

一、统一建模语言 UML

今天开始重温设计模式,我想把自己学习的过程分享给大家,同时希望大家多多留言来讨论,相互学习相互进步。

设计模式学习过程中需要借助 UML 来建模,把设计模式图形化,从而让我们更好的理解设计模式内容。什么是 UML, UML 是统一建模语言 (Unified Modeling Language) 的缩写,是当今软件设计的标准图标式设计语言。UML 包括

- 1、用例图 (Use case diagrams),
- 2、类图 (Class diagrams),
- 3、序列图 (Sequence diagrams),
- 4、协作图 (Collaboration diagrams),
- 5、状态图 (Statechart diagrams),
- 6、活动图 (Activity diagrams),
- 7、构件图 (Component diagrams),
- 8、部署图 (Deployment diagrams)

按照这些图的用意大致可以将他们分为两类: 结构图 和 行为图

结构图:

名称	介绍
类图	类图描述一些类,包的静态结构和它们之间的静态关系
对象图	对象图给出一个系统中的对象快照
构件图	描述可以部署的软件构件 (比如 jar, ejb 等) 之间的关系
部署图	描述一个系统软件的拓扑结构

行为图:

名称	介绍
用例图	用例图描述一系列的角色和用例以及他们之间的关系,用来对系统的基本行为进行建模
活动图	描述不同过程之间的动态接触,活动图是用例图所描述的行为的具体化表现
状态图	描述一系列对象内部状态及其状态变化和转移。
时序图	时序图是一种相互作用图,描述不同对象之间信息传递的时序
协作图	是一种相互作用图,描述发出信息,接收信息的一系列对象的组织结构

最常用的 UML 图有: 类图,用例图,时序图 UML 的建模工具有很多,如 Visio,Rose,EA, PD 等。

二、面向对象设计原则

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

软件设计的核心是提高软件的可复用性和可维护性。通常一个软件之所以可复用性和可扩展性差的原因在于设计过于僵硬, 过于脆弱, 复用率低, 粘度过高等原因导致的, 这时候需要想办法提高可扩展性, 灵活性和可插入性, 从而提高软件的可复用性和可维护性。一般可维护性和可复用性不能同时能达到目的, 只有遵循一定的设计原则, 设计出来的系统才能同时满足可复用性和可维护性。面向对象设计原则主要有如下几条:

- 1、“开闭”原则 (Open-Closed Principle) 简称 OCP, 讲的是一个软件实体应该对扩展开放对修改关闭。
- 2、里氏代换原则 (Liskov Substitution Principle) 简称 LSP, 讲的是任何父类出现的地方都可以被子类代替。
- 3、依赖倒转原则 (Dependency INversion Principle) 简称 DIP, 讲的是要依赖于抽象不要依赖于实现。
- 4、接口隔离原则 (Interface Segregation Principle) 简称 ISP, 讲的是为客户端提供尽可能小的单独的接口, 而不是提供大的总接口。
- 5、组合/聚合服用原则 (Composition/Aggregation Principle) 简称 CARP, 讲的是要尽可能使用组合, 聚合来达到复用目的而不是利用继承。
- 6、迪米特法则 (Law of Demeter) 简称 LoD, 讲的是一个软件实体应当与尽可能少的其他软件实体发生相互作用。

为什么要在讲设计模式前讲设计原则, 是因为设计模式是面向对象设计原则的具体指导, 所以有了理论和设计指导我们就可以进入设计模式学习了, 设计模式大家常说的有23中, 其实现实中要多的多, 大概分为三类: 创建模式, 结构模式和行为模式。

三、设计模式概述

上一节里提到设计模式分为创建模式, 结构模式和行为模式, 这节课我们来学习它们的定义以及它们包含哪些具体的设计模式。

一、创建模式

创建模式是对类的实例化过程的抽象化。在一些系统里, 可能需要动态的决定怎样创建对象, 创建哪些对象, 以及如何组合和表示这些对象。创建模式描述了怎么构造和封装这些动态的决定。

创建模式分为类的创建模式和对象的创建模式两种。

- 1、类的创建模式 类的创建模式使用继承关系, 把类的创建延迟到子类, 从而封装了客户端将得到哪些具体类的信息, 并且影藏了这些类的实例是如何被创建和放在一起的。
- 2、对象的创建模式 对象的创建模式描述的是把对象的创建过程动态地委派给另外一个对象, 从而动态地决定客户端将得到哪些具体的类的实例, 以及这些类的实例是如何被创建和组合在一起的。

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

创建模式主要包括: 简单工厂模式, 工厂方法模式, 抽象工厂模式, 单例模式, 多例模式, 建造模式, 原始模式。

二、结构模式

结构模式描述如何将类或对象结合在一起形成更大的结构, 结构模式也包括类的结构模式和对象的结构模式。

1、类的结构模式 类的结构模式使用继承把类、接口等组合在一起, 以形成更大的结构。当一个类从父类继承并实现某接口时, 这个新的类就把父类的结构和接口的结构结合起来。类的结构模式是静态的, 一个类的结构模式的经典列子就是适配器模式。

2、对象的结构模式 对象的结构模式描述怎么把各种不同的类型的对象组合在一起, 以实现新的功能的方法。对象的结构模式是动态的。

结构模式主要包括: 适配器模式, 缺省适配器模式, 合成模式, 装饰模式, 代理模式, 享元模式, 门面模式, 桥模式。

三、行为模式

行为模式是对在不同的对象之间划分责任和算法的抽象化。行为模式不仅仅是关于类和对象的, 而且是关于它们之间相互作用的。

1、类的行为模式 类的行为模式使用继承关系在几个类之间分配行为。

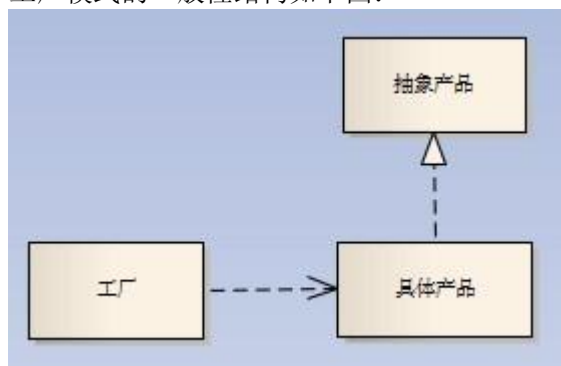
2、对象的行为模式 对象的行为模式是使用对象聚合类分配行为的。

行为模式主要包括: 不变模式, 策略模式, 模板方法模式, 观察者模式, 迭代子模式, 责任链模式, 命令模式, 备忘录模式, 状态模式, 访问者模式, 解释器模式, 调停者模式。

四、简单工厂模式

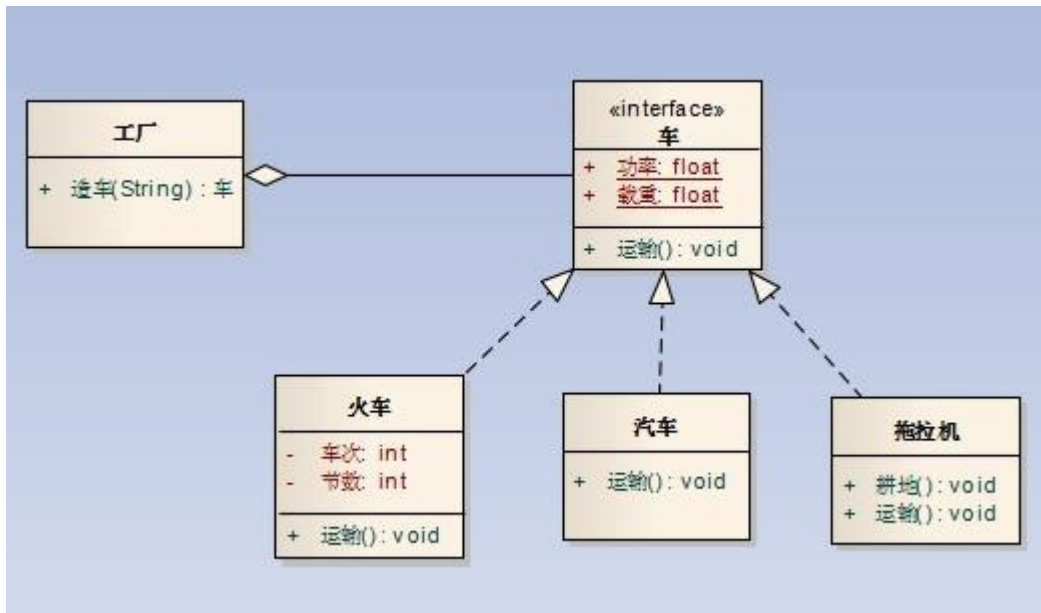
从这节开始学习设计模式, 首先学习创建模式, 其中工厂模式是创建模式里面最常见也常用的一种, 工厂模式又分简单工厂模式(Simple Factory), 工厂方法模式(Factory Method)和抽象工厂模式(Abstractor Factory), 这里先学习最简单的也就是简单工厂模式。

简单工厂模式 (Simple Factory) 也称静态工厂方法模式, 是工厂方法模式的特殊实现。简单工厂模式的一般性结构如下图:



Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

简单工厂模式就是由一个工厂类根据传入的参量决定创建出哪一种产品类型的实例,下面我们拿实例来介绍简单工厂模式。如下图,抽象类型车包括子类火车,汽车,拖拉机。



工厂根据传入的参数来创建具体车的类型。上图中无法形象地表示抽象类所以用接口代替了。

java 代码如下:

Java 代码

```
1 package com.pattern.SimpleFactory;
2 /**
3  *
4  * 【描述】: 工厂类
5  * 【作者】: alaric
6  * 【时间】: May 20, 2012
7  * 【文件】: com.pattern.SimpleFactoryFactory.java
8  *
9  */
10 public class Factory {
11 /**
12 *
13 * 【描述】: 创建车的实例 这个类里面的判断代码在实际应用中多配置成 map, 如果用
spring 则可以配置在 bean 的 xml 内
14 * 【作者】: alaric
15 * 【时间】: May 20, 2012
16 * @throws TypeErrorException
17 *
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
18 */
19     public TrafficMachine creator(String type) throws TypeErrorException{
20         if(type.equals("Automobile")){
21             return new Automobile();
22         }else if (type.equals("Tractor")) {
23             return new Tractor();
24         }else if (type.equals("Train")) {
25             return new Train();
26         }else{
27             throw new TypeErrorException("not find "+type);
28         }
29     }
30 }
31 package com.pattern.SimpleFactory;
32 /**
33  *
34  * 【描述】: 汽车类
35  * 【作者】: alaric
36  * 【时间】: May 20, 2012
37  * 【文件】: com.pattern.SimpleFactoryAutomobile.java
38  *
39  */
40 public class Automobile extends TrafficMachine{
41     @Override
42     public void traffic() {
43         // TODO Auto-generated method stub
44     }
45 }
46 package com.pattern.SimpleFactory;
47 /**
48  *
49  * 【描述】: 拖拉机
50  * 【作者】: alaric
51  * 【时间】: May 20, 2012
52  * 【文件】: com.pattern.SimpleFactoryTractor.java
53  *
54  */
55 public class Tractor extends TrafficMachine{
56     @Override
57     public void traffic() {
58         // TODO Auto-generated method stub
59     }
60 /**
61  *
62  * 【描述】: 耕地
63  * 【作者】: alaric
64  * 【时间】: May 20, 2012
```

```
65 *
66 */
67 public void plough() {
68 }
69 }
70 package com.pattern.SimpleFactory;
71 /**
72 *
73 * 【描述】: 火车
74 * 【作者】: alaric
75 * 【时间】: May 20, 2012
76 * 【文件】: com.pattern.SimpleFactoryTrain.java
77 *
78 */
79 public class Train extends TrafficMachine {
80 private int nodeNum; //节数
81 private int trainNum; //车次
82 @Override
83 public void traffic() {
84 // TODO Auto-generated method stub
85 }
86 public int getNodeNum() {
87 return nodeNum;
88 }
89 public void setNodeNum(int nodeNum) {
90 this.nodeNum = nodeNum;
91 }
92 public int getTrainNum() {
93 return trainNum;
94 }
95 public void setTrainNum(int trainNum) {
96 this.trainNum = trainNum;
97 }
98 }
99 package com.pattern.SimpleFactory;
100 /**
101 *
102 * 【描述】: 抽象类 车
103 * 【作者】: alaric
104 * 【时间】: May 20, 2012
105 * 【文件】: com.pattern.SimpleFactoryMachine.java
106 *
107 */
108 public abstract class TrafficMachine {
109 public float power ;
110 public float load ;
111 public abstract void traffic();
```

```
112 }
113 package com.pattern.SimpleFactory;
114 /**
115 *
116 * 【描述】: 类型异常类
117 * 【作者】: alaric
118 * 【时间】: May 20, 2012
119 * 【文件】: com.pattern.SimpleFactoryTypeErrorException.java
120 *
121 */
122 public class TypeErrorException extends Exception{
123 /**
124 *
125 */
126 private static final long serialVersionUID = 562037380358960152L;
127 public TypeErrorException(String message) {
128 super(message);
129 // TODO Auto-generated constructor stub
130 }
131 }
```

通过以上分析及其代码列举可知,简单工厂类的构造有三种角色,它们分别是工厂角色,抽象产品角色和具体产品角色。工厂类的创建方法根据传入的参数来判断实例化那个具体的产品实例。

工厂类角色:这个角色是工厂方法模式的核心,含有与应用紧密相连的商业逻辑。工厂类在客户端的直接调用下创建产品对象,它往往由一个具体的 java 类来实现。

抽象产品角色:担当这个角色的是一个 java 接口或者 java 抽象类来实现。往往是工厂产生具体类的父类。

具体产品角色:工厂方法模式所创建的任何对象都是这个角色的实例,具体产品往往就是一个具体的 java 类来承担。

简单工厂的优缺点:

1、优点是因为客户端可以直接消费产品,而不关心具体产品的实现,免除了客户端直接创建产品对象的责任,简单工厂模式就是通过这种方法实现了对责任的分割。

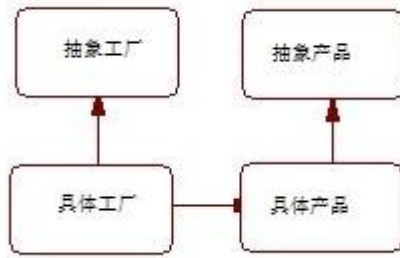
2、缺点是简单工厂在当产品多层次结构复杂时工厂只能依靠自己,这样就形成了一个万能类,如果这个工厂不能工作了,所有的创建都将不能实现,而且当产品类别多结构复杂的情况下,把所有创建放进一个工厂来,是的后期程序的扩展较为困难。这个困难将在下节(工厂方法)进行讲述。

五、工厂方法模式

工厂方法模式(Factory Method)又称虚拟构造子模式,可以说是简单工厂的抽象,也可以理解为简单工厂是退化了的工厂方法模式,其表现在简单工厂丧失了工厂方法的多态性。我们前一节中提到当产品结构变的复杂的时候,简单工厂就变的难以应付,如果增加一种产品,核心工厂

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

类必须改动,使得整个工厂的可扩展性变得很差,对开闭原则支持不够。工厂方法模式克服了这些缺点,它定义一个创建产品对象的工厂接口,将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建,这样核心类成为一个抽象工厂角色,仅负责具体工厂子类必须实现的接口,这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。一般结构图如下:



工厂方法模式对简单工厂模式进行了抽象。有一个抽象的 **Factory** 类(可以是抽象类和接口),这个类将不再负责具体的产品生产,而是只制定一些规范,具体的生产工作由其子类去完成。在这个模式中,工厂类和产品类往往可以依次对应。即一个抽象工厂对应一个抽象产品,一个具体工厂对应一个具体产品,这个具体的工厂就负责生产对应的产品。

工厂方法模式有如下角色:

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

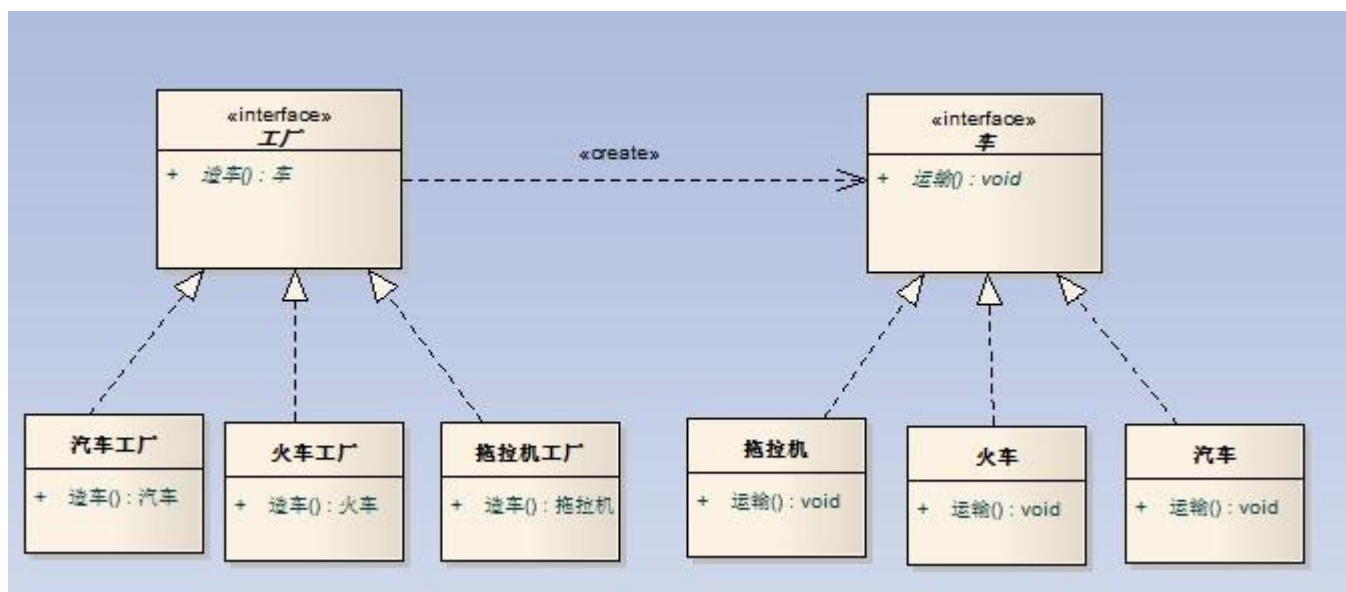
具体工厂(Concrete Creator)角色: 这是实现抽象工厂接口的具体工厂类,包含与应用程序密切相关的逻辑,并且受到应用程序调用以创建产品对象。在上图中有两个这样的角色:

BulbCreator 与 TubeCreator。

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

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

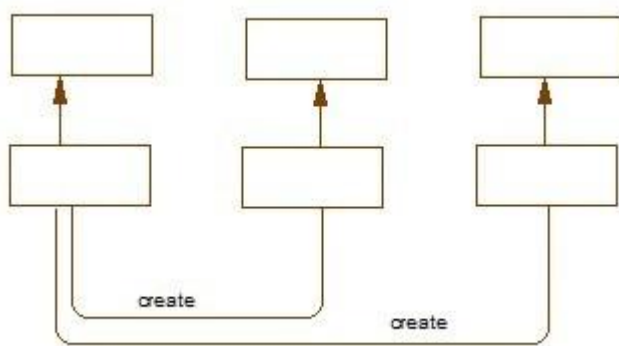
实例: 延续上节中例子来讲,上节中我们提到造车工厂会造拖拉机,汽车,火车,在一个工厂里明显不能完成,在现实世界中,一定是有自己独立的工厂来做。因为我们知道拖拉机,汽车,火车有很多共性也有很大差异,共性还是车,肯定都是重工生产,需要钢材,车床加工,都需要动力,都有座椅,车灯等等,那差异就多了,动力不同,火车可能是电动力,汽车是汽油,拖拉机是柴油等等。我们利用工厂方法来抽象这个造车工厂的模型如下:



通过以上模型可以看出,工厂方法模式是把原来简单工厂里创建对象的过程延迟到了具体实现的子类工厂,这时的工厂从一个类变成了一个接口类型。

六、抽象工厂模式

前面我们介绍了简单工厂,工厂方法模式,这节来看看抽象工厂模式,抽象工厂模式 (Abstract Factory) 是工厂方法里面最为抽象和最具一般性的形态,是指当有多个抽象角色时,使用的一种工厂模式。抽象工厂模式可以向客户端提供一个接口,使客户端在不必指定产品的具体的情况下,创建多个产品族中的产品对象。抽象工厂模式和工厂方法模式的最大区别在于,工厂方法模式针对的是一个产品等级结构;而抽象工厂模式则需要面对多个产品族,从而使得产品具有二维性质。抽象工厂模式的一般示意类图如下:



下面我们先看抽象工厂的角色都有哪些:

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

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

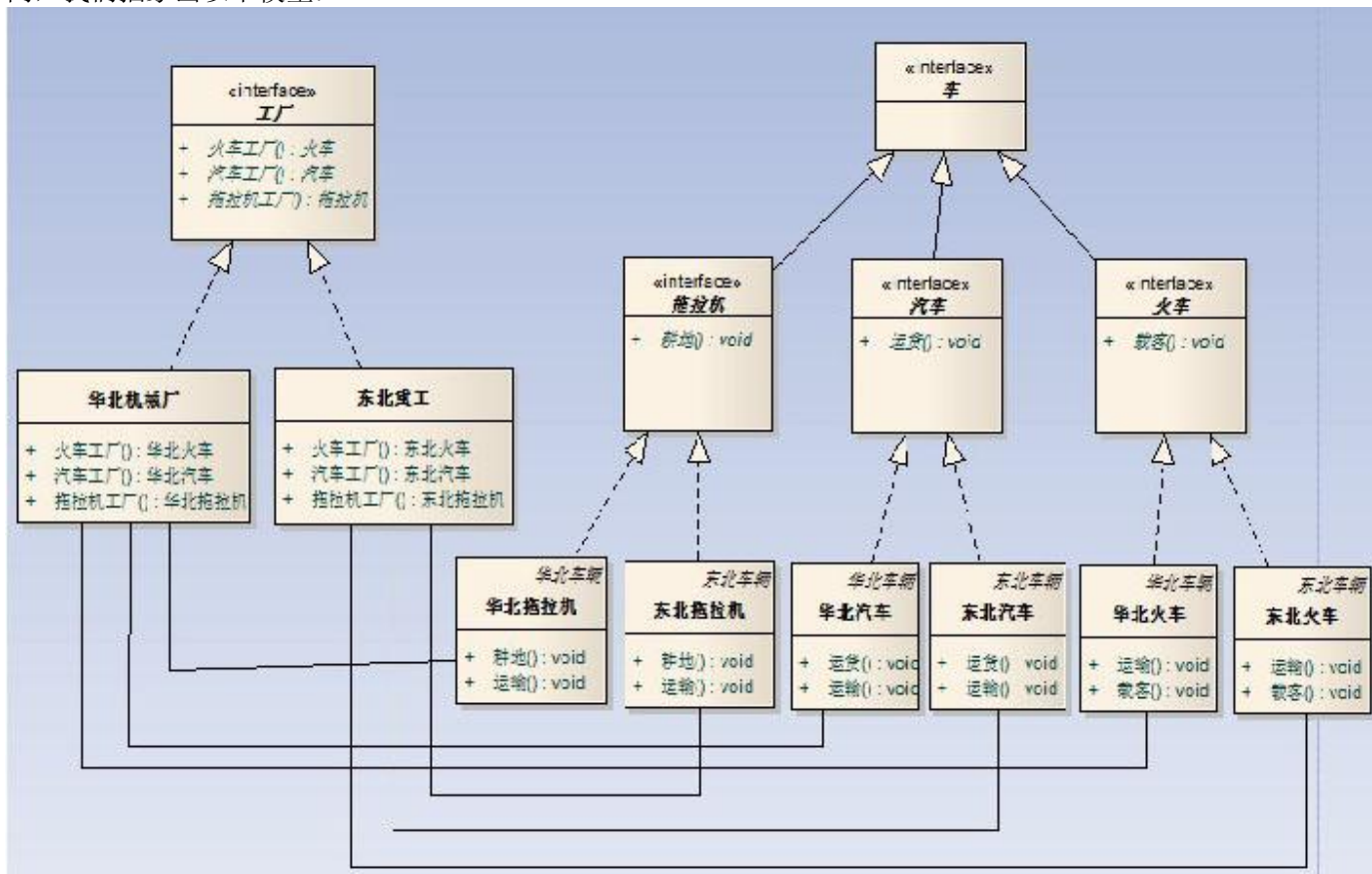
具体工厂(Concrete Creator)角色: 这是实现抽象工厂接口的具体工厂类, 包含与应用程序密切
相关的逻辑, 并且受到应用程序调用以创建产品对象。

抽象产品(Product)角色: 工厂方法模式所创建的对象超类型, 也就是产品对象的共同父类或
共同拥有的接口。

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

如果你很留心, 你就发现抽象工厂的角色和工厂方法的角色一样, 其实抽象工厂就是在工厂方
法的基础上进一步推广。

下面我们来举实例说明, 我们还是延续车的例子, 我们说我们原有的造车厂扩建, 分东北重工
厂和华北机械厂, 这两个厂都可以造拖拉机, 汽车, 火车, 但是他们在工艺和品牌上都有所不
同, 我们抽象出以下模型:



通过上图, 我们可以看出, 我们系统模型中有两个产品族, 一个产品族是东北重工厂产出的

所有产品，另一个产品族是华北机械厂生产出的所有产品。我们也可以看出有多少个实现工厂就有多少个产品族，在工厂角色中有多少工厂方法在同一个产品族类就有多少个具体产品。

七、单例模式

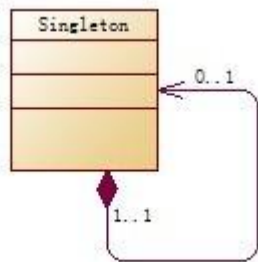
设计模式的创建模式中前面说了工厂模式，这里我们继续来讨论设计模式中另一个创建模式——单例模式。

单例模式（Singleton）是指确保一个类有且仅有一个实例，而且自行实例化并向整个系统提供这个实例。这个类我们也称它为单例类。单例模式的使用在现实世界里很多，比如我们常见的打印机打印的作业队列，一个没打印完，那么只有在队列等待；windows 回收站，windows 视窗里回收站有且只有一个实例。

单例模式的目的是有且只提供一个实例，所以它有以下几个特点：

- 1、单例类只能有一个实例；
- 2、单例类必须自己创建自己惟一的实例；
- 3、单例类必须给所有其他对象提供这一实例。

单例模式的一般结构如下：



上图可以看出，单例类自己提供一个实例给自己。

由于 java 语言的特点在单例的实现上有不同的做法，主要体现在单例类如何实例化自己上。基于上面三个特点我们可以有两种创建单例类实例的方法，第一个是提前创建好，用的时候直接使用；第二种是等到使用的时候再创建实例，业界称第一种为饿汉式，后者成为懒汉式。

饿汉式单例设计模式

Java 代码

```
1 package com.pattern.singleton;
2 /**
3  *
4  * 【描述】: 饿汉式单例模式
5  * 【作者】: alaric
6  * 【时间】: Jul 8, 2012
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
7  *【文件】: com.pattern.singleton.HungrySingleton.java
8  *
9  */
10 publicclass HungrySingleton {
11
12  //创建实例
13  private static final HungrySingleton singleton =new HungrySingleton();
14
15  //私有构造子
16  private HungrySingleton() {}
17
18  //静态工厂方法
19  public static HungrySingleton getInstance() {
20  return singleton;
21  }
22
23 }
```

懒汉式单例设计模式

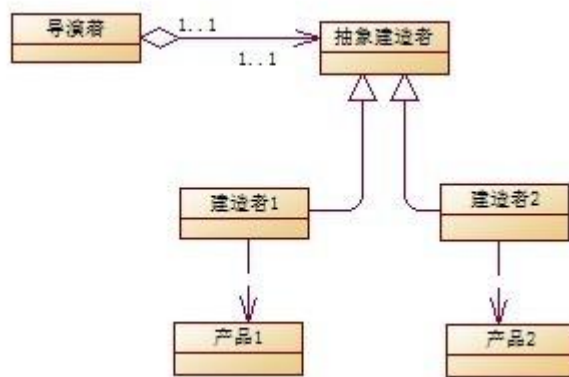
Java 代码   

```
24 packagecom.pattern.singleton;
25 /**
26  *
27  *【描述】: 懒汉式单例模式
28  *【作者】: alaric
29  *【时间】: Jul 8, 2012
30  *【文件】: com.pattern.singleton.LazySingleton.java
31  *
32  */
33 public class LazySingleton {
34
35  //创建实例
36  private static LazySingleton singleton =null;
37
38  //私有构造子
39  private LazySingleton() {}
40
41  //静态工厂方法
42  synchronized public LazySingleton getInstance() {
43  //如果为空就 new 一个实例
44  if(singleton==null){
45      singleton =new LazySingleton();
46  }
47  return singleton;
48  }
49 }
```

通过上面代码,可以看出他们之间的区别,相对而言单例设计模式比较简单,我们只要记住它的特点就可以简单掌握了。

八、建造模式

建造模式 (Builder) 是对象的创建模式,建造模式可以将一个产品的内部表象与产品的生产过程分割开来,从而可以是建造过程生成具有不同内部表象的产品对象。一个产品常有不同的组成成分作为产品的零件,这些零件有可能是对象,也有可能不是对象,通常我们称作内部表象,不同的产品可以有不同的内部表象,也就是不同的零件。使用建造模式可以使客户端不需要知道所生成的产品有那些零件,每个产品对应的零件彼此有何不同,是怎么建造出来的,以及怎样组成产品的。建造模式的简略图如下图所示:



建造模式的角色:

抽象建造者 (Builder) 角色: 给出一个抽象接口,用来规范产品对象各个组成成分的建造,跟商业逻辑无关,但是一般而言,有多少方法产品就有几部分组成。

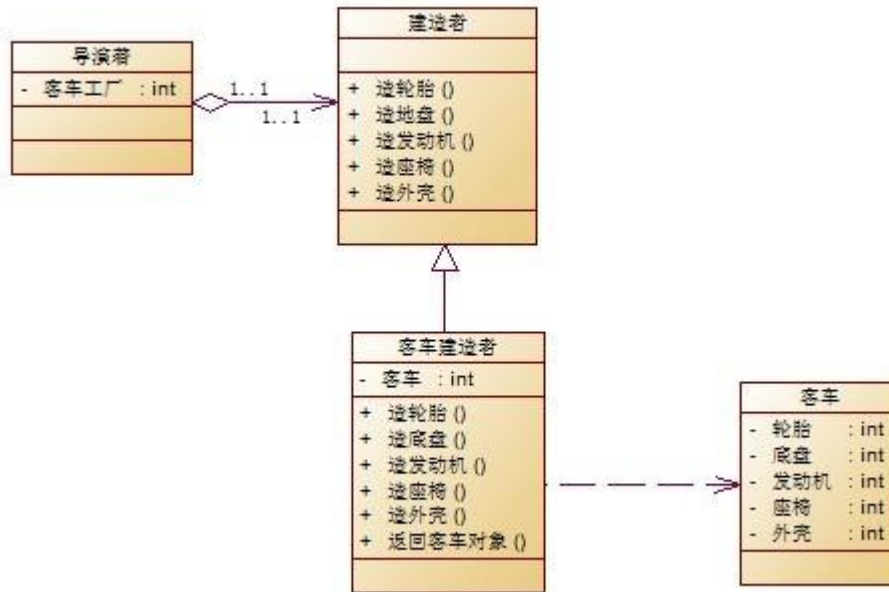
具体建造者 (Concrete Builder) 角色: 担任这个角色的是与应用程序紧密相关的一些类,它们在应用程序调用下创建产品的实例包括创建产品各个零件和产品本身。

导演者 (Director) 角色: 担任这个角色的类调用具体建造者角色创建产品对象。导演者角色并没有产品类的具体知识,真正拥有产品类的具体知识是具体建造者。

产品 (Product) 角色: 产品便是建造中的复杂对象,一般来说产品对象有多个部分组成,在一个体统中会有多余一个的产品类。

实例:

我们还举车的例子,现在要生产一辆客车,客车包括各个部分,如轮胎,底盘,发动机,座椅,外壳。建立以下模型:



上面的模型代码表示如下:

1、导演着:

Java 代码

```
1 package com.pattern.builder;
2 /**
3  *
4  * 【描述】: 导演者
5  * 【作者】: alaric
6  * 【时间】: Jul 14, 2012
7  * 【文件】: com.pattern.builder.Director.java
8  *
9  */
10 public class Director {
11
12     private Builder builder;
13     public TrafficMachine construct()
14     {
15         builder = new CoachBuilder();
16         builder.buildTire(); // 创建轮胎
17         builder.buildChassis(); // 创建底盘
18         builder.buildEngine(); // 创建发动机
19         builder.buildSeat(); // 创建座椅
20         builder.buildShell(); // 创建外壳
21     }
22     return builder.retrieveResult();
23 }
```

```
23 }  
24  
25 }
```

2、抽象建造者

Java 代码   

```
26 package com.pattern.builder;  
27 /**  
28  *  
29  * 【描述】: 抽象建造者  
30  * 【作者】: alaric  
31  * 【时间】: Jul 14, 2012  
32  * 【文件】: com.pattern.builder.Builder.java  
33  *  
34  */  
35 public interface Builder {  
36 /**  
37  *  
38  * 【描述】: 建造轮胎  
39  * 【作者】: alaric  
40  * 【时间】: Jul 14, 2012  
41  *  
42  */  
43 public void buildTire();  
44  
45 /**  
46  *  
47  * 【描述】: 建造底盘  
48  * 【作者】: alaric  
49  * 【时间】: Jul 14, 2012  
50  *  
51  */  
52 public void buildChassis();  
53  
54 /**  
55  *  
56  * 【描述】: 建造引擎  
57  * 【作者】: alaric  
58  * 【时间】: Jul 14, 2012  
59  *  
60  */  
61 public void buildEngine();  
62  
63 /**  
64  *  
65  * 【描述】: 建造座椅
```

```
66      * 【作者】: alaric
67      * 【时间】: Jul 14, 2012
68      *
69      */
70 public void buildSeat();
71
72 /**
73      *
74      * 【描述】: 建造外壳
75      * 【作者】: alaric
76      * 【时间】: Jul 14, 2012
77      *
78      */
79 public void buildShell();
80
81 /**
82      *
83      * 【描述】: 返回产品
84      * 【作者】: alaric
85      * 【时间】: Jul 14, 2012
86      *
87      */
88 public TrafficMachine retrieveResult();
89
90 }
```

3、实现建造者

Java 代码   

```
91 package com.pattern.builder;
92 /**
93      *
94      * 【描述】: 客车建造者
95      * 【作者】: alaric
96      * 【时间】: Jul 14, 2012
97      * 【文件】: com.pattern.builder.CoachBuilder.java
98      *
99      */
100 public class CoachBuilder implements Builder {
101
102     private Coach coach = new Coach();
103
104     @Override
105     public void buildTire() {
106         // 此处根据实际业务写相关逻辑
107         coach.setTire(new Tire());
108     }
109 }
```



```
109
110     @Override
111 public void buildChassis() {
112 // 此处根据实际业务写相关逻辑
113     coach.setChassis(new Chassis());
114 }
115
116     @Override
117 public void buildEngine() {
118 // 此处根据实际业务写相关逻辑
119     coach.setEngine(new Engine());
120 }
121
122     @Override
123 public void buildSeat() {
124 // 此处根据实际业务写相关逻辑
125     coach.setSeat(new Seat());
126 }
127
128     @Override
129 public void buildShell() {
130 // 此处根据实际业务写相关逻辑
131     coach.setShell(new Shell());
132 }
133
134 public TrafficMachine retrieveResult() {
135 // 此处根据实际业务写相关逻辑
136 return coach;
137 }
138
139 }
```

4、产品接口（可以没有）

Java 代码   

```
140 package com.pattern.builder;
141 /**
142  *
143  * 【描述】: 抽象的车
144  * 【作者】: alaric
145  * 【时间】: Jul 14, 2012
146  * 【文件】: com.pattern.builder.TrafficMachine.java
147  *
148  */
149 public interface TrafficMachine {
150
151 }
```

5、客车类—实现产品类

Java 代码   

```
152 package com.pattern.builder;
153 /**
154  *
155  * 【描述】: 客车
156  * 【作者】: alaric
157  * 【时间】: Jul 14, 2012
158  * 【文件】: com.pattern.builder.Coach.java
159  *
160  */
161 public class Coach implements TrafficMachine {
162
163
164     private Chassis chassis;
165     private Tire tire;
166     private Engine engine;
167     private Seat seat;
168     private Shell shell;
169
170     public Chassis getChassis() {
171         return chassis;
172     }
173     public void setChassis(Chassis chassis) {
174         this.chassis = chassis;
175     }
176     public Tire getTire() {
177         return tire;
178     }
179     public void setTire(Tire tire) {
180         this.tire = tire;
181     }
182     public Engine getEngine() {
183         return engine;
184     }
185     public void setEngine(Engine engine) {
186         this.engine = engine;
187     }
188     public Seat getSeat() {
189         return seat;
190     }
191     public void setSeat(Seat seat) {
192         this.seat = seat;
193     }
194     public Shell getShell() {
```

```
195 return shell;
196 }
197 public void setShell(Shell shell) {
198     this.shell = shell;
199 }
200
201 }
```

6、部件类

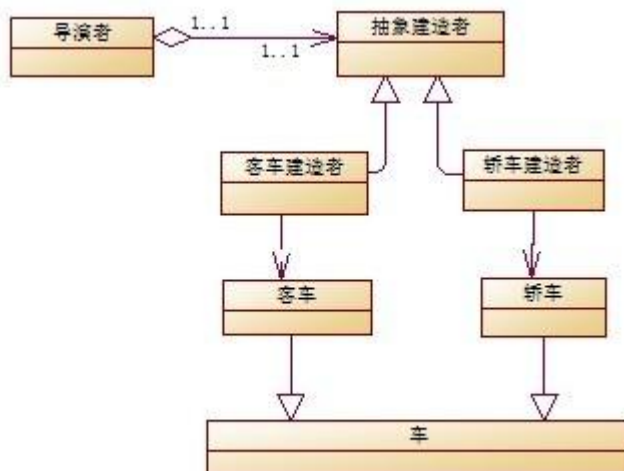
Java 代码   

```
202 package com.pattern.builder;
203 /**
204  *
205  * 【描述】: 轮胎 —— 不错具体的实现
206  * 【作者】: alaric
207  * 【时间】: Jul 14, 2012
208  * 【文件】: com.pattern.builder.Tire.java
209  *
210  */
211 public class Tire {
212
213 }
214 package com.pattern.builder;
215 /**
216  *
217  * 【描述】: 底盘
218  * 【作者】: alaric
219  * 【时间】: Jul 14, 2012
220  * 【文件】: com.pattern.builder.Chassis.java
221  *
222  */
223 public class Chassis {
224
225 }
226 package com.pattern.builder;
227 /**
228  *
229  * 【描述】: 发动机
230  * 【作者】: alaric
231  * 【时间】: Jul 14, 2012
232  * 【文件】: com.pattern.builder.Engine.java
233  *
234  */
235 public class Engine {
236
237 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
238 package com.pattern.builder;
239 /**
240  *
241  * 【描述】: 座椅
242  * 【作者】: alaric
243  * 【时间】: Jul 14, 2012
244  * 【文件】: com.pattern.builder.Seat.java
245  *
246  */
247 public class Seat {
248
249 }
250 package com.pattern.builder;
251 /**
252  *
253  * 【描述】: 外壳
254  * 【作者】: alaric
255  * 【时间】: Jul 14, 2012
256  * 【文件】: com.pattern.builder.Shell.java
257  *
258  */
259 public class Shell {
260
261 }
```

通过上面例子我们可以更明确创建模式的各个角色职能和作用, 在现实开发中可能会省略一些, 要根据实际业务来决定, 以上是一个产品的例子, 事实上多个产品的更为常见。我们将上面的模型扩展, 加入多产品, 模型如下:



多产品的实现和单一产品的实现基本一样, 这里就不赘述了。当产品为多产品的时候整体结构有点像抽象工厂模式, 但是我们只要把握重点, 抽象工厂是在客户端在不指定产品的具体的

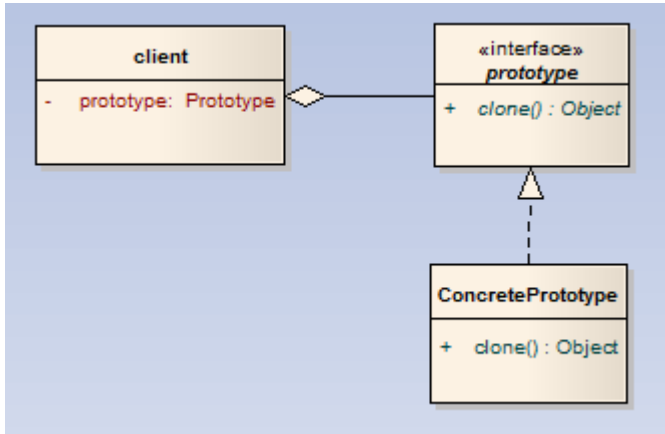
Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

情况下, 创建多个产品族中的产品对象; 而建造模式重点在于影藏和封装复杂产品的内部建造过程和结构, 侧重点和用意完全不同, 这点要掌握了区分就很简单了。

九、原型模式

原型模式 (prototype) 它是指通过给定一个原型对象来指明所要创建的对象类型, 然后复制这个原型对象的办法创建出同类型的对象。原型模式也属于创建模式。

我们先来看一下原型模式的模型:



原型模型涉及到三个角色:

客户角色 (client): 客户端提出创建对象的请求;

抽象原型 (prototype): 这个往往由接口或者抽象类来担任, 给出具体原型类的接口;

具体原型 (Concrete prototype): 实现抽象原型, 是被复制的对象;

模拟代码如下:

Java 代码   

```
1 package prototype;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-7-18下午10:40:49
6  *描述: 抽象接口
7  */
8 public interface Prototype extends Cloneable {
9
10     public Object clone();
11
12 }
```

Java 代码   

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
13 package prototype;
14 /**
15  *
16  *作者: alaric
17  *时间: 2013-7-18下午10:41:39
18  *描述: 实现接口
19  */
20 public class ConcretePrototype implements Prototype {
21
22     @Override
23     public Object clone() {
24         try {
25             return super.clone();
26         } catch (CloneNotSupportedException e) {
27             // TODO Auto-generated catch block
28             e.printStackTrace();
29             return null;
30         }
31     }
32 }
33
34 }
```

Java 代码   

```
35 package prototype;
36 /**
37  *
38  *作者: alaric
39  *时间: 2013-7-18下午10:41:14
40  *描述: 客户端
41  */
42 public class Client {
43     private Prototype prototype;
44     /**
45      * @param args
46      */
47     public static void main(String[] args) {
48         Client c = new Client();
49         c.prototype = c.getNewPrototype(new ConcretePrototype());
50     }
51 }
52 /**
53  *
54  * @param prototype
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
55     * @return
56     */
57     public Prototype getNewPrototype(Prototype prototype) {
58         return (Prototype) prototype.clone();
59     }
60 }
```

以上代码简单描述了原型模式的实现,说到这里估计很多人要跳了,因为说到原型模式不能不说的的问题是 java 的深拷贝和浅拷贝,那下面我们就来讨论下深拷贝和浅拷贝。

浅拷贝:是指拷贝引用,实际内容并没有复制,改变后者等于改变前者。

深拷贝:拷贝出来的东西和被拷贝者完全独立,相互没有影响。

引用一哥们举的例子(博客地址忘记了)

有一个人叫张三,人们给他取个别名叫李四,不管张三还是李四都是一个人,张三胳膊疼,李四也是一个样的不爽。这个就是浅拷贝,只是个别名而已。

同样还是有一个人叫张三,通过人体克隆技术(如果法律允许)得到一个李四,这个李四和被克隆的张三完全是两个人,张三就是少个胳膊,李四也不会感到疼痛。这个就是深拷贝。

java 语言提供 Cloneable 接口,在运行时通知虚拟机可以安全的在这个类上使用 clone () 方法,通过这个方法可以复制一个对象,但是 Object 并没有实现这个接口,所以在拷贝是必须实现此标识接口,否则会抛出 CloneNotSupportedException。

但是 clone () 方法出来的默认都是浅拷贝,如果要深拷贝,那么可以考虑自己编写 clone 方法,但是深度很难控制,编写这个 clone 方法也不是最佳方案,还有个比较好的方案就是串行化来实现,代码如下:

Java 代码   

```
61 public Object deepClone() {
62     ByteArrayOutputStream bos = new ByteArrayOutputStream();
63     ObjectOutputStream oos = new ObjectOutputStream(bos);
64     oos.writeObject(this);
65     ByteArrayInputStream bis = new ByteArrayInputStream(baos.toByteArray());
66     ObjectInputStream ois = new ObjectInputStream(bis);
67     return ois.readObject();
68 }
```

这样就可以实现深拷贝,前提是对象实现 java.io.Serializable 接口。

注意:除了基本数据类型外,其他对象默认拷贝都是浅拷贝,String 类型是个例外,虽然是基本类型,但是也是浅拷贝,这个跟它实际在 java 内存存储情况有关。超出了设计模式讨论范围,大家可自行查看相关资料。

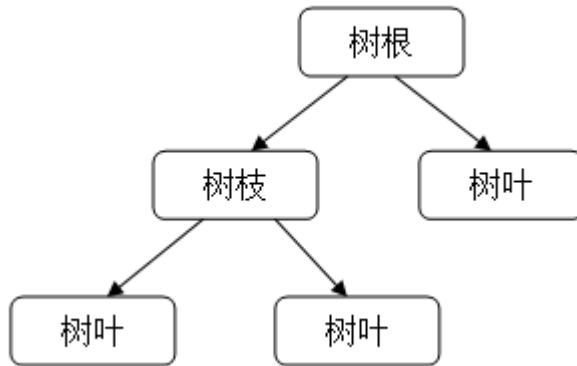
十、组合模式

这节开始学习结构模式,结构模式包括:组合模式、门面模式、适配器模式、代理模式、装饰

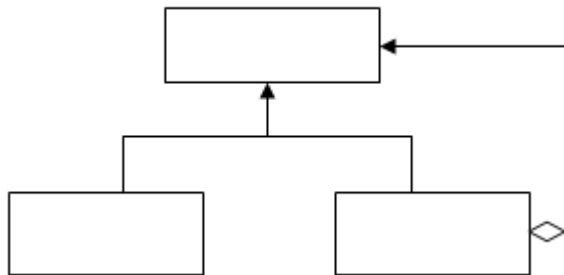
模式、桥模式、享元模式。从组合模式开始学习。

组合模式 (Composite) 就是把部分和整体的关系用树形的结构来表示,从而使客户端能够把部分对象和组合起来的对象采用同样的方式来看待。

树图结构一般包含一个根节点,若干个树枝和叶子节点。如下图:



可以用一个类图描述树结构的静态结构,把根节点当做树枝节点来描述,同时和叶子节点具有共同的父类:



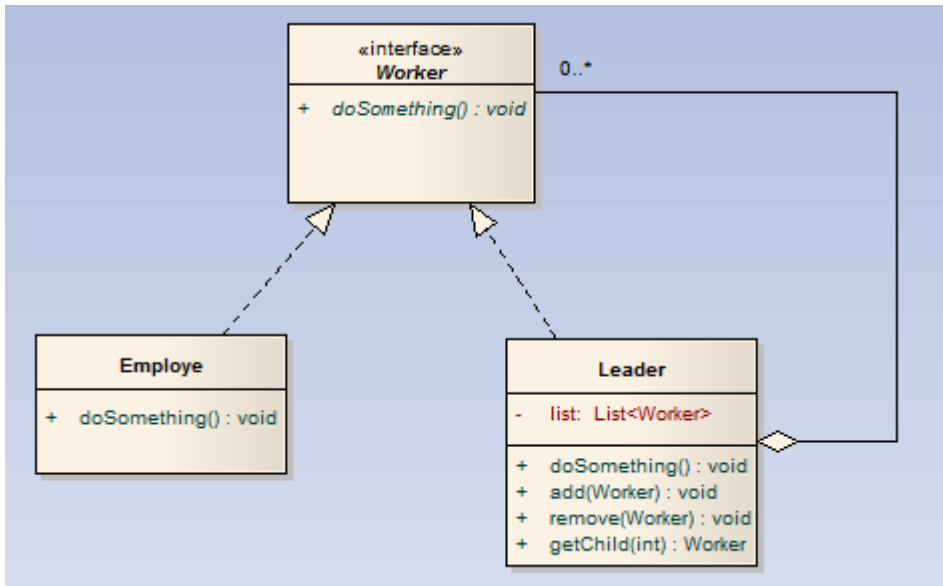
树结构的类图,其实就是组合模式的简略类图,最上面为抽象节点,左下方为叶子节点,右下方为树枝节点,它含有其他的节点。

通过以上结构图可以看出,组合模式有以下角色:

- 1、**抽象构建角色 (component)**: 作为抽象角色,给组合对象的统一接口。
- 2、**树叶构建角色 (leaf)**: 代表组合对象中的树叶对象。
- 3、**树枝构建角色 (composite)**: 参加组合的所有子对象的对象,并给出树枝构建对象的行为。

组合模式在现实中使用很常见,比如文件系统中目录和文件的组成,算式运算,android 里面的 view 和 viewgroup 等控件,淘宝产品分类信息的展示等都是组合模式的例子。还有个故事:从前山里有座庙,庙里有个老和尚,老和尚对象小和尚讲故事说,从前山里有座庙.....退出循环的条件是听厌烦了,或者讲的人讲累了。

这个模式的举例最多的是公司员工的例子。我们这里也以故事雇员为例子来描述:



Java 代码

```
1 package composite;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-7-20下午5:11:41
6  *描述: 员工和领导的统一接口
7  */
8 public interface Worker {
9
10     public void doSomething();
11
12 }
```

Java 代码

```
13 package composite;
14 /**
15  *
16  *作者: alaric
17  *时间: 2013-7-20下午5:59:11
18  *描述: 普通员工类
19  */
20 public class Employee implements Worker {
21
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
22     private String name;
23
24     public Employee(String name) {
25         super();
26         this.name = name;
27     }
28     @Override
29     public void doSomething() {
30         System.out.println(toString());
31     }
32
33
34     @Override
35     public String toString() {
36         // TODO Auto-generated method stub
37         return "我叫"+getName()+"，就一普通员工!";
38     }
39
40     public String getName() {
41         return name;
42     }
43
44     public void setName(String name) {
45         this.name = name;
46     }
47
48 }
```

Java 代码   

```
49 package composite;
50
51 import java.util.Iterator;
52 import java.util.List;
53 import java.util.concurrent.CopyOnWriteArrayList;
54
55 /**
56  *
57  *作者: alaric
58  *时间: 2013-7-20下午5:14:50
59  *描述: 领导类
60  */
61 public class Leader implements Worker {
62     private List<Worker> workers = new CopyOnWriteArrayList<Worker>();
63     private String name;
64 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
65     public Leader(String name) {
66         super();
67         this.name = name;
68     }
69     public void add(Worker worker) {
70         workers.add(worker);
71     }
72
73     public void remove(Worker worker) {
74         workers.remove(worker);
75     }
76
77     public Worker getChild(int i) {
78         return workers.get(i);
79     }
80     @Override
81     public void doSomething() {
82         System.out.println(toString());
83         Iterator<Worker> it = workers.iterator();
84         while(it.hasNext()) {
85             it.next().doSomething();
86         }
87     }
88
89
90
91     @Override
92     public String toString() {
93         // TODO Auto-generated method stub
94         return "我叫"+getName()+"，我是一个领导,有 "+workers.size()+"下属。";
95     }
96     public String getName() {
97         return name;
98     }
99
100    public void setName(String name) {
101        this.name = name;
102    }
103
104 }
```

Java 代码   

```
105 package composite;
106 /**
```

```
107 *
108 *作者: alaric
109 *时间: 2013-7-20下午5:49:37
110 *描述: 测试类
111 */
112 public class Client {
113
114     /**
115     *作者: alaric
116     *时间: 2013-7-20下午5:49:32
117     *描述:
118     */
119     public static void main(String[] args) {
120         // TODO Auto-generated method stub
121         Leader leader1 = new Leader("张三");
122         Leader leader2 = new Leader("李四");
123         Employee employe1 = new Employee("王五");
124         Employee employe2 = new Employee("赵六");
125         Employee employe3 = new Employee("陈七");
126         Employee employe4 = new Employee("徐八");
127         leader1.add(leader2);
128         leader1.add(employe1);
129         leader1.add(employe2);
130         leader2.add(employe3);
131         leader2.add(employe4);
132         leader1.doSomething();
133
134     }
```

运行结果如下:

我叫张三, 我是一个领导,有 3个直接下属。

我叫李四, 我是一个领导,有 2个直接下属。

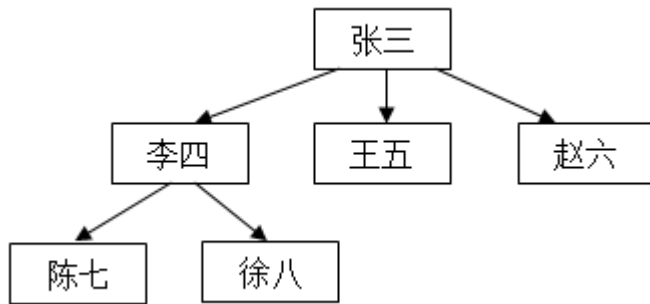
我叫陈七,就一普通员工!

我叫徐八,就一普通员工!

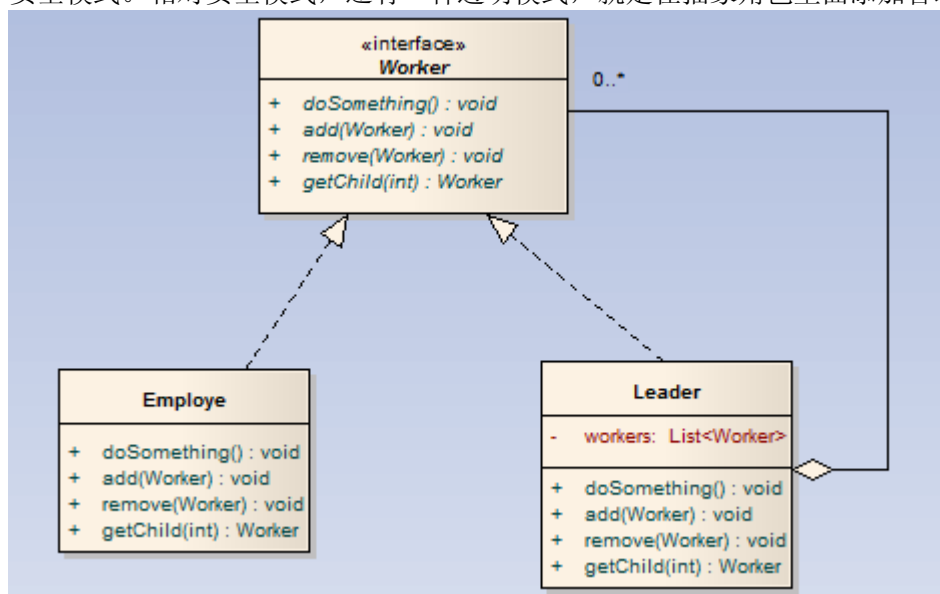
我叫王五,就一普通员工!

我叫赵六,就一普通员工!

上面员工关系的的树形结构如下:



上面例子给出的组合模式抽象角色里,并没有管理子节点的方法,而是在树枝构建角色,这种模式使得叶子构建角色和树枝构建角色有区分,客户端要分别对待树叶构建角色和树枝构建角色,好处是客户端对叶子节点不会调用管理的方法,当调用时,在编译时就会报错,所以也称安全模式。相对安全模式,还有一种透明模式,就是在抽象角色里面添加管理方法,如下图:



这种做法是对客户端来说叶子和树枝都是一致的接口,相对透明,但是叶子节点在调用管理方法是编译时不会报错,运行时才报错,所以不够安全。两种做法均有利弊,使用时要根据具体情况而权衡。

十二、适配器模式

适配器 (adapter) 模式,把一个类的接口变成客户端所期待的另一种接口,从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

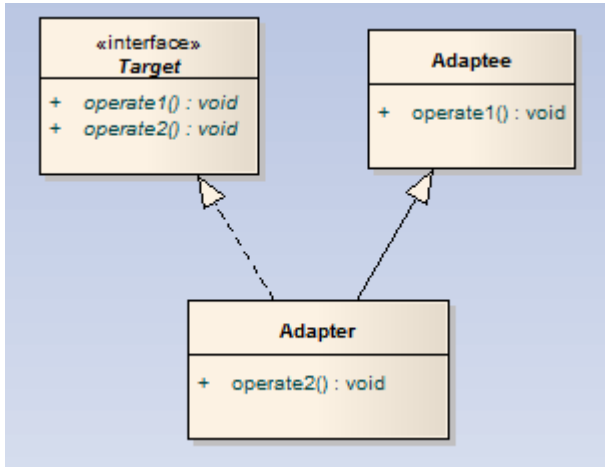
适配器模式有两种形式,**类**的适配器模式和**对象**的适配器模式。我们先看类的适配器模式,类的适配器模式有以下角色:

目标 (Target) 角色:是期待得到的接口类型。这里谈类的适配器模式所以这个不能是类,原因是 java 单继承。

源 (adaptee) 角色: 现有待适配的接口类型。

适配器 (adapter) 角色: 适配器类模式的核心, 这个角色负责把源接口转换成目标角色的接口。

类的适配器模式类图:



通过上图可以看出, 组合对象 **Adapter** 持有源 **Adaptee** 的对象, 利用聚合代替了继承, 在 **Adapter** 里面的代码如下编写:

Java 代码

```
1 package adapter;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-7-21下午6:44:53
6  *描述: 目标类
7  */
8 public interface Target {
9
10     public void operate1();
11
12     public void operate2();
13 }
```

Java 代码

```
14 package adapter;
15 /**
16  *
17  *作者: alaric
18  *时间: 2013-7-21下午6:44:29
19  *描述: 源
20  */
21 public class Adaptee {
22
23     public void operate1() {
```

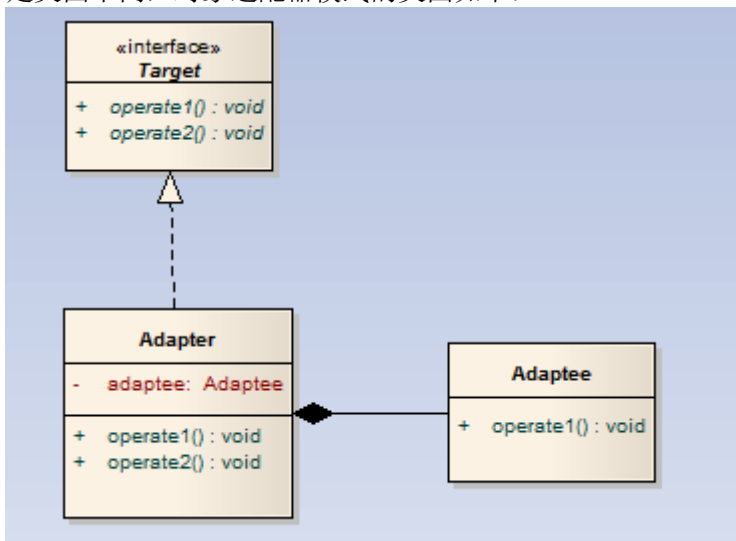
Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
24      //业务逻辑
25  }
26 }
```

Java 代码   

```
27 package adapter;
28 /**
29  *
30  *作者: alaric
31  *时间: 2013-7-21下午6:44:05
32  *描述: 适配器类
33  */
34 public class Adapter extends Adaptee implements Target{
35
36     @Override
37     public void operate2() {
38         // 业务逻辑
39     }
40 }
41
42 }
```

适配器类继承了源类, 实现了目标类在源里没有的接口, 达到了适配转换作用。java 是单继承的语言, 这种类的适配模式往往受到使用环境的限制, 在面向对象设计原则中, 有一条叫做组合/聚合复用原则, 讲的是尽可能使用组合和聚合达到复用的目的而不是继承, 所以一般推荐用对象适配器模式达到目的, 对象适配器的角色和类的适配器模式的角色没什么具体的区别, 只是类图不同, 对象适配器模式的类图如下:



通过上图可以看出, 组合对象 **Adapter** 持有源 **Adaptee** 的对象, 利用聚合代替了继承, 在 **Adapter** 里面的代码如下编写:

Java 代码   

```
43 package adapter;
44 /**
45  *
46  *作者: alaric
47  *时间: 2013-7-21下午6:44:05
48  *描述: 适配器类
49  */
50 public class Adapter2 implements Target{
51
52     private Adaptee adaptee;
53
54     public Adapter2(Adaptee adaptee) {
55         super();
56         this.adaptee = adaptee;
57     }
58
59     @Override
60     public void operate2() {
61         // 业务逻辑
62     }
63
64     @Override
65     public void operate1() {
66         //调用源的方法
67         adaptee.operate1();
68     }
69 }
70
71 }
```

为了不改变原有系统的实现而对目标接口需求的满足而做适配,利用具体的类的适配器模式还是对象的适配器模式,要根据具体的业务场景,如果两种都可以的话最好选择对象的适配器模式,适配器模式使得原本不能在一起工作的类在一起工作成为可能。但是对于变化很大的系统对每个接口都写一个适配器类变的很难维护,这时候应该考虑对原有代码的重构,而不是系统中存在大量的适配器类。

十二、装饰器模式

装饰（Decorator）模式属于设计模式里的结构模式,通过装饰类动态的给一个对象添加一些额外的职责。装饰模式也叫包装（wrapper）模式。装饰模式有如下的角色:

抽象构件（component）角色: 这个角色用来规范被装饰的对象,一般用接口方式给出。

具体构件（concrete component）角色: 被装饰的类。

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

装饰 (decorator) 角色: 持有一个构件对象的实例。并定义一个跟抽象构件一直的接口。

具体 (concrete decorator) 装饰角色: 负责给具体构件添加附加职责的类。在实际使用中多数情况下装饰角色和具体装饰角色可能由一个类来承担。

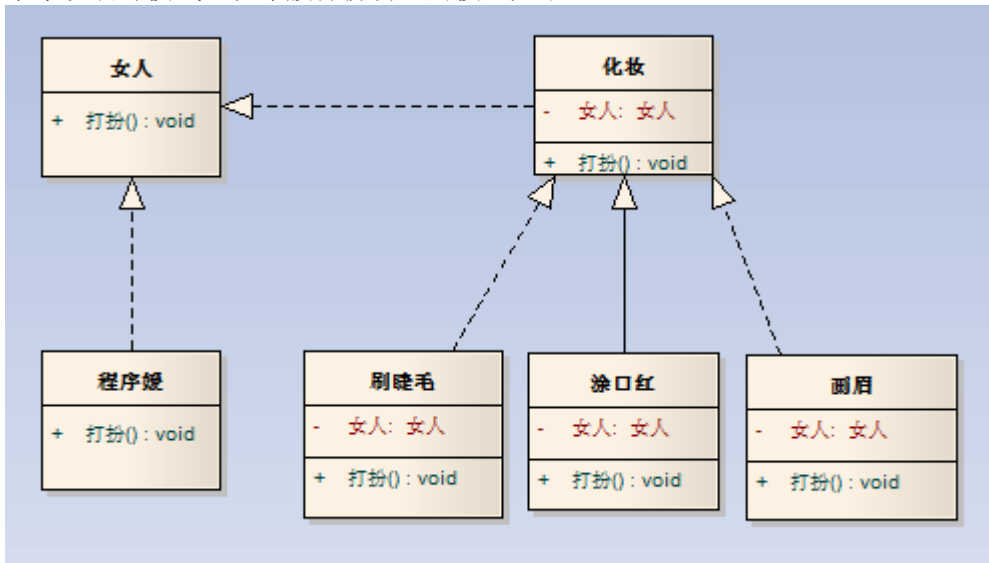
装饰模式的类图如下:

装饰模式的例子在生活中很常见, 比如房子装修, 那房子还是那个房子, 只是在房子墙壁上做粉刷, 贴墙纸, 挂饰品等。java 里面的 io 流, 我们经常如下使用:

Java 代码

```
1  BufferedReader br = new BufferedReader(new InputStreamReader(new
    FileInputStream(file)));
```

这个就是装饰模式的使用。还有句话说人靠衣装, 美靠靓装。这里就举个程序媛打扮自己的例子来说装饰模式。程序媛打扮自己的模型如下



Java 代码

```
2  package decorator;
3
4  public interface 女人 {
5
6      public void 打扮();
7
8  }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

Java 代码   

```
9 package decorator;
10
11 public class 程序媛 implements 女人 {
12
13     @Override
14     public void 打扮() {
15
16         System.out.println("开始我是素颜。");
17
18     }
19
20 }
```

Java 代码   

```
21 package decorator;
22
23 public class 化妆 implements 女人 {
24
25     private 女人 女人;
26
27     @Override
28     public void 打扮() {
29         女人.打扮();
30
31     }
```

Java 代码   

```
32 package decorator;
33
34 public class 画眉 extends 化妆 {
35
36     private 女人 女人;
37
38
39     public 画眉(女人 女人) {
40         super();
41         this.女人 = 女人;
42     }
43
44
45     public void 打扮() {
46         System.out.println("画眉了，漂亮了一些。");
47         女人.打扮();
48     }
49 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
48     }  
49  
50 }
```

Java 代码   

```
51 package decorator;  
52  
53 public class 刷睫毛 extends 化妆 {  
54  
55     private 女人 女人;  
56  
57  
58     public 刷睫毛(女人 女人) {  
59         super();  
60         this.女人 = 女人;  
61     }  
62  
63  
64     public void 打扮() {  
65         System.out.println("刷睫毛了, 更漂亮了一些。");  
66         女人.打扮();  
67     }  
68  
69 }
```

Java 代码   

```
70 package decorator;  
71  
72 public class 涂口红 extends 化妆 {  
73  
74     private 女人 女人;  
75  
76  
77     public 涂口红(女人 女人) {  
78         super();  
79         this.女人 = 女人;  
80         打扮();  
81     }  
82  
83  
84     public void 打扮() {  
85         System.out.println("有了口红, 哇塞, 狐狸精一个!");  
86         女人.打扮();  
87     }  
88  
89 }
```

```
88
89 }
```

Java 代码   

```
90 package decorator;
91
92 public class Client {
93
94     /**
95      *作者: alaric
96      *时间: 2013-7-22下午10:57:13
97      *描述: 测试
98      */
99     public static void main(String[] args) {
100
101         女人 女1 = new 程序媛();
102         new 涂口红(new 刷睫毛(new 画眉(女1)));
103
104     }
105
106 }
```

运行结果:

有了口红, 哇塞, 狐狸精一个!

刷睫毛了, 更漂亮了一些。

画眉了, 漂亮了一些。

开始我是素颜。

通过上面例子我们可以看到, 装饰模式在于在原有的功能上添加新的职责。装饰模式能够提供比继承更灵活的对象扩展能力, 但是也往往由于这种灵活性会是系统调用变的复杂。

十三、代理模式

代理 (proxy) 模式: 指目标对象给定代理对象, 并由代理对象代替真实对象控制客户端对真实对象的访问。

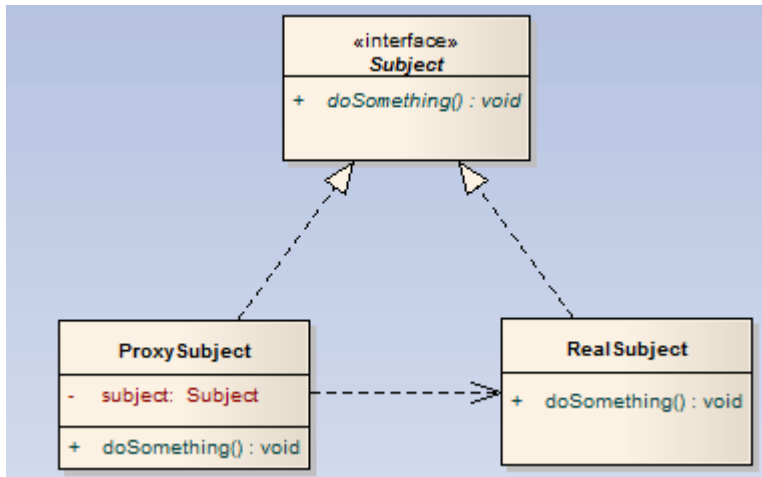
代理模式模式有以下角色:

抽象主题 (subject) 角色: 声明真实主题和代理主题的共同接口。

真实主题 (real subject) 角色: 定义代理对象需要代理的真实对象。

代理主题 (proxy subject) 角色: 代替真实对象来控制对真实对象的访问, 代理对象持有真实对象的应用, 从而可以随时控制客户端对真实对象的访问。

代理模式结构类图:



代理模式在 java 里面很常见，在开源框架里如 spring，mybatis 等里面大量使用。现实生活中也很常见，比如我们访问 facebook 主站，常会选择一些代理，通过代理访问 facebook。代理分静态代理和动态代理，java 对动态代理有很好的支持，提供了 InvocationHandler 接口和 Proxy 类。

java API 对 InvocationHandler 接口和 Proxy 类的介绍:

Proxy 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。创建某一接口 Foo 的代理:

```
InvocationHandler handler = new MyInvocationHandler(...);
Class proxyClass = Proxy.getProxyClass(
    Foo.class.getClassLoader(), new Class[] { Foo.class });
Foo f = (Foo) proxyClass.
    getConstructor(new Class[] { InvocationHandler.class }).
    newInstance(new Object[] { handler });
```

或使用以下更简单的方法:

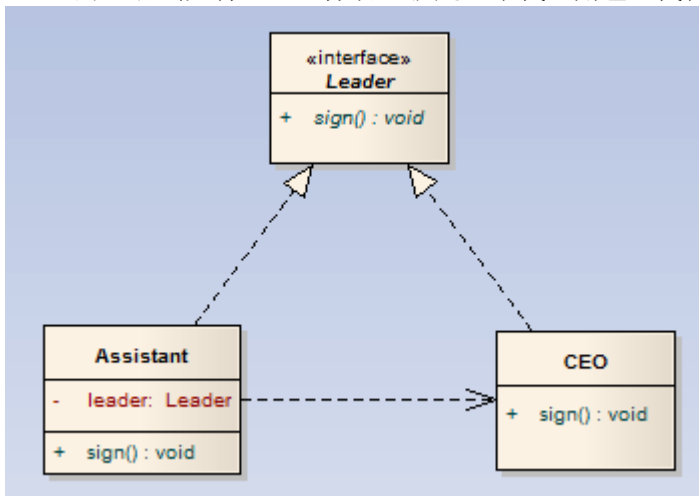
```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
    new Class[] { Foo.class },
    handler);
```

动态代理类（以下简称为代理类）是一个实现在创建类时在运行时指定的接口列表的类，该类具有下面描述的行为。代理接口 是代理类实现的一个接口。代理实例 是代理类的一个实例。每个代理实例都有一个关联的调用处理程序 对象，它可以实现接口 InvocationHandler。

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

通过其中一个代理接口的代理实例上的方法调用将被指派到实例的调用处理程序的 `Invoke` 方法, 并传递代理实例、识别调用方法的 `java.lang.reflect.Method` 对象以及包含参数的 `Object` 类型的数组。调用处理程序以适当的方式处理编码的方法调用, 并且它返回的结果将作为代理实例上方法调用的结果返回。`InvocationHandler` 是代理实例的调用处理程序实现的接口。每个代理实例都具有一个关联的调用处理程序。对代理实例调用方法时, 将对方法调用进行编码并将其指派到它的调用处理程序的 `invoke` 方法。

这里举个好理解的例子: 公司项目部需要 CEO 签署一个文件, 项目负责人会把文件交给 CEO 助理, 助理会收文件, 等到 CEO 回来后递 CEO, CEO 签署后交给助理, 助理收好交给项目负责人。这个过程中项目负责人其实不知道是否真的是 CEO 签署的文件, 有可能是助理打印的 CEO 的签名到文件上。这样助理就是一个代理角色, 代替 CEO 处理事务。静态代理类图如下:



代码如下:

Java 代码

```
1 package proxy;
2
3 /**
4
5  *
6
7  *作者: alaric
8
9  *时间: 2013-7-24下午10:44:12
10
11 *描述: 抽象主题
12
13 */
14
15 public interface Leader {
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
16
17     public void sign();
18
19 }
```

Java 代码

```
20 package proxy;
21 /**
22  *
23  *作者: alaric
24  *时间: 2013-7-24下午10:45:05
25  *描述: ceo 真实主题
26  */
27 public class CEO implements Leader {
28
29     @Override
30     public void sign() {
31         System.out.println("CEO 签文件");
32     }
33
34 }
```

Java 代码

```
35 package proxy;
36 /**
37  *
38  *作者: alaric
39  *时间: 2013-7-24下午10:45:25
40  *描述: 代理主题
41  */
42 public class Assistant implements Leader{
43
44     private Leader leader ;
45
46
47     public Assistant(Leader leader) {
48         super();
49         this.leader = leader;
50     }
51
52
53     @Override
54     public void sign() {
55         System.out.println("递给领导");
56     }
57 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
56         leader.sign();
57         System.out.println("装入袋子, 送出");
58     }
59
60
61 }
```

Java 代码

```
62 package proxy;
63
64 import java.lang.reflect.InvocationHandler;
65 import java.lang.reflect.Method;
66 import java.lang.reflect.Proxy;
67 /**
68  *
69  *作者: alaric
70  *时间: 2013-7-24下午10:46:04
71  *描述: 动态代理的 Handler
72  */
73 public class AssistantHandler implements InvocationHandler {
74
75     /**
76      * 目标对象
77      */
78     private Object targetObject;
79
80
81     /**
82      *
83      *作者: alaric
84      *时间: 2013-7-24下午10:46:59
85      *描述: 创建代理对象 这段也可以不在此类, 也可以放在客户端里面
86      */
87     public Object createProxy(Object targetObject) {
88         this.targetObject = targetObject;
89         return
Proxy.newProxyInstance(targetObject.getClass().getClassLoader(),
90             targetObject.getClass().getInterfaces(), this);
91     };
92
93
94     /**
95      * 此方法为必须实现的, 在代理实例上处理方法调用并返回结果。在与方法关联
的代理实例上调用方法时, 将在调用处理程序上调用此方法。
96      */
}
```


Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
97     @Override
98     public Object invoke(Object proxy, Method method, Object[] args)
99         throws Throwable {
100         Object result = null;
101         System.out.println("递给领导");
102         result = method.invoke(this.targetObject, args);
103         System.out.println("装入袋子, 送出");
104         return result;
105     }
106
107 }
```

Java 代码

```
108 package proxy;
109
110 import java.lang.reflect.Proxy;
111
112 /**
113  *
114  * 作者: alaric
115  * 时间: 2013-7-24下午10:44:37
116  * 描述: 测试类 包括静态代理和动态代理
117  */
118 public class Client {
119
120     /**
121      * @param args
122      */
123     public static void main(String[] args) {
124         //静态代理测试
125         CEO ceo = new CEO();
126         Leader leader1 = new Assistant(ceo);
127         leader1.sign();
128
129         System.out.println("=====");
130         //动态代理测试, 一些三种方式都可以获得动态代理对象
131         AssistantHandler ah = new AssistantHandler(ceo);
132         //Leader leader2 = (Leader) ah.createProxy(new CEO());
133         //leader2.sign();
134
135         //Leader leader3 = (Leader)
136         Proxy.newProxyInstance(CEO.class.getClassLoader(),
137             // ceo.getClass().getInterfaces(), ah);
138         //leader3.sign();
139     }
140 }
```

```
139      Leader leader4 = (Leader)
Proxy.newProxyInstance(Leader.class.getClassLoader(),
140                      new Class[] { Leader.class },
141                      ah);
142      leader4.sign();
143
144  }
145
146 }
```

运行结果如下:

递给领导

CEO 签文件

装入袋子, 送出

=====

递给领导

CEO 签文件

装入袋子, 送出

通过上面例子和代码可以看出, 动态代理显得更为灵活, 实际过程中动态代理也较为常用。

十四、享元模式

享元 (Flyweight) 模式: 通过共享技术以便有效的支持大量细粒度的对象。

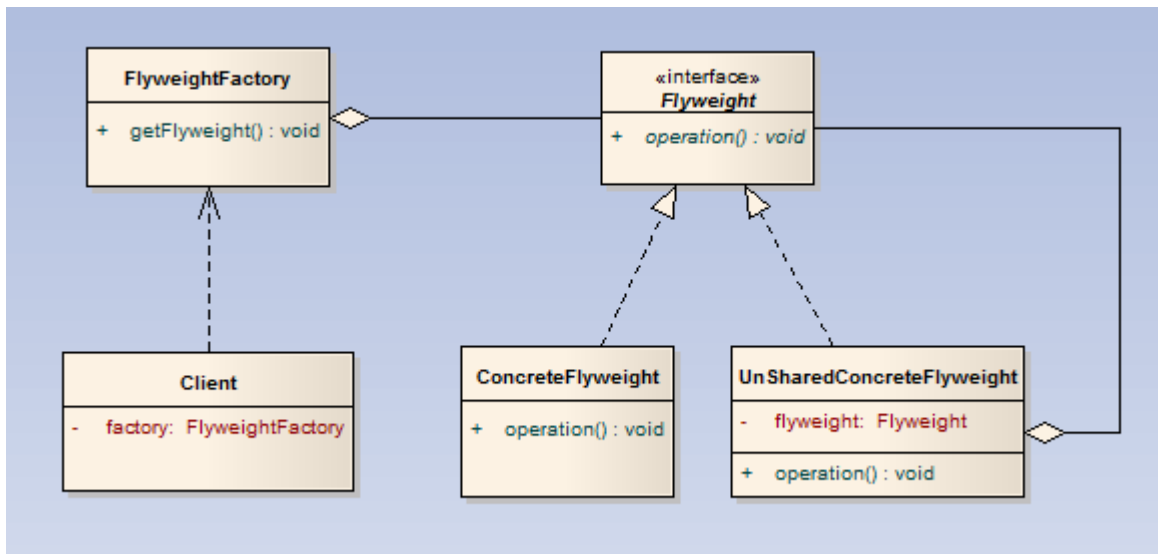
享元模式在阎宏的《java 与模式》中分为单纯享元模式和复合享元模式, 复合模式的复合享元是不可以共享的, 享元对象能做到共享的关键是区分内蕴态 (Internal State) 和外蕴态 (External State)。这两个“蕴态”翻译的太难懂, 我不是说翻译的不好, 可能是我理解能力差, 还是《Design Pattern Elements of Reusable Object-Oriented Software》的翻译版《设计模式可复用面向对象软件的基础》一书总翻译为内部对象和外部对象, 相对直白, 对概念性的东西文学气味太强了就觉得很别扭。这里的角色也采用《设计模式可复用面向对象软件的基础》的说法, 不区分单纯模式和复合模式, 而是有一个 UnSharedConcreteFlyweight (在《java 与模式》里称复合享元, 指明复合享元不能共享), 我们这里称它不可以共享享元角色, 这样享元模式的角色有:
抽象享元 (Flyweight) 角色: 是给实现享元提供的接口。

具体享元 (ConcreteFlyweight) 角色: 实现抽象角色, 此对象必须是共享的, 所含的状态必须是内部状态。

不共享享元 (UnSharedConcreteFlyweight) 角色: 此对象不可共享, 不是所有实现抽象享元接口的对象都要共享, 此对象通常将 ConcreteFlyweight 作为组成元素。

享元工厂 (FlyweightFactory) 角色: 负责创建和管理享元角色, 确保合理共享。

客户端 (Client) 角色: 维持一个 Flyweight 对象的引用, 计算或存储一个 (多个) 外部存储状态。享元模式的类的机构图如下:



享元模式在 `java.lang.String` 设计上的使用, 我们知道 `java` 中字符串始终保持共享一份, 如下面代码片段:

```
String m = "a";
```

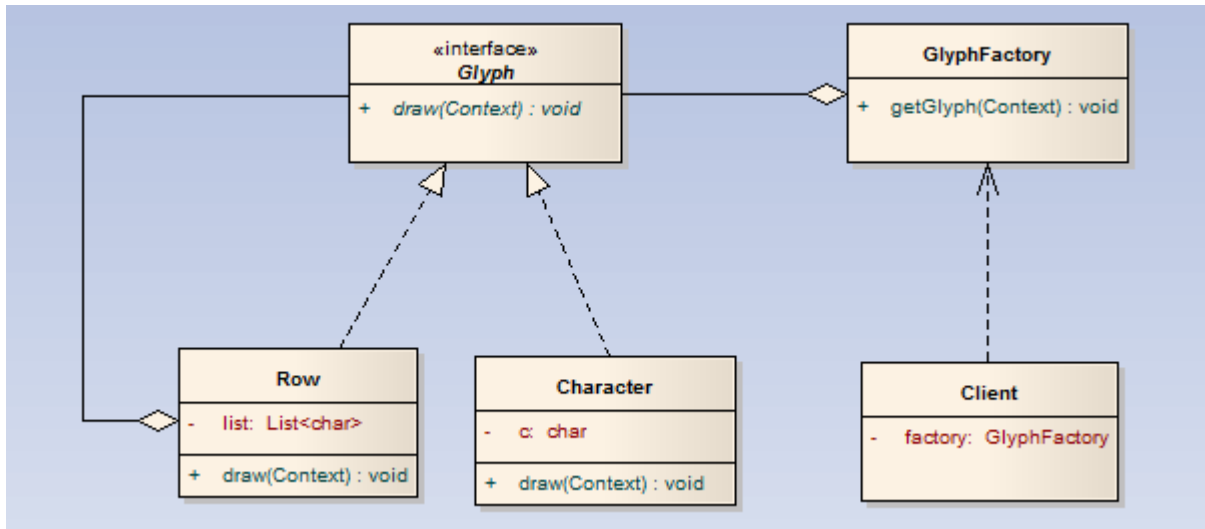
```
String n = "a";
```

```
System.out.println(m==n);
```

这样会输出 `true`, 说明 `m` 和 `n` 指向了同一个实例, 内存中也只有一个 "a"。这就是享元模式在 `String` 上的使用。

享元模式在文字编辑存贮过程中的使用, 这里假定文章由行对象组成, 行对象由若干个字符对象组成, 但是如果每个字符都保存自己的对象, 那么一篇文章成千上万个字符对象, 这样严重消耗系统内存, 造成不可接受的运行时开销, 好的方法是利用享元模式, 只保存 `ASCII` 字符编码值, 作为内部不变的状态, 对当个字符对象进行共享, 而相对字符颜色、大小这样的格式化数据作为外部状态, 由客户端维护, 运行时由外部传入即可。每个行作为不可共享享元对象, 它是由享元对象 (字符对象) 组合而成的。

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>



举例代码如下:

Java 代码



```
1 package flyWeight;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-7-27下午4:52:41
6  *描述: 抽象享元
7  */
8 public interface Glyph {
9
10     public void draw(Context context);
11
12 }
```

Java 代码

```
13 package flyWeight;
14 /**
15  *
16  *作者: alaric
17  *时间: 2013-7-27下午4:53:12
18  *描述: 具体享元
19  */
20 public class Character implements Glyph {
21
22     private char c;
23     private int size;
24     @Override
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
25     public void draw(Context context) {
26         // TODO Auto-generated method stub
27         this.size = context.getSize();
28         System.out.println(size+"号"+c+"被画出!");
29     }
30     public char getC() {
31         return c;
32     }
33     public void setC(char c) {
34         this.c = c;
35     }
36     public Character(char c) {
37         super();
38         this.c = c;
39         System.out.println(c+"被创建!");
40     }
41 }
```

Java 代码   

```
42 package flyWeight;
43
44 import java.util.ArrayList;
45 import java.util.List;
46 /**
47  *
48  *作者: alaric
49  *时间: 2013-7-27下午4:52:26
50  *描述: 行 不可共享享元
51  */
52 public class Row implements Glyph {
53
54     private List<Character> list = new ArrayList<>();
55
56     @Override
57     public void draw(Context context) {
58
59     }
60
61     public Row() {
62     }
63
64     public void setCharacter(Glyph r) {
65         list.add((Character) r);
66     }
67 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
68     public int getSize() {
69         return list.size();
70     }
71
72     public String getRow() {
73         StringBuilder sb = new StringBuilder();
74         for(Character g:list){
75             sb.append(g.getC());
76         }
77         return sb.toString();
78     }
79
80 }
```

Java 代码   

```
81 package flyWeight;
82
83 import java.util.HashMap;
84 import java.util.Map;
85 /**
86  *
87  *作者: alaric
88  *时间: 2013-7-27下午4:52:08
89  *描述: 享元工厂
90  */
91 public class GlyphFactory {
92
93     private Map<String,Glyph> map = new HashMap<>();
94
95     public Glyph getGlyph(Context context){
96         String cStr = context.getC()+" ";
97         Glyph gl = map.get(cStr);
98         if(gl == null){
99             gl = new Character(context.getC());
100             map.put(cStr, gl);
101         }
102         gl.draw(context);
103         return gl;
104     }
105
106 }
```

Java 代码   

```
107 package flyWeight;
```

```
108 /**
109  *
110  *作者: alaric
111  *时间: 2013-7-27下午3:34:57
112  *描述: 数据类
113  */
114 public class Context {
115
116     private int size;
117     private char c;
118     public int getSize() {
119         return size;
120     }
121     public void setSize(int size) {
122         this.size = size;
123     }
124     public char getC() {
125         return c;
126     }
127     public void setC(char c) {
128         this.c = c;
129     }
130     public Context(int size, char c) {
131         super();
132         this.size = size;
133         this.c = c;
134     }
135
136 }
```

Java 代码   

```
137 package flyWeight;
138 /**
139  *
140  *作者: alaric
141  *时间: 2013-7-27下午4:56:01
142  *描述: 客户端 为了简单 就直接写 main 方法里的
143  */
144 public class Client {
145
146
147     /**
148     *作者: alaric
149     *时间: 2013-7-27下午4:20:08
150     *描述: 测试
```

```
151     */
152     public static void main(String[] args) {
153         Row r =new Row();
154         GlyphFactory factory = new GlyphFactory();
155         Context context1= new Context(12, 'a');
156         Glyph gly1 = factory.getGlyph(context1);
157         r.setCharacter(gly1);
158
159         Context context2= new Context(13, 'a');
160         Glyph gly2 = factory.getGlyph(context2);
161         r.setCharacter(gly2);
162
163         Context context3= new Context(13, 'b');
164         Glyph gly3 = factory.getGlyph(context3);
165         r.setCharacter(gly3);
166
167         System.out.println(r.getRow());
168
169     }
170 }
```

运行结果:

a 被创建!

12号 a 被画出!

13号 a 被画出!

b 被创建!

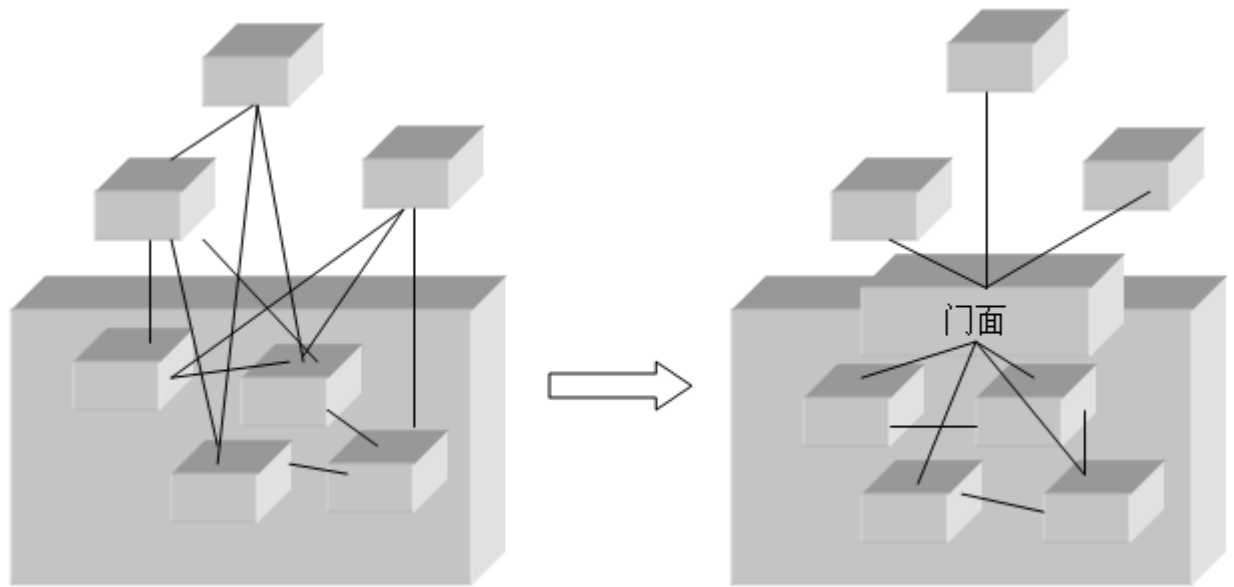
13号 b 被画出!

aab

可以看出 a 创建了一次 , a 的大小12 , 13是外部状态所以是外部传入, 外部状态不能在享元内保存, 而 a 字符是内部状态, 行 Row 是有字符 Character 组成, Row 虽然实现了抽象享元接口, 但是并没有再工厂中体现共享, 因为他是不可共享的享元。

十五、门面模式

门面 (Facade) 模式 (也叫外观模式), 为子系统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。如下图表示:



从上图可以看出门面模式是把复杂的业务封装到了子系统内部,而对外部来说只有一个统一的访问界面,使得子系统更加简单,容易被客户端使用。门面模式的体现的是面向对象设计里面的迪米特法则 (Law of Demeter) 简称 LoD, 讲的是一个软件实体应当与尽可能少的其他软件实体发生相互作用,通过上面的示意图很明天体现的就是这一点。

门面模式的角色:

门面 (Facade) 角色: 客户端通过此角色能了解到子系统提供的功能,门面角色会委派任务到相应的子系统中去。

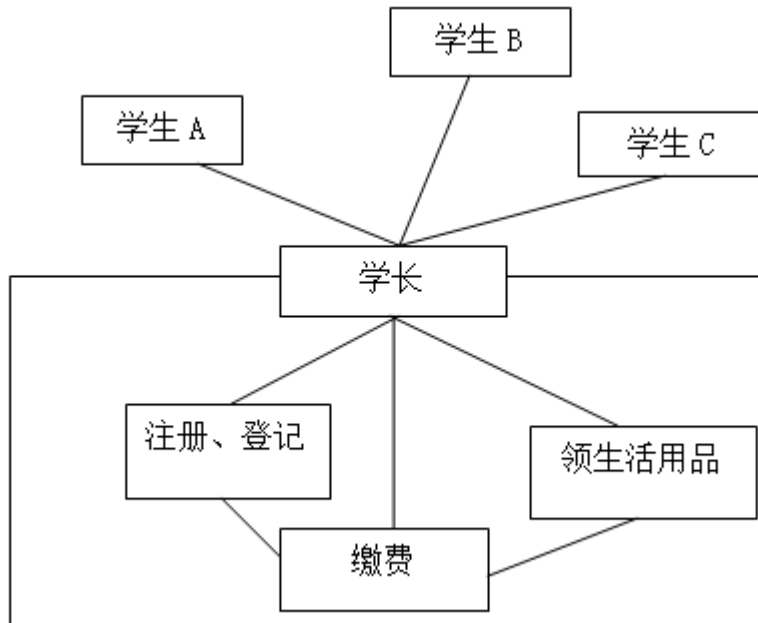
子系统 (SubSystem) 角色: 子系统不是单独的类,而是类的集合。负责提供业务逻辑,对子系统来说门面就是它的一个客户端。

在 GOF 中门面模式没有一个一般化的类图表示,这里用实例说明。门面模式在生活中常见,小时候玩过的游戏机,零花钱都买币玩那个三国志了,一个操作面板相当于一个门面,上面操作杆和操作按钮相当于门面提供的方法,这些方法实现是委托游戏机里面的各个部件,各个部件相互调用,相互配合完成了游戏的控制和操作,把复杂的业务逻辑封装起来,只提供简单有效的操作,这样十几岁的小孩子,乃至8,9岁的孩子都玩的很憋,如果不封装起来,给他电路,显示器,控制元件,估计没人会玩。

还记得我们大学的第一天吗,一个陌生的城市,陌生的环境,入学报到可以说相对复杂和麻烦

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

的事情,但是往往都会有学生会的学长、学姐的接待和帮助,基本上跟着他走,所有事情都由他给你办了,只要签字或交钱就 ok 了,一切便的很简单。



代码如下:

Java 代码   

```
1 package facade;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-7-30下午7:53:07
6  *描述: 招生办
7  */
8 public class EnrollmentOffice{
9
10     /**
11      * 注册
12      */
13     public void register(){
14         System.out.println("注册");
15     }
16 }
```

Java 代码   

```
17 package facade;
18 /**
19  *
20  *作者: alaric
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
21 *时间: 2013-7-30下午7:54:24
22 *描述: 财务办
23 */
24 public class FinanceSection {
25
26     /**
27      * 缴费
28      */
29     public void payment() {
30         System.out.println("交学费");
31     }
32 }
```

Java 代码   

```
33 package facade;
34
35 /**
36  *
37  *作者: alaric
38  *时间: 2013-7-30下午7:54:44
39  *描述: 学生办
40  */
41 public class StudentAffairsOffice {
42
43     public void getSomeGoods() {
44         System.out.println("领取生活用品");
45     }
46 }
```

Java 代码   

```
47 package facade;
48 /**
49  *
50  *作者: alaric
51  *时间: 2013-7-30下午7:53:46
52  *描述: 门面类, 就是我们的学长, 学姐
53  */
54 public class Facade {
55
56     EnrollmentOffice enroll = new EnrollmentOffice();
57     FinanceSection finance = new FinanceSection();
58     StudentAffairsOffice studentAffairs = new StudentAffairsOffice();
59     public void helpJion() {
60         enroll.register();
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
61         finance.payment();
62         studentAffairs.getSomeGoods();
63     }
64
65 }
```

Java 代码   

```
66 package facade;
67 /**
68  *
69  *作者: alaric
70  *时间: 2013-7-30下午7:52:39
71  *描述: 客户端
72  */
73 public class Client {
74
75     public static void main(String[] args) {
76         Facade facade = new Facade();
77         facade.helpJion();
78     }
79 }
```

运行结果:

注册

交学费

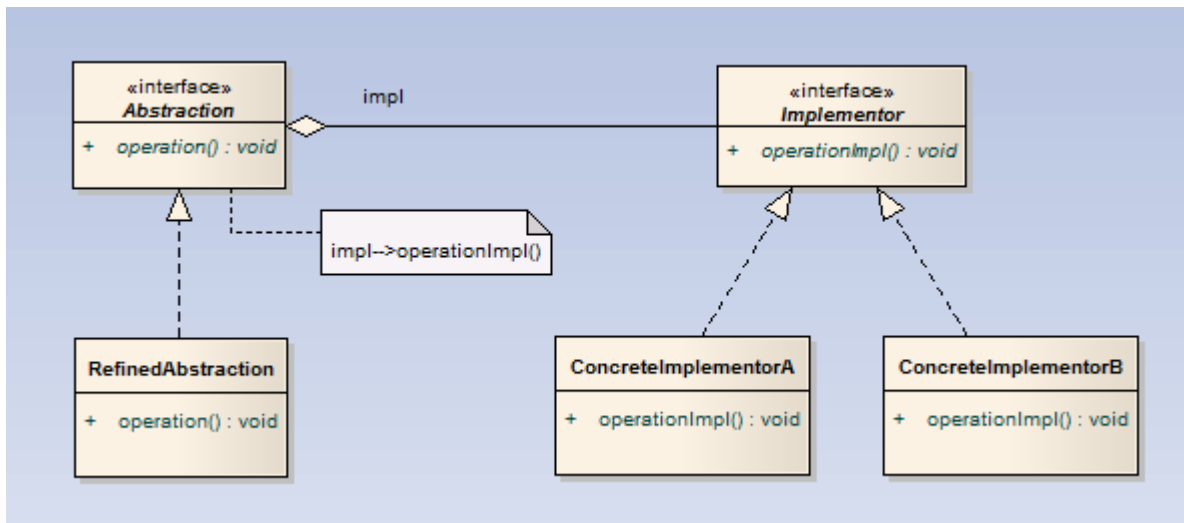
领取生活用品

门面模式的好处在于对于客户端子系统屏蔽了内部组件,减少客户单处理对象的数目,使客户端调用变的简单;将客户端和子系统解耦,子系统内部紧耦合,达到了系统设计的高内聚低耦合的要求。

[十六、桥梁模式](#)

桥梁 (Bridge) 模式: 又称 Handle/Body。将抽象部分和实现部分分离,使它们都可以独立的变化。桥梁模式属于对象的结构模式。

GOF 桥梁模式的示意性结构类图如下:



通过上图可以看出桥梁模式有以下角色:

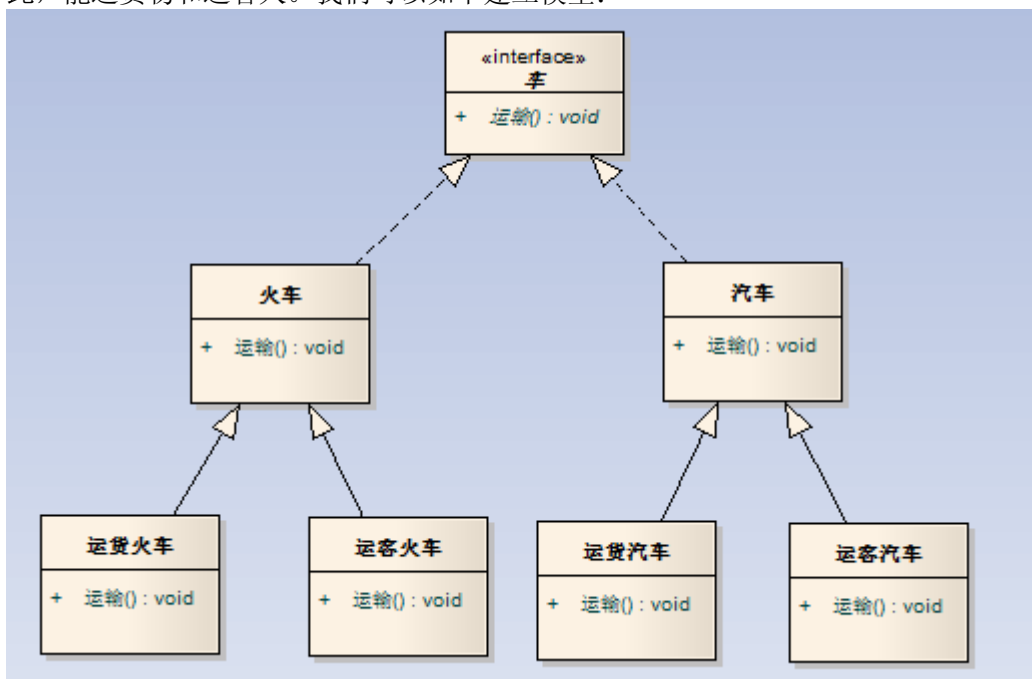
抽象化 (Abstraction) 角色: 给出抽象化定义并持有一个实现化对象的引用。

修正抽象化 (Refined Abstraction) 角色: 扩展抽象化角色, 改变和修正父类对抽象化的定义。

实现化 (Implementor) 角色: 给出实现化的接口角色的接口, 但不给出具体的实现。

具体实现化 (Concrete Implementor) 角色: 给出实现化角色接口的具体实现。

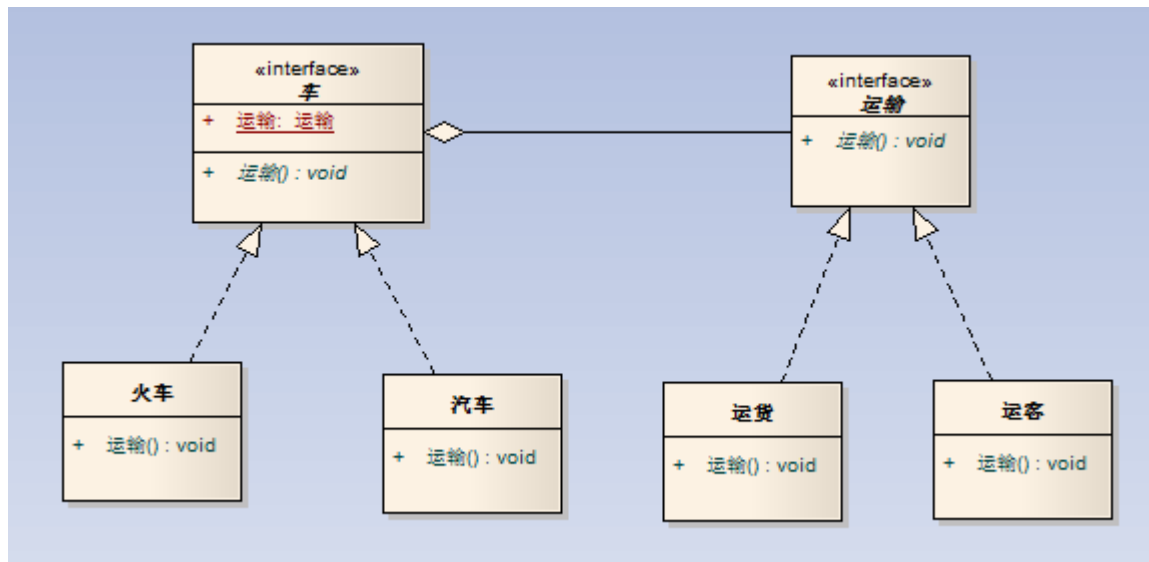
桥梁模式在于将抽象和实现分离 (也就是我们常说的解耦), 让抽象和实现能够独立变化。来举个例子: 抽象的车, 有火车和汽车, 而火车有客车和火车, 也就是能运货物和人; 汽车也是如此, 能运货物和运客人。我们可以如下建立模型:



Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

这种模式将车的属性和行为分类,让车 和运输相互独立演变,类的数量由原来的 $M*N$ 变成 $M+N$ 的数量,利用组合方式代替继承,这也符合‘组合聚合复用原则’,组合聚合复用原则讲的是要尽可能使用组合、聚合来达到复用目的而不是利用继承。对于增加一个运输方式或者增加一个车的种类,直接添加即可,不必修改其他类,也只需修改一处即可。

上述描述代码如下:



这种模式将车的属性和行为分类,让车 和运输相互独立演变,类的数量由原来的 $M*N$ 变成 $M+N$ 的数量,利用组合方式代替继承,这也符合‘组合聚合复用原则’,组合聚合复用原则讲的是要尽可能使用组合、聚合来达到复用目的而不是利用继承。对于增加一个运输方式或者增加一个车的种类,直接添加即可,不必修改其他类,也只需修改一处即可。上述描述代码如下:

Java 代码   

```
1 package bridge;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-8-2上午7:41:31
6  *描述: 车的抽象类
7  */
8 public abstract class Vehicle {
9
10     private Transport implementor;
11
12     public void transport() {
13         implementor.transport();
14     }
15     public Transport getImplementor() {
16         return implementor;
17     }
18 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
17     }
18     public void setImplementor(Transport implementor) {
19         this.implementor = implementor;
20     }
21
22 }
```

Java 代码   

```
23 package bridge;
24 /**
25  *
26  *作者: alaric
27  *时间: 2013-8-2上午7:45:40
28  *描述: 汽车的实现类
29  */
30 public class Car extends Vehicle {
31
32     @Override
33     public void transport() {
34
35         System.out.print("汽车");
36         super.transport();
37     }
38
39 }
```

Java 代码   

```
40 package bridge;
41 /**
42  *
43  *作者: alaric
44  *时间: 2013-8-2上午7:45:58
45  *描述: 火车的实现类
46  */
47 public class Train extends Vehicle {
48
49     @Override
50     public void transport() {
51
52         System.out.print("火车");
53         super.transport();
54     }
55 }
```

```
56 }
```

Java 代码   

```
57 package bridge;
58 /**
59  *
60  *作者: alaric
61  *时间: 2013-8-2上午7:46:17
62  *描述: 车的实现接口
63  */
64 public interface Transport {
65
66     /**
67      *
68      *作者: alaric
69      *时间: 2013-8-2上午7:46:38
70      *描述: 运输
71      */
72     public void transport();
73
74 }
```

Java 代码   

```
75 package bridge;
76 /**
77  *
78  *作者: alaric
79  *时间: 2013-8-2上午7:41:00
80  *描述: 货车
81  */
82 public class Goods implements Transport{
83
84     @Override
85     public void transport() {
86
87         System.out.println("运货");
88     }
89
90 }
```

Java 代码   

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
91 package bridge;
92 /**
93  *
94  *作者: alaric
95  *时间: 2013-8-1下午10:58:25
96  *描述: 客车
97  */
98 public class Guest implements Transport{
99
100     @Override
101     public void transport() {
102
103         System.out.println("运客");
104
105     }
106
107 }
```

Java 代码   

```
108 package bridge;
109 /**
110  *
111  *作者: alaric
112  *时间: 2013-8-2上午7:45:24
113  *描述: 测试类
114  */
115 public class Client {
116
117     /**
118     *作者: alaric
119     *时间: 2013-8-2上午7:45:16
120     *描述:
121     */
122     public static void main(String[] args) {
123         //造出一辆火车来
124         Train train = new Train();
125         //装入货物
126         train.setImplementor(new Goods());
127         //运输
128         train.transport();
129         //上客
130         train.setImplementor(new Guest());
131         //运输
132         train.transport();
133     }
```

```
134
135     //造出一汽车来
136     Car car = new Car();
137     //装入货物
138     car.setImplementor(new Goods());
139     //运输
140     car.transport();
141     //上客
142     car.setImplementor(new Guest());
143     //运输
144     car.transport();
145
146 }
147
148
149 }
```

运行结果如下:

火车运货

火车运客

汽车运货

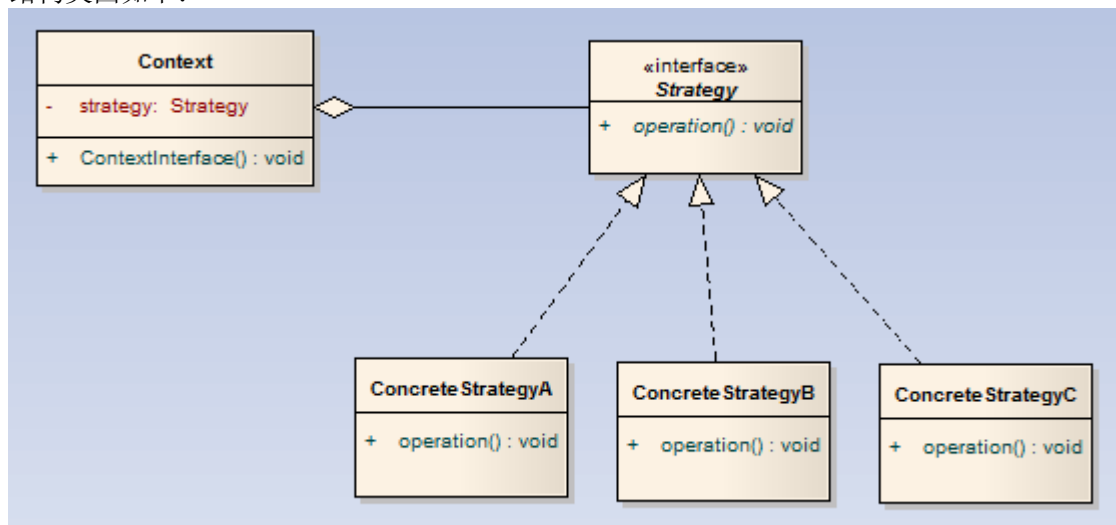
汽车运客

通过上面代码可以看出, 桥梁模式有如下优点:

- 1、分离接口及实现部分 一个实现不必一直绑定在一个接口上;
- 2、提高可扩充性, 使扩展变得简单;

十七、策略模式

策略 (Strategy) 模式: 又名 Policy, 它的用意是定义一组算法, 把它们一个个封装起来, 并且使他们可以相互替换。策略模式可以独立于使用他们的客户端而变化。GOF 策略模式静态结构类图如下:



Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

通过上图可以看出策略模式有以下角色构成:

- 1、抽象策略 (**Strategy**) 角色: 抽象策略角色由抽象类或接口来承担, 它给出具体策略角色需要实现的接口;
- 2、具体策略 (**ConcreteStrategy**) 角色: 实现封装了具体的算法或行为;
- 3、场景 (**Context**) 角色: 持有抽象策略类的引用。

策略模式重点是封装不同的算法和行为, 不同的场景下可以相互替换。策略模式是开闭原则的体现, 开闭原则讲的是一个软件实体应该对扩展开放对修改关闭。策略模式在新的策略增加时, 不会影响其他类的修改, 增加了扩展性, 也就是对扩展是开放的; 对于场景来说, 只依赖于抽象, 而不依赖于具体实现, 所以对修改是关闭的。策略模式的认识可以借助《java 与模式》一书中写到诸葛亮的锦囊妙计来学习, 在不同的场景下赵云打开不同的锦囊, 便化险为夷, 锦囊便是抽象策略, 具体的锦囊里面的计策便是具体的策略角色, 场景就是赵云, 变化的处境选择具体策略的条件。

策略模式在程序设计中也很常用, 在板桥 (banq) 的博客里有篇文章叫 “你还在用 if else 吗?” “<http://www.jdon.com/articheck/ifelse.htm>” 讲的很好, 策略模式不但是继承的代替方案而且能很好地解决 if else 问题, 下面举个实例来说明, 怎么使用策略模式。

需求如下:

某支付系统接入以下几种商户进行充值: 易宝网易, 快线网银, 19pay 手机支付, 支付宝支付, 骏网一卡通, 由于每家充值系统的结算比例不一样, 而且同一家商户的不同充值方式也有所不同, 具体系统情况比较复杂, 像支付宝既有支付宝账号支付和支付宝网银支付等这些暂时不考虑, 为了讲述策略模式这里简单描述, 假如分为四种, 手机支付, 网银支付, 商户账号支付和点卡支付。因为没个支付结算比例不同, 所以对手续费低的做一些优惠活动, 尽可能让用户使用手续费低的支付方式来充值, 这样降低渠道费用, 增加收入, 具体优惠政策如下:

- ①网银充值, 8.5折;
- ②商户充值, 9折;
- ③手机充值, 没有优惠;
- ④点卡充值, 收取1%的渠道费;

对于一个新手的代码如下:

Java 代码   

```
1 package strategy;
2
3 public class Example {
4
5     /**
6     *
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
7      *作者: alaric
8      *时间: 2013-8-5上午11:00:06
9      *描述: 计算用户所付金额
10     */
11     public Double calRecharge(Double charge ,RechargeTypeEnum type ){
12
13         if(type.equals(RechargeTypeEnum.E_BANK)){
14             return charge*0.85;
15         }else if(type.equals(RechargeTypeEnum.BUSI_ACCOUNTS)){
16             return charge*0.90;
17         }else if(type.equals(RechargeTypeEnum.MOBILE)){
18             return charge;
19         }else if(type.equals(RechargeTypeEnum.CARD_RECHARGE)){
20             return charge+charge*0.01;
21         }else{
22             return null;
23         }
24
25     }
26
27 }
```

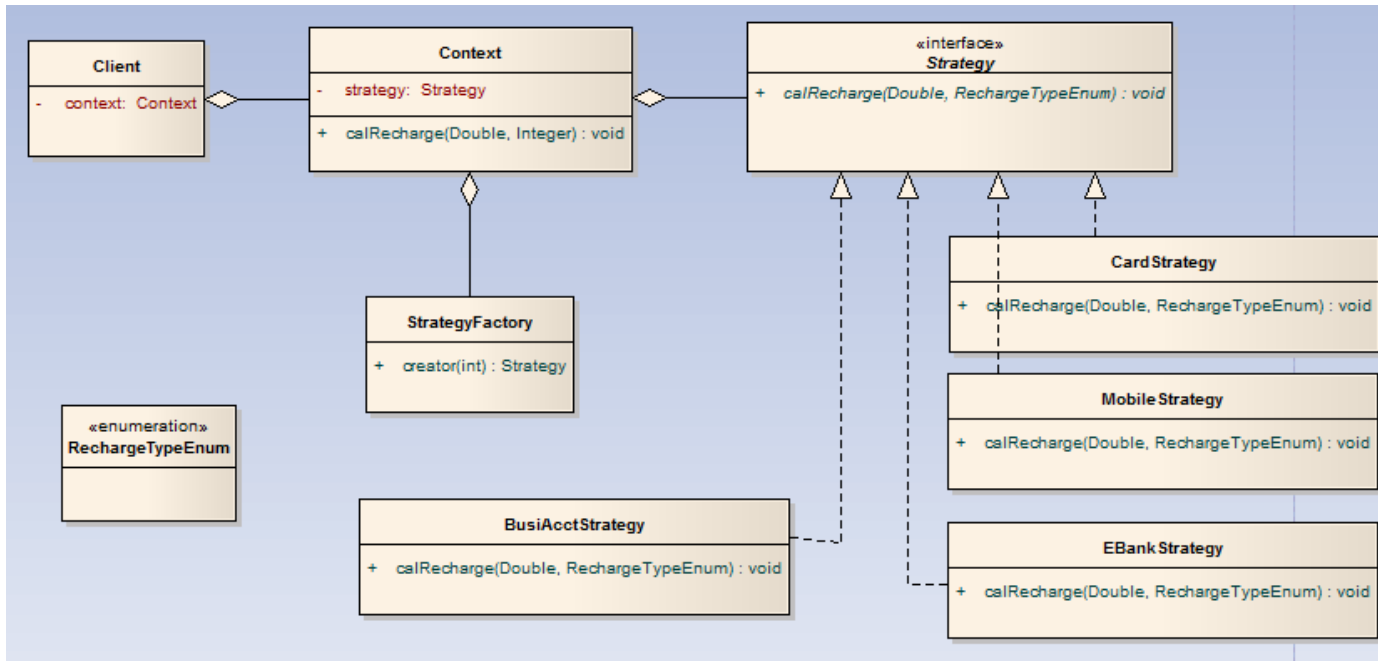
Java 代码   

```
28 package strategy;
29
30 public enum RechargeTypeEnum {
31
32     E_BANK(1, "网银"),
33
34     BUSI_ACCOUNTS(2, "商户账号"),
35
36     MOBILE(3, "手机卡充值"),
37
38     CARD_RECHARGE(4, "充值卡")
39     ;
40
41     /**
42      * 状态值
43      */
44     private int value;
45
46     /**
47      * 类型描述
48      */
49     private String description;
```

```
50
51
52
53     private RechargeTypeEnum(int value, String description) {
54         this.value = value;
55         this.description = description;
56     }
57
58     public int value() {
59         return value;
60     }
61     public String description() {
62         return description;
63     }
64
65
66     public static RechargeTypeEnum valueOf(int value) {
67         for(RechargeTypeEnum type : RechargeTypeEnum.values()) {
68             if(type.value() == value) {
69                 return type;
70             }
71         }
72         return null;
73     }
74 }
```

可以看出上面四种不同的计算方式在一个方法内部,不利于扩展和维护,当然也不符合面向对象设计原则。对以上的代码利用策略模式进行修改,类图如下:

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>



实例代码如下:

Java 代码

```
75 package strategy.strategy;
76
77 import strategy.RechargeTypeEnum;
78
79 /**
80  *
81  *作者: alaric
82  *时间: 2013-8-5上午11:03:17
83  *描述: 策略抽象类
84  */
85 public interface Strategy {
86
87     /**
88      *
89      *作者: alaric
90      *时间: 2013-8-5上午11:05:11
91      *描述: 策略行为方法
92      */
93     public Double calRecharge(Double charge ,RechargeTypeEnum type );
94 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

Java 代码   

```
95 package strategy.strategy;
96
97 import strategy.RechargeTypeEnum;
98 /**
99  *
100  *作者: alaric
101  *时间: 2013-8-5上午11:14:23
102  *描述: 网银充值
103  */
104 public class EBankStrategy implements Strategy{
105
106     @Override
107     public Double calRecharge(Double charge, RechargeTypeEnum type) {
108         return charge*0.85;
109     }
110
111
112
113 }
```

Java 代码   

```
114 package strategy.strategy;
115
116 import strategy.RechargeTypeEnum;
117 /**
118  *
119  *作者: alaric
120  *时间: 2013-8-5上午11:14:08
121  *描述: 商户账号充值
122  */
123 public class BusiAcctStrategy implements Strategy{
124
125     @Override
126     public Double calRecharge(Double charge, RechargeTypeEnum type) {
127         // TODO Auto-generated method stub
128         return charge*0.90;
129     }
130
131 }
```

Java 代码   

```
132 package strategy.strategy;
133
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
134 import strategy.RechargeTypeEnum;
135 /**
136  *
137  *作者: alaric
138  *时间: 2013-8-5上午11:14:43
139  *描述: 手机充值
140  */
141 public class MobileStrategy implements Strategy {
142
143     @Override
144     public Double calRecharge(Double charge, RechargeTypeEnum type) {
145         // TODO Auto-generated method stub
146         return charge;
147     }
148
149 }
```

Java 代码   

```
150 package strategy.strategy;
151
152 import strategy.RechargeTypeEnum;
153 /**
154  *
155  *作者: alaric
156  *时间: 2013-8-5上午11:13:46
157  *描述: 充值卡充值
158  */
159 public class CardStrategy implements Strategy{
160
161     @Override
162     public Double calRecharge(Double charge, RechargeTypeEnum type) {
163         return charge+charge*0.01;
164     }
165
166 }
```

Java 代码   

```
167 package strategy.strategy;
168
169 import strategy.RechargeTypeEnum;
170
171 /**
172  *
173  *作者: alaric
```


Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
174 *时间: 2013-8-5上午11:03:38
175 *描述: 场景类
176 */
177 public class Context {
178
179     private Strategy strategy;
180
181     public Double calRecharge(Double charge, Integer type) {
182         strategy = StrategyFactory.getInstance().creator(type);
183         return strategy.calRecharge(charge, RechargeTypeEnum.valueOf(type));
184     }
185
186     public Strategy getStrategy() {
187         return strategy;
188     }
189
190     public void setStrategy(Strategy strategy) {
191         this.strategy = strategy;
192     }
193
194 }
```

Java 代码   

```
195 package strategy.strategy;
196
197 import java.util.HashMap;
198 import java.util.Map;
199
200 import strategy.RechargeTypeEnum;
201 /**
202  *
203  *作者: alaric
204  *时间: 2013-8-5上午11:31:12
205  *描述: 策略工厂 使用单例模式
206  */
207 public class StrategyFactory {
208
209     private static StrategyFactory factory = new StrategyFactory();
210     private StrategyFactory() {
211     }
212     private static Map<Integer, Strategy> strategyMap = new HashMap<>();
213     static {
214         strategyMap.put(RechargeTypeEnum.E_BANK.value(), new
215         EBankStrategy());
216         strategyMap.put(RechargeTypeEnum.BUSI_ACCOUNTS.value(), new
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
BusiAcctStrategy());
216     strategyMap.put(RechargeTypeEnum.MOBILE.value(), new
MobileStrategy());
217     strategyMap.put(RechargeTypeEnum.CARD_RECHARGE.value(), new
CardStrategy());
218 }
219 public Strategy creator(Integer type){
220     return strategyMap.get(type);
221 }
222 public static StrategyFactory getInstance(){
223     return factory;
224 }
225 }
```

Java 代码   

```
226 package strategy.strategy;
227
228 import strategy.RechargeTypeEnum;
229
230 public class Client {
231
232     /**
233      * 作者: alaric 时间: 2013-8-5上午11:33:52 描述:
234      */
235     public static void main(String[] args) {
236
237         Context context = new Context();
238         // 网银充值100 需要付多少
239         Double money = context.calRecharge(100D,
240             RechargeTypeEnum.E_BANK.value());
241         System.out.println(money);
242
243         // 商户账户充值100 需要付多少
244         Double money2 = context.calRecharge(100D,
245             RechargeTypeEnum.BUSI_ACCOUNTS.value());
246         System.out.println(money2);
247
248         // 手机充值100 需要付多少
249         Double money3 = context.calRecharge(100D,
250             RechargeTypeEnum.MOBILE.value());
251         System.out.println(money3);
252
253         // 充值卡充值100 需要付多少
254         Double money4 = context.calRecharge(100D,
255             RechargeTypeEnum.CARD_RECHARGE.value());
```

```
256         System.out.println(money4);
257     }
258
259 }
```

运行结果:

85.0

90.0

100.0

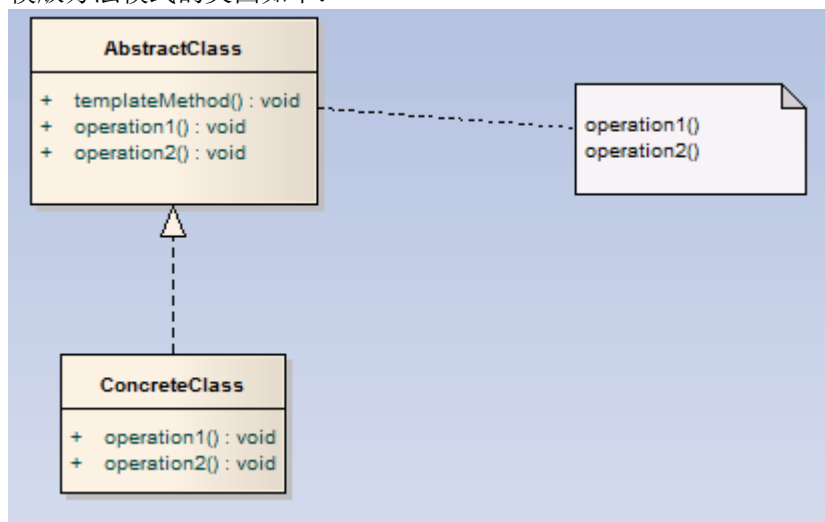
101.0

从上面类图和代码可以看出, 策略模式把具体的算法封装到了具体策略角色内部, 增强了可扩展性, 隐蔽了实现细节; 它替代继承来实现, 避免了 **if-else** 这种不易维护的条件语句。当然我们也可以看到, 策略模式由于独立策略实现, 使得系统内增加了很多策略类; 对客户端来说必须知道兜友哪些具体策略, 而且需要知道选择具体策略的条件。

十八、模版方法模式

模版方法 (Template Method) 模式: 属于类的行为模式, 它的用意是定义一个操作中的算法的骨架, 将一些操作延迟到子类中。使得子类可以不改变一个算法的结构即可重新定义该算法的某些特定步骤。

模版方法模式的类图如下:

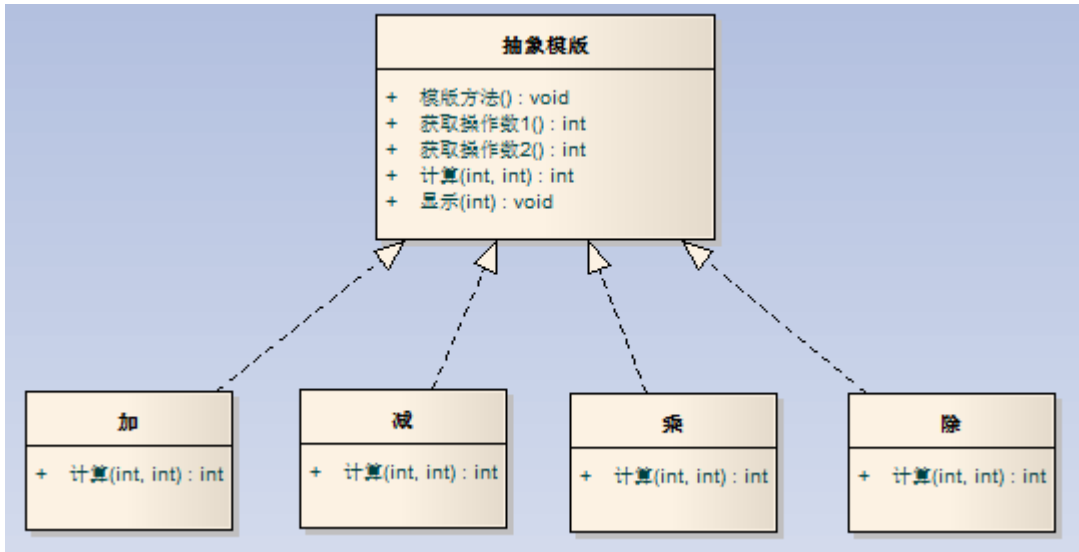


模版方法模式有两个角色:

- 1、**抽象模版 (Abstract Template) 角色:** 定义了一个或多个抽象操作, 同时定义了一个模版方法, 它是一个具体的方法, 作为抽象模版的骨架。实际的逻辑在抽象操作中, 推迟到子类实现。
- 2、**实现模版 (Concrete Template) 角色:** 实现了抽象模版中一个或多个抽象方法。

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

这里举个模拟计算器的例子来说模版方法模式，我们知道计算器计算的时候首先获取操作数1，然后获取操作数2，再进行加减乘除运算，最后显示在屏幕上，对于计算器来说不管是计算加减乘除那一种，都是要获取两个操作数，计算后再显示。可以建立以下类图模型：



举例代码如下：

Java 代码   

```
1 package templateMethod;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-8-9下午8:15:18
6  *描述: 抽象模版角色
7  */
8 public abstract class AbstractClass {
9
10     /**
11     *
12     *作者: alaric
13     *时间: 2013-8-9下午8:17:00
14     *描述: 模版方法
15     */
16     public final void templateMethod() {
17         int m = getNum1();
18         int n = getNum2();
19         int s = operate(m, n);
20         show(s);
21     }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
22
23  /**
24   *
25   *作者: alaric
26   *时间: 2013-8-9下午8:21:49
27   *描述: 获取第一个操作数
28   */
29  private int getNum1() {
30      return 8;
31  }
32  /**
33   *
34   *作者: alaric
35   *时间: 2013-8-9下午8:21:49
36   *描述: 获取第二个操作数
37   */
38  private int getNum2() {
39      return 2;
40  }
41  /**
42   *
43   *作者: alaric
44   *时间: 2013-8-9下午8:19:53
45   *描述: 算法
46   */
47  public abstract int operate(int m,int n);
48  /**
49   *
50   *作者: alaric
51   *时间: 2013-8-9下午8:20:59
52   *描述: 显示
53   */
54  public void show(int s) {
55      System.out.println("结果是:" + s);
56  }
57
58 }
```

Java 代码   

```
59 package templateMethod;
60 /**
61  *
62  *作者: alaric
63  *时间: 2013-8-9下午8:57:06
64  *描述: 加法
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
65  */
66  public class AddClass extends AbstractClass{
67
68      @Override
69      public int operate(int m,int n) {
70          return m+n;
71      }
72
73 }
```

Java 代码   

```
74 package templateMethod;
75 /**
76  *
77  *作者: alaric
78  *时间: 2013-8-9下午8:57:24
79  *描述: 减法
80  */
81 public class SubClass extends AbstractClass{
82
83     @Override
84     public int operate(int m,int n) {
85         return m-n;
86     }
87
88 }
```

Java 代码   

```
89 package templateMethod;
90 /**
91  *
92  *作者: alaric
93  *时间: 2013-8-9下午8:57:38
94  *描述: 乘法
95  */
96 public class MultiClass extends AbstractClass{
97
98     @Override
99     public int operate(int m,int n) {
100         return m*n;
101     }
102
103 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

Java 代码   

```
104 package templateMethod;
105 /**
106  *
107  *作者: alaric
108  *时间: 2013-8-9下午8:57:50
109  *描述: 除法
110  */
111 public class DivClass extends AbstractClass{
112
113     @Override
114     public int operate(int m,int n) {
115         return m/n;
116     }
117
118 }
```

Java 代码   

```
119 package templateMethod;
120 /**
121  *
122  *作者: alaric
123  *时间: 2013-8-9下午8:14:34
124  *描述: 测试类
125  */
126 public class Client {
127
128     /**
129     *作者: alaric
130     *时间: 2013-8-9下午8:14:28
131     *描述:
132     */
133     public static void main(String[] args) {
134         //加
135         AbstractClass c1 = new AddClass();
136         c1.templateMethod();
137         //减
138         AbstractClass c4 = new SubClass();
139         c4.templateMethod();
140         //乘
141         AbstractClass c2 = new MultiClass();
142         c2.templateMethod();
143         //除
144         AbstractClass c3 = new DivClass();
145         c3.templateMethod();
146     }
147 }
```

```
146
147
148     }
149
150 }
```

运行结果:

结果是:10

结果是:6

结果是:16

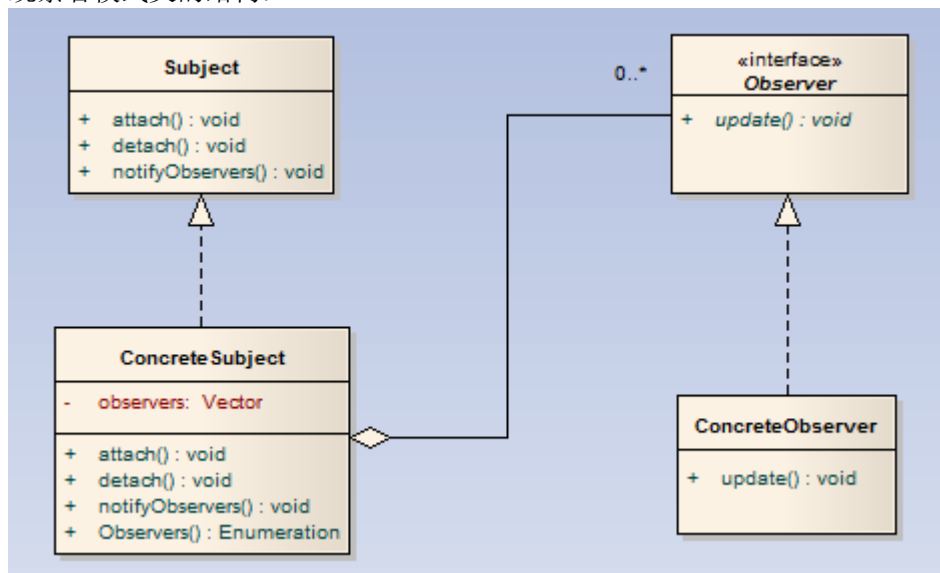
结果是:4

整个计算过程(取操作数, 计算, 显示)使用了模版方法模式, 模版方法规定程序实现步骤, 并且在子类中无法改变, 而子类必须实现 **abstract** 修饰的抽象方法, 不同的实现类可以有不同的实现, 这也是模版方法模式的用意。如果你了解策略模式, 你会发现在每条算法的封装上很像策略模式, 而又不是策略模式, 策略模式是利用委派的方法提供不同的算法行为, 而模版方法是利用继承来提供不同的算法行为的。

十九、观察者模式

观察者 (Observer) 模式: 是对象的行为模式, 又叫做发布-订阅 (Publish/Subscribe) 模式、模型-视图 (Model/View) 模式、源-监听 (Source/Listener) 模式或者从属 (Dependents) 模式。观察者模式定义了一种一对多的依赖关系, 让多个观察者对象同时监听某一个主题对象, 这个主题对象在状态上发生变化时, 会通知所有观察者对象, 使它们能够自动更新自己。

观察者模式类的结构:



观察者模式角色如下:

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

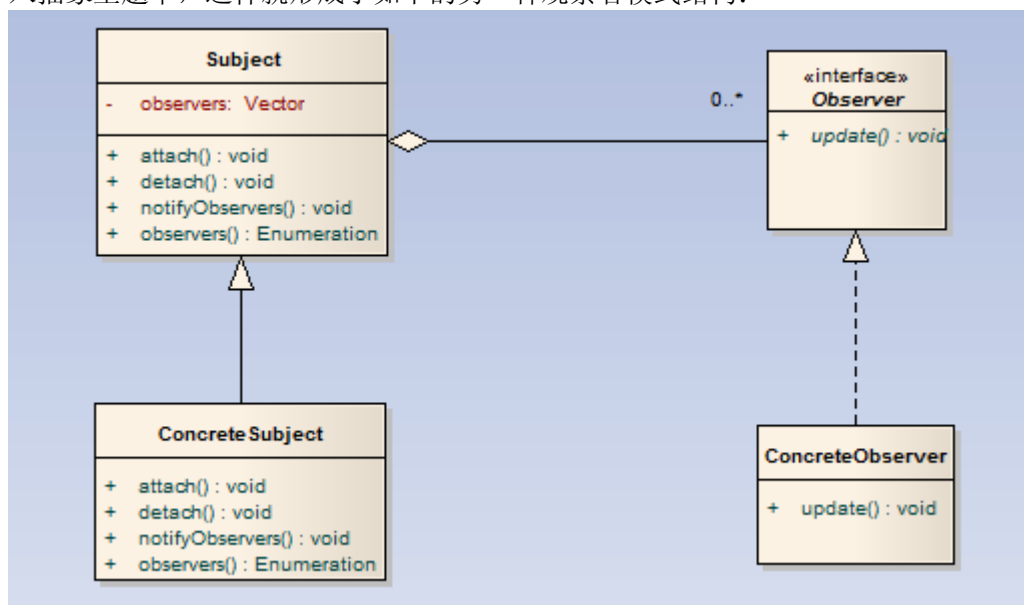
抽象主题 (Subject) 角色: 抽象主题角色提供维护一个观察者对象聚集的操作方法, 对聚集的增加、删除等。

具体主题 (ConcreteSubject) 角色: 将有关状态存入具体的观察者对象; 在具体主题的内部状态改变时, 给所有登记过的观察者发出通知。具体主题角色负责实现抽象主题中聚集的管理方法。

抽象观察者 (Observer) 角色: 为具体观察者提供一个更新接口。

具体观察者 (ConcreteObserver) 角色: 存储与主题相关的自洽状态, 实现抽象观察者提供的更新接口。

仔细观察上面的类图, 发现具体主题角色和抽象观察者之间的连线, 是因为具体主题角色维护了一个观察者引用的聚集, 如果有多个具体主题角色, 意味着每个具体角色都要维护一个观察者的聚集, 那么能不能将聚集提升到抽象主题里面呢? 这个就需要考虑场景, 如果多个主题实现在管理上都有很大差异, 那么就不能提升到抽象角色中, 但是绝大多数情况下, 这些聚集管理方法本身就是所有具体主题所共有的, 所以大多数情况下都是可以将聚集和管理都移入到抽象主题中的, 因为 `notifyObserver()` 方法是依赖于聚集的, 所以将 `notifyObserver()` 也移入抽象主题中, 这样就形成了如下的另一种观察者模式结构:



我们这里简单用代码描述如下:

Java 代码

```
1 package observer.desc;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-8-13下午8:05:08
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
6  *描述: 抽象观察者
7  */
8  public interface Observer {
9
10     public void update();
11
12 }
```

Java 代码   

```
13 package observer.desc;
14 /**
15  *
16  *作者: alaric
17  *时间: 2013-8-13下午8:05:34
18  *描述: 具体观察者
19  */
20 public class ConcreteObserver implements Observer{
21
22     @Override
23     public void update() {
24         // 写业务逻辑
25     }
26
27 }
```

Java 代码   

```
28 package observer.desc;
29 /**
30  *
31  *作者: alaric
32  *时间: 2013-8-13下午8:05:55
33  *描述: 抽象主题
34  */
35 public interface Subject {
36
37     public void attach(Observer observer);
38
39     public void detach(Observer observer);
40
41     void notifyObservers();
42 }
```

Java 代码   

```
43 package observer.desc;
44
45 import java.util.Enumeration;
46 import java.util.Vector;
47 /**
48  *
49  *作者: alaric
50  *时间: 2013-8-13下午8:09:21
51  *描述: 具体主题类
52  */
53 public class ConcreteSubject implements Subject {
54     private Vector<Observer>observersVector = new Vector<Observer>();
55     public void attach(Observer observer) {
56         observersVector.addElement(observer);
57     }
58
59     public void detach(Observer observer) {
60         observersVector.removeElement(observer);
61     }
62
63     public void notifyObservers() {
64         Enumeration<Observer>enumeration = observers();
65         while (enumeration.hasMoreElements()) {
66             ((Observer) enumeration.nextElement()).update();
67         }
68     }
69
70     @SuppressWarnings("unchecked")
71     public Enumeration<Observer> observers() {
72         return ((Vector<Observer>) observersVector.clone()).elements();
73     }
74
75 }
```

上面代码描述第一种形式, 第二种读者可以自己实现, 这里不再赘述。

接下来我们看 java 语言是如何支持观察者模式的, java 提供一个被观察者类 `java.util.Observable` 和一个观察者接口 `java.util.Observer`。

jdk1.6中 API 文档如下描述:

public interface Observer

一个可在观察者要得到 *observable* 对象更改通知时可实现 *Observer* 接口的类。
从以下版本开始:

JDK1.0

另请参见:

Observable

public class Observable extends Object

此类表示模型视图范例中的 *observable* 对象, 或者说“数据”。可将其子类化, 表示应用程序想要观察的对象。

一个 *observable* 对象可以有一个或多个观察者。观察者可以是实现了 *Observer* 接口的任意对象。一个 *observable* 实例改变后, 调用 *Observable* 的 *notifyObservers* 方法的应用程序会通过调用观察者的 *update* 方法来通知观察者该实例发生了改变。

未指定发送通知的顺序。*Observable* 类中所提供的默认实现将按照其注册的重要性顺序来通知 *Observers*, 但是子类可能改变此顺序, 从而使用非固定顺序在单独的线程上发送通知, 或者也可能保证其子类遵从其所选择的顺序。

注意, 此通知机制与线程无关, 并且与 *Object* 类的 *wait* 和 *notify* 机制完全独立。新创建一个 *observable* 对象时, 其观察者集是空的。当且仅当 *equals* 方法为两个观察者返回 *true* 时, 才认为它们是相同的。

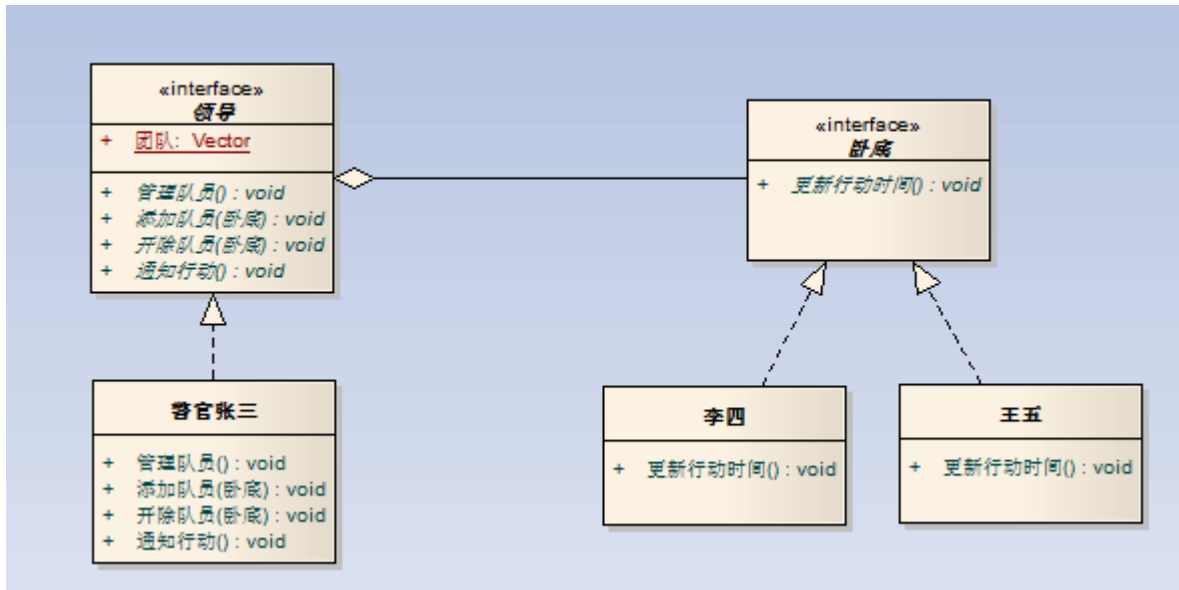
从以下版本开始:

JDK1.0

另请参见:

notifyObservers(), *notifyObservers(java.lang.Object)*, *Observer*,
Observer.update(java.util.Observable, java.lang.Object)

举个例子, 如果你看过 **TVB** 的警匪片, 你就知道卧底的工作方式。一般一个警察可能有几个卧底, 潜入敌人内部, 打探消息, 卧底完全靠他的领导的指示干活, 领导说几点行动, 他必须按照这个时间去执行, 如果行动时间改变, 他也要立马改变自己配合行动的时间。领导派两个卧底去打入敌人内部, 那么领导相当于抽象主题, 而督察警官张三这个人派了两个卧底李四和王五, 张三就相当于具体主题, 卧底相当于抽象观察者, 这两名卧底是李四和王五就是具体观察者, 派的这个动作相当于观察者在主题的登记。那么这个类图如下:



利用 javaAPI 来实现，代码描述如下：

Java 代码

```
76 package observer;
77
78 import java.util.List;
79 import java.util.Observable;
80 import java.util.Observer;
81 /**
82  *
83  *作者: alaric
84  *时间: 2013-8-13下午9:32:40
85  *描述: 警察张三
86  */
87 public class Police extends Observable {
88
89     private String time ;
90     public Police(List<Observer> list) {
91         super();
92         for (Observer o:list) {
93             addObserver(o);
94         }
95     }
96     public void change(String time){
97         this.time = time;
98         setChanged();
99         notifyObservers(this.time);
100     }
101 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

Java 代码   

```
102 package observer;
103
104 import java.util.Observable;
105 import java.util.Observer;
106 /**
107  *
108  *作者: alaric
109  *时间: 2013-8-13下午9:32:59
110  *描述: 卧底 A
111  */
112 public class UndercoverA implements Observer {
113
114     private String time;
115     @Override
116     public void update(Observable o, Object arg) {
117         time = (String) arg;
118         System.out.println("卧底 A 接到消息, 行动时间为: "+time);
119     }
120
121
122 }
```

Java 代码   

```
123 package observer;
124
125 import java.util.Observable;
126 import java.util.Observer;
127 /**
128  *
129  *作者: alaric
130  *时间: 2013-8-13下午9:33:14
131  *描述: 卧底 B
132  */
133 public class UndercoverB implements Observer {
134     private String time;
135     @Override
136     public void update(Observable o, Object arg) {
137         time = (String) arg;
138         System.out.println("卧底 B 接到消息, 行动时间为: "+time);
139     }
140
141
142
143 }
```

Java 代码   

```
144 package observer;
145
146 import java.util.ArrayList;
147 import java.util.List;
148 import java.util.Observer;
149 /**
150  *
151  *作者: alaric
152  *时间: 2013-8-13下午9:32:26
153  *描述: 测试
154  */
155 public class Client {
156
157     /**
158      * @param args
159      */
160     public static void main(String[] args) {
161         UndercoverA o1 = new UndercoverA();
162         UndercoverB o2 = new UndercoverB();
163         List<Observer> list = new ArrayList<>();
164         list.add(o1);
165         list.add(o2);
166         Police subject = new Police(list);
167         subject.change("02:25");
168         System.out.println("=====由于消息败露, 行动时间提前
=====");
169         subject.change("01:05");
170
171     }
172
173 }
```

测试运行结果:

卧底 B 接到消息, 行动时间为: 02:25

卧底 A 接到消息, 行动时间为: 02:25

=====由于消息败露, 行动时间提前=====

卧底 B 接到消息, 行动时间为: 01:05

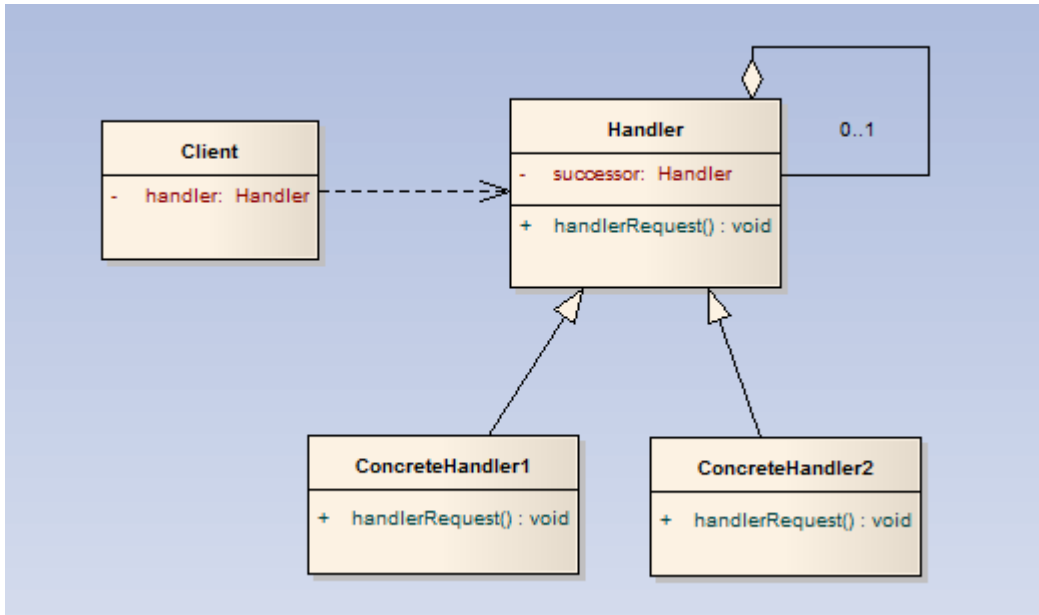
卧底 A 接到消息, 行动时间为: 01:05

观察者模式的优点是只要订阅/登记了之后, 当被观察者改变时, 观察者能自动更新。

跟 JMS 一样, 消息发布者发出消息时, 只要注册过的都会收到消息。

二十、责任链模式

责任链（Chain of Responsibility）模式：责任链模式是对象的行为模式。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受者直接的耦合关系。将这些对象连成一条链，沿着这条链传递该请求，直到有一个对象处理它为止。责任链模式强调的是每一个对象及其对下家的引用来组成一条链，利用这种方式将发送者和接收者解耦，类图如下：



通过上图可以看出责任链模式有两个角色：

抽象处理者（Handler）角色：定义一个请求的接口。如果需要可以定义个一个方法用来设定和返回下家对象的引用。

具体处理者（ConcreteHandler）角色：如果可以处理就处理请求，如果不能处理，就把请求传给下家，让下家处理。也就是说它处理自己能处理的请求且可以访问它的下家。

上述模式的测试代码如下：

Java 代码

```
1 package chainOfResp;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-8-17上午11:01:58
6  *描述: 抽象处理角色
7  */
8 public abstract class Handler {
9
10     protected Handler successor;
11     /**
12     *
13     *作者: alaric
```


Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
14      *时间: 2013-8-17上午11:04:22
15      *描述: 处理方法
16      */
17      public abstract void handlerRequest(String condition);
18
19
20      public Handler getSuccessor() {
21          return successor;
22      }
23      public void setSuccessor(Handler successor) {
24          this.successor = successor;
25      }
26
27 }
```

Java 代码   

```
28 package chainOfResp;
29 /**
30  *
31  *作者: alaric
32  *时间: 2013-8-17上午11:25:54
33  *描述: 具体处理角色
34  */
35 public class ConcreteHandler1 extends Handler {
36
37     @Override
38     public void handlerRequest(String condition) {
39         // 如果是自己的责任, 就自己处理, 负责传给下家处理
40         if(condition.equals("ConcreteHandler1")){
41             System.out.println( "ConcreteHandler1 handled ");
42             return ;
43         }else{
44             System.out.println( "ConcreteHandler1 passed ");
45             getSuccessor().handlerRequest(condition);
46         }
47     }
48
49 }
```

Java 代码   

```
50 package chainOfResp;
51 /**
52  *
53  *作者: alaric
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
54 *时间: 2013-8-17上午11:25:54
55 *描述: 具体处理角色
56 */
57 public class ConcreteHandler2 extends Handler {
58
59     @Override
60     public void handlerRequest(String condition) {
61         // 如果是自己的责任, 就自己处理, 负责传给下家处理
62         if(condition.equals("ConcreteHandler2")){
63             System.out.println( "ConcreteHandler2 handled ");
64             return ;
65         }else{
66             System.out.println( "ConcreteHandler2 passed ");
67             getSuccessor().handlerRequest(condition);
68         }
69     }
70
71 }
```

Java 代码   

```
72 package chainOfResp;
73 /**
74  *
75  *作者: alaric
76  *时间: 2013-8-17上午11:25:54
77  *描述: 具体处理角色
78  */
79 public class ConcreteHandlerN extends Handler {
80
81     /**
82      * 这里假设 n 是链的最后一个节点必须处理掉
83      * 在实际情况下, 可能出现环, 或者是树形,
84      * 这里并不一定是最后一个节点。
85      *
86      */
87     @Override
88     public void handlerRequest(String condition) {
89
90         System.out.println( "ConcreteHandlerN handled");
91     }
92
93
94 }
```

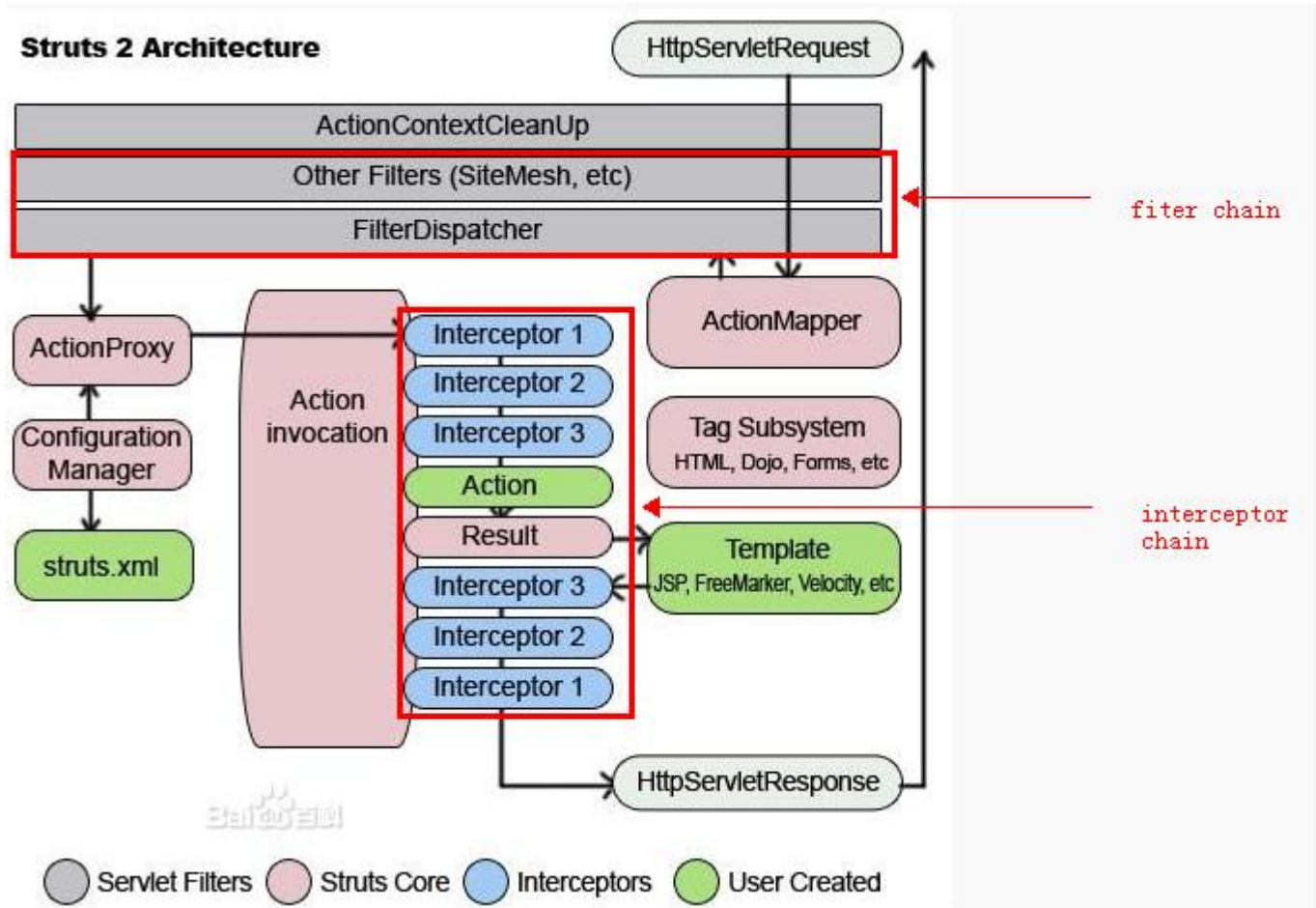
Java 代码   

```
95 package chainOfResp;
96 /**
97  *
98  *作者: alaric
99  *时间: 2013-8-17上午10:59:06
100 *描述: 测试类
101 */
102 public class Client {
103
104     /**
105     *作者: alaric
106     *时间: 2013-8-17上午10:58:58
107     *描述:
108     */
109     public static void main(String[] args) {
110
111         Handler handler1 = new ConcreteHandler1();
112         Handler handler2 = new ConcreteHandler2();
113         Handler handlern = new ConcreteHandlerN();
114
115         //链起来
116         handler1.setSuccessor(handler2);
117         handler2.setSuccessor(handlern);
118
119         //假设这个请求是 ConcreteHandler2的责任
120         handler1.handlerRequest("ConcreteHandler2");
121
122
123     }
124
125 }
```

举这样一个例子，在玩具工厂的生产车间，流水线就是一条责任链，假如一个玩具飞机有外壳装配员，引擎装配员，螺旋桨装配员，模型包装员组成。当这个物件飞机流到谁那里，谁就负责安装他负责的这一部分，这部分安装完成后流到下一个环节，知道所有环境完成。这个是一生成的责任链。还有一个质量检测链，质量检测也分多部，外壳检测，引擎检测，螺旋桨检测，包装检测。当产品留到检测员那里检测自己负责的那一块，如果有问题直接拎出来，如果没问题则传给下一个检测员，直到所有检测完成。这两个都是责任链，但是区别是，生成责任链每个人都会处理，并处理一部分；而质量检测责任链经过判断，要么处理掉，要么不处理流下去。这就是责任链的两种分类，后一种叫做纯的责任链，前一种叫做不纯的责任链，纯的责任链在实际应用中很少存在，常见的为不纯的责任链，上面的模型是模拟纯的责任链来处理的。

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

责任链模式在现实中被使用的很多，常见的就是 OA 系统中的工作流。在 java 中的实际应用有 Servlet 中的过滤器 (Filter)，Struts2 的拦截器 (Interceptor)。Struts2 本身在 Servlet 中也是以 Filter 的形式出现的，所以 Struts2 的结构图中，也可以明显看出 Filter 和 Interceptor 这两条链的存在。



可以看出它们每个节点都可以做一些事情，所以不算一个纯的责任链。

在上面提到了 OA 系统，那么我们再模拟一下 OA 系统中请假审批流程，假如员工直接上司为小组长，小组长直接上司项目经理，项目经理直接上司部门经理，部门经理直接上司总经理。公司规定请假审批如下：

请假时间为 t ，时间单位 day，简写 d：

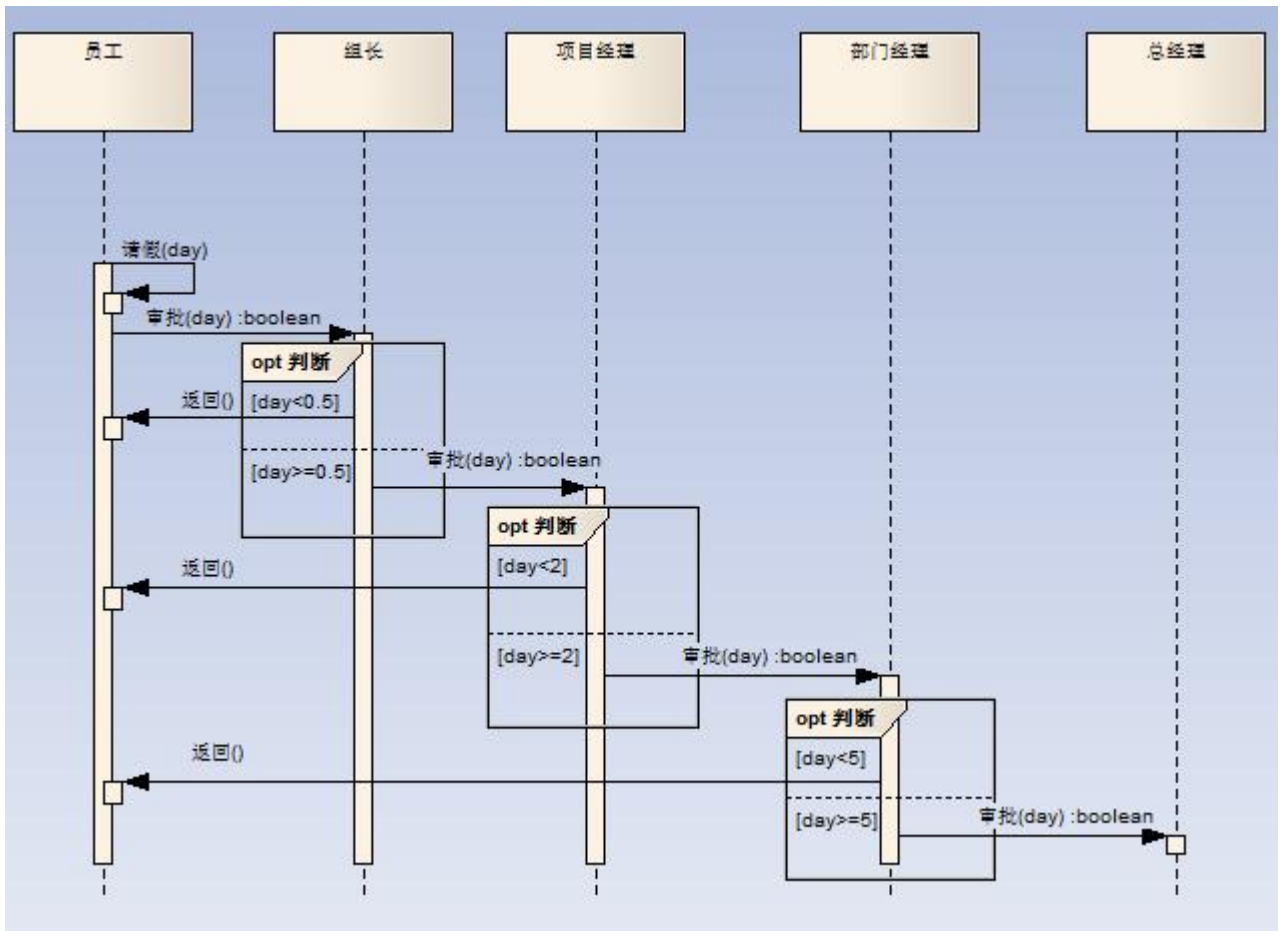
$t < 0.5d$ ，小组长审批；

$t \geq 0.5d, t < 2$ ，项目经理审批；

$t \geq 2, t < 5$ 部门经理审批；

$t \geq 5$ 总经理审批；

审批时序图如下:



用代码描述:

Java 代码

```
126 package chainOfResp. example;
127 /**
128  *
129  *作者: alaric
130  *时间: 2013-8-17下午1:02:51
131  *描述: 审批处理抽象类
132  */
133 public abstract class Handler {
134
135     protected Handler handler;
136
137     /**
138     *
139     *作者: alaric
140     *时间: 2013-8-17下午1:07:40
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
141     *描述: 审批
142     */
143     public abstract boolean approve(double day);
144
145     public Handler getHandler() {
146         return handler;
147     }
148     public void setHandler(Handler handler) {
149         this.handler = handler;
150     }
151
152 }
```

Java 代码   

```
153 package chainOfResp.example;
154
155
156 public class GroupLeader extends Handler {
157
158     @Override
159     public boolean approve(double day) {
160         if(day<0.5) {
161             System.out.println("小组长审批通过");
162             return true;
163         } else {
164             System.out.println("小组长传给了他的上司");
165             return getHandler().approve(day);
166         }
167     }
168
169
170 }
```

Java 代码   

```
171 package chainOfResp.example;
172
173
174 public class ProjectManager extends Handler {
175
176     @Override
177     public boolean approve(double day) {
178         if(day<2) {
179             System.out.println("项目经理审批通过");
180             return true;
181         }
182     }
183 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
181         }else {
182             System.out.println("项目经理传给了他的上司");
183             return getHandler().approve(day);
184         }
185     }
186
187
188 }
```

Java 代码   

```
189 package chainOfResp.example;
190
191
192 public class DepartmentManager extends Handler {
193
194     @Override
195     public boolean approve(double day) {
196         if(day<5){
197             System.out.println("部门经理审批通过");
198             return true;
199         }else {
200             System.out.println("部门经理传给了他的上司");
201             return getHandler().approve(day);
202         }
203     }
204
205
206 }
```

Java 代码   

```
207 package chainOfResp.example;
208
209
210 public class CEO extends Handler {
211
212     @Override
213     public boolean approve(double day) {
214         System.out.println("部门经理审批通过");
215         return true;
216     }
217
218
219 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

Java 代码   

```
220 package chainOfResp.example;
221 /**
222  *
223  *作者: alaric
224  *时间: 2013-8-17下午12:54:51
225  *描述: 测试类, 首先来创建责任链, 然后发出请求模拟员工来请假
226  */
227 public class Client {
228
229     /**
230      *作者: alaric
231      *时间: 2013-8-17下午12:54:44
232      *描述:
233      */
234     public static void main(String[] args) {
235
236         //创建节点
237         GroupLeader gl = new GroupLeader();
238         ProjectManager pm = new ProjectManager();
239         DepartmentManager dm = new DepartmentManager();
240         CEO ceo = new CEO();
241         //建立责任链
242         gl.setHandler(pm);
243         pm.setHandler(dm);
244         dm.setHandler(ceo);
245
246         //向小组长发出申请, 请求审批4天的假期
247         gl.approve(4D);
248
249     }
250 }
251
252 }
```

运行结果:

小组长传给了他的上司

项目经理传给了他的上司

部门经理审批通过

这里模拟的是一个理想的状态, 所以是一个纯的责任链; 在实际当中, 可能小组长签字, 项目经理签字...一堆的签字, 而不是不参与请求的处理。

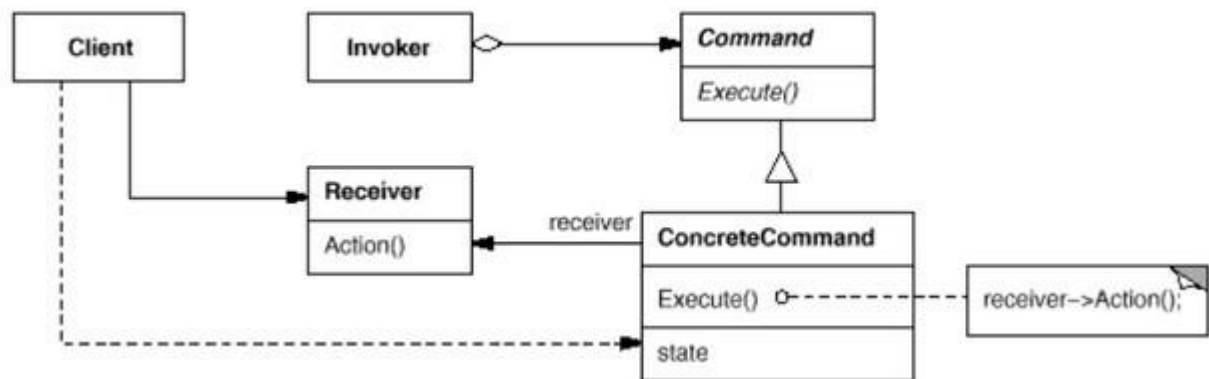
责任链模式的优点是调用者不需知道具体谁来处理请求, 也不知道链的具体结构, 降低了节点

域节点的耦合度；可在运行时动态修改链中的对象职责，增强了给对象指派职责的灵活性；缺点是没有明确的接收者，可能传到链的最后，也没得到正确的处理。

二十一、命令模式

命令（Command）模式：又称 Action 模式或者 Transaction 模式。它属于对象的行为模式。命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和撤销功能。

GoF 命令模式结构图如下：



命令模式是有以下角色：

抽象命令（Command）角色：声明执行操作的接口。

具体命令（ConcreteCommand）角色：将一个接收者对象绑定到一个动作上。调用接收者相应的操作，以实现 Execute 方法。

客户端（Client）角色：创建一个命令对象并设定它的接收者。

请求者（Invoker）角色：负责调用命令对象的执行请求；

接收者（Receiver）角色：负责具体实施和执行一个请求相关的操作。任何一个类都可以作为一个接收者。

上面模型的模拟代码如下：

Java 代码

```
1 package command;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-8-20下午7:39:48
6  *描述: 接收者角色
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
7  */
8  public class Receiver {
9
10     public Receiver() {
11         super();
12     }
13
14     /**
15      *
16      *作者: alaric
17      *时间: 2013-8-20下午7:40:00
18      *描述: 行动方法
19      */
20     public void action() {
21         System.out.println("接收者接到命令, 开始行动");
22     }
23 }
```

Java 代码   

```
24 package command;
25 /**
26  *
27  *作者: alaric
28  *时间: 2013-8-20下午7:36:51
29  *描述: 抽象命令角色
30  */
31 public interface Command {
32
33     /**
34      *
35      *作者: alaric
36      *时间: 2013-8-20下午7:36:40
37      *描述: 执行方法
38      */
39     public void execute();
40 }
```

Java 代码   

```
41 package command;
42 /**
43  *
44  *作者: alaric
45  *时间: 2013-8-20下午7:37:10
46  *描述: 具体命令角色
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
47  */
48  public class ConcreteCommand implements Command {
49
50      private Receiver receiver;
51
52
53      public ConcreteCommand(Receiver receiver) {
54          super();
55          this.receiver = receiver;
56      }
57
58
59      @Override
60      public void execute() {
61
62          receiver.action();
63      }
64  }
65
66
67 }
```

Java 代码   

```
68 package command;
69 /**
70  *
71  *作者: alaric
72  *时间: 2013-8-20下午7:41:07
73  *描述:
74  */
75 public class Invoker {
76
77     private Command command;
78
79     public Invoker(Command command) {
80         super();
81         this.command = command;
82     }
83
84     public void action() {
85         command.execute();
86     }
87 }
```

Java 代码   

```
88 package command;
89 /**
90  *
91  *作者: alaric
92  *时间: 2013-8-20下午7:33:51
93  *描述: 客户端角色
94  */
95 public class Client {
96
97     /**
98     *作者: alaric
99     *时间: 2013-8-20下午7:33:44
100    *描述:
101    */
102    public static void main(String[] args) {
103        Receiver receiver = new Receiver();
104        Command command = new ConcreteCommand(receiver);
105        Invoker invoker = new Invoker(command);
106        invoker.action();
107    }
108
109 }
```

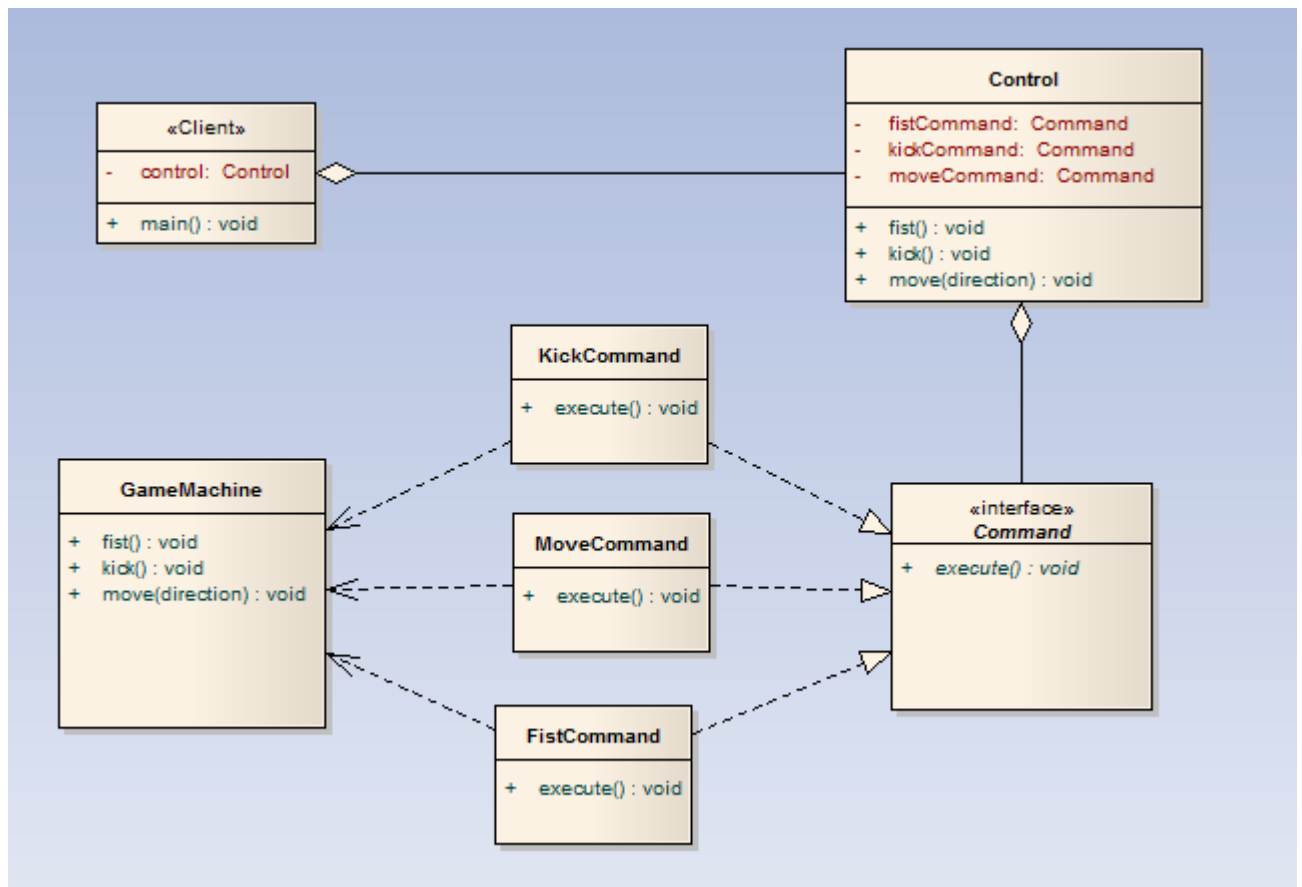
命令模式是对命令的封装,它把发出命令的责任和执行命令的责任分隔开,委派给不同的对象。每一个命令都是一个操作,请求的一方发出请求要求执行一个操作;接收的一方收到请求并执行操作。命令模式允许请求的一方和接收的一方独立开来,使得请求的一方不必知道接收请求的一方接口,更不必知道请求是怎么被接收,以及操作是否被执行,何时被执行,以及是怎么被执行的。命令允许请求方和接收方各自都能独立演化,从而具有以下优点:

- 1、命令模式使新的命令在不改变现有结构代码的情况下很容易被加入到系统里;
- 2、允许接收请求的一方决定是否否决请求;
- 3、能较容易地设计一个命令队列;
- 4、可以容易地实现对请求的 **Undo** 和 **Redo** 操作;
- 5、在需要的情况下以较容易地将命令记入日志。

在讲门面模式的时候曾经提过一个例子,我们小时玩过投币的那种游戏机,一个方向操作杆,四个动作按钮,在一个操作面板上,封装了复杂的逻辑在机器内部,提供简单的操作界面,是一个门面的例子,然而每个操作发出一个命令,来控制游戏人物的运动和各种动作。方向操作杆是一个移动的命令,传入移动的方向和距离作为参数,还有出拳按键发出出拳命令,脚踢按键发出脚踢的命令,那还有组合操作命令,比如下踢腿(操作杆向下和踢脚按钮按下)。现在我

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

们用命令模式来模拟这个场景。类的模型图如下:



如上图所示: **Client** 相当于小时候的我们, **Control** 相关于控制执行器, 我们可以控制踢 (**KickCommand**)、打 (**FistCommand**)、移动 (**MoveCommand**), 这些命令的最终实现者也是接收者是游戏机 (**GameMachine**)。

代码如下:

Java 代码

```
110 package command.example;
111 /**
112  *
113  *作者: alaric
114  *时间: 2013-8-21上午7:15:53
115  *描述: 接收者
116  */
117 public class GameMachine {
118
119     public void fist() {
120         System.out.println("出拳");
121     }
122 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
121     }
122
123     public void kick() {
124         System.out.println("出脚");
125     }
126
127     public void move(String direction) {
128         System.out.println("向"+direction+"移动");
129     }
130
131 }
```

Java 代码   

```
132 package command.example;
133 /**
134  *
135  *作者: alaric
136  *时间: 2013-8-20下午10:20:35
137  *描述: 命令接口
138  */
139 public interface Command {
140
141     //执行方法
142     public void execute();
143
144     //这里还可以加入撤销方法, 回滚方法等
145 }
```

Java 代码   

```
146 package command.example;
147 /**
148  *
149  *作者: alaric
150  *时间: 2013-8-21上午7:17:02
151  *描述: 拳打
152  */
153 public class FistCommand implements Command {
154     private GameMachine machine;
155
156
157
158     public FistCommand(GameMachine machine) {
159         super();
160         this.machine = machine;
161     }
162 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
161     }
162
163
164
165     @Override
166     public void execute() {
167         machine.fist();
168     }
169
170
171 }
```

Java 代码   

```
172 package command.example;
173 /**
174  *
175  *作者: alaric
176  *时间: 2013-8-21上午7:42:21
177  *描述: 脚踢命令
178  */
179 public class KickCommand implements Command {
180     private GameMachine machine;
181
182     public KickCommand(GameMachine machine) {
183         super();
184         this.machine = machine;
185     }
186
187     @Override
188     public void execute() {
189         machine.kick();
190     }
191
192 }
```

Java 代码   

```
193 package command.example;
194 /**
195  *
196  *作者: alaric
197  *时间: 2013-8-21上午7:17:02
198  *描述: 移动命令
199  */
200 public class MoveCommand implements Command {
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
201     private GameMachine machine;
202     private String direction;
203
204
205     public MoveCommand(GameMachine machine, String direction) {
206         super();
207         this.machine = machine;
208         this.direction = direction;
209     }
210
211
212
213     @Override
214     public void execute() {
215         machine.move(direction);
216     }
217
218
219
220 }
```

Java 代码   

```
221 package command.example;
222 /**
223  *
224  *作者: alaric
225  *时间: 2013-8-21上午7:43:14
226  *描述: 控制类
227  */
228 public class Control {
229
230     private Command fistCommand;
231     private Command kickCommand;
232     private Command moveCommand;
233
234     public Control(Command fistCommand, Command kickCommand, Command
moveCommand) {
235         super();
236         this.fistCommand = fistCommand;
237         this.kickCommand = kickCommand;
238         this.moveCommand = moveCommand;
239     }
240
241     public void fist() {
242         fistCommand.execute();
```


Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
243     }
244
245     public void kick() {
246         kickCommand.execute();
247     }
248
249     public void move() {
250         moveCommand.execute();
251     }
252
253 }
```

Java 代码   

```
254 package command.example;
255 /**
256  *
257  *作者: alaric
258  *时间: 2013-8-20下午9:26:42
259  *描述: 客户端角色
260  */
261 public class Client {
262
263     /**
264      *作者: alaric
265      *时间: 2013-8-20下午9:26:36
266      *描述: 测试
267      */
268     public static void main(String[] args) {
269
270         GameMachine machine = new GameMachine();
271         Command fistCommand = new FistCommand(machine);
272         Command kickCommand = new KickCommand(machine);
273         Command moveCommand = new MoveCommand(machine, "左");
274
275         Control control = new Control(fistCommand, kickCommand, moveCommand);
276         control.fist();
277         control.kick();
278         control.move();
279
280         //其实在不同命令模式的情况下就是下面这样直接调用,
281         //就会让调用者和实际命令执行者紧紧耦合在一起, 还有一个好处
282         //就是可以在
283         //machine.fist();
284         //machine.kick();
285         //machine.move("左");
```

```
286     }  
287  
288 }
```

运行结果如下:

出拳

出脚

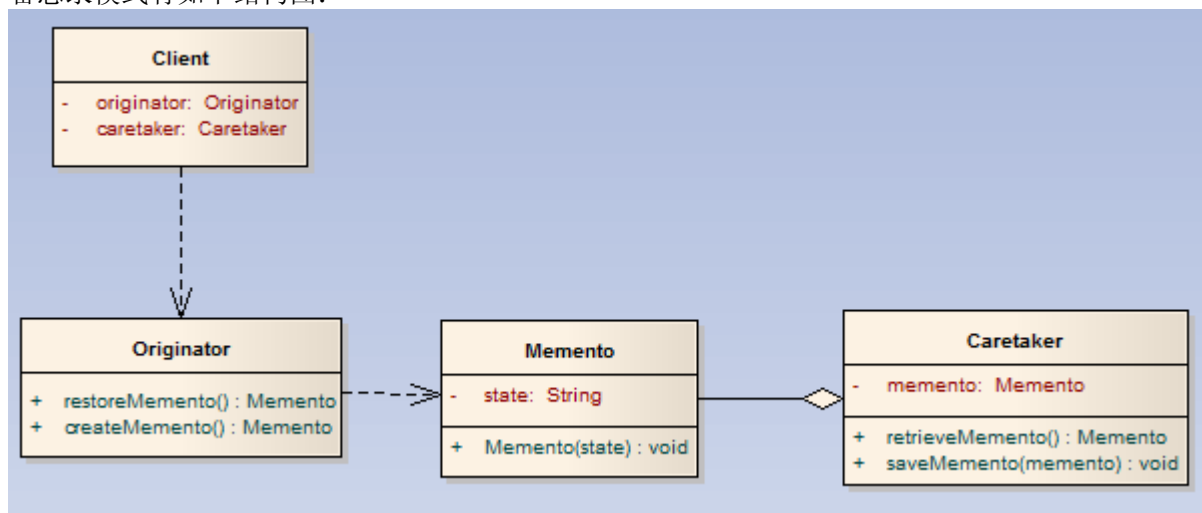
向左移动

通过上面代码可以看出,本来客户端可以直接调用接收者来执行动作的,现在在中间引入了命令,这些命令由调用者(Invoker 这里是 Control)来调用,从而对客户端和命令接收者解耦了,增加了命令后,使得命令除了 `execute` 方法外,可以插入很多其它动作,比如 `redo`, `undo`, 或者记录日志等。

二十二、备忘录模式

备忘录 (Memento) 模式: 又叫做快照模式 (Snapshot Pattern) 或 Token 模式,属于行为模式。在不破坏封闭的前提下,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。

备忘录模式有如下结构图:



备忘录模式涉及角色如下:

发起人 (Originator): 负责创建一个备忘录 Memento,用以记录当前时刻自身的内部状态,并可使用备忘录恢复内部状态。Originator 可以根据需要决定 Memento 存储自己的哪些内部状态。

备忘录(Memento): 负责存储 Originator 对象的内部状态,并可以防止 Originator 以外的其他对象访问备忘录。备忘录有两个接口: Caretaker 只能看到备忘录的窄接口,他只能将备忘录传递给其他对象。Originator 却可看到备忘录的宽接口,允许它访问返回到先前状态所需要的所有

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

数据。

管理者(Caretaker):负责备忘录 Memento，不能对 Memento 的内容进行访问或者操作。

对上面结构图的代码模拟：

Java 代码   

```
1  package memento;
2  /**
3   *
4   *作者: alaric
5   *时间: 2013-8-25下午2:04:27
6   *描述: 备忘录角色
7   */
8  public class Memento {
9
10     private String state;
11
12     public Memento(String state) {
13         this.state = state;
14     }
15
16     public String getState() {
17         return state;
18     }
19
20     public void setState(String state) {
21         this.state = state;
22     }
23
24 }
```

Java 代码   

```
25 package memento;
26 /**
27 *
28 *作者: alaric
29 *时间: 2013-8-25下午2:48:32
30 *描述: 发起人
31 */
32 public class Originator {
33
34     private String state;
35
36     public Memento createMemento() {
37         return new Memento(state);
38     }
39 }
```

```
39
40  /**
41   *
42   *作者: alaric
43   *时间: 2013-8-25下午4:05:39
44   *描述: 还原
45   */
46  public void restoreMemento(Memento memento) {
47      this.state = memento.getState();
48  }
49
50  public String getState() {
51      return state;
52  }
53
54  public void setState(String state) {
55      this.state = state;
56  }
57 }
```

Java 代码   

```
58 package memento;
59 /**
60  *
61  *作者: alaric
62  *时间: 2013-8-25下午2:48:05
63  *描述: 管理者
64  */
65 public class Caretaker {
66
67     private Memento memento;
68
69     /**
70      *
71      *作者: alaric
72      *时间: 2013-8-25下午3:47:18
73      *描述: 取值
74      */
75     public Memento retrieveMemento() {
76         return memento;
77     }
78     /**
79      *
80      *作者: alaric
81      *时间: 2013-8-25下午4:05:01
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
82     *描述: 设置
83     */
84     public void saveMemento(Memento memento) {
85         this.memento = memento;
86     }
87 }
```

Java 代码   

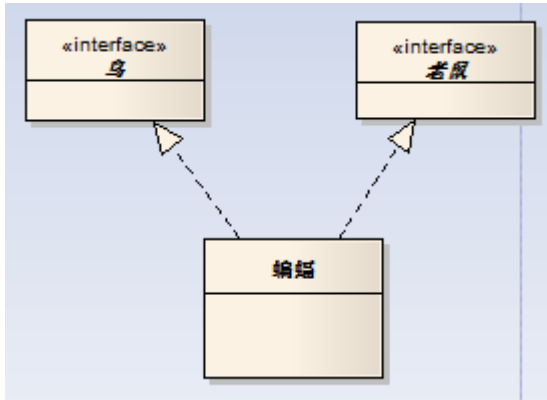
```
88 package memento;
89 /**
90  *
91  *作者: alaric
92  *时间: 2013-8-25下午2:03:49
93  *描述: 测试类
94  */
95 public class Client {
96
97     private static Originator o = new Originator();
98     private static Caretaker c = new Caretaker();
99
100    /**
101     *作者: alaric
102     *时间: 2013-8-25下午2:03:43
103     *描述:
104     */
105    public static void main(String[] args) {
106        //改变发起人的状态
107        o.setState("on");
108        //创建备忘录对象, 并保持于管理保持
109        c.saveMemento(o.createMemento());
110        //改变发起人的状态
111        o.setState("off");
112        //还原状态
113        o.restoreMemento(c.retrieveMemento());
114    }
115
116 }
```

对于上述描述中, 客户端语句 `o.createMemento()` 得到备忘录后, 是可以直接获取备忘录中信息的, 因为备忘录类没有提供窄接口, 这样就破坏了原有的封装性。这种设计备忘录角色的内部所存状态对所有对象是公开的, 所以叫做“白箱”实现。有白箱就有黑箱, “黑箱”的实现方式就是利用 **java** 双接口的方式来隔离不同的对象访问的。

什么是双接口? 就是一个类实现两个接口, 不同的类看到的是不同的类型, 就像蝙蝠一样, 在

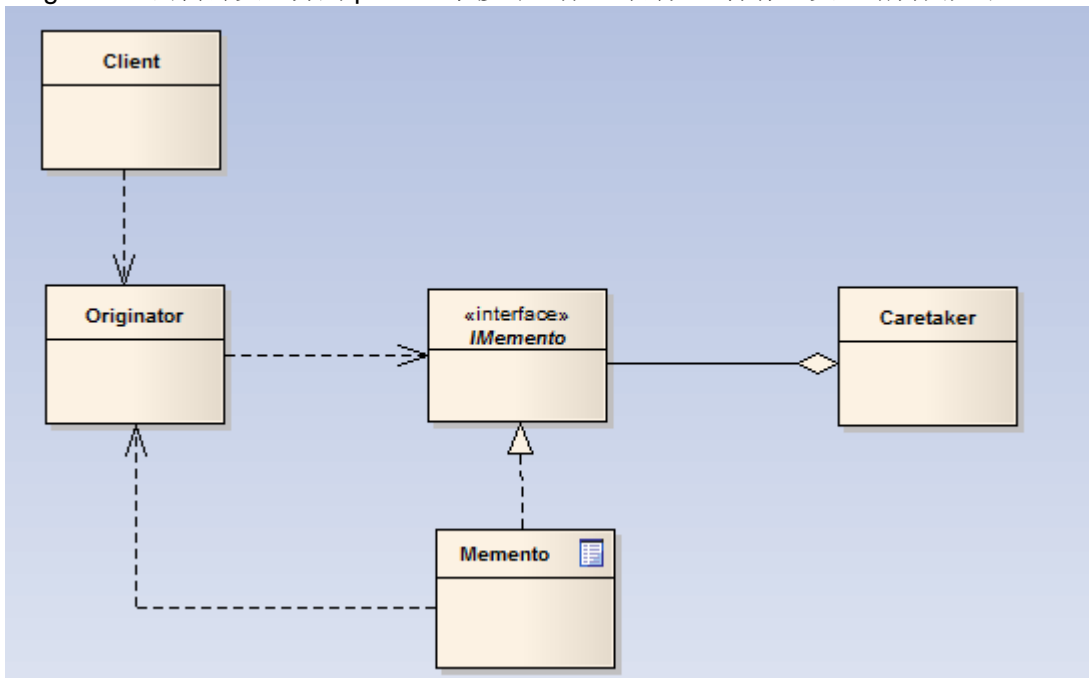
Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

老鼠一块他就展现的是老鼠的接口；在鸟一块就展现的是鸟的接口。



对于备忘录来说实现双接口，给发起人（Originator）角色展现宽接口，给管理者（Caretaker）提供窄接口。宽接口由 Memento 本身就可以展现，窄接口只是个标识接口，不提供任何操作的方法。接下来看看如何用双接口方式“黑箱”实现。

首先以 Memento 以标识接口方式提供给除了发起人角色以外的对象，然后把 Memento 类作为 Originator 的内部类，并用 private 来修饰，保证外部无法操作此类。结构图如下：



描述代码如下：

Java 代码   

```
117 package memento.black;
118 /**
119  *
120  *作者: alaric
121  *时间: 2013-8-25下午6:12:48
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
122 *描述: 标识接口
123 */
124 public interface IMemento {
125
126 }
```

Java 代码   

```
127 package memento.black;
128 /**
129  *
130  *作者: alaric
131  *时间: 2013-8-25下午2:48:32
132  *描述: 发起人
133  */
134 public class Originator {
135
136     private String state;
137
138     /**
139      *
140      *作者: alaric
141      *时间: 2013-8-25下午6:18:36
142      *描述: 穿件备忘录
143      */
144     public IMemento createMemento() {
145         return (IMemento) new Memento(state);
146     }
147
148     /**
149      *
150      *作者: alaric
151      *时间: 2013-8-25下午4:05:39
152      *描述: 还原
153      */
154     public void restoreMemento(IMemento memento) {
155         Memento m = (Memento) memento;
156         setState(m.getState());
157     }
158
159     public String getState() {
160         return state;
161     }
162
163     public void setState(String state) {
164         this.state = state;
165     }
166 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
165         System.out.println("current state:"+state);
166     }
167     /**
168     *
169     *作者: alaric
170     *时间: 2013-8-25下午9:22:02
171     *描述: 内部类
172     */
173     public class Memento implements IMemento{
174
175         private String state;
176
177         public Memento(String state){
178             this.state = state;
179         }
180
181         public String getState() {
182             return state;
183         }
184
185         public void setState(String state) {
186             this.state = state;
187         }
188     }
189 }
```

Java 代码   

```
190 package memento.black;
191 /**
192 *
193 *作者: alaric
194 *时间: 2013-8-25下午2:48:05
195 *描述: 管理者
196 */
197 public class Caretaker {
198
199     private IMemento memento;
200
201     /**
202     *
203     *作者: alaric
204     *时间: 2013-8-25下午3:47:18
205     *描述: 取值
206     */
207     public IMemento retrieveMemento() {
```


Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
208         return memento;
209     }
210     /**
211     *
212     *作者: alaric
213     *时间: 2013-8-25下午4:05:01
214     *描述: 设值
215     */
216     public void saveMemento(IMemento memento) {
217         this.memento = memento;
218     }
219 }
```

Java 代码   

```
220 package memento.black;
221
222
223 /**
224 *
225 *作者: alaric
226 *时间: 2013-8-25下午2:03:49
227 *描述: 测试类
228 */
229 public class Client {
230
231     private static Originator o = new Originator();
232     private static Caretaker c = new Caretaker();
233
234     /**
235     *作者: alaric
236     *时间: 2013-8-25下午2:03:43
237     *描述:
238     */
239     public static void main(String[] args) {
240         //改变发起人的状态
241         o.setState("on");
242         //创建备忘录对象,并保持于管理保持
243         c.saveMemento(o.createMemento());
244
245         //改变发起人的状态
246         o.setState("off");
247         //还原状态
248         o.restoreMemento(c.retrieveMemento());
249     }
250 }
```

251 }

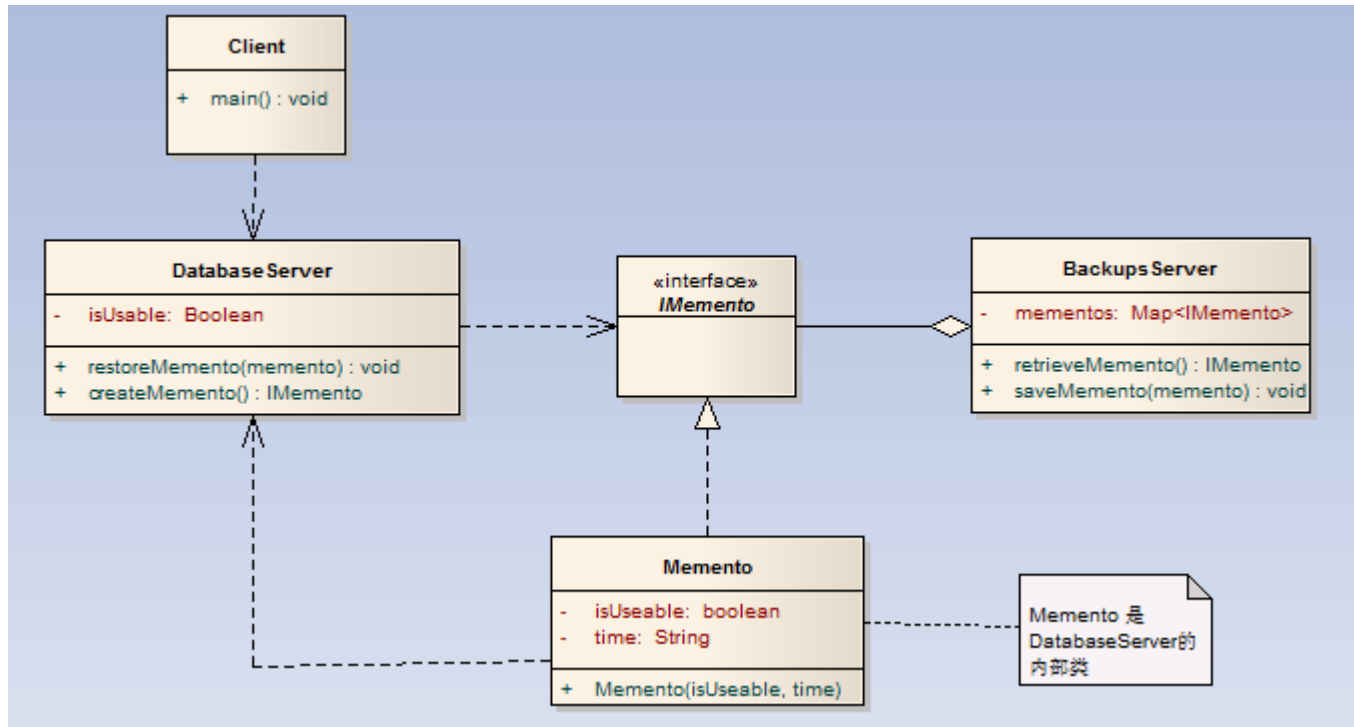
运行结果:

current state:on

current state:off

current state:on

举个栗子, 数据库系统设定一天一个全备, 10分钟一个差备, 当数据库系统出现问题的时候, 就可以还原最近的一个备份。数据库的备份是一个黑箱的, 在没还原之前, 一个备份文件我们看不出里面都是什么样的数据, 所以这里用黑箱实现来描述, 先给出类结构图如下:



描述代码如下:

Java 代码

```
252 package memento.example;
253 /**
254  *
255  *作者: alaric
256  *时间: 2013-8-25下午6:12:48
257  *描述: 标识接口
258  */
259 public interface IMemento {
260
261 }
```

Java 代码   

```
262 package memento.example;
263
264
265 /**
266  *
267  *作者: alaric
268  *时间: 2013-8-25下午2:48:32
269  *描述: 数据库系统 (发起人角色)
270  */
271 public class DatabaseServer {
272
273     private boolean isUseable;
274
275     /**
276      *
277      *作者: alaric
278      *时间: 2013-8-25下午6:18:36
279      *描述: 穿件备忘录
280      */
281     public IMemento createMemento() {
282         return (IMemento) new Memento(isUseable);
283     }
284
285     /**
286      *
287      *作者: alaric
288      *时间: 2013-8-25下午4:05:39
289      *描述: 还原
290      */
291     public boolean restoreMemento(IMemento memento) {
292         Memento m = (Memento) memento;
293         setUseable(m.isUseable());
294         return this.isUseable;
295     }
296
297
298     public boolean isUseable() {
299         return isUseable;
300     }
301
302     public void setUseable(boolean isUseable) {
303         this.isUseable = isUseable;
304         System.out.println("DB state useable is: "+isUseable);
305     }
306 }
```

```
305     }
306
307
308     /**
309     *
310     *作者: alaric
311     *时间: 2013-8-25下午9:22:02
312     *描述: 内部类
313     */
314     public class Memento implements IMemento{
315
316         private boolean isUseable;
317
318         public Memento(boolean isUseable) {
319             super();
320             this.isUseable = isUseable;
321         }
322
323         public boolean isUseable() {
324             return isUseable;
325         }
326
327         public void setUseable(boolean isUseable) {
328             this.isUseable = isUseable;
329         }
330
331     }
332 }
333 }
```

Java 代码   

```
334 package memento.example;
335
336 import java.util.Date;
337 import java.util.Iterator;
338 import java.util.Map;
339 import java.util.concurrent.ConcurrentHashMap;
340
341 /**
342 *
343 *作者: alaric
344 *时间: 2013-8-25下午2:48:05
345 *描述: 备份服务器(管理者)
346 */
347 public class BackupsServer {
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
348 private DatabaseServer dbServer ;//增强管理者的功能，把发起人的操作放在
    这里
349 private Map<Long, IMemento> mementos ;//用一个 map 来对数据库服务多点备份
350
351 public BackupsServer(DatabaseServer dbServer) {
352     super();
353     this.dbServer = dbServer;
354     mementos = new ConcurrentHashMap<>();
355 }
356
357 /**
358  *
359  *作者: alaric
360  *时间: 2013-8-25下午3:47:18
361  *描述: 还原
362  */
363 public void retrieveMemento() {
364     Iterator<Long> it = mementos.keySet().iterator();
365     //还原到最近一个可用的状态
366     while(it.hasNext()) {
367         Long key = it.next();
368         IMemento val = mementos.get(key);
369         boolean isUseable = dbServer.restoreMemento(val);
370         if(isUseable) {
371             break;
372         }
373     }
374 }
375 /**
376  *
377  *作者: alaric
378  *时间: 2013-8-25下午4:05:01
379  *描述: 备份
380  */
381 public void createMemento() {
382     IMemento memento = dbServer.createMemento();
383     this.mementos.put(new Date().getTime(), memento);
384 }
385
386
387 }
```

Java 代码   

```
388 package memento.example;
389
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
390 import java.util.Map;
391
392 import memento.Caretaker;
393
394
395 /**
396  *
397  *作者: alaric
398  *时间: 2013-8-25下午2:03:49
399  *描述: 测试类
400  */
401 public class Client {
402
403
404     private static DatabaseServer dbServer = new DatabaseServer();
405     private static BackupsServer backupServer = new BackupsServer(dbServer);
406
407     /**
408      *作者: alaric
409      *时间: 2013-8-25下午2:03:43
410      *描述:
411      * @throws InterruptedException
412      */
413     public static void main(String[] args) throws InterruptedException {
414         //数据库系统设置可用状态
415         dbServer.setUseable(true);
416         //备份
417         backupServer.createMemento();
418
419         //1秒钟备份一次
420         Thread.sleep(1000);
421         dbServer.setUseable(true);
422         backupServer.createMemento();
423
424         Thread.sleep(1000);
425         dbServer.setUseable(true);
426         backupServer.createMemento();
427
428         Thread.sleep(1000);
429         //设置系统故障
430         dbServer.setUseable(false);
431         //系统故障立即还原到最近一次可用状态
432         System.out.println("-----系统还原-----");
433         backupServer.retrieveMemento();
434     }
435
436 }
```

运行结果:

DB state useable is: true

DB state useable is: true

DB state useable is: true

DB state useable is: false

-----系统还原-----

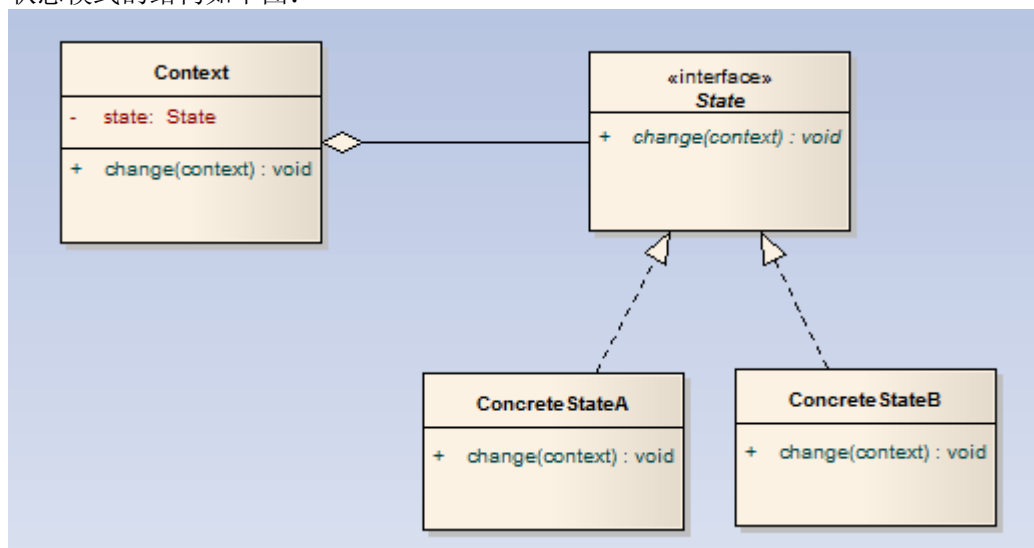
DB state useable is: true

上述代码利用黑箱实现方式模拟了数据库备份还原过程。备忘录模式优点是简化了发起人 (Originator) 类, 它不在需要管理负责备份一个副本在自己内部, 当发起人内部状态失效时, 可以用外部状态来还原。备忘录模式的缺点是完整保存发起人状态在整个过程中, 备忘录角色的资源消耗可能很大, 还有就是管理者不知道资源到底多大, 外部能不能承担, 外部无法预料, 也无法给用户一个提醒。

二十三、状态模式

状态 (state) 模式: 状态模式的意图是, 允许一个对象在其内部状态改变时改变它的行为。看起来就像是改变了它的类一样。主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同的一系列类当中, 可以把复杂的逻辑判断简单化。

状态模式的结构如下图:



从图中可以看出状态模式有以下角色:

1、抽象状态 (State) 角色: 定义一个接口, 用以封装环境对象的一个特定的状态所对应的行

为。

2、具体状态 (ConcreteState) 角色: 每一个具体状态类都实现了环境的一个状态所对的行为。

3、场景 (Context) 角色: 定义客户端所感兴趣的接口, 并且保留一个具体状态类的实例。这个具体状态类的实例给出此环境对象现有的状态。

上图用代码模拟如下:

Java 代码   

```
1 package state;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-9-3下午10:12:05
6  *描述: 抽象状态类
7  */
8 public interface State {
9
10     public void change(Context context);
11
12 }
```

Java 代码   



```
13 package state;
14 /**
15  *
16  *作者: alaric
17  *时间: 2013-9-3下午10:12:27
18  *描述: 实现状态类
19  */
20 public class ConcreteStateA implements State {
21
22     @Override
23     public void change(Context context) {
24         System.out.println("this is ConcreteStateA");
25         context.setState(new ConcreteStateB());
26     }
27 }
```

Java 代码   

```
28 package state;
29 /**
30  *
31  *作者: alaric
32  *时间: 2013-9-3下午10:13:02
```


Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
33  *描述: 实现状态类
34  */
35  public class ConcreteStateB implements State {
36
37      @Override
38      public void change(Context context) {
39          System.out.println("this is ConcreteStateB");
40          context.setState(new ConcreteStateA());
41      }
42  }
```

Java 代码   

```
43  package state;
44  /**
45   *
46   *作者: alaric
47   *时间: 2013-9-3下午10:13:20
48   *描述: 环境角色类
49   */
50  public class Context {
51
52      private State state;
53
54      public void change() {
55          state.change(this);
56      }
57
58
59      public Context(State state) {
60          super();
61          this.state = state;
62      }
63
64
65      public State getState() {
66          return state;
67      }
68
69
70      public void setState(State state) {
71          this.state = state;
72      }
73
74
75  }
```

Java 代码   

```
76 package state;
77 /**
78  *
79  *作者: alaric
80  *时间: 2013-9-3下午10:13:37
81  *描述: 测试类
82  */
83 public class Client {
84
85     /**
86     *作者: alaric
87     *时间: 2013-9-3下午7:52:05
88     *描述:
89     */
90     public static void main(String[] args) {
91
92         State state = new ConcreteStateA();
93         Context context = new Context(state);
94         //初始状态是 A
95         context.change();
96         //装换一次后变成 B
97         context.change();
98         //再转换一次后又变成 A
99         context.change();
100     }
101
102 }
```

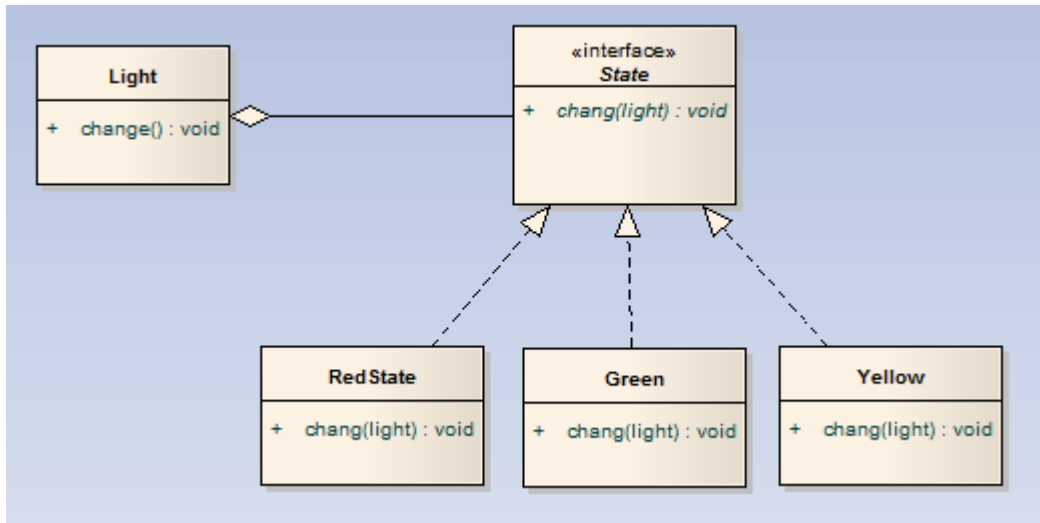
运行结果如下:

this is ConcreteStateA

this is ConcreteStateB

this is ConcreteStateA

上面代码是两个状态切换, 很符合家里灯的开关, A 表示关, B 表示开, 按一下打开, 再按一下关闭。下面举个例子, 马路上的红绿灯大家都知道的, 它有三个状态控制三个不同颜色的灯, 我们分别用 **RedState** (红灯状态), **GreenState**(绿灯状态), **YellowState** (黄灯状态) 表示三个灯的状态, 用 **Light** 表示灯来模拟, 类图如下:



模拟代码如下:

Java 代码

```
103 package state.example;
104 /**
105  *
106  *作者: alaric
107  *时间: 2013-9-7上午11:14:32
108  *描述: 抽象状态类
109  */
110 public interface State {
111
112     public void change(Light light);
113 }
```

Java 代码

```
114 package state.example;
115
116
117 public class GreenState implements State {
118     private static final Long SLEEP_TIME = 2000L;
119     @Override
120     public void change(Light light) {
121
122         System.out.println("现在是绿灯，可以通行");
123         //绿灯亮1秒
124         try {
125             Thread.sleep(SLEEP_TIME);
126         } catch (InterruptedException e) {
127             // TODO Auto-generated catch block
128             e.printStackTrace();
129         }
130     }
131 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
129     }
130     light.setState(new YellowState());
131 }
132
133 }
```

Java 代码   

```
134 package state.example;
135
136
137 public class YellowState implements State {
138     private static final Long SLEEP_TIME = 500L;
139     @Override
140     public void change(Light light) {
141
142         System.out.println("现在是黄灯，警示");
143         //红灯亮0.5秒
144         try {
145             Thread.sleep(SLEEP_TIME);
146         } catch (InterruptedException e) {
147             // TODO Auto-generated catch block
148             e.printStackTrace();
149         }
150         light.setState(new RedState());
151     }
152
153 }
```

Java 代码   

```
154 package state.example;
155
156
157 public class RedState implements State {
158     private static final Long SLEEP_TIME = 1000L;
159     @Override
160     public void change(Light light) {
161
162         System.out.println("现在是红灯，禁止通行");
163         //红灯亮1秒
164         try {
165             Thread.sleep(SLEEP_TIME);
166         } catch (InterruptedException e) {
167             // TODO Auto-generated catch block
168             e.printStackTrace();
169         }
170     }
171 }
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
169     }
170     light.setState(new GreenState());
171 }
172
173 }
```

Java 代码   

```
174 package state.example;
175
176 public class Light {
177
178     private State state;
179
180     private void change() {
181         state.change(this);
182     }
183
184     public void work() {
185         while(true) {
186             change();
187         }
188     }
189     public Light(State state) {
190         super();
191         this.state = state;
192     }
193
194     public State getState() {
195         return state;
196     }
197
198     public void setState(State state) {
199         this.state = state;
200     }
201
202 }
```

Java 代码   

```
203 package state.example;
204 /**
205  *
206  *作者: alaric
207  *时间: 2013-9-7上午11:27:41
208  *描述: 测试客户端
```

```
209 */
210 public class Client {
211
212     /**
213      *作者: alaric
214      *时间: 2013-9-7上午11:27:34
215      *描述:
216      */
217     public static void main(String[] args) {
218
219         //假设路灯开始是绿灯
220         State state = new GreenState();
221         Light light = new Light(state);
222         light.work();
223
224     }
225
226 }
```

运行结果:

现在是绿灯, 可以通行

现在是黄灯, 警示

现在是红灯, 禁止通行

现在是绿灯, 可以通行

现在是黄灯, 警示

现在是红灯, 禁止通行

现在是绿灯, 可以通行

现在是黄灯, 警示

.....

通过上面例子可以看出, 状态模式将与特定状态相关的行为局部化, 并且将不同状态的行为分割开来; 所有状态相关的代码都存在于某个 **ConcreteState** 中, 所以通过定义新的子类很容易地增加新的状态和转换; 状态模式通过把各种状态转移逻辑分散到 **State** 的子类之间, 来减少相互间的依赖。缺点是: 会导致有很多 **State** 的子类。

状态模式和策略模式结构完全一样, 很容易混淆, 这里列举下状态模式和策略模式的区别:

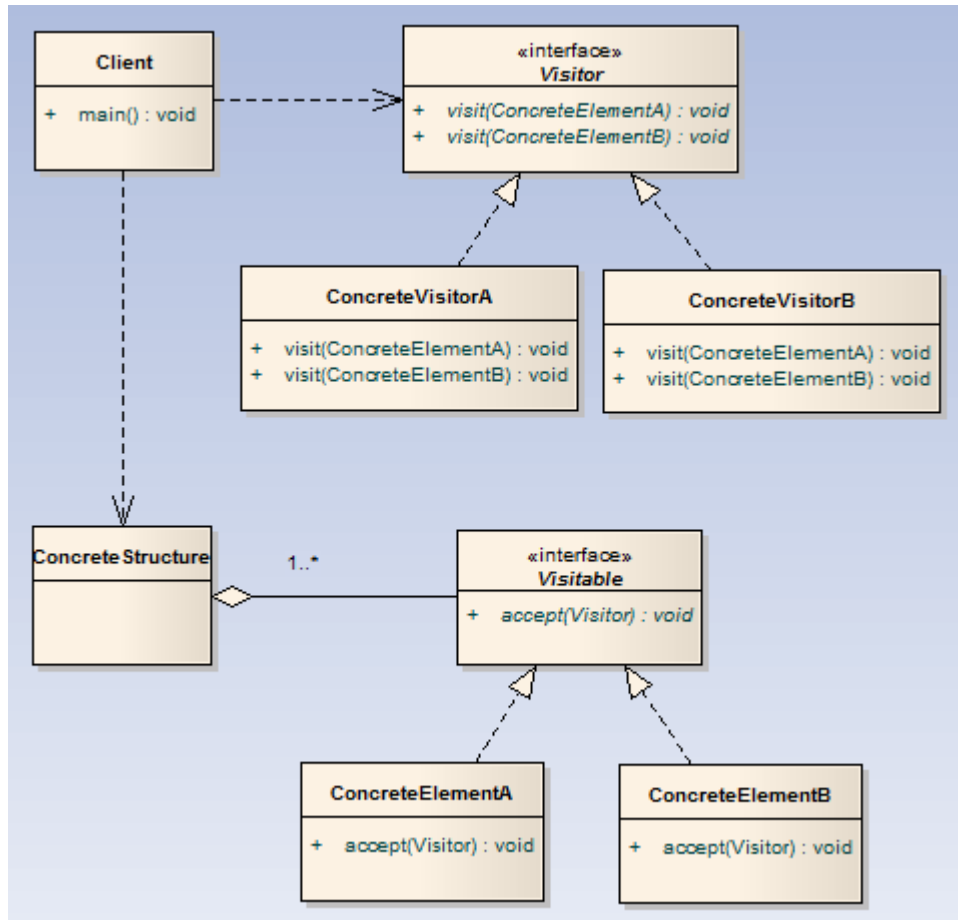
- 1、状态模式有明显的状态过渡, 从一个状态到另一个状态转换过程, 在整个生命周期里有多个状态转换; 而策略模式一旦环境角色选择了一个具体的策略类, 那么在整个生命周期它始终不会改变;
- 2、状态模式多数是外在原因在环境角色中放入一个具体的状态类, 而策略模式是自主选择一个

具体的策略类;

3、状态模式选择一个状态是会明显告诉客户端的,而策略模式则不会告诉客户端选择了什么策略。

二十四、访问者模式

访问者 (Visitor) 模式: 封装一些作用于某种数据结构中的各元素的操作,它可以在不改变这个数据结构的前提下定义作用于这些元素的新的操作。访问者模式的结构图如下:



通过上图可以看到他有如下角色:

抽象访问者 (Visitor) 角色: 定义接口, 声明一个或多个访问操作。

具体访问者 (ConcreteVisitor) 角色: 实现抽象访问者所声明的接口, 也就是抽象访问者所声明的各个访问操作。

抽象元素 (Visitable) 角色: 声明一个接受操作, 接受一个访问者对象作为一个参数。

具体元素结点 (ConcreteElement) 角色: 实现抽象结点所规定的接受操作。

数据结构对象 (ObjectStructure) 角色: 可以遍历结构中的所有元素, 提供一个接口让访问者对象都可以访问每一个元素。

模拟代码如下:

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

Java 代码   

```
1 package visitor;
2 /**
3  *
4  *作者: alaric
5  *时间: 2013-9-13下午11:31:28
6  *描述: 抽象访问者
7  */
8 public interface Visitor {
9
10     public void visit(ConcreteElementB able );
11     public void visit(ConcreteElementA able );
12 }
```

Java 代码   

```
13 package visitor;
14 /**
15  *
16  *作者: alaric
17  *时间: 2013-9-13下午11:31:46
18  *描述: 抽象角色元素
19  */
20 public interface Visitable {
21
22     public void accept(Visitor v);
23
24 }
```

Java 代码   

```
25 package visitor;
26 /**
27  *
28  *作者: alaric
29  *时间: 2013-9-13下午11:33:29
30  *描述: 具体访问者 A
31  */
32 public class ConcreteVisitorA implements Visitor{
33
34     @Override
35     public void visit(ConcreteElementB able) {
36         able.operate();
37     }
38
39     @Override
```


Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
40     public void visit(ConcreteElementA able) {
41         // TODO Auto-generated method stub
42         able.operate();
43     }
44
45
46 }
```

Java 代码   

```
47 package visitor;
48
49 /**
50  *
51  *作者: alaric
52  *时间: 2013-9-13下午11:32:55
53  *描述: 具体访问者 B
54  */
55 public class ConcreteVisitorB implements Visitor{
56
57     @Override
58     public void visit(ConcreteElementB able) {
59         able.operate();
60     }
61
62     @Override
63     public void visit(ConcreteElementA able) {
64         // TODO Auto-generated method stub
65         able.operate();
66     }
67
68
69
70 }
```

Java 代码   

```
71 package visitor;
72 /**
73  *
74  *作者: alaric
75  *时间: 2013-9-13下午11:34:02
76  *描述: 具体元素 A
77  */
78 public class ConcreteElementA implements Visitable {
79
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
80     @Override
81     public void accept(Visitor v) {
82         v.visit(this);
83     }
84
85     public void operate() {
86         System.out.println("ConcreteElementA ....");
87     }
88 }
```

Java 代码   

```
89 package visitor;
90 /**
91  *
92  *作者: alaric
93  *时间: 2013-9-13下午11:33:40
94  *描述: 具体元素 B
95  */
96 public class ConcreteElementB implements Visitable {
97
98     @Override
99     public void accept(Visitor v) {
100         v.visit(this);
101     }
102
103     public void operate() {
104         System.out.println("ConcreteElementB ....");
105     }
106 }
```

Java 代码   

```
107 package visitor;
108
109 import java.util.ArrayList;
110 import java.util.List;
111 /**
112  *
113  *作者: alaric
114  *时间: 2013-9-13下午11:34:22
115  *描述: 客户端
116  */
117 public class Client {
118
119     /**
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
120     * @param args
121     */
122     public static void main(String[] args) {
123
124         Visitor v1 = new ConcreteVisitorA();
125         List<Visitable> list = new ArrayList<>();
126         list.add(new ConcreteElementA());
127         list.add(new ConcreteElementB());
128
129         for(Visitable able :list){
130             able.accept(v1);
131         }
132     }
133 }
134
135 }
```

看了很多设计模式的书，讲访问者设计模式都要提到一个概念“双重分派”，所谓“分派”简单理解就是根据类的特性，特征进行选择，这些选择都是程序语言设计的特征，比如多态（重载，重写）等等，我个人不太注重概念，只要深入掌握面向对象的基础就很好理解了。

设计模式相对其他模式来说结构有点复杂，上面是访问者模式的模拟实现，为了利于学习找了个真实的例子。**dom4j** 里面利用访问者模式来对 **xml** 文档进行逐个节点访问，所有文档的对象之父类接口都是 **Node**，对于不同类型的文档对象又做了不同的抽象，所有可能访问的节点如 **Visitor** 类中所示，**dom4j** 中定义的 **Visitor** 接口如下：

Java 代码   

```
136 /*
137  * Copyright 2001-2005 (C) MetaStuff, Ltd. All Rights Reserved.
138  *
139  * This software is open source.
140  * See the bottom of this file for the licence.
141  */
142
143 package org.dom4j;
144
145 /**
146  * <p>
147  * <code>Visitor</code> is used to implement the <code>Visitor</code>
148  * pattern in DOM4J. An object of this interface can be passed to a
149  * <code>Node</code> which will then call its typesafe methods. Please refer
150  * to the <i>Gang of Four</i> book of Design Patterns for more details on the
151  * <code>Visitor</code> pattern.
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
152 * </p>
153 *
154 * <p>
155 * This <a href="http://www.patterndepot.com/put/8/JavaPatterns.htm">site
156 * has further discussion on design patterns and links to the GOF book. This
157 * <a href="http://www.patterndepot.com/put/8/visitor.pdf">link </a> describes
158 * Visitor pattern in detail.
159 * </p>
160 *
161 * @author <a href="mailto:james.strachan@metastuff.com">James Strachan </a>
162 * @version $Revision: 1.8 $
163 */
164 public interface Visitor {
165     /**
166      * <p>
167      * Visits the given <code>Document</code>
168      * </p>
169      *
170      * @param document
171      *         is the <code>Document</code> node to visit.
172      */
173     void visit(Document document);
174
175     /**
176      * <p>
177      * Visits the given <code>DocumentType</code>
178      * </p>
179      *
180      * @param documentType
181      *         is the <code>DocumentType</code> node to visit.
182      */
183     void visit(DocumentType documentType);
184
185     /**
186      * <p>
187      * Visits the given <code>Element</code>
188      * </p>
189      *
190      * @param node
191      *         is the <code>Element</code> node to visit.
192      */
193     void visit(Element node);
194
195     /**
```

```
196      * <p>
197      * Visits the given <code>Attribute</code>
198      * </p>
199      *
200      * @param node
201      *         is the <code>Attribute</code> node to visit.
202      */
203      void visit(Attribute node);
204
205      /**
206      * <p>
207      * Visits the given <code>CDATA</code>
208      * </p>
209      *
210      * @param node
211      *         is the <code>CDATA</code> node to visit.
212      */
213      void visit(CDATA node);
214
215      /**
216      * <p>
217      * Visits the given <code>Comment</code>
218      * </p>
219      *
220      * @param node
221      *         is the <code>Comment</code> node to visit.
222      */
223      void visit(Comment node);
224
225      /**
226      * <p>
227      * Visits the given <code>Entity</code>
228      * </p>
229      *
230      * @param node
231      *         is the <code>Entity</code> node to visit.
232      */
233      void visit(Entity node);
234
235      /**
236      * <p>
237      * Visits the given <code>Namespace</code>
238      * </p>
239      *
240      * @param namespace
241      *         is the <code>Namespace</code> node to visit.
242      */
```

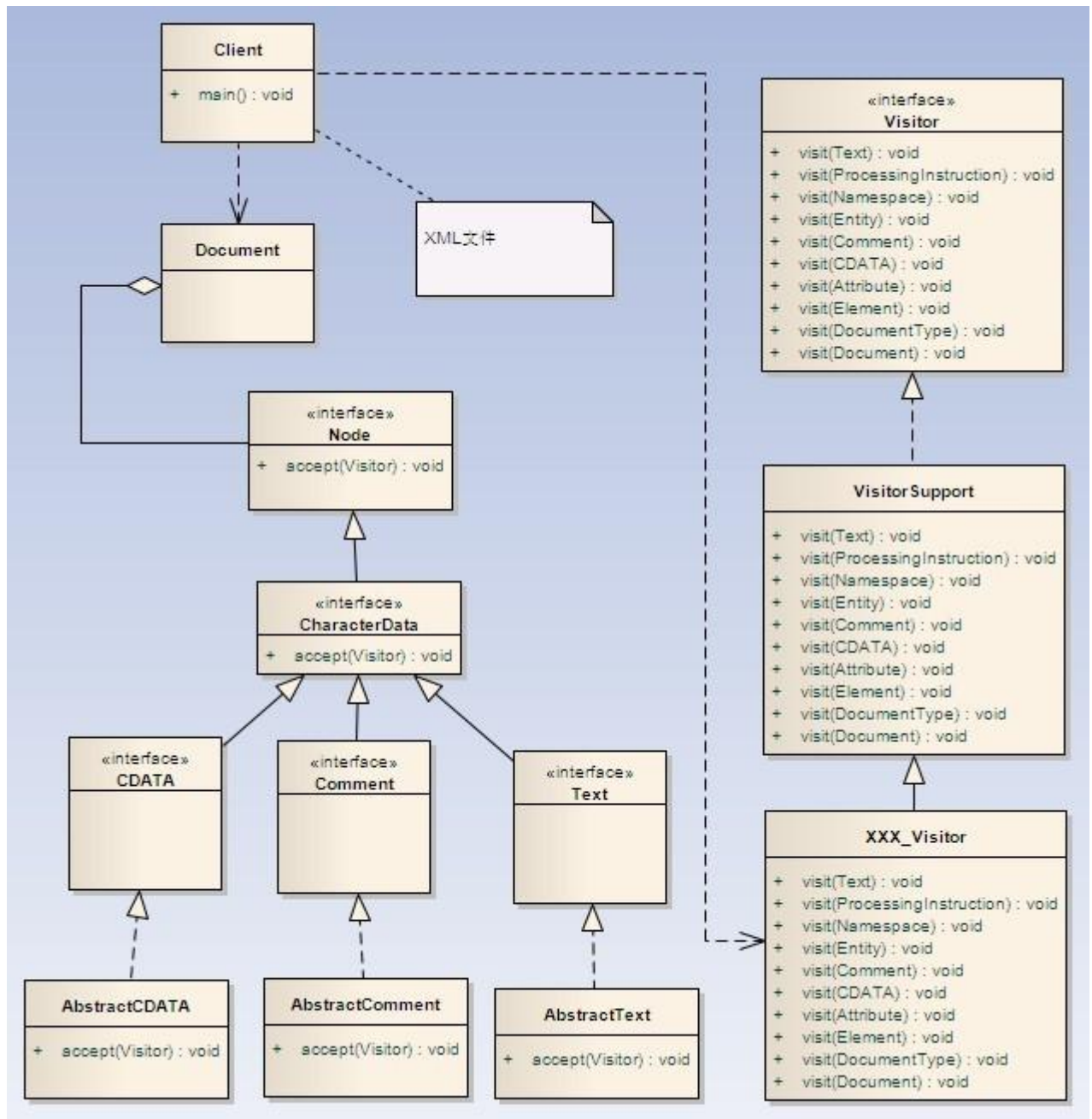
Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
243     void visit(Namespace namespace);
244
245     /**
246      * <p>
247      * Visits the given <code>ProcessingInstruction</code>
248      * </p>
249      *
250      * @param node
251      *         is the <code>ProcessingInstruction</code> node to visit.
252      */
253     void visit(ProcessingInstruction node);
254
255     /**
256      * <p>
257      * Visits the given <code>Text</code>
258      * </p>
259      *
260      * @param node
261      *         is the <code>Text</code> node to visit.
262      */
263     void visit(Text node);
264 }
265
266 /*
267  * Redistribution and use of this software and associated documentation
268  * ("Software"), with or without modification, are permitted provided that the
269  * following conditions are met:
270  *
271  * 1. Redistributions of source code must retain copyright statements and
272  * notices. Redistributions must also contain a copy of this document.
273  *
274  * 2. Redistributions in binary form must reproduce the above copyright notice,
275  * this list of conditions and the following disclaimer in the documentation
276  * and/or other materials provided with the distribution.
277  *
278  * 3. The name "DOM4J" must not be used to endorse or promote products derived
279  * from this Software without prior written permission of MetaStuff, Ltd. For
280  * written permission, please contact dom4j-info@metastuff.com.
281  *
282  * 4. Products derived from this Software may not be called "DOM4J" nor may
283  * "DOM4J" appear in their names without prior written permission of MetaStuff,
284  * Ltd. DOM4J is a registered trademark of MetaStuff, Ltd.
285  *
286  * 5. Due credit should be given to the DOM4J Project - http://www.dom4j.org
287  *
288  * THIS SOFTWARE IS PROVIDED BY METASTUFF, LTD. AND CONTRIBUTORS ``AS IS'' AND
289  * ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
290 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
291 * ARE DISCLAIMED. IN NO EVENT SHALL METASTUFF, LTD. OR ITS CONTRIBUTORS BE
292 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
293 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
294 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
295 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
296 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
297 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
298 * POSSIBILITY OF SUCH DAMAGE.
299 *
300 * Copyright 2001-2005 (C) MetaStuff, Ltd. All Rights Reserved.
301 */
```

dom4j 里面有个缺省的访问者（**Visitor**）的实现 **VisitorSupport**,我们解析一个文档只需继承这个类,然后重写 **visit** 方法即可。一个简单的类图表示 **dom4j** 是怎么利用 **visitor** 设计模式的,如下图:



上图中 **Node** 的继承接口和实现类 还有很多这里就只标示了 **CharacterData** 及其他的子类。

我只需要写 **XXX_Visitor** 就可以了。接下来我们写个例子看看：

我们要解析的 XML 如下：

Java 代码

```
302 <?xml version="1.0" encoding="UTF-8"?>
303 <table name="test">
304   <rows>
305     <row>
306       <id>1</id>
```


Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
307         <test>Test</test>
308     </row>
309     <row>
310         <id>2</id>
311         <test>Test2</test>
312     </row>
313 </rows>
314 </table>
```

我们写个客户端测试, 为了简单, 把 **Visitor** 作为内部类, 直接就一个类完成, 代码如下:

Java 代码   

```
315 package com.alaric.dom4j;
316
317 import java.io.File;
318
319 import org.dom4j.Attribute;
320 import org.dom4j.Document;
321 import org.dom4j.DocumentException;
322 import org.dom4j.Element;
323 import org.dom4j.VisitorSupport;
324 import org.dom4j.io.SAXReader;
325
326 public class Dom4jTest {
327
328
329     public class MyVisitor extends VisitorSupport {
330
331         public void visit(Attribute node) {
332             System.out.println("属性: "+node.getName()+"="+node.getValue());
333         }
334
335         public void visit(Element node) {
336             if(node.isTextOnly()) {
337                 System.out.println("节点: "+node.getName()+"="+node.getText());
338             } else {
339                 System.out.println("节点: "+node.getName());
340             }
341         }
342     }
343
344
345     public static void main(String[] args) throws Exception {
346
347         SAXReader saxReader=new SAXReader();
348         File file=new File("d:\\test.xml");
349         try{
```

Java 设计模式奥妙揭秘: <http://down.51cto.com/zt/240>

```
350         Document doc=saxReader.read(file);
351         doc.accept(new Dom4jTest(). new MyVisitor());
352     }catch(DocumentException de){
353         de.printStackTrace();
354     }
355 }
356 }
357 }
358 }
```

运行结果:

节点: table

属性 : name = test

节点: rows

节点: row

节点: id = 1

节点: test = Test

节点: row

节点: id = 2

节点: test = Test2

可以看出把 xml 节点顺序的访问了一遍。每个人可以根据不同的 xml 来实现自己的 Visitor, 不论怎么写都可以遍历出你所有的节点, 这就是 visitor 的厉害之处。访问者模式也不是万能的, 他的缺点是当数据结构变化时, 他的 visitor 接口及其实现都要改变。所以访问者模式不能使用在经常变化的数据接口上。在 Gof 的设计模式中,有以下情形可以考虑使用设计模式:

- 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。

这些个人看来都是建议, 项目中还要具体问题具体分析了。