个产品。

3）抽象工厂用的是组合，可以创建产品家族或将一群相关产品集合起来，但是有新产品把必须修改接口，其中的具体工厂会使用工厂方法创建产品。

## 单件模式

当一个类只能有一个实例时要用单件模式

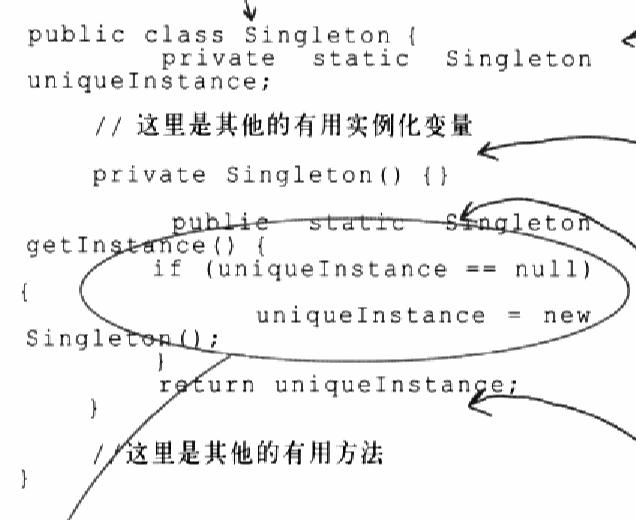
**代码**：要是私有的，静态的唯一实例，

优点：

隐藏了其他对象实例化单例对象的入口，保证整个系统始终只有一个对象提供服务；

缺点：

没有抽象层，使得单例的扩展性不强；在单例对象中提供了过多的职责，破坏了“单一职责”原则”

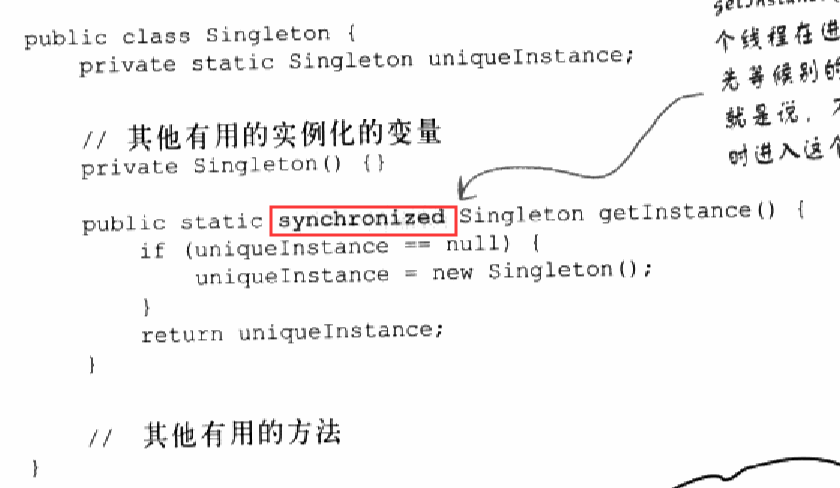


**定义：**确保一个类只有一个实例，并提供一个全局访问点。

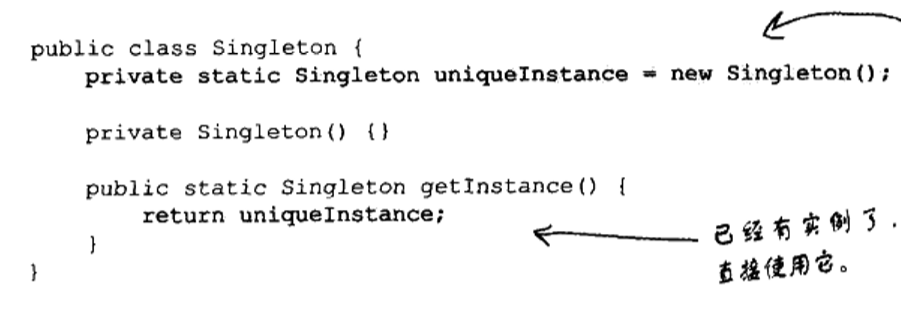
可以延迟实例化

## 用单件模式处理多线程

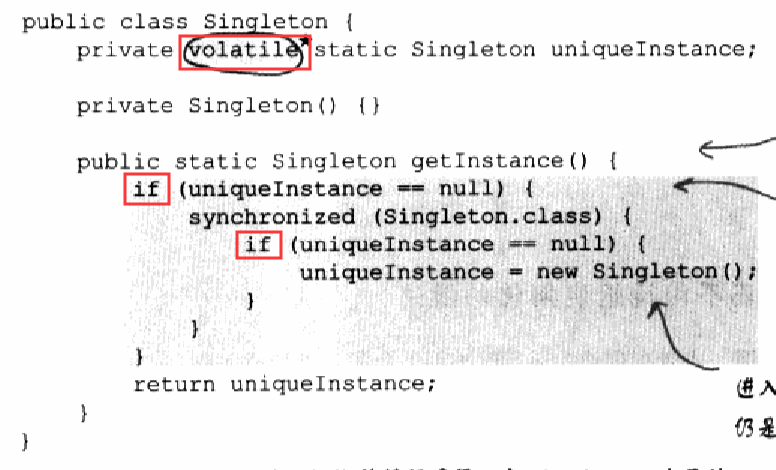
方法一：加上synchronized，但会降低性能



方法二：急切创建实例



方法三：双重检查加锁



**设计原则7：**一个类一个责任原则（SRP）

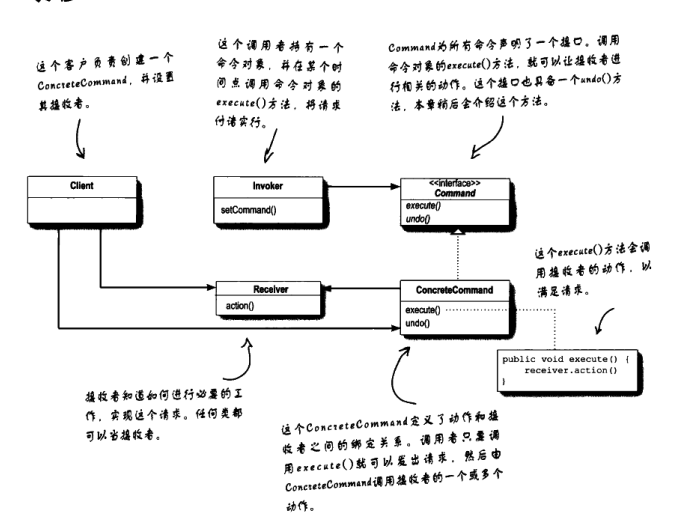
单件模式可以被视为有两个责任，违反了SRP原则

## 对比单件模式和全局变量：

1）单件模式确保了是有一个实例并且提供全局访问

2）全局变量可以提供全局访问却不能确保只有一个实例，并且全局变量需要急切实例化，会浪费资源，用全局变量指向小对象会造成命名空间的污染。

## 命令模式

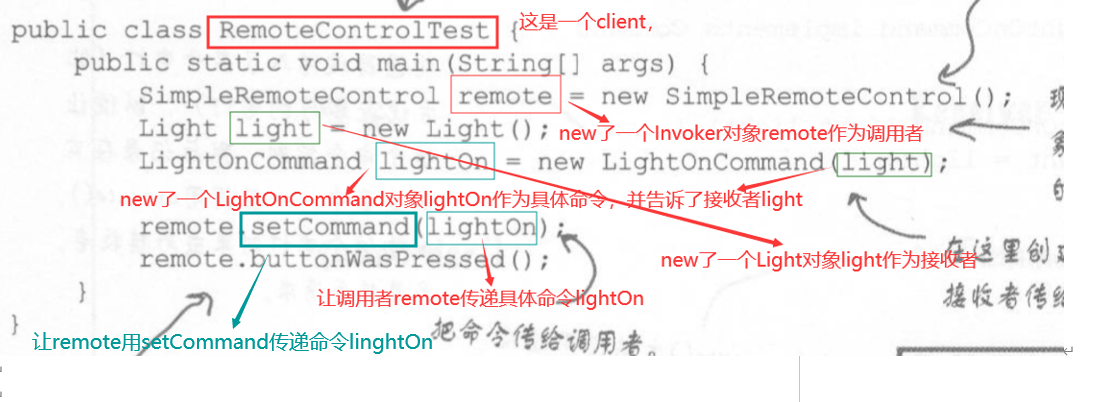


例子：设计遥控器API

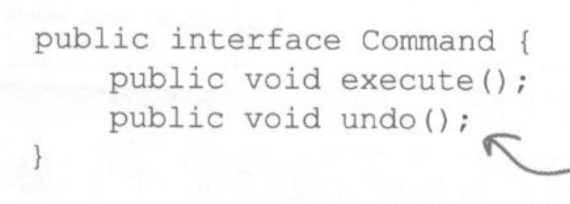
**定义：**将“请求”封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。命令模式也支持可撤销的操作。把方法调用封装起来，将动作的请求者从动作的执行者对象中解耦。

类图：

Client实例代码：

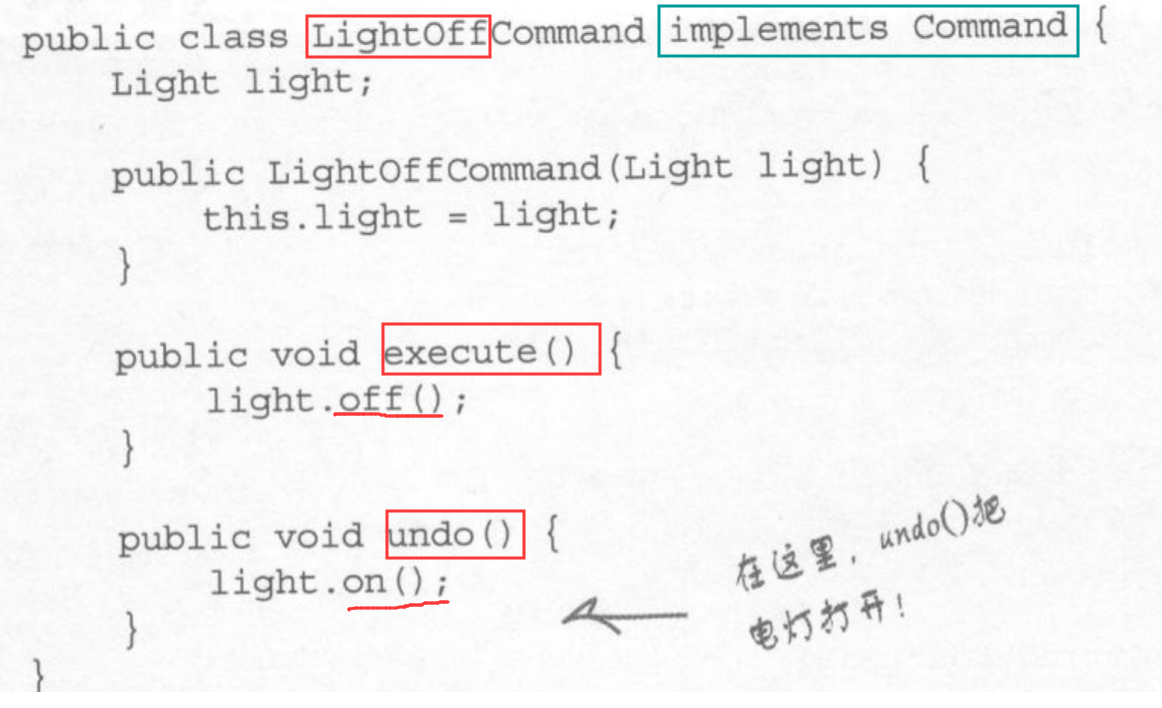


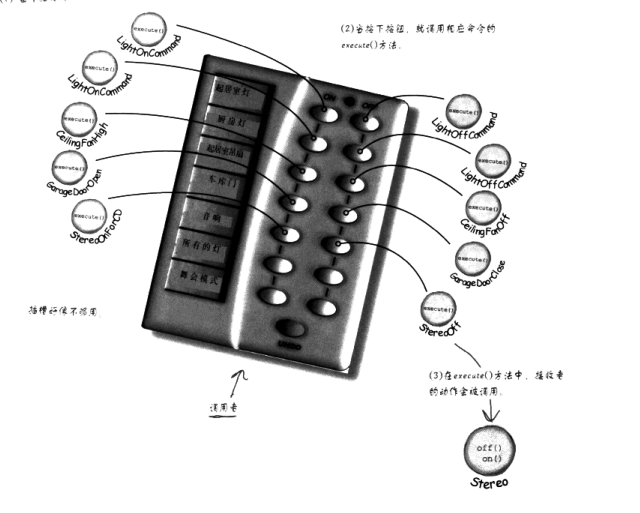
命令接口代码：



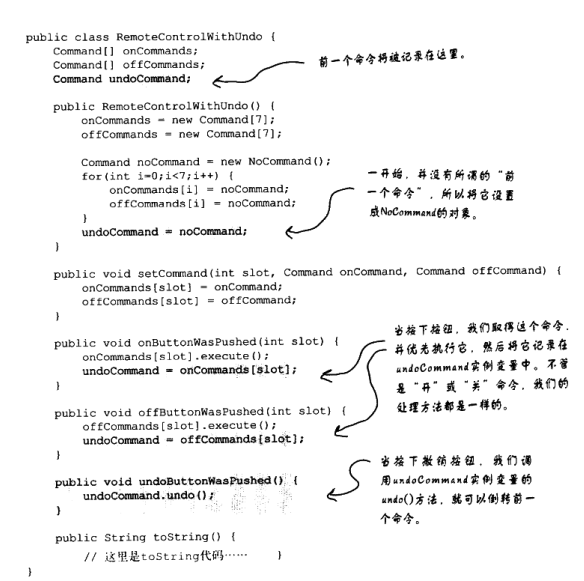
具体命令代码：重写两个方法，开灯和关灯的两个方法内容相反







遥控器这个调用者的代码



请注意上面的NoCommand，使用newCommand可以是我们免于判断某个插槽是否为null。

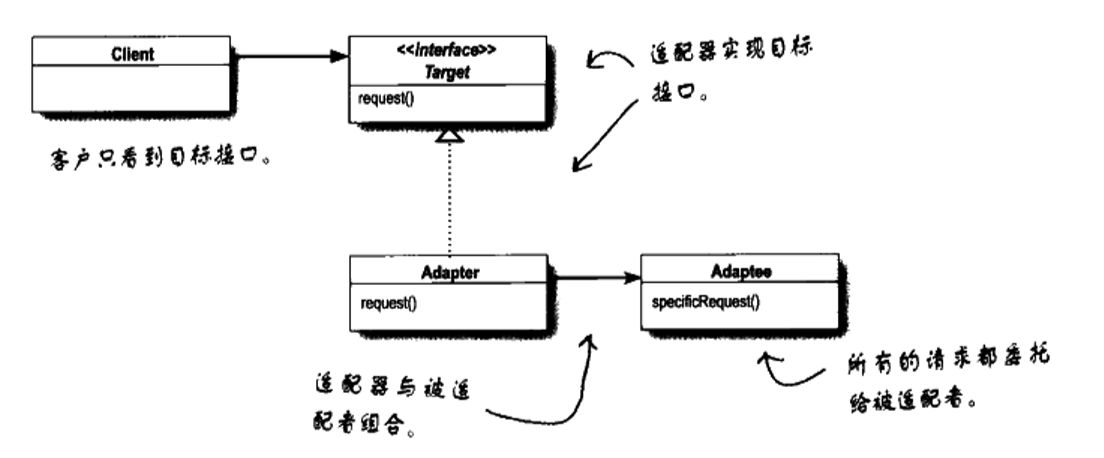
宏命令：利用命令模式和命令数组来执行一堆命令而不是一个命令。



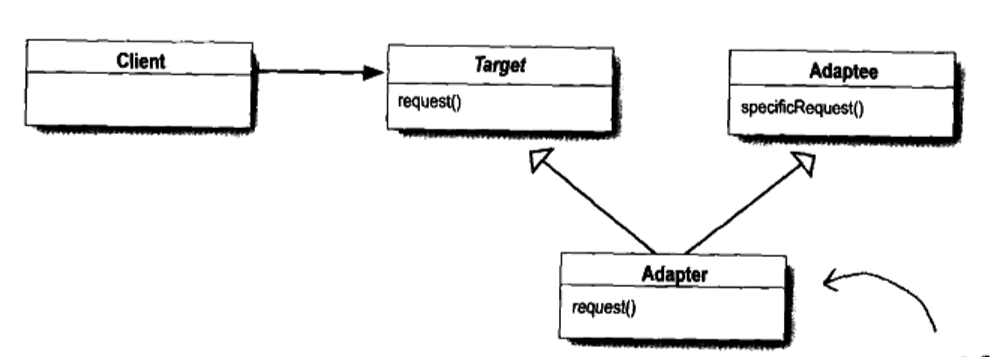
## 适配器模式（转换接口）

定义：将一个类的接口转换成客户期望的另一个接口，让原本接口不兼容的类可以合作无间。

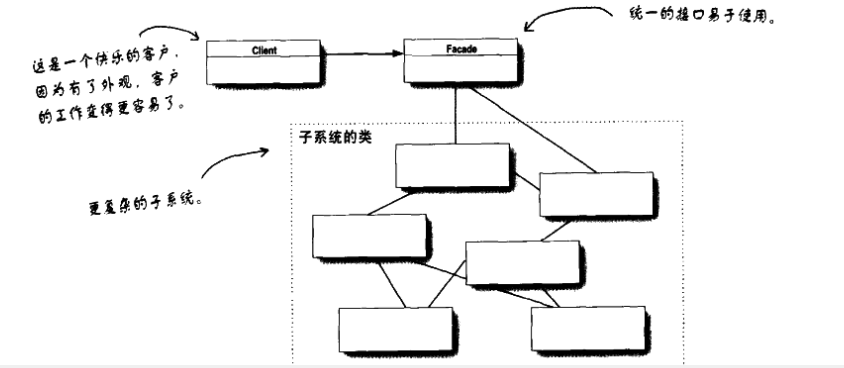
对象适配器类图：



类适配器类图：



## 外观模式（简化统一接口）



关键：将一个或数个类的复杂的一切都隐藏在背后，只露出一个干净美好的外观，让接口更简单。

**定义**：提供一个统一的接口，用来访问子系统中的一群接口。定义一个高层接口**让子系统更容易使用。**

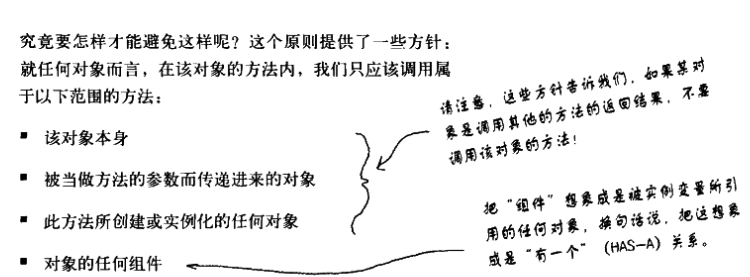
注意：

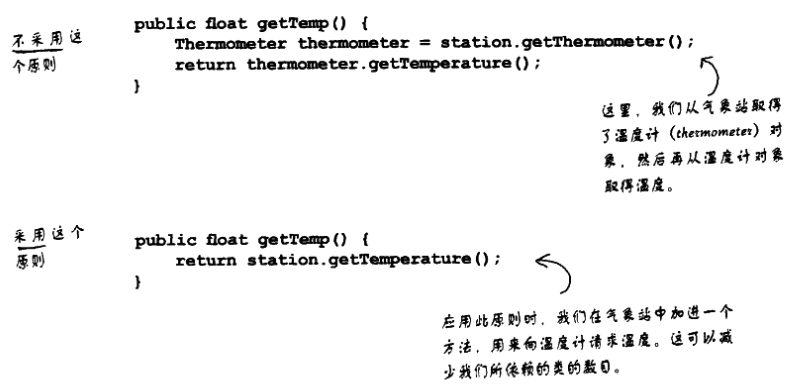
* 外观没有封装子系统，知识提供简化的接口。所以客户如果觉得有必要，依然可以使用子系统的类。
* 外观模式将客户从组件的子系统中解耦

**设计原则8：**最少知识原则，只和你的密友谈话。

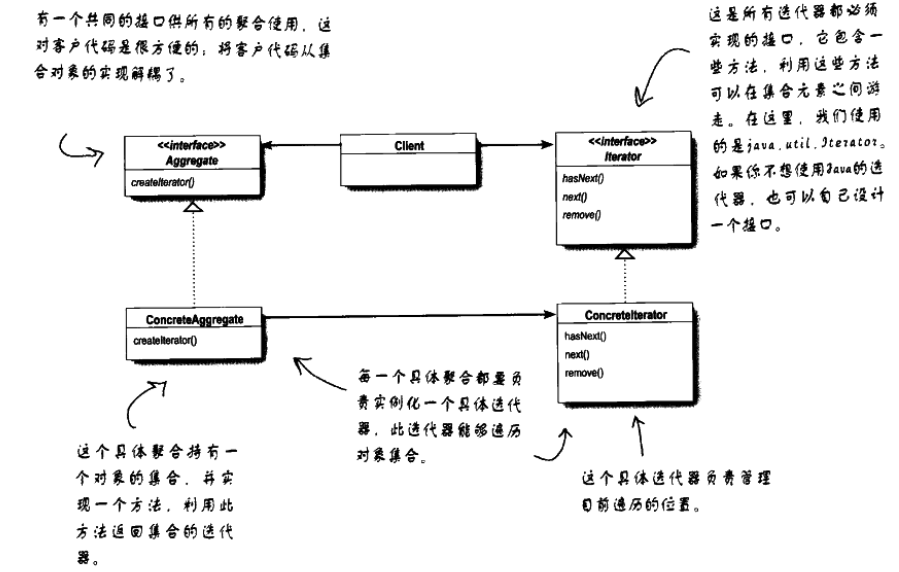
对于外观模式的客户端，它可以只有一个密友，就是外观类。

如何不要赢得太多的朋友和影响太多的对象？（不这样做的害处：增加我们直接认识对象的数目）





## 迭代器模式



**定义：**迭代器模式提供一种方法顺序访问一个聚合对象中的各个元素，又不暴露其内部表示（就是不知道这个聚合对象到底是用数组还是链表实现的）。把游走的任务放在迭代器上，而不是聚合类上，这样简化了聚合对象的接口和实现，也让责任各得其所。

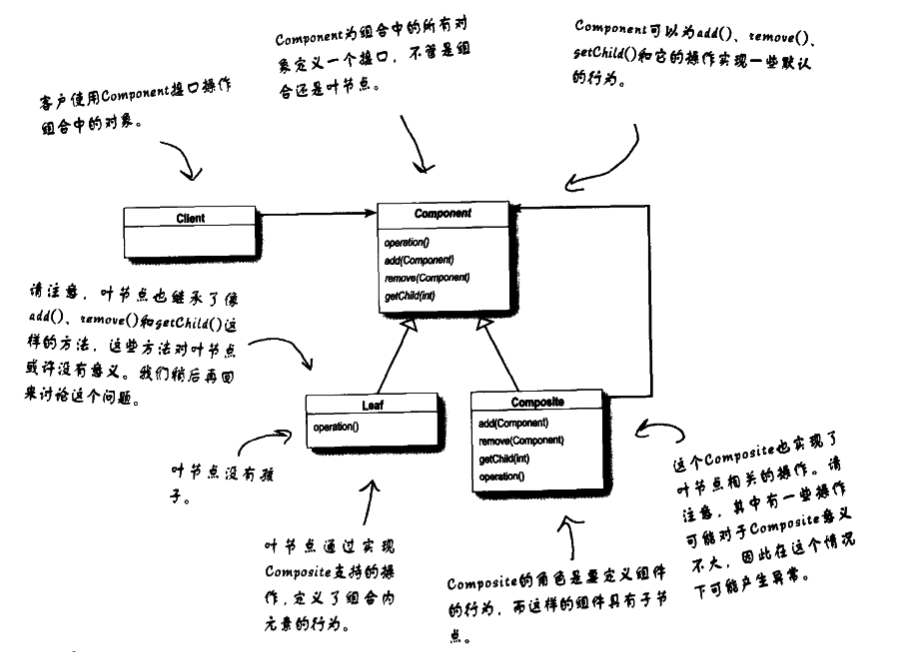
符合的原则：

单一职责原则

**内部迭代器：**使用内部迭代器，迭代器自行在元素之间游走，客户无法控制遍历的过程，所以比较容易使用，但是造成了弹性的损失。

## 组合模式

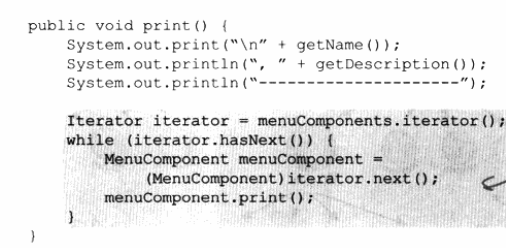
P359图



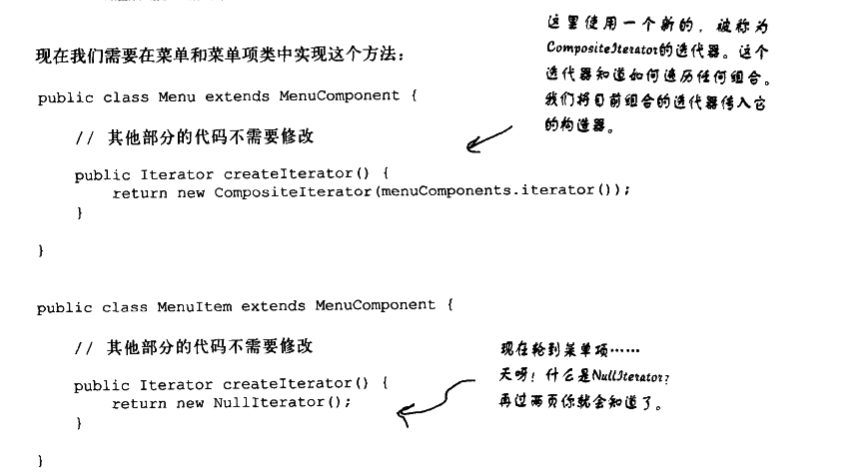
**定义：**组合模式允许你将对象组合成树形结构来表现“整体/部分”的层次结构。组合能让客户以一致的方式处理个别对象以及对象组合。

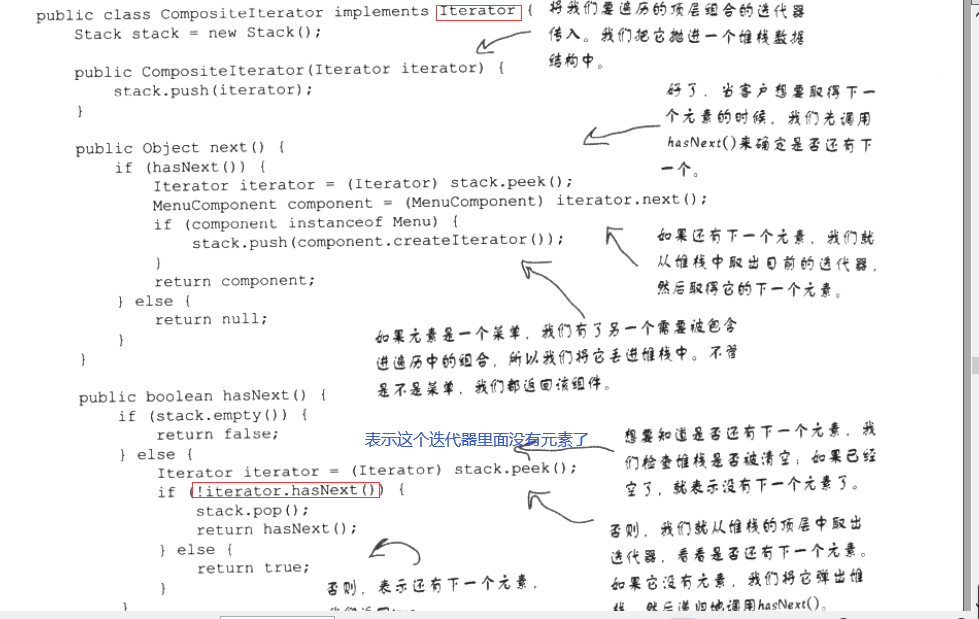
Composite 其实是一个聚合对象，这样当需要遍历其里面的子项目（包括其字字项目的时候，就可以使用迭代器模式）。

使用内部迭代器遍历对象：



使用外部迭代器

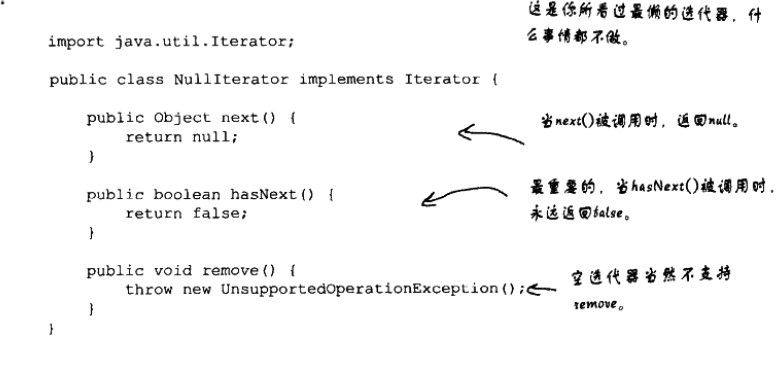




使用外部迭代器的遍历



空迭代器



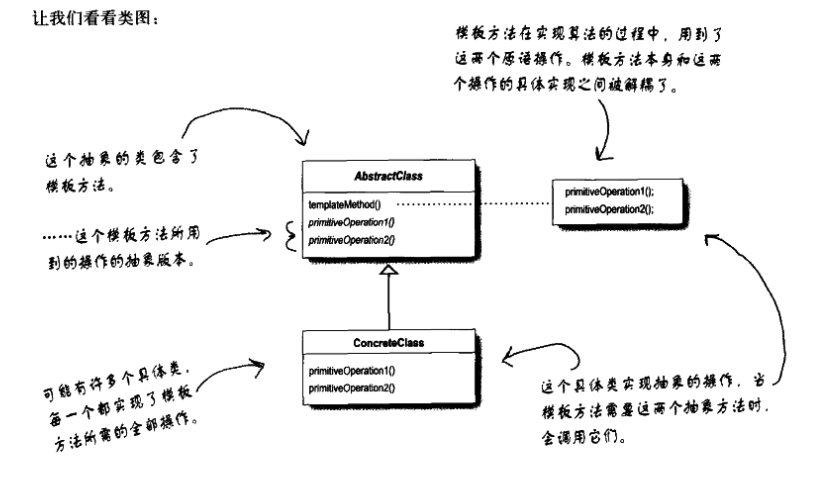
注意点：

* 组合组合模式牺牲单一职责原则换取了设计透明性（即通过让组件的接口同时包含一些管理子节点和叶节点的操作，客户端就可以将组合和叶节点一视同仁）。

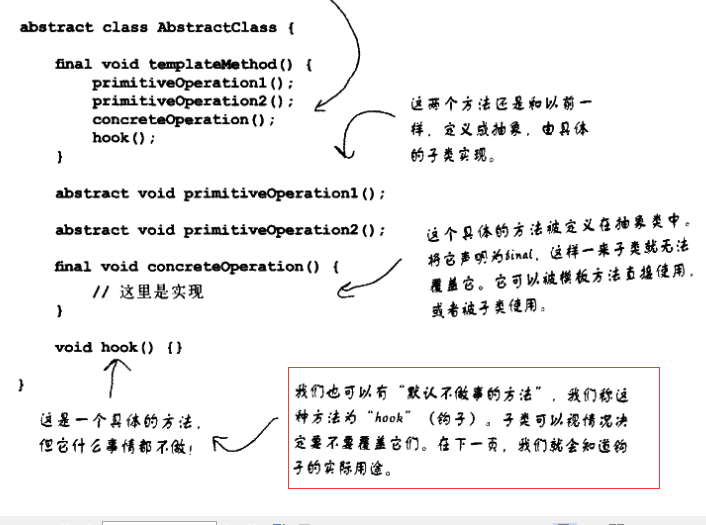
## 模板方法模式

定义：在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

类图：

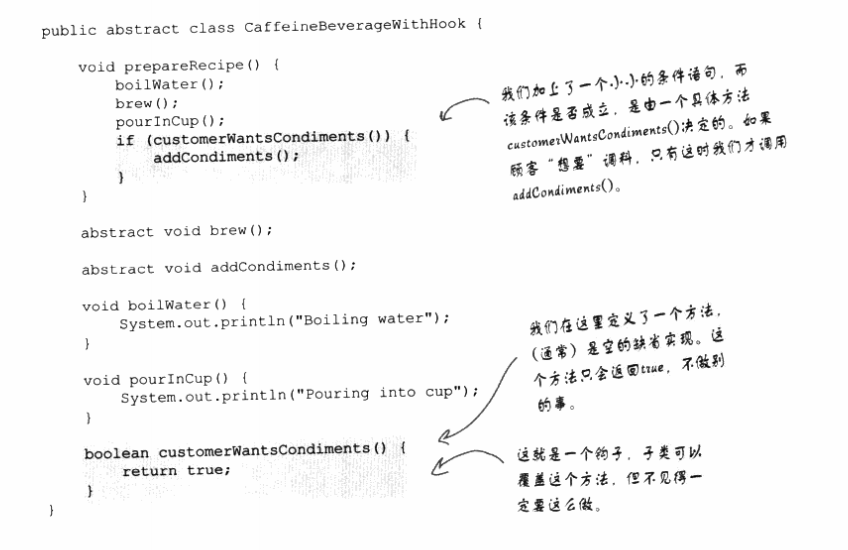


钩子，hook的实现。



钩子：钩子是一种被声明在抽象类中的方法，但是只有空的或者默认的实现。钩子的存在可以让子类有能力对算法的不同点挂钩。

使用钩子的例子：



问答：

1. 什么时候使用钩子？

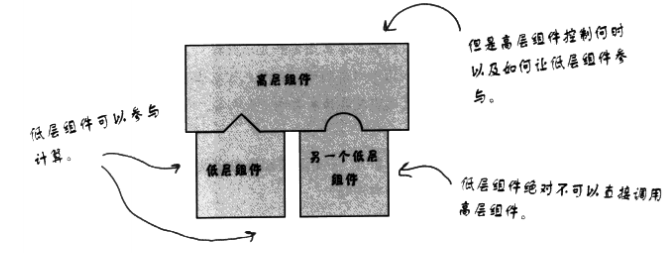
当子类必须提供算法中某个方法或步骤的实现时，就使用抽象方法。**如果算法的这个步骤是可选的，就用钩子**。如果是钩子的话子类可以选择实现，也可以选择不实现。

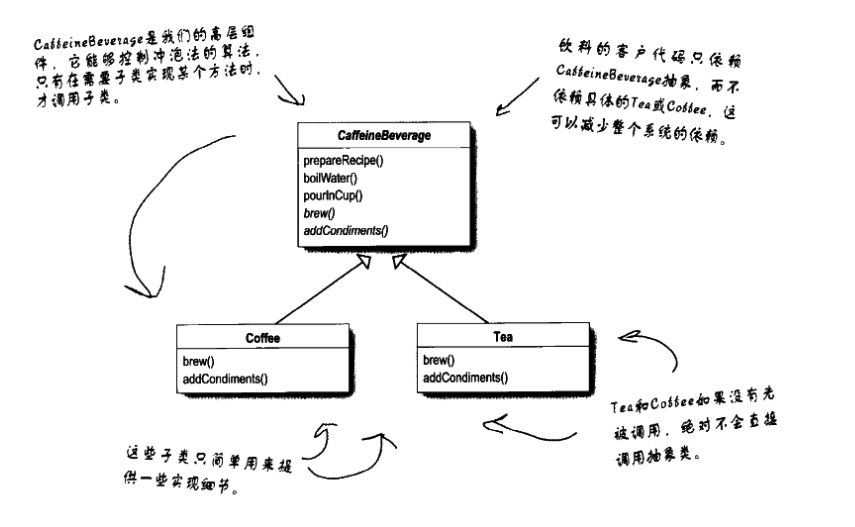
1. 使用钩子的目的？

让子类有能力为其抽象类作一些决定。

使用的原则：

**好莱坞原则：别调用我们，我们会调用你。**在这个原则之下，允许低层组件将自己挂钩到系统之上，但是高层组件会决定什么时候和怎样使用这些底层组件。（工厂方法、观察者都采用了好莱坞原则）





注意：

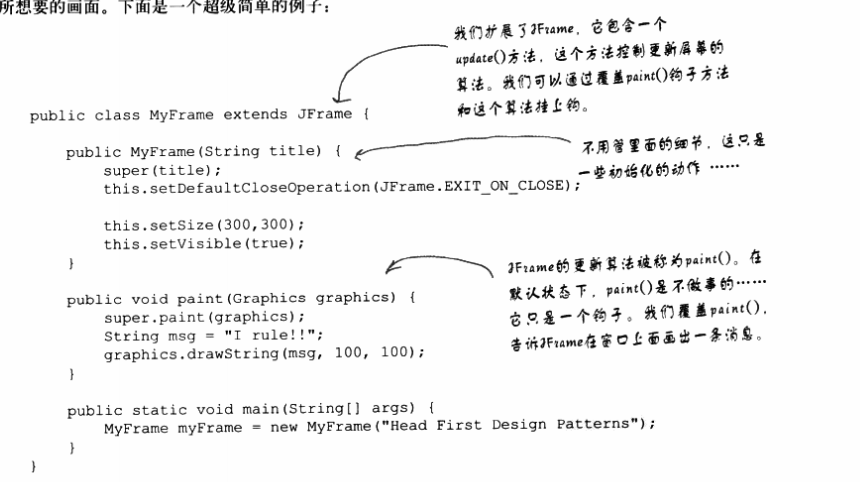
* 为了防止子类改变模板中的算法，可以将模板方法声明为final
* 策略模式和模板方法模式都封装算法，一个用的组合，一个用的继承。

### 好莱坞原则和依赖倒转原则的区别

依赖倒转原则教我们尽量避免使用具体类，而多用抽象。而好莱坞原则是在创建框架或组件上的一种技巧，好让底层组件能被挂进计算中，而又不会让高层组件依赖底层组件。

### 使用到模板方法的例子

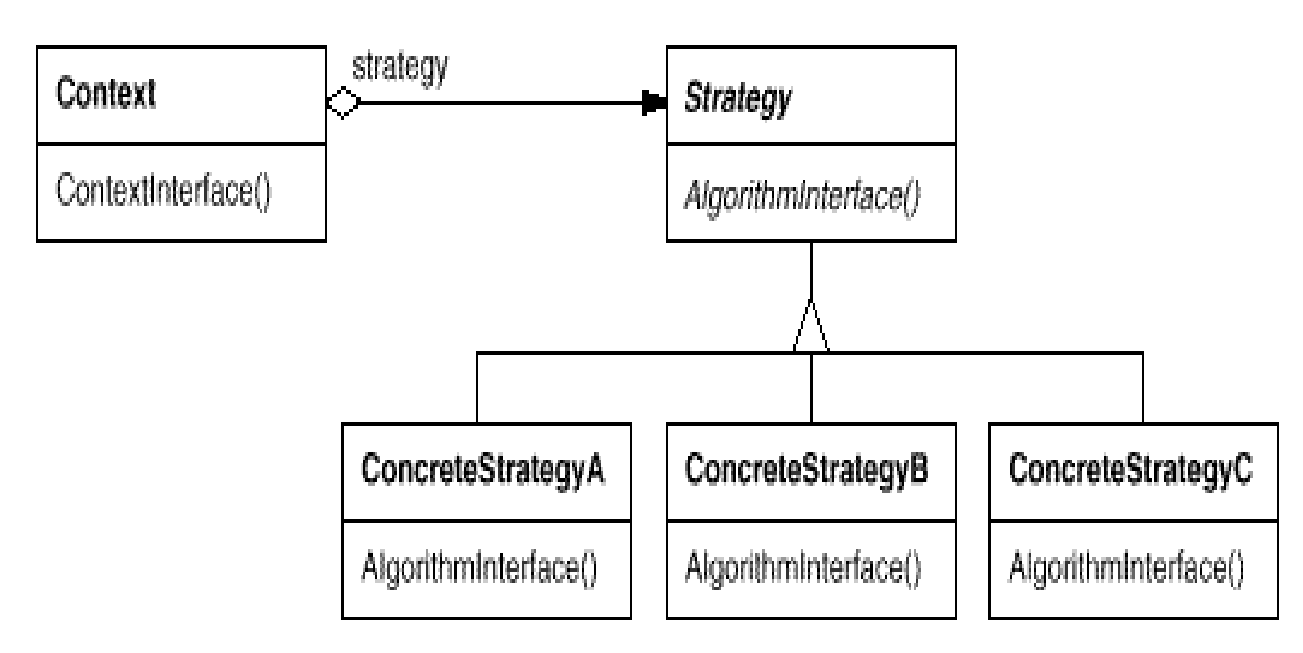
* Arrays.sort(Object[] args); //需要子类继承Comparable，实现里面的compareTo
* Jframe留给子类的钩子方法，paint



## 策略模式

**定义：**定义了算法族，分别封装起来，让它们之间可以相互替换，此模式让算法的变化独立于使用算法的客户。

框架：



设计原则：

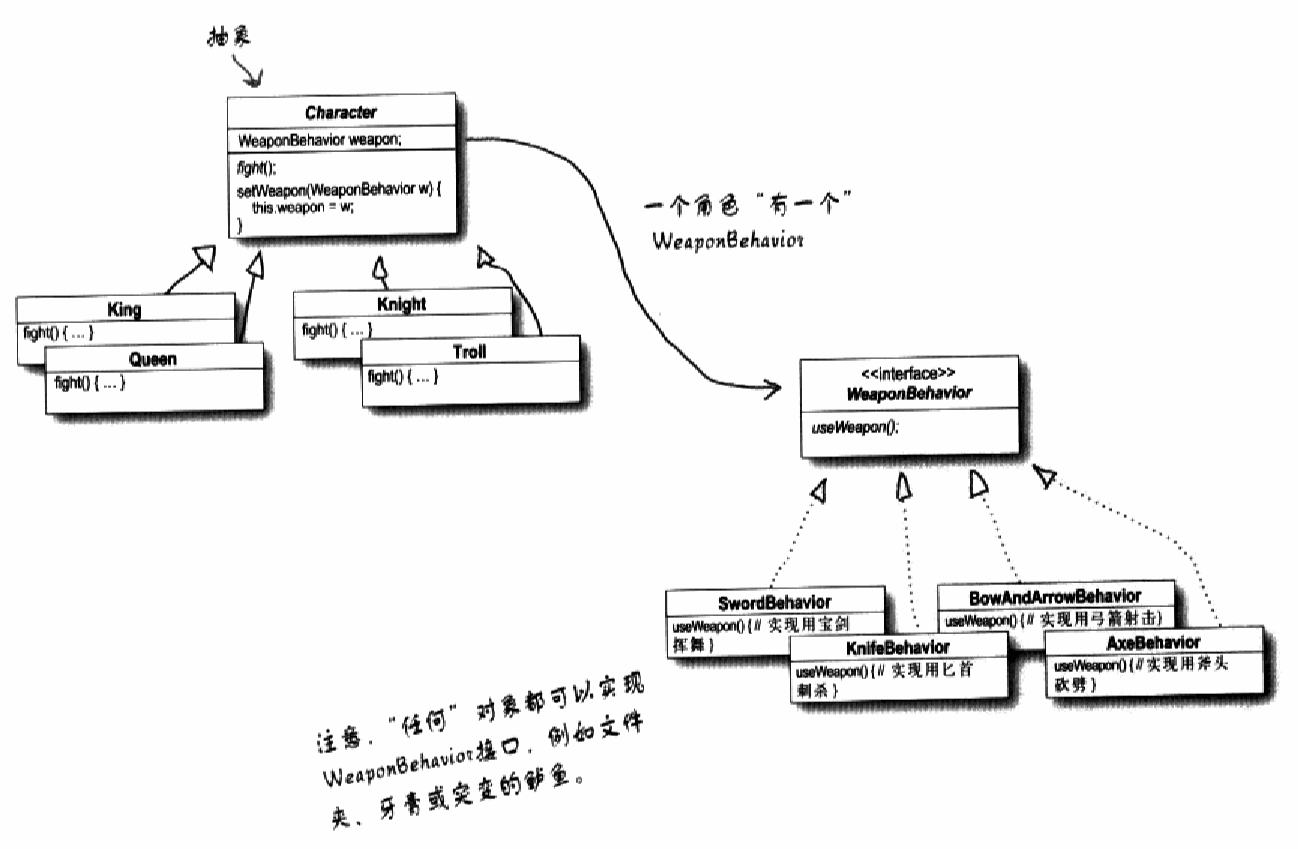
**设计原则1：**找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。（封装变化）

**设计原则1.1：**把会变化的部分取出来并封装起来，以便以后可以轻易地改动或扩充此部分，而不影响不需要变化的其他部分。（封装变化）

**设计原则2**：针对接口编程，而不是针对实现编程。

**设计原则3**：多用组合，少用继承

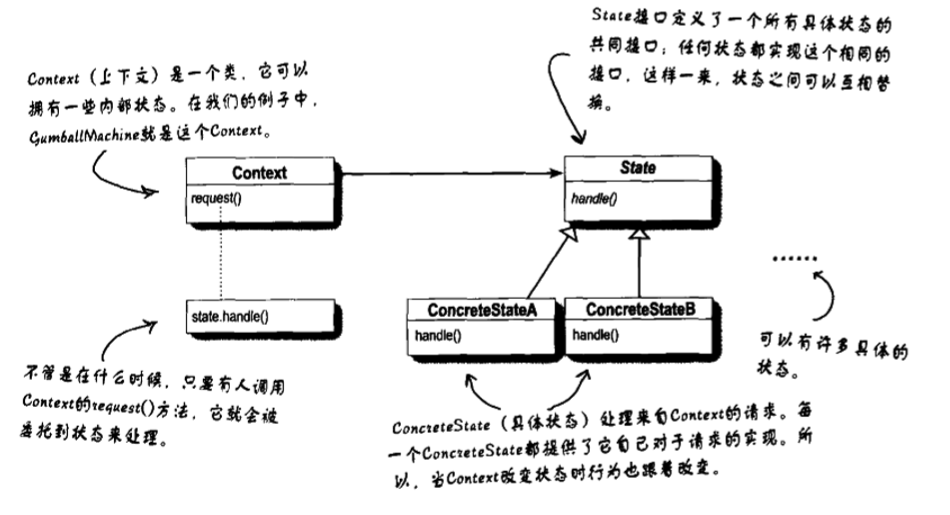
下图，**例如：**各种角色继承超类，超类有一个角色WeaponBehavior接口，使用具体的兵器用这个接口去实现，它连着具体的兵器接口SwordBehavior等。它们共享着useWeapon方法，具体的实现在WeaponBehavior中。角色想要换武器，可以调用超类中的setWeapon()方法。



## 状态模式

**定义：**允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类

类图：



好处：

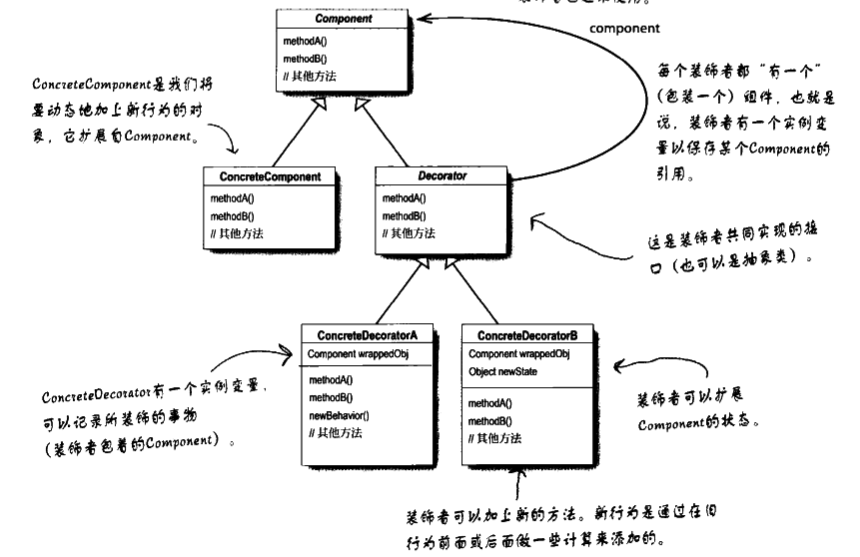
* 封装变化。将每个状态的的行为局部化到它自己的内部。
* 将容易产生问题的if语句删除，以方便日后的维护。
* 符合开闭原则。让每一个状态“对修改关闭”，让糖果机“对扩展开放”，因为可以加入新的状态类。

注意点：

1. 当状态转换是固定的时候，就（将转变这个动作）适合放到context中，当转换是动态的时候，就会放在状态类中，**但是缺点是：状态类之间产生了依赖**。
2. 客户不会直接和状态进行交互
3. Context对象如果有许多实例，那么这些实例可以共享状态对象，但前提是这些状态对象不能持有它们自己的内部状态（就是不能持有context某个特性实例的引用，因为此时状态需要对所有的context实例服务，如果状态需要用到context，那么可以将状态方法定义为： handle(Context context)），此时这些状态都被指定为静态实例。

第十二种：迭代器与组合模式

## 装饰者模式



动态地将责任附加到对象上，若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

**继承的缺点**：弹性有限，不便于维护

改进：组合与委托，组合可以扩展对象的行为有利于动态扩展；组合可以动态维护代码，有新功能就写新代码，无需修改现有代码。

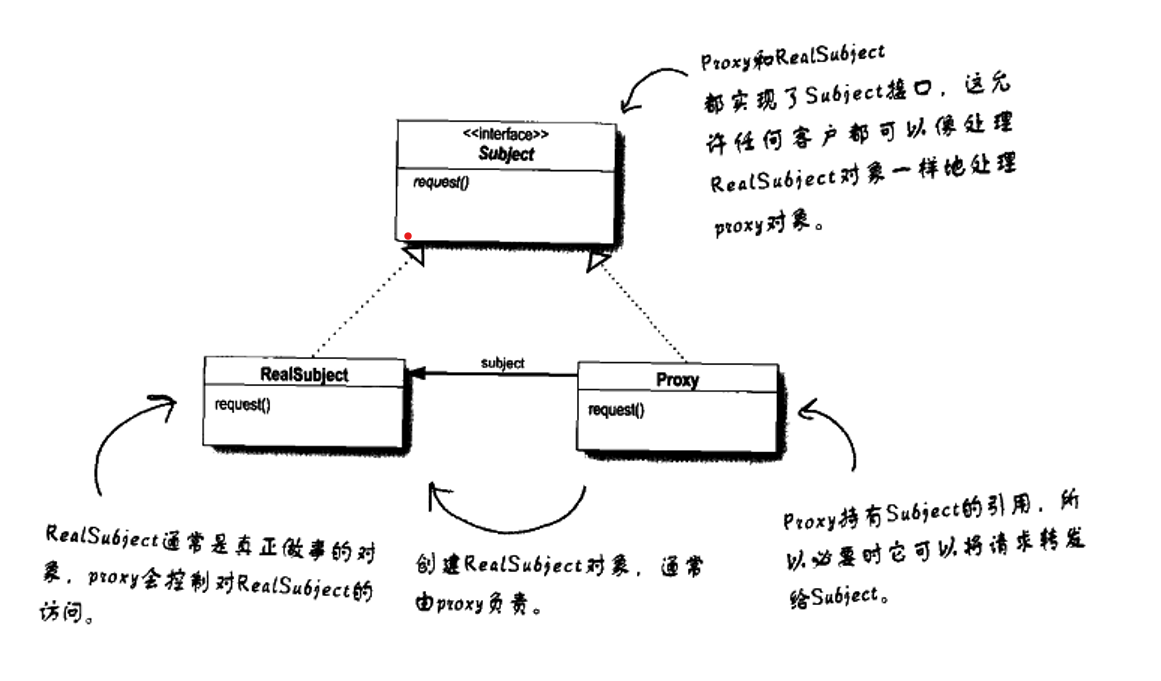
**设计原则5：**类应该对扩展开放，对修改关闭。（开闭原则）

**关键点：**装饰者可以在所委托被装饰者的行为之前或之后加上自己的行为，以达到特定的目的。

**优点：**为设计注入弹性。使用组合代替继承；动态扩展类的功能，继承是静态的；组件类与装饰类可独立变化，相互组合，可组合出功能更加强大的类。

**缺点：**引入大量小类，工厂模式和生成器模式可以改进这一点。

## 代理模式（Proxy/Stub）：



定义：代理模式为另一个对象提供一个替身或占位符以控制对这个对象的访问。

适用情形：

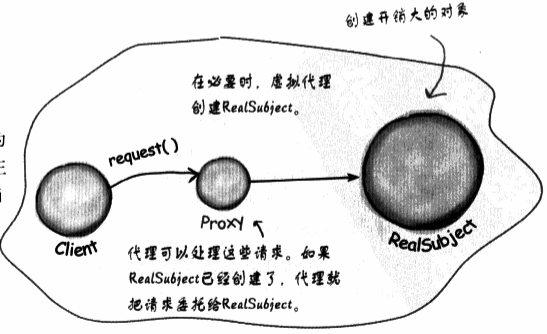
使用代理模式创建代表，让代表对象控制某对象的访问，被代理的对象可以是远程对象、创建开销大的对象，或者需要安全控制的对象

几种代理方式：

**远程代理**：远程代理可以作为另一个JVM上对象的本地代表，调用代理的方法，会被代理利用通过网络转发到远程执行并且结果会通过网络返回给代理，再由代理将结果转给客户。



**虚拟代理**：虚拟代理作为创建**开销大**的对象的代表。虚拟代理经常直到我们真正需要一个对象的时候才创建它。当对象在创建前和创建中，由虚拟对象来扮演对象的替身。对象创建后，代理就会直接委托给对象。



保护代理。

注意点：

* 在真实世界代理模式有许多变体，但是这些变体都有共同点：都会将客户对主题施加的方法拦截下来**。（控制对象的访问）**

问答：

1. 代理模式和装饰者模式的区别？

他们的目的不同。装饰着为对象增加行为，而代理则是控制对象的访问。

1. 如何让对象只能使用代理？

使用工厂模式，实例化返回主题，返回的是代理对象。

1. 代理模式和适配器模式的区别？

代理和适配器都是挡在其他对象前面，并负责将请求转发给他们。但是：适配器会改变对象适配的接口（适配器对象和需要适配的对象继承不同的接口），而代理对象则和需要代理的对象实现同一个接口。