设计模式十八篇

设计模式之 Flyweight——打球篇

周末没事，和朋友约好去体育馆（Gymnasium）打球，这个体育馆（Gymnasium）提供各种球种，因为我们人多，因此选择了羽毛球（badminton），兵乓球（pingpangball）、排球（vollyball）等球种，我们首先要填写选球清单（playBallList），然后去器材部领球。

下面我们看看该如何实现这个过程呢？

1、我们首先先定义玩球（PlayBall）这样一个接口类：

```
public interface PlayBall {
      public void playBall( String ballName); //玩球
}
```

2、玩具体的球（PlayConcreteBall）是对玩球（PlayBall）接口的具体实现：

```
public class PlayConcreteBall implements PlayBall{
      public void playBall(String concreteBallName) {
          System.out.println("玩"+concreteBallName+"！");
    }
}
```

3、定义体育馆（Gymnasium）类：

```
public class Gymnasium {
  private Hashtable playBallList = new Hashtable();  //体育馆根据我们的需要填写的选球清单

  public PlayBall getPlayBall(Object key) {  //得到要玩的球
    PlayBall playBall = (PlayBall) playBallList.get(key); //根据清单获得所需的球
    if (playBall == null) {  //清单上没有这种球
                        //虽然清单上没有这种球，但还想玩，那你先得到这种球，然后补清单
      playBall = new PlayConcreteBall();
      playBallList.put(key, playBall);   //将这种球补写进清单
    }
    return playBall;
  }

  public Hashtable getPlayBallList() {  //获得选球清单
     return playBallList;
  }
}
```

4、编写测试类：

```
public class Test {
```

```
  public static void main(String args[]) {
    Gymnasium gymnasium = new Gymnasium();   //我们去的体育馆
    PlayBall badminton = gymnasium.getPlayBall( "羽毛球" );   //想得到羽毛球
    PlayBall pingpangball = gymnasium.getPlayBall( "乒乓球" ); //想得到乒乓球
    PlayBall vollyball = gymnasium.getPlayBall( "排球" );    //想得到排球

    Hashtable selectedBallList = gymnasium.getPlayBallList(); //器材部得到选球清单

    ((PlayBall)selectedBallList.get("羽毛球")).playBall("羽毛球"); //得到羽毛球
    ((PlayBall)selectedBallList.get("乒乓球")).playBall("乒乓球"); //得到乒乓球
    ((PlayBall)selectedBallList.get("排球")).playBall("排球");  //得到排球
  }
}
```

5、说明：

A：Flyweight 定义：避免大量拥有相同内容的小类的开销(如耗费内存),使大家共享一个类(元类)。

B：从本例中我们可以看到通过选球清单，我们获得了所需的球种，因此关键点就是填写好这张选球清单，其实 Flyweight 的重点也就在这里。

设计模式之 Bridge——游戏篇

今天从电子市场买来两张游戏碟，一张是三国游戏（SanGuoGame），一张是 CS 游戏（CSGame）。玩过游戏的人可能都知道，游戏对计算机系统（ComputerSystem）是有要求的，可能一个游戏在 Windows98 系统下能玩，到了 Windows2000 系统下就不能玩了，因此为了不走冤枉路，先看看游戏要求的计算机系统（ComputerSystem）可是一个好习惯呦！

好了，闲话少叙开始我们的游戏旅程吧：

1、在这里，先定义计算机系统（ComputerSystem）接口类：

```
public interface ComputerSystem {
    public abstract void playGame(); //玩游戏
}
```

2、再定义对计算机系统（ComputerSystem）接口的具体实现类：

A：Windows98 系统

```
public class Windows98 implements ComputerSystem{
  public void  playGame(){
    System.out.println("玩 Windows98 游戏！");
  }
}
```

B：Windows2000 系统

```
public class Windows2000 implements ComputerSystem{
  public void  playGame(){
    System.out.println("玩 Windows2000 游戏！");
  }
}
```

3、 定义游戏（Game）类：

```
public abstract class  Game  {
  public abstract void play(); //玩游戏

  protected ComputerSystem getSetupSystem(String type) { //获得要安装的系统
    if (type.equals("Windows98")) { //如果游戏要求必须安装在 Windows98 下
      return new Windows98(); //使用 Windows98 系统
    }
    else if (type.equals("Windows2000")) { //如果游戏要求必须安装在 Windows2000 下
      return new Windows2000(); //使用 Windows2000 系统
    }
    else {
      return new Windows98(); //默认启动的是 Windows98 系统
    }
  }
}
```

4、 定义游戏（Game）类的子类：

A：三国游戏（SanGuoGame）

```
public class SanGuoGame extends Game {
  private ComputerSystem computerSystem;
  public SanGuoGame(String type) {//看游戏要求安装在那个系统上
    computerSystem = getSetupSystem(type);//那么使用这个系统
  }

  public void play() {//玩游戏
    computerSystem.playGame();
    System.out.println("我正在玩三国，不要烦我！");
  }
}
```

B：CS 游戏（CSGame）

```
public class CSGame extends Game {
  private ComputerSystem computerSystem;
  public CSGame(String type) { //看游戏要求安装在那个系统上
    computerSystem = getSetupSystem(type); //那么使用这个系统
  }

  public void play() { //玩游戏
    computerSystem.playGame();
    System.out.println("我正在玩 CS，不要烦我！");
  }
}
```

5、编写测试类：

```
public class Test  {
  public static void main(String[] args) {
    Game sanguo = new SanGuoGame("Windows98"); //游戏要求 Windows98
    sanguo.play();

    sanguo = new SanGuoGame("Windows2000");//游戏要求 Windows2000
    sanguo.play();

    Game cs = new CSGame("Windows98"); //游戏要求 Windows98
    cs.play();

    cs = new CSGame("Windows2000");//游戏要求 Windows2000
    cs.play();
  }
}
```

运行结果：

```
玩 Windows98 游戏!
我正在玩三国，不要烦我!
玩 Windows2000 游戏!
我正在玩三国，不要烦我!
玩 Windows98 游戏!
我正在玩 CS，不要烦我!
玩 Windows2000 游戏!
我正在玩 CS，不要烦我!
```

6、说明：

A：Bridge 定义：将抽象和行为划分开来,各自独立,但能动态的结合。

B：从本例中我们可以看到，不同的游戏对系统的要求是不同的，三国和 CS 可能都需要 Windows98 系统，也可能需要 Windows2000 系统，甚至要求不相同的系统，因此处理这类问题，我们就可以用 Bridge 这种模式了。

C：这里行为是指游戏，抽象是指系统!

设计模式之 Chain of Responsibility——项目篇

最近单位接到一个软件项目，要求在规定的时间内完成，因此我们项目组成员就开始忙活了，我们都知道机械加工是有工序（Procedure）要求的，其实我们软件开发也是有工序（Procedure）要求的，即首先先由分析设计人员对系统进行分析设计，然后再由程序员进行编码，最后再由测试人员对整个系统进行测试。有人可能会说，我就偏不这样，我就要先编码，再分析设计，当然了，你要这样做我也没办法，不过你要真这么做，嘿嘿，我想你最后可要吃苦头的，不信你就试试看。

好了，闲话少叙，我们开始吧：

1、我们先定义工序（Procedure）这样一个接口类：

```
public interface Procedure  {
    public abstract void nextProcedure(Procedure procedure);  //下一工序
```

```
    public abstract void executeProcedure(String aim);  //执行工序
}
```

2、定义工序（Procedure）接口的实现类：

A：分析设计工序（DesignProcedure）

```
public class DesignProcedure implements Procedure {
    private Procedure nextProcedure = null;
    private String procedureName = "Design";    //该工序名

    public void nextProcedure(Procedure procedure) { //下一工序
        nextProcedure = procedure;
    }

    public void executeProcedure(String currentProcedure) { //执行工序
        if(currentProcedure.equals(procedureName)) {//如果当前工序和该工序相符
            System.out.println("进行系统分析设计");
        } else {
            if(nextProcedure != null) { //如果当前工序和该工序不相符则转入下一工序
                nextProcedure.executeProcedure(currentProcedure);
            }
        }
    }
}
```

B：编码工序（CodeProcedure）

```
public class CodeProcedure implements Procedure {
    private Procedure nextProcedure = null;
    private String procedureName = "Code"; //该工序名

    public void nextProcedure(Procedure procedure) { //下一工序
        nextProcedure = procedure;
    }

    public void executeProcedure(String currentProcedure) { //执行工序
        if(currentProcedure.equals(procedureName)) {//如果当前工序和该工序相符
            System.out.println("进行编码工作");
        } else {
            if(nextProcedure != null) { //如果当前工序和该工序不相符则转入下一工序
                nextProcedure.executeProcedure(currentProcedure);
            }
        }
    }
}
```

C：测试工序（TestProcedure）

```
public class TestProcedure implements Procedure {
  private Procedure nextProcedure = null;
   private String procedureName = "Test"; //该工序名

   public void nextProcedure(Procedure procedure) { //下一工序
       nextProcedure = procedure;
   }

   public void executeProcedure(String currentProcedure) { //执行工序
       if(currentProcedure.equals(procedureName)) {//如果当前工序和该工序相符
           System.out.println("进行系统测试");
       } else {
           if(nextProcedure != null) { //如果当前工序和该工序不相符则转入下一工序
               nextProcedure.executeProcedure(currentProcedure);
           }
       }
   }
}
```

3、编写测试类：

```
public class test {
  public static void main(String[] args) {
       DesignProcedure design = new DesignProcedure();  //分析设计工序
       CodeProcedure code = new CodeProcedure();  //编码工序
       TestProcedure test = new TestProcedure();  //测试工序

       design.nextProcedure(code);  //定义分析设计工序的下一工序
       code.nextProcedure(test);   //定义编码工序的下一工序

       design.executeProcedure("Design"); //开始执行工序
       design.executeProcedure("Code");
       design.executeProcedure("Test");
   }
```

4、说明：

A： Chain of Responsibility 定义：Chain of Responsibility 模式是用一系列类(classes)试图处理一个请求 request，这些类之间是一个松散的耦合，唯一共同点是在他们之间传递 request.，也就是说，来了一个请求，A 类先处理，如果没有处理，就传递到 B 类处理，如果没有处理，就传递到 C 类处理，就这样象一个链条 (chain)一样传递下去。

B：在本例中，分析设计工序（DesignProcedure）、编码工序（CodeProcedure）和测试工序（TestProcedure）中的 executeProcedure 执行工序方法中都对当前工序和该工序进行了判断，如果当前工序和该工序相符就执行该工序，如果当前工序和该工序不相符则转入下一工序执行。

C：嘿嘿，其实说白了就是：是你的工作你干，不是你的工作你就不要干，顺序往下传该谁干谁干就是了。

设计模式之 Mediator——联通篇

中午吃完饭没事，我（133 用户）就和同事张三（130 用户）、李四（131 用户）一块去高新联通大厅（gaoxinLianTongHall）交手机费。到了高新联通大厅（gaoxinLianTongHall），我们发现因为是中午吃饭时间大厅里只有一个工作人员，因此我们只好一个一个来办理交费业务了，首先是张三（130 用户），然后是李四（131 用户），最后是我（133 用户）。

好了，让我们看看这个过程如何来实现呢：

1、我们先定义联通大厅（LianTongHall）这样一个接口类：

```
public interface LianTongHall {
     public void identifyUserType(LianTongUser user, String type); //判断用户类型
     public void getUserMoney(String type); //获得用户交的钱
}
```

2、定义联通大厅（LianTongHall）的具体实现类：

```
public class ConcreteLianTongHall implements LianTongHall {
  private User130 user130;
  private User131 user131;
  private User133 user133;

  public void identifyUserType(LianTongUser user, String type) {
    if (type == "130") {
      user130 = (User130) user; //130 用户
    }
    else if (type == "131") {
      user131 = (User131) user; //131 用户
    }
    else if (type == "133") {
      user133 = (User133) user; //133 用户
    }
  }

  public void  getUserMoney(String type) {  //得到用户交的钱
    if (type == "130") {
      user131.pleaseWait();  //131 用户请先等
      user133.pleaseWait();  //133 用户请先等
    }
    else if (type == "131") {
      user130.pleaseWait();   //130 用户请先等
      user133.pleaseWait();   //133 用户请先等
    }
    else if (type == "133") {
```

```
        user130.pleaseWait();  //130 用户请先等
        user131.pleaseWait();  //131 用户请先等
      }
    }
}
```

3、定义联通用户（LianTongUser）接口类：

```
public interface LianTongUser {
  public void HandInMoney();  //交钱
  public void pleaseWait();   //等待
}
```

4、定义联通用户（LianTongUser）接口的实现类：

A：130 用户（User130）

```
public class User130 implements LianTongUser {
  private final String type = "130";
  private LianTongHall liantongHall;

  public User130(LianTongHall liantongHall) {
      this.liantongHall = liantongHall;
      liantongHall.identifyUserType(this, type);  //联通大厅判断是那种用户
  }
  public void HandInMoney() {
      System.out.println("130 用户正在交钱！");
      liantongHall.getUserMoney(type);  //联通大厅得到用户交的钱
  }
  public void pleaseWait() {
      System.out.println("130 用户请先等一会！");
  }
}
```

B：131 用户（User131）

```
public class User131 implements LianTongUser {
  private final String type = "131";
  private LianTongHall liantongHall;

  public User131(LianTongHall liantongHall) {
      this.liantongHall = liantongHall;
      liantongHall.identifyUserType(this, type); //联通大厅判断是那种用户
  }
  public void HandInMoney() {
      System.out.println("131 用户正在交钱！");
      liantongHall.getUserMoney(type); //联通大厅得到用户交的钱
  }
```

```
  public void pleaseWait() {
      System.out.println("131 用户请先等一会！");
  }
}
```

C：133 用户（User133）

```
public class User133 implements LianTongUser {
  private final String type = "133";
  private LianTongHall liantongHall;

  public User133(LianTongHall liantongHall) {
      this.liantongHall = liantongHall;
      liantongHall.identifyUserType(this, type); //联通大厅判断是那种用户
  }
  public void HandInMoney() {
      System.out.println("133 用户正在交钱！");
      liantongHall.getUserMoney(type); //联通大厅得到用户交的钱
  }

  public void pleaseWait() {
      System.out.println("133 用户请先等一会！");
  }
}
```

5、编写测试类：

```
public class Test  {
    public static void  main(String[] args) {
        LianTongHall gaoxinLianTongHall = new ConcreteLianTongHall(); //高新联通大厅
        User130 zhangsan = new User130(gaoxinLianTongHall); //张三
        User131 lisi = new User131(gaoxinLianTongHall);   //李四
        User133 me = new User133(gaoxinLianTongHall);   //我

        zhangsan.HandInMoney();  //张三交钱
        lisi.HandInMoney();  //李四交钱
        me.HandInMoney();   //我交钱
    }
}
```

6、说明：

A：Mediator 定义：用一个中介对象来封装一系列关于对象交互行为。

B：每个成员都必须知道中介对象，并且和中介对象联系，而不是和其他成员联系。

C：在本例中，中介对象就相当于我们的联通大厅，我们都是和联通大厅发生关系，张三、李四和我之间是没有交钱关系的。

设计模式之 Strategy——简历篇

表弟马上就要大学毕业，找工作要写简历（Resume），因此他就来问我关于这方面的问题。我告诉他最好写两种类型的简历，一种是用中文写的，一种是用英文写的，如果是国企的话，就投中文简历（ChineseResume），如果是外企的话，就投英文简历（EnglishResume），嘿嘿，原因在这里就没必要多说了吧。

下面让我们看看这个过程该如何实现呢？

1、我们先定义简历（Resume）接口类：

```
public interface Resume {
      public void writeText();
}
```

2、再定义对简历（Resume）接口的具体实现：

A：中文简历（ChineseResume）

```
public class ChineseResume implements Resume{
      public void writeText(){
            System.out.println("用中文写简历！");
    }
}
```

B：英文简历（EnglishResume）

```
public class EnglishResume implements Resume{
  public void writeText(){
    System.out.println("用英文写的简历！");
  }
}
```

3、定义投递策略（Strategy）类：

```
public class Strategy {
  private Resume resume;
  public Strategy(Resume resume) { //使用简历的策略
    this.resume=resume;
  }
  public void postResume() { //投递简历
    System.out.println("投递");
    resume.writeText();
  }
}
```

4、编写测试类：

```
public class Test {
  public static void main(String args[]) {
    //如果是国企
```

```
        Resume brotherResume = new ChineseResume();  //表弟用中文写的简历
        Strategy strategy = new Strategy(brotherResume); //使用用中文写的简历
        strategy.postResume();  //给国企投递该简历

        //如果是私企
        brotherResume = new EnglishResume();   //表弟用英文写的简历
        strategy = new Strategy(brotherResume);  //使用用英文写的简历
        strategy.postResume();  //给私企投递该简历
    }
}
```

5、说明：

A：Strategy 模式主要是定义一系列的算法，把这些算法一个个封装成单独的类。

B：在本例中，中文简历（ChineseResume）和英文简历（EnglishResume）就相当于两种算法，同时我们把它定义成两个单独的类。

C：在找工作时，我们可以根据企业类型选择投递那种简历，Strategy 模式和 Factory 模式的不同之处是：Strategy 模式主要是用来选择不同的算法，而 Factory 模式的重点是用来创建对象。

设计模式之 Observer——公交篇

说到公交车，我想大家都不陌生吧，坐过公交车的朋友可能都知道，一般公交车上都有售票员（BusConductor），当然无人售票车要除外了。售票员（BusConductor）除了收取乘客（Passenger）的车费还起着监控的作用。

下面让我们看看这个过程该如何实现呢？

1、我们先定义售票员（BusConductor）接口类：

```
public interface BusConductor {
    public void getCurrentPassenger(Passenger passenger);  //获得当前乘客情况
}
```

2、再定义对售票员（BusConductor）接口的具体实现：

```
public class ConcreteBusConductor implements BusConductor{
    private Vector vectorBus;  //公交车 vectorBus
    private Passenger passenger;

    public ConcreteBusConductor(Passenger passenger) {
        this.passenger=passenger;
    }

    public void getCurrentPassenger(Passenger passenger) {
        vectorBus = passenger.getCurrentPassenger();  //获得当前的乘客情况
        for(int i = 0; i < vectorBus.size(); i++) {
            System.out.println("公交车上有:" + (String)vectorBus.get(i));
        }
    }
```

```
  public void findPassengerChange(String action, String str) { //公交车乘客变化
      passenger.setCurrentPassenger(action, str);
  }

  public void observeResult() {  //观察到的情况
      passenger.showPassengerInfo();
  }
}
```

3、定义乘客（Passenger）接口类：

```
public interface Passenger {
  public abstract void attach(BusConductor busConductor);  //将乘客和售票员关联起来
  public abstract void  showPassengerInfo();  //传递乘客情况
  public abstract Vector getCurrentPassenger();  //获得当前乘客情况
  public abstract void  setCurrentPassenger(String act, String str);  //设置当前乘客情况
}
```

4、定义对乘客（Passenger）接口的具体实现：

```
public class ConcretePassenger implements Passenger{
  private List observerList; //观察者列表
  private Vector vectorBus;  //公交车 vectorBus

  public ConcretePassenger() {
    observerList =  new ArrayList();
    vectorBus = new Vector();
  }

  public void attach(BusConductor busConductor) {
      observerList.add(busConductor);   //将公交车售票员增加到观察者列表
  }

  public void showPassengerInfo() {
      for(int i = 0; i < observerList.size(); i++) {
              //使公交车售票员获得当前乘客情况
          ((BusConductor)observerList.get(i)).getCurrentPassenger(this);
      }
  }

  public void setCurrentPassenger(String act, String str) {
      if(act.equals("up")) {  //乘客上车
          vectorBus.add(str);  //将该乘客增加到公交车 vectorBus 中
      } else if(act.equals("down")) {  //乘客下车
          vectorBus.remove(str);  //将该乘客从公交车 vectorBus 中删除
```

```
        }
    }

    public Vector getCurrentPassenger() {  //获得当前乘客情况
        return vectorBus;
    }
}
```

5、编写测试类：

```
public class Test  {
    public static void main(String[] args) {
        Passenger passenger = new ConcretePassenger();
        ConcreteBusConductor busConductor = new ConcreteBusConductor(passenger);
        passenger.attach(busConductor);  //将公交车车售票员和乘客联系起来

        //公交车售票员观察到的情况
        System.out.println("公交车售票员观察到的情况:");
        passenger.setCurrentPassenger("up", "乘客张三");  //上来乘客张三
        passenger.setCurrentPassenger("up", "乘客李四");  //上来乘客李四
        busConductor.observeResult();

        //公交车售票员观察到的情况
        System.out.println("公交车售票员观察到的情况:");
        busConductor.findPassengerChange("down", "乘客李四"); //下去乘客李四
        busConductor.findPassengerChange("up", "乘客王五");   //上来乘客王五
        busConductor.observeResult();
    }
}
```

5、说明：

A：定义：反映对象间的的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

B：在本例中，公交车售票员是观察者，当乘客情况发生变化时，公交车售票员能及时获得这个信息。

C：Observer 用于需要及时展现信息变化的系统、比如可以用于股票、税务上等。

设计模式之 Singleton——生育篇

老妈对我们没给她生个小子而是个女儿始终耿耿于怀，这不最近不知从那里听说现在政策允许再生一胎的消息后，不停的在我耳边唠叨。说道生孩子，就不得不提一提我国的计划生育政策，"只生一个好，儿女都一样"，"少生，优生，幸福一生"等等这些标语满大街都是，计划生育政策也确实为我国控制人口立下了汗马功劳，不过我觉得让许多人真正只想生一个的应该归功于教育产业化，医疗产业化等等这一大群产业化，至少我就是这样想的。嘿嘿，好象说远了，那好，让我们言归正传开始吧。

1、我们定义孩子（Child）类：

```
public class Child {
```

```
     private static Child myChild = null; //我的孩子
      private Child(){}

public static Child getChild() { //生孩子
  if(myChild == null) {
    System.out.println("你还没有孩子，可以生一个！");
    myChild = new Child();
  }
  else{
    System.out.println("你已经有孩子了，不能再生了！");
  }
    return myChild ;
 }
}
```

2、编写测试类：

```
public class Test {
  public static void main(String args[]) {
    Child.getChild();
    Child.getChild();  //当你已有一个孩子而再想生一个时，会提示你不能再生了
  }
}
```

3、说明：

A：定义：Singleton 模式的作用是保证在 Java 应用程序中，一个类 Class 只有一个实例存在。

B：其实本例也可以通过使用 synchronized 关键字来实现，板桥兄的文章对此阐述的很清楚，恕在此就不多说了。

4、特别感谢：

感谢 zdr29473 、凌寒、flylyke 等广大网友的支持，现已将代码进行了修改，希望大家多提宝贵意见，让我们共同进步吧，再次感谢。

设计模式之 Command——电视篇

每天晚上，抢电视遥控器都是我们家的保留节目。女儿喜欢看卡通屏道，老婆喜欢看电视剧屏道，我呢则喜欢看足球屏道。

因此谁控制了遥控器，就等于实现了自己的节目梦想了。嘿嘿，其实每次都是我女儿成功得到，而且她还每次都阵阵有词的说：

"大的应该让小的吗？"，你看这孩子，不知跟谁学的。然后遥控器就是老婆的，最后才轮到我，当我高兴的按到足球屏道时，

播音员说："今天的节目就到这里了，请明天再看！"，我倒地狂呕血 i 不止。

大家都知道电视遥控器节目面板（ProgramPan）是由节目按钮组成，通过选择相应的节目按钮，就可以切换到相应的节目屏道。下来让我们看看如何实现通过遥控器按钮选择节目屏道的过程吧。

1、在这里，先定义遥控器按钮（RemoteControlButton）接口：

```
public interface RemoteControlButton {
  public abstract void  selectProgram(); //选择节目屏道
}
```

2、再定义遥控器按钮（RemoteControlButton）接口的实现类：

A：卡通节目按钮（CartonProgramButton）类：

```
public class CartonProgramButton implements RemoteControlButton{
  public void selectProgram() {
    System.out.println("选择了卡通屏道！");
  }
}
```

B：电视剧节目按钮（TvPlanProgramButton）类：

```
public class TvPlanProgramButton implements RemoteControlButton {
  public void selectProgram() {
    System.out.println("选择了电视剧屏道！");
  }
}
```

C：足球节目按钮（FootProgramButton）类：

```
public class FootProgramButton implements RemoteControlButton {
  public void selectProgram() {
    System.out.println("选择了足球屏道！");
  }
}
```

3、遥控器节目面板（ProgramPan）类：用来控制节目按钮，显示节目

```
public class ProgramPan {
  public static List programList() {
    List list = new ArrayList();  //节目屏道按钮列表
    list.add(new CartonProgramButton()); //卡通屏道按钮
    list.add(new TvPlanProgramButton()); //电视剧屏道按钮
    list.add(new FootProgramButton());   //足球屏道按钮
    return list;
  }
}
```

4、编写测试类：

```
public class TestCommand {
  public static void main(String[] args) {
    List list = ProgramPan.programList();  //获得节目屏道按钮
    for (Iterator it = list.iterator();it.hasNext();)
      ( (RemoteControlButton) it.next()).selectProgram();  //选择节目屏道中对应的节目
  }
```

```
}
```

运行：

选择了卡通屏道！

选择了电视剧屏道！

选择了足球屏道！

5、说明：

A：Command 说白了就是通过选择一个个命令，然后执行相应动作。

B：Command 是对行为进行封装的典型模式，在本例中通过遥控器节目面板（ProgramPan）这个封装类来实现我们看电视节目的目的。

C：Command 模式和 Facade(外观)模式似乎比较相似。都是通过封装类来进行访问的。如何区分，对这点我也比较疑惑。

D：Command 模式是用 collection 的对象容器类，把另一些类放到里面，以实现集体的一块操作，以进行封装。facade 模式是把某个

功能的操作，集中放在一起，使之用一个统一的，对外接口，比如：封装数据库操作，发邮件操作等等。

6、特此感谢：

感谢 changlich 网友对 Command 模式和 facade 模式的区别的解释，特此将这个解释加入到说明中，希望能对大家有所帮助。再次感谢大家的支持。

设计模式之 State——交通篇

"小朋友过马路，左右看，红灯停，绿灯行，阿姨夸我是乖宝宝。"，我给女儿念着儿歌，突然女儿问我,什么是红绿灯啊？为了给她说清楚,我特意带她看我们家附近的交通灯(NearMyFamilyTrafficLight)的运行。我们都知道，交通灯有三种状态：红灯（RedLight）、黄灯（YellowLight）和绿灯（GreenLight）。交通灯状态的改变是由控制中心（ControlCenter）来控制的。

下面让我们来看看这个过程是如何实现的。

1、 在这里，先定义交通灯（TrafficLight）接口类：

```
public interface TrafficLight {
    public void showRedLight(); //显示红灯
    public void showGreenLight();//显示绿灯
     public void showYellowLight(); //显示黄灯
}
```

2、我们家附近的交通灯（NearMyFamilyTrafficLight）是对交通灯（TrafficLight）接口的具体实现：

```
public class NearMyFamilyTrafficLight implements TrafficLight{
  public void showRedLight(){
    System.out.println("红灯亮了，不能通过！");
  }

  public void showGreenLight(){
    System.out.println("绿灯亮了，可以通过！");
  }
```

```
  public void showYellowLight(){
    System.out.println("黄灯亮了!");
  }
}
```

3、定义控制中心（ControlCenter）类：

```
public class ControlCenter {
  private NearMyFamilyTrafficLight trafficLight; //我们家附近的交通灯

  public void changeState(NearMyFamilyTrafficLight trafficLight) {
      this.trafficLight = trafficLight;
  }

  public void showRedLight() { //显示红灯
      trafficLight.showRedLight();
  }

  public void showGreenLight() { //显示绿灯
      trafficLight.showGreenLight();
  }

  public void showYellowLight() { //显示黄灯
      trafficLight.showYellowLight();
  }
}
```

4、我们家附近的交通灯（NearMyFamilyTrafficLight）实际上有红、黄、绿三盏灯组成：

A：红灯（RedLight）类：

```
public class RedLight extends NearMyFamilyTrafficLight{
  public static boolean  existRedLight = false;

  public static RedLight getRedLight() {  //获得红灯
      if(existRedLight==false) {
          existRedLight = true;
          return new RedLight();
      }
      return null;
  }
}
```

B：绿灯（GreenLight）类：

```
public class GreenLight extends NearMyFamilyTrafficLight{
```

```
  public static boolean  existGreenLight = false;

  public static GreenLight getGreenLight() { //获得绿灯
      if(existGreenLight==false) {
          existGreenLight = true;
          return new GreenLight();
      }
      return null;
  }
}
```

C：黄灯（YellowLight）类：

```
public class YellowLight extends NearMyFamilyTrafficLight{
  public static boolean  existYellowLight = false;

  public static YellowLight getYellowLight() { //获得黄灯
      if(existYellowLight==false) {
          existYellowLight = true;
          return new YellowLight();
      }
      return null;
  }
}
```

5、编写测试类：

```
public class Test {
  public static void  main(String args[]){
    ControlCenter controlCenter = new ControlCenter();   //控制中心

    NearMyFamilyTrafficLight redLight = RedLight.getRedLight();  //红灯
    NearMyFamilyTrafficLight greenLight = GreenLight.getGreenLight(); //绿灯
    NearMyFamilyTrafficLight yellowLight = YellowLight.getYellowLight(); //黄灯

    controlCenter.changeState(redLight); //改变成红灯状态
    controlCenter.showRedLight();  //显示红灯
    controlCenter.changeState(yellowLight); //改变成黄灯状态
    controlCenter.showYellowLight();  //显示黄灯
    controlCenter.changeState(greenLight);  //改变成绿灯状态
    controlCenter.showGreenLight(); //显示绿灯
  }
}
```

6、说明：

A：State 的定义： 不同的状态,不同的行为;或者说,每个状态有着相应的行为。

B：我们可以看到，灯状态的改变是有控制中心来控制，通过显示不同的灯，实现了交通的正常运转。

C：因此当有状态切换这种事情要处理时，我们就可以用 State 这种模式了。

设计模式之 Proxy——买票篇

今年过年手气好，打牌赢了 100 块，我得意的笑，我得意的笑，总之一个字"爽"。因为往年打牌从没赢过啊！我高兴的回到家里，还没等我开口报告战况，老婆撂给我一句话"我弟要回上海，你给买张票吧。"我心里虽然不高兴，但脸上却表现出很开心的样子，立刻用坚定语气说到："请领导放心，保证完成任务！"。保证归保证，可是大过年的票也确实难买，在经过一番挫折后（呜呜），我只好去找票贩子（Proxy）。

说到代理这个词，大家可能都不陌生，其实现在社会上的好多中介也可以理解成为代理，说白了就是帮你办事，拿中介费而已。

1、在这里，先把买票这个活动定义成一个接口（BuyTicket）类：

```
public interface BuyTicket {
       public void buyTicket();
}
```

2、下面我们要对这接口进行实现

A：正常情况下的买票活动（NormalBuyTicket）类：

```
public class NormalBuyTicket implements BuyTicket{
  public void  buyTicket(){
     System.out.println("买火车票！");
  }
}
```

B：代理情况下的买票活动（ProxyBuyTicket）类：

```
//当不能直接访问 NormalBuyTicket 对象时，必须要用代理对象
public class ProxyBuyTicket implements BuyTicket{
  private NormalBuyTicket normalBuyTicket;

  public void  buyTicket(){
     if(normalBuyTicket==null){
       normalBuyTicket = new NormalBuyTicket();
     }
     normalBuyTicket.buyTicket();
       getMoney();
  }

  public void getMoney(){
     System.out.println("获得代理费！");
  }
}
```

3、编写测试类：

```
public class Test {
  public static void  main(String args[]){
    BuyTicket buyTicket = new ProxyBuyTicket();
```

```
    buyTicket.buyTicket();
  }
}
```

4、说明：

A：定义：为其他对象提供一种代理以控制对这个对象的访问。也就是说当我们不能直接访问我们想访问的对象时，必须通过一个代理对象来访问。

B：在本例中，我想买票，但当我直接买不到票时，就只好通过票贩子来买，这个道理应该谁都知道啊。

5、后记：最终通过票贩子，我获得了去上海的票，老婆很高兴，还说我很能干，但是我付出了 100 元的代理费，我到底该哭还是该笑呢，但又想一下："花 100 元让老婆表扬我，说我很能干，也不错啊，你说呢！嘿嘿"。

设计模式之 Prototype——作业篇

今天要交作业，可是由于我这几天沉迷于 CS 之中，到现在还没写作业，这可该怎么办呢，谁都知道我们老师最讨厌不写作业的学生了。嘿嘿，还好我有一门优秀的技能，那就是——Clone 大法（俗称 COPY 大法），正是由于拥有该技能，才能使我残酷的斗争中立于不败之地。于是我以迅雷不及眼耳盗铃之势拿来了张三的作业，开始运功。

说道这里就不得不先说说什么叫 Clone？

有一个对象 A，在某一时刻 A 中已经包含了一些有效值，此时可能会需要一个和 A 完全相同新对象 B，并且此后对 B 任何改动都不会影响到 A 中的值，也就是说，A 与 B 是两个独立的对象，但 B 的初始值是由 A 对象确定的。

好了言归正传，让我们开始吧。

1、在这里，先定义一个拷贝作业（CopyHomeWork）接口：

```
public interface CopyHomeWork extends Cloneable{
  public String getHomeWork() ;
  public void setHomeWork(String homeWork);
}
```

2、再定义一个作业（HomeWork）的实现类：

```
public abstract class HomeWork implements CopyHomeWork{
  String homeWork;

  public void setHomeWork(String homeWork) {
    this.homeWork = homeWork;
  }

  public String getHomeWork() {
    return this.homeWork;
  }

// 典型的调用 clone()代码
  public Object clone() {
    Object object = null;
```

```
    try {
      object = super.clone();
    }

    catch (CloneNotSupportedException exception) {
      System.err.println("*** is not Cloneable");
    }
    return object;
  }

  public abstract void  DoHomeWork();  //做作业的抽象类
}
```

3、定义张三的作业（ZhangSanHomeWork）类：

```
public class ZhangSanHomeWork extends HomeWork{
  public void DoHomeWork() {
      System.out.println("张三作完了作业！");
  }
 }
```

4、编写测试类：

```
public class TestCopyHomeWork {
  public static void  main(String args[]){
    ZhangSanHomeWork zhangsanHomeWork = new ZhangSanHomeWork();
    HomeWork myHomeWork = (HomeWork)zhangsanHomeWork.clone();
    myHomeWork.DoHomeWork();
    myHomeWork.setHomeWork("我正在抄作业，请勿打扰！");
    System.out.println(myHomeWork.getHomeWork());
  }
}
```

5、说明：

A：定义：用原型实例指定创建对象的种类,并且通过拷贝这些原型创建新的对象。

B：Prototype 模式允许一个对象再创建另外一个可定制的对象，根本无需知道任何如何创建的细节,工作原理是:通过将一个原型对象传给那个要发动创建的对象,这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。

C：Prototype 模式最终演变成 clone 的使用。

设计模式之 Facade——家庭篇

今年十一国庆节，我呆在家里美美的享受了一下家的温馨。首先让我来介绍一下我的家庭成员：

妻子（Wife）

女儿（Daughter）

我（My）

我们都是家庭（Family）的一分子，我们是以家庭对外的。就象我们国家对外是以"中国"，外国人都称我们是中国人，但在中国这个大家庭内部，包括了汉、回、蒙、。。。等 56 个民族一样。

可见对外我们是要以统一的身份，或叫统一的外观（Facade）进行展现。

好了言归正传。

1、 在这里，先定义家庭的各个成员类：

妻子（Wife）：

```
public class Wife {
    public Wife() {
      System.out.println("老婆伟大 !");
}
}
```

女儿（Daughter）：

```
public class Daughter {
    public Daughter () {
        System.out.println("女儿可爱 !");
}
}
```

我（My）：

```
public class My {
      public My () {
          System.out.println("我爱我家 !");
}
}
```

2、定义家庭对外展现（FamilyOutShow）类：

```
public class FamilyOutShow {
    public void MyFamily() {
        Wife myWife = new Wife();
        Daughter myDaughter = new Daughter ();
        My mySelf = new My();
    }
}
```

3、编写测试类：

```
public class MyFamilyTest {
    public static void main(String[] args) {
        FamilyOutShow myFamily = new FamilyOutShow ();
        myFamily.MyFamily();
}
}
```

4、说明：

A：Facade 的定义：为子系统中的一组接口提供一个一致的界面。

B：使用统一对外接口，可以降低系统的复杂性,增加了灵活性。

C：从例子中可以看到，外界只是访问了家庭对外展现（FamilyOutShow）类，而没有直接与成员类打交道。这样比如说增加了一个新的成员类(比如说儿子(Son)类)，只要修改家庭对外展现(FamilyOutShow)类即可，而不用修改调用。

设计模式之 Decorator——家装篇

最近家里搞装修，做了一套家具，需要刷一下漆，因此我就去市场找了油漆师傅和油漆徒弟两个人。

油漆师傅主要买油漆和调油漆，油漆徒弟主要来刷油漆（团队精神？哈哈，不禁让让我想起 CS，你先冲，我拣菜。）。

1、在这里，我们先把这个油漆工作定义成一个接口类：

```
public interface Work{
       public void brush(); //刷油漆
}
```

2、因为油漆师傅和油漆徒弟的任务是刷油漆，因此他们要对 Work 接口进行实现：

A：油漆徒弟

刷油漆的工作主要是由油漆徒弟来完成，所以我们把油漆徒弟定义成 Brusher（油漆师傅在一旁说："徒弟吗，就是要多干活。"，油漆徒弟小声嘀咕："多你个头。"）。

```
public class Brusher implements Work{
  public void brush() {
    System.out.println("刷油漆");
  }
}
```

B：油漆师傅

我们把油漆师傅定义成 Decorator。

```
public class Decorator implements Work{
  private Work work;
//油漆师傅的工作被放置在这个 List 中

  private ArrayList prework = new ArrayList();
//油漆师傅的默认工作
  public Decorator(Work work) {
    this.work = work;
    prework.add("买油漆");
    prework.add("调油漆");
  }

  public void brush() {//刷油漆, 油漆师傅也要实现此方法
    newWork();          //当油漆师傅接到活，就开始一个新的工作
  }

//新的工作
  public void newWork() {
    preWork();      //油漆师傅做的前期辅助工作
```

```
    work.brush();   //让徒弟干的刷油漆的工作
  }

//油漆师傅做的前期辅助工作
  public void preWork() {
    ListIterator listIterator = prework.listIterator();
    while (listIterator.hasNext()) {
      System.out.println( ( (String) (listIterator.next())) + "完成");
    }
  }
```

3、编写测试类：

```
public class test {
  public static void  main(String args[]) {
    Work bursher = new Brusher();
    Work decorator = new Decorator(bursher);
    decorator.brush();
   //我把活交给油漆师傅，油漆师傅下来再把实际刷油漆的工作指派给油漆徒弟干
  }
 }
```

4、说明：

A：代码只用来学习 Decorator 模式,要运行的话，必须要做一点改动。

B：在这过程中，我只和油漆师傅打交道，具体的刷油漆那是由油漆师傅和油漆徒弟之间的事，我是不用关心的。

C：使用 Decorator 的理由是:这些功能需要由用户动态决定加入的方式和时机.Decorator 提供了"即插即用"的方法,在运行期间决定何时增加何种功能.

设计模式之 Visitor——送礼篇

今年过年不收礼，收礼只收脑白金。听到这暗示性的广告词，我的脑袋突然一亮。因为最近因为要办某事，必须要给单位的领导要表示一下。到底送什么，还真让人头痛，还好有脑白金，奶奶的。。。，腐败啊，罪过！

首先要对送礼的对象进行分析，单位有两个领导，一正，一副。因此给不同的领导送的礼也是不同的（哈，收入要和产出成正比吗），好了言归正传。

1、在这里，先把领导定义成一个接口类：

```
public interface Leader{
      //主要任务----收 visitor（拜访者）的礼
      public void accept(Visitor visitor);
}
```

在把拜访者定义成另一个接口类：

```
public interface Visitor
{
```

```
   public void visitFirstHand(FirstHand first);  //拜访一把手（带的礼物）
   public void visitSecondHand(SecondHand second);  //拜访二把手（带的礼物）
   public void visitCollection(Collection collection);  //判断是拜访一把手还是二把手
}
```

2、下面我们要对这两个接口进行实现：

A：一把手

```
public class FirstHand implements Leader {
  private String value; //注意此处是 String

  public FirstHand (String string) {//一把手的构造函数
    value = string;
  }

  public String getValue() { //获得礼物
    return value;
  }

//定义 accept 的具体内容 这里是很简单的一句调用
  public void accept(Visitor visitor) {
    visitor.visitFirstHand (this);  //接收拜访人送的礼
  }
}
```

B：二把手

```
public class SecondHand implements Leader {
  private Float value; //注意此处是 Float

  public SecondHand (Float string) {//二把手的构造函数
    value = string;
  }

  public Float getValue() { //获得礼物
    return value;
  }

//定义 accept 的具体内容 这里是很简单的一句调用
  public void accept(Visitor visitor) {
    visitor.visitFirstHand (this);  //接收拜访人送的礼
  }
}
```

C：拜访人（我）

```
public class visitMe implements Visitor{
```

```
    public void visitCollection(Collection collection) {
      Iterator iterator = collection.iterator();
      while (iterator.hasNext()) {
               Object o = iterator.next();
          if (o instanceof Leader)          //判断要送给哪个领导
           ((Leader)o).accept(this);      //不同的领导进入不同的实现类
      }
    }

    public void visitFirstHand (FirstHand first) {
      System.out.println("送的礼是："+ first.getValue());
    }

    public void visitSecondHand (SecondHand second) {
      System.out.println("送的礼是：" + second.getValue());
    }
}
```

3、编写测试类：

```
public class test {
   public static void  main(String args[]){
     Visitor visitor = new visitMe ();
     FirstHand present = new FirstHand ("十盒脑白金");
     visitor.visitFirstHand (present);
     Collection list = new ArrayList();
     list.add(new FirstHand ("十盒脑白金"));
     list.add(new SecondHand (new Float("一斤小点心"))); //为了说明不同，如要运行，要做类型转换。
     visitor.visitCollection(list);
   }
```

4、说明：

A：代码只用来学习 Visitor 模式,要运行的话，必须要做一点改动。

B：FirstHand，SecondHand 只是一个个具体实现，实际上还可以拓展为更多的实现，整个核心奥妙在 accept 方法中，在遍历 Collection 时，通过相应的 accept 方法调用具体类型的被访问者。这一步确定了被访问者类型

C：使用访问者模式是对象群结构中(Collection) 中的对象类型很少改变，也就是说领导很少变化。

设计模式之 Builder——购机篇

最近想买一台电脑用于学习，因此我就去了一家电脑公司，经过分析，选用了下面的配置：

CPU P2.4

主板 Intel

硬盘 80G

。。。

买过电脑的朋友可能都知道，我们选好配置后，电脑公司就会有专门的组装师（Assembler）来给我们装机。电脑（Computer）就是由这些东西（我们称之为Part）组成的。学过经济学的朋友可能都知道,如果这台组装好的电脑不卖掉，那它就不是商品（Commodity），而仅仅是台电脑而已。

1、 在这里，我们先定义商品（Commodity）类：

```
public class Commodity {
  String commodity ="";

  public Commodity (Part partA,Part partB,Part partC) {//由各个部分组成
    this. commodity = partA.part+"\n";
    this. commodity = product+partB.part+"\n";
    this. commodity = product+partC.part;
    System.out.println("我的机器配置为：\n"+ commodity);
  }
}
```

2、 下来我们再定义电脑的组成部分（Part）类：

```
public class Part {
  String part="";

  public Part(String part){
    this.part = part;
  }
}
```

3、 我们把电脑（Computer）定义成一个接口类：

```
public interface Computer {
//组装部件A  比如CPU
  void buildPartA();

//组装部件B  比如主板
  void buildPartB();

//组装部件C  比如硬盘
  void buildPartC();

//返回最后组装成品结果（返回最后组装好的电脑）
//成品的组装过程不在这里进行，而是由组装师（Assembler）类完成的。
//从而实现了过程和部件的分离
  Product getProduct();
}
```

4、 定义电脑的组装师（Assembler）类：

```
public class Assembler {
```

```
  private Computer computer;

  public Assembler(Computer computer) {  //主要任务是装电脑
    this.computer = computer;
  }
// 将部件 partA partB partC 最后组成复杂对象
//这里是将主板、CPU 和硬盘组装成 PC 的过程

  public void construct() {
    computer.buildPartA();
    computer.buildPartB();
    computer.buildPartC();
  }
}
```

5、 我的电脑是对电脑（Computer）接口的具体实现，因此再定义 MyComputer 实现类：

```
public class MyComputer implements Computer {
  Part partA, partB, partC;

  public void buildPartA() {
    partA = new Part("P42.4 CPU");
  }

  public void buildPartB() {
    partB = new Part("Inter 主板");
  }

  public void buildPartC() {
    partC = new Part("80G 硬盘");
  }

  public Product getProduct() {
   //返回最后组装成品结果
    Commodity myComputer = new Commodity (partA,partB,partC);
    return myComputer;
  }
}
```

6、 编写测试类：

```
public class MyComputerTest {
  public static void  main(String args[]){
    MyComputer myComputer = new MyComputer();      //组装我的电脑
    Assembler assembler = new Assembler( myComputer );  //派某一位组装师
```

```
    assembler.construct();    //组装师进行组装过程
    Commodity commodity = myComputer.getProduct();   //卖给我的电脑（商品）
  }
}
```

7、说明：

A：代码只用来学习 Builder 模式,要运行的话，必须要做一点改动。

B：将一个复杂对象的构建与它的表示分离,使得同样的构建过程可以创建不同的表示。因为每个人的电脑配置可能都是不同的。

C：我们使用 Builer 是为了构建复杂对象的过程和它的部件解耦，也就是说将过程分的尽可能细，而且每一部分只用完成自己的功能即可（各司其职嘛）。

设计模式之 Factory——买货篇

今天老婆让我去市场买一些水果，具体买什么自己定（哈，老婆放放权了！）。来到市场，我发现主要有一些水果：苹果（Apple），葡萄（Grape）和鸭梨（Pear）。

到底买什么好呢？我一阵思量。俗话说："饭后一只烟，赛过活神仙。饭后吃苹果，西施见我躲。"为了老婆的漂亮，我决定买苹果。好，言归正传，开始买吧！

主要有以下三种 Factory 模式：

Simple Factory 模式:专门定义一个类来负责创建其它类的实例，被创建的实例通常都具有共同的父类。

Factory Method 模式:将对象的创建交由父类中定义的一个标准方法来完成，而不是其构造函数，究竟应该创建何种对象由具体的子类负责决定。

Abstract Factory 模式:提供一个共同的接口来创建相互关联的多个对象。

一、Simple Factory 模式：

1、 在这里，我们先定义水果（Fruit）接口：

```
public interface Fruit {
  void plant();  //水果是被种植的
  void enableEat();  //水果能吃
}
```

2、 苹果（Apple）是对水果（Fruit）接口的实现：

```
public class Apple implements Fruit{
  public void plant(){
    System.out.println("种苹果!");
  }

  public void enableEat(){
    System.out.println("苹果好吃!");
  }
}
```

3、 葡萄（Grape）是对水果（Fruit）接口的实现：

```
public class Grape implements Fruit{
  public void plant(){
    System.out.println("种葡萄!");
  }

  public void enableEat(){
    System.out.println("葡萄好吃!");
  }
}
```

4、 鸭梨（Pear）是对水果（Fruit）接口的实现：

```
public class Pear implements Fruit{
  public void plant(){
    System.out.println("种鸭梨!");
  }

  public void enableEat(){
    System.out.println("鸭梨好吃!");
  }
}
```

5、定义买水果（BuyFruit）这一过程类：

```
public class BuyFruit {
  /**
  * 简单工厂方法
  */

  public static Fruit buyFruit(String which){
    if (which.equalsIgnoreCase("apple")) {  //如果是苹果，则返回苹果实例
      return new Apple();
    }
    else if (which.equalsIgnoreCase("pear")){  //如果是鸭梨，则返回鸭梨实例
      return new Strawberry();
    }
    else if (which.equalsIgnoreCase("grape")) { //如果是葡萄，则返回葡萄实例
      return new Grape();
    }
    else{
      return null;
    }
  }
}
```

6、 编写测试类：

```
public class FruitTest {
  public static void  main(String args[]){
    BuyFruit buy = new BuyFruit();   //开始买水果这个过程
    buy.buyFruit("apple").enableEat(); //调用苹果的 enableEat()方法
  }
}
```

7、 说明：

A：我要购买苹果，只需向工厂角色（BuyFruit）请求即可。而工厂角色在接到请求后，会自行判断创建和提供哪一个产品。

B：但是对于工厂角色（BuyFruit）来说，增加新的产品（比如说增加草莓）就是一个痛苦的过程。工厂角色必须知道每一种产品，如何创建它们，以及何时向客户端提供它们。换言之，接纳新的产品意味着修改这个工厂。

C：因此 Simple Factory 模式的开放性比较差。

有什么办法可以解决这个问题吗？那就需要 Factory Method 模式来为我们服务了。

二、Factory Method 模式：

1、同样，我们先定义水果（Fruit）接口：

```
public interface Fruit {
  void plant();  //水果是被种植的
  void enableEat();  //水果能吃
}
```

2、苹果（Apple）是对水果（Fruit）接口的实现：

```
public class Apple implements Fruit{
  public void plant(){
    System.out.println("种苹果!");
  }

  public void enableEat(){
    System.out.println("苹果好吃!");
  }
}
```

3、葡萄（Grape）是对水果（Fruit）接口的实现：

```
public class Grape implements Fruit{
  public void plant(){
    System.out.println("种葡萄!");
  }

  public void enableEat(){
    System.out.println("葡萄好吃!");
  }
}
```

4、鸭梨（Pear）是对水果（Fruit）接口的实现：

```
public class Pear implements Fruit{
  public void plant(){
    System.out.println("种鸭梨!");
  }

  public void enableEat(){
    System.out.println("鸭梨好吃!");
  }
}
```

5、在这里我们将买水果（BuyFruit）定义为接口类：

```
public interface BuyFruit{
  /**
  * 工厂方法
  */
  public Fruit buyFruit();   //定义买水果这一过程
}
```

6、买苹果是（BuyApple）对买水果（BuyFruit）这个接口的实现

```
public class BuyApple implements BuyFruit{
  public Fruit buyFruit(){
    return new Apple();  //返回苹果实例
}
}
```

7、买鸭梨是（BuyPear）对买水果（BuyFruit）这个接口的实现

```
public class BuyPear implements BuyFruit{
  public Fruit BuyPear (){
    return new Pear();  //返回鸭梨实例
}
}
```

8、买葡萄是（BuyGrape）对买水果（BuyFruit）这个接口的实现

```
public class BuyGrape implements BuyFruit{
  public Fruit BuyGrape (){
    return new Grape ();  //返回葡萄实例
}
}
```

9、编写测试类：

```
public class FruitTest {
  public static void  main(String args[]){
```

```
    BuyApple buy = new BuyApple(); //开始买水果这个过程
    buy.buyFruit().enableEat();       //调用苹果的 enableEat()方法
  }
}
```

10、说明：

A:工厂方法模式和简单工厂模式在结构上的不同是很明显的。工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上。工厂方法模式可以允许很多具体工厂类从抽象工厂类中将创建行为继承下来，从而可以成为多个简单工厂模式的综合，进而推广了简单工厂模式。

B：工厂方法模式退化后可以变得很像简单工厂模式。设想如果非常确定一个系统只需要一个具体工厂类，那么就不妨把抽象工厂类合并到具体的工厂类中去。由于反正只有一个具体工厂类，所以不妨将工厂方法改成为静态方法，这时候就得到了简单工厂模式。C：如果需要加入一个新的水果，那么只需要加入一个新的水果类以及它所对应的工厂类。没有必要修改客户端，也没有必要修改抽象工厂角色或者其他已有的具体工厂角色。对于增加新的水果类而言，这个系统完全支持"开-闭"原则。

D：对 Factory Method 模式而言，它只是针对一种类别（如本例中的水果类 Fruit），但如果我们还想买肉，那就不行了，这是就必须要 Factory Method 模式帮忙了。

三、Abstract Factory 模式

1、同样，我们先定义水果（Fruit）接口：

```
public interface Fruit {
  void plant();  //水果是被种植的
  void enableEat();  //水果能吃
}
```

2、苹果（Apple）是对水果（Fruit）接口的实现：

```
public class Apple implements Fruit{
  public void plant(){
    System.out.println("种苹果!");
  }

  public void enableEat(){
    System.out.println("苹果好吃!");
  }
}
```

3、葡萄（Grape）是对水果（Fruit）接口的实现：

```
public class Grape implements Fruit{
  public void plant(){
    System.out.println("种葡萄!");
  }

  public void enableEat(){
    System.out.println("葡萄好吃!");
  }
}
```

4、鸭梨（Pear）是对水果（Fruit）接口的实现：

```
public class Pear implements Fruit{
  public void plant(){
    System.out.println("种鸭梨!");
  }

  public void enableEat(){
    System.out.println("鸭梨好吃!");
  }
}
```

5、 定义肉（Meat）接口：

```
public interface Meat {
  void feed();  //肉是喂养的
  void enableEat();  //肉能吃
}
```

6、 猪肉（BigMeat）是对肉（Meat）接口的实现：

```
public class BigMeat implements Meat{
  public void feed(){
   System.out.println("养猪!");
  }

  public void enableEat(){
    System.out.println("猪肉好吃!");
  }
}
```

7、 牛肉（CowMeat）是对肉（Meat）接口的实现：

```
public class CowMeat implements Meat {
  public void feed(){
    System.out.println("养牛!");
  }

  public void enableEat(){
    System.out.println("牛肉好吃!");
  }
}
```

8、 我们可以定义买货人（Buyer）接口：

```
public interface Buyer {
  /**
   * 买水果工厂方法
```

```
     */

    public Fruit buyFruit(Fruit whichFruit);
    /**
     * 买肉的工厂方法
     */
    public Meat buyMeat(Meat whichMeat);
}
```

9、 我（MyBuyer）是对买货人（Buyer）接口的实现：

```
public class MyBuyer implements Buyer{
    /**
     * 买水果工厂方法
     */

    public Fruit buyFruit(Fruit whichFruit){
        return whichFruit;
    }

    /**
     * 买肉的工厂方法
     */
    public Meat buyMeat(Meat whichMeat){
        return whichMeat;
    }
}
```

10、编写测试类：

```
public class MyBuyerAbstractTest {
    public static void  main(String args[]){
        Fruit apple = new Apple();   //苹果实例
        Meat big = new BigMeat();  //猪肉实例
        MyBuyer my = new MyBuyer();  //我是买者的实例
        my.buyFruit(apple).enableEat();  //我买苹果
        my.buyMeat(big).enableEat();    //我买猪肉
    }
}
```

11、说明：

A：抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象。这就是抽象工厂模式的用意。

B：抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。

C：抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品（Fruit）等级结构；而抽象工厂模式则需要面对多个产品等级结构(Fruit、Meat)。

设计模式之 Iterator——点名篇

上了这么多年学，我发现一个问题，好象老师都很喜欢点名，甚至点名都成了某些老师的嗜好，一日不点名，就饭吃不香，觉睡不好似的，我就觉得很奇怪，你的课要是讲的好，同学又怎么会不来听课呢，殊不知："误人子弟，乃是犯罪！"啊。

好了，那么我们现在来看老师这个点名过程是如何实现吧：

1、老规矩，我们先定义老师（Teacher）接口类：

```
public interface Teacher {
  public Iterator createIterator();   //点名
}
```

2、具体的老师（ConcreteTeacher）类是对老师（Teacher）接口的实现：

```
public class ConcreteTeacher implements Teacher{
  private Object[] present = {"张三来了","李四来了","王五没来"};  //同学出勤集合
  public Iterator createIterator(){
    return new ConcreteIterator(this);      //新的点名
  }
  public Object getElement(int index){  //得到当前同学的出勤情况
    if(index      return present[index];
    }
    else{
      return null;
    }
  }
  public int getSize(){
    return present.length;  //得到同学出勤集合的大小,也就是说要知道班上有多少人
  }
}
```

3、定义点名（Iterator）接口类：

```
public interface Iterator {
  void first();  //第一个
  void next();   //下一个
  boolean isDone();  //是否点名完毕
  Object currentItem();  //当前同学的出勤情况
}
```

4、具体的点名（ConcreteIterator）类是对点名（Iterator）接口的实现：

```
public class ConcreteIterator implements Iterator{
  private ConcreteTeacher teacher;
  private int index = 0;
  private int size = 0;
  public ConcreteIterator(ConcreteTeacher teacher){
```

```
    this.teacher = teacher;
    size = teacher.getSize(); //得到同学的数目
    index = 0;
  }
  public void first(){  //第一个
    index = 0;
  }
  public void next(){ //下一个
    if(index      index++;
    }
  }
  public boolean isDone(){ //是否点名完毕
    return (index>=size);
  }
  public Object currentItem(){ //当前同学的出勤情况
    return teacher.getElement(index);
  }
}
```

5、编写测试类：

```
public class Test {
  private Iterator it;
  private Teacher teacher = new ConcreteTeacher();
  public void operation(){
    it = teacher.createIterator();  //老师开始点名
    while(!it.isDone()){ //如果没点完
      System.out.println(it.currentItem().toString()); //获得被点到同学的情况
      it.next();   //点下一个
    }
  }
  public static void main(String agrs[]){
    Test test = new Test();
    test.operation();
  }
}
```

6、说明：

A：定义：Iterator 模式可以顺序的访问一个聚集中的元素而不必暴露聚集的内部情况。

B：在本例中， 老师（Teacher）给出了创建点名（Iterator）对象的接口，点名（Iterator）定义了遍历同学出勤情况所需的接口。

C：Iterator 模式的优点是当（ConcreteTeacher）对象中有变化是，比如说同学出勤集合中有加入了新的同学，或减少同学时，这种改动对客户端是没有影响的。

设计模式之 Memento——系统篇

经常使用计算机的人恐怕对系统备份（Memento）不会陌生，当你的 Windows 系统运行正常时，对它进行备份，当系统运行有问题时，就可以调用备份快速的将系统恢复，这样就可以大量节省重新装系统的痛苦，特别是当你缺少某一驱动，或在装系统是出现一些怪问题时，犹为痛苦。我想有过这种经历的人应该很了解吧，呵呵！

好了，下面让我们看看这个过程该如何实现吧：

1、我们先定义 Windows 系统（WindowsSystem）类：

```
public class WindowsSystem {
      private String state;

      //创建备份，保存当前状态
      public Memento createMemento() {
            return new Memento(state);
      }

public void restoreMemento(Memento memento){ //从备份中恢复系统
    this.state=memento.getState();
  }
  public String getState(){  //获得状态
    return this.state;
  }
  public void setState(String state){  //设置状态
    this.state=state;
    System.out.println("当前系统处于"+this.state);
  }
}
```

2、再定义备份（Memento）类：

```
public class Memento {
  private String state;
  public Memento(String state) {  //备份
    this.state=state;
  }
  public String getState(){ //获得状态
    return this.state;
  }
  public void setState(String state){  //设置状态
    this.state=state;
  }
}
```

3、定义用户（User）类：

```
public class User {
  private Memento memento;
```

```
  public Memento retrieveMemento() {  //恢复系统
    return this.memento;
  }
  public void saveMemento(Memento memento){  //保存系统
    this.memento=memento;
  }
}
```

4、编写测试类：

```
public class Test {
  public static void main(String args[]) {
    WindowsSystem Winxp = new WindowsSystem(); //Winxp 系统
    User user = new User();   //某一用户
    Winxp.setState("好的状态");   //Winxp 处于好的运行状态
   user.saveMemento(Winxp.createMemento());//用户对系统进行备份，Winxp 系统要产生备份文件
    Winxp.setState("坏的状态");   //Winxp 处于不好的运行状态
    Winxp.restoreMemento(user.retrieveMemento());   //用户发恢复命令，系统进行恢复
    System.out.println("当前系统处于"+Winxp.getState());
  }
}
```

5、说明：

A：定义：Memento 对象是一个保存另外一个对象内部状态拷贝的对象，这样以后就可以将该对象恢复到原先保存的状态。

B：Memento 模式的用意是在不破坏封装的条件下，将一个对象的状态捕捉住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

C：Memento 模式所涉及的角色有三个，备忘录角色、发起人角色和负责人角色。

备忘录角色的作用：

（1） 将发起人对象的内部状态存储起来，备忘录可以根据发起人对象的判断来决定存储多少发起人对象的内部状态。

（2） 备忘录可以保护其内容不被发起人对象之外的任何对象所读取。

发起人角色的作用：

（1） 创建一个含有当前内部状态的备忘录对象。

（2） 使用备忘录对象存储其内部状态。

负责人角色的作用：

（1） 负责保存备忘录对象。

（2） 不检查备忘录对象的内容。

D：在本例中，备份（Memento）类是备忘录角色、Windows 系统（WindowsSystem）类是发起人角色、用户（User）类是负责人角色。

文章来自： http://garyfeng.blogchina.com/index.htm

http://www.java3z.com/cwbwebhome/article/article3/3250.html?id=847