《Head First 设计模式》读书笔记

1、欢迎来到设计模式的世界：设计模式入门

策略模式：定义算法族，分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

OO原则：

封装变化

多用组合，少用继承

针对接口编程，不针对实现编程

OO基础：

抽象

封装

多态

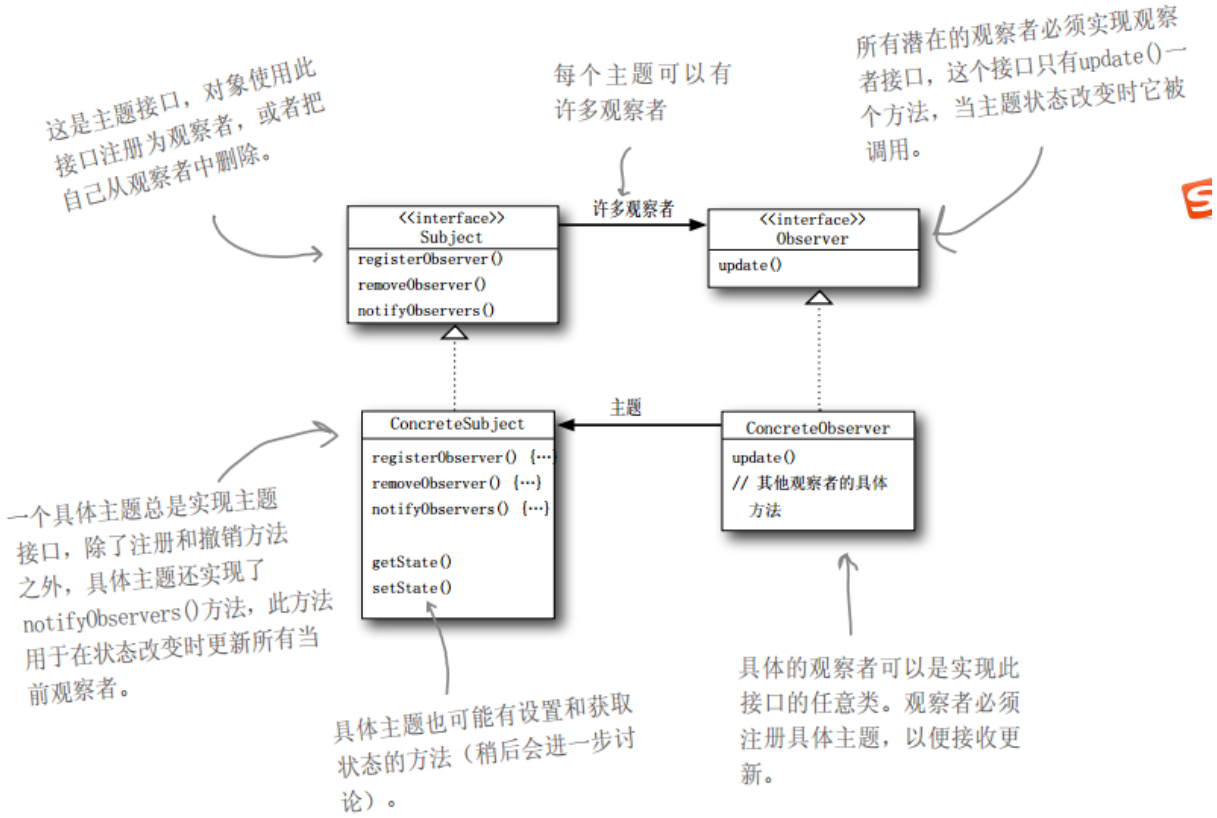
继承

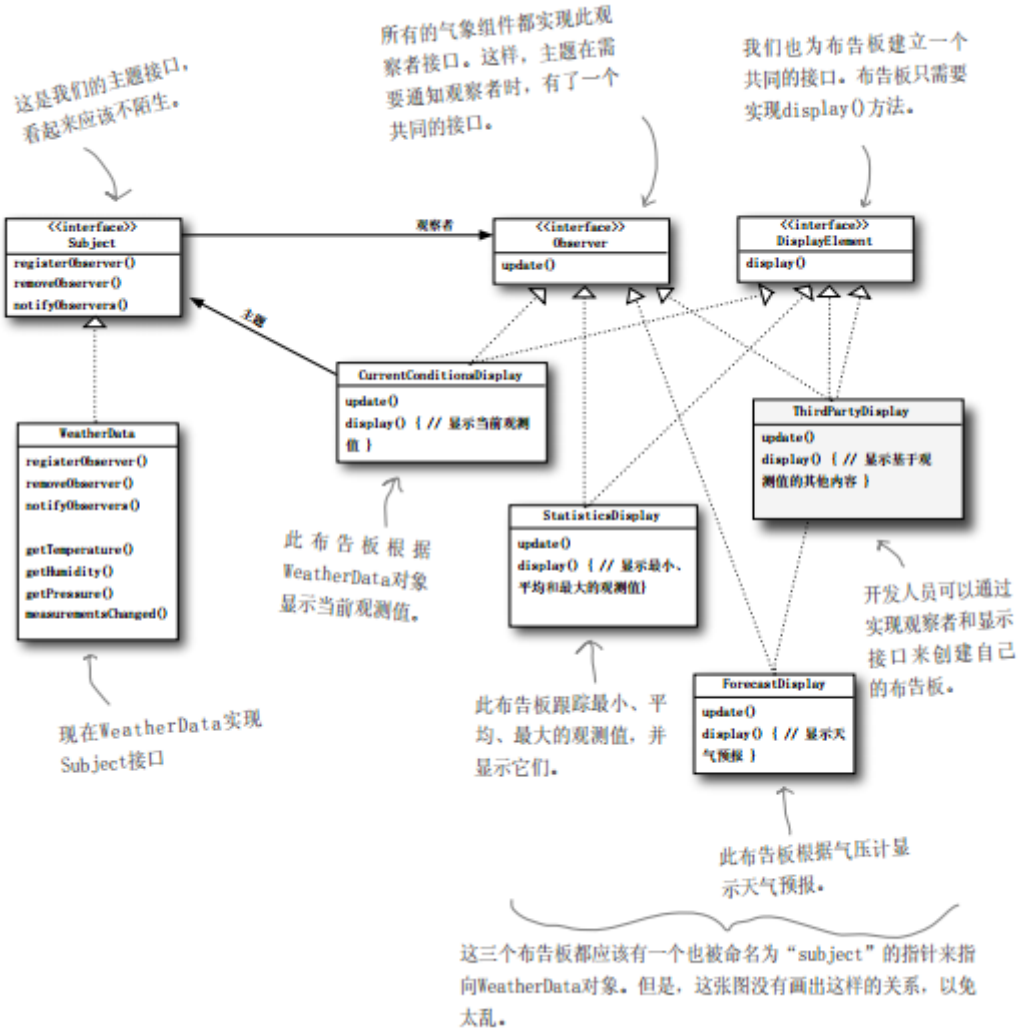
2、让你的对象知悉现况：观察者模式

观察者模式：在对象之间定义一对多的依赖，这样一来，当一个对象改变状态，依赖它的对象都会收到通知，并自动更新。

OO原则：

为交互对象之间的松耦合设计而努力



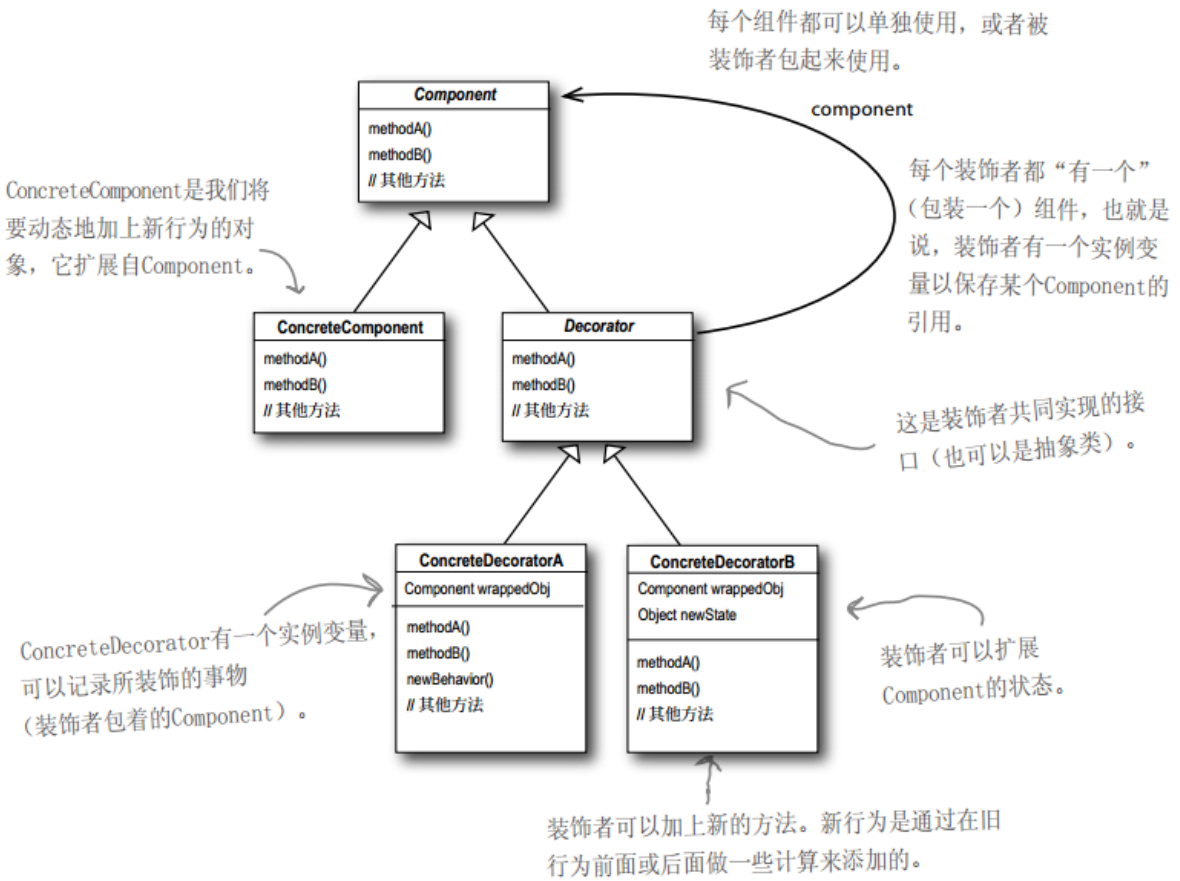


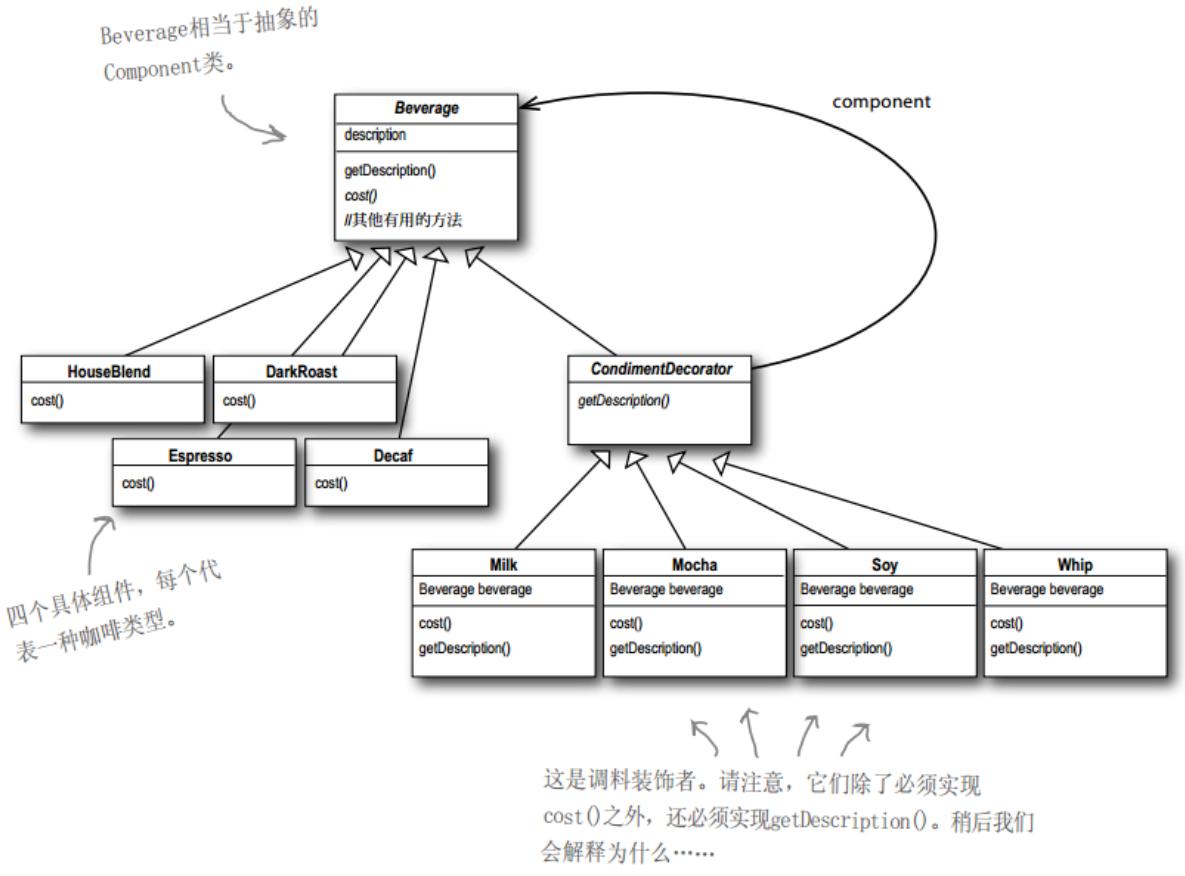
3、装饰对象：装饰者模式

装饰者模式：动态地将责任附加到对象上。想扩展功能，装饰者提供有别于继承的另一种选择。

OO原则：

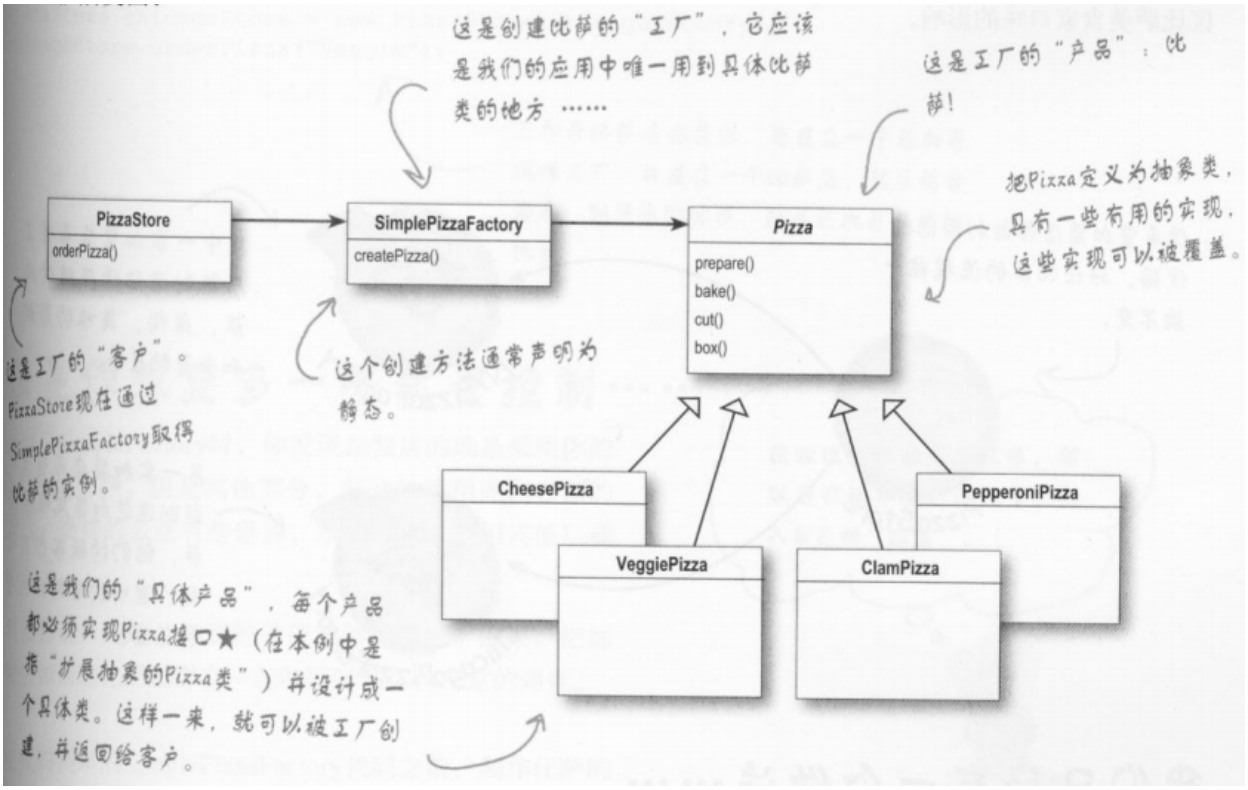
对扩展开放，对修改关闭。


每个组件都可以单独使用，或者被装饰者包起来使用。
Component
methodA()
methodB()
// 其他方法
component
每个装饰者都“有一个”（包装一个）组件，也就是说，装饰者有一个实例变量以保存某个Component的引用。
ConcreteComponent是我们将要动态地加上新行为的对象，它扩展自Component。
ConcreteComponent
methodA()
methodB()
// 其他方法
Decorator
methodA()
methodB()
// 其他方法
这是装饰者共同实现的接口（也可以是抽象类）。
ConcreteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// 其他方法
ConcreteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// 其他方法
ConcreteDecorator有一个实例变量，可以记录所装饰的事物（装饰者包着的Component）。
装饰者可以扩展Component的状态。
装饰者可以加上新的方法。新行为是通过在旧行为前面或后面做一些计算来添加的。

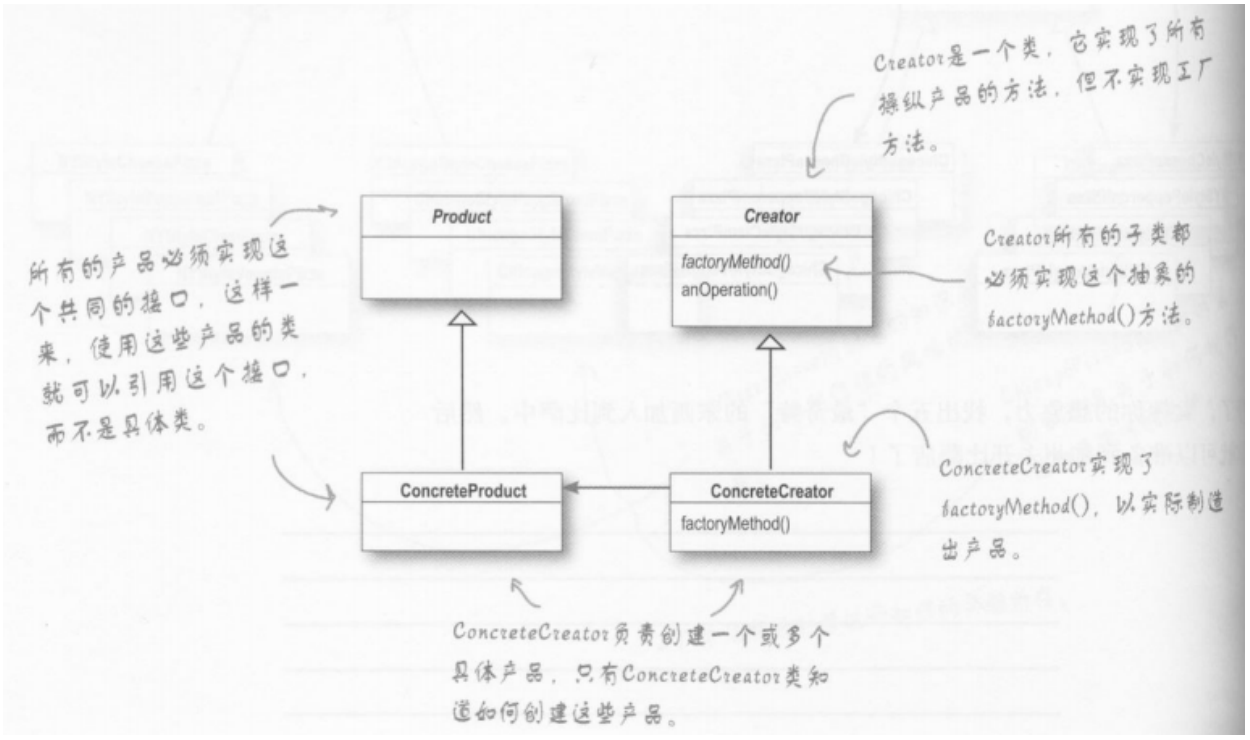

Beverage相当于抽象的Component类。
Beverage
description
getDescription()
cost()
//其他有用的方法
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
四个具体组件，每个代表一种咖啡类型。
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
这是调料装饰者。请注意，它们除了必须实现cost()之外，还必须实现getDescription()。稍后我们会解释为什么……

4、烘烤OO的精华：工厂模式

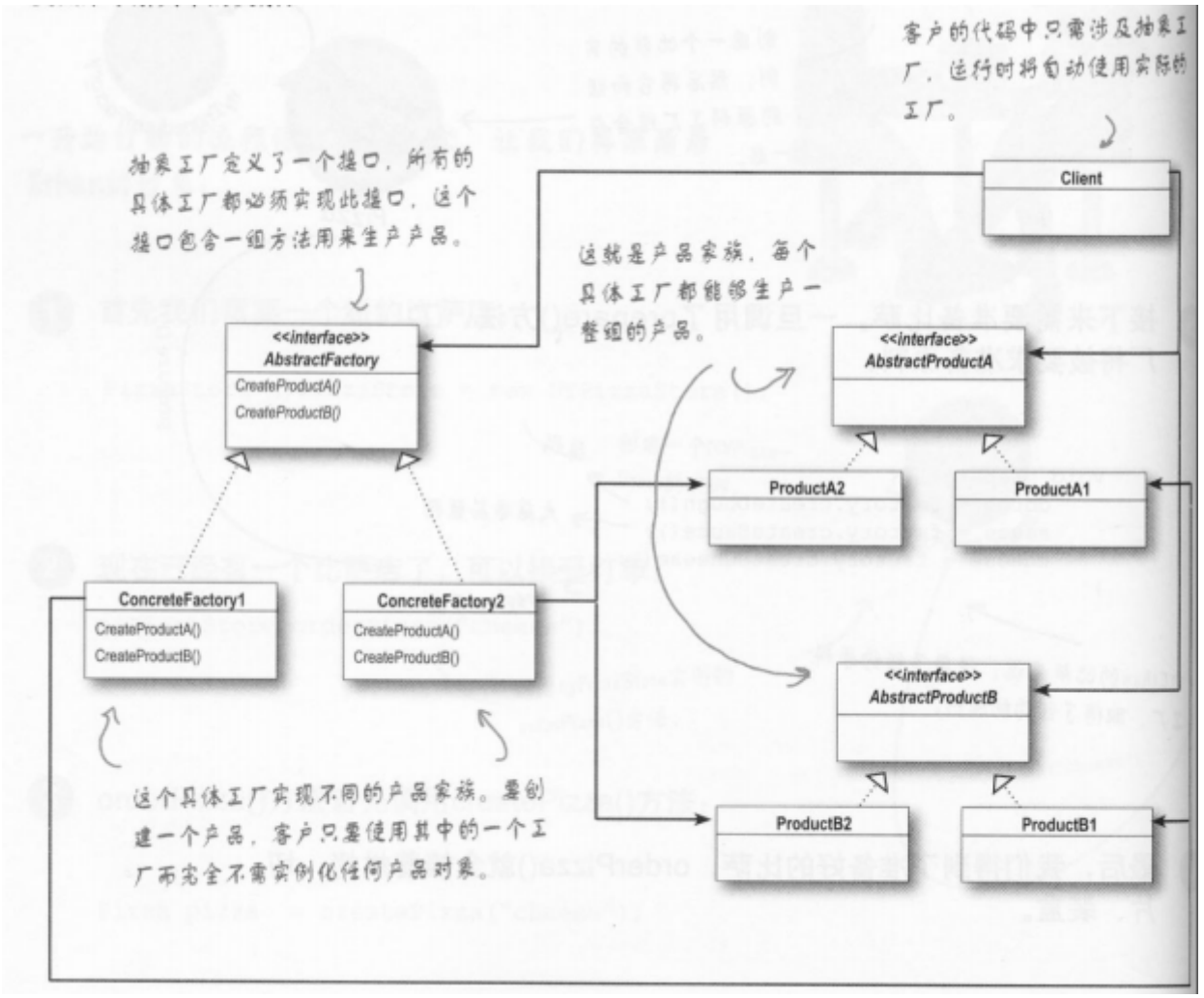
简单工厂模式：

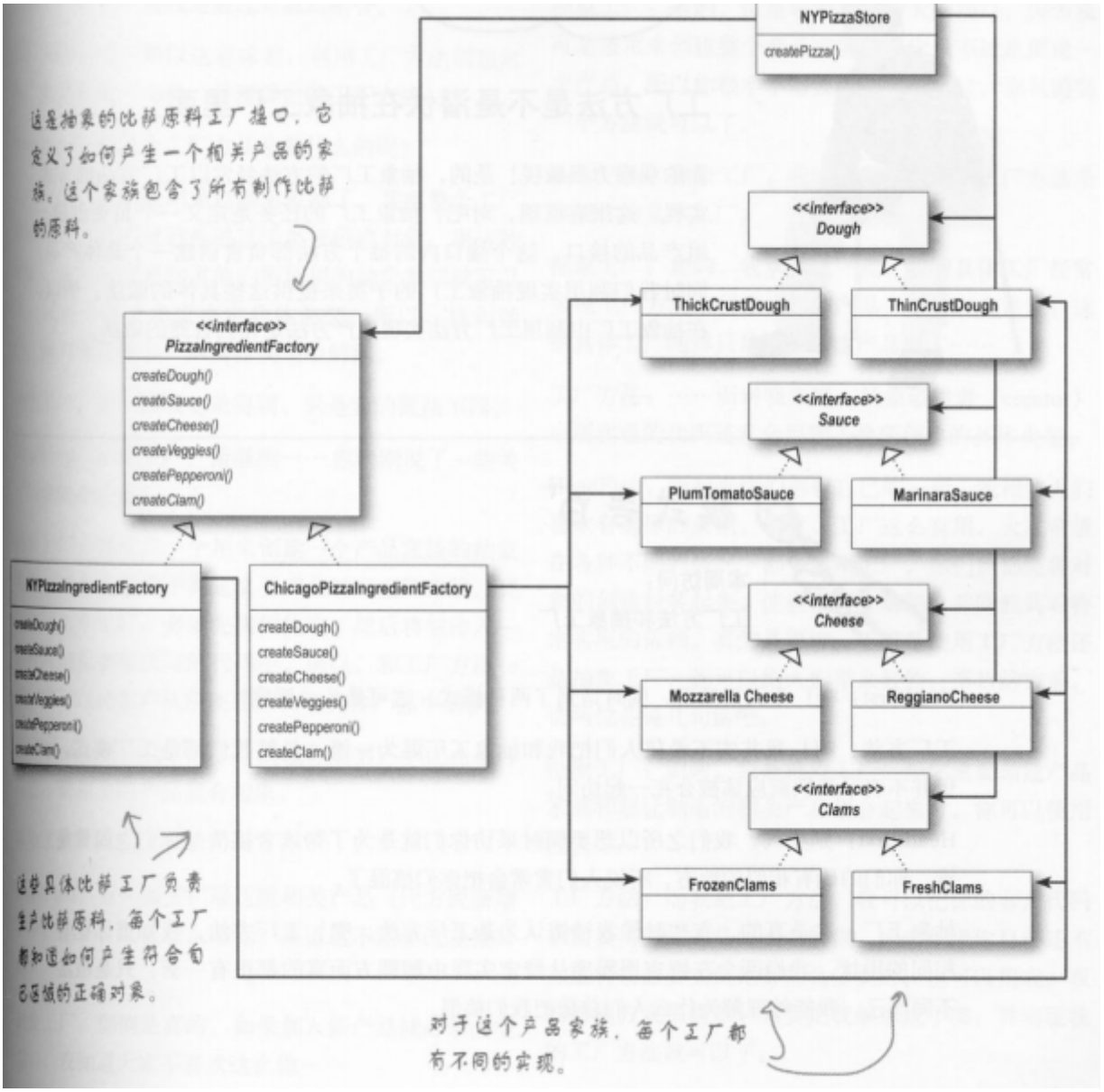


工厂方法模式：定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。



- [ ] 抽象工厂模式：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。


客户的代码中只需涉及抽象工厂，运行时将自动使用实际的工厂。
抽象工厂定义了一个接口，所有的具体工厂都必须实现此接口，这个接口包含一组方法用来生产产品。
这就是产品家族，每个具体工厂都能够生产一整组的产品。
Client
<<interface>>
AbstractFactory
CreateProductA()
CreateProductB()
<<interface>>
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
<<interface>>
AbstractProductB
ProductB2
ProductB1
这个具体工厂实现不同的产品家族。要创建一个产品，客户只要使用其中的一个工厂而完全不需实例化任何产品对象。

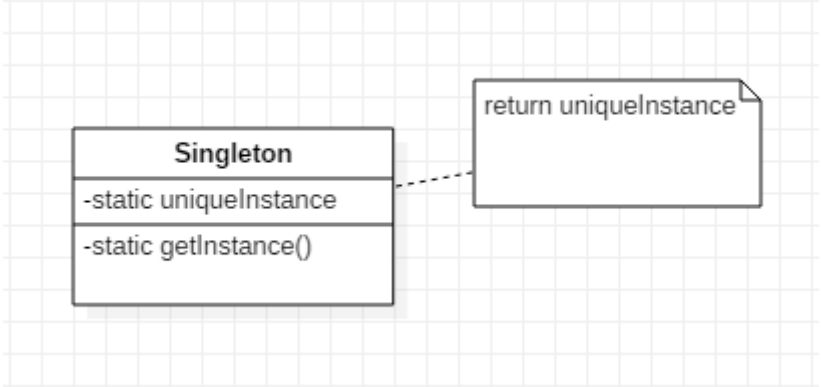


OO原则：

依赖抽象，不要依赖具体类。

5、独一无二的对象：单件模式

单件模式：确保一个类只有一个实例，并提供全局访问点。



适用场景：

1.需要生成唯一序列的环境

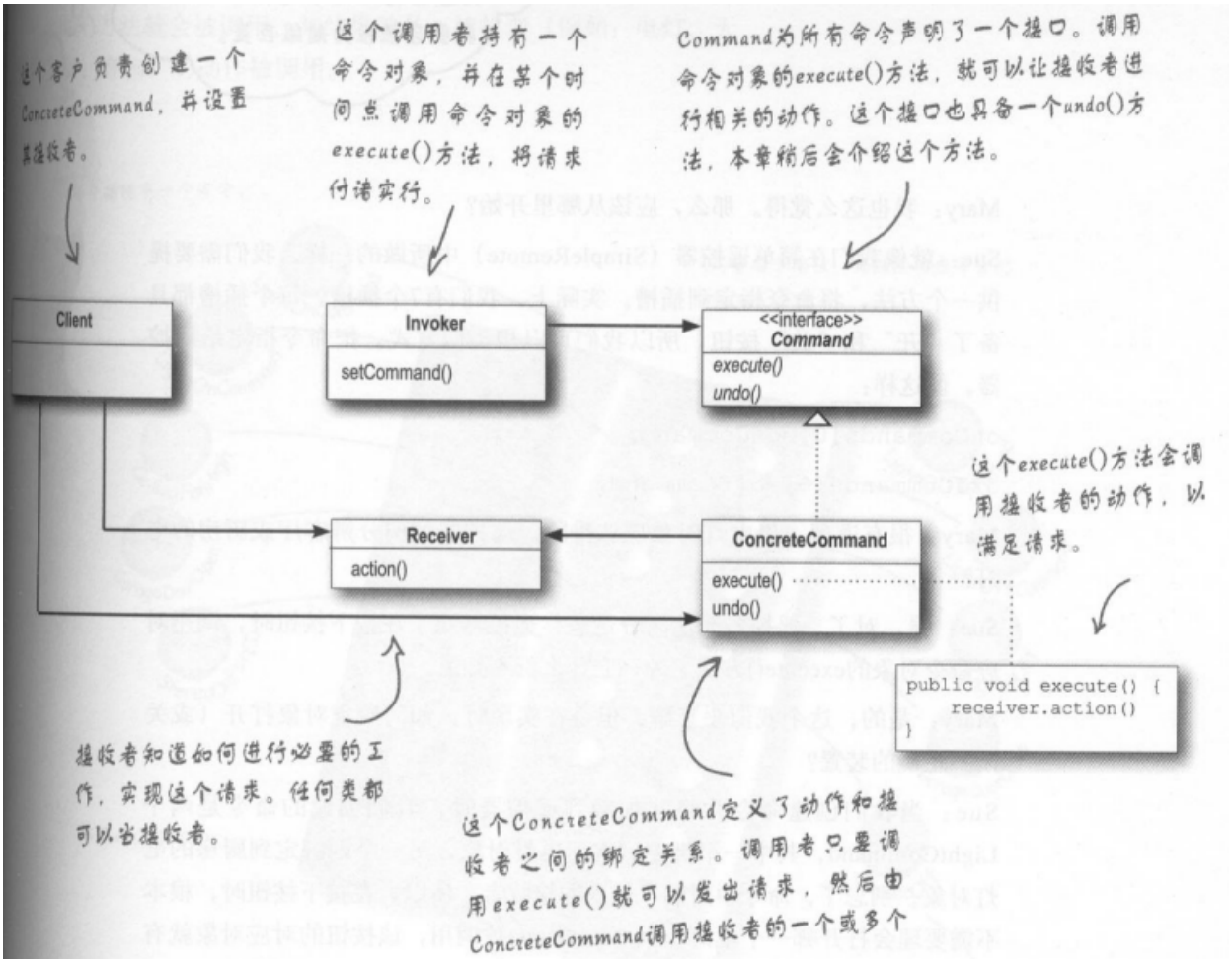
2.需要频繁实例化然后销毁的对象
3.创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
4.方便资源相互通信的环境

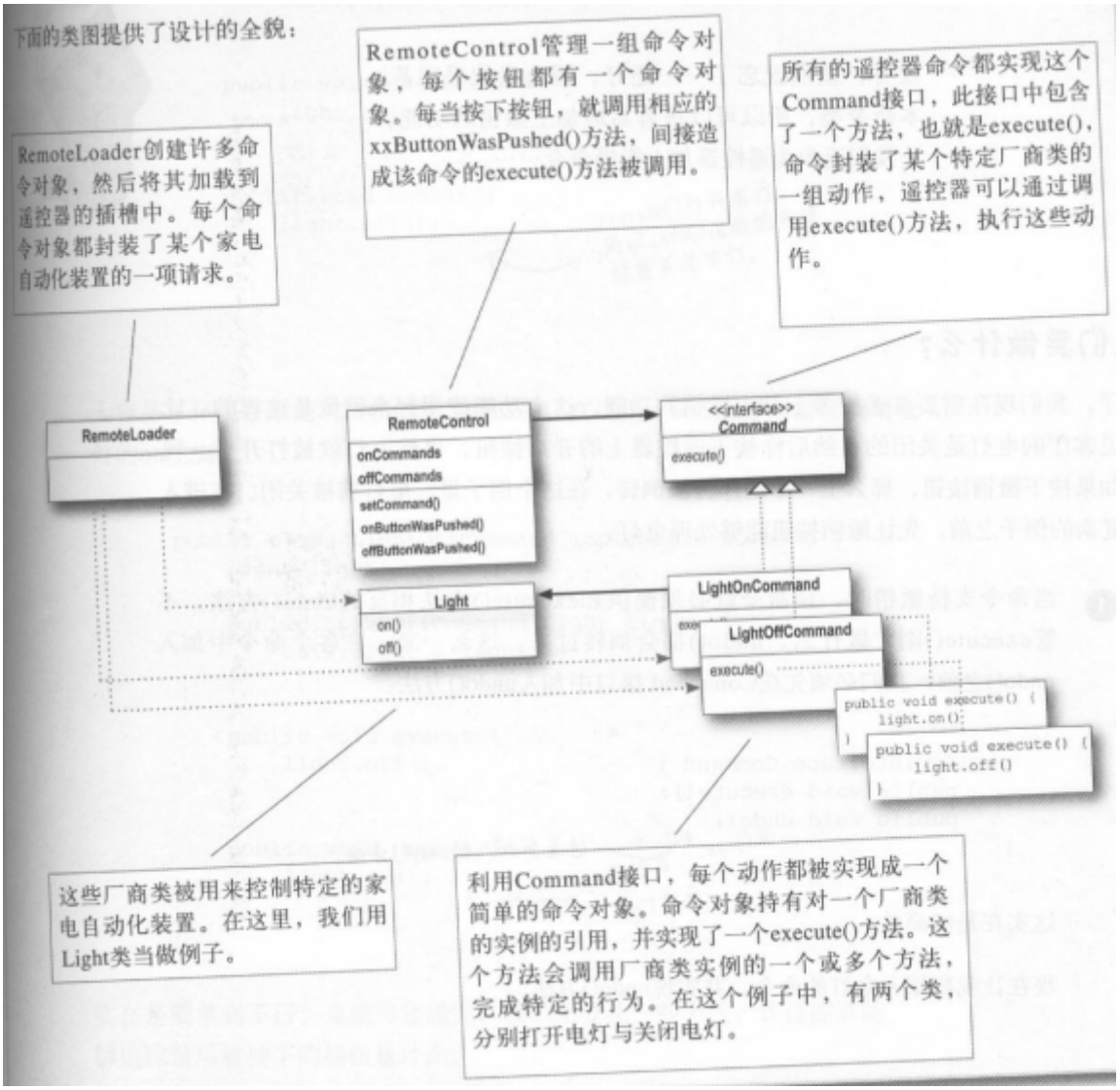
优点：
1.实现了对唯一实例访问的可控
2.对于一些需要频繁创建和销毁的对象来说可以提高系统的性能。

缺点：
1. 不适用于变化频繁的对象
2.滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出。
3.如果实例化的对象长时间不被利用，系统会认为该对象是垃圾而被回收，这可能会导致对象状态的丢失。

6、封装调用：命令模式

命令模式：将请求封装成对象，这可以让你使用不同的请求、队列，或日志请求来参数化其他对象。命令模式也可以支持撤销操作。
四人组：将请求封装为一个对象，从而使你可用不同的请求对客户进行参数化;对请求排队或记录请求日志，以及支持可撤回的操作。


这个客户负责创建一个ConcreteCommand，并设置其接收者。
这个调用者持有一个命令对象，并在某个时间点调用命令对象的execute()方法，将请求付诸实行。
Command为所有命令声明了一个接口。调用命令对象的execute()方法，就可以让接收者进行相关的动作。这个接口也具备一个undo()方法，本章稍后会介绍这个方法。
Client
Invoker
setCommand()
<<interface>>
Command
execute()
undo()
这个execute()方法会调用接收者的动作，以满足请求。
Receiver
action()
ConcreteCommand
execute()
undo()
public void execute() {
receiver.action()
}
接收者知道如何进行必要的工作，实现这个请求。任何类都可以当接收者。
这个ConcreteCommand定义了动作和接收者之间的绑定关系。调用者只要调用execute()就可以发出请求，然后由ConcreteCommand调用接收者的一个或多个

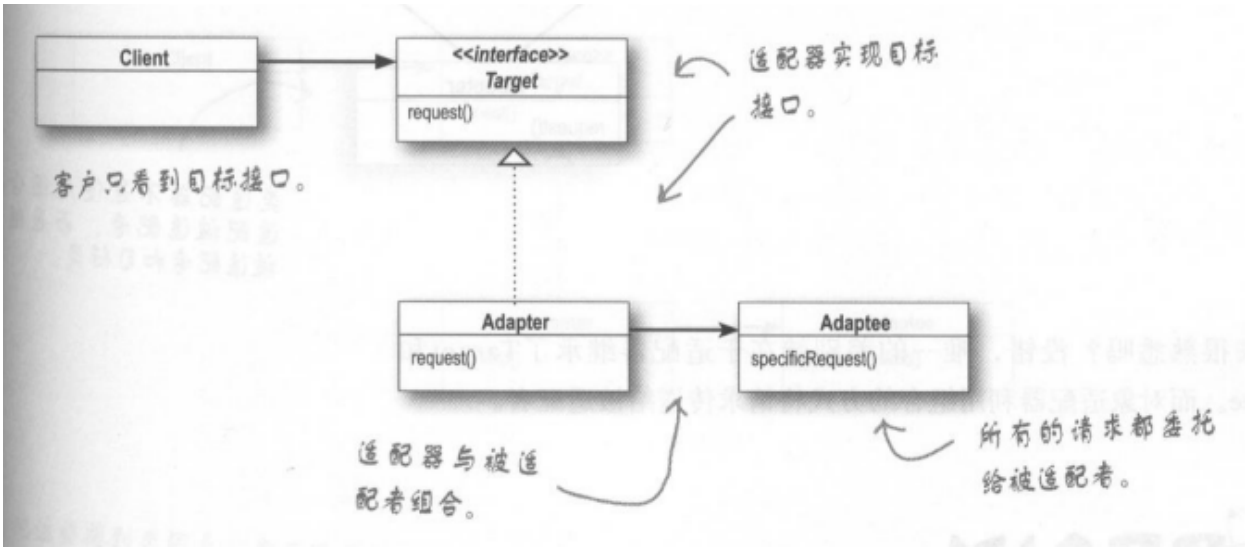


解说：

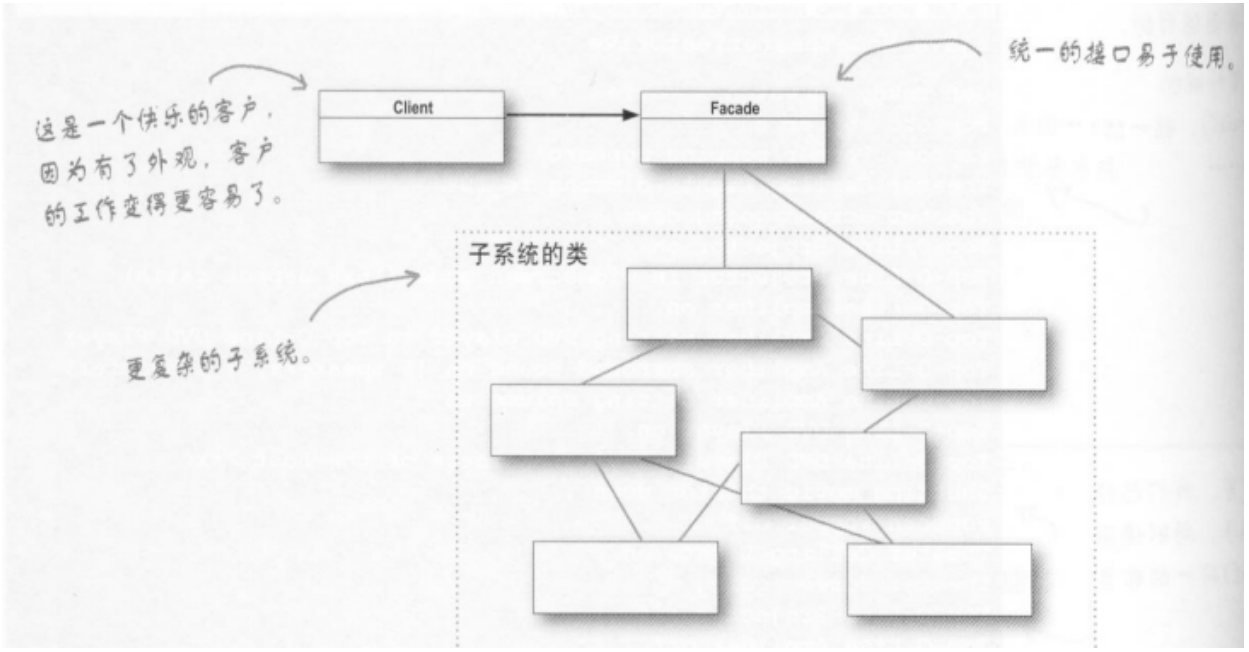
light是接收者:receiver

7、随遇而安：适配器与外观模式（facade）

适配器模式：将一个类的接口，转换成客户期望的另一个接口。适配器让原本不兼容的类可以合作无间。



外观模式：提供一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。



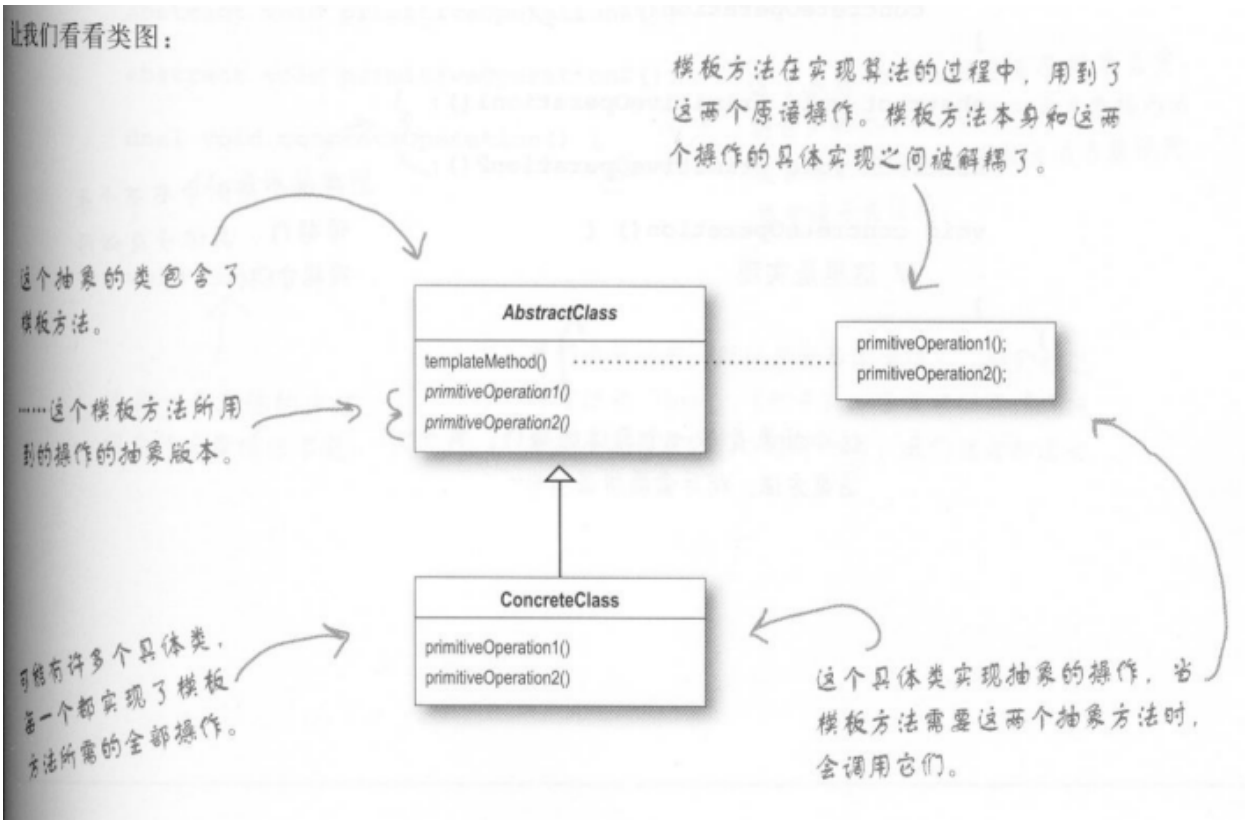
OO原则：

只和朋友交谈。

8、封装算法：模板方法模式

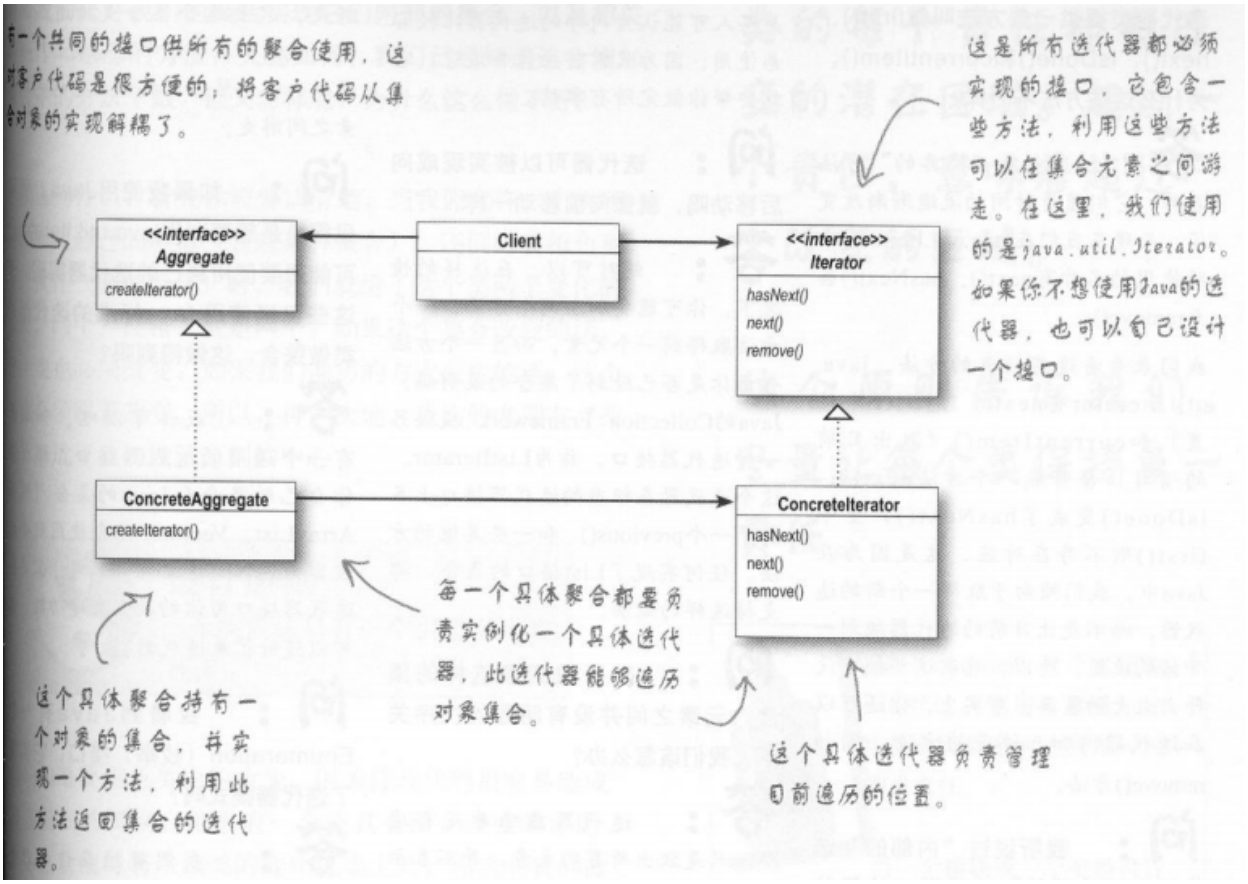
模板方法模式：在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

OO原则：别找我，我会找你。

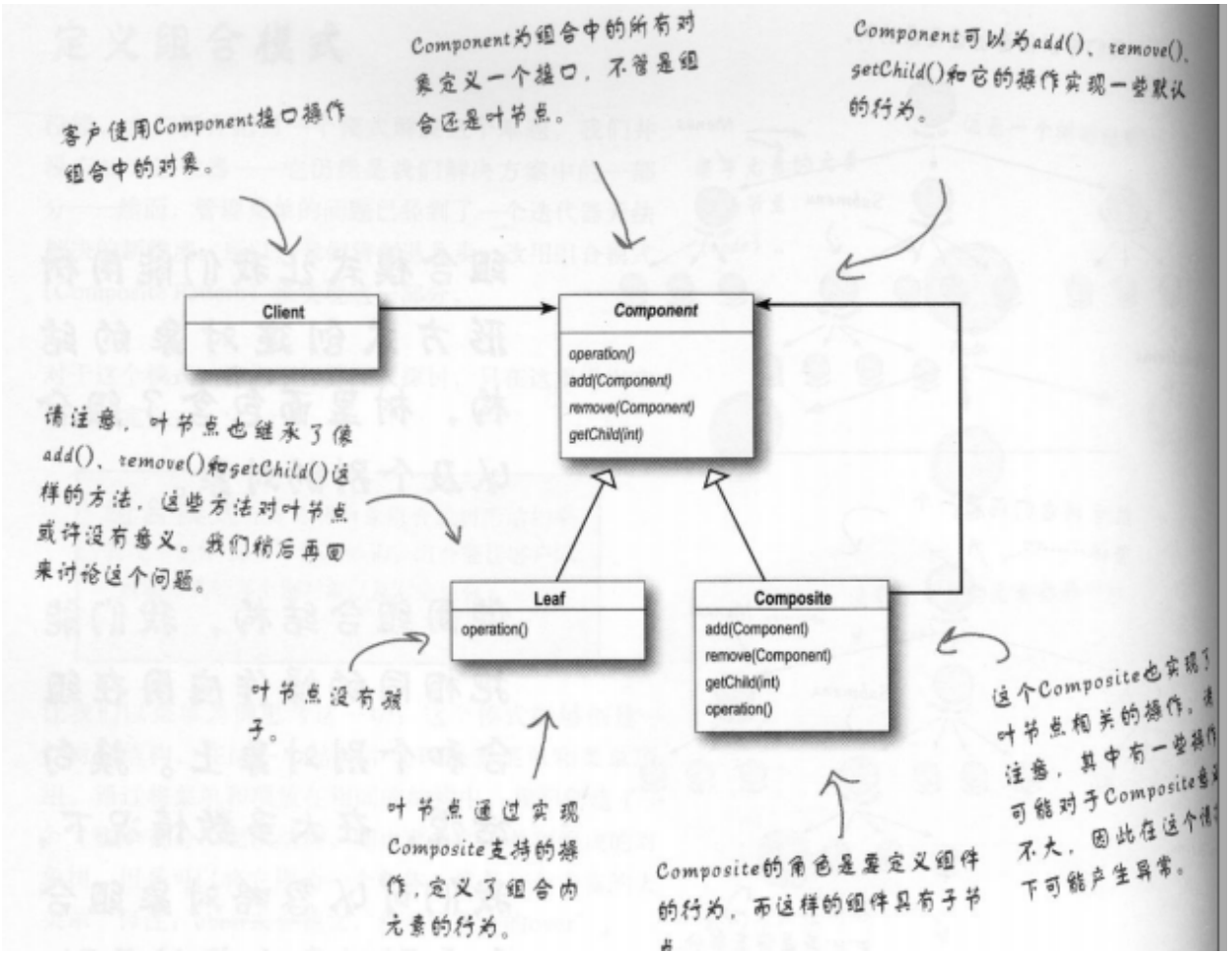


9、管理良好的集合：迭代器与组合模式

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，而不暴露其内部的表示。



组合模式：允许你将对象组成树形结构来表现"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和对象组合。



OO原则：

类应该只有一个改变的理由。

10、事物的状态：状态模式

状态模式：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。

11、控制对象访问：代理模式

代理模式：为另一个对象提供一个替身或占位符以访问这个对象。

代理：

远程代理、虚拟代理、保护代理、防火墙代理、智能引用代理、缓存代理、同步代理、复杂隐藏代理、写入时复制代理。

12、模式中的模式：复合模式

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。

MVC：结合观察者模式、策略模式、组合模式的复合模式。

13、真实世界中的模式：与设计模式相处

模式：在某情境下，针对某问题的某种解决方案。

“力”：塑造并控制解决方案。

模式分类：
创建型：涉及将对象实例化，提供一个方法将客户从所需要实例化的对象中解耦。
行为型：涉及到类和对象如何交互及分配职责。
结构型：把类或对象组合到更大的结构中。
反模式：告诉你如何采用一个不好的解决方案解决一个问题。

14、剩下的模式

桥接模式：不止改变你的实现，也改变你的抽象。适合使用在需要跨越多个平台的图形和窗口系统上。
生成器模式：封装一个产品的构造过程，并允许按步骤构造。适合创建组合结构。
责任链模式：让一个以上的对象有机会能够处理某个请求。适用于 窗口系统中，处理鼠标和键盘之类的事件。
蝇量模式：让某个类的一个势力能用来提供许多"虚拟实例"。适用于一个类的许多实例能被同一方法控制的情况。
解释器模式：为语言创建解析器。适用于创建一个简单的语言。
中介者模式：集中相关对间的复杂的沟通和控制方式。适用于协调相关的GUI组件。
备忘录模式：让对象返回之前的状态。适用于储存状态。
原型模式：创建给定类的实例的过程很昂贵或很复杂时。适用于处理复杂的类层次。
访问者模式：让一个对象的组合增加新的能力，且封装并不重要时。