深入浅出设计模式

作者：AI92　　yuanyk@gmail.com

工厂模式（静态工厂模式、工厂方法模式、抽象工厂模式）

一、引子

话说十年前，有一个暴发户，他家有三辆汽车——Benz 奔驰、Bmw 宝马、Audi 奥迪，还雇了司机为他开车。不过，暴发户坐车时总是怪怪的：上 Benz 车后跟司机说“开奔驰车！”，坐上 Bmw 后他说“开宝马车！”，坐上 Audi 说“开奥迪车！”。你一定说：这人有病！直接说开车不就行了？！

而当把这个暴发户的行为放到我们程序设计中来时，会发现这是一个普遍存在的现象。幸运的是，这种有病的现象在 OO（面向对象）语言中可以避免了。下面就以 Java 语言为基础来引入我们本文的主题：工厂模式。

二、分类

工厂模式主要是为创建对象提供过渡接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。

工厂模式在《Java 与模式》中分为三类：

1）简单工厂模式（Simple Factory）

2）工厂方法模式（Factory Method）

3）抽象工厂模式（Abstract Factory）

这三种模式从上到下逐步抽象，并且更具一般性。

GOF 在《设计模式》一书中将工厂模式分为两类：工厂方法模式（Factory Method）与抽象工厂模式（Abstract Factory）。将简单工厂模式（Simple Factory）看为工厂方法模式的一种特例，两者归为一类。

两者皆可，在本文使用《Java 与模式》的分类方法。下面来看看这些工厂模式是怎么来“治病”的。

三、简单工厂模式

简单工厂模式又称静态工厂方法模式。重命名上就可以看出这个模式一定很简单。它存在的目的很简单：定义一个用于创建对象的接口。

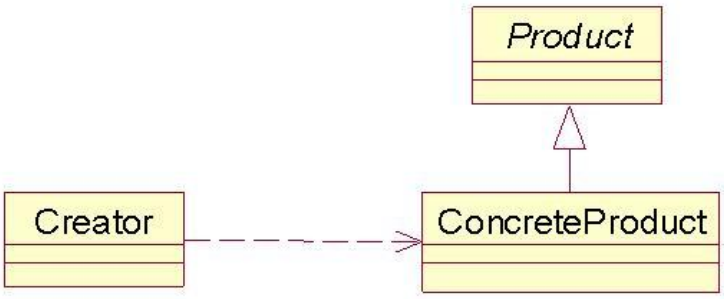
先来看看它的组成：

1) 工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑。在java中它往往由一个具体类实现。

2) 抽象产品角色：它一般是具体产品继承的父类或者实现的接口。在java中由接口或者抽象类来实现。

3) 具体产品角色：工厂类所创建的对象就是此角色的实例。在java中由一个具体类实现。

来用类图来清晰的表示下的它们之间的关系（如果对类图不太了解，请参考我关于类图的文章）：



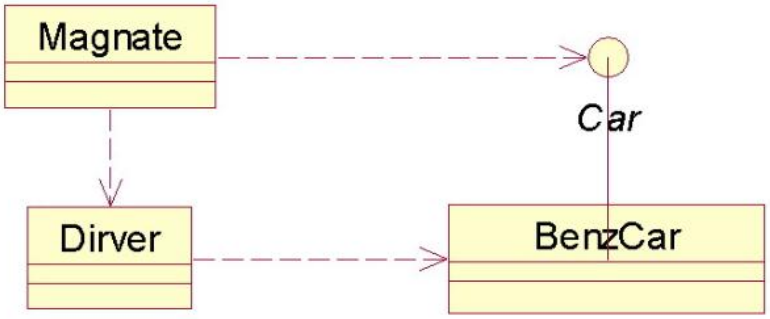
那么简单工厂模式怎么来使用呢？我们就以简单工厂模式来改造暴发户坐车的方式——现在暴发户只需要坐在车里对司机说句："开车"就可以了。

```
//抽象产品角色
public interface Car{
    public void drive();
}

//具体产品角色
public class Benz implements Car{
    public void drive()  {
        System.out.println("Driving Benz ");
    }
}
public class Bmw implements Car{
    public void drive()  {
        System.out.println("Driving Bmw ");
    }
}
。。。（奥迪我就不写了:P）

//工厂类角色
public class Driver{
        //工厂方法.注意 返回类型为抽象产品角色
        public static Car driverCar(String s)throws Exception  {
        //判断逻辑，返回具体的产品角色给 Client
        if(s.equalsIgnoreCase("Benz"))
            return new Benz();
        else if(s.equalsIgnoreCase("Bmw"))
            return new Bmw();
             ......
        else throw new Exception();
    。。。

//欢迎暴发户出场.....
```

```
public class Magnate{
    public static void main(String[] args){
        try{
            //告诉司机我今天坐奔驰
            Car car = Driver.driverCar("benz");
            //下命令：开车
            car.drive();
        。。。
```

将本程序空缺的其他信息填充完整后即可运行。如果你将所有的类放在一个文件中，请不要忘记只能有一个类被声明为 public。本程序在 jdk1.4 下运行通过。

程序中各个类的关系表达如下：



这便是简单工厂模式了。怎么样，使用起来很简单吧？那么它带来了什么好处呢？

首先，使用了简单工厂模式后，我们的程序不在"有病"，更加符合现实中的情况；而且客户端免除了直接创建产品对象的责任，而仅仅负责"消费"产品（正如暴发户所为）。

下面我们从开闭原则（对扩展开放；对修改封闭）上来分析下简单工厂模式。当暴发户增加了一辆车的时候，只要符合抽象产品制定的合同，那么只要通知工厂类知道就可以被客户使用了。所以对产品部分来说，它是符合开闭原则的；但是工厂部分好像不太理想，因为每增加一辆车，都要在工厂类中增加相应的业务逻辑或者判断逻辑，这显然是违背开闭原则的。可想而知对于新产品的加入，工厂类是很被动的。对于这样的工厂类（在我们的例子中是为司机师傅），我们称它为全能类或者上帝类。

我们举的例子是最简单的情况，而在实际应用中，很可能产品是一个多层次的树状结构。由于简单工厂模式中只有一个工厂类来对应这些产品，所以这可能会把我们的上帝累坏了，也累坏了我们这些程序员:(

于是工厂方法模式作为救世主出现了。

四、工厂方法模式

工厂方法模式去掉了简单工厂模式中工厂方法的静态属性，使得它可以被子类继承。这样在简单工厂模式里集中在工厂方法上的压力可以由工厂方法模式里不同的工厂子类来分担。

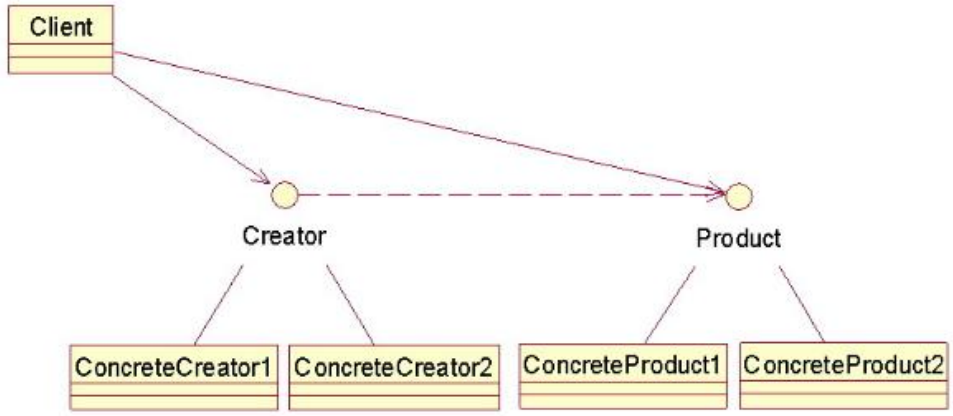
你应该大致猜出了工厂方法模式的结构，来看下它的组成：

1) 抽象工厂角色： 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在 java 中它由抽象类或者接口来实现。
2) 具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。

3) 抽象产品角色：它是具体产品继承的父类或者是实现的接口。在 java 中一般有抽象类或者接口来实现。

4) 具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在 java 中由具体的类来实现。

用类图来清晰的表示下的它们之间的关系：



工厂方法模式使用继承自抽象工厂角色的多个子类来代替简单工厂模式中的"上帝类"。正如上面所说，这样便分担了对象承受的压力；而且这样使得结构变得灵活起来——当有新的产品（即暴发户的汽车）产生时，只要按照抽象产品角色、抽象工厂角色提供的合同来生成，那么就可以被客户使用，而不必去修改任何已有的代码。可以看出工厂角色的结构也是符合开闭原则的！

我们还是老规矩，使用一个完整的例子来看看工厂模式各个角色之间是如何来协调的。话说暴发户生意越做越大，自己的爱车也越来越多。这可苦了那位司机师傅了，什么车它都要记得，维护，都要经过他来使用！于是暴发户同情他说：看你跟我这么多年的份上，以后你不用这么辛苦了，我给你分配几个人手，你只管管好他们就行了！于是，工厂方法模式的管理出现了。代码如下：

```
//抽象产品角色，具体产品角色与简单工厂模式类似，只是变得复杂了些，这里略。
//抽象工厂角色
public interface Driver{
    public Car driverCar();
}
public class BenzDriver implements Driver{
    public Car driverCar(){
        return new Benz();
    }
}
public class BmwDriver implements Driver{
    public Car driverCar() {
        return new Bmw();
    }
}
//应该和具体产品形成对应关系...
//有请暴发户先生
 public class Magnate
```

```
{
    public static void main(String[] args)
    {
        try{
            Driver driver = new BenzDriver();
            Car car = driver.driverCar();
            car.drive();
        }
.....
}
```

可以看出工厂方法的加入，使得对象的数量成倍增长。当产品种类非常多时，会出现大量的与之对应的工厂对象，这不是我们所希望的。因为如果不能避免这种情况，可以考虑使用简单工厂模式与工厂方法模式相结合的方式来减少工厂类：即对于产品树上类似的种类（一般是树的叶子中互为兄弟的）使用简单工厂模式来实现。

五、小结

工厂方法模式仿佛已经很完美的对对象的创建进行了包装，使得客户程序中仅仅处理抽象产品角色提供的接口。那我们是否一定要在代码中遍布工厂呢？大可不必。也许在下面情况下你可以考虑使用工厂方法模式：

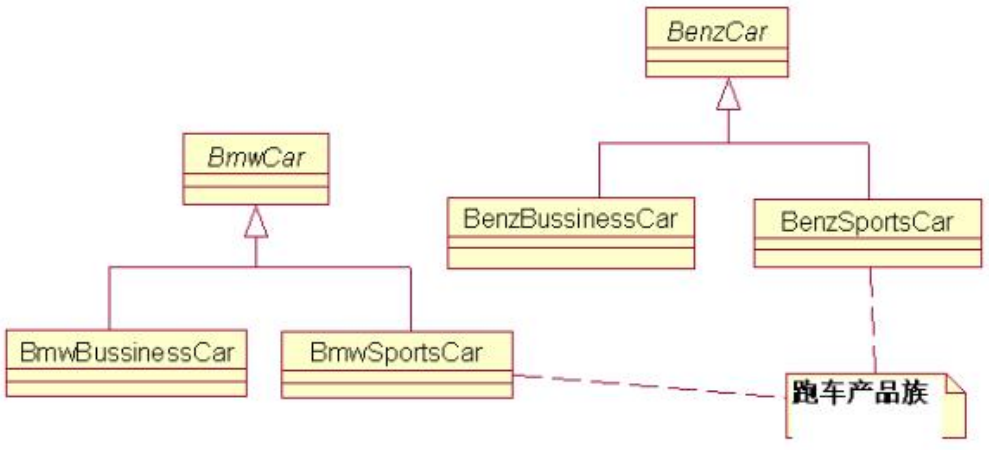
1) 当客户程序不需要知道要使用对象的创建过程。
2) 客户程序使用的对象存在变动的可能，或者根本就不知道使用哪一个具体的对象。

简单工厂模式与工厂方法模式真正的避免了代码的改动了？没有。在简单工厂模式中，新产品的加入要修改工厂角色中的判断语句；而在工厂方法模式中，要么将判断逻辑留在抽象工厂角色中，要么在客户程序中将具体工厂角色写死（就象上面的例子一样）。而且产品对象创建条件的改变必然会引起工厂角色的修改。

面对这种情况，Java 的反射机制与配置文件的巧妙结合突破了限制——这在 Spring 中完美的体现了出来。

六、抽象工厂模式

先来认识下什么是产品族： 位于不同产品等级结构中，功能相关联的产品组成的家族。还是让我们用一个例子来形象地说明一下吧。



图中的 BmwCar 和 BenzCar 就是两个产品树（产品层次结构）；而如图所示的 BenzSportsCar 和 BmwSportsCar 就是一个产品族。他们都可以放到跑车家族中，因此功能有所关联。同理 BmwBussinessCar 和 BenzSportsCar 也是一个产品族。

回到抽象工厂模式的话题上。

可以说，抽象工厂模式和工厂方法模式的区别就在于需要创建对象的复杂程度上。而且抽象工厂模式是三个里面最为抽象、最具一般性的。

抽象工厂模式的用意为：给客户端提供一个接口，可以创建多个产品族中的产品对象

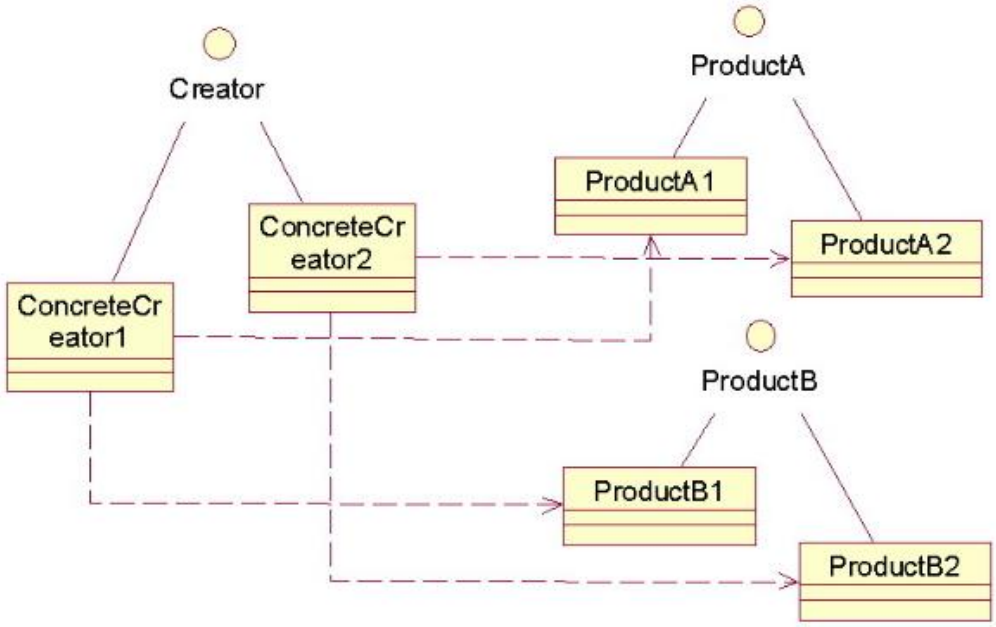
而且使用抽象工厂模式还要满足一下条件：

1) 系统中有多个产品族，而系统一次只可能消费其中一族产品。
2) 同属于同一个产品族的产品以其使用。

来看看抽象工厂模式的各个角色（和工厂方法的如出一辙）：

1) 抽象工厂角色： 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在 java 中它由抽象类或者接口来实现。
2) 具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。在 java 中它由具体的类来实现。
3) 抽象产品角色：它是具体产品继承的父类或者是实现的接口。在 java 中一般有抽象类或者接口来实现。
4) 具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在 java 中由具体的类来实现。

类图如下：



看过了前两个模式，对这个模式各个角色之间的协调情况应该心里有个数了，我就不举具体的例子了。只是一定要注意满足使用抽象工厂模式的条件哦。

单例模式

一、引子

单例模式是设计模式中使用很频繁的一种模式，在各种开源框架、应用系统中多有应用，在我前面的几篇文章中也结合其它模式使用到了单例模式。这里我们就单例模式进行系统的学习。并对有人提出的"单例模式是邪恶的"这个观点进行了一定的分析。

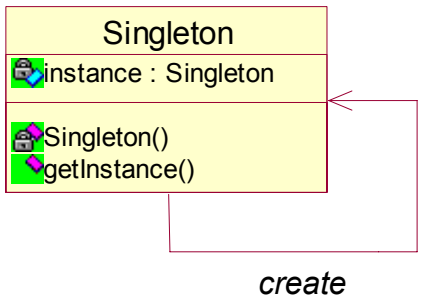
二、定义与结构

单例模式又叫做单态模式或者单件模式。在 GOF 书中给出的定义为：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。单例模式中的"单例"通常用来代表那些本质上具有唯一性的系统组件（或者叫做资源）。比如文件系统、资源管理器等等。

单例模式的目的就是要控制特定的类只产生一个对象，当然也允许在一定情况下灵活的改变对象的个数。那么怎么来实现单例模式呢？一个类的对象的产生是由类构造函数来完成的，如果想限制对象的产生，一个办法就是将构造函数变为私有的（至少是受保护的），使得外面的类不能通过引用来产生对象；同时为了保证类的可用性，就必须提供一个自己的对象以及访问这个对象的静态方法。

现在对单例模式有了大概的了解了吧，其实单例模式在实现上是非常简单的——只有一个角色，而客户则通过调用类方法来得到类的对象。

放上一个类图吧，这样更直观一些：



单例模式可分为有状态的和无状态的。有状态的单例对象一般也是可变的单例对象，多个单态对象在一起就可以作为一个状态仓库一样向外提供服务。没有状态的单例对象也就是不变单例对象，仅用做提供工具函数。

三、实现

在单例模式的实现上有几种不同的方式，我在这里将一一讲解。先来看一种方式，它在《java 与模式》中被称为饿汉式。

```
public class Singleton {
    //在自己内部定义自己一个实例
    //注意这是 private 只供内部调用
    private static Singleton instance = new Singleton();
    //如上面所述，将构造函数设置为私有
    private Singleton(){
    }
    //静态工厂方法，提供了一个供外部访问得到对象的静态方法
```

```
    public static Singleton getInstance() {
        return instance;
    }
}
```

下面这种方式被称为懒汉式：P

```
public class Singleton {
    //和上面有什么不同？
    private static Singleton instance = null;
    //设置为私有的构造函数
    private Singleton(){
    }
    //静态工厂方法
    public static synchronized Singleton getInstance() {
    //这个方法比上面有所改进
        if (instance==null)
            instance＝new Singleton();
        return instance;
    }
}
```

先让我们来比较一下这两种实现方式。

首先他们的构造函数都是私有的，彻底断开了使用构造函数来得到类的实例的通道，但是这样也使得类失去了多态性（大概这就是为什么有人将这种模式称作单态模式）。

在第二种方式中，对静态工厂方法进行了同步处理，原因很明显——为了防止多线程环境中产生多个实例；而在第一种方式中则不存在这种情况。

在第二种方式中将类对自己的实例化延迟到第一次被引用的时候。而在第一种方式中则是在类被加载的时候实例化，这样多次加载会照成多次实例化。但是第二种方式由于使用了同步处理，在反应速度上要比第一种慢一些。

在 《java 与模式》书中提到，就 java 语言来说，第一种方式更符合 java 语言本身的特点。

以上两种实现方式均失去了多态性，不允许被继承。还有另外一种灵活点的实现，将构造函数设置为受保护的，这样允许被继承产生子类。这种方式在具体实现上又有所不同，可以将父类中获得对象的静态方法放到子类中再实现；也可以在父类的静态方法中进行条件判断来决定获得哪一个对象；在 GOF 中认为最好的一种方式是维护一张存有对象和对应名称的注册表（可以使用 HashMap 来实现）。下面的实现参考《java 与模式》采用带有注册表的方式。

```
import java.util.HashMap;

public class Singleton
{
    //用来存放对应关系
```

```
    private static HashMap sinRegistry = new HashMap();
    static private Singleton s = new Singleton();
    //受保护的构造函数
    protected Singleton()
    {}
    public static Singleton getInstance(String name)
    {
        if(name == null)
            name = "Singleton";
        if(sinRegistry.get(name)==null)
        {
            try{
                sinRegistry.put(name , Class.forName(name).newInstance());
            }catch(Exception e)
            {
                e.printStackTrace();
            }
        }
        return (Singleton)(sinRegistry.get(name));
    }
    public void test()
    {
        System.out.println("getclasssuccess!");
    }
}

public class SingletonChild1 extends Singleton
{
    public SingletonChild1(){}
    static    public SingletonChild1 getInstance()
    {
        return (SingletonChild1)Singleton.getInstance("SingletonChild1");
    }
    public void test()
    {
        System.out.println("getclasssuccess111!");
    }
}
```

由于在 java 中子类的构造函数的范围不能比父类的小，所以可能存在不守规则的客户程序使用其构造函数来产生实例，造成单例模式失效。

四、单例模式邪恶论

看这题目也许有点夸张，不过这对初学者是一个很好的警告。单例模式在 java 中的使

用存在很多陷阱和假象，这使得没有意识到单例模式使用局限性的你在系统中布下了隐患……

其实这个问题早在 2001 年的时候就有人在网上系统的提出来过，我在这里只是老生常谈了。但是对于大多的初学者来说，可能这样的观点在还很陌生。下面我就一一列举出单例模式在 java 中存在的陷阱。

多个虚拟机

当系统中的单例类被拷贝运行在多个虚拟机下的时候，在每一个虚拟机下都可以创建一个实例对象。在使用了 EJB、JINI、RMI 技术的分布式系统中，由于中间件屏蔽掉了分布式系统在物理上的差异，所以对你来说，想知道具体哪个虚拟机下运行着哪个单例对象是很困难的。

因此，在使用以上分布技术的系统中，应该避免使用存在状态的单例模式，因为一个有状态的单例类，在不同虚拟机上，各个单例对象保存的状态很可能是不一样的，问题也就随之产生。而且在 EJB 中不要使用单例模式来控制访问资源，因为这是由 EJB 容器来负责的。在其它的分布式系统中，当每一个虚拟机中的资源是不同的时候，可以考虑使用单例模式来进行管理。

多个类加载器

当存在多个类加载器加载类的时候，即使它们加载的是相同包名，相同类名甚至每个字节都完全相同的类，也会被区别对待的。因为不同的类加载器会使用不同的命名空间（namespace）来区分同一个类。因此，单例类在多加载器的环境下会产生多个单例对象。

也许你认为出现多个类加载器的情况并不是很多。其实多个类加载器存在的情况并不少见。在很多 J2EE 服务器上允许存在多个 servlet 引擎，而每个引擎是采用不同的类加载器的；浏览器中 applet 小程序通过网络加载类的时候，由于安全因素，采用的是特殊的类加载器，等等。

这种情况下，由状态的单例模式也会给系统带来隐患。因此除非系统由协调机制，在一般情况下不要使用存在状态的单例模式。

错误的同步处理

在使用上面介绍的懒汉式单例模式时，同步处理的恰当与否也是至关重要的。不然可能会达不到得到单个对象的效果，还可能引发死锁等错误。因此在使用懒汉式单例模式时一定要对同步有所了解。不过使用饿汉式单例模式就可以避免这个问题。

子类破坏了对象控制

在上一节介绍最后一种扩展性较好的单例模式实现方式的时候，就提到，由于类构造函数变得不再私有，就有可能失去对对象的控制。这种情况只能通过良好的文档来规范。

串行化（可序列化）

为了使一个单例类变成可串行化的，仅仅在声明中添加“implements Serializable”是不够的。因为一个串行化的对象在每次返串行化的时候，都会创建一个新的对象，而不仅仅是一个对原有对象的引用。为了防止这种情况，可以在单例类中加入 readResolve 方法。关于这个方法的具体情况请参考《Effective Java》一书第 57 条建议。

其实对象的串行化并不仅局限于上述方式，还存在基于 XML 格式的对象串行化方式。这种方式也存在上述的问题，所以在使用的时候要格外小心。

上面罗列了一些使用单例模式时可能会遇到的问题。而且这些问题都和 java 中的类、线程、虚拟机等基础而又复杂的概念交织在一起，你如果稍不留神……。但是这并不代表着单例模式就一无是处，更不能一棒子将其打死。它还是不可缺少的一种基础设计模式，它对一些问题提供了非常有效的解决方案，在 java 中你完全可以把它看成编码规范来学习，只是使用的时候要考虑周全些就可以了。

五、题外话

抛开单例模式，使用下面一种简单的方式也能得到单例，而且如果你确信此类永远是单例的，使用下面这种方式也许更好一些。

```
public static final Singleton INSTANCE = new Singleton();
```

而使用单例模式提供的方式，这可以在不改变 API 的情况下，改变我们对单例类的具体要求。

建造模式

一、引子

前几天陪朋友去装机店攒了一台电脑，看着装机工在那里熟练的装配着机器，不禁想起来了培训时讲到的建造模式。作为装机工，他们不用管你用的 CPU 是 Intel 还是 AMD，也不管你的显卡是 2000 千大元还是白送的，都能三下五除二的装配在一起——一台 PC 就诞生了！当然对于客户来说，你也不知道太多关于 PC 组装的细节。这和建造模式是多么的相像啊！

今天就来探讨一下建造模式

二、定义与结构

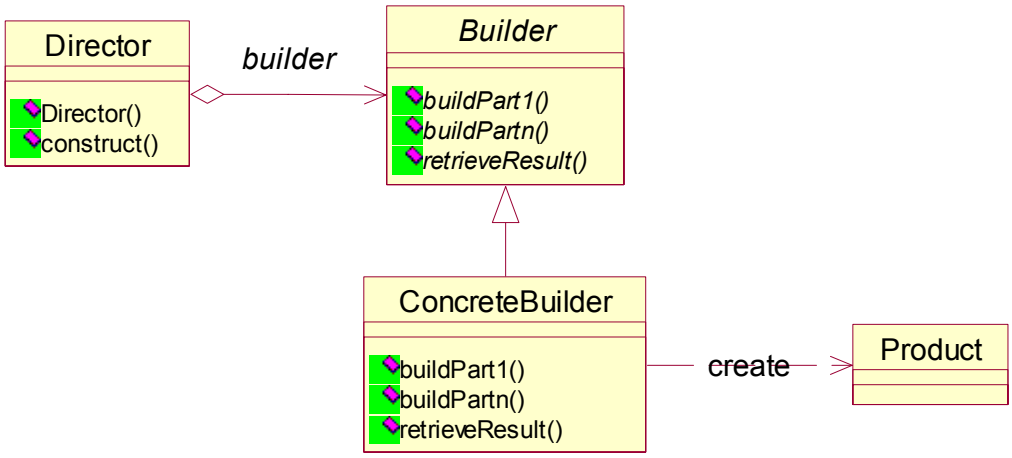
GOF 给建造模式的定义为：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。这句话说得很抽象，不好理解，其实它的意思可以理解为：将构造复杂对象的过程和组成对象的部件解耦。就像攒电脑一样，不管什么品牌的配件，只要兼容就可以装上；同样，一样的配件，可以有好多组装的方式。这是对降低耦合、提高可复用性精神的一种贯彻。

当要生成的产品有复杂的内部结构——比如由多个对象组成；而系统中对此产品的需求将来可能要改变产品对象的内部结构的构成，比如说产品的一些属性现在由一个小对象组成，而更改后的型号可能需要 N 个小对象组成；而且不能将产品的内部构造完全暴露给客户程序，一是为了可用性，二是为了安全等因素。满足上面的设计环境就可以考虑使用建造模式来搭建框架了。

来看看建造模式的组成吧。

1) 抽象建造者角色：这个角色用来规范产品对象的各个组成成分的建造。一般而言，此角色独立于应用程序的业务逻辑。
2) 具体建造者角色：担任这个角色的是于应用程序紧密相关的类，它们在指导者的调用下创建产品实例。这个角色在实现抽象建造者角色提供的方法的前提下，达到完成产品组装，提供成品的功能。
3) 指导者角色：调用具体建造者角色以创建产品对象。指导者并没有产品类的具体知识，真正拥有产品类的具体知识的是具体建造者对象。
4) 产品角色：建造中的复杂对象。它要包含那些定义组件的类，包括将这些组件装配成产品的接口。

来看下这些角色组成的类图：



首先客户程序创建一个指导者对象，一个建造者角色，并将建造者角色传入指导者对象进行配置。然后，指导者按照步骤调用建造者的方法创建产品。最后客户程序从建造者或者指导者那里得到产品。

从建造模式的工作流程来看，建造模式将产品的组装"外部化"到了建造者角色中来。这是和任何正规的工厂模式不一样的——产品的创建是在产品类中完成的。

三、实现

没有找到好的例子，觉得《java 与模式》中发邮件的例子有点牵强。于是我将 Bruce Eckel 在《Think in Patterns with Java》中用的例子放到这里权且充个门面。我们知道媒体可以存在不同的表达形式，比如书籍、杂志和网络。这个例子表示不同形式的媒体构造的步骤是相似的，所以可以被提取到指导者角色中去。

```
import java.util.*;
import junit.framework.*;

//不同的媒体形式:
class Media extends ArrayList {}
class Book extends Media {}
class Magazine extends Media {}
class WebSite extends Media {}

// 进而包含不同的媒体组成元素:
class MediaItem {
    private String s;
    public MediaItem(String s) { this.s = s; }
    public String toString() { return s; }
}
class Chapter extends MediaItem {
    public Chapter(String s) { super(s); }
}
class Article extends MediaItem {
    public Article(String s) { super(s); }
```

```
}
class WebItem extends MediaItem {
    public WebItem(String s) { super(s); }
}

// 抽象建造者角色，它规范了所有媒体建造的步骤:
class MediaBuilder {
    public void buildBase() {}
    public void addMediaItem(MediaItem item) {}
    public Media getFinishedMedia() { return null; }
}
//具体建造者角色
class BookBuilder extends MediaBuilder {
    private Book b;
    public void buildBase() {
        System.out.println("Building book framework");
        b = new Book();
    }
    public void addMediaItem(MediaItem chapter) {
        System.out.println("Adding chapter " + chapter);
        b.add(chapter);
    }
    public Media getFinishedMedia() { return b; }
}

class MagazineBuilder extends MediaBuilder {
    private Magazine m;
    public void buildBase() {
        System.out.println("Building magazine framework");
        m = new Magazine();
    }
    public void addMediaItem(MediaItem article) {
        System.out.println("Adding article " + article);
        m.add(article);
    }
    public Media getFinishedMedia() { return m; }
}

class WebSiteBuilder extends MediaBuilder {
    private WebSite w;
    public void buildBase() {
        System.out.println("Building web site framework");
        w = new WebSite();
    }
```

```
    public void addMediaItem(MediaItem webItem) {
        System.out.println("Adding web item " + webItem);
        w.add(webItem);
    }
    public Media getFinishedMedia() { return w; }
}
//指导者角色，也叫上下文
class MediaDirector {
    private MediaBuilder mb;
    public MediaDirector(MediaBuilder mb) {
        this.mb = mb; //具有策略模式相似特征的
    }
    public Media produceMedia(List input) {
        mb.buildBase();
        for(Iterator it = input.iterator(); it.hasNext();)
            mb.addMediaItem((MediaItem)it.next());
        return mb.getFinishedMedia();
    }
}

//测试程序——客户程序角色
public class BuildMedia extends TestCase {
    private List input = Arrays.asList(new MediaItem[] {
        new MediaItem("item1"), new MediaItem("item2"),
        new MediaItem("item3"), new MediaItem("item4"),
    });
    public void testBook() {
        MediaDirector buildBook = new MediaDirector(new BookBuilder());
        Media book = buildBook.produceMedia(input);
        String result = "book: " + book;
        System.out.println(result);
        assertEquals(result, "book: [item1, item2, item3, item4]");
    }
    public void testMagazine() {
        MediaDirector buildMagazine = new MediaDirector(new MagazineBuilder());
        Media magazine = buildMagazine.produceMedia(input);
        String result = "magazine: " + magazine;
        System.out.println(result);
        assertEquals(result, "magazine: [item1, item2, item3, item4]");
    }
    public void testWebSite() {
        MediaDirector buildWebSite = new MediaDirector(new WebSiteBuilder());
        Media webSite = buildWebSite.produceMedia(input);
        String result = "web site: " + webSite;
```

```
            System.out.println(result);
            assertEquals(result, "web site: [item1, item2, item3, item4]");
        }
        public static void main(String[] args) {
            junit.textui.TestRunner.run(BuildMedia.class);
        }
    }
```

四、应用优点

建造模式可以使得产品内部的表象独立变化。在原来的工厂方法模式中，产品内部的表象是由产品自身来决定的；而在建造模式中则是"外部化"为由建造者来负责。这样定义一个新的具体建造者角色就可以改变产品的内部表象，符合"开闭原则"。

建造模式使得客户不需要知道太多产品内部的细节。它将复杂对象的组建和表示方式封装在一个具体的建造角色中，而且由指导者来协调建造者角色来得到具体的产品实例。

每一个具体建造者角色是毫无关系的。

建造模式可以对复杂产品的创建进行更加精细的控制。产品的组成是由指导者角色调用具体建造者角色来逐步完成的，所以比起其它创建型模式能更好的反映产品的构造过程。

五、扩展

建造模式中很可能要用到组成成品的各种组件类，对于这些类的创建可以考虑使用工厂方法或者原型模式来实现，在必要的时候也可以加上单例模式来控制类实例的产生。但是要坚持一个大前提就是要使引入的模式给你的系统带来好处，而不是臃肿的结构。

建造模式在得到复杂产品的时候可能要引用多个不同的组件，在这一点上来看，建造模式和抽象工厂模式是相似的。可以从以下两点来区分两者：创建模式着重于逐步将组件装配成一个成品并向外提供成品，而抽象工厂模式着重于得到产品族中相关的多个产品对象；抽象工厂模式的应用是受限于产品族的（具体参见《深入浅出工厂模式》），建造模式则不会。

由于建造模式和抽象工厂模式在实现功能上相似，所以两者使用的环境都比较复杂并且需要更多的灵活性。

组合模式中的树枝构件角色（Composite）往往是由多个树叶构件角色（Leaf）组成，因此树枝构件角色的产生可以由建造模式来担当。

原型模式

一、引子

古人云：书非借不能读也。我深谙古人教诲，更何况现在 IT 书籍更新快、价格贵、质量水平更是参差不齐，实在不忍心看到用自己的血汗钱买的书不到半年就要被淘汰，更不想供养使用金山快译、词霸等现代化工具的翻译们。于是我去书店办了张借书卡，这样便没有了后顾之忧了——书不好我可以换嘛！

但是，借书也有不爽的地方，就是看到有用或者比较重要的地方，不能在书旁标记下来。一般我会将这页内容复印下来，这样作为我自己的东西就可以对其圈圈画画，保存下来了。

在软件设计中，往往也会遇到类似或者相似的问题，GOF 将这种解决方案叫作原型模式。也许原形模式会给你一些新的启迪。

二、定义与结构

原型模式属于对象创建模式，GOF 给它的定义为：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。

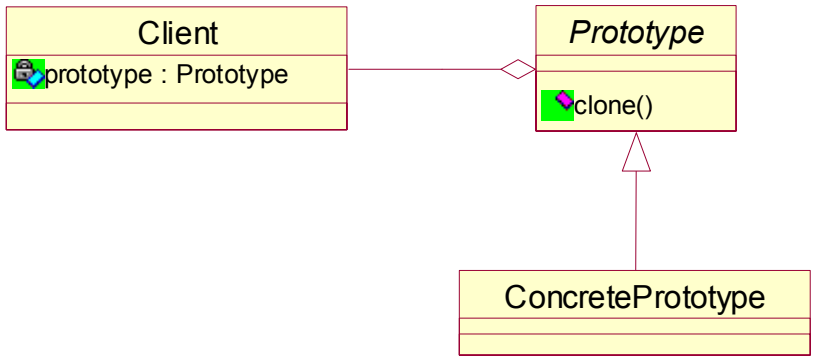
在 Java 中提供了 clone()方法来实现对象的克隆，所以 Prototype 模式实现变得简单许多。注：clone()方法的使用，请参考《Thinking in Java》或者《Effective Java》，对于许多原型模式中讲到的浅克隆、深克隆，本文不作为谈论话题。

使用克隆方式来创建对象与同样用来创建对象的工厂模式有什么不同？前面已经提过工厂模式对新产品的适应能力比较弱：创建新的产品时，就必须修改或者增加工厂角色。而且为了创建产品对象要先额外的创建一个工厂对象。那通过原型模式来创建对象会是什么样子呢？

先让我们来看看原型模式的结构吧。

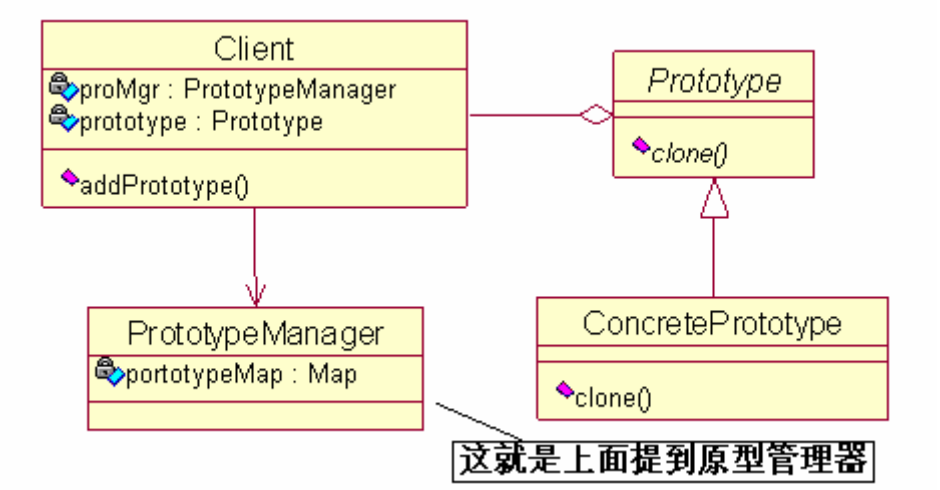
1) 客户角色：让一个原型克隆自己来得到一个新对象。
2) 抽象原型角色：实现了自己的 clone 方法，扮演这种角色的类通常是抽象类，且它具有许多具体的子类。
3) 具体原型角色：被复制的对象，为抽象原型角色的具体子类。

放上一张类图：



按照定义客户角色不仅要负责使用对象，而且还要负责对象原型的生成和克隆。这样造成客户角色分工就不是很明确，所以我们把对象原型生成和克隆功能单拿出来放到一个原型管理器中。原型管理器维护了已有原型的清单。客户在使用时会向原型管理器发出请求，而且可以修改原型管理器维护的清单。这样客户不需要编码就可以实现系统的扩展。

类图表示如下：



三、分析

对于抽象原型角色和具体原型角色，它们就是一个继承或者实现关系，没有什么好讲的，记住实现好 clone 方法就好了。

那么客户是怎么来使用这些角色的对象的呢？最简单的方式就是：

```
//先 new 一个具体原型角色作为样本
Prototype p = new ConcretePrototype();
……
//使用原型 p 克隆出一个新对象 p1
Prototype p1 = (Prototype)p.clone();
```

当然这只是简单的表述原型模式的运行过程。实际运用中，客户程序与原型角色之间往往存在一个原型管理器（例子见下）。因此创建原型角色、拷贝原型角色就与客户程序分离开来。这时才能真正的体会到原型模式带给我们的效果。

```
//使用原型管理器后，客户获得对象的方式
Prototype p1 = PrototypeManager. getManager().getPrototype("ConcretePrototype");
```

上面提到的原型管理器的实现，简单来说就是对原型清单的维护。可以考虑一下几点：要保存一个原型对象的清单，我们可以使用一个 HashMap 来实现，使原型对象和它的名字相对应；原型管理器只需要一个就够了，所以可以使用单例模式来实现控制；实现得到、注册、删除原型对象的功能只是对 HashMap 的对应操作而已。代码如下：

```
class PrototypeManager {
    private static PrototypeManager pm;
    private Map prototypes=null;
    private PrototypeManager() {
        prototypes=new HashMap();
    }
```

```
//使用单例模式来得到原型管理器的唯一实例
public static PrototypeManager getManager() {
    if(pm==null) {
        pm=new PrototypeManager();
    }
    return pm;
}
public void register(String name , Object prototype) {
    prototypes.put(name , prototype);
}
public void unregister(String name) {
    prototypes.remove(name);
}
public Prototype getPrototype(String name) {
    if(prototypes.containsKey(name)) {
        //将清单中对应原型的复制品返回给客户
        return (Prototype) ((Prototype)prototypes.get(name)).clone();
    }else {
        Prototype object=null;
        try {
            object =（Prototype）Class.forName(name).newInstance();
            register(name , object);
        }  catch(Exception e) {
            System.err.println("Class "+name+"没有定义!");
        }
        return object;
。。。
```

这样当客户自定义新的产品对象时，同时向原型管理器注册一个原型对象，而使用的类只需要根据客户的需要来从原型管理器中得到一个对象就可以了。这样就使得功能扩展变得容易些。

原型模式与其它创建型模式有着相同的特点：它们都将具体产品的创建过程进行包装，使得客户对创建不可知。就像上面例子中一样，客户程序仅仅知道一个抽象产品的接口。当然它还有过人之处：

通过增加或者删除原型管理器中注册的对象，可以比其它创建型模式更方便的在运行时增加或者删除产品。

如果一个对象的创建总是由几种固定组件不同方式组合而成；如果对象之间仅仅实例属性不同。将不同情况的对象缓存起来，直接克隆使用。也许这比采用传递参数重新 new 一个对象要来的快一些。

你也许已经发现原型模式与工厂模式有着千丝万缕的联系：原型管理器不就是一个工厂么。当然这个工厂经过了改进（例如上例采用了 java 的反射机制），去掉了像抽象工厂模式或者工厂方法模式那样繁多的子类。因此可以说原型模式就是在工厂模式的基础上加入了克隆方法。

也许你要说：我实在看不出来使用 clone 方法产生对象和 new 一个对象有什么区别；

原型模式使用 clone 能够动态的抽取当前对象运行时的状态并且克隆到新的对象中，新对象就可以在此基础上进行操作而不损坏原有对象；而 new 只能得到一个刚初始化的对象，而在实际应用中，这往往是不够的。

特别当你的系统需要良好的扩展性时，在设计中使用原型模式也是很必要的。比如说，你的系统可以让客户自定义自己需要的类别，但是这种类别的初始化可能需要传递多于已有类别的参数，而这使得用它的类将不知道怎么来初始化它（因为已经写死了），除非对类进行修改。

可见 clone 方法是不能使用构造函数来代替的。

分析了这么多了，举一个使用原型模式较为经典的例子：绩效考核软件要对今年的各种考核数据进行年度分析，而这一组数据是存放在数据库中的。一般我们会将这一组数据封装在一个类中，然后将此类的一个实例作为参数传入分析算法中进行分析，得到的分析结果返回到类中相应的变量中。假设我们决定对这组数据还要做另外一种分析以对分析结果进行比较评定。这时对封装有这组数据的类进行 clone 要比再次连接数据库得到数据好的多。

任何模式都是存在缺陷的。原型模式主要的缺陷就是每个原型必须含有 clone 方法，在已有类的基础上来添加 clone 操作是比较困难的；而且当内部包括一些不支持 copy 或者循环引用的对象时，实现就更加困难了。

四、总结

由于 clone 方法在 java 实现中有着一定的弊端和风险，所以 clone 方法是不建议使用的。因此很少能在 java 应用中看到原型模式的使用。但是原型模式还是能够给我们一些启迪。

适配器模式

一、引子

昨天在给新买的 MP3 充电的时候，发现这款 MP3 播放器只提供了 USB 接口充电的方式，而它所配备的充电器无法直接给 USB 接口充电，聪明的厂商为充电器装上了一个 USB 接口转换器解决了问题。

这个 USB 接口转接器正是我们今天要谈到的适配器。而在软件开发中采用类似于上面方式的编码技巧被称为适配器模式。

二、定义和结构

《设计模式》一书中是这样给适配器模式定义的：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。由引子中给出的例子可知，这个定义描述的功能和现实中的适配器的功能是一致的。

可能你还是不太明白为什么要使用适配器模式。我们来举个例子也许能更直接的解除你的疑惑。

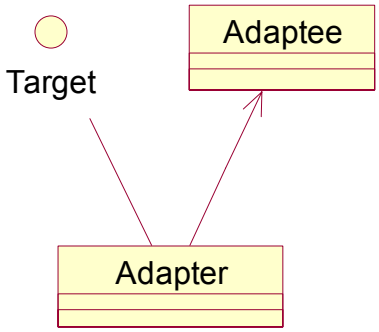
比如，在一个画图的小程序中，你已经实现了绘制点、直线、方块等图形的功能。而且为了让客户程序在使用的时候不用去关心它们的不同，还使用了一个抽象类来规范这些图形的接口。现在你要来实现圆的绘制，这时你发现在系统其他的地方已经有了绘制圆的实现。在你庆幸之余，发现系统中已有的方法和你在抽象类中规定的方法名称不一样！这可怎么办？修改绘制圆的方法名，就要去修改所有使用它的地方；修改你的抽象类的方法名，也要去修改所有图形的实现方法以及已有的引用。还有其它的方法没有？那就是适配器模式了。

可以看出使用适配器模式是为了在面向接口编程中更好的复用。如果你的系统中没有使用到面向接口编程，没有使用到多态，我想大概也不会使用到适配器模式。

下面来看看适配器模式的组成吧。

1) 目标（Target）角色：定义 Client 使用的接口。
2) 被适配（Adaptee）角色：这个角色有一个已存在并使用了的接口，而这个接口是需要我们适配的。
3) 适配器（Adapter）角色：这个适配器模式的核心。它将被适配角色已有的接口转换为目标角色希望的接口。

放上一个简单的类图，这只是适配器模式实现的一种情况：



三、分类

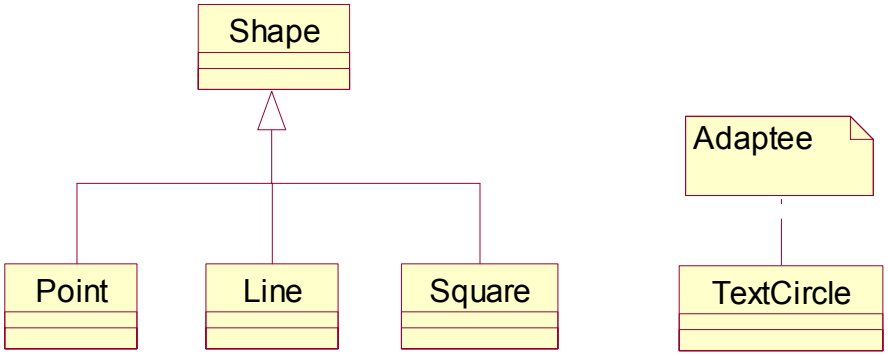
在《设计模式》一书中将适配器模式分为类适配器模式和对象适配器模式。区别仅在于适配器角色对于被适配角色的适配是通过继承完成的还是通过组合来完成的。由于在 java 中不支持多重继承，而且继承有破坏封装之嫌，众多的书中（包括《设计模式》）都提倡使用组合来代

替继承。因此这里我们就不再对类适配器模式进行介绍（其实用的也很少）。

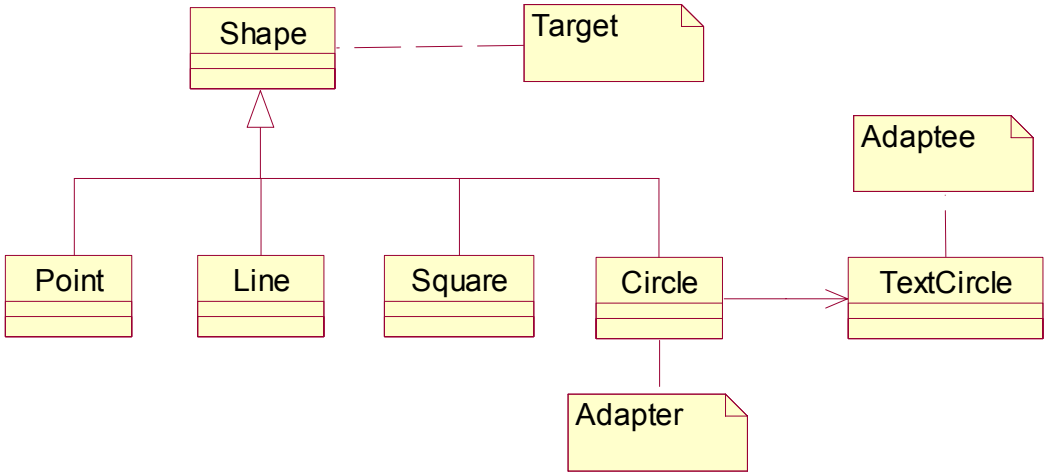
在上一小节的类图中描述的就是对象适配器模式。Adapter 对 Adaptee 的转换是通过组合来完成的（如果你还搞不懂类图中基本元素的含义，请先阅读我的《UML 类图介绍》）。

四、举例

接着上面举的画图程序的例子，先来看看在添加绘制圆的需求前的类结构：



添加了圆的绘制以后的类结构：



可以看出 Shape、Circle 和 TextCircle 三者的关系是和标准适配器模式中 Target、Apater、Apatee 三者的关系相对应的。我们只关心这个画图程序中是怎么来使用适配器模式的。看看 Circle 的实现代码吧：

```
class Circle extends Shape
{
  //这里引用了 TextCircle
  private TextCircle tc;
  public Circle ()
  {
      tc= new TextCircle();  //初始化
  }
  void public display()
```

```
    {
        tc.displayIt();                    //在规定的方法里面调用 TextCircle 原来的方法
    }
}
```

这样一个简单的适配器实现就完成了。

其实在适配器角色中不仅仅可以完成接口转换的过程，而且还可以对其功能进行改进和扩充，当然这就不属于适配器模式描述的范围内了。

前面我介绍过了代理模式，两者的主要区别在于代理模式应用的情况是不改变接口命名的，而且是对已有接口功能的一种控制；而适配器模式则强调接口转换。

五、题外话

在 java 中有一种叫做“缺省适配模式”的应用，它和我们所讲的适配器模式是完全的两种东西。缺省适配模式是为一个接口提供缺省的实现，这样子类型就可以从缺省适配模式中进行扩展，避免了从原有接口中扩展时要实现一些自己不关心的接口。在 java.awt.event 中的 XXXAdapter 就是它的很好的例子，有兴趣的可以看看。

桥梁模式

一、引子

桥梁（bright）模式是我介绍的 23 种模式中的最后一个结构模式。它是一个功能非常强大而且适用于多种情况的模式。

二、定义与结构

GOF 在《设计模式》中给桥梁模式的定义为：将抽象部分与它的实现部分分离，使它们都可以独立地变化。这里的抽象部分和实现部分不是我们通常认为的父类与子类、接口与实现类的关系，而是组合关系。也就是说，实现部分是被抽象部分调用，以用来完成（实现）抽象部分的功能。

在《Thinking in Patterns with Java》一书中，作者将抽象部分叫做“front-end”（权且翻译为“前端”），而实现部分叫做“back-end”（后端）。这种叫法要比抽象实现什么的好理解多了。

系统设计中，总是充满了各种变数，这是防不慎防的。面对这样那样的变动，你只能去不停的修改设计和代码，并且要开始新的一轮测试.....。

那采取什么样的方式可以较好的解决变化带给系统的影响？你可以分析变化的种类，将不变的框架使用抽象类定义出来，然后再将变化的内容使用具体的子类来分别实现。这样面向客户的只是一个抽象类，这种方式可以较好的避免为抽象类中现有接口添加新的实现所带来的影响，缩小了变化带来的影响。但是这可能会造成子类数量的爆炸，并且在某些时候不是很灵活。

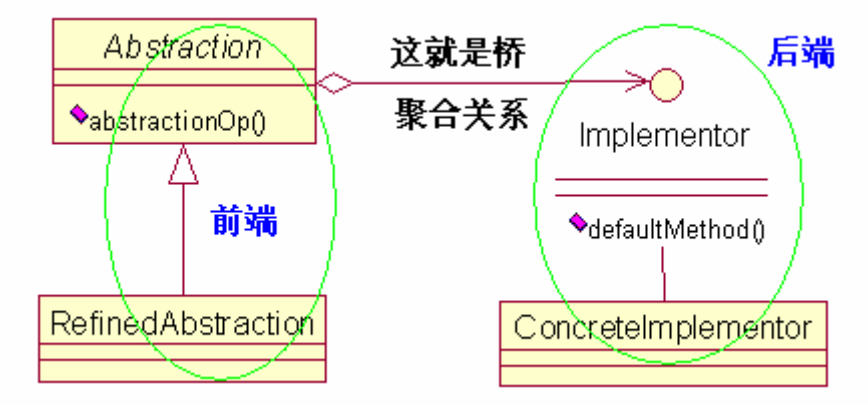
当这颗继承树上一些子树存在了类似的行为。这意味着这些子树中存在了几乎重复的功能代码。这时我们不妨将这些行为提取出来，也采用接口的方式提供出来，然后以组合的方式将服务提供给原来的子类。这样就达到了前端和被使用的后端独立的变化，而且还达到了后端的重用。

其实这就是桥梁模式的诞生。桥梁模式由如下四种角色组成：

1) 抽象（Abstraction）角色：它定义了抽象类的接口而且维护着一个指向实现（Implementor）角色的引用。
2) 精确抽象（RefinedAbstraction）角色：实现并扩充由抽象角色定义的接口。
3) 实现（Implementor）角色：给出了实现类的接口，这里的接口与抽象角色中的接口可以不一致。
4) 具体实现（ConcreteImplementor）角色：给出了实现角色定义接口的具体实现。

它

再放上个类图就更清晰了：



三、实例

我现在唯一知道的使用桥梁模式的应用就是 java AWT 框架。使用过 java AWT 的人都知道，在不同系统下开发的软件界面都带有不同系统独有的风格。而在使用 AWT 的 API 的时候根本就没有对不同系统的区分，你也根本就不需要去关心这一点。AWT 中正是使用桥梁模式来做到这一点的，而且桥梁模式的应用使得 AWT 的结构层次更加灵活。

不过我对 AWT 的代码不熟悉，所以也没有办法在这里讲解一下。下面只能举一个常见的教学代码了：（

以下代码来自《Thinking in Patterns with Java》：

```
//抽象部分（前端）的抽象角色
    class Abstraction {
        //维护着一个指向实现（Implementor）角色的引用
        private Implementation implementation;
        public Abstraction(Implementation imp) {
            implementation = imp;
        }
        // 下面定义了前端（抽象部分）应该有的接口
        public void service1() {
            //使用了后端（实现部分）已有的接口
            //组合实现功能
            implementation.facility1();
            implementation.facility2();
        }
        public void service2() {
            implementation.facility2();
            implementation.facility3();
        }
        public void service3() {
            implementation.facility1();
            implementation.facility2();
            implementation.facility4();
        }
        // For use by subclasses:
```

```
    protected Implementation getImplementation() {
        return implementation;
    }
}

//抽象部分（前端）的精确抽象角色
class ClientService1 extends Abstraction {
    public ClientService1(Implementation imp) { super(imp); }
    //使用抽象角色提供的方法组合起来完成某项功能
    //这就是为什么叫精确抽象角色（修正抽象角色）
    public void serviceA() {
        service1();
        service2();
    }
    public void serviceB() {
        service3();
    }
}

//另一个精确抽象角色，和上面一样的被我省略了
class ClientService2 extends Abstraction {
 。。。。
    //这里是直接通过实现部分的方法来实现一定的功能
    public void serviceE() {
        getImplementation().facility3();
    }
}

//实现部分（后端）的实现角色
interface Implementation {
  //这个接口只是定义了一定的接口
    void facility1();
    void facility2();
    void facility3();
    void facility4();
}

//具体实现角色就是要将实现角色提供的接口实现
//并完成一定的功能
//这里省略了
class Implementation1 implements Implementation {
  。。。
```

在上面的程序中还体现出一点特色：就是不仅实现部分和抽象部分所提供的接口可以完

全不一样；而且实现部分内部、抽象部分内部的接口也完全可以不一样。但是实现部分要提供类似的功能才行。

四、使用环境与优势

由上面我们分析得来的桥梁模式，可以看出来桥梁模式应该适用于以下环境：

1) 当你的系统中有多个地方要使用到类似的行为，或者是多个类似行为的组合时，可以考虑使用桥梁模式来提高重用，并减少因为行为的差异而产生的子类。
2) 系统中某个类的行为可能会有几种不同的变化趋势，为了有效的将变化封装，可以考虑将类的行为抽取出来。
3) 当然上面的情况也可以是这样，行为可能要被不同相似类使用，也可以考虑使用桥梁模式来实现。

桥梁模式使用了低耦合性的组合代替继承，使得它具备了不少好处：

1) 将可能变化的部分单独封装起来，使得变化产生的影响最小，不用编译不必要的代码。
2) 抽象部分和实现部分可以单独的变动，并且每一部分的扩充都不会破坏桥梁模式搭起来架子。
3) 对于客户程序来说，你的实现细节是透明的。

Bruce Eckel 在《Thinking in patterns with Java》中提到，可以把桥梁模式当作帮助你编码前端和后端独立变化的框架。

五、扩展

在《设计模式》一书中提到了使用抽象工厂模式来创建和配置一个桥梁模式。在上面的例子中也使用到了工厂方法模式来得到具体的实现部分。

组合模式

一、引子

在大学的数据结构这门课上，树是最重要的章节之一。还记得树是怎么定义的吗？树(Tree)是 n(n≥0)个结点的有限集 T，T 为空时称为空树，否则它满足如下两个条件：

1) 有且仅有一个特定的称为根(Root)的结点；
2) 其余的结点可分为 m(m≥0)个互不相交的子集 Tl，T2，…，Tm，其中每个子集本身又是一棵树，并称其为根的子树(SubTree)。

上面给出的递归定义刻画了树的固有特性：一棵非空树是由若干棵子树构成的，而子树又可由若干棵更小的子树构成。而这里的子树可以是叶子也可以是分支。

今天要学习的组合模式就是和树型结构以及递归有关系。

二、定义与结构

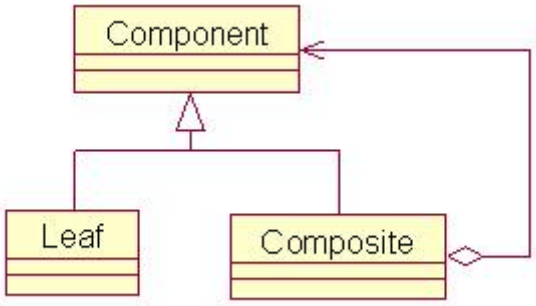
组合(Composite)模式的其它翻译名称也很多，比如合成模式、树模式等等。在《设计模式》一书中给出的定义是：将对象以树形结构组织起来，以达成"部分－整体"的层次结构，使得客户端对单个对象和组合对象的使用具有一致性。

从定义中可以得到使用组合模式的环境为：在设计中想表示对象的"部分－整体"层次结构；希望用户忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象。

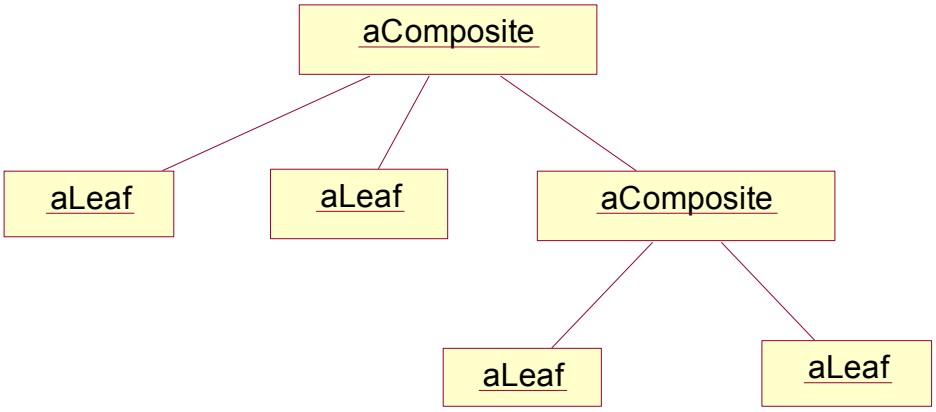
看下组合模式的组成。

1) 抽象构件角色(Component)：它为组合中的对象声明接口，也可以为共有接口实现缺省行为。
2) 树叶构件角色(Leaf)：在组合中表示叶节点对象——没有子节点，实现抽象构件角色声明的接口。
3) 树枝构件角色(Composite)：在组合中表示分支节点对象——有子节点，实现抽象构件角色声明的接口；存储子部件。

下图为组合模式的类图表示。



如图所示，不管你使用的是 Leaf 类还是 Composite 类，对于客户程序来说都是一样的——客户仅仅知道 Component 这个抽象类。而且在 Composite 类中还持有对 Component 抽象类的引用，这使得 Composite 中可以包含任何 Component 抽象类的子类。

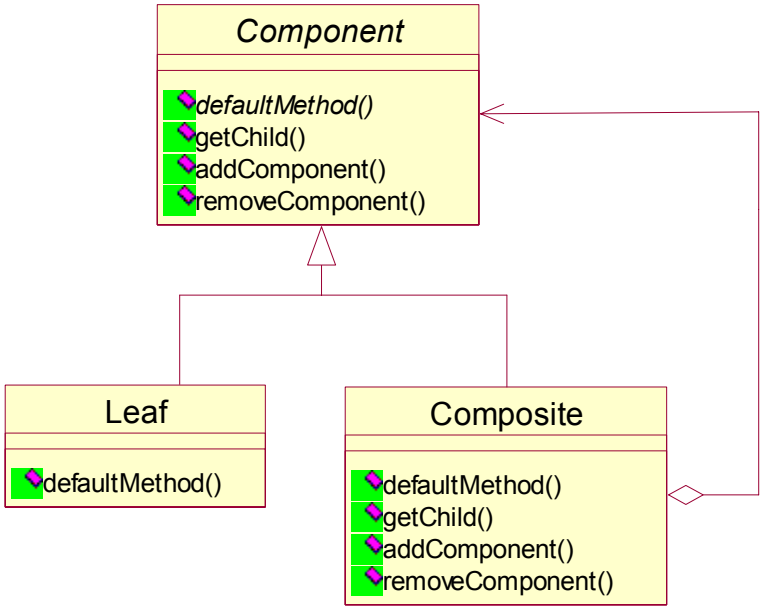


就向上图表示的那样，Composite 对象和 leaf 对象组成了树型结构，而且符合树的定义。

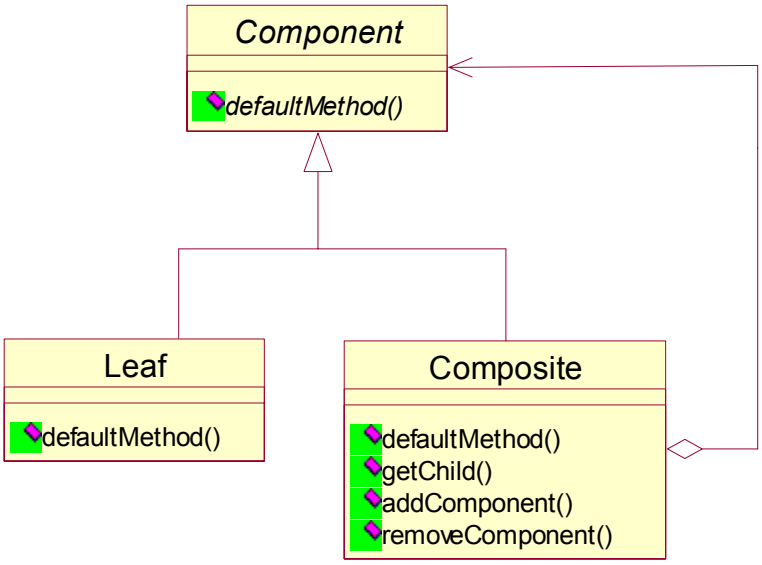
三、安全性与透明性

组合模式中必须提供对子对象的管理方法，不然无法完成对子对象的添加删除等等操作，也就失去了灵活性和扩展性。但是管理方法是在 Component 中就声明还是在 Composite 中声明呢？

一种方式是在 Component 里面声明所有的用来管理子类对象的方法，以达到 Component 接口的最大化（如下图所示）。目的就是为了使客户看来在接口层次上树叶和分支没有区别——透明性。但树叶是不存在子类的，因此 Component 声明的一些方法对于树叶来说是不适用的。这样也就带来了一些安全性问题。



另一种方式就是只在 Composite 里面声明所有的用来管理子类对象的方法（如下图所示）。这样就避免了上一种方式的安全性问题，但是由于叶子和分支有不同的接口，所以又失去了透明性。



《设计模式》一书认为：在这一模式中，相对于安全性，我们比较强调透明性。对于第一种方式中叶子节点内不需要的方法可以使用空处理或者异常报告的方式来解决。

四、举例

这里以 JUnit 中的组合模式的应用为例。

JUnit 是一个单元测试框架，按照此框架下的规范来编写测试代码，就可以使单元测试自动化。为了达到"自动化"的目的，JUnit 中定义了两个概念：TestCase 和 TestSuite。TestCase 是编写的测试类；而 TestSuite 是一个不同 TestCase 的集合，当然这个集合里面也可以包含 TestSuite 元素，这样运行一个 TestSuite 会将其包含的 TestCase 全部运行。

然而在真实运行测试程序的时候，是不需要关心这个类是 TestCase 还是 TestSuite，我们只关心测试运行结果如何。这就是为什么 JUnit 使用组合模式的原因。

JUnit 为了采用组合模式将 TestCase 和 TestSuite 统一起来，创建了一个 Test 接口来扮演抽象构件角色，这样原来的 TestCase 扮演组合模式中树叶构件角色，而 TestSuite 扮演组合模式中的树枝构件角色。下面将这三个类的有关代码分析如下：

```
//Test 接口——抽象构件角色
public interface Test {
    /**
     * Counts the number of test cases that will be run by this test.
     */
    public abstract int countTestCases();
    /**
     * Runs a test and collects its result in a TestResult instance.
     */
    public abstract void run(TestResult result);
}
```

```
//TestSuite 类的部分有关源码——Composite 角色，它实现了接口 Test
public class TestSuite implements Test {
    //用了较老的 Vector 来保存添加的 test
    private Vector fTests= new Vector(10);
    private String fName;
    ……
    /**
     * Adds a test to the suite.
     */
    public void addTest(Test test) {
        //注意这里的参数是 Test 类型的。这就意味着 TestCase 和 TestSuite 以及以后
    实现 Test 接口的任何类都可以被添加进来
        fTests.addElement(test);
    }
    ……
    /**
     * Counts the number of test cases that will be run by this test.
     */
    public int countTestCases() {
        int count= 0;
        for (Enumeration e= tests(); e.hasMoreElements(); ) {
            Test test= (Test)e.nextElement();
            count= count + test.countTestCases();
        }
        return count;
    }
    /**
     * Runs the tests and collects their result in a TestResult.
     */
    public void run(TestResult result) {
        for (Enumeration e= tests(); e.hasMoreElements(); ) {
            if (result.shouldStop() )
                break;
            Test test= (Test)e.nextElement();
            //关键在这个方法上面
            runTest(test, result);
        }
    }
      //这个方法里面就是递归的调用了，至于你的 Test 到底是什么类型的只有在运行
的时候得知
      public void runTest(Test test, TestResult result) {
          test.run(result);
      }
    ……
```

```
}

//TestCase 的部分有关源码——Leaf 角色，你编写的测试类就是继承自它
public abstract class TestCase extends Assert implements Test {
    ……
    /**
     * Counts the number of test cases executed by run(TestResult result).
     */
    public int countTestCases() {
        return 1;
    }
    /**
     * Runs the test case and collects the results in TestResult.
     */
    public void run(TestResult result) {
        result.run(this);
    }
    ……
}
```

可以看出这是一个偏重安全性的组合模式。因此在使用 TestCase 和 TestSuite 时，不能使用 Test 来代替。

五、优缺点

从上面的举例中可以看到，组合模式有以下优点：

1) 使客户端调用简单，客户端可以一致的使用组合结构或其中单个对象，用户就不必关心自己处理的是单个对象还是整个组合结构，这就简化了客户端代码。
2) 更容易在组合体内加入对象部件．客户端不必因为加入了新的对象部件而更改代码。这一点符合开闭原则的要求，对系统的二次开发和功能扩展很有利！

当然组合模式也少不了缺点：组合模式不容易限制组合中的构件。

六、总结

组合模式是一个应用非常广泛的设计模式，在前面已经介绍过的解释器模式、享元模式中都是用到了组合模式。它本身比较简单但是很有内涵，掌握了它对你的开发设计有很大的帮助。

装饰模式

一、引子

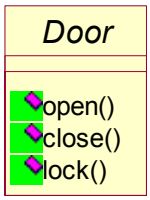
装饰模式？肯定让你想起又黑又火的家庭装修来。其实两者在道理上还是有很多相像的地方。家庭装修无非就是要掩盖住原来实而不华的墙面，抹上一层华而不实的涂料，让生活多一点色彩。而墙还是那堵墙，他的本质一点都没有变，只是多了一层外衣而已。

那设计模式中的装饰模式，是什么样子呢？

二、定义与结构

装饰模式（Decorator）也叫包装器模式（Wrapper）。GOF 在《设计模式》一书中给出的定义为：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。

让我们来理解一下这句话。我们来设计"门"这个类。假设你根据需求为"门"类作了如下定义：



现在，在系统的一个地方需要一个能够报警的 Door，你来怎么做呢？你或许写一个 Door 的子类 AlarmDoor，在里面添加一个子类独有的方法 alarm()。嗯，那在使用警报门的地方你必须让客户知道使用的是警报门，不然无法使用这个独有的方法。而且，这个还违反了 Liskov 替换原则。

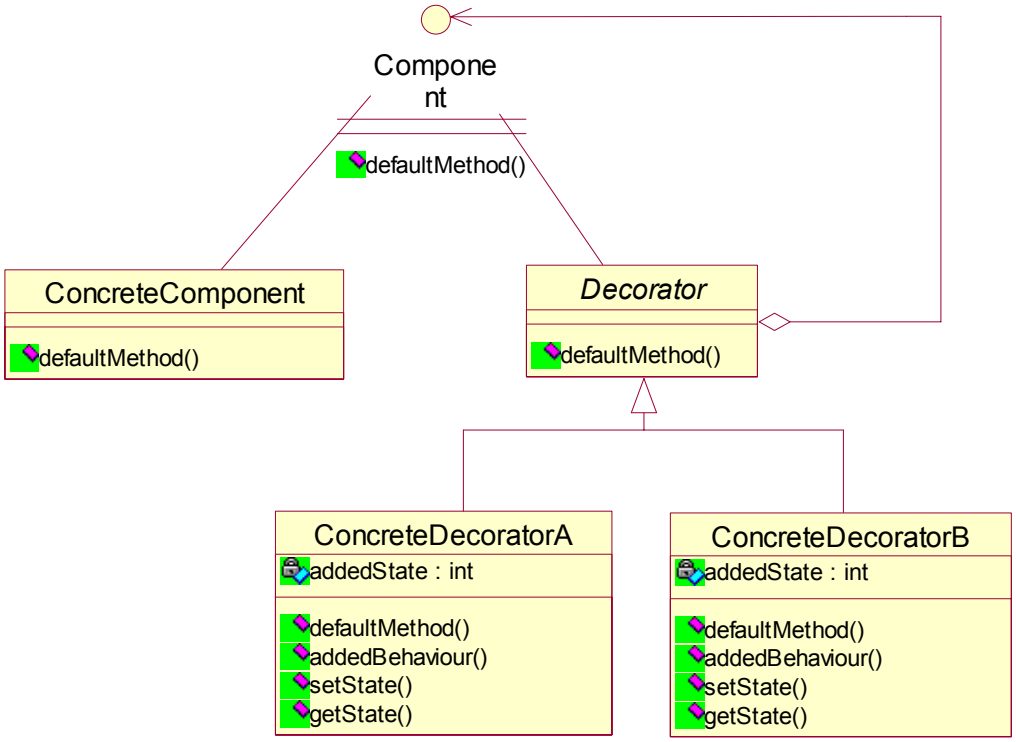
也许你要说，那就把这个方法添加到 Door 里面，这样不就统一了？但是这样所有的门都必须有警报，至少是个"哑巴"警报。而当你的系统仅仅在一两个地方使用了警报门，这明显是不合理的——虽然可以使用缺省适配器来弥补一下。

这时候，你可以考虑采用装饰模式来给门动态的添加些额外的功能。

下面我们来看看装饰模式的组成，不要急着去解决上面的问题，到了下面自然就明白了！

1) 抽象构件角色（Component）：定义一个抽象接口，以规范准备接收附加责任的对象。
2) 具体构件角色(Concrete Component)：这是被装饰者，定义一个将要被装饰增加功能的类。
3) 装饰角色(Decorator)：持有一个构件对象的实例，并定义了抽象构件定义的接口。
4) 具体装饰角色(Concrete Decorator)：负责给构件添加增加的功能。

看下装饰模式的类图：



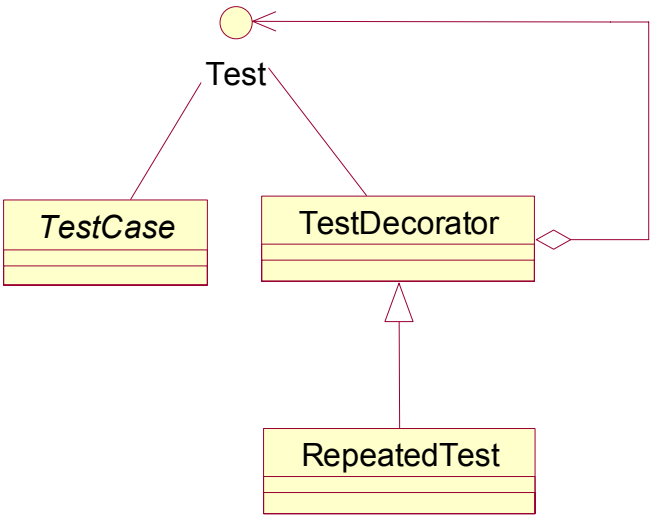
图中 ConcreteComponent 可能继承自其它的体系，而为了实现装饰模式，他还要实现 Component 接口。整个装饰模式的结构是按照组合模式来实现的——两者都有类似的结构图，都基于递归组合来组织可变数目的对象。但是两者的目的是截然不同的，组合（Composite）模式侧重通过递归组合构造类，使不同的对象、多重的对象可以“一视同仁”；而装饰（Decorator）模式仅仅是借递归组合来达到定义中的目的。

三、举例

这个例子还是来自我最近在研究的 JUnit，如果你对 JUnit 还不太了解，可以浏览一些关于 Junit 的文章。不愧是由 GoF 之一的 Erich Gamma 亲自开发的，小小的东西使用了 N 种的模式在里面。下面就来看看 JUnit 中的装饰模式。

在 JUnit 中，TestCase 是一个很重要的类，允许对其进行功能扩展。

在 junit.extensions 包中，TestDecorator、RepeatedTest 便是对 TestCase 的装饰模式扩展。下面我们将它们和上面的角色对号入座。



呵呵，看看源代码吧，这个来的最直接！

```
//这个就是抽象构件角色
public interface Test {
    /**
     * Counts the number of test cases that will be run by this test.
     */
    public abstract int countTestCases();
    /**
     * Runs a test and collects its result in a TestResult instance.
     */
    public abstract void run(TestResult result);
}

//具体构件对象，但是这里是个抽象类
public abstract class TestCase extends Assert implements Test {
    ……
    public int countTestCases() {
        return 1;
    }
    ……
    public TestResult run() {
        TestResult result= createResult();
        run(result);
        return result;
    }
    public void run(TestResult result) {
        result.run(this);
    }
    ……
}
```

```
//装饰角色
public class TestDecorator extends Assert implements Test {
    //这里按照上面的要求，保留了一个对构件对象的实例
    protected Test fTest;

    public TestDecorator(Test test) {
        fTest= test;
    }
    /**
     * The basic run behaviour.
     */
    public void basicRun(TestResult result) {
        fTest.run(result);
    }
    public int countTestCases() {
        return fTest.countTestCases();
    }
    public void run(TestResult result) {
        basicRun(result);
    }
    public String toString() {
        return fTest.toString();
    }
    public Test getTest() {
        return fTest;
    }
}

//具体装饰角色，这个类的增强作用就是可以设置测试类的执行次数
public class RepeatedTest extends   TestDecorator {
private int fTimesRepeat;

public RepeatedTest(Test test, int repeat) {
    super(test);
    if (repeat < 0)
        throw new IllegalArgumentException("Repetition count must be > 0");
    fTimesRepeat= repeat;
}
//看看怎么装饰的吧
public int countTestCases() {
    return super.countTestCases()*fTimesRepeat;
}
public void run(TestResult result) {
```

```
        for (int i= 0; i < fTimesRepeat; i++) {
            if (result.shouldStop())
                break;
            super.run(result);
        }
    }
    public String toString() {
        return super.toString()+"(repeated)";
    }
}
```

使用的时候，就可以采用下面的方式：

```
TestDecorator test = new RepeatedTest(new TestXXX() , 3);
```

让我们在回想下上面提到的“门”的问题，这个警报门采用了装饰模式后，可以采用下面的方式来产生。

```
DoorDecorator alarmDoor = new AlarmDoor(new Door());
```

四、应用环境

GOF 书中给出了以下使用情况：

1) 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
2) 处理那些可以撤消的职责。
3) 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

来分析下 JUnit 的使用是属于哪种情况。首先实现了比静态继承更加灵活的方式，动态的增加功能。试想为 Test 的所有实现类通过继承来增加一个功能，意味着要添加不少的功能类似的子类，这明显是不太合适的。

而且，这就避免了高层的类具有太多的特征，比如上面提到的带有警报的抽象门类。

五、透明和半透明

对于面向接口编程，应该尽量使客户程序不知道具体的类型，而应该对一个接口操作。这样就要求装饰角色和具体装饰角色要满足 Liskov 替换原则。像下面这样：

```
Component c = new ConcreteComponent();
Component c1 = new ConcreteDecorator(c);
```

JUnit 中就属于这种应用，这种方式被称为透明式。而在实际应用中，比如 java.io 中往往因为要对原有接口做太多的扩展而需要公开新的方法（这也是为了重用）。所以往往不能对客户程序隐瞒具体的类型。这种方式称为“半透明式”。

在 java.io 中，并不是纯装饰模式的范例，它是装饰模式、适配器模式的混合使用。

六、其它

采用 Decorator 模式进行系统设计往往会产生许多看上去类似的小对象，这些对象仅仅在他们相互连接的方式上有所不同，而不是它们的类或是它们的属性值有所不同。尽管对于那些了解这些系统的人来说，很容易对它们进行定制，但是很难学习这些系统，排错也很困难。这是 GOF 提到的装饰模式的缺点，你能体会吗？他们所说的小对象我认为是指的具体装饰角色。这是为一个对象动态添加功能所带来的副作用。

门面模式

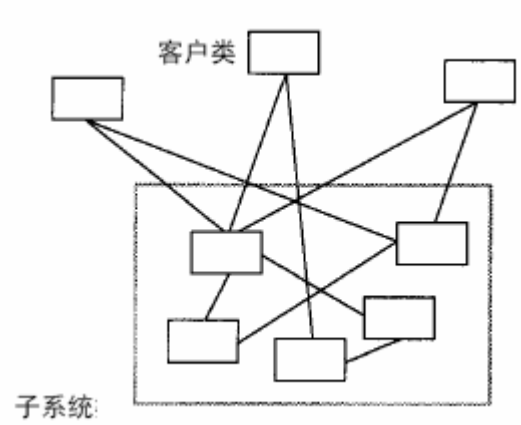
一、引子

门面模式是非常简单的设计模式。

二、定义与结构

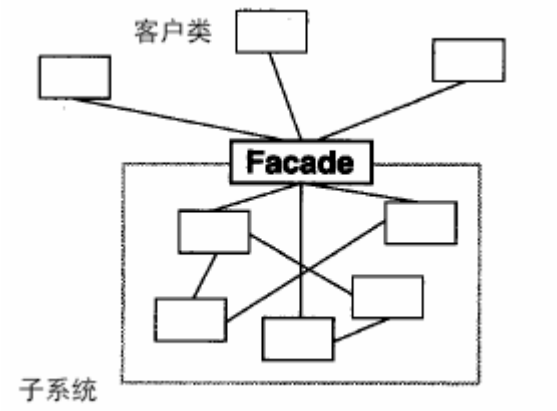
门面模式（facade）又称外观模式。GOF在《设计模式》一书中给出如下定义：为子系统中的一组接口提供一个一致的界面， Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

定义中提到的子系统是指在设计中为了降低复杂性根据一定的规则（比如业务、功能），对系统进行的划分。子系统中封装有一些类。客户程序在使用子系统的时候，可能会像下图一样零乱。



在上面的实现方法中，客户类紧紧地依赖在子系统的实现上。子系统发生的变化，很可能要影响到客户类的调用。而且子系统在不断优化、可重用化的重构路上，会产生更多更小的类。这对使用子系统的客户类来说要完成一个工作流程，似乎要记住的接口太多了。

门面模式就是为了解决这种问题而产生的。看看使用了门面模式后的图：



这样就减少了客户程序和子系统之间的耦合，增加了可维护性。

很明显，门面模式有三个角色组成：

1) 门面角色（facade）：这是门面模式的核心。它被客户角色调用，因此它熟悉子系统的功能。它内部根据客户角色已有的需求预定了几种功能组合。
2) 子系统角色：实现了子系统的功能。对它而言，facade角色就和客户角色一样是未知的，它没有任何facade角色的信息和链接。

3) 客户角色：调用facade角色来完成要得到的功能。

三、举例

Facade 模式的一个典型应用就是进行数据库连接。一般我们在每一次对数据库进行访问，都要进行以下操作：先得到 connect 实例，然后打开 connect 获得连接，得到一个 statement，执行 sql 语句进行查询，得到查询结果集。

我们可以将这些步骤提取出来，封装在一个类里面。这样，每次执行数据库访问只需要将必要的参数传递到这个类中就可以了。

有兴趣可以在你正在进行的系统中实现一下。

四、使用环境和优点

《设计模式》给出了门面模式的使用环境：

1) 当你要为一个复杂子系统提供一个简单接口时。在上面已经描述了原因。
2) 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入 facade 将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性（上面也提到了）。
3) 当你需要构建一个层次结构的子系统时，使用 facade 模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过 facade 进行通讯，从而简化了它们之间的依赖关系。

以下是它的优点：

1) 它对客户屏蔽子系统组件，因而减少了客户处理的对象的数目并使得子系统使用起来更加方便。
2) 它实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件往往是紧耦合的。松耦合关系使得子系统的组件变化不会影响到它的客户。Facade 模式有助于建立层次结构系统，也有助于对对象之间的依赖关系分层。Facade 模式可以消除复杂的循环依赖关系。这一点在客户程序与子系统是分别实现的时候尤为重要。在大型软件系统中降低编译依赖性至关重要。在子系统类改变时，希望尽量减少重编译工作以节省时间。用 Facade 可以降低编译依赖性，限制重要系统中较小的变化所需的重编译工作。Facade 模式同样也有利于简化系统在不同平台之间的移植过程，因为编译一个子系统一般不需要编译所有其他的子系统。
3) 如果应用需要，它并不限制它们使用子系统类。因此你可以让客户程序在系统易用性和通用性之间加以选择。

五、总结

从整体上来看门面模式给我的感觉是，它对于使两层之间的调用粗颗粒化很有帮助，避免了大量细颗粒度的访问。这和 SOA 中的一些观点是相同的。

享元模式

一、引子

让我们先来复习下 java 中 String 类型的特性：String 类型的对象一旦被创造就不可改变；当两个 String 对象所包含的内容相同的时候，JVM 只创建一个 String 对象对应这两个不同的对象引用。让我们来证实下着两个特性吧（如果你已经了解，请跳过直接阅读第二部分）。

先来验证下第二个特性：

```
public class TestPattern {
    public static void main(String[] args){
        String n = "I Love Java";
        String m = "I Love Java";
        System.out.println(n==m);
    }
}
```

这段代码会告诉你 n==m 是 true，这就说明了在 JVM 中 n 和 m 两个引用了同一个 String 对象。

那么接着验证下第一个特性：

在系统输出之前加入一行代码"m = m + "hehe";"，这时候 n==m 结果为 false，为什么刚才两个还是引用相同的对象，现在就不是了呢？原因就是在执行后添加语句时，m 指向了一个新创建的 String 对象，而不是修改引用的对象。

呵呵，说着说着就差点跑了题，并不是每个 String 的特性都跟我们今天的主题有关的。

String 类型的设计避免了在创建 N 多的 String 对象时产生的不必要的资源损耗，可以说是享元模式应用的范例，那么让我们带着对享元的一点模糊的认识开始，来看看怎么在自己的程序中正确的使用享元模式！

注：使用 String 类型请遵循《Effective Java》中的建议。

二、定义与分类

享元模式英文称为"Flyweight Pattern"，又译为羽量级模式或者蝇量级模式。我非常认同将 Flyweight Pattern 翻译为享元模式，因为这个词将这个模式使用的方式明白得表示了出来。

享元模式的定义为：采用一个共享类来避免大量拥有相同内容的“小类”的开销。这种开销中最常见、直观的影响就是增加了内存的损耗。享元模式以共享的方式高效的支持大量的细粒度对象，减少其带来的开销。

在名字和定义中都体现出了共享这一个核心概念，那么怎么来实现共享呢？事物之间都是不同的，但是又存在一定的共性，如果只有完全相同的事物才能共享，那么享元模式可以说就是不可行的；因此我们应该尽量将事物的共性共享，而又保留它的个性。为了做到这点，享元模式中区分了内蕴状态和外蕴状态。内蕴状态就是共性，外蕴状态就是个性了。

内蕴状态存储在享元内部，不会随环境的改变而有所不同，是可以共享的；外蕴状态是不可以共享的，它随环境的改变而改变的，因此外蕴状态是由客户端来保持（因为环境的变化是由客户端引起的）。在每个具体的环境下，客户端将外蕴状态传递给享元，从而创建不同的对象出来。

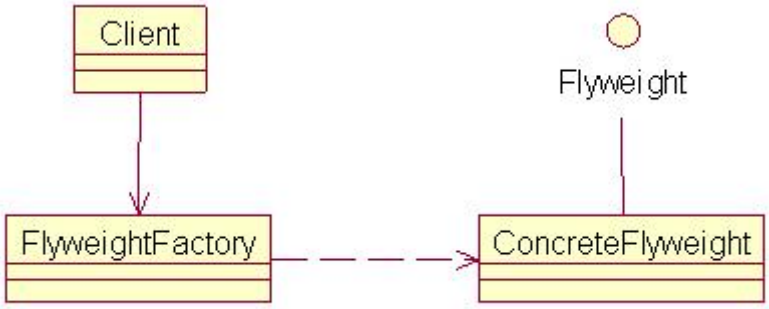
我们引用《Java 与模式》中的分类，将享元模式分为：单纯享元模式和复合享元模式。在下一个小节里面我们将详细的讲解这两种享元模式。

三、结构

先从简单的入手，看看单纯享元模式的结构。

1) 抽象享元角色：为具体享元角色规定了必须实现的方法，而外蕴状态就是以参数的形式通过此方法传入。在 Java 中可以由抽象类、接口来担当。
2) 具体享元角色：实现抽象角色规定的方法。如果存在内蕴状态，就负责为内蕴状态提供存储空间。
3) 享元工厂角色：负责创建和管理享元角色。要想达到共享的目的，这个角色的实现是关键！
4) 客户端角色：维护对所有享元对象的引用，而且还需要存储对应的外蕴状态。

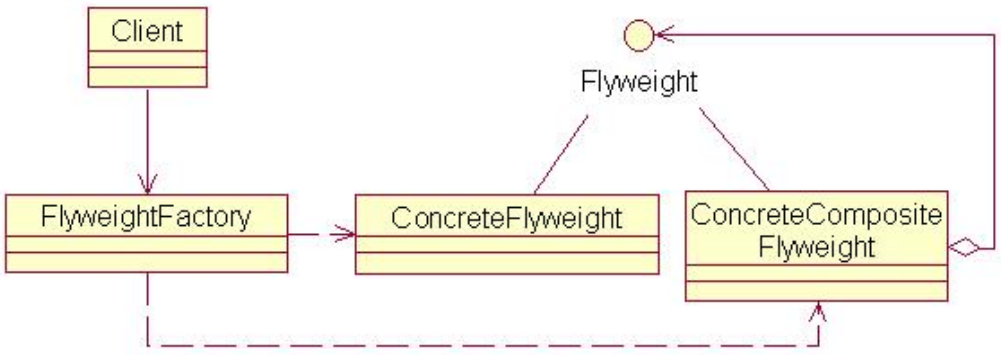
来用类图来形象地表示出它们的关系吧。



再来看看复合享元模式的结构。

1) 抽象享元角色：为具体享元角色规定了必须实现的方法，而外蕴状态就是以参数的形式通过此方法传入。在 Java 中可以由抽象类、接口来担当。
2) 具体享元角色：实现抽象角色规定的方法。如果存在内蕴状态，就负责为内蕴状态提供存储空间。
3) 复合享元角色：它所代表的对象是不可以共享的，并且可以分解成为多个单纯享元对象的组合。
4) 享元工厂角色：负责创建和管理享元角色。要想达到共享的目的，这个角色的实现是关键！
5) 客户端角色：维护对所有享元对象的引用，而且还需要存储对应的外蕴状态。

统比一下单纯享元对象和复合享元对象，里面只多出了一个复合享元角色，但是它的结构就发生了很大的变化。我们还是使用类图来表示下：



正如你所想，复合享元模式采用了组合模式——为了将具体享元角色和复合享元角色同等对待和处理。这也就决定了复合享元角色中所包含的每个单纯享元都具有相同的外蕴状态，而这些单纯享元的内蕴状态可以是不同的。

四、举例

很遗憾，没有看到享元模式实用的例子。享元模式如何来共享内蕴状态的？在能见到的教学代码中，大概有两种实现方式：实用列表记录（或者缓存）已存在的对象和使用静态属性。下面的例子来自于 Bruce Eckel 的《Thinking in Patterns with java》一书。

设想一下有一个含有多个属性的对象，要被创建一百万次，并使用它们。这时候正是使用享元模式的好时机：

```
//这便是使用了静态属性来达到共享
//它使用了数组来存放不同客户对象要求的属性值
//它相当于享元角色（抽象角色被省略了）
class ExternalizedData {
    static final int size = 5000000;
    static int[] id = new int[size];
    static int[] i = new int[size];
    static float[] f = new float[size];
    static {
        for(int i = 0; i < size; i++)
        id[i] = i;
    }
}

//这个类仅仅是为了给 ExternalizedData 的静态属性赋值、取值
//这个充当享元工厂角色
class FlyPoint {
    private FlyPoint() {}
    public static int getI(int obnum) {
        return ExternalizedData.i[obnum];
    }
    public static void setI(int obnum, int i) {
        ExternalizedData.i[obnum] = i;
    }
    public static float getF(int obnum) {
        return ExternalizedData.f[obnum];
    }
    public static void setF(int obnum, float f) {
        ExternalizedData.f[obnum] = f;
    }
    public static String str(int obnum) {
        return "id: " +
```

```
            ExternalizedData.id[obnum] +
            ", i = " +
            ExternalizedData.i[obnum] +
            ", f = " +
            ExternalizedData.f[obnum];
        }
    }

    //客户程序
    public class FlyWeightObjects {
        public static void main(String[] args) {
            for(int i = 0; i < ExternalizedData.size; i++) {
                FlyPoint.setI(i, FlyPoint.getI(i) + 1);
                FlyPoint.setF(i, 47.0f);
            }
        System.out.println(
            FlyPoint.str(ExternalizedData.size -1));
        }
    } ///:~
```

另外一种实现方式大概是将已存在内蕴状态不同的对象储存在一个列表当中，通过享元工厂角色来控制重复对象的生成。而对于上面提到的复合享元模式，仅仅是在抽象享元角色下面添加一个有组合模式来构造的复合享元角色。而且复合享元中所包含的每个单纯享元都具有相同的外蕴状态，而这些单纯享元的内蕴状态往往是不同的。由于复合享元模式不能共享，所以不存在什么内外状态对应的问题。所以在复合享元类中我们不用实现抽象享元对象中的方法，因此这里采用的是透明式的合成模式。

复合享元角色仿佛没有履行享元模式存在的义务。复合享元角色是由多个具体享元角色来组成的，虽然复合享元角色不能被共享使用，但是组成它的具体享元角色还是使用了共享的方式。因此复合享元模式并没有违背享元模式的初衷。

五、使用优缺点

享元模式优点就在于它能够大幅度的降低内存中对象的数量；而为了做到这一步也带来了它的缺点：它使得系统逻辑复杂化，而且在一定程度上外蕴状态影响了系统的速度。

所以一定要切记使用享元模式的条件：

1)系统中有大量的对象，他们使系统的效率降低。

2)这些对象的状态可以分离出所需要的内外两部分。

外蕴状态和内蕴状态的划分以及两者关系的对应也是非常值得重视的。只有将内外划分妥当才能使内蕴状态发挥它应有的作用；如果划分失误，在最糟糕的情况下系统中的对象是一个也不会减少的！两者的对应关系的维护和查找也是要花费一定的空间（当然这个比起不使用共享对象要小得多）和时间的，可以说享元模式就是使用时间来换取空间的。可以采用相应的算法来提高查找的速度。

代理模式

一、引子

我们去科技市场为自己的机器添加点奢侈的配件，很多 DIYer 都喜欢去找代理商，因为在代理商那里拿到的东西不仅质量有保证，而且价格和售后服务上都会好很多。客户通过代理商得到了自己想要的东西，而且还享受到了代理商额外的服务；而生产厂商通过代理商将自己的产品推广出去，而且可以将一些销售服务的任务交给代理商来完成（当然代理商要和厂商来共同分担风险，分配利润），这样自己就可以花更多的心思在产品的设计和生产上了。

在美国，任何企业的产品要想拿到市场上去卖就必须经过代理商这一个环节，否则就是非法的。看来代理商在商业运作中起着很关键的作用。

不小心把话题扯远了，回过头来，在我们的面向对象的程序设计中，也存在着代理商这样的角色。下面就跟着这篇文章来看看代理模式的奇妙吧~~~~

二、简介

代理模式有两个英文名字：Proxy Pattern 和 Surrogate Pattern。代理模式的定义为：为其他对象提供一种代理以控制对这个对象的访问。说白了就是，在一些情况下客户不想或者不能直接引用一个对象，而代理对象可以在客户和目标对象之间起到中介作用，去掉客户不能看到的内容和服务或者增添客户需要的额外服务。

根据《Java 与模式》书中对代理模式的分类，代理模式分为 8 种，这里将几种常见的、重要的列举如下：

1) 远程（Remote）代理：为一个位于不同的地址空间的对象提供一个局域代表对象。比如：你可以将一个在世界某个角落一台机器通过代理假象成你局域网中的一部分。
2) 虚拟（Virtual）代理：根据需要将一个资源消耗很大或者比较复杂的对象延迟的真正需要时才创建。比如：如果一个很大的图片，需要花费很长时间才能显示出来，那么当这个图片包含在文档中时，使用编辑器或浏览器打开这个文档，这个大图片可能就影响了文档的阅读，这时需要做个图片 Proxy 来代替真正的图片。
3) 保护（Protect or Access）代理：控制对一个对象的访问权限。比如：在论坛中，不同的身份登陆，拥有的权限是不同的，使用代理模式可以控制权限（当然，使用别的方式也可以实现）。
4) 智能引用（Smart Reference）代理：提供比对目标对象额外的服务。比如：纪录访问的流量（这是个再简单不过的例子），提供一些友情提示等等。

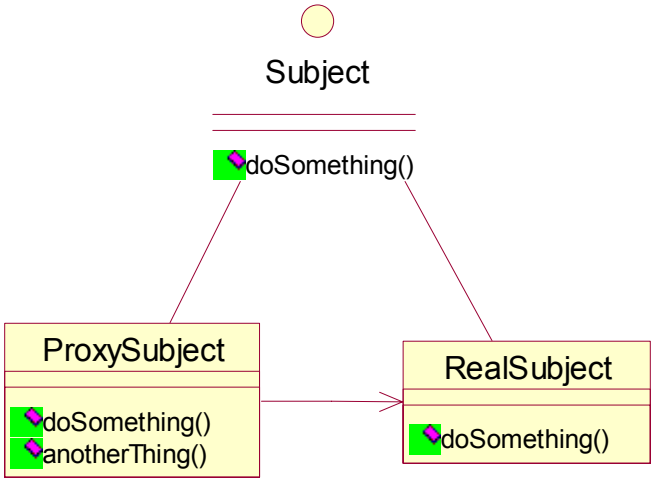
代理模式是一种比较有用的模式，从几个类的"小结构"到庞大系统的"大结构"都可以看到它的影子。

三、结构

代理模式中的"代理商"要想实现代理任务，就必须和被代理的"厂商"使用共同的接口（你可以想象为产品）。于是代理模式就有三个角色组成了：

1) 抽象主题角色：声明了真实主题和代理主题的共同接口。
2) 代理主题角色：内部包含对真实主题的引用，并且提供和真实主题角色相同的接口。
3) 真实主题角色：定义真实的对象。

使用类图来表示下三者间的关系如下：



当然，图上所示的是代理模式中的一个具体情况。而代理模式可以非常灵活的使用其他方式来实现，这样就与图上所示有很大的区别。

也许，现在你已经对代理模式已经有了一个宏观的认识了，下面我们来看看怎么实际的使用代理模式。

四、举例

以论坛中已注册用户和游客的权限不同来作为第一个例子：已注册的用户拥有发帖，修改自己的注册信息，修改自己的帖子等功能；而游客只能看到别人发的帖子，没有其他权限。为了简化代码，更好的显示出代理模式的骨架，我们这里只实现发帖权限的控制。

首先我们先实现一个抽象主题角色 MyForum，里面定义了真实主题和代理主题的共同接口——发帖功能。

代码如下：

```
public interface MyForum
{
    public void AddFile();
}
```

这样，真实主题角色和代理主题角色都要实现这个接口。其中真实的主题角色基本就是将这个接口的方法内容填充进来。所以在这里就不再赘述它的实现。我们把主要的精力放到关键的代理主题角色上。代理主题角色代码大体如下：

```
public class MyForumProxy implements MyForum
{
    private RealMyForum forum = new RealMyForum() ;
    private int permission ;          //权限值
    public MyForumProxy(int permission)
    {
        this.permission = permission ;
    }
    //实现的接口
    public void AddFile()
    {
        //满足权限设置的时候才能够执行操作
```

```
            //Constants 是一个常量类
            if(Constants.ASSOCIATOR == permission)
            {
                forum.AddFile();
            }
            else
                System.out.println("You are not a associator of MyForum ,please registe!");
        }
    }
```

这样就实现了代理模式的功能。

五、总结

代理模式能够协调调用者和被调用者，能够在一定程度上降低系统的耦合度。代理模式中的真实主题角色可以结合组合模式来构造，这样一个代理主题角色就可以对一系列的真实主题角色有效，提高代码利用率，减少不必要子类的产生。

责任链模式

一、 引言

初看责任链模式，心里不禁想起了一个以前听过的相声：看牙。说一个病人看牙的时候，医生不小心把拔下的一个牙掉进了病人嗓子里。各个科室的医生推卸责任，搞得病人因此楼上楼下的跑了不少冤枉路，最后无果而终。

责任链模式就是这种"推卸"责任的模式，你的问题在我这里能解决我就解决，不行就把你推给另一个对象。至于到底谁解决了这个问题了呢？我管呢！

二、 定义与结构

从名字上大概也能猜出这个模式的大概模样——系统中将会存在多个有类似处理能力的对象。当一个请求触发后，请求将在这些对象组成的链条中传递，直到找到最合适的"责任"对象，并进行处理。

《设计模式》中给它的定义如下：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

呵呵，从定义上可以看出，责任链模式的提出是为了"解耦"，以应变系统需求的变更和不明确性。

下面是《设计模式》中给出的适用范围：

1) 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
2) 你想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) 可处理一个请求的对象集合应被动态指定。

责任链模式真的能给发送者和接收者之间解耦（这好像很神奇）吗？先来看下它的组成角色。这个问题我会在下面提及。

责任链模式由两个角色组成：

1) 抽象处理者角色（Handler）：它定义了一个处理请求的接口。当然对于链子的不同实现，也可以在这个角色中实现后继链。
2) 具体处理者角色（Concrete Handler）：实现抽象角色中定义的接口，并处理它所负责的请求。如果不能处理则访问它的后继者。

至于类图不放也罢。毕竟就是一个继承或者实现。

三、 纯与不纯

责任链模式的纯与不纯的区别，就像黑猫、白猫的区别一样。不要刻意的去使自己的代码来符合一个模式的公式。只要能够使代码降低耦合、提高重用，满足系统需求并能很好的适应变化就好了。正所谓：管它黑猫白猫，抓住老鼠就是好猫！

纯的责任链模式，规定一个具体处理者角色只能对请求作出两种动作：自己处理；传给下家。不能出现处理了一部分，把剩下的传给了下家的情况。而且请求在责任链中必须被处理，而不能出现无果而终的结局。

反之，则就是不纯的责任链模式。

不纯的责任链模式还算是责任链模式吗？比如一个请求被捕获后，每个具体处理者都尝试去处理它，不管结果如何都将请求再次转发。我认为这种方式的实现，算不算是责任链模式的一种倒不重要，重要的是我们也能从中体味到责任链模式的思想：通过将多个处理者之

间建立联系，来达到请求与具体的某个处理者的解耦。

下面的例子就是采用了上面提到的"不纯的责任链模式"。

四、 举例

这个例子来源于项目中我刚刚完成的一个小功能点——"代号自动生成器"。在项目中存在很多地方，比如：员工工号、档案代号，要求客户在使用时输入。而这些代号对于一个特定的企业或者类别，往往有一定的规则。因此可以让用户在系统参数中维护一定的规则，然后通过"代号自动生成器"来给用户生成代号。

根据初期需求，用户代号中往往存在以下几种变动元素：年份、月份、日期、流水号。由于需求比较简单，因此考虑到用户可能存在其他变动元素，所以我打算在"被第一颗子弹击中"后重构一下现有的结构。下面就是我在头脑中演绎过的使用责任链模式的重构。

这里只用来说明下责任链模式的结构和使用，因此不体现功能细节。

```
//这是抽象处理者角色
public interface CodeAutoParse {

  //这里就是统一的处理请求使用的接口
   String[] generateCode(String moduleCode, int number, String rule,String[] target)
         throws BaseException;
}

//这个为处理日期使用的具体处理者
public class DateAutoParse implements CodeAutoParse{
   //获取当前时间
   private final Calendar currentDate = Calendar.getInstance();

    //这里用来注入下一个处理者，系统中采用 Spring Bean 管理
    private CodeAutoParse theNextParseOfDate;

    public void setTheNextParseOfDate(CodeAutoParse theNextParseOfDate){
    this.theNextParseOfDate = theNextParseOfDate ;
   }

   /*
    *实现的处理请求的接口
   *这个接口首先判断用户定义的格式是否有流水号,有则解析,没有则跳过
    *下传到下一个处理者
  */
   public String[] generateCode(String moduleCode, int number, String rule, String[]
target)
         throws BaseException {
         //这里省略了处理的业务
         ……
         if(theNextParseOfDate != null)
```

```
                return theNextParseOfDate.generateCode(moduleCode , number , rule,
target)
            else
                return target;
    }
```

其它具体处理者也是如此的结构，每一个里面都设置有一个用来存放下一个处理者的引用，不管你有没有下一个处理者。

其实责任链模式本身的结构和使用都没有什么，就是一个继承或者实现。在处理请求的时候，按照规定去调用下一个处理者。但是怎么来维护这样一条链子呢？

《设计模式》一书中仅仅说必须自己引入它，可以参考使用 list 或者 map 来进行注册。而在上面我使用 spring 来管理具体处理者角色的引入。当有了新的处理者需要添加的时候，仅仅需要修改下配置文件。

五、 其他

责任链模式优点，上面已经体现出来了。无非就是降低了耦合、提高了灵活性。但是责任链模式可能会带来一些额外的性能损耗，因为它每次执行请求都要从链子开头开始遍历。

命令模式

一、引言

忙里偷闲，终于动笔了。命令模式是从界面设计中提取出来的一种分离耦合，提高重用的方法。被认为是最优雅而且简单的模式，它的应用范围非常广泛。让我们一起来认识下它吧。

先从起源说起。在设计界面时，大家可以注意到这样的一种情况，同样的菜单控件，在不同的应用环境中的功能是完全不同的；而菜单选项的某个功能可能和鼠标右键的某个功能完全一致。按照最差、最原始的设计，这些不同功能的菜单、或者右键弹出菜单是要分开来实现的，你可以想象一下，word 文档上面的一排菜单要实现出多少个"形似神非"的菜单类来？这完全是行不通的。这时，就要运用分离变化与不变的因素，将菜单触发的功能分离出来，而制作菜单的时候只是提供一个统一的触发接口。这样修改设计后，功能点可以被不同的菜单或者右键重用；而且菜单控件也可以去除变化因素，很大的提高了重用；而且分离了显示逻辑和业务逻辑的耦合。这便是命令模式的雏形。

下面我们将仔细的讨论下命令模式。

二、定义与结构

《设计模式》中命令模式的定义为：将一个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，以及支持可撤消的操作。

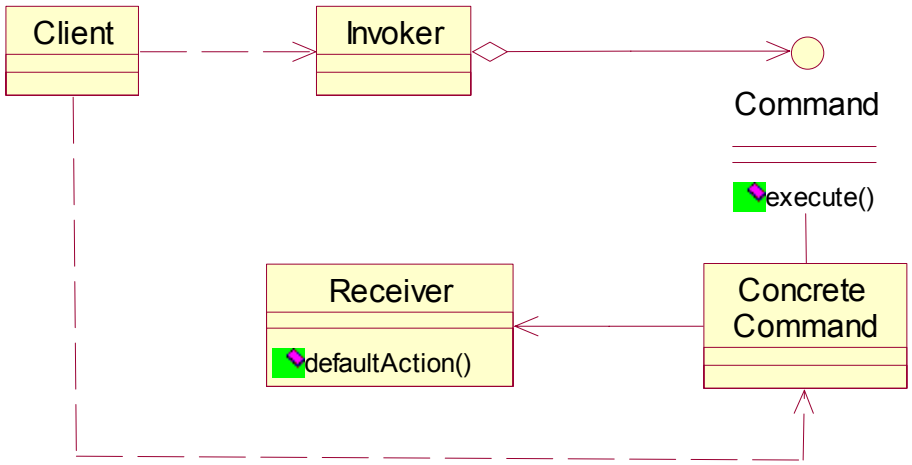
看起来，命令模式好像神通广大。其实命令模式的以上功能还要看你是怎么写的——程序总是程序员写出来的，你写啥它才能干啥：）

在我看来，其实命令模式像很多设计模式一样——通过在你的请求和处理之间加上了一个中间人的角色，来达到分离耦合的目的。通过对中间人角色的特殊设计来形成不同的模式。当然命令模式就是一种特殊设计的结果。

看下命令模式是有哪些角色来组成的吧。

1) 命令角色（Command）：声明执行操作的接口。有 java 接口或者抽象类来实现。
2) 具体命令角色（Concrete Command）：将一个接收者对象绑定于一个动作；调用接收者相应的操作，以实现命令角色声明的执行操作的接口。
3) 客户角色（Client）：创建一个具体命令对象（并可以设定它的接收者）。
4) 请求者角色（Invoker）：调用命令对象执行这个请求。
5) 接收者角色（Receiver）：知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。

以下是命令模式的类图，从中可以大致的了解到各个角色之间是怎么来协调工作的。



三、举例

本来想接着我的 JUnit 分析来讲解命令模式。但是由于在 JUnit 中，参杂了其它的模式在里面，使得命令模式的特点不太明显。所以这里将以命令模式在 Web 开发中最常见的应用——Struts 中 Action 的使用作为例子。

在 Struts 中 Action 控制类是整个框架的核心，它连接着页面请求和后台业务逻辑处理。按照框架设计，每一个继承自 Action 的子类，都实现 execute 方法——调用后台真正处理业务的对象来完成任务。

注：继承自 DispatchAction 的子类，则可以一个类里面处理多个类似的操作。这个在这不做讨论。

下面我们将 Struts 中的各个类与命令模式中的角色对号入座。

先来看下命令角色——Action 控制类

```
public class Action {
    ……
      /*
      *可以看出，Action 中提供了两个版本的执行接口，而且实现了默认的空实现。
      */
    public ActionForward execute( ActionMapping mapping,
                                  ActionForm form,
                                  ServletRequest request,
                                  ServletResponse response)
        throws Exception {
        try {
            return execute(mapping, form, (HttpServletRequest) request,
                           (HttpServletResponse) response);
        } catch (ClassCastException e) {
            return null;
        }
    }

    public ActionForward execute( ActionMapping mapping,
                                  ActionForm form,
```

```
                                    HttpServletRequest request,
                                    HttpServletResponse response)
        throws Exception {
        return null;
    }
}
```

下面的就是请求者角色，它仅仅负责调用命令角色执行操作。

```
public class RequestProcessor {
    ……
    protected ActionForward processActionPerform(HttpServletRequest request,
                                                 HttpServletResponse response,
                                                 Action action,
                                                 ActionForm form,
                                                 ActionMapping mapping)
        throws IOException, ServletException {
        try {
            return (action.execute(mapping, form, request, response));
        } catch (Exception e) {
            return (processException(request, response,e, form, mapping));
        }
    }
}
```

Struts 框架为我们提供了以上两个角色，要使用 struts 框架完成自己的业务逻辑，剩下的三个角色就要由我们自己来实现了。步骤如下：

1) 很明显我们要先实现一个 Action 的子类，并重写 execute 方法。在此方法中调用业务模块的相应对象来完成任务。
2) 实现处理业务的业务类，来充当接收者角色。
3) 配置 struts-config.xml 配置文件，将自己的 Action 和 Form 以及相应页面结合起来。
4) 编写 jsp，在页面中显式的制定对应的处理 Action。

一个完整的命令模式就介绍完了。当你在页面上提交请求后，Struts 框架会根据配置文件中的定义，将你的 Action 对象作为参数传递给 RequestProcessor 类中的 processActionPerform()方法，由此方法调用 Action 对象中的执行方法，进而调用业务层中的接收角色。这样就完成了请求的处理。

四、Undo、事务及延伸

在定义中提到，命令模式支持可撤销的操作。而在上面的举例中并没有体现出来。其实命令模式之所以能够支持这种操作，完全得益于在请求者与接收者之间添加了中间角色。为了实现 undo 功能，首先需要一个历史列表来保存已经执行过的具体命令角色对象；修改具体命令角色中的执行方法，使它记录更多的执行细节，并将自己放入历史列表中；并在具体命令角色中添加 undo 方法，此方法根据记录的执行细节来复原状态（很明显，首先程序员要清楚怎么来实现，因为它和 execute 的效果是一样的）。

同样，redo 功能也能够照此实现。

命令模式还有一个常见的用法就是执行事务操作。这就是为什么命令模式还叫做事务模式的原因吧。它可以在请求被传递到接收者角色之前，检验请求的正确性，甚至可以检查和数据库中数据的一致性，而且可以结合组合模式的结构，来一次执行多个命令。

使用命令模式不仅仅可以解除请求者和接收者之间的耦合，而且可以用来做批处理操作，这完全可以发挥你自己的想象——请求者发出的请求到达命令角色这里以后，先保存在一个列表中而不执行；等到一定的业务需要时，命令模式再将列表中全部的操作逐一执行。

哦，命令模式实在太灵活了。真是一个很有用的东西啊！

五、优点及适用情况

由上面的讲解可以看出命令模式有以下优点：

1) 命令模式将调用操作的请求对象与知道如何实现该操作的接收对象解耦。
2) 具体命令角色可以被不同的请求者角色重用。
3) 你可将多个命令装配成一个复合命令。
4) 增加新的具体命令角色很容易，因为这无需改变已有的类。

GOF 总结了命令模式的以下适用环境。

1) 需要抽象出待执行的动作，然后以参数的形式提供出来——类似于过程设计中的回调机制。而命令模式正是回调机制的一个面向对象的替代品。
2) 在不同的时刻指定、排列和执行请求。一个命令对象可以有与初始请求无关的生存期。
3) 需要支持取消操作。
4) 支持修改日志功能。这样当系统崩溃时，这些修改可以被重做一遍。
5) 需要支持事务操作。

六、总结

从面向对象的角度来看，命令模式是不完美的。命令角色仅仅包含一个方法，没有任何属性存在。这是将函数层面的人物提升到了类的层面。但不可否认的是：命令模式很成功的解决了许多问题，正如遍布在 Struts 那样。

解释器模式

一、引子

解释器模式描述了如何构成一个简单的语言解释器，主要应用在使用面向对象语言开发编译器中；在实际应用中，我们可能很少碰到去构造一个语言的文法的情况。

虽然你几乎用不到这个模式，但是看一看还是能受到一定的启发的。

二、定义与结构

解释器模式的定义如下：定义语言的文法，并且建立一个解释器来解释该语言中的句子。它属于类的行为模式。这里的语言意思是使用规定格式和语法的代码。

在 GOF 的书中指出：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。而且当文法简单、效率不是关键问题的时候效果最好——这也就是解释器模式适用的环境。

让我们来看看神秘的解释器模式是由什么来组成的吧。

1) 抽象表达式角色：声明一个抽象的解释操作，这个接口为所有具体表达式角色（抽象语法树中的节点）都要实现的。

   什么叫做抽象语法树呢？《java 与模式》中给的解释为：抽象语法树的每一个节点都代表一个语句，而在每个节点上都可以执行解释方法。这个解释方法的执行就代表这个语句被解释。由于每一个语句都代表这个语句被解释。由于每一个语句都代表一个常见的问题的实例，因此每一个节点上的解释操作都代表对一个问题实例的解答。

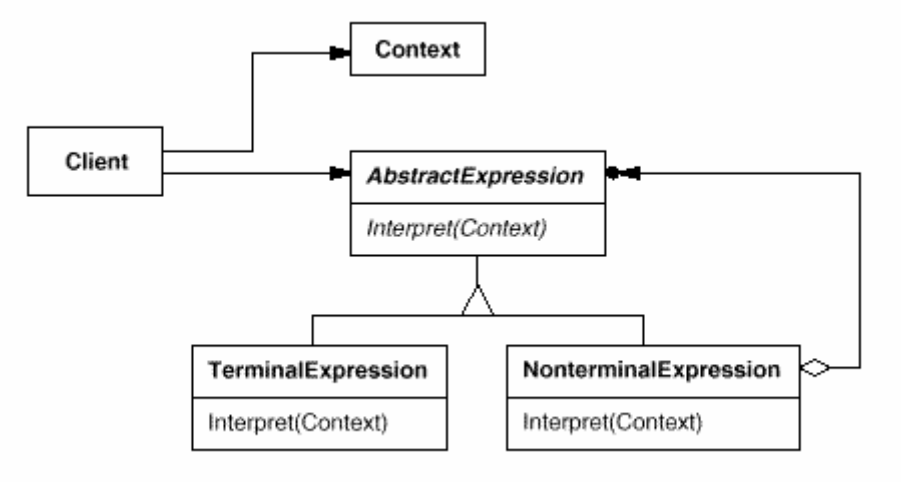
2) 终结符表达式角色：具体表达式。
   a) 实现与文法中的终结符相关联的解释操作
   b) 而且句子中的每个终结符需要该类的一个实例与之对应

3) 非终结符表达式角色：具体表达式。
   a) 文法中的每条规则 R::=R1R2...Rn 都需要一个非终结符表带式角色
   b) 对于从 R1 到 Rn 的每个符号都维护一个抽象表达式角色的实例变量
   c) 实现解释操作，解释一般要递归地调用表示从 R1 到 Rn 的那些对象的解释操作

4) 上下文（环境）角色：包含解释器之外的一些全局信息。

5) 客户角色：
   a) 构建（或者被给定）表示该文法定义的语言中的一个特定的句子的抽象语法树
   b) 调用解释操作

放上张解释器结构类图吧，这也是来自于 GOF 的书中。



呵呵，对每一个角色都给出了详细的职责，而且在类图中给出五个角色之间的关系。这样实现起来也不是很困难了，下面举了一个简单的例子，希望能加深你对解释器模式的理解。

三、举例

来举一个加减乘除的例子吧，实现思路来自于《java 与模式》中的例子。每个角色的功能按照上面提到的规范来实现。

```
//上下文（环境）角色，使用 HashMap 来存储变量对应的数值
class Context{
    private Map valueMap = new HashMap();
    public void addValue(Variable x , int y){
        Integer yi = new Integer(y);
        valueMap.put(x , yi);
    }
    public int LookupValue(Variable x){
        int i = ((Integer)valueMap.get(x)).intValue();
        return i ;
    }
}
//抽象表达式角色，也可以用接口来实现
abstract class Expression{
    public abstract int interpret(Context con);
}
//终结符表达式角色
class Constant extends Expression{
    private int i ;
    public Constant(int i){
        this.i = i;
    }
    public int interpret(Context con){
        return i ;
```

```
    }
}
class Variable extends Expression{
    public int interpret(Context con)    {
        //this 为调用 interpret 方法的 Variable 对象
        return con.LookupValue(this);
    }
}
//非终结符表达式角色
class Add extends Expression{
    private Expression left ,right ;
    public Add(Expression left , Expression right) {
        this.left = left ;
        this.right= right ;
    }
    public int interpret(Context con){
        return left.interpret(con) + right.interpret(con);
    }
}
class Subtract extends Expression{
    private Expression left , right ;
    public Subtract(Expression left , Expression right) {
        this.left = left ;
        this.right= right ;
    }
    public int interpret(Context con){
        return left.interpret(con) - right.interpret(con);
    }
}
class Multiply extends Expression{
    private Expression left , right ;
    public Multiply(Expression left , Expression right){
        this.left = left ;
        this.right= right ;
    }
    public int interpret(Context con){
        return left.interpret(con) * right.interpret(con);
    }
}
class Division extends Expression{
    private Expression left , right ;
    public Division(Expression left , Expression right){
        this.left = left ;
        this.right= right ;
```

```
        }
        public int interpret(Context con){
            try{
                    return left.interpret(con) / right.interpret(con);
            }catch(ArithmeticException ae) {
                System.out.println("被除数为 0！ ");
                return -11111;
            }
        }
    }
    //测试程序，计算 (a*b)/(a-b+2)
    public class Test
    {
        private static Expression ex ;
        private static Context con ;
        public static void main(String[] args){
            con = new Context();
            //设置变量、常量
            Variable a = new Variable();
            Variable b = new Variable();
            Constant c = new Constant(2);
            //为变量赋值
            con.addValue(a , 5);
            con.addValue(b , 7);
            //运算，对句子的结构由我们自己来分析，构造
            ex = new Division(new Multiply(a , b), new Add(new Subtract(a , b) , c));
            System.out.println("运算结果为："+ex.interpret(con));
        }
    }
```

解释器模式并没有说明如何创建一个抽象语法树，因此它的实现可以多种多样，在上面我们是直接在 Test 中提供的，当然还有更好、更专业的实现方式。

对于终结符，GOF 建议采用享元模式来共享它们的拷贝，因为它们要多次重复出现。但是考虑到享元模式的使用局限性，我建议还是当你的系统中终结符重复的足够多的时候再考虑享元模式（关于享元模式，请参考我的《深入浅出享元模式》）。

四、优缺点

解释器模式提供了一个简单的方式来执行语法，而且容易修改或者扩展语法。一般系统中很多类使用相似的语法，可以使用一个解释器来代替为每一个规则实现一个解释器。而且在解释器中不同的规则是由不同的类来实现的，这样使得添加一个新的语法规则变得简单。

但是解释器模式对于复杂文法难以维护。可以想象一下，每一个规则要对应一个处理类，而且这些类还要递归调用抽象表达式角色，多如乱麻的类交织在一起是多么恐怖的一件事啊！

五、总结

这样对解释器模式应该有了些大体的认识了吧，由于这个模式使用的案例匮乏，所以本文大部分观点直接来自于 GOF 的原著。只是实例代码是亲自实现并调试通过的。

迭代器模式

一、 引言

迭代这个名词对于熟悉 Java 的人来说绝对不陌生。我们常常使用 JDK 提供的迭代接口进行 java collection 的遍历：

```
Iterator it = list.iterator();
while(it.hasNext()){
    //using  “it.next();” do some businesss logic
}
```

而这就是关于迭代器模式应用很好的例子。

二、 定义与结构

迭代器（Iterator）模式，又叫做游标（Cursor）模式。GOF 给出的定义为：提供一种方法访问一个容器（container）对象中各个元素，而又不需暴露该对象的内部细节。

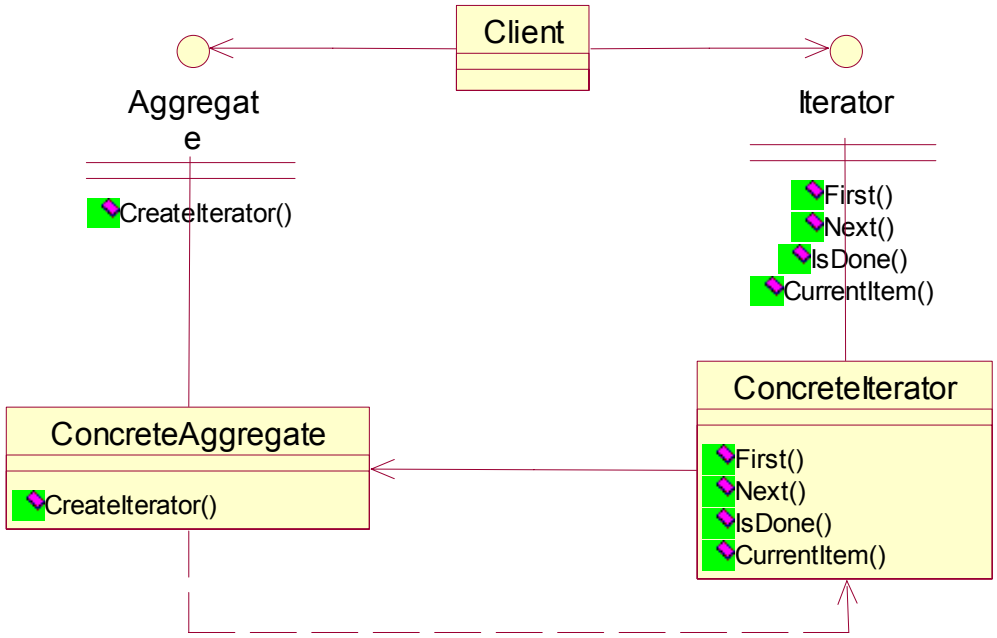
从定义可见，迭代器模式是为容器而生。很明显，对容器对象的访问必然涉及到遍历算法。你可以一股脑的将遍历方法塞到容器对象中去；或者根本不去提供什么遍历算法，让使用容器的人自己去实现去吧。这两种情况好像都能够解决问题。

然而在前一种情况，容器承受了过多的功能，它不仅要负责自己“容器”内的元素维护（添加、删除等等），而且还要提供遍历自身的接口；而且由于遍历状态保存的问题，不能对同一个容器对象同时进行多个遍历。第二种方式倒是省事，却又将容器的内部细节暴露无遗。

而迭代器模式的出现，很好的解决了上面两种情况的弊端。先来看下迭代器模式的真面目吧：

1) 迭代器角色（Iterator）：迭代器角色负责定义访问和遍历元素的接口。
2) 具体迭代器角色（Concrete Iterator）：具体迭代器角色要实现迭代器接口，并要记录遍历中的当前位置。
3) 容器角色（Container）：容器角色负责提供创建具体迭代器角色的接口。
4) 具体容器角色（Concrete Container）：具体容器角色实现创建具体迭代器角色的接口——这个具体迭代器角色于该容器的结构相关。

迭代器模式的类图如下：



从结构上可以看出，迭代器模式在客户与容器之间加入了迭代器角色。迭代器角色的加入，就可以很好的避免容器内部细节的暴露，而且也使得设计符合“单一职责原则”。

注意，在迭代器模式中，具体迭代器角色和具体容器角色是耦合在一起的——遍历算法是与容器的内部细节紧密相关的。为了使客户程序从与具体迭代器角色耦合的困境中脱离出来，避免具体迭代器角色的更换给客户程序带来的修改，迭代器模式抽象了具体迭代器角色，使得客户程序更具一般性和重用性。这被称为多态迭代。

三、 举例

由于迭代器模式本身的规定比较松散，所以具体实现也就五花八门。我们在此仅举一例，根本不能将实现方式一一呈现。因此在举例前，我们先来列举下迭代器模式的实现方式。

1．迭代器角色定义了遍历的接口，但是没有规定由谁来控制迭代。在 Java collection 的应用中，是由客户程序来控制遍历的进程，被称为外部迭代器；还有一种实现方式便是由迭代器自身来控制迭代，被称为内部迭代器。外部迭代器要比内部迭代器灵活、强大，而且内部迭代器在 java 语言环境中，可用性很弱。

2．在迭代器模式中没有规定谁来实现遍历算法。好像理所当然的要在迭代器角色中实现。因为既便于一个容器上使用不同的遍历算法，也便于将一种遍历算法应用于不同的容器。但是这样就破坏掉了容器的封装——容器角色就要公开自己的私有属性，在 java 中便意味着向其他类公开了自己的私有属性。

那我们把它放到容器角色里来实现好了。这样迭代器角色就被架空为仅仅存放一个遍历当前位置的功能。但是遍历算法便和特定的容器紧紧绑在一起了。

而在 Java Collection 的应用中，提供的具体迭代器角色是定义在容器角色中的内部类。这样便保护了容器的封装。但是同时容器也提供了遍历算法接口，你可以扩展自己的迭代器。

好了，我们来看下 Java Collection 中的迭代器是怎么实现的吧。

```
//迭代器角色，仅仅定义了遍历接口
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
//容器角色，这里以 List 为例。它也仅仅是一个接口，就不罗列出来了
//具体容器角色，便是实现了 List 接口的 ArrayList 等类。为了突出重点这里指罗列和迭代器相关的内容
//具体迭代器角色，它是以内部类的形式出来的。AbstractList 是为了将各个具体容器角色的公共部分提取出来而存在的。
public abstract class AbstractList extends AbstractCollection implements List {
    ……
    //这个便是负责创建具体迭代器角色的工厂方法
    public Iterator iterator() {
        return new Itr();
    }

    //作为内部类的具体迭代器角色
    private class Itr implements Iterator {

        int cursor = 0;
        int lastRet = -1;
        int expectedModCount = modCount;

        public boolean hasNext() {
            return cursor != size();
        }

        public Object next() {
            checkForComodification();
            try {
                Object next = get(cursor);
                lastRet = cursor++;
                return next;
            } catch(IndexOutOfBoundsException e) {
                checkForComodification();
                throw new NoSuchElementException();
            }
        }

        public void remove() {
            if (lastRet == -1)
                throw new IllegalStateException();
```

```
            checkForComodification();

            try {
                AbstractList.this.remove(lastRet);
                if (lastRet < cursor)
                    cursor--;
                lastRet = -1;
                expectedModCount = modCount;
            } catch(IndexOutOfBoundsException e) {
                throw new ConcurrentModificationException();
            }
        }

        final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }
```

至于迭代器模式的使用。正如引言中所列那样，客户程序要先得到具体容器角色，然后再通过具体容器角色得到具体迭代器角色。这样便可以使用具体迭代器角色来遍历容器了……

四、 实现自己的迭代器

在实现自己的迭代器的时候，一般要操作的容器有支持的接口才可以。而且我们还要注意以下问题：

在迭代器遍历的过程中，通过该迭代器进行容器元素的增减操作是否安全呢？

在容器中存在复合对象的情况，迭代器怎样才能支持深层遍历和多种遍历呢？

以上两个问题对于不同结构的容器角色，各不相同，值得考虑。

五、 适用情况

由上面的讲述，我们可以看出迭代器模式给容器的应用带来以下好处：

1) 支持以不同的方式遍历一个容器角色。根据实现方式的不同，效果上会有差别。
2) 简化了容器的接口。但是在 java Collection 中为了提高可扩展性，容器还是提供了遍历的接口。
3) 对同一个容器对象，可以同时进行多个遍历。因为遍历状态是保存在每一个迭代器对象中的。

由此也能得出迭代器模式的适用范围：

1) 访问一个容器对象的内容而无需暴露它的内部表示。
2) 支持对容器对象的多种遍历。
3) 为遍历不同的容器结构提供一个统一的接口（多态迭代）。

调停者模式

一、引子

Mediator Pattern 中文译为“中介者模式”、“调停者模式”。其实都不是很好，由于现实生活中的“中介”是要和客户打交道，而省去客户原本繁琐的手续，这一点和门面模式的初衷很相像；而在 Mediator Pattern 中 Mediator 是不可见的。“调停”也不好，因为 Mediator 在程序中存在的初衷仅仅是规范信息传递的方式。

因此叫做“传递器模式”仿佛更能体贴一些，但是本文还是称其为“调停者模式”。

二、定义与结构

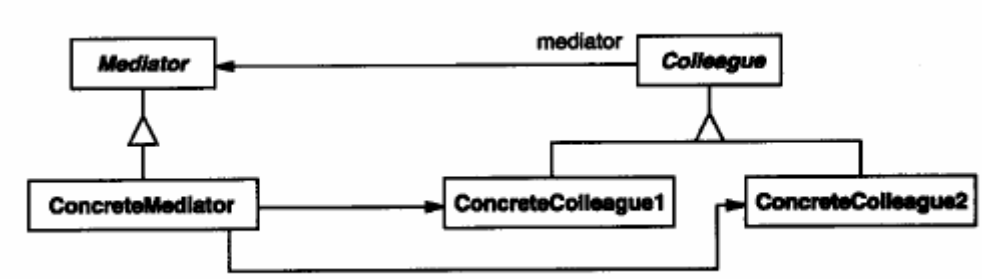
GOF 给调停者模式下的定义是：用一个调停对象来封装一系列的对象交互。调停者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。简单点来说，将原来两个直接引用或者依赖的对象拆开，在中间加入一个"调停"对象，使得两头的对象分别和"调停"对象引用或者依赖。

当然并不是所有的对象都需要加入"调停"对象。如果对象之间的关系原本一目了然，调停对象的加入便是"画蛇添足"。

来看下调停者模式的组成部分吧。

1) 抽象调停者（Mediator）角色：抽象调停者角色定义统一的接口用于各同事角色之间的通信。
2) 具体调停者（Concrete Mediator）角色：具体调停者角色通过协调各同事角色实现协作行为。为此它要知道并引用各个同事角色。
3) 同事（Colleague）角色：每一个同事角色都知道对应的具体调停者角色，而且与其他的同事角色通信的时候，一定要通过调停者角色协作。

来自《设计模式》一书的类图：



由于调停者的行为与要使用的数据与具体业务紧密相关，抽象调停者角色提供一个能方便很多对象使用的接口是不太现实的。所以抽象调停者角色往往是不存在的，或者只是一个标示接口。如果有幸能够提炼出真正带有行为的抽象调停者角色，我想同事角色对具体调停者角色的选择也是策略的一种应用。

呵呵，“恰到好处，过犹不及”。适合自己系统的便是最好的。

三、进一步讨论

是否还记得应用广泛的 MVC 分为哪三层？模型层（Model）、表现层（View）还有控制层（Control\Mediator）。控制层便是位于表现层与模型层之间的调停者。笼统地说 MVC 也算是调停者模式在框架设计中的一个应用。

由于调停者模式在定义上比较松散，在结构上和观察者模式、命令模式十分相像；而应

用目的又与结构模式“门面模式”有些相似。

在结构上，调停者模式与观察者模式、命令模式都添加了中间对象——只是调停者去掉了后两者在行为上的方向。因此调停者的应用可以仿照后两者的例子去写。但是观察者模式、命令模式中的观察者、命令都是被客户所知的，具体哪个观察者、命令的应用都是由客户来指定的；而大多调停者角色对于客户程序却是透明的。当然造成这种区别的原因是由于它们要达到的目的不同。

从目的上看，调停者模式与观察者模式、命令模式便没有了任何关系，倒是与前面讲过的门面模式有些相似。

但是门面模式是介于客户程序与子系统之间的，而调停者模式是介于子系统与子系统之间的。这也注定了它们有很大的区别：门面模式是将原有的复杂逻辑提取到一个统一的接口，简化客户对逻辑的使用。它是被客户所感知的，而原有的复杂逻辑则被隐藏了起来。而调停者模式的加入并没有改变客户原有的使用习惯，它是隐藏在原有逻辑后面的，使得代码逻辑更加清晰可用。

前面已经陆陆续续的将调停者模式的特点写了出来。这里再总结一下。使用调停者模式最大的好处就是将同事角色解耦。这带来了一系列的系统结构改善：提高了原有系统的可读性、简化原有系统的通信协议——将原有的多对多变为一对多、提高了代码的可复用性……

呵呵，但是调停者角色集中了太多的责任，所有有关的同事对象都要由它来控制。这不由得让我想起了简单工厂模式，但是由于调停者模式的特殊性——与业务逻辑密切相关，不能采用类似工厂方法模式的解决方法。因此建议在使用调停者模式的时候注意控制调停者角色的大小。

讨论了这么多关于调停者模式的特点。可以总结出调停者模式的使用时机：一组对象以定义良好但是复杂的方式进行通信，产生了混乱的依赖关系，也导致对象难以复用。

四、总结

调停者模式的使用有着分层设计的雏形。分层是开发人员分离复杂系统时最常用、最普通的技术。J2EE 的三层结构估计大家都耳熟能详。而调停者模式的作用便是将关系错乱复杂、层次不清晰的对象群分割成两层或者三层。

调停者模式很容易在系统中应用，也很容易在系统中误用。当系统出现了“多对多”交互复杂的对象群，不要急于使用调停者模式，而要先反思你的系统在设计上是不是合理。

备忘录模式

一、引子

俗话说：世上难买后悔药。所以凡事讲究个“三思而后行”，但总常见有人做“痛心疾首”状：当初我要是……。如果真的有《大话西游》中能时光倒流的“月光宝盒”，那这世上也许会少一些伤感与后悔——当然这只能是痴人说梦了。

但是在我们手指下的程序世界里，却有的后悔药买。今天我们要讲的备忘录模式便是程序世界里的“月光宝盒”。

二、定义与结构

备忘录（Memento）模式又称标记（Token）模式。GOF 给备忘录模式的定义为：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。这样以后就可将该对象恢复到原先保存的状态。

在讲命令模式的时候，我们曾经提到利用中间的命令角色可以实现 undo、redo 的功能。从定义可以看出备忘录模式是专门来存放对象历史状态的，这对于很好的实现 undo、redo 功能有很大的帮助。所以在命令模式中 undo、redo 功能可以配合备忘录模式来实现。

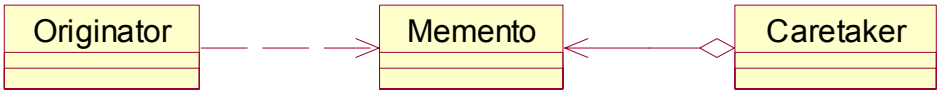
其实单就实现保存一个对象在某一时刻的状态的功能，还是很简单的——将对象中要保存的属性放到一个专门管理备份的对象中，需要的时候则调用约定好的方法将备份的属性放回到原来的对象中去。但是你要好好看看为了能让你的备份对象访问到原对象中的属性，是否意味着你就要全部公开或者包内公开对象原本私有的属性呢？如果你的做法已经破坏了封装，那么就要考虑重构一下了。

备忘录模式只是 GOF 对“恢复对象某时的原有状态”这一问题提出的通用方案。因此在如何保持封装性上——由于受到语言特性等因素的影响，备忘录模式并没有详细描述，只是基于 C++阐述了思路。那么基于 Java 的应用应该怎样来保持封装呢？我们将在实现一节里面讨论。

来看下“月光宝盒”备忘录模式的组成部分：

1) 备忘录（Memento）角色：备忘录角色存储“备忘发起角色”的内部状态。“备忘发起角色”根据需要决定备忘录角色存储“备忘发起角色”的哪些内部状态。为了防止“备忘发起角色”以外的其他对象访问备忘录。备忘录实际上有两个接口，“备忘录管理者角色”只能看到备忘录提供的窄接口——对于备忘录角色中存放的属性是不可见的。“备忘发起角色”则能够看到一个宽接口——能够得到自己放入备忘录角色中属性。
2) 备忘发起（Originator）角色：“备忘发起角色”创建一个备忘录，用以记录当前时刻它的内部状态。在需要时使用备忘录恢复内部状态。
3) 备忘录管理者（Caretaker）角色：负责保存好备忘录。不能对备忘录的内容进行操作或检查。

备忘录模式的类图真是再简单不过了：



三、举例

按照定义中的要求，备忘录角色要保持完整的封装。最好的情况便是：备忘录角色只应该暴露操作内部存储属性的的接口给“备忘发起角色”。而对于其他角色则是不可见的。GOF 在书中以 C++为例进行了探讨。但是在 Java 中没有提供类似于 C++中友元的概念。在 Java

中怎样才能保持备忘录角色的封装呢？

下面对三种在 Java 中可保存封装的方法进行探讨。

第一种就是采用两个不同的接口类来限制访问权限。这两个接口类中，一个提供比较完备的操作状态的方法，我们称它为宽接口；而另一个则可以只是一个标示，我们称它为窄接口。备忘录角色要实现这两个接口类。这样对于“备忘发起角色”采用宽接口进行访问，而对于其他的角色或者对象则采用窄接口进行访问。

这种实现比较简单，但是需要人为的进行规范约束——而这往往是没有力度的。

第二种方法便很好的解决了第一种的缺陷：采用内部类来控制访问权限。将备忘录角色作为“备忘发起角色”的一个私有内部类。好处我不详细解释了，看看代码吧就明白了。下面的代码是一个完整的备忘录模式的教学程序。它便采用了第二种方法来实现备忘录模式。

还有一点值得指出的是，在下面的代码中，对于客户程序来说“备忘录管理者角色”是不可见的，这样简化了客户程序使用备忘录模式的难度。下面采用“备忘发起角色”来调用访问“备忘录管理者角色”，也可以参考门面模式在客户程序与备忘录角色之间添加一个门面角色。

```
class Originator{
    //这个是要保存的状态
    private int state= 90;
    //保持一个“备忘录管理者角色”的对象
    private Caretaker c = new Caretaker();
    //读取备忘录角色以恢复以前的状态
    public void setMemento(){
        Memento memento = (Memento)c.getMemento();
        state = memento.getState();
        System.out.println("the state is "+state+" now");
    }
    //创建一个备忘录角色，并将当前状态属性存入，托给“备忘录管理者角色”存放。
    public void createMemento(){
        c.saveMemento(new Memento(state));
    }

    //this is other business methods...
    //they maybe modify the attribute state

    public void modifyState4Test(int m){
        state = m;
        System.out.println("the state is "+state+" now");
    }
    //作为私有内部类的备忘录角色，它实现了窄接口，可以看到在第二种方法中宽接口已经不再需要
    //注意：里面的属性和方法都是私有的
    private class Memento implements MementoIF{

        private int state ;
```

```
        private Memento(int state){
            this.state = state ;
        }

        private int getState(){
            return state;
        }
    }
}
//测试代码——客户程序
public class TestInnerClass{

    public static void main(String[] args){
        Originator o = new Originator();
        o.createMemento();
        o.modifyState4Test(80);
        o.setMemento();
    }
}
//窄接口
interface MementoIF{}
//“备忘录管理者角色”
class Caretaker{

    private MementoIF m ;

    public void saveMemento(MementoIF m){
        this.m = m;
    }
    public MementoIF getMemento(){
        return m;
    }
}
```

第三种方式是不太推荐使用的：使用 clone 方法来简化备忘录模式。由于 Java 提供了 clone 机制，这使得复制一个对象变得轻松起来。使用了 clone 机制的备忘录模式，备忘录角色基本可以省略了，而且可以很好的保持对象的封装。但是在为你的类实现 clone 方法时一定要慎重。

在上面的教学代码中，我们简单的模拟了备忘录模式的整个流程。在实际应用中，我们往往需要保存大量“备忘发起角色”的历史状态。这时就要对我们的“备忘录管理者角色”进行改造，最简单的方式就是采用容器来按照顺序存放备忘录角色。这样就可以很好的实现 undo、redo 功能了。

四、适用情况

从上面的讨论可以看出，使用了备忘录模式来实现保存对象的历史状态可以有效地保持封装边界。使用备忘录可以避免暴露一些只应由“备忘发起角色”管理却又必须存储在“备忘发起角色”之外的信息。把“备忘发起角色”内部信息对其他对象屏蔽起来，从而保持了封装边界。

但是如果备份的“备忘发起角色”存在大量的信息或者创建、恢复操作非常频繁，则可能造成很大的开销。

GOF 在《设计模式》中总结了使用备忘录模式的前提：

1) 必须保存一个对象在某一个时刻的(部分)状态，这样以后需要时它才能恢复到先前的状态。
2) 如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

五、总结

介绍了怎样来使用备忘录模式实现存储对象历史状态的功能，并对基于 Java 的实现进行了讨论。

观察者模式

一、引子

还记得警匪片上，匪徒们是怎么配合实施犯罪的吗？一个团伙在进行盗窃的时候，总有一两个人在门口把风——如果有什么风吹草动，则会立即通知里面的同伙紧急撤退。也许放风的人并不一定认识里面的每一个同伙；而在里面也许有新来的小弟不认识这个放风的。但是这没什么，这个影响不了他们之间的通讯，因为他们之间有早已商定好的暗号。

呵呵，上面提到的放风者、偷窃者之间的关系就是观察者模式在现实中的活生生的例子。

二、定义与结构

观察者（Observer）模式又名发布-订阅（Publish/Subscribe）模式。GOF 给观察者模式如下定义：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。

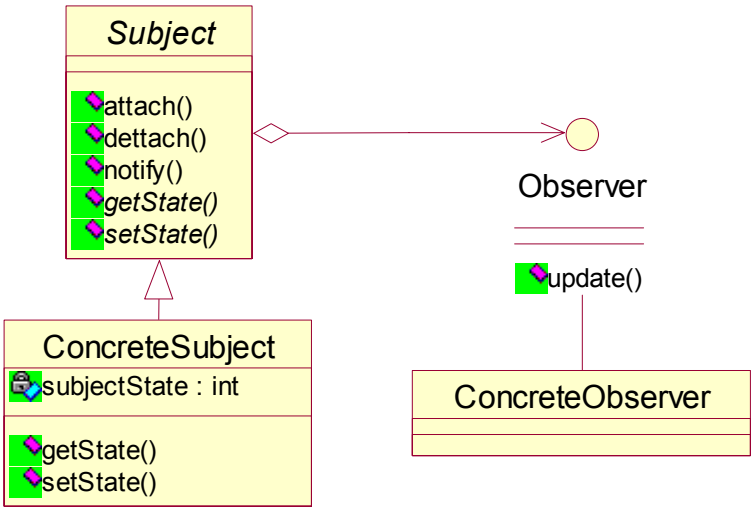
在这里先讲一下面向对象设计的一个重要原则——单一职责原则。系统的每个对象应该将重点放在问题域中的离散抽象上。因此理想的情况下，一个对象只做一件事情。这样在开发中也就带来了诸多的好处：提供了重用性和维护性，也是进行重构的良好的基础。

因此几乎所有的设计模式都是基于这个基本的设计原则来的。观察者模式的起源我觉得应该是在 GUI 和业务数据的处理上，因为现在绝大多数讲解观察者模式的例子都是这一题材。但是观察者模式的应用决不仅限于此一方面。

下面我们就来看看观察者模式的组成部分。

1) 抽象目标角色（Subject）：目标角色知道它的观察者，可以有任意多个观察者观察同一个目标。并且提供注册和删除观察者对象的接口。目标角色往往由抽象类或者接口来实现。
2) 抽象观察者角色（Observer）：为那些在目标发生改变时需要获得通知的对象定义一个更新接口。抽象观察者角色主要由抽象类或者接口来实现。
3) 具体目标角色（Concrete Subject）：将有关状态存入各个 Concrete Observer 对象。当它的状态发生改变时，向它的各个观察者发出通知。
4) 具体观察者角色（Concrete Observer）：存储有关状态，这些状态应与目标的状态保持一致。实现 Observer 的更新接口以使自身状态与目标的状态保持一致。在本角色内也可以维护一个指向 Concrete Subject 对象的引用。

放上观察者模式的类图，这样能将关系清晰的表达出来。



在 Subject 这个抽象类中，提供了上面提到的功能，而且存在一个通知方法：notify。还可以看到 Subject 和 ConcreteSubject 之间可以说是使用了模板模式（这个模式真是简单普遍到一不小心就用到了）。

这样当具体目标角色的状态发生改变，按照约定则会去调用通知方法，在这个方法中则会根据目标角色中注册的观察者名单来逐个调用相应的 update 方法来调整观察者的状态。这样观察者模式就走完了一个流程。

三、举例

观察者模式是我在《JUnit 源代码分析》中遗留的一个模式，因此这里将采用 JUnit 来作为例子。JUnit 为用户提供了三种不同的测试结果显示界面，以后还可能会有其它方式的现实界面……。怎么才能将测试的业务逻辑和显示结果的界面很好的分离开？不用问，就是观察者模式！

下面我们来看看 JUnit 中观察者模式的使用代码：

```
//下面是我们的抽象观察者角色，JUnit 是采用接口来实现的，这也是一般采用的方式。
//可以看到这里面定义了四个不同的 update 方法，对应四种不同的状态变化
public interface TestListener {
    /**
     * An error occurred.
     */
    public void addError(Test test, Throwable t);
    /**
     * A failure occurred.
     */
    public void addFailure(Test test, AssertionFailedError t);
    /**
     * A test ended.
     */
    public void endTest(Test test);
    /**
     * A test started.
```

```
         */
        public void startTest(Test test);
    }
```

//具体观察者角色，我们采用最简单的 TextUI 下的情况来说明（AWT，Swing 对于整天做 Web 应用的人来说，已经很陌生了）

```
    public class ResultPrinter implements TestListener {
        //省略好多啊，主要是显示代码
        ……
        //下面就是实现接口 TestListener 的四个方法
        //填充方法的行为很简单的说
        /**
         * @see junit.framework.TestListener#addError(Test, Throwable)
         */
        public void addError(Test test, Throwable t) {
            getWriter().print("E");
        }
        /**
         * @see junit.framework.TestListener#addFailure(Test, AssertionFailedError)
         */
        public void addFailure(Test test, AssertionFailedError t) {
            getWriter().print("F");
        }
        /**
         * @see junit.framework.TestListener#endTest(Test)
         */
        public void endTest(Test test) {
        }
        /**
         * @see junit.framework.TestListener#startTest(Test)
         */
        public void startTest(Test test) {
            getWriter().print(".");
            if (fColumn++ >= 40) {
                getWriter().println();
                fColumn= 0;
            }
        }
    }
```

来看下我们的目标角色，随便说下，由于 JUnit 功能的简单，只有一个目标——TestResult，因此 JUnit 只有一个具体目标角色。

```
    //好长的代码，好像没有重点。去掉了大部分与主题无关的信息
    //下面只列出了当 Failures 发生时是怎么来通知观察者的
```

```
public class TestResult extends Object {
    //这个是用来存放测试 Failures 的集合
    protected Vector fFailures;
    //这个就是用来存放注册进来的观察者的集合
    protected Vector fListeners;

    public TestResult() {
        fFailures= new Vector();
        fListeners= new Vector();
    }
    /**
     * Adds a failure to the list of failures. The passed in exception
     * caused the failure.
     */
    public synchronized void addFailure(Test test, AssertionFailedError t) {
        fFailures.addElement(new TestFailure(test, t));
        //下面就是通知各个观察者的 addFailure 方法
        for (Enumeration e= cloneListeners().elements(); e.hasMoreElements(); ) {
            ((TestListener)e.nextElement()).addFailure(test, t);
        }
    }
    /**
     * 注册一个观察者
     */
    public synchronized void addListener(TestListener listener) {
        fListeners.addElement(listener);
    }
    /**
     * 删除一个观察者
     */
    public synchronized void removeListener(TestListener listener) {
        fListeners.removeElement(listener);
    }
    /**
     * 返回一个观察者集合的拷贝，当然是为了防止对观察者集合的非法方式操作了
     * 可以看到所有使用观察者集合的地方都通过它
     */
    private synchronized Vector cloneListeners() {
        return (Vector)fListeners.clone();
    }
    ……
}
```

嗯，观察者模式组成所需要的角色在这里已经全了。不过好像还是缺点什么……。呵呵，

对！就是它们之间还没有真正的建立联系。在 JUnit 中是通过 TestRunner 来作的，而你在具体的系统中可以灵活掌握。

看一下 TestRunner 中的代码：

```
public class TestRunner extends BaseTestRunner {
    private ResultPrinter fPrinter;
    public TestResult doRun(Test suite, boolean wait) {
    //就是在这里注册的
        result.addListener(fPrinter);
    ……
```

四、使用情况

GOF 给出了以下使用观察者模式的情况：

1) 当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。
2) 当对一个对象的改变需要同时改变其它对象，而不知道具体有多少对象有待改变。
3) 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之，你不希望这些对象是紧密耦合的。

五、我推你拉

观察者模式在关于目标角色、观察者角色通信的具体实现中，有两个版本。一种情况便是目标角色在发生变化后，仅仅告诉观察者角色"我变化了"；观察者角色如果想要知道具体的变化细节，则就要自己从目标角色的接口中得到。这种模式被很形象的称为：拉模式——就是说变化的信息是观察者角色主动从目标角色中"拉"出来的。

还有一种方法，那就是我目标角色"服务一条龙"，通知你发生变化的同时，通过一个参数将变化的细节传递到观察者角色中去。这就是"推模式"——管你要不要，先给你啦。

这两种模式的使用，取决于系统设计时的需要。如果目标角色比较复杂，并且观察者角色进行更新时必须得到一些具体变化的信息，则"推模式"比较合适。如果目标角色比较简单，则"拉模式"就很合适啦。

策略模式

一、引子

18 日下午 3 时一刻，沈阳，刚刚下完一场几年罕见的大雪，天气格外的冷，公交车在"车涛汹涌"的公路上举步维艰，我坐在里面不时的看表——回公司的班车就要发车了，我还离等车的地方好远……。都是这可恶的天气打乱了我的计划！看来我要重新盘算下下了公交车的计划了：如果在半点以前能够到达等班车的地方，我就去旁边卖书报的小店里面买份《南方周末》，顺便逼逼严寒；如果可恶的公交到时候还不能拱到的话，我就只好去附近的家乐福里面打发两个小时直到下一趟班车发车！

其实在上面提到的就是对两种不同情况所采取的不同的策略。这种情况在实际系统中也是经常遇到，那么你是怎么来实现不同的策略的呢？也许你看了策略模式后会增加一种不错的选择！

二、定义

策略模式（Strategy）属于对象行为型设计模式，主要是定义一系列的算法，把这些算法一个个封装成拥有共同接口的单独的类，并且使它们之间可以互换。策略模式使这些算法在客户端调用它们的时候能够互不影响地变化。这里的算法不要狭义的理解为数据结构中算法，可以理解为不同的业务处理方法。

这种做法会带来什么样的好处呢？

它将算法的使用和算法本身分离，即将变化的具体算法封装了起来，降低了代码的耦合度，系统业务策略的更变仅需少量修改。

算法被提取出来，这样可以使算法得到重用，这种情况还可以考虑使用享元模式来共享算法对象，来减少系统开销（但要注意使用享元模式的建议条件）。

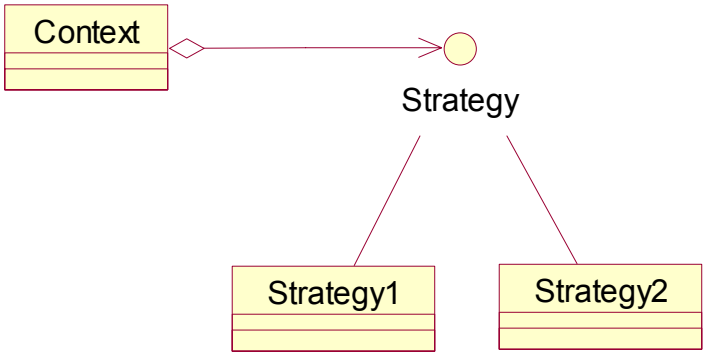
三、结构

先由定义来想象下它的结构吧：要使算法拥有共同的接口，这样就要实现一个接口或者一个抽象类出来才行。这样基本上轮廓也就出来了，我们来看看吧：

策略模式由三个角色组成：

1) 算法使用环境(Context)角色：算法被引用到这里和一些其它的与环境有关的操作一起来完成任务。
2) 抽象策略(Strategy)角色：规定了所有具体策略角色所需的接口。在 java 它通常由接口或者抽象类来实现。
3) 具体策略(Concrete Strategy)角色：实现了抽象策略角色定义的接口。

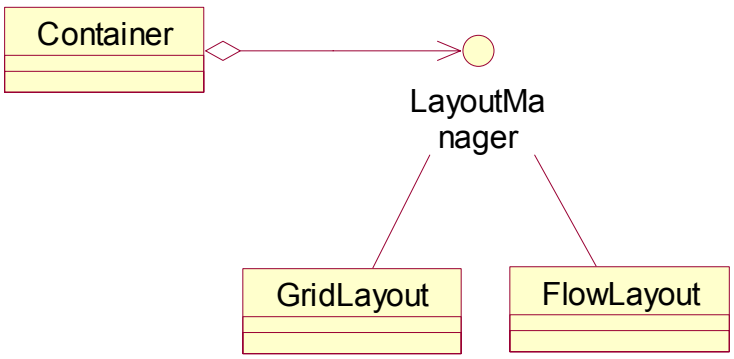
策略模式各个角色之间关系的类图表示：



四、举例

在 Java 语言中对策略模式的应用是很多的，我们这里举个布局管理器的例子。在 java.awt 类库中有很多种设定好了的 Container 对象的布局格式，这些格式你可以在创建软件界面的时候使用到。如果不使用策略模式，那么就没有了对布局格式扩展的可能，因为你要去修改 Container 中的方法，去让它知道你这种布局格式，这显然是不可行的。

让我们来看看 java 源码中的实现吧。先来看看参与的类和他们扮演的角色吧，使用类图是再清楚不过的了。



这里我只放上了能够用来讲解的最少部分。

先来看看 Container 中有关的代码：

```
LayoutManager layoutMgr;       //对布局管理器接口的引用
//获得在使用的具体布局管理器
public LayoutManager getLayout() {
    return layoutMgr;
}
//设置要使用的具体布局管理器
public void setLayout(LayoutManager mgr) {
    layoutMgr = mgr;
    if (valid) {
        invalidate();
    }
}
```

可以看到，Container 根本就不关心你使用的是什么具体的布局管理器，这样也就使得 Container 不会随着布局管理器的增多而修改本身。所以说策略模式是对变化的封装。

下面是布局管理器接口的代码：

```
public interface LayoutManager {
    void addLayoutComponent(String name, Component comp);
    ……
    Dimension minimumLayoutSize(Container parent);
    void layoutContainer(Container parent);
}
```

而具体的布局管理器就是对上面接口方法的实现和扩展，我这里不再展现给大家了，有兴趣的可以查看 JDK 源代码。

来看看对它的使用吧，以下是示意性代码：

```
public class FlowLayoutWindow extends JApplet
{
    public void init()
    {
        Containter cp = getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i = 0 ; i < 20 ; i++)
            cp.add(new JButton("Button" + i));
    }
    ......
}
```

对具体布局管理器的感知直到最后的客户程序中才得到，在此之前是不关心的。但是这一点不能忽视——客户必须知道有哪些策略方法可以使用，这也限制了它的使用范围。

五、使用建议

下面是使用策略模式的一些建议：

1) 系统需要能够在几种算法中快速的切换。
2) 系统中有一些类它们仅行为不同时，可以考虑采用策略模式来进行重构
3) 系统中存在多重条件选择语句时，可以考虑采用策略模式来重构。

但是要注意一点，策略模式中不可以同时使用多于一个的算法。

状态模式

一、引子

状态模式自身结构非常简单——前面刚刚介绍了几个结构比较简单的设计模式，和他们一样，状态模式在具体实现上留下了可变换的余地。我前面已经介绍过它的孪生兄妹策略模式了，大家可以两者比较着阅读。本文将会讨论两者的区别。

二、定义与结构

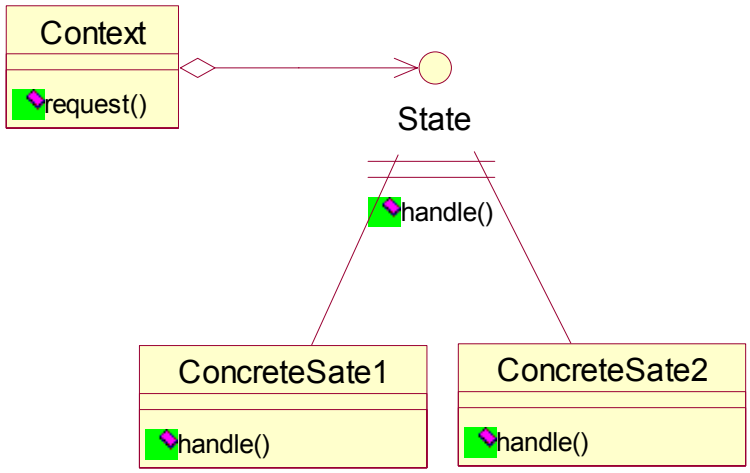
GOF《设计模式》中给状态模式下的定义为：允许一个对象在其内部状态改变时改变它的行为。这个对象看起来似乎修改了它的类。看起来，状态模式好像是神通广大——居然能够"修改自身的类"！

能够让程序根据不同的外部情况来做出不同的响应，最直接的方法就是在程序中将这些可能发生的外部情况全部考虑到，使用 if else 语句来进行代码响应选择。但是这种方法对于复杂一点的状态判断，就会显得杂乱无章，容易产生错误；而且增加一个新的状态将会带来大量的修改。这个时候“能够修改自身”的状态模式的引入也许是个不错的主意。

状态模式可以有效的替换充满在程序中的 if else 语句：将不同条件下的行为封装在一个类里面，再给这些类一个统一的父类来约束他们。来看一下状态模式的角色组成吧：

1) 使用环境（Context）角色：客户程序是通过它来满足自己的需求。它定义了客户程序需要的接口；并且维护一个具体状态角色的实例，这个实例来决定当前的状态。
2) 状态（State）角色：定义一个接口以封装与使用环境角色的一个特定状态相关的行为。
3) 具体状态（Concrete State）角色：实现状态角色定义的接口。

类图如下，结构非常简单也与策略模式非常相似。



三、实现

由于状态模式结构非常简单，所以在这里罗列一些反映状态模式实现结构的代码没有什么太大的作用。如果你有兴趣的话可以按照上面类图来编写一下。

在引子中已经提到，状态模式在具体实现上存在不同的方案。因此这里重点就这些不同的实现方式进行介绍和讨论。

首先，实现时是否将状态角色、具体状态角色暴露给客户程序？按照 GOF 的建议是不希望将状态角色暴露给客户程序的，与客户程序打交道的仅仅是使用环境角色，客户是不知

道系统是怎么实现的，更不关心什么有几个具体状态。但是当使用环境角色中的初始状态紧紧依赖于客户程序时，适乎暴露是在所难免的——这就与策略模式异常相似了！

具体状态角色中的行为一般是与使用环境角色密切相关的。因此这里便有了一个小细节：我们把使用环境角色作为参数传递进入具体状态角色后，是在具体状态角色中来实现状态响应行为；还是仅仅调用在使用环境角色中已经实现了的方法？由于这些行为往往与使用环境角色相关，所以按照《重构》一书的"指导"——后一种实现方法是比较地道的。

从定义可知，状态模式是要应对状态转换的。那么状态的转换在哪里定义呢？你可以选择在使用环境角色的代码中来表现出来，当然这便意味着状态转变的规则就固定下来了。GOF 还给出了另外一种稍微灵活一点的实现方式：在每一个具体状态角色中来指定后续状态以及何时进行转换。

其实在 java 强大的反射机制的支持下，我们还可以将状态的转换做的更加灵活——我们可以将状态转换的规则写在.xml 等等的配置文件里面甚至是数据库中，我们姑且叫做状态转换表。进行转换前，根据状态转换表来读取下一个状态，然后利用反射获得具体的状态对象……。哈哈，看起来很不错的样子，只是效率可能低一些——当然在企业应用中这应该不是最重要的。

状态模式已经被我们想象着"实现"了一番。那么状态模式的引入会给我们的程序带来哪些优势呢？前面我们已经说过：状态模式的引入免除了代码中复杂而庸长的逻辑判断语句。而且具体状态角色将具体状态和它对应的行为封装了起来，这使得增加一种新的状态变得简单一些。而且如果设计合理得话，具体状态角色可以被重用（和策略模式一样，可以考虑使用享元模式来实现）。

使用状态模式也会带来一些问题。每个状态对应一个具体的状态类，使得整体分散，逻辑不太清晰。当然对于一个状态非常多的系统，状态模式带来的优点还是大于它的缺点的。

由上面的分析就可以很明确的知道什么时候该使用状态模式了。下面是 GOF 在《设计模式》中给出的状态模式的适用情况：

1) 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。
2) 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。

四、状态 VS 策略

仔细对比状态模式和策略模式，难免会产生疑问：这两个明明是一个东西嘛！下面我们就来分析下两者区别。

首先我要声明，在实际应用中只要能够使得你的代码灵活漂亮起来，何必计较这些方方面面的差别呢？

Brandon Goldfedder 在《模式的乐趣》里是怎么说的："strategy 模式在结构上与 state 模式非常相似，但是在概念上，他们的目的差异非常大。区分这两个模式的关键是看行为是由状态驱动还是由一组算法驱动，这条规则似乎有点随意，但是在判断时还是需要考虑它。通常，State 模式的"状态"是在对象内部的，Strategy 模式的"策略"可以在对象外部，不过这也不是一条严格、可靠的规则。"

我很同意 Brandon Goldfedder 的观点。这两个模式的划分，就在于使用的目的是不同的——策略模式用来处理算法变化，而状态模式则是处理状态变化（好玄乎阿）。

策略模式中，算法是否变化完全是由客户程序开决定的，而且往往一次只能选择一种算法，不存在算法中途发生变化的情况。从《深入浅出策略模式》中的例子可以很好的看出。

而状态模式如定义中所言，在它的生命周期中存在着状态的转变和行为得更改，而且状态变化是一个线形的整体；对于客户程序来言，这种状态变化往往是透明的。

模板模式

一、引子

这是一个很简单的模式，却被非常广泛的使用。之所以简单是因为在这个模式中仅仅使用到了继承关系。

继承关系由于自身的缺陷，被专家们扣上了"罪恶"的帽子。"使用委派关系代替继承关系"，"尽量使用接口实现而不是抽象类继承"等等专家警告，让我们这些菜鸟对继承"另眼相看"。

其实，继承还是有很多自身的优点所在。只是被大家滥用的似乎缺点更加明显了。合理的利用继承关系，还是能对你的系统设计起到很好的作用的。而模板方法模式就是其中的一个使用范例。

二、定义与结构

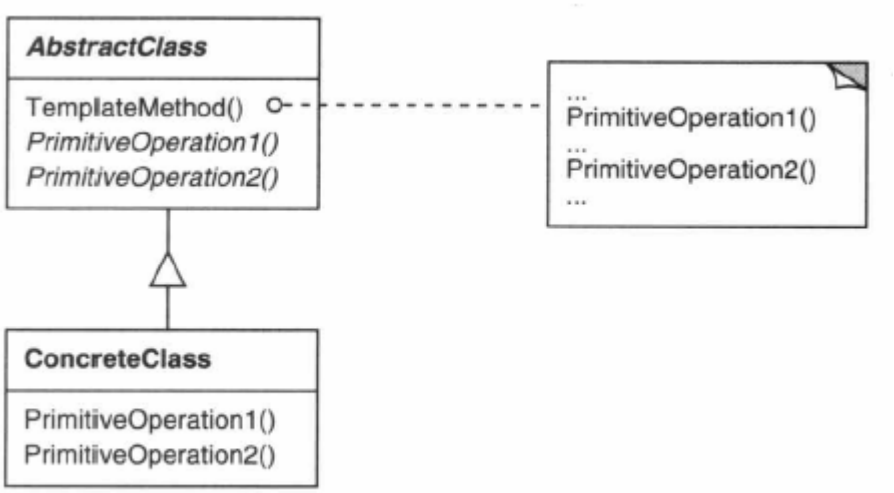
模板方法（Template Method）模式：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。这里的算法的结构，可以理解为你根据需求设计出来的业务流程。特定的步骤就是指那些可能在内容上存在变数的环节。

可以看出来，模板方法模式也是为了巧妙解决变化对系统带来的影响而设计的。使用模板方法使系统扩展性增强，最小化了变化对系统的影响。这一点，在下面的举例中可以很明显的看出来。

来看下这个简单模式的结构吧：

1) 抽象类（Abstract Class）：定义了一到多个的抽象方法，以供具体的子类来实现它们；而且还要实现一个模板方法，来定义一个算法的骨架。该模板方法不仅调用前面的抽象方法，也可以调用其他的操作，只要能完成自身的使命。
2) 具体类（Concrete Class）：实现父类中的抽象方法以完成算法中与特定子类相关的步骤。

下面是模板方法模式的结构图。直接把《设计模式》上的图拿过来用下：



三、举例

还是在我刚刚分析完源码的 JUnit 中找个例子吧。JUnit 中的 TestCase 以及它的子类就是一个模板方法模式的例子。在 TestCase 这个抽象类中将整个测试的流程设置好了，比如

先执行 Setup 方法初始化测试前提，在运行测试方法，然后再 TearDown 来取消测试设置。但是你将在 Setup、TearDown 里面作些什么呢？鬼才知道呢!! 因此，而这些步骤的具体实现都延迟到子类中去，也就是你实现的测试类中。

来看下相关的源代码吧。

这是 TestCase 中，执行测试的模板方法。你可以看到，里面正像前面定义中所说的那样，它制定了“算法”的框架——先执行 setUp 方法来做下初始化，然后执行测试方法，最后执行 tearDown 释放你得到的资源。

```
public void runBare() throws Throwable {
    setUp();
    try {
        runTest();
    }
    finally {
        tearDown();
    }
}
```

这就是上面使用的两个方法。与定义中不同的是，这两个方法并没有被实现为抽象方法，而是两个空的无为方法（被称为钩子方法）。这是因为在测试中，我们并不是必须要让测试程序使用这两个方法来初始化和释放资源的。如果是抽象方法，则子类们必须给它一个实现，不管用到用不到。这显然是不合理的。使用钩子方法，则你在需要的时候，可以在子类中重写这些方法。

```
protected void setUp() throws Exception {
}
protected void tearDown() throws Exception {
}
```

四、适用情况

根据上面对定义的分析，以及例子的说明，可以看出模板方法适用于以下情况：

1) 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。
2) 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。其实这可以说是一种好的编码习惯了。
3) 控制子类扩展。模板方法只在特定点调用操作，这样就只允许在这些点进行扩展。比如上面 runBare（）方法就只在 runTest 前面适用 setUp 方法。如果你不愿子类来修改你的模板方法定义的框架，你可以采用两种方式来做：一是在 API 中不体现出你的模板方法；或者将你的模板方法置为 final 就可以了。

可以看出，使用模板方法模式可以将代码的公共行为提取出来，达到复用的目的。而且，在模板方法模式中，是由父类的模板方法来控制子类中的具体实现。这样你在实现子类的时候，根本不需要对业务流程有太多的了解。

访问者模式

一、引子

对于系统中一个已经完成的类层次结构，我们已经给它提供了满足需求的接口。但是面对新增加的需求，我们应该怎么做呢？如果这是为数不多的几次变动，而且你不用为了一个需求的调整而将整个类层次结构统统地修改一遍，那么直接在原有类层次结构上修改也许是个不错的主意。

但是往往我们遇到的却是：这样的需求变动也许会不停的发生；更重要的是需求的任何变动可能都要让你将整个类层次结构修改个底朝天……。这种类似的操作分布在不同的类里面，不是一个好现象，我们要对这个结构重构一下了。

那么，访问者模式也许是你很好的选择。

二、定义与结构

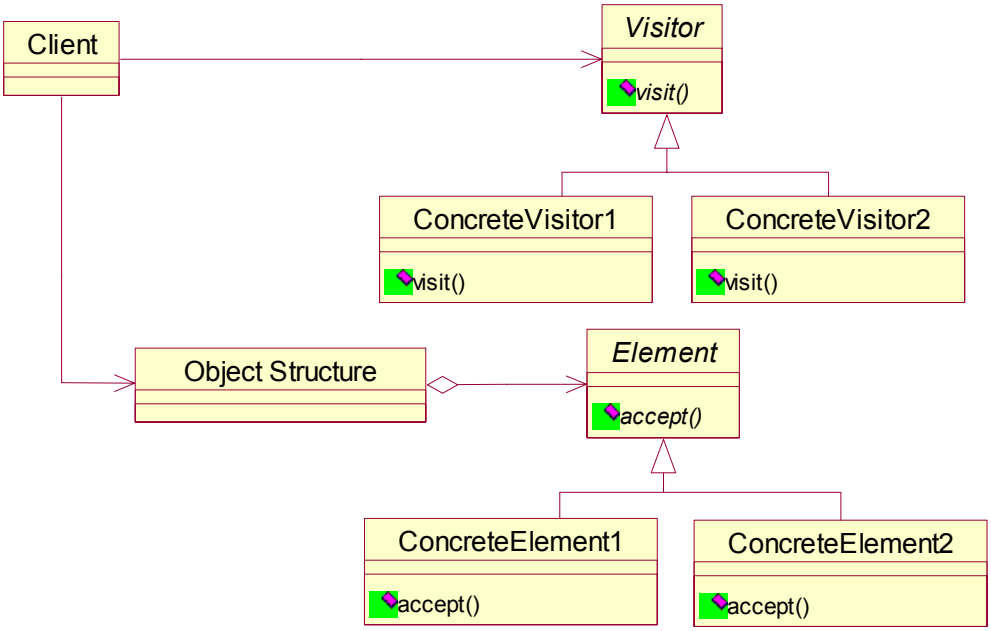
访问者模式，顾名思义使用了这个模式后就可以在不修改已有程序结构的前提下，通过添加额外的"访问者"来完成对已有代码功能的提升。

《设计模式》一书对于访问者模式给出的定义为：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。从定义可以看出结构对象是使用访问者模式必须条件，而且这个结构对象必须存在遍历自身各个对象的方法。这便类似于 java 中的 collection 概念了。

以下是访问者模式的组成结构：

1) 访问者角色（Visitor）：为该对象结构中具体元素角色声明一个访问操作接口。该操作接口的名字和参数标识了发送访问请求给具体访问者的具体元素角色。这样访问者就可以通过该元素角色的特定接口直接访问它。
2) 具体访问者角色（Concrete Visitor）：实现每个由访问者角色（Visitor）声明的操作。
3) 元素角色（Element）：定义一个 Accept 操作，它以一个访问者为参数。
4) 具体元素角色（Concrete Element）：实现由元素角色提供的 Accept 操作。
5) 对象结构角色（Object Structure）：这是使用访问者模式必备的角色。它要具备以下特征：能枚举它的元素；可以提供一个高层的接口以允许该访问者访问它的元素；可以是一个复合（组合模式）或是一个集合，如一个列表或一个无序集合。

来张类图就能更加清晰的看清访问者模式的结构了。



那么像引言中假想的，我们应该做些什么才能让访问者模式跑起来呢？首先我们要在原有的类层次结构中添加 accept 方法。然后将这个类层次中的类放到一个对象结构中去。这样再去创建访问者角色……

三、举例

由于没能在实际应用中找到使用访问者模式的例子。只好借《Thinking in Patterns with java》中的教学代码一用。我稍微做了下修改。

```
import java.util.*;
import junit.framework.*;

//访问者角色
interface Visitor {
  void visit(Gladiolus g);
  void visit(Runuculus r);
  void visit(Chrysanthemum c);
}

// The Flower hierarchy cannot be changed:
//元素角色
interface Flower {
  void accept(Visitor v);
}

//以下三个具体元素角色
class Gladiolus implements Flower {
```

```
  public void accept(Visitor v) { v.visit(this);}
}

class Runuculus implements Flower {
  public void accept(Visitor v) { v.visit(this);}
}

class Chrysanthemum implements Flower {
  public void accept(Visitor v) { v.visit(this);}
}

// Add the ability to produce a string:
//实现的具体访问者角色
class StringVal implements Visitor {

  String s;

  public String toString() { return s; }

  public void visit(Gladiolus g) {
    s = "Gladiolus";
  }

  public void visit(Runuculus r) {
    s = "Runuculus";
  }

  public void visit(Chrysanthemum c) {
    s = "Chrysanthemum";
  }
}


// Add the ability to do "Bee" activities:
//另一个具体访问者角色
class Bee implements Visitor {

  public void visit(Gladiolus g) {
    System.out.println("Bee and Gladiolus");
  }

  public void visit(Runuculus r) {
    System.out.println("Bee and Runuculus");
  }
```

```
  public void visit(Chrysanthemum c) {
    System.out.println("Bee and Chrysanthemum");
  }
}

//这是一个对象生成器
//这不是一个完整的对象结构，这里仅仅是模拟对象结构中的元素
class FlowerGenerator {

  private static Random rand = new Random();

  public static Flower newFlower() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Gladiolus();
      case 1: return new Runuculus();
      case 2: return new Chrysanthemum();
    }
  }
}

//客户测试程序
public class BeeAndFlowers extends TestCase  {

  /*
   在这里你能看到访问者模式执行的流程：
    首先在客户端先获得一个具体的访问者角色
    遍历对象结构
    对每一个元素调用 accept 方法，将具体访问者角色传入
    这样就完成了整个过程
   */
  //对象结构角色在这里才组装上
  List flowers = new ArrayList();

  public BeeAndFlowers() {
    for(int i = 0; i < 10; i++)
      flowers.add(FlowerGenerator.newFlower());
  }

  Visitor sval ;

  public void test() {
    // It's almost as if I had a function to
```

```
        // produce a Flower string representation:
        //这个地方你可以修改以便使用另外一个具体访问者角色
        sval = new StringVal();
        Iterator it = flowers.iterator();
        while(it.hasNext()) {
          ((Flower)it.next()).accept(sval);
          System.out.println(sval);
        }
      }

      public static void main(String args[]) {
        junit.textui.TestRunner.run(BeeAndFlowers.class);
      }
    }
```

四、双重分派

对了，你在上面的例子中体会到双重分派的实现了没有？

首先在客户程序中将具体访问者模式作为参数传递给具体元素角色（加亮的地方所示）。这便完成了一次分派。

进入具体元素角色后，具体元素角色调用作为参数的具体访问者模式中的 visitor 方法，同时将自己（this）作为参数传递进去。具体访问者模式再根据参数的不同来选择方法来执行（加亮的地方所示）。这便完成了第二次分派。

五、优缺点及适用情况

先来看下访问者模式的使用能否避免引言中的痛苦。使用了访问者模式以后，对于原来的类层次增加新的操作，仅仅需要实现一个具体访问者角色就可以了，而不必修改整个类层次。而且这样符合"开闭原则"的要求。而且每个具体的访问者角色都对应于一个相关操作，因此如果一个操作的需求有变，那么仅仅修改一个具体访问者角色，而不用改动整个类层次。

看来访问者模式确实能够解决我们面临的一些问题。

而且由于访问者模式为我们的系统多提供了一层"访问者"，因此我们可以在访问者中添加一些对元素角色的额外操作。

但是"开闭原则"的遵循总是片面的。如果系统中的类层次发生了变化，会对访问者模式产生什么样的影响呢？你必须修改访问者角色和每一个具体访问者角色……

看来访问者角色不适合具体元素角色经常发生变化的情况。而且访问者角色要执行与元素角色相关的操作，就必须让元素角色将自己内部属性暴露出来，而在 java 中就意味着其它的对象也可以访问。这就破坏了元素角色的封装性。而且在访问者模式中，元素与访问者之间能够传递的信息有限，这往往也会限制访问者模式的使用。

《设计模式》一书中给出了访问者模式适用的情况：

1) 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。
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，而你想避免让这些操作"污染"这些对象的类。Visitor 使得你可以将相关的操作集中起来定义在一个类中。
3) 当该对象结构被很多应用共享时，用 Visitor 模式让每个应用仅包含需要用到的操作。

4) 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

你是否能很好的理解呢？

六、总结

这是一个巧妙而且复杂的模式，它的使用条件比较苛刻。当系统中存在着固定的数据结构（比如上面的类层次），而有着不同的行为，那么访问者模式也许是个不错的选择。

附件——《话说分派》

一、引言

这篇文章，完全是为了更好的讲解访问者(Visitor)模式而写的。让我们进入这扑朔迷离的分派世界吧（是不是有点夸张了，汗）。

二、名词解释

先来解释下分派的意思吧。。

在 OO（object-oriented）语言中使用了继承来描述不同的类之间的"社会关系"——类型层次。而这些类实例化的对象们则是对这个类型层次的体现。因此大部分 OO 语言的对象都存在两个身份证：静态类型和实际类型。所谓静态类型，就是对象被声明时的类型；而实际类型则便是创建对象时的类型。举个例子：

B 是 A 的子类。则

A object1 = new B ( );

中 object1 的静态类型便是 A，而实际类型却是 B。在 Java 语言中，编译器会根据对象的静态类型来检查错误；而在运行时，则使用对象的真实身份。

OO 还有一个重要的特点：一个类中可以存在两个相同名称的方法，而它们是根据参数类型的不同来区分的。

正因以上两个原因，便产生了分派——根据类型的不同来选择不同的方法的过程——OO 语言的重要特性。

三、分类

分派可以发生在编译期或者是运行期。因此按此标准，分派分为静态分派和动态分派。

在程序的编译期，只有对象的静态类型是有效的，因此静态分派就是根据对象（包括参数对象）的静态类型来选择方法的。最典型的便是方法重载（overloading）。

在运行期，动态分派会根据对象的实际类型来选择方法。典型的例子便是方法重置（overriding）

而 OO 语言正是由以上两种分派方式来提供多态特性的。

按照选择方法时所参照的类型的个数，分派分为单分派和多分派。OO 语言也因此分为了单分派（Uni-dispatch）语言和多分派（Multi-dispatch）语言。比如 Smalltalk 就是单分派语言，而 CLOS 和 Cecil 就是多分派语言。

说道多分派，就不得提到另一个概念：多重分派（multiple dispatch）。它指由多个单分派组成的分派过程（而多分派则往往不能分割的）。因此单分派语言可以通过多重分派的方式来实现和多分派语言一样的效果。

那么我们熟悉的 Java 语言属于哪一种分派呢？

四、Java 分派实践

先来看看在 Java 中最常见的特性：重载（overloading）与重置（overriding）。

下面是重载的一个具体的小例子，这是一个再简单不过的代码了：

```
//Test For OverLoading
public class Test{
    public void doSomething(int i){
        System.out.println("doString int = "+ i );
    }
```

```
    public void doSomething(String s){
        System.out.println("doString String = "+ s);
    }
    public void doSomething(int i , String s){
        System.out.println("doString int = "+ i +" String = "+ s);
    }
    public static void main(String[] rags){
        Test t = new Test();
        int i = 0;
        t.doSomething(i);
    }
}
```

没什么好稀奇的，你对这部分知识已经熟练掌握了，那么你对下面这段代码的用意也一定了如指掌了吧。

```
//Test For Overriding
public class Test{
    public static void main(String[] rags){
        Father f = new Father();
        Father s = new Son();
        f.dost();
        s.dost();
    }
}
class Father {
    public void dost(){
        System.out.println("Welcome Father!");
    }
}
class Son extends Father{
    public void dost(){
        System.out.println("Welcome Son!");
    }
}
```

那么下面这个代码呢？

```
public class Test{
    public static void main(String[] rags){
        Father f = new Father();
        Father s = new Son();

        f.dost(1);
        s.dost(4);
        s.dost("dispatchTest");
        //s.dost("test" , 5);
    }
```

```
}
class Father {

    public void dost(int i){
        System.out.println("Welcome Father! int = "+ i);
    }
    public void dost(String s){
        System.out.println("Welcome Father! String = "+ s);
    }
}
class Son extends Father{

    public void dost(int i){
        System.out.println("Welcome Son! int = "+i);
    }
    public void dost(String s ,int i ){
        System.out.println("Welcome Son! String = "+s+" int = "+i);
    }
}
```

在编译期间，编译器根据 f、s 的静态类型来为他们选择了方法，当然都选择了父类 Father 的方法。而到了运行期，则又根据 s 的实际类型动态的替换了原来选择的父类中的方法。这便是结果产生的原因。

如果把上面代码中的注释去掉，则会出现编译错误。原因便是在编译期，编译器根据 s 的静态类型 Father 找不到带有两个参数的方法 dost。

再来一个，可要注意看了：

```
public class Test{
    //这几个方法，唯独的不同便在这参数上
    public void dost(Father f , Father f1){
        System.out.println("ff");
    }
    public void dost(Father f , Son s){
        System.out.println("fs");
    }
    public void dost(Son s , Son s2){
        System.out.println("ss");
    }
    public void dost(Son s , Father f){
        System.out.println("sf");
    }

    public static void main(String[] rags){
        Father f = new Father();
        Father s = new Son();
        Test t = new Test();
```

```
            t.dost(f , new Father());
            t.dost(f , s);
            t.dost(s, f);
        }
}
class Father {}
class Son extends Father{}
```

执行结果没有像预期的那样输出 ff、fs、sf 而是输出了三个 ff。为什么？原因便是在编译期，编译器使用 s 的静态类型为其选择方法，于是这三个调用都选择了第一个方法；而在运行期，由于 Java 仅仅根据方法所属对象的实际类型来分派方法，因此这个"错误"就没有被纠正而一直错了下去.....

可以看出，Java 在静态分派时，可以根据 n（n>0）个参数类型来选择不同的方法，这按照上面的定义应该属于多分派的范围。而在运行期时，则只能根据方法所属对象的实际类型来进行方法的选择，这又属于单分派的范围。

因此可以说 Java 语言支持静态多分派和动态单分派。

五、小插曲

你看看下面的代码会怎么执行呢？

```
public class Test{
    public static void main(String[] rags){
        Father f = new Father();
        Father s = new Son();

        System.out.println("f.i " + f.i);
        System.out.println("s.i " + s.i);
        f.dost();
        s.dost();
    }
}
class Father {
    int i = 0 ;
    public void dost(){
        System.out.println("Welcome Father!");
    }
}
class Son extends Father{
    int i = 9 ;
    public void dost(){
        System.out.println("Welcome Son!");
    }
}
```

运行结果：

```
  \>java Test
```

f.i 0
s.i 0
Welcome Father!
Welcome Son!

产生的原因是 Java 编译和运行程序的机制。“数据是什么”是由编译时决定的；而“方法是哪个”则在运行时决定。

六、双重分派

Java 不能支持动态多分派，但是可以通过代码设计来实现动态的多重分派。这里举一个双重分派的实现例子。

大致的思想便是通过一个参数来传递 JVM 不能判断的类型。通过 Java 的动态单分派来完成一次分派后，在方法中使用 instanceof 来判断参数的类型，进而决定执行哪个相关方法。

```
public class Test{
public static void main(String[] rags){
    Father f = new Father();
    Father s = new Son();
    s.dosh(f);
    s.dosh(s);
    f.dosh(s);
    f.dosh(f);
}
}
class Father {
public void dosh(Father f){
    if(f instanceof Son){
        System.out.println("Here is Father's Son");
    }else if(f instanceof Father){
        System.out.println("Here is Father's Father");
    }
}
}

class Son extends Father{
public void dosh(Father f){
    if(f instanceof Son){
        System.out.println("Here is Son's Son");
    }else if(f instanceof Father){
        System.out.println("Here is Son's Father");
    }
}
}
```

执行结果：

Here is Son's Father
Here is Son's Son

Here is Father's Son
Here is Father's Father

呵呵，慢慢在代码中琢磨吧。用这种方式来实现双重分派，思路比较简单清晰。但是对于复杂一点的程序，则代码显得冗长，不易读懂。而且添加新的类型比较麻烦，不是一种好的设计方案。访问者（Visitor）模式则较好的解决了这种模式的不足。至于访问者模式的实现.....

请关注我的《深入浅出访问者模式》。