设计模式总结

创建时间： 2014/03/19 10:13

更新时间： 2014/05/03 09:51

作　　者： hacke2

来　　源： http://blog.csdn.net/hacke2

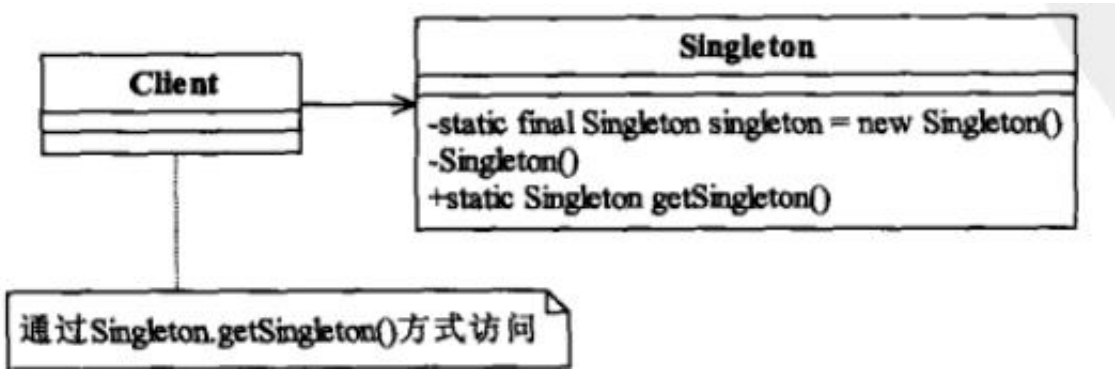
本系列主要记录设计模式的意图是什么，它要解决一个什么问题，什么时候可以使用它；它是如何解决的，掌握它的结构图，记住它的关键代码；能够想到至少两个它的应用实例，一个生活中的，一个软件中的；这个模式的优缺点是什么，其有哪些使用场景，在使用时要注意什么。

1.单例模式

意图：保证一个类仅有一个实例,并提供一个访问它的全局访问点
主要解决：生产一系列产品
什么时候使用：当你想控制实例数目，节省系统资源的时候

如何解决 ：判断系统是否是否已经有这个单例，如果有则返回，如果没有，则创建

结构图：



关键代码：构造函数式私有的

应用实例：1.一个党只能有一个主席.2.Windows是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。3.一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

优点： 1.在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存） 2.避免对资源的多重占用（比如写文件操作）

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，儿不关心外面怎么样来实例化

使用场景：1.要求生产唯一序列号 2.WEB中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来 3.创建的一个对象需要消耗的资源过多，比如I/O与数据库的连接等

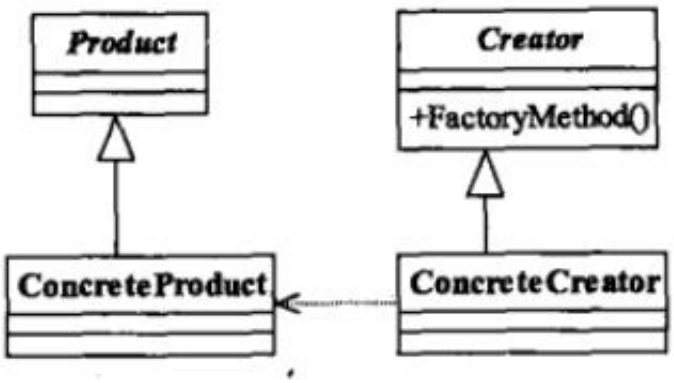
注意事项：getInstance()方法中需要使用同步锁synchronized (Singleton.class)防止多线程同时进入造成instance被多次实例化

2.工厂模式

意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行
主要解决：主要解决接口选择的问题
什么时候使用：我们明确的计划不同条件下创建不同实例时

如何解决 ：让其子类实现工厂接口，返回的也是一个抽象的产品

结构图：



关键代码：创建过程在其子类执行
应用实例：1.你需要一辆汽车，你可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现 2.Hibernate换数据库只需换方言和驱动就可以

优点：1.一个调用者想创建一个对象，只要知道其名称就可以了 2.扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以 3.屏蔽产品的具体实现，调用者只关心产品的接口

缺点：每次增加一个产品时，都需要增加一个具体类和对象实现工厂，是的系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

使用场景：1.日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。2.数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时3.设计一个连接服务器的框架，需要三个协议，"POP3", "IMAP", "HTTP"，可以把这三作为产品类，共同实现一个接口

注意事项：作为一种创建类模式，在任何需要生成**复杂对象**的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过new就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

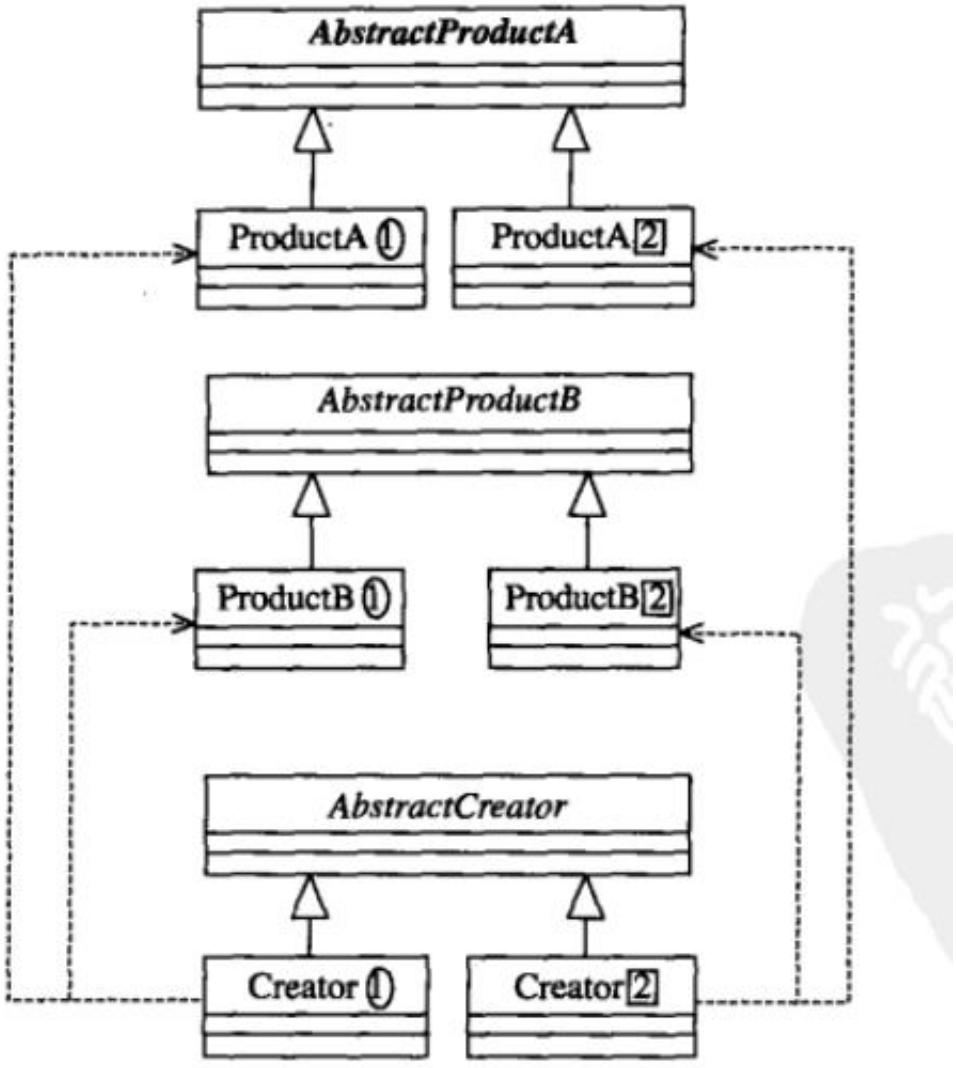
3.抽象工厂

意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类
主要解决：主要解决接口选择的问题

什么时候使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品

如何解决 ：在一个产品族里面，定义多个产品

结构图：



关键代码：在一个工厂里聚合多个同类产品
应用实例：工作了，为了参加一些聚会，肯定有两套或多套衣服吧，比如说有商务装（成套，一系列具体产品）、时尚装（成套，一系列具体产品），甚至对于一个家庭来说，可能有商务女装、商务男装、时尚女装、时尚男装，这些也都是成套的，即一系列具体产品.咱们假设一种情况(现实中是不存在的,要不然,没法进入共产主义了,但有利于说明抽象工厂模式)，在你的家中，某一个衣柜（具体工厂）只能存放某一种这样的衣服（成套，一系列具体产品），每次拿这种成套的衣服时也自然要从这个衣柜中取出了.用OO的思想去理解，所有的衣柜（具体工厂）都是衣柜类的（抽象工厂）某一个，而每一件成套的衣服又包括具体的上衣（某一具体产品），裤子（某一具体产品），这些具体的上衣其实也都是上衣（抽象产品），具体的裤子也都是裤子（另一个抽象产品）.

优点：1当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

缺点：产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的Creator里加代码，又要在具体的里面加代码

使用场景：1.QQ换皮肤，一整套一起换 2.生成不同操作系统的程序

注意事项：产品族难扩展，产品等级易扩展

4.建造者模式

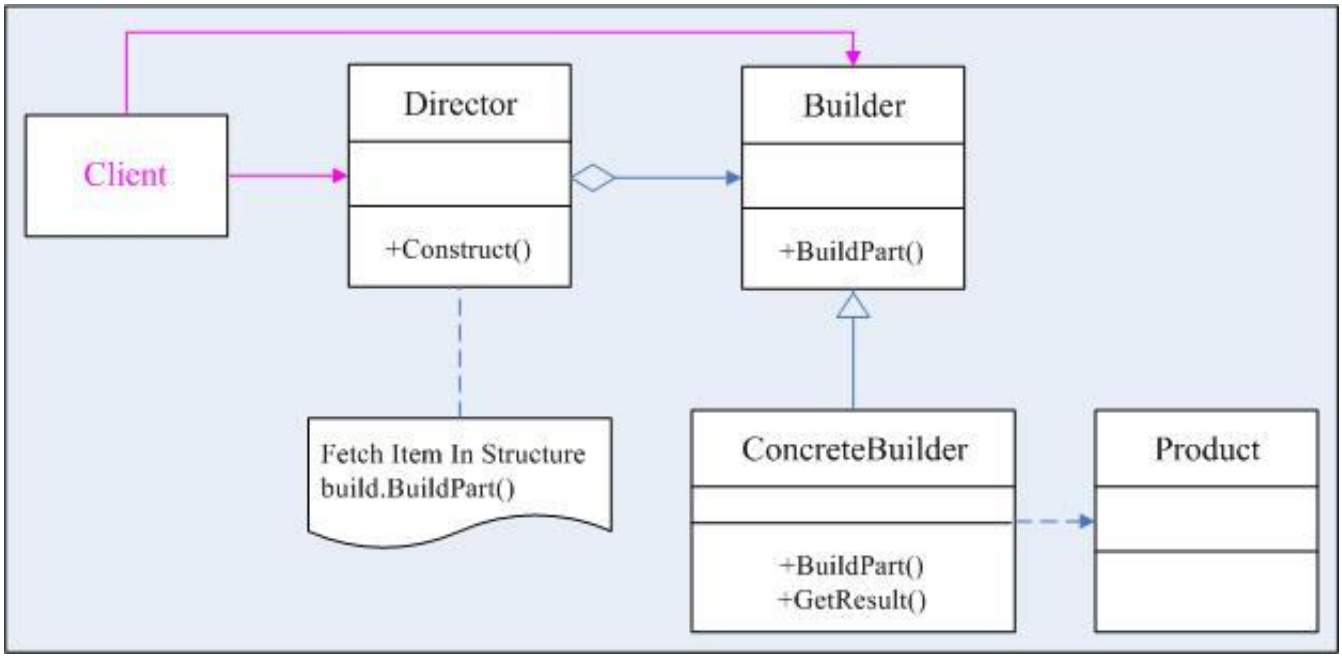
意图：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

主要解决：主要解决在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法确相对稳定。

什么时候使用：一些基本部件不会变，而其组合经常变化的时候

如何解决 ：将变与不变分离开

结构图：



关键代码：建造者：创建和提供实例，导演：管理建找出来的实例的依赖关系

应用实例：1.去肯德基，汉堡，可乐，薯条，炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的“套餐” 2.JAVA中的StringBuilder

优点：1建造者独立，易扩展 2.便于控制细节风险

缺点：1.产品必须有共同点，范围有限制 2.如内部变化复杂，会有很多的建造类

使用场景：1.需要生成的对象具有复杂的内部结构 2.需要生成的对象内部属性本身相互依赖

注意事项：与工厂模式的区别是：建造者模式更加关注与零件装配的顺序

5.原型模式

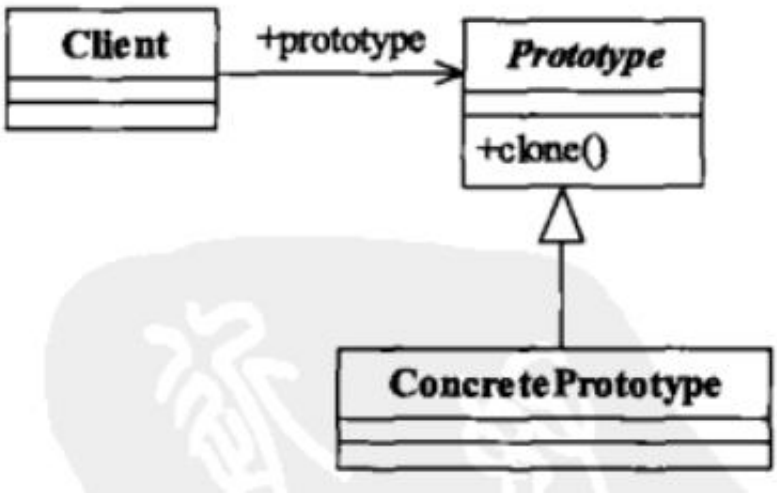
意图：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

主要解决：在运行期建立和删除原型

什么时候使用：1．当一个系统应该独立于它的产品创建，构成和表示时；2．当要实例化的类是在运行时刻指定时，例如，通过动态装载；3．为了避免创建一个与产品类层次平行的工厂类层次时；4．当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些

如何解决 ：利用已有的一个原型对象，快速的生成和原型对象一样的实例

结构图：



关键代码：1.实现克隆操作，在JAVA继承Cloneable，重写clone(),在.NET中可以使用Object类的MemberwiseClone()方法来实现对象的浅表拷贝或通过序列化的方式来实现深拷贝。2．Prototype模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些“易变类”拥有稳定的接口。

应用实例：1.细胞分裂 2.JAVA中的Object clone()方法

优点：1 性能提高 2,逃避构造函数的约束

缺点：1.配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。2. 必须实现Cloneable接口 3,逃避构造函数的约束

使用场景：1.资源优化场景 2.类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。 3.性能和安全要求的场景 4.通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。 5.一个对象多个修改者的场景 6.一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。7. 在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过clone的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与Java融为浑然一体，大家可以随手拿来使用。

注意事项：与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的.浅拷贝实现Cloneable,重写，深拷贝实通过实现Serializable读取二进制流，

6.适配器模式

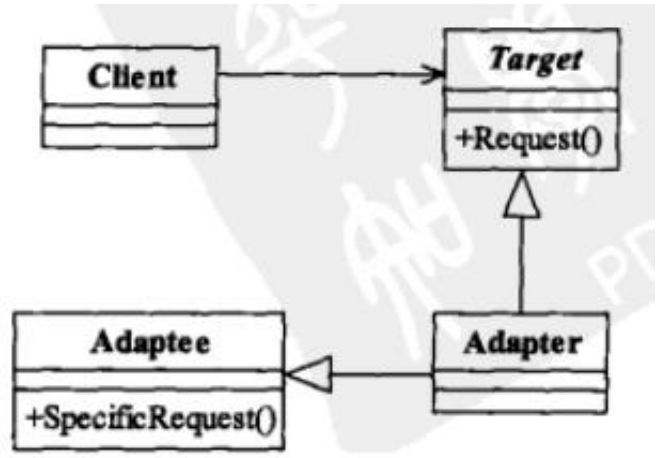
意图：将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

主要解决：主要解决在软件系统中，常常要将一些“现存的对象”放到新的环境中，而新环境要求的接口是现对象不能满足的

什么时候使用：1.系统需要使用现有的类，而此类的接口不符合系统的需要。2.想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有与一致的接口。3.通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口），

如何解决 ：继承或依赖（推荐）

结构图：



关键代码：适配器继承或依赖已有的对象，实现想要目标接口

应用实例：1.美国电器110V，中国220V，就要有一个适配器将110V转化为220V 2.JAVA JDK 1.1提供了Enumeration接口，而在1.2中提供了Iterator接口，想要使用1.2 的JDK，则要将以前系统的Enumeration接口转化为Iterator接口，这时就需要适配器模式 3.在LINUX上运行WINDOWS程序 4.java中的jdbc

优点：1可以让任何两个没有关联的类一起运行 2.提高了类的复用 3.增加了类的透明度 4.灵活性好

缺点：1.过多的使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。2.由于JAVA至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类

使用场景：有动机的修改一个正常运行的系统的接口，这是应该考虑使用适配器模式

注意事项：适配器不是在详细设计师添加的，而是解决正在服役的项目的问题

7.组合模式

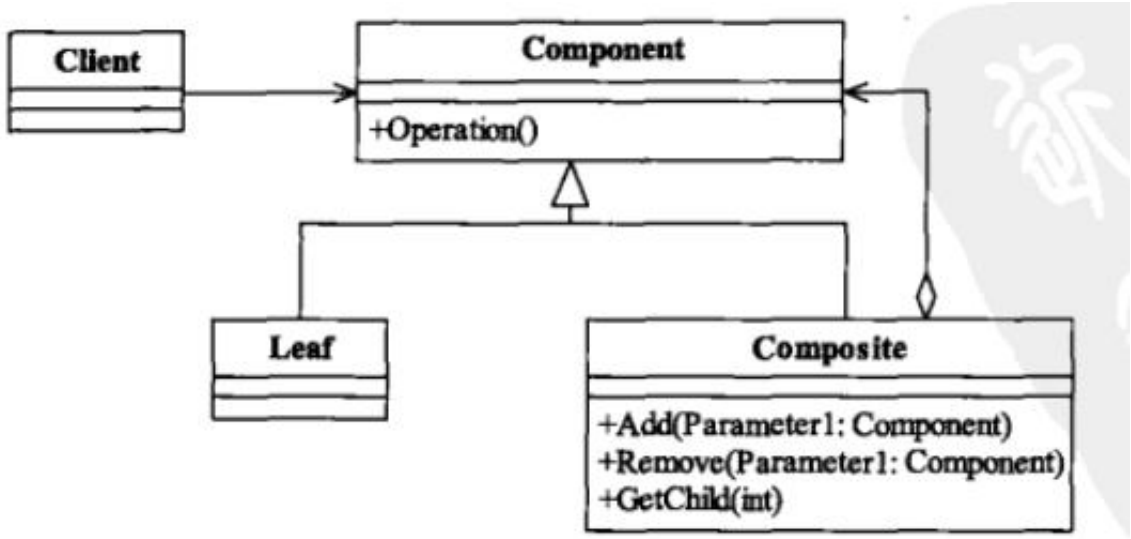
意图：将对象组合成树形结构以表示“部分-整体”的层次结构。Composite模式使得用户对单个对象和组合对象的使用具有一致性。

主要解决：它使我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以向处理简单元素一样来处理复杂元素,从而使得客户程序与复杂元素的内部结构解耦。

什么时候使用：1.你想表示对象的部分-整体层次结构（树形结构） 2.你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

如何解决 ：树枝和叶子实现统一接口，树枝内部组合该接口

结构图：



关键代码：树枝内部组合该接口，并且含有内部属性List，里面放Component

应用实例：1.算术表达式包括操作数、操作符和另一个操作数,其中，另一个操作符也可以是操作树，操作符和另一个操作数 2.在JAVA AWT和SWING中，对于Button和Checkbox是树叶，Container是树枝

优点：1. 高层模块调用简单 2.节点自由增加

缺点：在使用组合模式时，起叶子和树枝的声明都是实现类，而不是接口，违反了依赖倒置原则

使用场景：部分、整体场景，如树形菜单，文件、文件夹的管理

注意事项：定义时为具体类

8.装饰模式

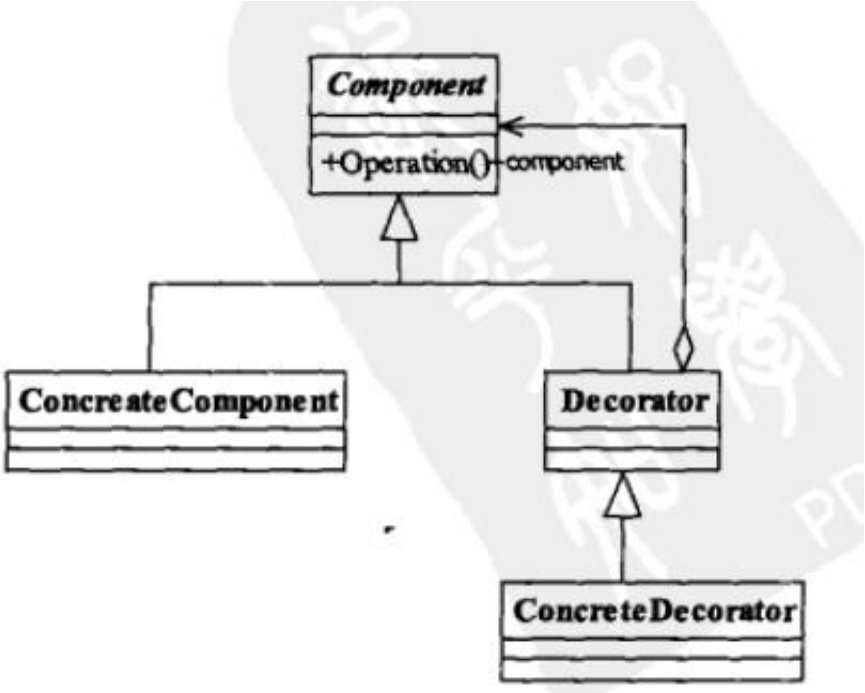
意图：动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

主要解决：一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀

什么时候使用：在不想增加很多子类的情况下扩展类

如何解决 ：将具体功能职责划分，同时继承装饰者模式

结构图：



关键代码：1.Component类充当抽象角色，不应该具体实现2 修饰类引用和继承Component类，具体扩展类重写父类方法

应用实例：1.孙悟空有72变，当他变成“庙宇”后，他的根本还是一只猴子，但是他又有了庙宇的功能 2.不论一幅画有没有画框都可以挂在墙上，但是通常都是有画框的，并且实际上是画框被挂在墙上。在挂在墙上之前，画可以被蒙上玻璃，装到框子里；这时画、玻璃和画框形成了一个物体。

优点：1. 装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能

缺点：多层装饰比较复杂

使用场景：1. 扩展一个类的功能 2.动态增加功能，动态撤销

注意事项：可代替继承

9.代理模式

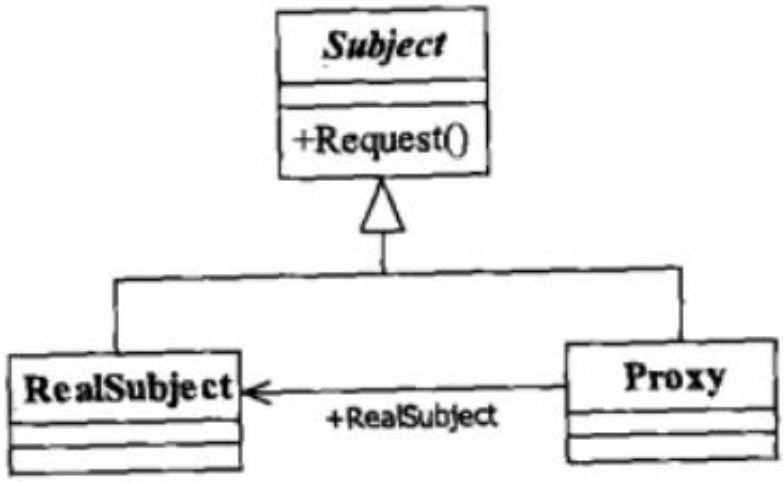
意图：为其他对象提供一种代理以控制对这个对象的访问。

主要解决：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上.在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层

什么时候使用：想在访问一个类时做一些控制

如何解决 ：增加中间层

结构图：



关键代码：实现与被代理类，与被代理类组合

应用实例：1.windows里面的快捷方式 2.猪八戒去找高翠兰结果是孙悟空变的，可以这样理解：把高翠兰的外貌抽象出来，高翠兰本人和孙悟空都实现了这个接口，猪八戒访问高翠兰的时候看不出来这个是孙悟空，所以说孙悟空是高翠兰代理类。3.买火车票票不一定在火车站买，也可以去代售点 4.一张支票或银行存单是账户中资金的代理。支票在市场交易中用来代替现金，并提供对签发人账号上资金的控制。 5.spring aop

优点：1. 职责清晰 2.高扩展性 3.智能化

缺点：1.由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。2 实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

使用场景：按职责来划分，通常有一下使用场景：1.远程代理 2.虚拟代理 3.Copy-on-Write代理 4.保护（Protect or Access）代理 5.Cache代理 6.防火墙（Firewall）代理 7.同步化（Synchronization）代理 7.智能引用（Smart Reference）代理

注意事项：1.和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理迷失不能改变所代理类的接口 2.和装饰模式的区别：装饰模式为了增强功能，而代理模式是为了加以控制

10.享元模式

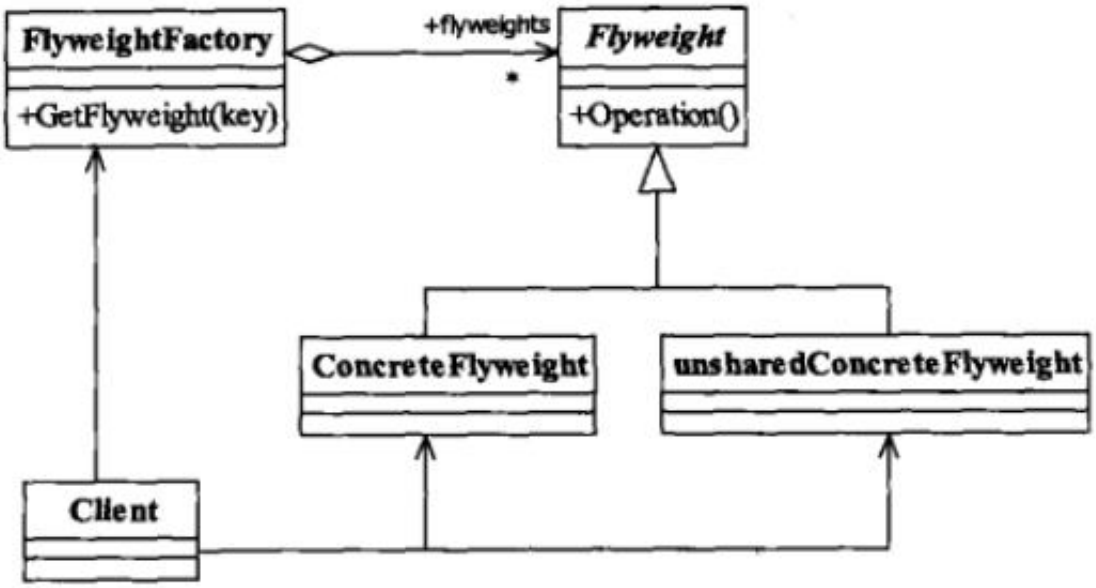
意图：运用共享技术有效地支持大量细粒度的对象.

主要解决：在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建

什么时候使用：1. 系统中有大量对象 2.这些对象消耗大量内存 3.这些对象的状态大部分可以外部化 4.这些对象可以按照内蕴状态分为很多组，当吧外蕴对象从对象中剔除出来是，每一组对象都可以用一个对象来代替 5.系统不依赖与这些对象身份，这些对象是不可分辨的

如何解决 ：用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象

结构图：



关键代码：用HashMap存储这些对象

应用实例：1.JAVA中的String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面 2.数据库的数据池

优点：大大减少对象的创建，降低系统的内存，使效率提高

缺点：提高了系统的负责都，需要分离出外部状态和内部状态，而且外部状态具有固有化的性质，不应该随着内部状态的变化而变化，否则会造成系统的混乱

使用场景：1. 系统有大量相似对象 2.需要缓冲池的场景

注意事项：1.注意划分外部状态和内部状态，否则可能会引起线程安全问题 2.这些类必须有一个工厂对象加以控制

11.门面模式

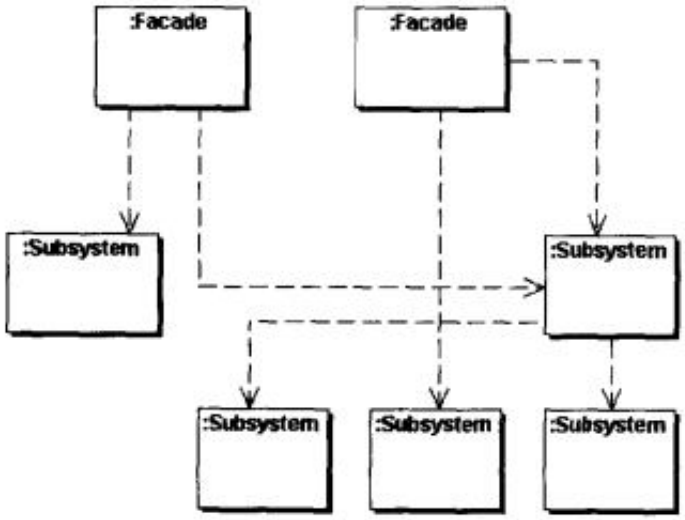
意图：为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口

什么时候使用：1. 客户端不需要知道系统内部的复杂联系，整个系统只需提供一个“接待员”即可 2.定义系统的入口

如何解决 ：客户端不与系统耦合，门面类与系统耦合

结构图：



关键代码：在客户端和复杂系统之间再加一层，这一次将调用顺序、依赖关系等处理好

应用实例：1.去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便 2.JAVA的三层开发模式

优点：1. 减少系统相互依赖 2.提高灵活性 3.提高了安全性

缺点：不符合开闭原则，如果要改东西很麻烦，继承重写都不合适

使用场景：1. 为复杂的模块或子系统提供供外界访问的模块 2.子系统相对独立 3.预防低水平人员带来的风险

注意事项：在层次化结构中，可以使用Facade模式定义系统中每一层的入口。

12.桥接模式

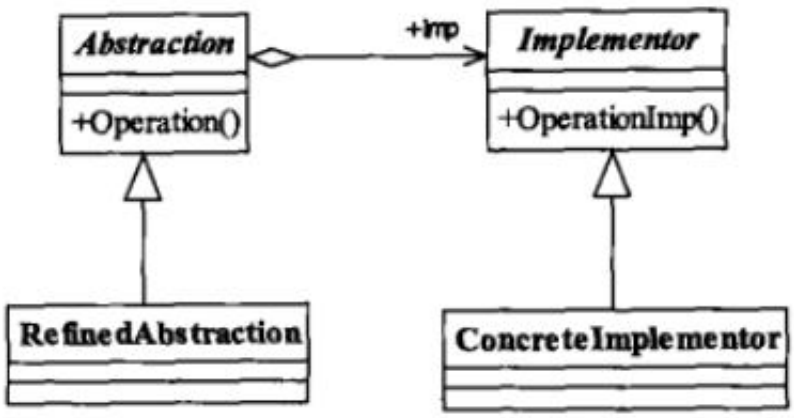
意图：将抽象部分与实现部分分离，使它们都可以独立的变化.

主要解决：在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活

什么时候使用：实现系统可能有多个角度分类，每一种角度都可能变化

如何解决 ：那么把这种多角度分类给分离出来让他们独立变化，减少他们之间耦合

结构图：



关键代码：抽象类依赖实现类

应用实例：1、猪八戒从天蓬元帅转世投胎到猪，转世投胎的机制将尘世划分为两个等级，即：灵魂和肉体，前者相当于抽象化，后者相当于实现化。生灵通过功能的委派，调用肉体对象的功能，使得生灵可以动态的选择 2、墙上的开关，自己可以看到的开关是抽象的，里面具体怎么实现我们不管

优点：1、抽象和实现的分离. 2、优秀的扩展能力 3、实现细节对客户透明

缺点：1、桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程.

使用场景：1、如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系. 2、对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用. 3、一个类存在两个独立变化的维度，且这两个维度都需要进行扩展.

注意事项：对于两个独立变化的维度，使用桥接模式再适合不过了.

13.策略模式

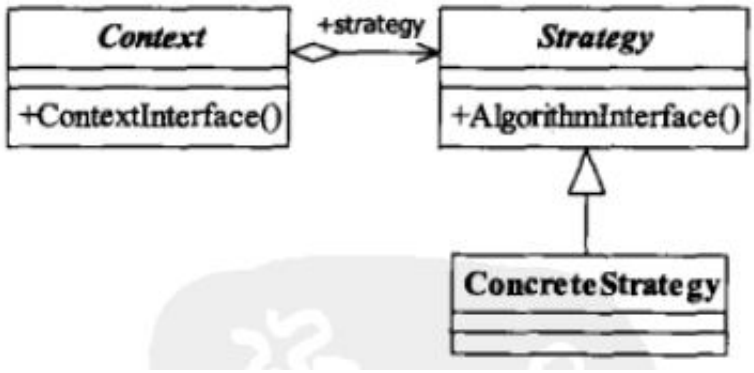
意图：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决：在有多种算法相似的情况下，使用if..else所带来的复杂和难以维护

什么时候使用：一个系统有许多许多类，而区分他们的只是他们直接的行为

如何解决 ：将这些算法封装成一个一个的类，任意的替换

结构图：



关键代码：实现同一个接口

应用实例：1、诸葛亮的锦囊妙计，每一个锦囊就是一个策略 2、旅行的出游方式，选择骑自行车、坐汽车每一种旅行方式都是一个策略 3、JAVA AWT中的LayoutManager

优点：1、算法可以自由切换 2、避免使用多重条件判断 3、扩展性良好

缺点：1、策略类会增多 2、所有策略类都需要对外暴露

使用场景：1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为 2、一个系统需要动态地在几种算法中选择一种 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现

注意事项：如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题

14.模板方法模式

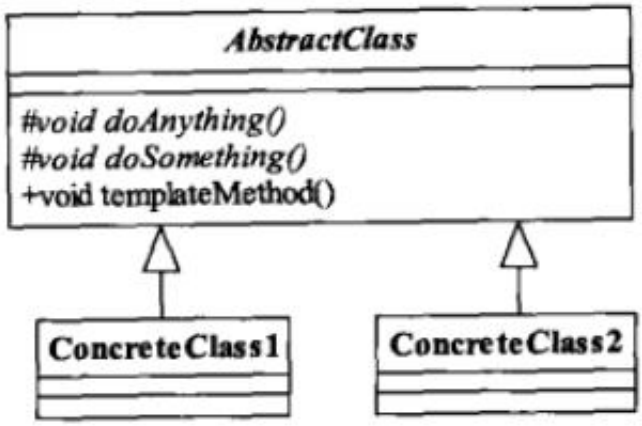
意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中.Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤.

主要解决：一些方法通用，却在每一个子类都重新写了这一方法

什么时候使用：有一些通用的方法

如何解决 ：将这些通用算法抽象出来

结构图：



关键代码：在抽象类实现，其他步骤在子类实现

应用实例：1、在造房子的时候，地基、走线、水管都一样，只有在建筑的后期才有加壁橱加栅栏等差异 2、西游记里面菩萨定好的81难，这就是一个顶层的逻辑骨架 3、Spirng中对Hibernate的支持，将一些已经定好的方法封装起来，比如开启事务、获取Session、关闭Session等，程序员不重复写那些已经规范好的代码，直接丢一个实体就可以保存

优点：1、封装不变部分，扩展可变部分 2、提取公共代码，便于维护 3、行为由父类控制，子类实现

缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大

使用场景：1、有多个子类共有的方法，且逻辑相同 2、重要的、复杂的方法，可以考虑作为模板方法

注意事项：为防止恶意操作，一般在模板方法都加上final关键词

15.观察者模式

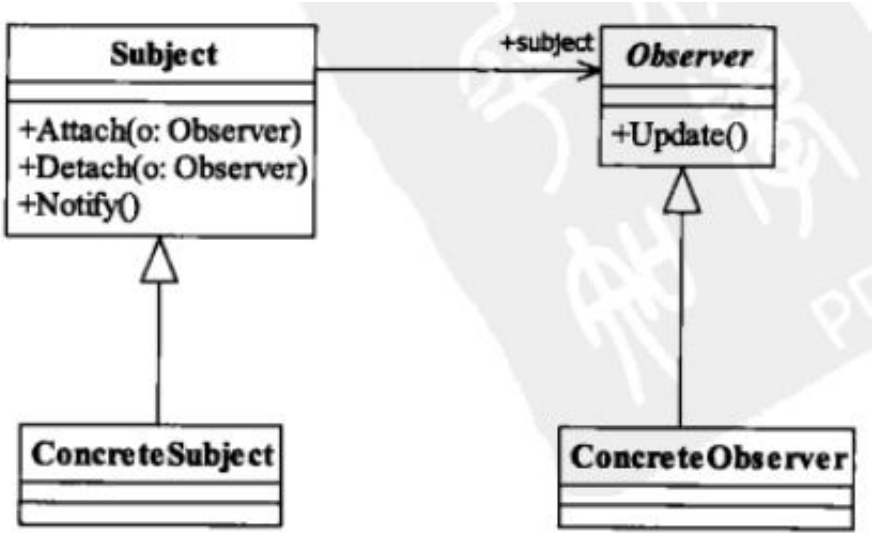
意图：定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作

什么时候使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知

如何解决 ：使用面向对象技术，可以将这种依赖关系弱化

结构图：



关键代码：在抽象类里有一个ArrayList放观察者们

应用实例：1、拍卖的时候，拍卖师观察最高标价，然后通知给其他竞价者竞价 2、西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作

优点：1、观察者和被观察者是抽象耦合的 2、建立一套触发机制

缺点：1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多

时间.2、 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进 行循环调用，可能导致系统崩溃.3、 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化.

使用场景：1、有多个子类共有的方法，且逻辑相同 2、重要的、复杂的方法，可以考虑作为模板方法

注意事项：1、JAVA中已经有了对观察者模式的支持类 2、避免循环引用 3、如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式

16.迭代器模式

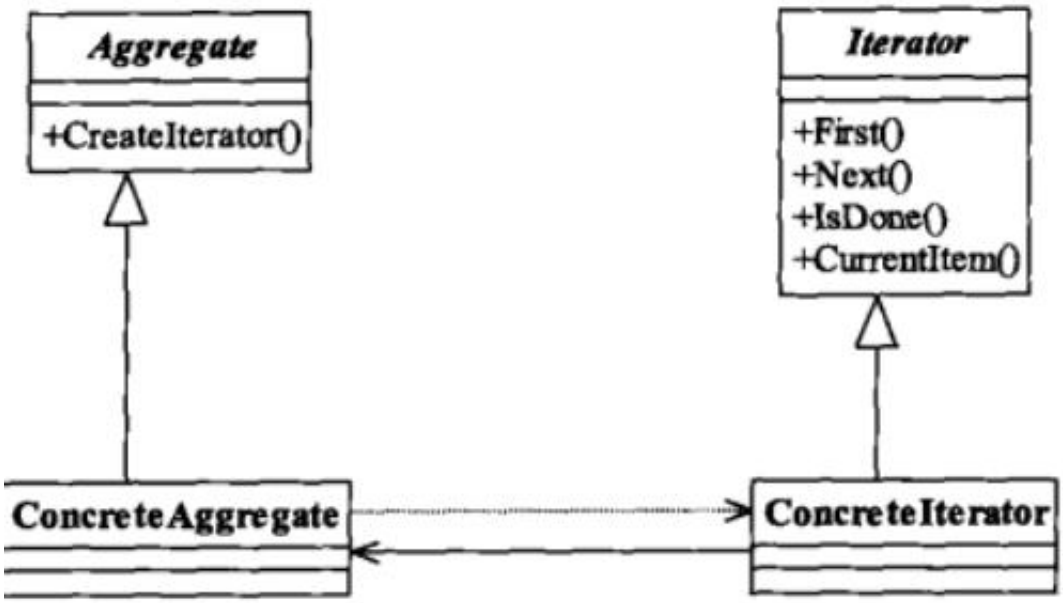
意图：提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示

主要解决：不同的方式来遍历整个整合对象

什么时候使用：遍历一个聚合对象

如何解决 ：把在元素之间游走的责任交给迭代器，而不是聚合对象

结构图：



关键代码：定义接口：hasNext, next

应用实例：JAVA中的iterator

优点：1、它支持以不同的方式遍历一个聚合对象。2、迭代器简化了聚合类。3、在同一个聚合上可以有多个遍历。 4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

缺点：由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

使用场景：1、访问一个聚合对象的内容而无须暴露它的内部表示。 2、需要为聚合对象提供多种遍历方式。 3、为遍历不同的聚合结构提供一个统一的接口。

注意事项：Iterator模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不

暴露集合的内部结构，又可让外部代码透明的访问集合内部的数据。

17.责任链模式

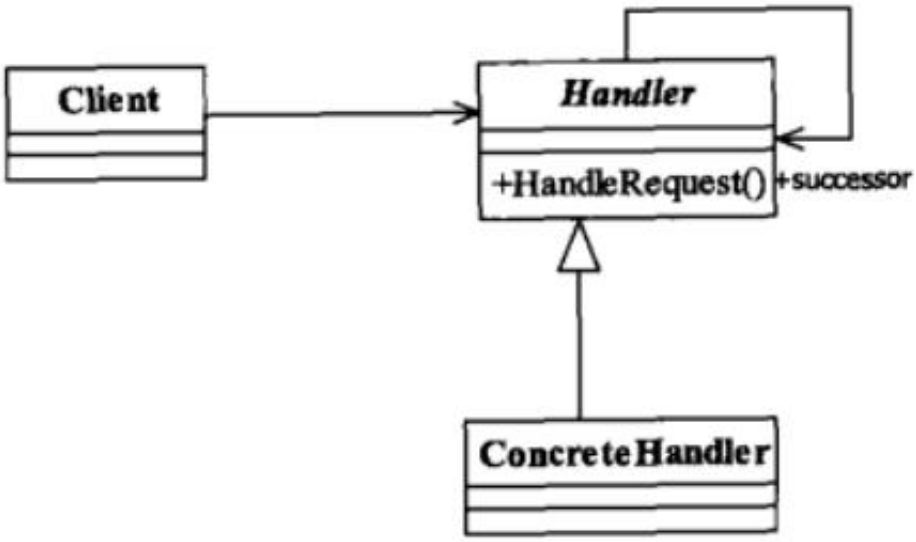
意图：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止

主要解决：职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦了

什么时候使用：在处理消息的时候以过滤很多道

如何解决 ：拦截的类都实现统一接口

结构图：



关键代码：Handler里面聚合他自己，在HanleRequest里判断是否合适，如果没达到条件则向下传递，向谁传递之前set进去

应用实例：1、红楼梦中的“击鼓传花”2、JS中的事件冒泡 3、JAVA WEB中 Apache Tomcat 的对Encoding的 处理，Struts2 的拦截器， jsp servlet 的Filte

优点：1、降低耦合度。它将请求的发送者和接受者解耦。2、简化了对象。使得对象不需要知道链的结构。 3、增强给对象指派职责的灵活性。通过改变链内的成员或者调动它们的次序，允许动态地新增或者删除责任。 4、增加新的请求处理类很方便。

缺点：1、不能保证请求一定被接收。 2、系统性能将受到一定影响，而且在进行代码调试时不太方便；可能会造成循环调用。 3、可能不容易观察运行时的特征，有碍于除错。

使用场景：1、有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。 2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。 3、可动态指定一组对象处理请求。

注意事项：在JAVA WEB中遇到很多应用

18.命令模式

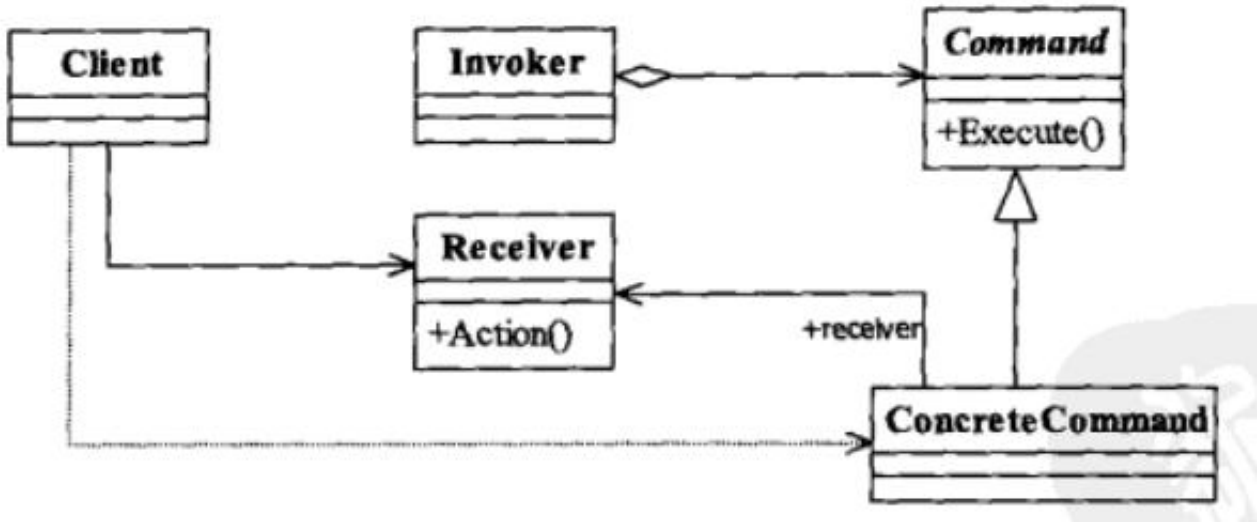
意图：将一个请求封装成一个对象，从而使你可以用不同的请求对客户进行参数化

主要解决：在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时，这种无法抵御变化的紧耦合的设计就不太合适.

什么时候使用：在某些场合，比如要对行为进行“记录、撤销/重做、事务”等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将“行为请求者”与“行为实现者”解耦？将一组行为抽象为对象，可以实现二者之间的松耦合

如何解决 ：通过调用者调用接受者执行命令 顺序：调用者→接受者→命令

结构图：



关键代码：定义三个角色：1、received 真正的命令执行对象 2、Command 3、invoker 使用命令对象的入口

应用实例：struts 1中的action 核心控制器ActionServlet只有一个，相当于Invoker，而模型层的类会随着不同的应用有不同的模型类，相当于具体的Command

优点： 1. 降低了系统耦合度 2. 新的命令可以很容易添加到系统中去。

缺点：使用命令模式可能会导致某些系统有过多的具体命令类。

使用场景：认为是命令的地方都可以使用命令模式，比如：1、GUI中每一个按钮都是一条命令 2、模拟CMD

注意事项：系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作也可以考虑使用命令模式，见命令模式的扩展。

19.备忘录模式

意图：在不破坏封装性的前提下,捕获一个对象的内部状态,并在该对象之外保存这个状态。

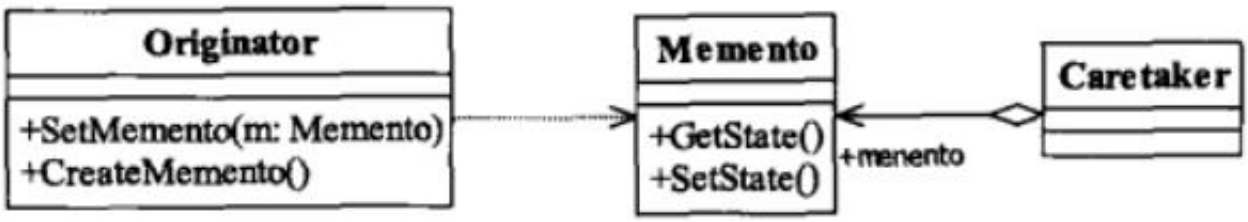
主要解决：所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存

这个状态，这样可以在以后将对象恢复到原先保存的状态

什么时候使用：很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，使得他有“后悔药”可吃

如何解决 ：通过一个备忘录类专门存储对象状态

结构图：



关键代码：客户不与备忘录类耦合，与备忘录管理类耦合

应用实例：1、后悔药 2、打游戏时的存档 3、Windows里的ctri + z 4、IE中的后退 4、数据库的事务管理 5、WEB中的SESSION和COOKIE 6、孙悟空将人参树推到，后求助于观音，观音撒了几滴水，数九附后了，其中孙悟空是发起人，人参树是具体对象(Originator)，那几滴水是备忘录角色(Memento)，观音是负责人角色(Caretaker)

优点：1、 给用户提供了一种可以恢复状态的机制。可以是用户能够比较方便地回到某个历史的状态。 2、实现了信息的封装。使得用户不需要关心状态的保存细节。

缺点：消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

使用场景：1、需要保存/恢复数据的相关状态场景 2、提供一个可回滚的操作

注意事项：1、为了符合迪米特原则，还要增加 一个管理备忘录的类 2、为了节约内存，可使用原型模式+备忘录模式

20.状态模式

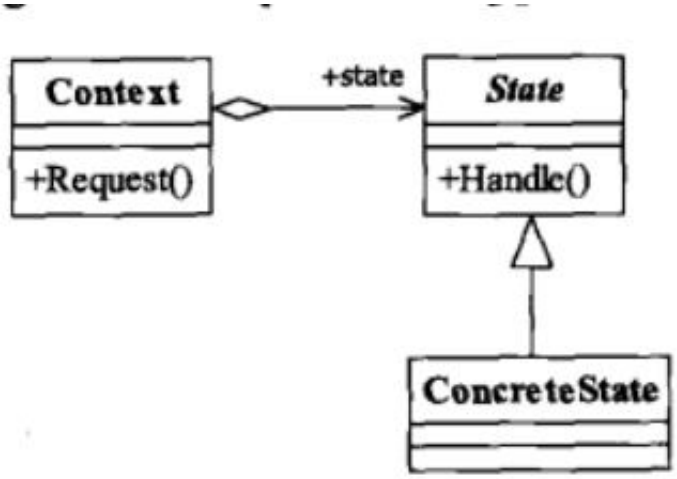
意图：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类.

主要解决：对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为

什么时候使用：代码中包含大量与对象状态有关的条件语句

如何解决 ：将各种具体的状态类抽象出来

结构图：



关键代码：通常命令模式的接口中只有一个方法. 而状态模式的接口中有1个或者多个方法.而且，状态模式的实现类的方法，一般返回值；或者是改变实例变量的值.也就是说，状态模式一般和对象的状态有关.实现类的方法有不同的功能，覆盖接口中的方法.状态模式和命令模式一样，也可以用于消除if...else等条件选择语句.

应用实例：1、打篮球的时候运动员可以有正常状态，不正常状态，和超常状态 2、曾侯乙编钟中，‘钟是抽象接口’，‘钟A’等是具体状态，'曾侯乙编钟'是具体环境(Context)

优点：1、封装了转换规则。 2、枚举可能的状态，在枚举状态之前需要确定状态种类。 3、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。 4、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。 5、可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

缺点：1、状态模式的使用必然会增加系统类和对象的个数。 2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。 3、状态模式对“开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态；而且修改某个状态类的行为也需修改对应类的源代码。

使用场景：1、行为岁状态改变而改变的场景 2、条件、分支语句的代替者

注意事项：在行为受状态约束的时候使用状态模式，而且状态不超过5个

21.访问者模式

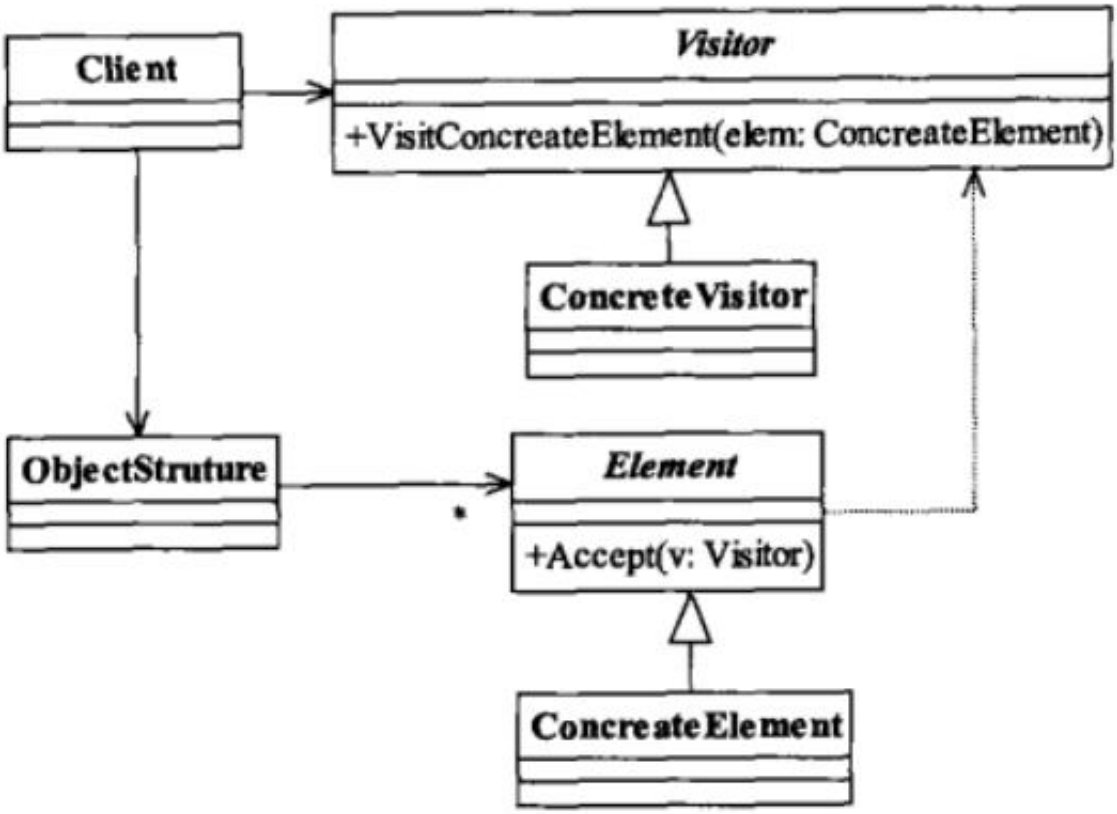
意图：主要讲数据结构与数据操作分离

主要解决：稳定的数据结构和易变的操作耦合问题

什么时候使用：与类本不相关的，为了避免这个污染 ，使用访问者模式将这些封装到访问者模式

如何解决 ：在被访问的类里面加一个对外提供接待访问者的接口

结构图：



关键代码：在数据基础类里面有一个方法接受访问者，将自身引用传入访问者

应用实例：你在朋友家做客，你是访问者，朋友接受你的访问，你通过朋友的描述，然后对朋友的 描述做出一个判断，这就是访问者模式

优点：1、符合单一职责原则。 2、优秀的扩展性。3、灵活性

缺点：1、具体元素对访问者公布细节，违反了迪米特原则2、具体元素变更比较困难 3、违反了依赖倒置原则，依赖了具体类，没有依赖抽象

使用场景：1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。 2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作“污染”这些对象的类，也不希望在增加新操作时修改这些类

注意事项：访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器

22.解释器模式

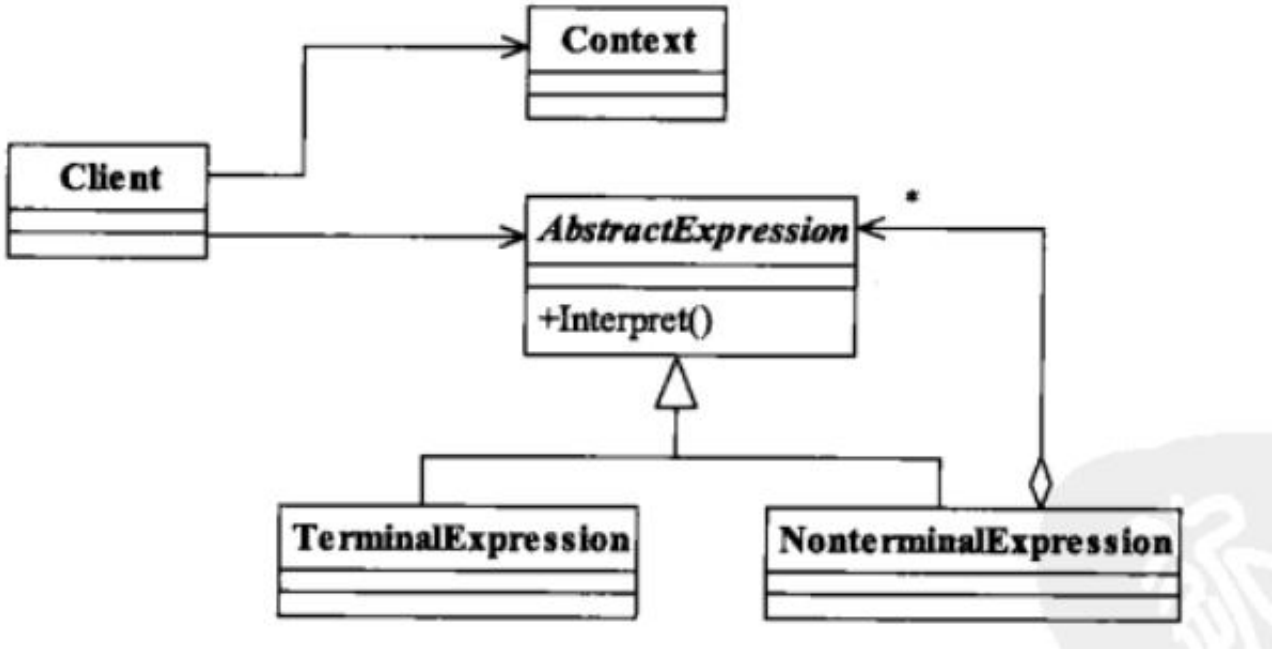
意图：给定一个语言， 定义它的文法表示， 并定 义一个解释器， 这个解释器使用该标识来解释语言中的句子。

主要解决：对于一些固定文法构建一个解释句子的解释器

什么时候使用：如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子.这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题.

如何解决 ：构件语法树，定义终结符与非终结符

结构图：



关键代码：构件环境类，包含解释器之外的一些全局信息，一般是HashMap

应用实例：编译器、运算表达式计算

优点： 1、 可扩展性比较好，灵活。 2、 增加了新的解释表达式的方式 3、 易于简单实现文法。

缺点： 1、可利用场景比较少 2、 对于复杂的文法比较难维护。3、解释器模式会引起类膨胀 4、解释器模式采用递归调用方法

使用场景：1、可以将一个需要解释执行的语言中的句子表示为一个抽象语法树 2、一些重复出现的问题可以用一种简单的语言来进行表达 3、一个简单语法需要解释的场景

注意事项：可利用场景比较少,JAVA中如果碰到可以用expression4J代替

23.中介者模式

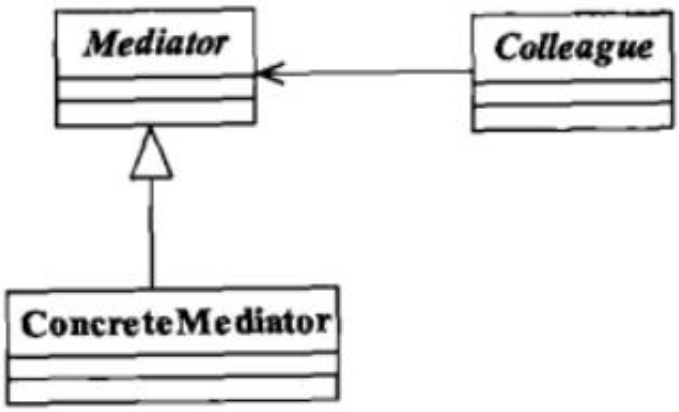
意图：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

主要解决：对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理

什么时候使用：多个类相互耦合，形成了网状结构

如何解决 ：将上述网状结构分离为星型结构

结构图：



关键代码：对象 Colleague 之间的通信封装到一个类种单独处理

应用实例：1 、中国加入WTO，之前是各个国家相互贸易，结构复杂，现在是各个通过WTO来互相贸易 2、机场调度系统 3、 MVC框架，其中C(控制器)就是M和V的中介者

优点： 1、降低了类的复杂度将一对多转化成了一对一 2、各个类之间的解耦 3、符合迪米特原则

缺点： 1、中介者会庞大，变得复杂难以维护

使用场景：1、 系统中对象之间存在比较复杂的引用关系，导致他们之间的依赖关系结构混乱而且难以复用该对象。 2、 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

注意事项：不应当在职责混乱的时候使用