大话设计模式

# 总结

抽象是核心

1.对类型的抽象,某一类型,可以是外观+行为+属性的一群具体的对象的抽象,

可以是某一功能或是某一功能使用的一群对象的抽象;

抽象的表现就是具体的实现者拥有相同的行为和属性;

2.对于功能的抽象,提炼出一个功能的抽象,功能和类型之间,只有抽象层面的交流;具体的功能和具体的类型之间无关联,通过里氏代换原则;

3.抽象---抽象

4.一个类或一个功能或一个模块用什么样的方式什么样的思维编写出来的代码,通过在客户端使用它们的方式就可以看出构造过程;

5.代码的责任分解;

# 面向对象

通过规划业务逻辑（包含未来即将使用到的），构建一个可归纳主干，可扩展子业务，可修改主业务和子业务，可灵活删除业务的系统；

脱离具体的业务逻辑，从多种系统使用情况，归纳业务逻辑，根据此抽象出的业务逻辑，创建可归纳此业务逻辑的系统，或是模块；

根本目的：活字印刷术

提高工作效率，通过重复利用工作成果；将劳动成果分割开来；

1. 可维护：在未来修改内容的时候可以在原有基础上进行修改，达到重复使用部分劳动成果的作用；

采用面向对象思想设计的结构，可读性高，由于继承的存在，即使改变需求，那么维护也只是在局部模块，所以维护起来是非常方便和较低成本的。

1. 可复用：将生产的劳动成果重复的使用；

在以前的项目的领域中已被测试过的类使系统满足业务需求并具有较高的质量。

1. 易扩展：当分割后的劳动成果无法满足当前需求的时候，可以添加同类的劳动成果，达到要求；

由于继承、封装、多态的特性，自然设计出高内聚、低耦合的系统结构，使得系统更灵活、更容易扩展，而且成本较低。

1. 灵活性好：在根据需求进行使用的时候非常方便，修改简单；在修改需求时尽量减少代码的修改；
2. 效率高

在软件开发时，根据设计的需要对现实世界的事物进行抽象，产生类。使用这样的方法解决问题，接近于日常生活和自然的思考方式，势必提高软件开发的效率和质量。

通过封装 继承 多态把程序的耦合度降低；

1. 业务的封装：

将不同的业务封装到不同的类或方法中，而不是混乱的交织在一起；

1.1 良好的封装能够减少耦合

1.2 类内部的实现可以自由修改

1.3类具有清晰的对外接口；

1. 继承

两个对象a b，如果说b是a，那么b可以集成a；

is a的关系，这种关系是不能颠倒的，b除了集成了a，还拥有自己的具体的定义；

如果子类继承父类1.子类拥有父类非private的属性和功能；2.子类具有自己的属性和功能；3.子类对父类的方法的重写；4.构造方法不能被继承，可以被调用；

优点1.公共部分放在子类，实现代码共享；2.使修改和扩展变得轻松；

缺点 1.父类变，子类不得不变；2.继承破坏包装，父类实现细节暴露给子类，增加了耦合性，建立强耦合关系；

1. 多态

表示不同的对象可以执行相同的操作，但要通过他们自己的实现代码来执行；

几点注意：1.子类以父类的身份出现；2.子类在工作的时候以自己的方式实现；3.子类以父类的身份出现时，子类特有的属性和方法不可以使用；4.使用override关键字重写；

多台原理：当方法被调用时，无论对象是否被转换为其父类，都只有位于对象继承链最末端的方法实现会被调用。也就是说虚方法是按照其运行时类型而非编译时类型进行动态绑定调用的。

紧耦合 vs 松耦合

根据需求和业务逻辑，将同类业务适当细分为子业务，变为可扩展，可修改（不影响其他子业务），可变更运算顺序；

高内聚，松耦合：

对象：对象是一个自包含的实体，用一组可识别的特性和行为来标识；

类：具有相同的属性和功能的对象的抽象的集合；

实例：就是一个真实的对象；

实例化：就是创建对象的过程，用new关键字；

构造方法：对类进行初始化，与类同名，无返回值，在new的时候调用；

Oop :针对对象来编程；

方法重载：

属性与修饰符：

重构：在原有的开放的接口上，进行私有的逻辑的改动，甚至是类的业务逻辑的改动；

抽象类：1.抽象类不能被实例化;2.抽象方法是必须被子类重写的方法;3.如果类中包含抽象方法,那么类就必须定义为抽象类,不论是否还包含其他一般方法;4.抽象类拥有尽可能多的共同的代码,拥有尽可能少的数据;

抽象类在一个以继承关系形成的等级结构里面，树叶节点应当是具体类，而树枝节点均英但是抽象类

接口：接口是把隐式公共方法和属性组合起来，以封装特定功能的一个集合。一旦类实现了接口，类就可以支持接口所指定的所有属性和成员。声明接口在语法上与声明抽象类完全相同，但不允许提供接口中任何成员的执行方式。

抽象类和接口的比较：

形态上区分

抽象类 一些成员实现，接口不行

抽象类的成员可被子类部分实现，接口需要完全实现

一个类只能继承一个抽象类，但接口可以有多个

1. 类是对对象的抽象 ；抽象类是对类的抽象； 接口是对行为的抽象；
2. 如果行为跨越不同类的对象，可使用接口；对于一些相似的类对象，用继承抽象类
3. 从设计角度上讲，抽象类是从子类中发现了公共的东西，泛化出父类，然后子类继承父类，接口是根本不知道子类的存在，方法如何实现还不确认，预先定义；
4. 接口就像比赛中举办方的比赛项目制定，比什么；抽象类就像对参赛者进行分类抽象；

总结：

对同种类型的对象抽象，对同一个业务需求的抽象，对同类运算的抽象；对同一模块的抽象；

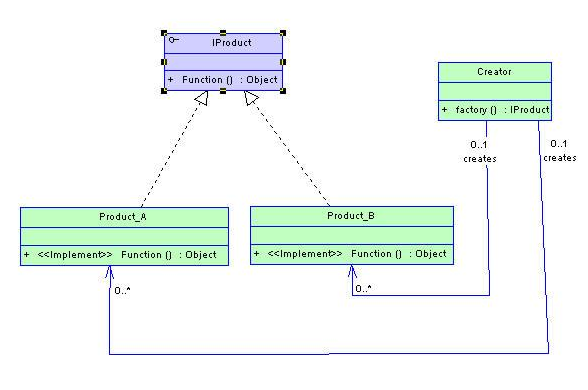
对各种抽象的扩展方便，对某一抽象的继承的增加 修改 删除的方便；

减少类与类之间的耦合，彼此之间的沟通更像是通过一个代理或接口来完成；也可以是某种消息机制；

# 简单工厂模式

根本目的：减少类的种类，从而避免在使用者的内部过多的牵扯这些类，不违背灵活性好的要求；对使用者来说，工厂产生的对象，使用方法一致；

根据传入的定义好的规则参数，返回不同的对象；对于不同的对象，对外界有相同的操作接口，也可由自己公开的操作方法和属性；



使用方式：

使用类创建一个工厂，向工厂传入不同的参数，工厂创造不同的产品，返回接口类型；

# 策略模式

根本目的：使业务易扩展，高复用，灵活性好，可维护

案例：收银台促销手段的实现；

他定义了算法家族，分别封装起来，让他们之间可以相互替换，次模式让算法的变化，不会影响到使用算法的客户。

将满足某业务的各种操作方式，看作一个需求；用策略模式管理这种需求；

结构图

Context：结合简单工厂模式，使业务模块与调用者减少类之间的耦合；暴露给调用者；

Strategy:抽象类，将各种具体算法或行为抽象出来。

Strategya:某具体算法，

优点

1. 策略模式是一种定义一系列算法的方法，从概念上来讲，所有这些算法完成的都是相同的工作，只是实现不同，它可以以相同的方式，调用所有的算法，减少了各种算法与使用算法类之间的耦合
2. Strategy类层次为context定义了一系列的可重用的算法或行为。继承有助于析取出这些算法的公共功能。
3. 简化测试单元

# 单一职责原则

根本目的：增加重复使用的几率，增加代码的可维护性，

案例：手机集合了多种功能，摄像不清晰

定义：就一个类而言，应该仅有一个引起他变化的原因；如果能够想到多余一个动机去改变一个类，那么这个类就具有多个职责；

优点：

1. 一个类承担的责任过多，当修改一个职责时，可能会消弱或抑制其他职责的能力。
2. 当需求发生变化后，设计会遭到意想不到的破坏；
3. 软件设计真正要做的许多内容，就是发现职责并把那些职责相互分离。

# 开放-封闭原则

案例：计算器增加乘法除法的计算逻辑等；

香港回归，一国两制；在不改变大陆的政治前提，增加香港的制度；

定义：软件实体（类，模块，函数等等）应该可以扩展，但是不可修改。

1. 对于扩展是开放的，对类结构和新的业务逻辑进行重构和扩展；
2. 对于更改是封闭的，原先的业务逻辑和方法等不必修改

在最初编写代码的时候，假设变化不会发生。当变化发生时，我们就创建抽象来隔离以后发生的同类变化；

面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。

我们希望的是在开发展开不久就知道可能发生的变化。查明可能发生的变化所等待的时间越长，要创建正确的抽象就越困难。

开放-封闭原则是面向对象设计的核心所在。遵循这个原则可以带来面向对象技术所声称的巨大好处，也就是可维护，可可扩展，可复用，灵活性好开发人员应该对程序中呈现出频繁变化的那些部分做出抽象，然而，对于应用程序中的每个部分都刻意的进行抽象同样不是一个好主意。拒绝不成熟的抽象和抽象本身一样重要。

这里的需求，是功能上的增加，对于原有业务逻辑上的修改或是优化，不必再增加新的结构；

比如计算器，原先只有加法，后来增加减法，乘法等，那么原先的类结构就要重构，添加抽象类，统一处理计算这个需求；

# 依赖倒转原则

案例：修收音机和修电脑（组装）的区别

A.高层模块不应该依赖低层模块。两个都应该依赖抽象；

主板与内存，cpu等的关系；主板依赖具体的内存，cpu，结果就是，当cpu ，内存需要更改时，主板完全不能兼容新的内存，导致高层模块主板不能使用；相反，如果低层模块只能在某一具体的主板上使用，当主板更换后，内存不能使用；

依赖抽象，就是把相互之间的关系转为抽象的行为规范的问题，统一行为规范，相互之间便不再依赖。

B.抽象不应该依赖细节，细节应该依赖抽象；

设计抽象是首要，细节实现符合抽象去做；

针对接口编程，不针对实现编程。不是针对外界某一具体的使用环境来设计模块或是类，而是针对同一接口来编写模块或类，如果没有接口，那么事先定义接口；

# 里氏代换原则

一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且他察觉不出父类对象和子类对象的区别。也就是说，在软件里面，把父类都替换成他的子类，程序的行为没有变化。

父类才能真正被复用，并且子类可以扩展自己。

由于子类类型的可替换性，才使得使用父类类型的模块在无需修改的情况下就可以扩展。

所以，高层模块不依赖与低层模块；高层模块使用的是抽象，在不同的运行环境中，可以使用不同的低层模块，通过里氏代换原则，传入不同的子类对象。

针对抽象编程而不是针对细节编程，即程序中所有的依赖关系都是终止于抽象类或者接口，那就是面向对象的设计，反之那就是过程话的设计了。

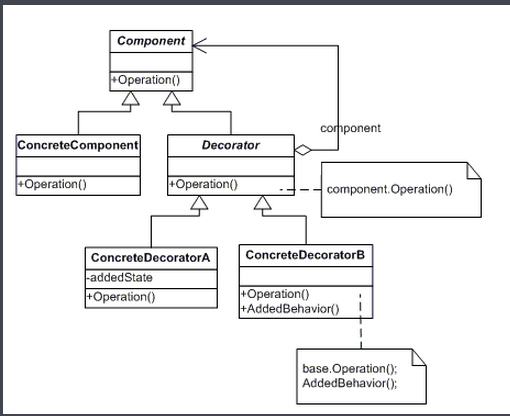
# 装饰模式 Decorator Pattern

设计原则：多用组合，少用继承

根本目的：扩展对象的功能，但是Decorator可以提供比继承更多的灵活性。

使用范围：动态的给类添加新功能，不是核心功能；

图册：



组成：

使用方式：

IthirdParty thirdPartyOne =new ThirdParty();  
　　IthirdParty decorator1 =new Decorator1(thirdPartyOne);  
　　IthirdParty decorator2 =new Decorator2(decorator1);

System.out.println(decorator2.sayMsg());

保存一个有一定顺序的对象链，这个对象链从某个concretecompent开始，到某个concreteDecorator结束；

优点：见百度百科

缺点：

在不必改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。

# 代理对象

根本目的：

对某些不能直接操作的对象提供代理对象，代理对象处理一些权限方面的问题

提供不能及时出现并使用的对象的代理，暂时替代；大图片的 加载，网络的延迟等

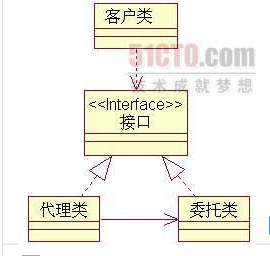
使对象集中注意力到自己的专业职责上，其他交给代理；

组成：

抽象角色：通过接口或抽象类声明真实角色实现的业务方法。

代理角色：实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。

真实角色：实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用。



优点：

使用方式：

定义一个一致的行为,客户类使用代理类操作,代理类处理委托类,将委托类封装起来

# 工厂方法模式

根本目的：工厂方法模式（FACTORY METHOD）是一种常用的对象创建型设计模式,此模式的核心精神是封装类中不变的部分，提取其中个性化善变的部分为独立类，通过依赖注入以达到解耦、复用和方便后期维护拓展的目的。

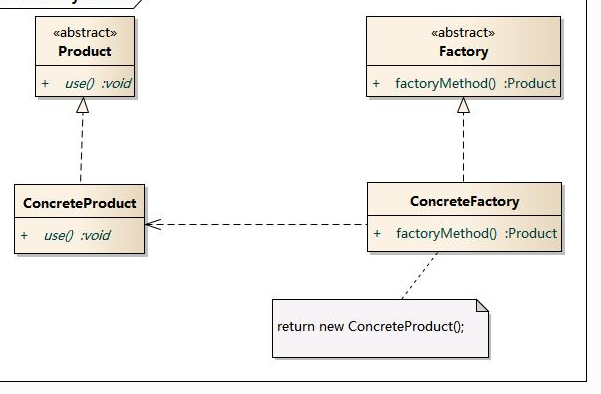
主要组成：它的核心结构有四个角色，分别是抽象工厂；具体工厂；抽象产品；具体产品；

抽象工厂(Creator)角色：是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象的工厂类必须实现这个接口。

具体工厂(Concrete Creator)角色：这是实现抽象工厂接口的具体工厂类，包含与应用程序密切相关的逻辑，并且受到应用程序调用以创建产品对象。在上图中有两个这样的角色：BulbCreator与TubeCreator。

抽象产品(Product)角色：工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。在上图中，这个角色是Light。

具体产品(Concrete Product)角色：这个角色实现了抽象产品角色所定义的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应。



使用方法：

在批量生产对象的时候

1.简单工厂模式只有一个工厂类,所有的对象都在这里生产;添加新产品时,需要改动工厂类,生产的产品亦不便管理;

2.工厂方法模式,将每个产品的生产指定给具体的生产工厂,在开发时更容易维护和扩展;

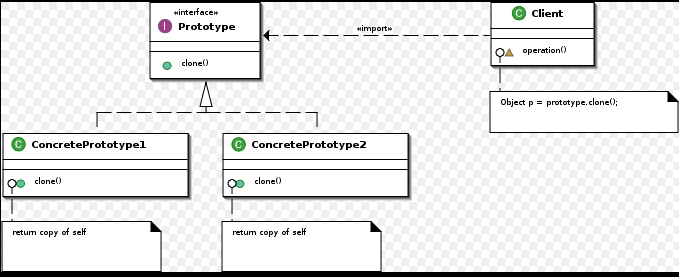
雷锋工厂模式

# 原型模式

范例:打印多分不同信息的个人简历;

根本目的:

在创建相同类对象的时候,不用重新初始化对象,而是动态的获得对象运行时的状态,隐藏对象创造细节,性能大大提高;



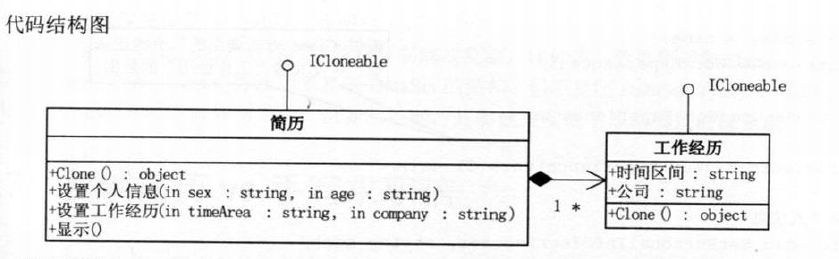
使用方式:

浅度复制

如上图,直接clone,克隆方法可能是语言已定义的,也可能是自定义的,返回克隆对象;

不复制保存对象引用的引用对象;

深度复制:



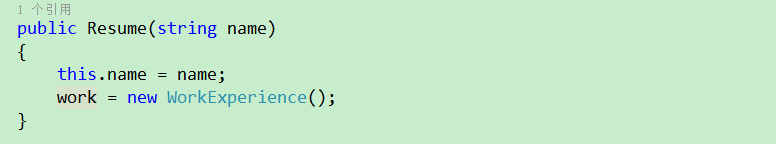
1.定义接口

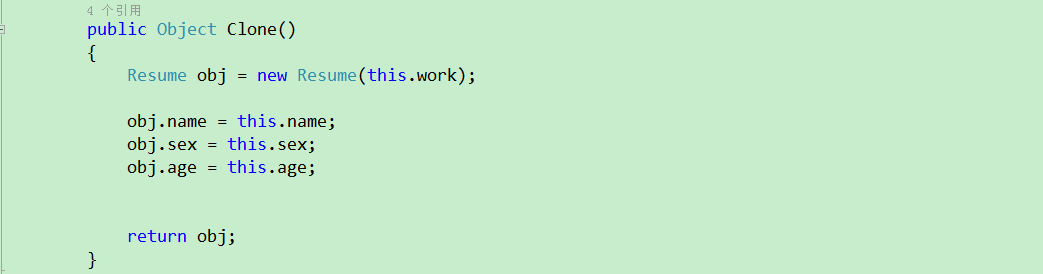
2.首先创建带有基本信息的类实例,

3.再创建带有不同工作经历的基本基本信息的类实例;



clone的时候是创建新对象;





# 模版方法模式

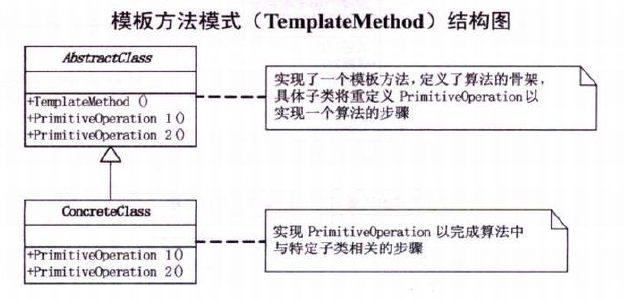
范例:

每个学生的大卷的方式一样,但题目的答案给出的可能不同;重写每个题目答题的内容;

根本目的:

把所有子类中重复的代码搬到父类中,去除子类的重复的代码

结构图:



使用方法:

每个子类重写属于自己的特定行为的方法;

# 迪米特法则

就是说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

根本目的: 迪米特法则的[初衷](http://baike.baidu.com/view/406303.htm)在于降低类之间的[耦合](http://baike.baidu.com/subview/156245/156245.htm)。由于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。

迪米特法则不希望类之间建立直接的联系。如果真的有需要建立联系，也希望能通过它的[友元类](http://baike.baidu.com/subview/8502539/8443743.htm)来转达。因此，应用迪米特法则有可能造成的一个后果就是：系统中存在大量的中介类，这些类之所以存在完全是为了传递类之间的相互调用关系——这在一定程度上增加了系统的复杂度。

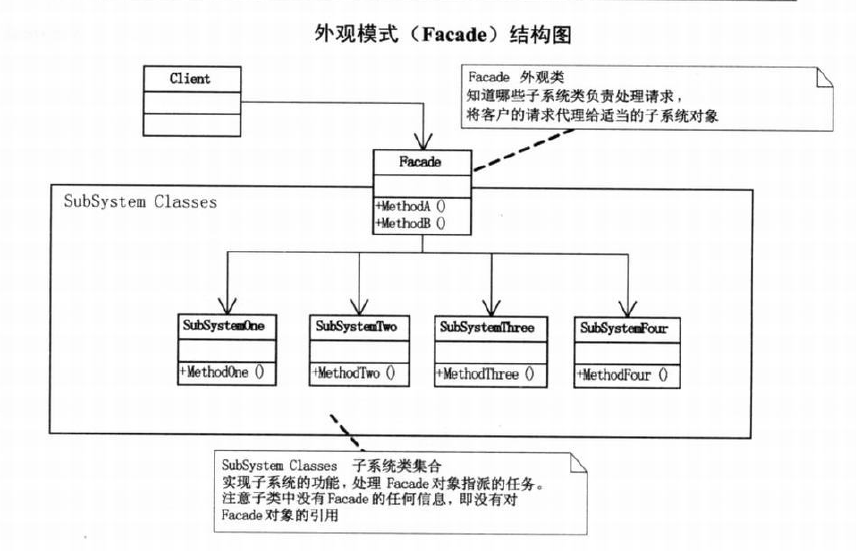
什么叫直接的朋友呢？每个对象都必然会与其他对象有耦合关系，两个对象直接的耦合就成为朋友关系，这种关系的类型有很多，例如组合、聚合、依赖等。下面举例介绍说明如何才能做到只与直接的朋友交流。

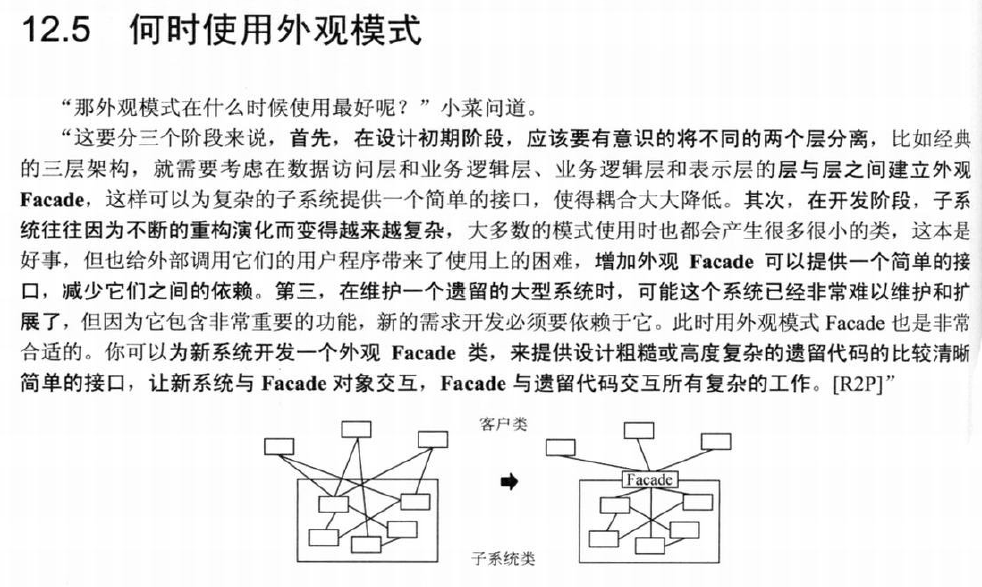
# 外观模式

范例:

对股票和基金的所有股进行搭配买卖,有多种搭配,外观组合这种搭配,供客户端使用;

根本目的:完美体现依赖倒转原则和迪米特法则





# 建造者模式

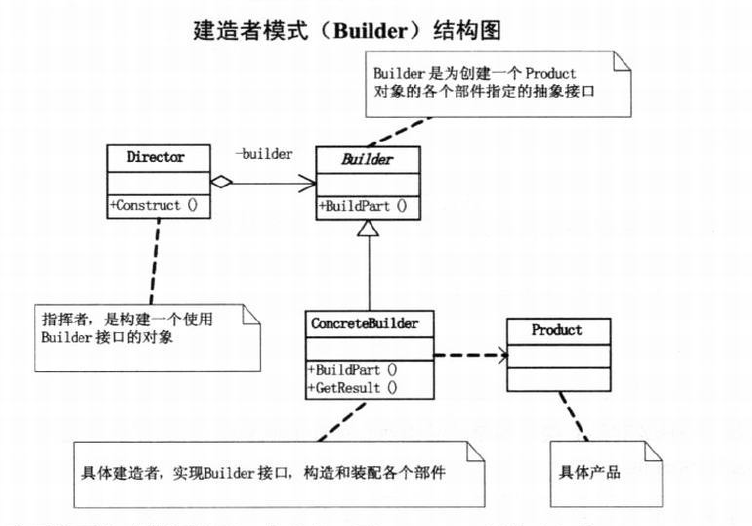
它可以将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方法可以构造出不同表现（属性）的对象。

范例:炒菜的时候,因为步骤很多,忘了放盐,导致菜没有味道;

根本目的:

使得复杂的建造过程变得清晰并且创造的东西多变;

不让任何对象遗漏任何一个建造过程;



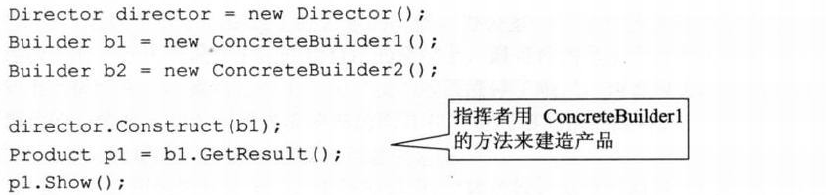
使用方式

客户端使用Director 生成并返回具体的一个产品

Director 使用接口builder 创建具体的产品

concretebuilder 拥有创建具体产品的方法,具体的属性等;

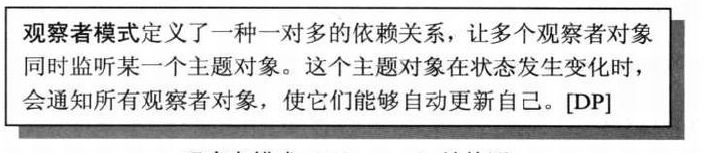
重点在于封装具体产品建造过程



如果产品需要批量生产,可以使用简单工厂模式;

如果批量生产的对象需要更多的操作,最好使用工厂方法模式,

# 观察者模式



范例:

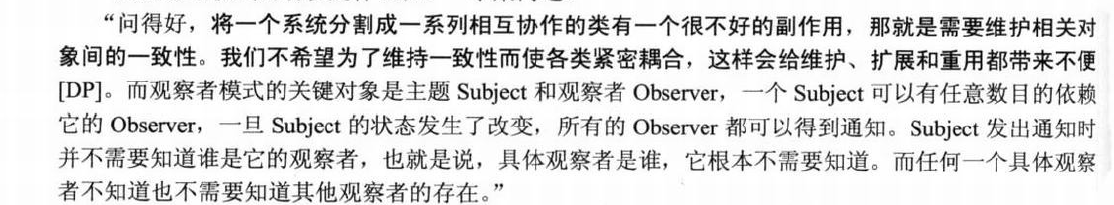
老板回来,前台批量通知已添加的员工;

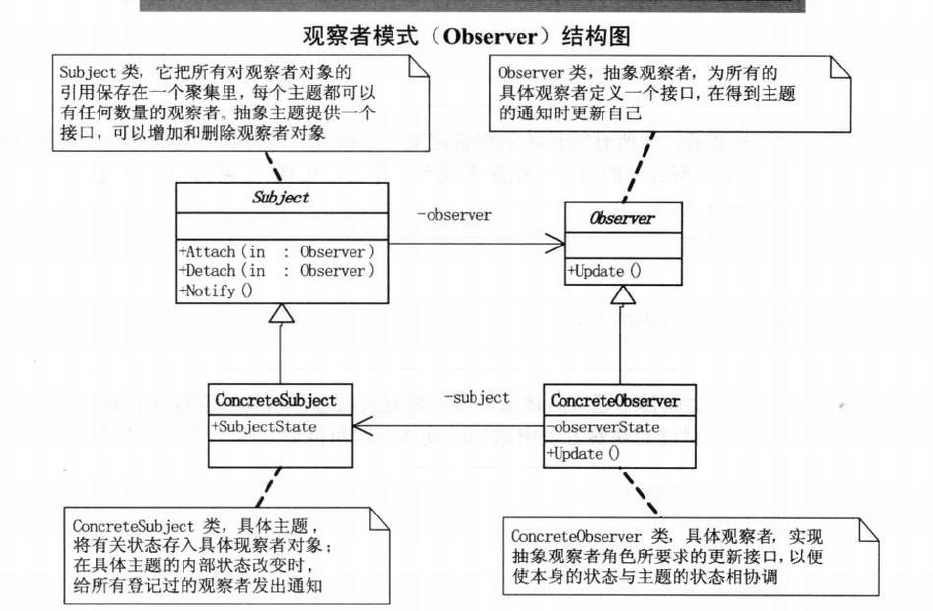
根本目的:

一个对象的状态改变,其他与之相关的对象跟着改变,通过观察者模式,对功能的抽象和对观察者类型的抽象,达到单一的功能--类型操作,维护和扩展;

并使具体的其他类之间无任何关联;

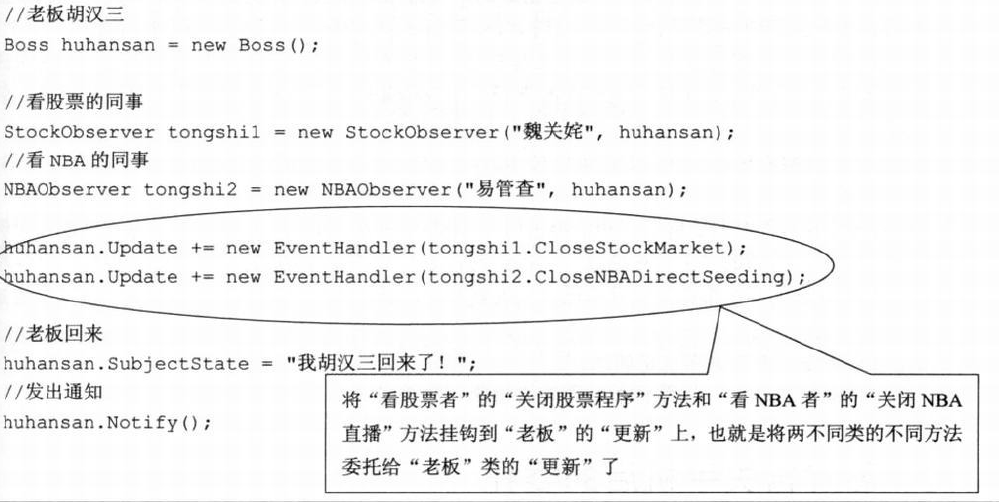
具体对象之间的解耦,抽象与抽象之间的依赖;





委托事件:

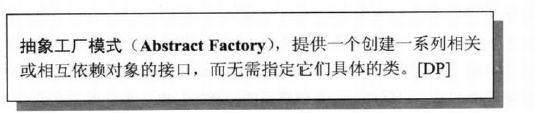
可以通过委托,去除在通知者里添加观察者这一步,而是将每一个观察者的响应函数通过委托附加到观察者的update 上,



消息机制:

在通知者保存每一种消息的key,当观察者状态改变时,根据变化类型,在执行notify时,将具体的消息的key,发送给观察者,由观察者决定执行什么响应;

# 抽象工厂模式



范例:

在项目不变的情况下,改变项目使用的数据库类型;

每种数据库的操作方式会有不同,那么就需要建立数据库操作模块来应对数据库类型的改变;

根本目的:

每个具体的工厂不再只生产一个对象,还生产更多抽象类型的对象;

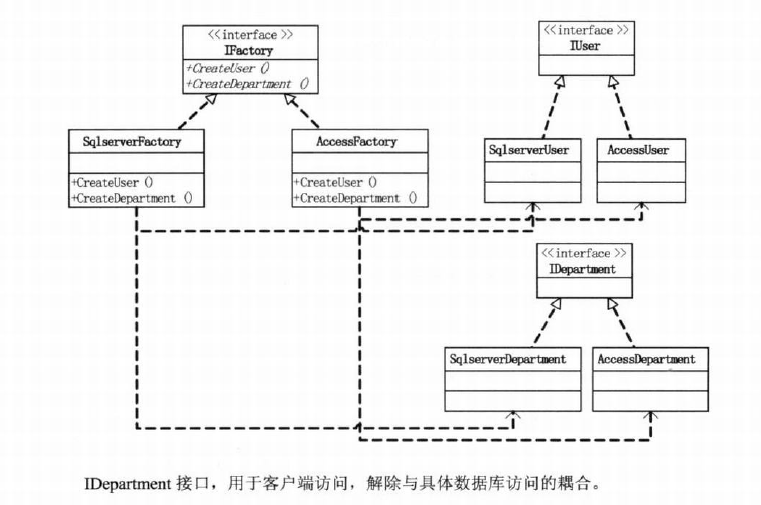
某一接口的实现的产品的数量,代表了工厂类型的数量,因为由接口实现的产品,是不同实现方式的;而每一个工厂,就是每一个实现方式的产品类的创造者;

解决在更换某一功能的实现方式时,更改了很多业务上的类,更改很多使用的方式和类的引用;在更改实现方式时,只需要添加抽象工厂定义的类便可,并且功能实现方式的切换也很方便;

## 抽象工厂实现

添加新功能:

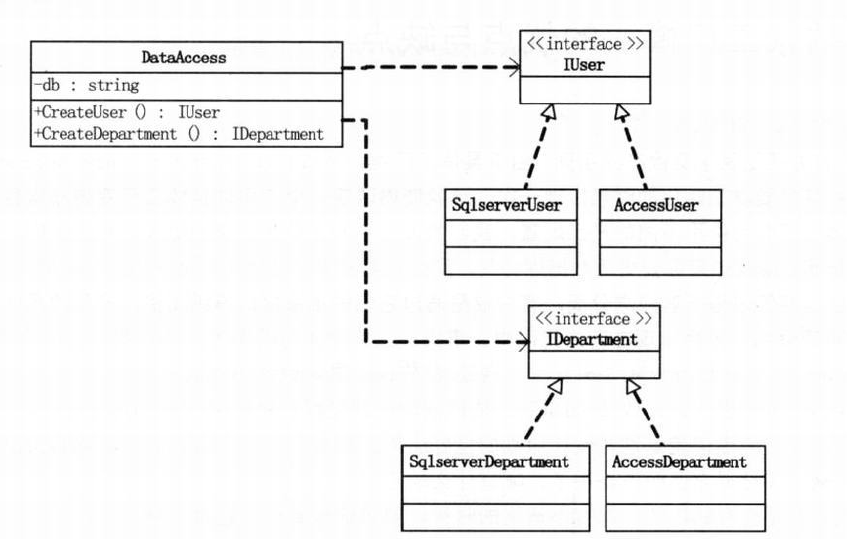


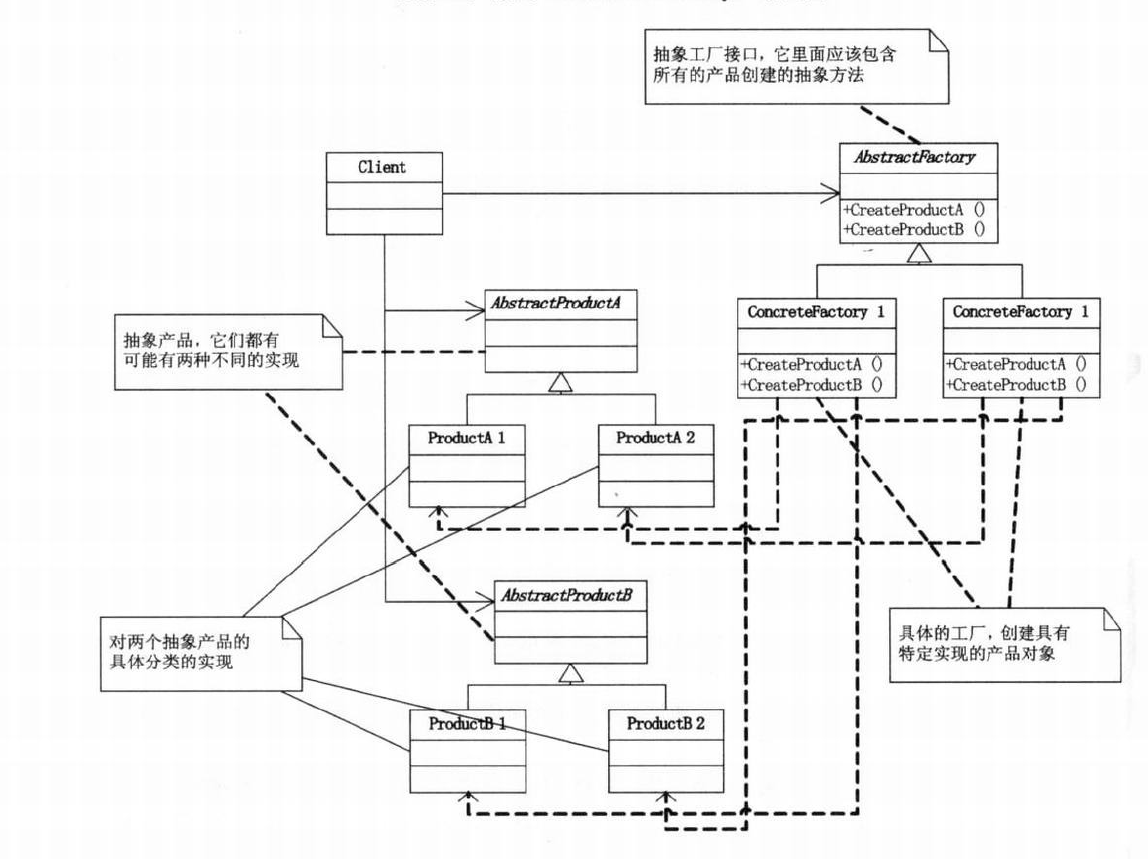


## 简单工厂实现

传入不同的字符串代表数据库类型,工厂create不同的功能类型

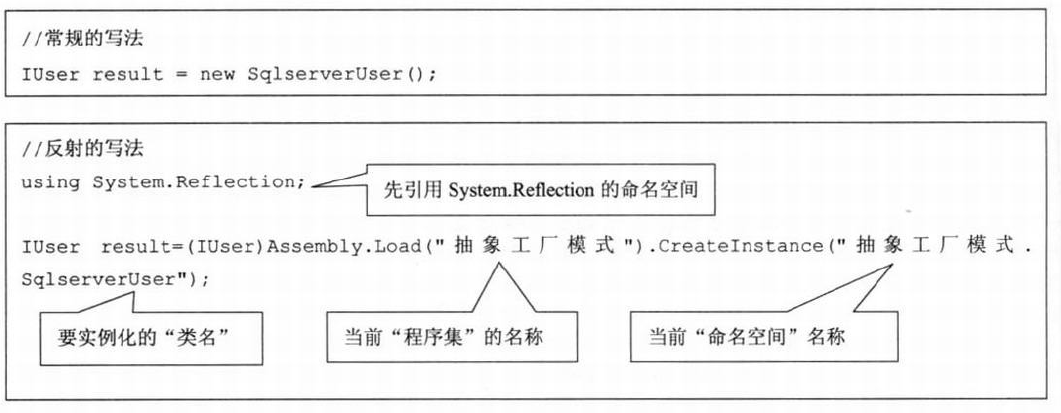
但是在添加一种新数据库的时候,需要添加一系列的功能类到工厂中;

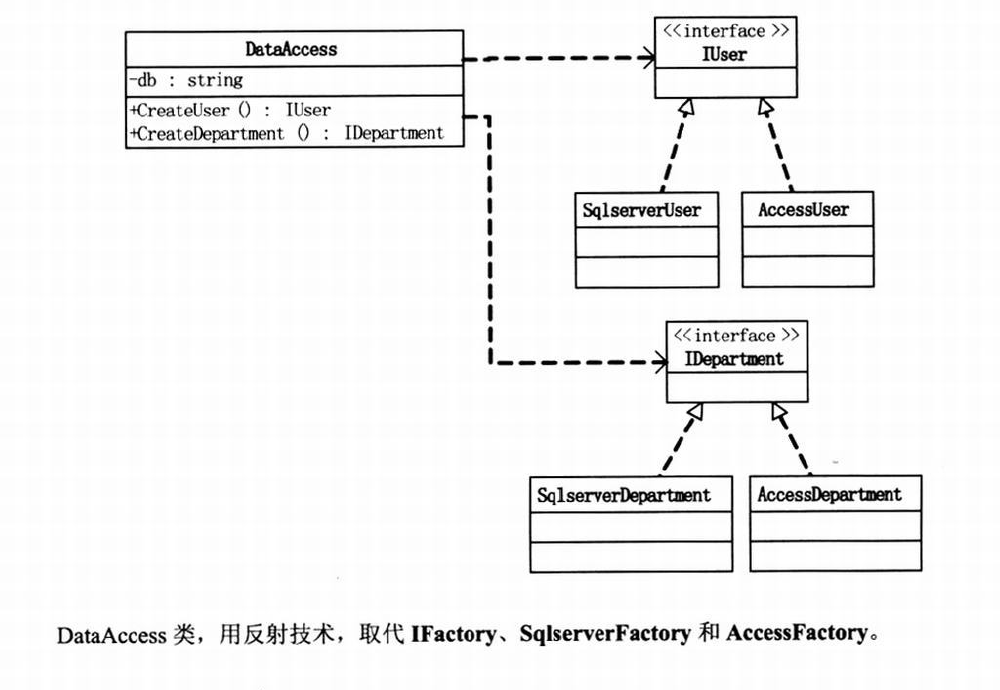




依赖注入方式

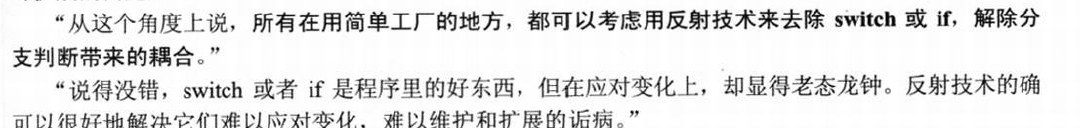
## 反射+抽象工厂



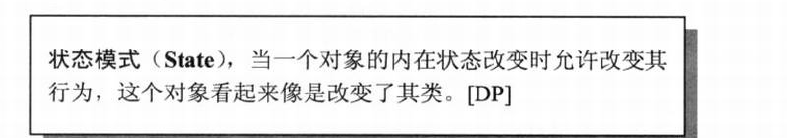


(IUser)Assembly.Load(AssemblyName).CreateInstance(className);

代码为c#系统提供的反射操作,通过类名称实例化一个类;



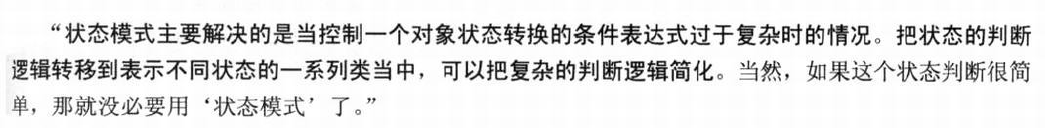
# 状态模式



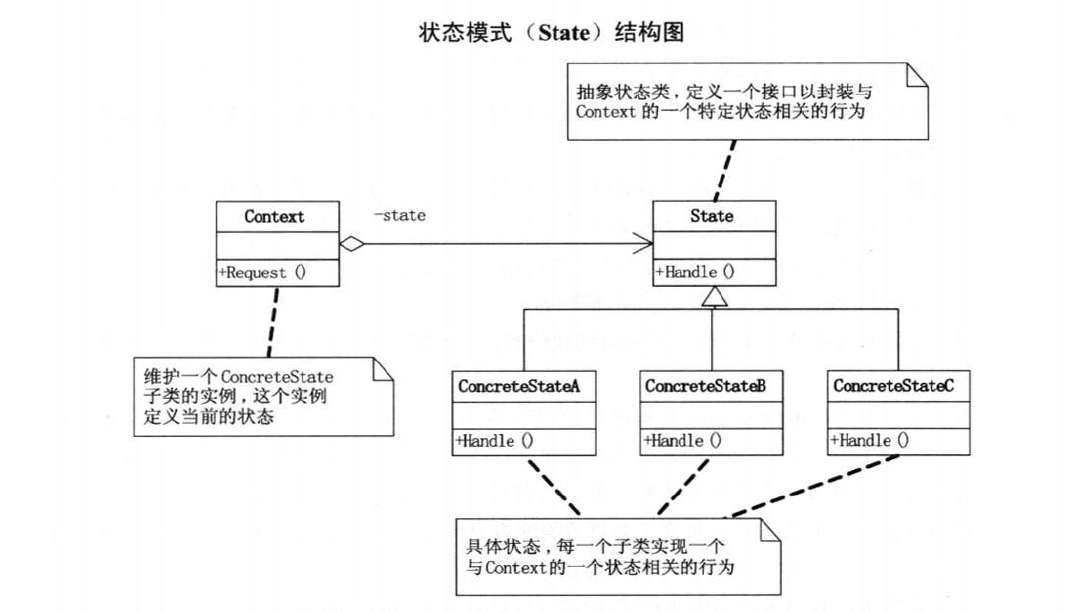
范例:

根据上班的时间点,反应员工的精神状态;

根本目的:



当一个对象的行为取决于他的状态时,并且她在运行时时刻根据他的状态改变他的行为的时候,就可以考虑使用状态模式了.

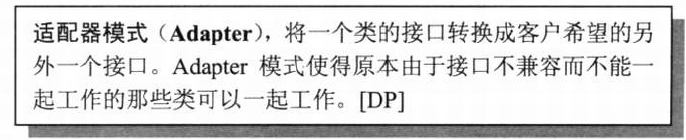


使用方式:

context 设置初始状态,记录状态,发送状态请求;

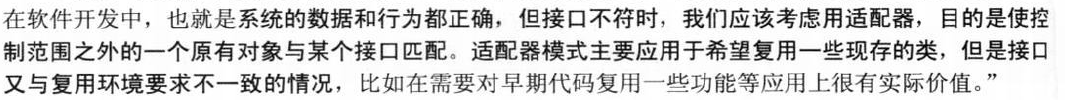
state 定义状态,具体的state响应状态改变并执行后转到下个状态,以此完成状态的改变;

# 适配器模式



范例:  
电源适配器

根本目的:



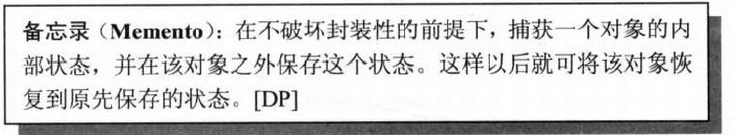
如果适配器两头都无法调整,那么添加适配器;



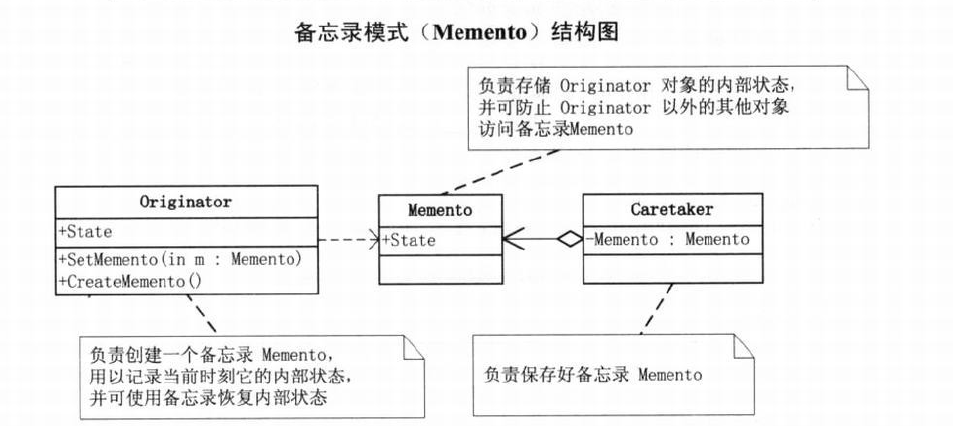
target定义客户端使用的函数

adapter 重写函数,并引用adeptee 对应的函数在这个重写的函数里;

# 备忘录模式



范例:游戏进度保存与提取;



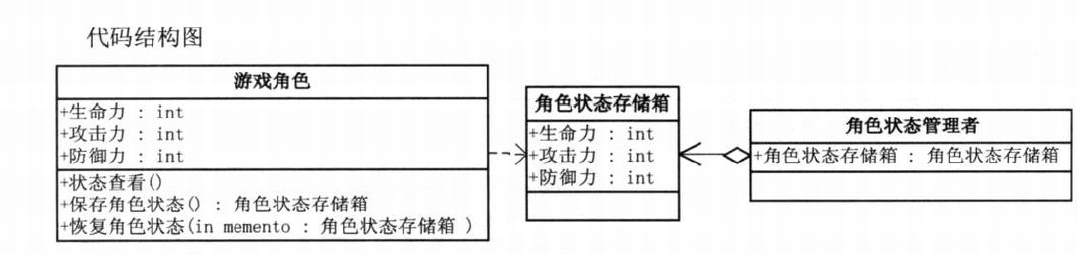
使用方式:

originator:拥有需要保存的属性的对象,比如游戏角色类;

创建Memento数据并保存到caretaker中;

Memento: 需要保存的数据字段集合;

caretaker: 保存每一个memento,并在使用时,通过caretaker返回给游戏角色类;



# 组合模式