Java设计模式

抽象类是对事物的抽象，接口是对行为的抽象

分开变化部分和不变化部分，把变化部分封装起来，以便以后可以轻易改动和扩展而不影响不需要变化的其他部分

"系统中某部分改变不会影响其他部分"

良好的OO设计要具备可扩展性、可维护性、可复用

设计模式是被证明的良好的OO设计经验

大多数设计模式和原则都专注于变化的主题

\*面向接口编程,实质是面向超类型编程，变量的声明应该是超类型(抽象类、接口)，具体实现此超类型的所有变量都可以指定给该超类型，声明类时不用理会以后执行的真正对象的类型，这样就能将编写的代码和具体执行时真正的类型是什么相解耦

封装变化

多用组合、少用继承(继承会使两个类耦合)

面向接口编程，不针对实现编程

为交互之间的松耦合设计而努力

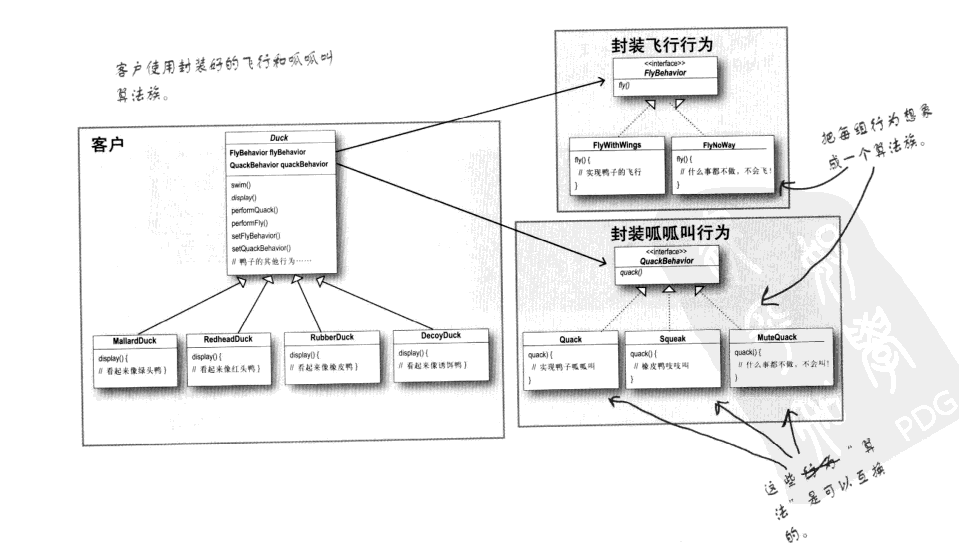
类应该对扩展开放，对修改关闭

依赖倒置原则，依赖抽象，不依赖具体实现，高层组件不依赖低层组件

最少知识原则，减少对象的依赖，不让太多类耦合在一起，避免系统修改一部分而影响另一部分

相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在java中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

策略模式：定义了算法族，把它们封装起来，算法之间可以相互替换，此模式让算法的变化独立于使用算法的客户，客户可以很方便的采用不同的算法，或者在运行时选择不同的算法(策略)

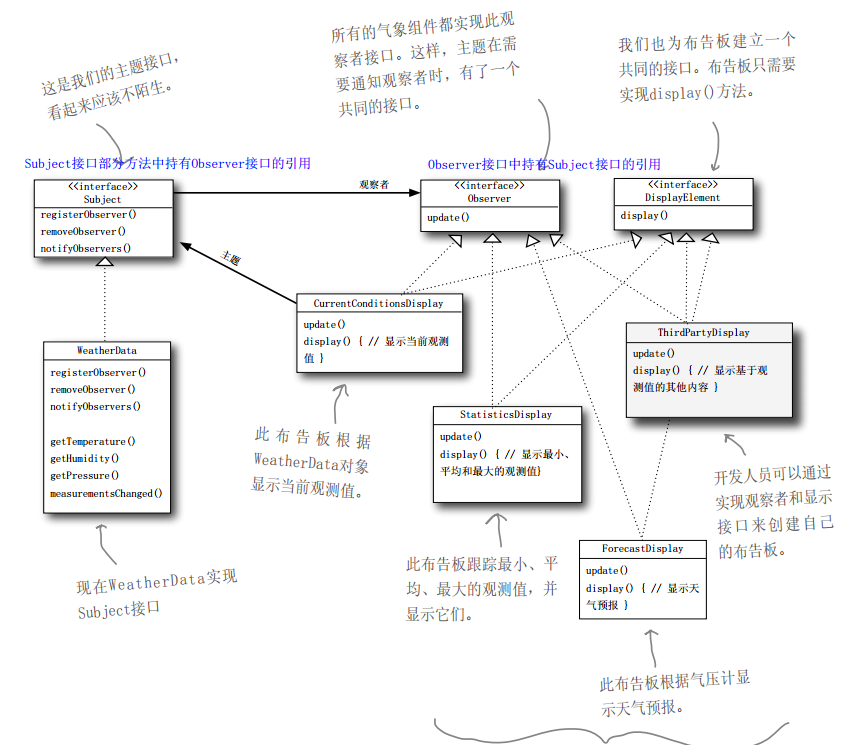


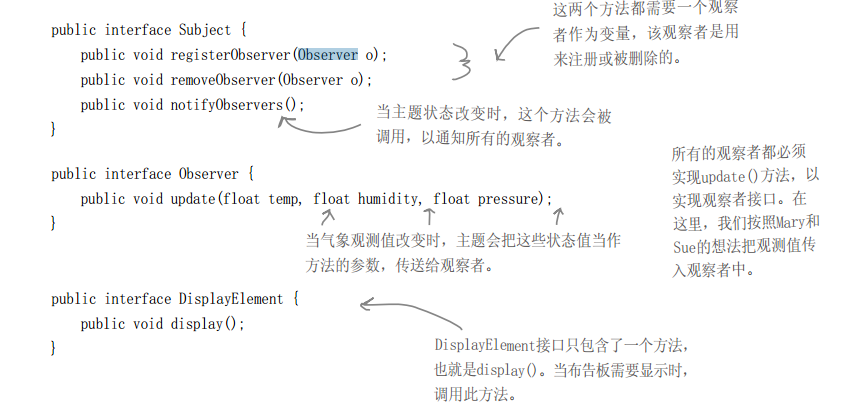
观察者模式：定义了对象(主题)之间的一对多的对应关系，当一个对象的状态发生改变时，他的所有依赖者(观察者)都会受到通知并自动更新

主题中的添加、删除观察者等方法，持有一个Observer接口的引用，可以将传入的观察者添加到某个地方，实现了Observer接口的观察者构造中持有一个Subject接口的引用，该引用用于指定观察者所订阅的主题，通过他调用主题中的添加、删除观察者等方法

观察者和主题(可观察者)之间是一种松耦合(loose-coupling)的方式结合，主题不知道观察者内部实现细节，只知道观察者实现了Observer接口

Swing中大量使用观察者模式，其他GUI中设计也是如此





Java内置的观察者模式包含最基本的Observer接口与Observable(这就是我们以前所称的“主题”Subject，从今以后也可以改称为“可观察者”)类，这和我们的Subject接口与Observer接口很相似

装饰者模式 装饰器 被装饰者 继承和装饰者模式都能够扩展类的功能，装饰者模式提供了比继承更有弹性的替代方案

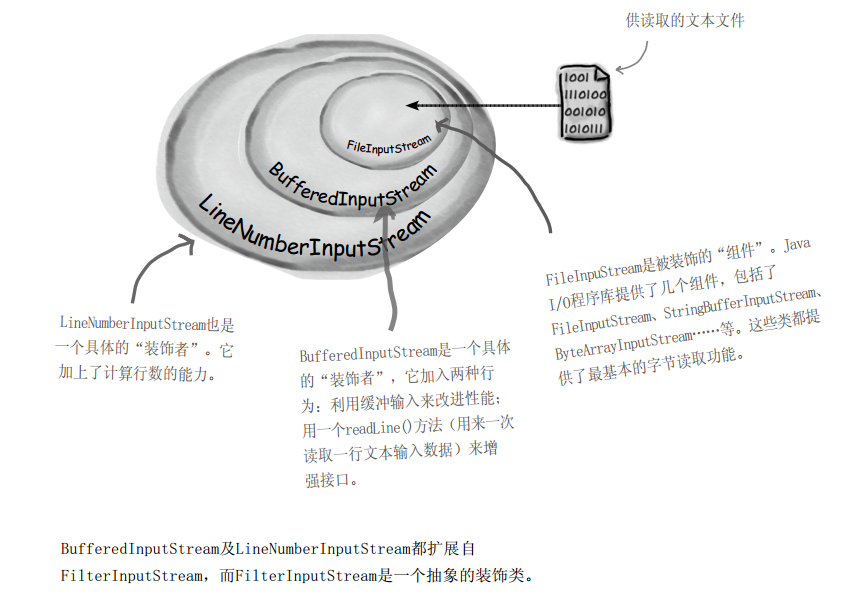
1、装饰器和被装饰者具有相同的接口类型，装饰器不但实现了被装饰者所实现的接口，还持有一个被装饰者这的接口的引用，所有实现了该接口的对象都可以被该装饰者进行装饰，一个对象经过装饰器装饰之后类型不变

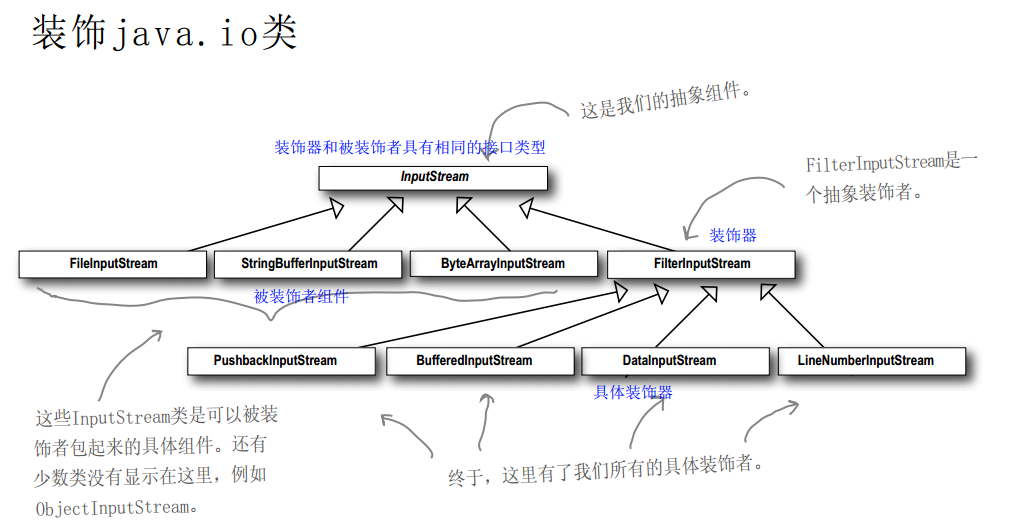
2、装饰着模式适用于功能相似的类的创建过程中，可以在运行时，通过对对象进行装饰，使对象具有更加强大的功能，过程中会产生大量的对象

3、但是通过继承来实现类的功能的扩展,会造成 “类爆炸” 出现大量功能相似的类的出现

4、装饰者模式在Java IO体系中有着很好的运用 FilterInputStream是装饰器的抽象类

5、既然装饰者和被装饰对象有相同的超类型，所以在任何需要原始对象（被包装的）的场合，可以用装饰过的对象代替它。装饰者可以在所委托被装饰者的行为之前与/或之后，加上自己的行为，以达到特定的目的



工厂方法模式

所有工厂模式都用来封装对象的创建，工厂方法让子类来决定该创建的对象是什么，将对象的创建过程封装在子类中实现，这样客户程序中关于超类的代码就和子类对象创建代码解耦了

定义一个抽象的工厂方法，子类必须实现这个方法，依赖于子类来实现对象的创建

在抽象的Creator类中，其他方法可能用到抽象的工厂方法创建的对象(产品)，但是只有子类真正实现这个抽象的工厂方法并创建对象，在抽象的Creator类中编写相关的代码，不需要知道工厂方法创建的对象具体的类型是什么

产品的“使用”与“实现(创建)”相解耦

抽象工厂定义一个接口，里面有一个生产抽象产品的抽象方法，具体的工厂类实现这个抽象工厂接口，重写里面的(生产抽象产品的)抽象方法，来生产具体的产品，因此该模式将对象(产品)的创建封装到子类来实现，由子类决定生产产品的具体类型

也就是抽象工厂定义生产产品，具体的工厂决定生产的产品具体是什么

车工厂用来封装车对象的创建，是一个抽象的工厂，轿车工厂实现这个工厂接口，成为具体的工厂，用来生产轿车，卡车工厂实现这个抽象的工厂，成为具体的卡车工厂，用来生产卡车，也就是具体的子类工厂决定生产对象的类型

各自工厂生产各自具体的产品

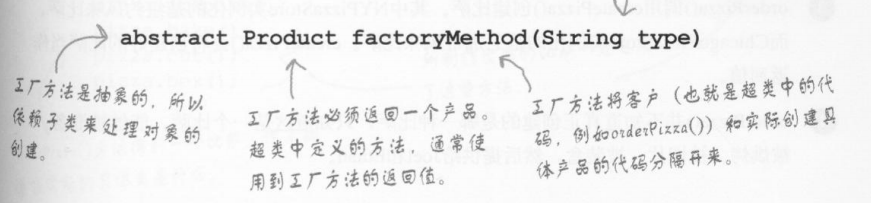
1.抽象产品：产品对象同一的基类，或者是同一的接口。

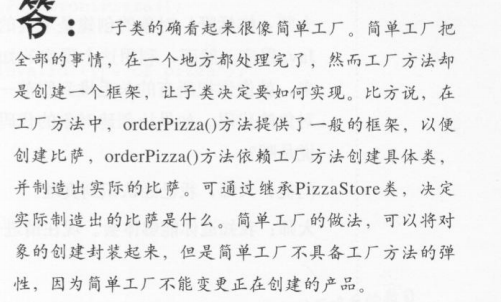
2.具体的产品：各个不同的实例对象类

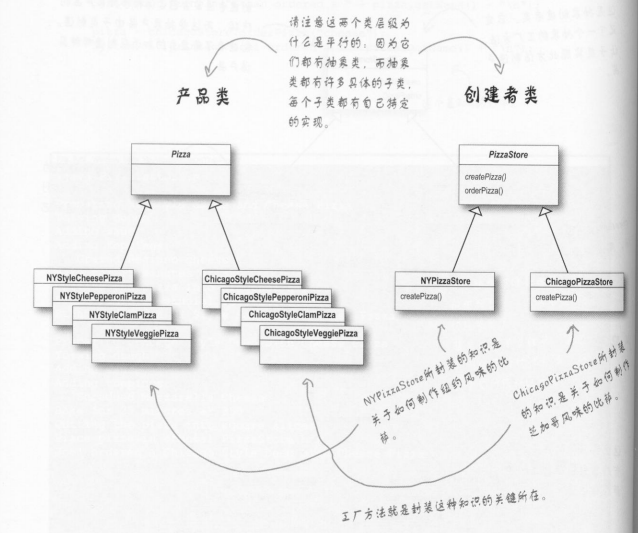
3.抽象工厂：所有的子类工厂类的基类，或是同一的接口

4.具体的工厂子类：负责每个不同的产品对象的实际创建









设计原则：依赖于抽象，不依赖于具体类，高层组件不依赖于低层组件，无论是高层组件还是低层组件，都依赖于抽象

依赖倒置：要创建一个pizza店，创造出各种各样pizza，这样高层组件pizza Store会依赖于具体的pizza实现，每当添加一种新的pizza或者下架一款pizza，pizza store就得修改。反过来思考，现在从各种各样的pizza中抽象出pizza抽象，让具体的pizza类都依赖与这个pizza抽象，高层组件pizza store也不再和具体的pizza类打交道，而是直接依赖于pizza抽象，现在满足了高层组件pizza store不依赖于低层组件具体的pizza(现在是低层组件依赖于高层组件)，无论是高层组件还是低层组件，都依赖于pizza抽象

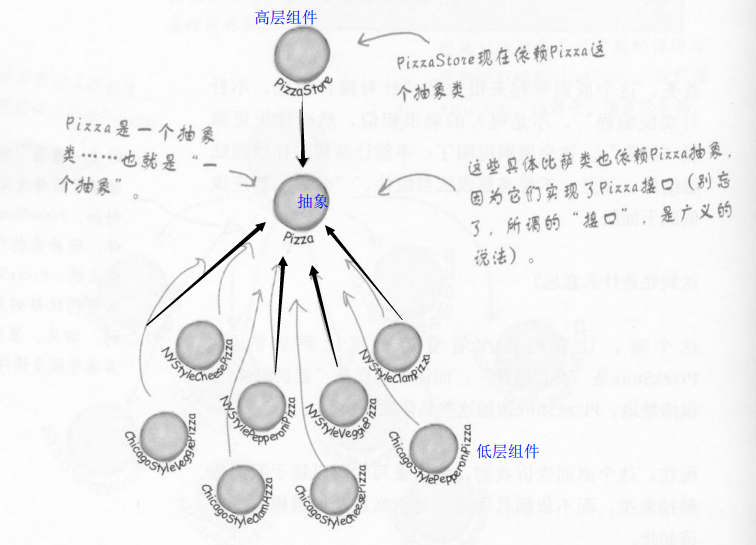
高层组件就是其行为是由低层组件来定义的类，pizza store是高层组件，其行为由pizza(低层组件)定义

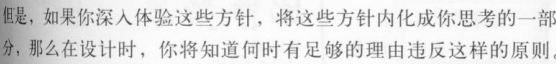
工厂方法与抽象工厂

工厂模式都用来封装对象的创建

工厂方法创建对象，要扩展一个类，并覆盖它的抽象工厂方法，让子类来决定创建哪个具体类（将对象的创建交给子类来实现），客户只需要知道所使用的抽象类型就可以了，而子类负责创建具体类型，工厂方法将产品的使用和产品的创建相解耦

抽象工厂提供一个用来创建一个产品家族的抽象类型，这个类型的子类定义了产品被生产的方法，要想使用这个工厂，必须先实例化它，然后将其传入一个针对抽象类型缩写的代码中





“了解规则，是为了更好的打破它！”

单例模式

在代码中只有类的一个实例，并提供全局访问入口

饿汉式：

/\* JVM在加载该类的时候，就会创建该类的对象，保证在任何线程访问静态变量instance之前

\* 存在该类的实例，如果该类的实例没有被使用，会造成空间浪费

\* \*/

**public** **class** Singleton {

**private** **static** Singleton *instance* = **new** Singleton();

**private** Singleton(){

}

**public** **static** Singleton getInstance(){

**return** *instance*;

}

}

懒汉式：

/\*同步方法保证线程安全，判断instance是否为null如果为null，创建对象并返回

否则直接返回原来创建的对象，同步方法导致效率低下，实际上只有第一次执行该方法的时候需要同步，以后每次调用该同步方法都是一种累赘

\* \*/

**public** **class** Singleton {

**private** **static** Singleton *instance* = **null**;

**private** Singleton(){

}

**public** **static** **synchronized** Singleton getInstance(){

**if**(*instance* == **null**)

*instance* = **new** Singleton();

**return** *instance*;

}

}

双重if嵌套：

/\* 双层if嵌套能保证线程安全且性能高效

只有instance为null的时，才进入同步代码块，同步代码块只会在第一次被执行一次

\* 进入同步代码块，再次判断，如果为null，创建实例返回

\* 再次if判断，可以避免instance还未创建，且有多个线程同时进入\*时，一个线程创建了Singleton实例，但是另一个线程还不知道的问题

\* 用volatile修饰instance，保证instance被初始化Singleton实例的时候，多线程能正确处理

\* \*/

**public** **class** Singleton {

**private** **volatile** **static** Singleton *instance* = **null**;

**private** Singleton(){

}

**public** **static** Singleton getInstance(){

**if**(*instance* == **null**){ //\*

**synchronized**(Singleton.**class**){

**if**(*instance* == **null**)

*instance* = **new** Singleton();

}

}

**return** *instance*;

}

}

静态内部类实现单例

/\* 静态内部类实现单利模式 线程安全、实现延迟加载

\* 外部类初始化的时候不会初始化内部类，只有当调用getInstance方法时候才会初始化

\* \*/

**public** **class** Singleton{

**private** Singleton(){

}

**private** **static** **class** Inner{

**private** **static** Singleton *instance* = **new** Singleton();

}

**public** **static** Singleton getInstance(){

**return** Inner.*instance*;

}

}

关于多个类加载器的问题

每个类加载器都定义了一个命名空间，若程序中又多个类加载器，一个类会被加载多次，使用单例模式，会出现问题，解决办法是，要指定同一个类加载器加载

单例模式的继承？

单例模式构造器是私有的，私有构造不能用来扩展类，因此如果构造器定义为public或者protected权限，就不算真正的单例了，因为其他类也可以实例化它

全局变量与单例模式

全局变量在Java中就是对对象的静态引用，全局变量虽然能够提供全局访问入口，但是不能保证只有一个某类的实例

适配器模式

将一个类的接口，经过适配器包装之后转换为客户期望的另一个接口，适配器可以让两个接口不同的类相互沟通

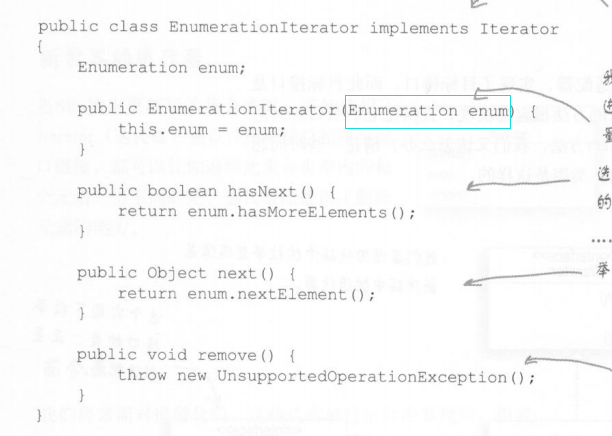
适配器实现了目标接口，并持有一个被适配对象的实例，适配器将收到的方法的调用委托给持有对象实例的相关方法的调用

类适配器和对象适配器

对象适配器通过组合(适配器持有一个被适配对象的实例)实现，能够适配某类和该类的任何子类

类适配器通过继承实现，不需要重新实现整个适配器，必要时还可以重写被适配者的行为，因为类适配器是通过继承来实现的(继承了目标了，同时也继承了被适配者)

Java已经采用了迭代器接口，但是还有大量遗留代码采用枚举接口，因此通过适配器将枚举接口转换为迭代器接口来使用

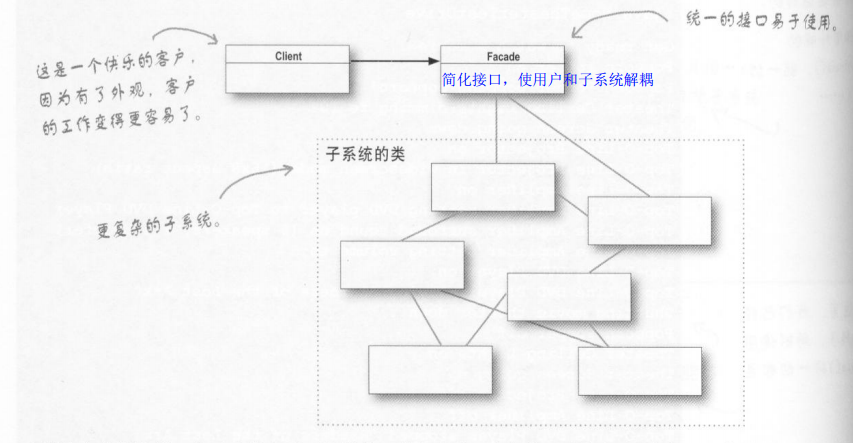


外观模式

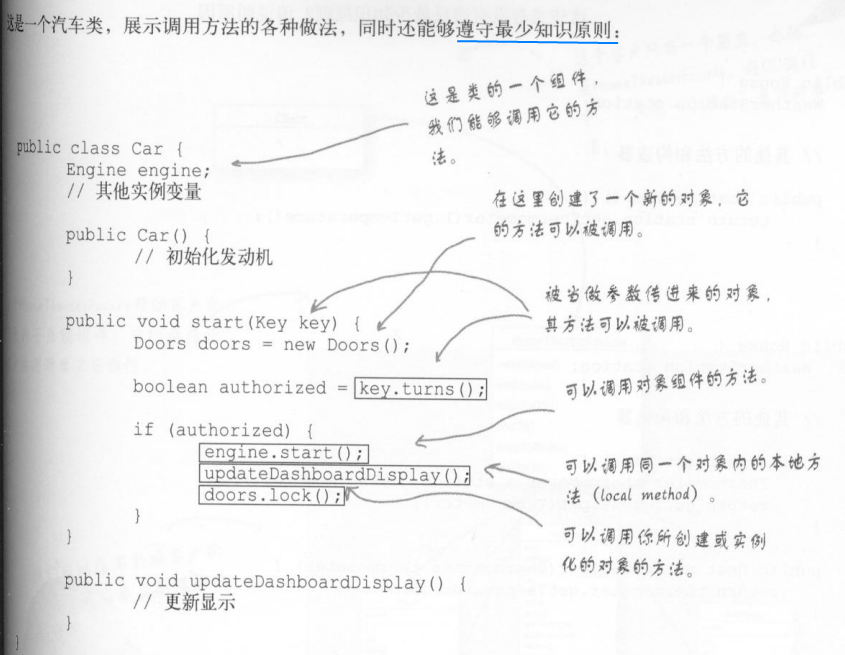
用于简化接口，将客户从子系统中解耦合，创建一个接口简化且统一的类，包装子系统一个或多个复杂的类，提供一个接口，去访问子系统中的一群接口，外观定义了一个高层接口，使子系统更加易于使用

适配器模式和外观模式

两种模式都可以包装多个类的，差别不在于包装了几个类，而在于他们真实的意图，适配器模式转换接口符合客户期望，外观模式是提供子系统的一个简化的接口，将客户从子系统中解耦合



将方法调用保持在界限内符合最少知识原则



模板方法设计模式 提供了一种复用代码的重要技巧，将重复使用的代码都放到超类中给子类共享，一部分用于子类自己实现

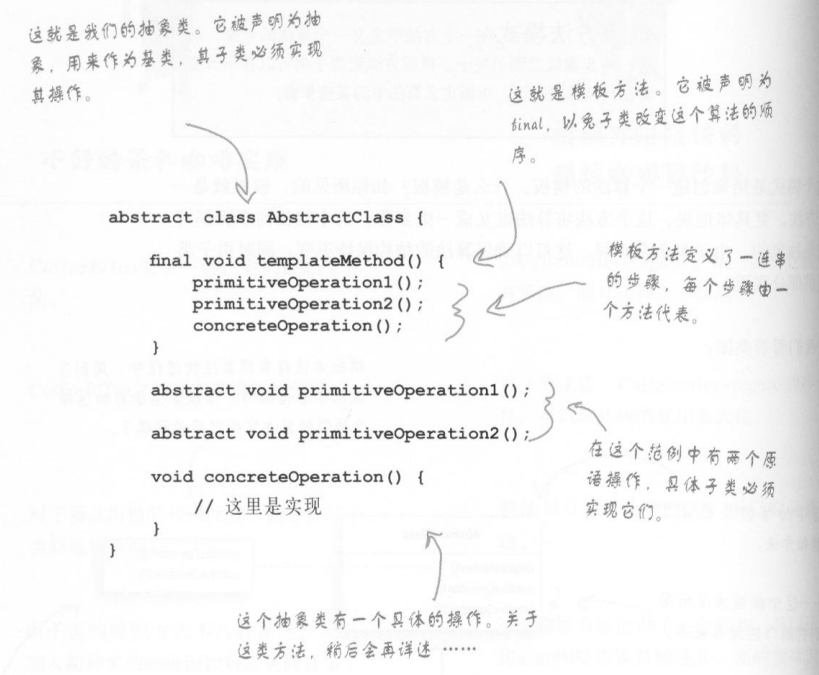
模板方法模式在一个方法中定义了一个算法骨架，将该算法的某些步骤延迟到子类来实现

模板方法模式允许子类在不改变算法骨架的情况下，重新定义算法中的某些步骤

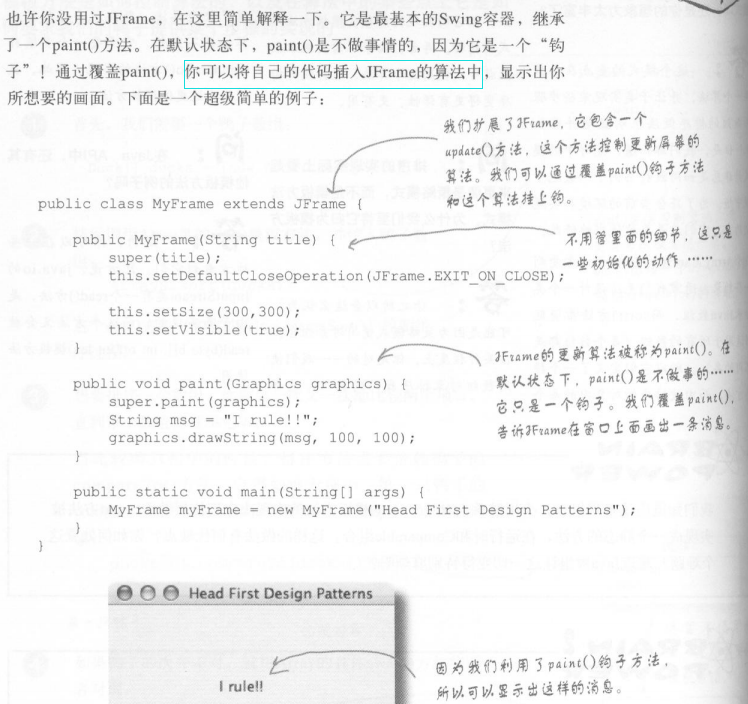
一般类定义为抽象类，模板方法定义了一个算法骨架，一般是一系列方法的集合，定义为final，这样子类就不能更改算法执行的顺序

将需要子类直接继承的定义为final，需要子类进行重写的定义为抽象方法，此外还有一个钩子的概念，让钩子实现算法中的可选部分，它能够影响算法的执行，可以根据需要进行选择

模板方法的抽象类中可以定义具体方法，抽象方法，以及钩子

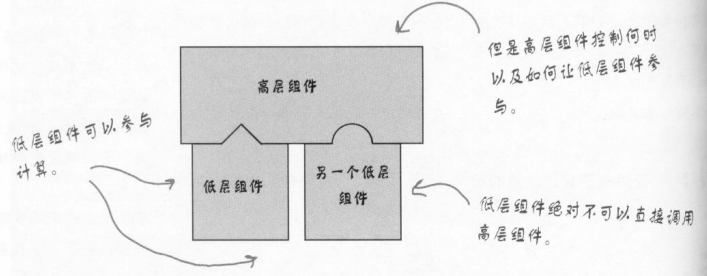


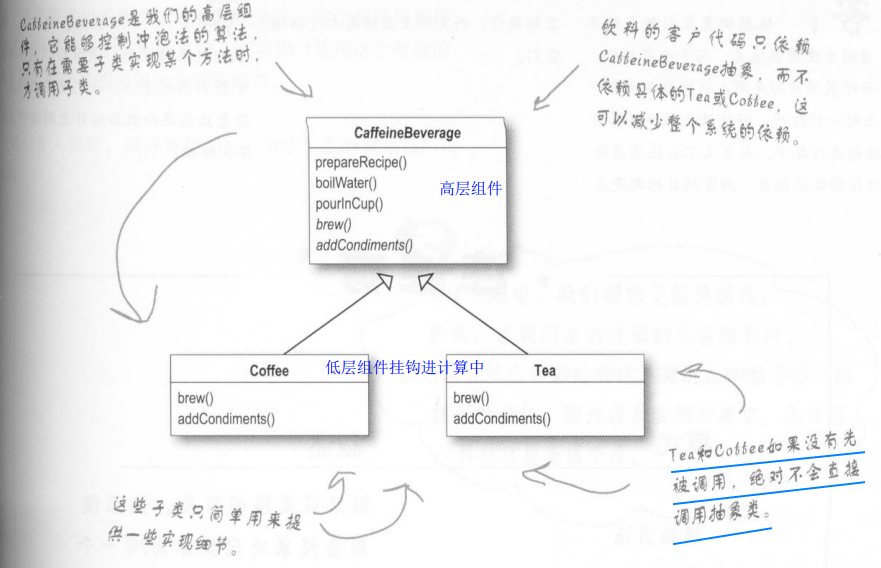
钩子



好莱坞原则 不要找我，我会找你

依赖倒置原则，依赖抽象，不依赖具体，好莱坞原则是在创建框架和组件上的一种技巧，让低层组件能够挂钩进计算中，而且又不会让高层组件依赖于该低层组件，两者的目标都是在于解耦





模板模式和策略设计区别

模板模式定义了一个算法骨架，将其中某些步骤延迟到子类来实现，子类决定这些步骤的实现细节，整体算法骨架不变

策略模式封装一个算法族，这些算法之间可以相互替换，每个算法都被封装起来了，因此客户可以很方便的采用不同的算法，或者在运行时选择不同的算法(策略)

模板算法用于排序

对象的数组

Arrays.sort(arr);Arrays调用模板方法sort，传入待排序的数组

需要一次又一次的比较两个对象，直到整个数组排序完毕

实现Comparable接口，使类具有可比较属性，自定义的类不像数值那样具有可比较属性，因此类要实现Comparable接口，重写compareTo方法，按照方法中规定的进行比较，该方法需要传入另外一个对象，和这个对象本身进行比较

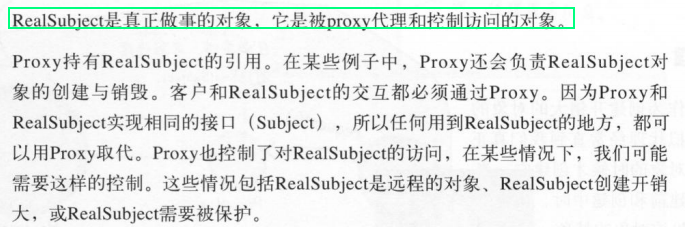
代理模式 创建代表对象，让代表对象控制对某对象的访问

所谓代理就是用来代表某个真实的对象，为一个对象提供一个替身或占位符，控制对该对象的访问，代理模式有很多变体，不同的代理都和控制访问有关

远程代理，远程对象的本地代表，控制访问远程对象，处理网络上的细节

虚拟代理，控制访问创建开销大的资源

保护代理，基于权限对资源的访问

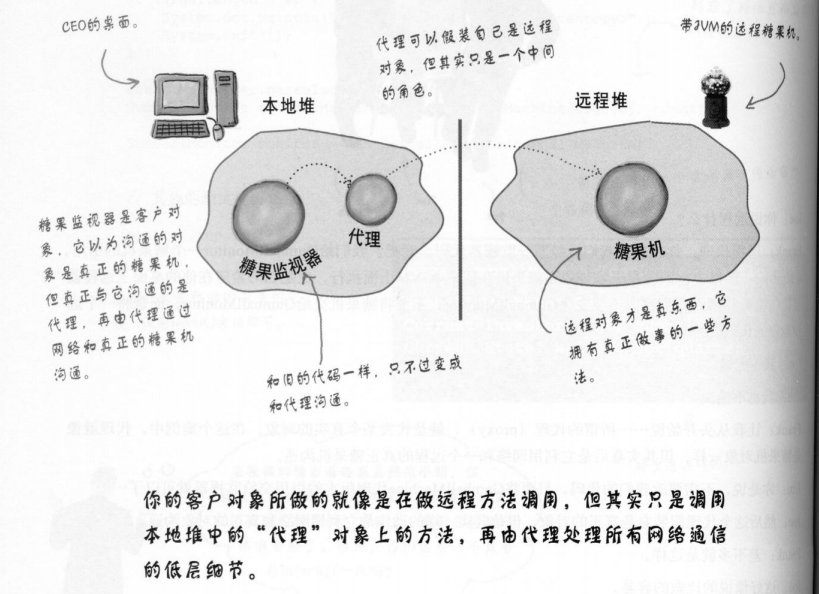


远程对象，运行在不同JVM堆中(在不同的地址空间运行的远程对象)

Duck d = <变量d只能引用和该代码在同一个堆空间的对象>

RMI Remote Method Invoke 远程方法调用

RMI提供了客户辅助对象和服务辅助对象，为客户辅助对象创建和服务对象相同的方法，RMI好处是不必写任何网络和IO代码，客户程序调用远程方法(真正的服务所在)跟运行在本地JVM上对对象正常方法调用一样





远程方法的变量和返回值，必须属于原语类型或者Serializable类型(实现某个接口可以转化为接口类型)，远程方法变量必须被打包并通过网络传输，这需要序列化来完成，原语类型、字符串，以及API中内定的类型(包括数组和集合)都可以被序列化，如果自定义类型，要实现序列化接口

Remote接口以及Serializable接口都是标记接口，接口中没有方法表示实现了这种接口的类具有接口规定的属性

远程调用:

1、制作远程接口

**public** **interface** MyRmote **extends** Remote {

**public** String sayHello() **throws** RemoteException;

//Remote接口是支持远程调用的接口，所有方法都要抛出RemoteException异常

且返回值是原语类型或者实现了Serializable接口的类型

}

2、制作远程服务：

/\* 远程实现类要扩展UnicastRemoteObject类，并实现远程接口MyRemote

\* UnicastRemoteObject(是超类)有抛出异常的构造函数，因此继承该类的子类要有抛出异常的构造

\* \*/

**public** **class** MyRemoteImpl **extends** UnicastRemoteObject **implements** MyRmote {

**protected** MyRemoteImpl() **throws** RemoteException {

**super**();

}

@Override

**public** String sayHello() { //所有方法不抛出异常

**return** **null**;

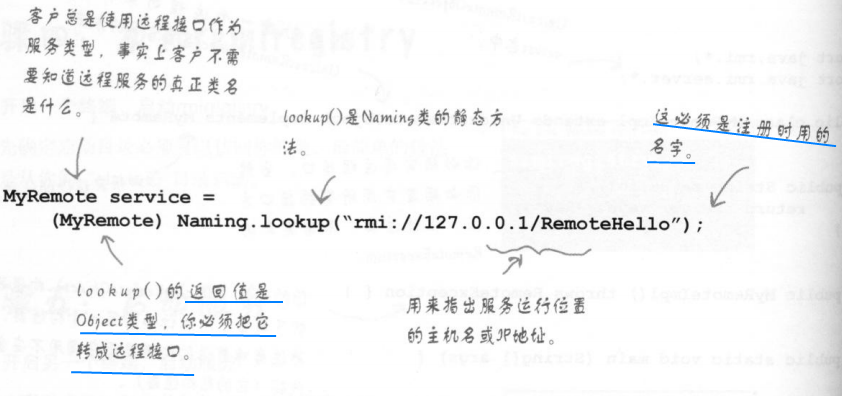
}

}

3、用RMI Registry注册远程服务



4、客户获取stub对象，需要用到RMI Registry中寻找stub





Java设计六大原则

单一职责原则

开闭原则

里氏替换原则

迪米特原则(最少知识原则)

接口隔离原则

依赖倒置原则