设计模式公司荣誉出品

您的设计模式

我们的设计模式

CBF4LIFE

2009 年 5 月

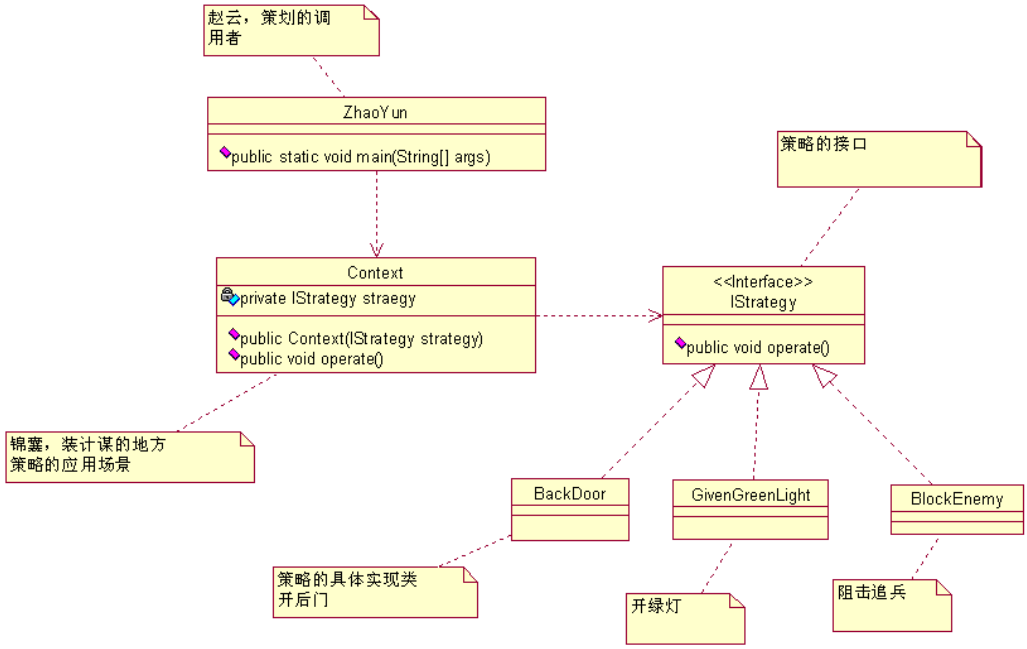
我希望这本书的阅读者具备最基本的代码编写能力，您是一个初级的 coder,可以从中领会到怎么设计一段优秀的代码；您是一个高级程序员，可以从中全面了解到设计模式以及 Java 的边角技术的使用；您是一个顶级的系统分析师，可以从中获得共鸣，寻找到项目公共问题的解决办法，呀，是不是把牛吹大了?!

目　录

1. 策略模式【Strategy Pattern】

刘备要到江东娶老婆了，走之前诸葛亮给赵云（伴郎）三个锦囊妙计，说是按天机拆开解决棘手问题，嘿，还别说，真是解决了大问题，搞到最后是周瑜陪了夫人又折兵呀，那咱们先看看这个场景是什么样子的。

先说这个场景中的要素：三个妙计，一个锦囊，一个赵云，妙计是小亮同志给的，妙计是放置在锦囊里，俗称就是锦囊妙计嘛，那赵云就是一个干活的人，从锦囊中取出妙计，执行，然后获胜，用 JAVA 程序怎么表现这个呢？我们先看类图：



三个妙计是同一类型的东东，那咱就写个接口：

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 首先定一个策略接口，这是诸葛亮老人家给赵云的三个锦囊妙计的接口
 *
 */
public interface IStrategy {

    //每个锦囊妙计都是一个可执行的算法
```

```
    public void operate();

}
```

然后再写三个实现类，有三个妙计嘛：

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 找乔国老帮忙，使孙权不能杀刘备
 */
public class BackDoor implements IStrategy {

    public void operate() {
        System.out.println("找乔国老帮忙，让吴国太给孙权施加压力");
    }

}
```

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 求吴国太开个绿灯
 */
public class GivenGreenLight implements IStrategy {


    public void operate() {
        System.out.println("求吴国太开个绿灯,放行！");
    }

}
```

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 孙夫人断后，挡住追兵
 */
```

```
public class BlockEnemy implements IStrategy {

    public void operate() {
        System.out.println("孙夫人断后，挡住追兵");
    }

}
```

好了，大家看看，三个妙计是有了，那需要有个地方放这些妙计呀，放锦囊呀：

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 计谋有了，那还要有锦囊
 */
public class Context {
    //构造函数，你要使用那个妙计
    private IStrategy straegy;
    public Context(IStrategy strategy){
        this.straegy = strategy;
    }

    //使用计谋了，看我出招了
    public void operate(){
        this.straegy.operate();
    }
}
```

然后就是赵云雄赳赳的揣着三个锦囊，拉着已步入老年行列的、还想着娶纯情少女的、色迷迷的刘老爷子去入赘了，嗨，还别说，小亮的三个妙计还真是不错，瞅瞅：

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class ZhaoYun {

    /**
     * 赵云出场了，他根据诸葛亮给他的交代，依次拆开妙计
```

```
     */
    public static void main(String[] args) {
        Context context;

        //刚刚到吴国的时候拆第一个
        System.out.println("-----------刚刚到吴国的时候拆第一个
-------------");
        context = new Context(new BackDoor()); //拿到妙计
        context.operate();  //拆开执行
        System.out.println("\n\n\n\n\n\n\n\n\n");

        //刘备乐不思蜀了，拆第二个了
        System.out.println("-----------刘备乐不思蜀了，拆第二个了
-------------");
        context = new Context(new GivenGreenLight());
        context.operate();  //执行了第二个锦囊了
        System.out.println("\n\n\n\n\n\n\n\n\n");

        //孙权的小兵追了，咋办？拆第三个
        System.out.println("-----------孙权的小兵追了，咋办？拆第三个
-------------");
        context = new Context(new BlockEnemy());
        context.operate();  //孙夫人退兵
        System.out.println("\n\n\n\n\n\n\n\n\n");

        /*
         *问题来了：赵云实际不知道是那个策略呀，他只知道拆第一个锦囊，
         *而不知道是BackDoor这个妙计，咋办？  似乎这个策略模式已经把计谋名称
写出来了
         *
         * 错！BackDoor、GivenGreenLight、BlockEnemy只是一个代码，你写成
first、second、third，没人会说你错！
         *
         * 策略模式的好处就是：体现了高内聚低耦合的特性呀，缺点嘛，这个那个，我
回去再查查
         */
    }

}
```

就这三招，搞的周郎是“陪了夫人又折兵”呀！这就是策略模式，高内聚低耦合的特点也表现出来了，还有一个就是扩展性，也就是 OCP 原则，策略类可以继续增加下去，只要修改 Context.java 就可以了，这个不多说了，自己领会吧。

2. 代理模式【Proxy Pattern】

什么是代理模式呢？我很忙，忙的没空理你，那你要找我呢就先找我的代理人吧，那代理人总要知道被代理人能做哪些事情不能做哪些事情吧，那就是两个人具备同一个接口，代理人虽然不能干活，但是被代理的人能干活呀。

比如西门庆找潘金莲，那潘金莲不好意思答复呀，咋办，找那个王婆做代理，表现在程序上时这样的：

先定义一种类型的女人：

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一种类型的女人，王婆和潘金莲都属于这个类型的女人
 */
public interface KindWomen {

    //这种类型的女人能做什么事情呢？
    public void makeEyesWithMan();  //抛媚眼


    public void happyWithMan();  //happy what? You know that!

}
```

一种类型嘛，那肯定是接口，然后定义潘金莲：

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定一个潘金莲是什么样的人
 */
public class PanJinLian implements KindWomen {
```

```
    public void happyWithMan() {
        System.out.println("潘金莲在和男人做那个.....");
    }

    public void makeEyesWithMan() {
        System.out.println("潘金莲抛媚眼");

    }

}
```

再定一个丑陋的王婆：

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 王婆这个人老聪明了，她太老了，是个男人都看不上，
 * 但是她有智慧有经验呀，她作为一类女人的代理！
 */
public class WangPo implements KindWomen {
    private KindWomen kindWomen;

    public WangPo(){ //默认的话，是潘金莲的代理
        this.kindWomen = new PanJinLian();
    }

    //她可以是KindWomen的任何一个女人的代理，只要你是这一类型
    public WangPo(KindWomen kindWomen){
        this.kindWomen = kindWomen;
    }

    public void happyWithMan() {
        this.kindWomen.happyWithMan();  //自己老了，干不了，可以让年轻的
代替
    }

    public void makeEyesWithMan() {
        this.kindWomen.makeEyesWithMan();  //王婆这么大年龄了，谁看她抛
媚眼？！

    }
```

```
}
```

两个女主角都上场了，男主角也该出现了：

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个西门庆，这人色中饿鬼
 */
public class XiMenQing {

    /*
     * 水浒里是这样写的：西门庆被潘金莲用竹竿敲了一下难道，痴迷了，
     * 被王婆看到了，就开始撮合两人好事，王婆作为潘金莲的代理人
     * 收了不少好处费，那我们假设一下：
     * 如果没有王婆在中间牵线，这两个不要脸的能成吗？难说的很！
     */
    public static void main(String[] args) {
        //把王婆叫出来
        WangPo wangPo = new WangPo();

        //然后西门庆就说，我要和潘金莲happy，然后王婆就安排了西门庆丢筷子的那出戏：
        wangPo.makeEyesWithMan();  //看到没，虽然表面上时王婆在做，实际上爽的是潘金莲
        wangPo.happyWithMan();   }
}
```

那这就是活生生的一个例子，通过代理人实现了某种目的，如果真去掉王婆这个中间环节，直接是西门庆和潘金莲勾搭，估计很难成就武松杀嫂事件。

那我们再考虑一下，水浒里还有没有这类型的女人？有，卢俊义的老婆贾氏（就是和那个固管家苟合的那个），这名字起的：“假使”，那我们也让王婆做她的代理：

把贾氏素描出来：

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
```

```
 */
public class JiaShi implements KindWomen {


    public void happyWithMan() {
        System.out.println("贾氏正在Happy中......");

    }

    public void makeEyesWithMan() {
        System.out.println("贾氏抛媚眼");

    }

}
```

西门庆勾贾氏：

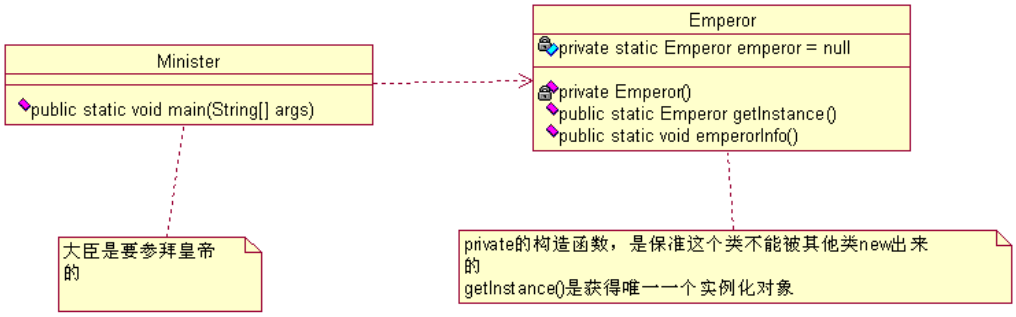
```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个西门庆，这人色中饿鬼
 */
public class XiMenQing {

    public static void main(String[] args) {
        //改编一下历史，贾氏被西门庆勾走：
        JiaShi jiaShi = new JiaShi();
        WangPo wangPo = new WangPo(jiaShi); //让王婆作为贾氏的代理人

        wangPo.makeEyesWithMan();
        wangPo.happyWithMan();
    }
}
```

说完这个故事，那额总结一下，代理模式主要使用了 Java 的多态，干活的是被代理类，代理类主要是接活，你让我干活，好，我交给幕后的类去干，你满意就成，那怎么知道被代理类能不能干呢？同根就成，大家知根知底，你能做啥，我能做啥都清楚的很，同一个接口呗。

3. 单例模式【Singleton Pattern】

这个模式是很有意思，而且比较简单，但是我还是要说因为它使用的是如此的广泛，如此的有人缘，单例就是单一、独苗的意思，那什么是独一份呢？你的思维是独一份，除此之外还有什么不能山寨的呢？我们举个比较难复制的对象：皇帝

中国的历史上很少出现两个皇帝并存的时期，是有，但不多，那我们就认为皇帝是个单例模式，在这个场景中，有皇帝，有大臣，大臣是天天要上朝参见皇帝的，今天参拜的皇帝应该和昨天、前天的一样（过渡期的不考虑，别找茬哦），大臣磕完头，抬头一看，嗨，还是昨天那个皇帝，单例模式，绝对的单例模式，先看类图：



然后我们看程序实现，先定一个皇帝：

```
package com.cbf4life.singleton1;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 中国的历史上一般都是一个朝代一个皇帝，有两个皇帝的话，必然要PK出一个皇帝出来
 */
public class Emperor {
    private static Emperor emperor = null;  //定义一个皇帝放在那里，然后给这个皇帝名字

    private Emperor(){
        //世俗和道德约束你，目的就是不让你产生第二个皇帝
    }
```

```
    public static Emperor getInstance(){
        if(emperor == null){  //如果皇帝还没有定义，那就定一个
            emperor = new Emperor();
        }
        return emperor;
    }

    //皇帝叫什么名字呀
    public static void emperorInfo(){
        System.out.println("我就是皇帝某某某....");
    }
}
```

然后定义大臣：

```
package com.cbf4life.singleton1;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 大臣是天天要面见皇帝，今天见的皇帝和昨天的，前天不一样那就出问题了！
 */
@SuppressWarnings("all")
public class Minister {

    /**
     * @param args
     */
    public static void main(String[] args) {
        //第一天
        Emperor  emperor1=Emperor.getInstance();
        emperor1.emperorInfo();  //第一天见的皇帝叫什么名字呢？

        //第二天
        Emperor  emperor2=Emperor.getInstance();
        Emperor.emperorInfo();

        //第三天
        Emperor  emperor3=Emperor.getInstance();
        emperor2.emperorInfo();

        //三天见的皇帝都是同一个人，荣幸吧！
    }
```

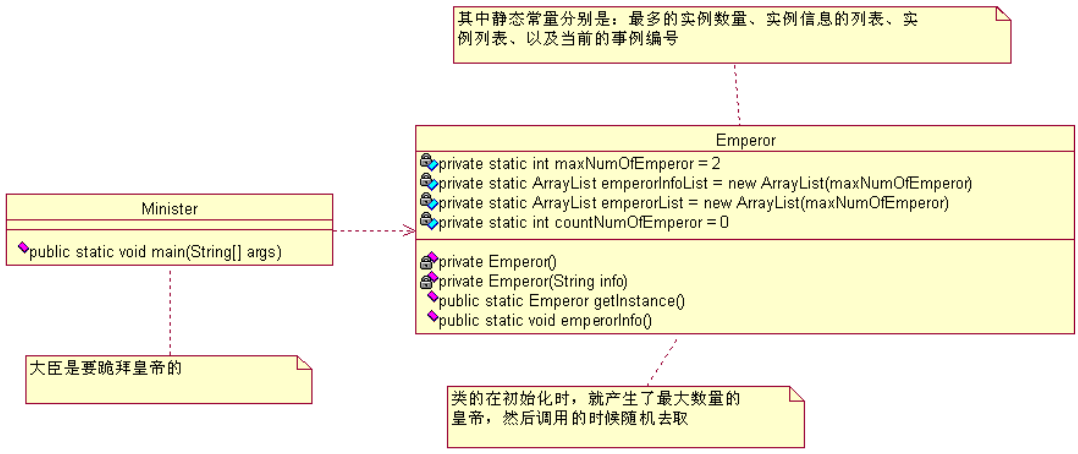
```
}
```

看到没，大臣天天见到的都是同一个皇帝，不会产生错乱情况，反正都是一个皇帝，是好是坏就这一个，只要提到皇帝，大家都知道指的是谁，清晰，而又明确。问题是这是通常情况，还有个例的，如同一个时期同一个朝代有两个皇帝，怎么办？

4. 多例模式【Multition Pattern】

这种情况有没有？有！大点声，有没有？ 有!，是，确实有，就出现在明朝，那三国期间的算不算，不算，各自称帝，各有各的地盘，国号不同。大家还记得那首诗《石灰吟》吗？作者是谁？于谦，他是被谁杀死的？明英宗朱祁镇，对，就是那个在土木堡之变中被瓦剌俘虏的皇帝，被俘虏后，他弟弟朱祁钰当上了皇帝，就是明景帝，估计当上皇帝后乐疯了，忘记把老哥朱祁镇削为太上皇了，我 Shit，在中国的历史上就这个时期是有 2 个皇帝，你说这期间的大臣多郁闷，两个皇帝耶，两个精神依附对象呀。

这个场景放到我们设计模式中就是叫有上限的多例模式（没上限的多例模式太容易了，和你直接 new 一个对象没啥差别，不讨论）怎么实现呢，看我出招，先看类图：



然后看程序，先把两个皇帝定义出来：

```
package com.cbf4life.singleton2;

import java.util.ArrayList;
import java.util.Random;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 中国的历史上一般都是一个朝代一个皇帝，有两个皇帝的话，必然要PK出一个皇帝出来。
 * 问题出来了：如果真在一个时间，中国出现了两个皇帝怎么办？比如明朝土木堡之变后，
 * 明英宗被俘虏，明景帝即位，但是明景帝当上皇帝后乐疯了，竟然忘记把他老哥明英宗
```

```
削为太上皇，
 * 也就是在这一个多月的时间内，中国竟然有两个皇帝！
 *
 */
@SuppressWarnings("all")
public class Emperor {
    private static int maxNumOfEmperor = 2;  //最多只能有连个皇帝
    private static ArrayList emperorInfoList=new
ArrayList(maxNumOfEmperor);  //皇帝叫什么名字
    private static ArrayList emperorList=new
ArrayList(maxNumOfEmperor); //装皇帝的列表；
    private static int countNumOfEmperor =0;  //正在被人尊称的是那个皇
帝

    //先把2个皇帝产生出来
    static{
        //把所有的皇帝都产生出来
        for(int i=0;i<maxNumOfEmperor;i++){
            emperorList.add(new Emperor("皇"+(i+1)+"帝"));
        }
    }

    //就这么多皇帝了，不允许再推举一个皇帝(new 一个皇帝）
    private Emperor(){
        //世俗和道德约束你，目的就是不让你产生第二个皇帝
    }

    private Emperor(String info){
        emperorInfoList.add(info);
    }

    public static Emperor getInstance(){
        Random random = new Random();
        countNumOfEmperor = random.nextInt(maxNumOfEmperor);  //随机
拉出一个皇帝，只要是个精神领袖就成
        return (Emperor)emperorList.get(countNumOfEmperor);
    }

    //皇帝叫什么名字呀
    public static void emperorInfo(){
       System.out.println(emperorInfoList.get(countNumOfEmperor));

    }
}
```

那大臣是比较悲惨了，两个皇帝呀，两个老子呀，怎么拜呀，不管了，只要是个皇帝就成：

```
package com.cbf4life.singleton2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 大臣们悲惨了，一个皇帝都伺候不过来了，现在还来了两个个皇帝
 * TND，不管了，找到个皇帝，磕头，请按就成了！
 */
@SuppressWarnings("all")
public class Minister {

    /**
     * @param args
     */
    public static void main(String[] args) {

        int ministerNum =10; //10个大臣

        for(int i=0;i<ministerNum;i++){
            Emperor emperor = Emperor.getInstance();
            System.out.print("第"+(i+1)+"个大臣参拜的是：");
            emperor.emperorInfo();
        }
    }

}
```

那各位看官就可能会不屑了：有的大臣可是有骨气，只拜一个真神，你怎么处理？这个问题太简单，懒的详细回答你，getInstance(param)是不是就解决了这个问题？！自己思考，太 Easy 了。

5. 工厂方法【Factory Method Pattern】

女娲补天的故事大家都听说过吧，今天不说这个，说女娲创造人的故事，可不是“造人”的工作，这个词被现代人滥用了。这个故事是说，女娲在补了天后，下到凡间一看，哇塞，风景太优美了，天空是湛蓝的，水是清澈的，空气是清新的，太美丽了，然后就待时间长了就有点寂寞了，没有动物，这些看的到都是静态的东西呀，怎么办？

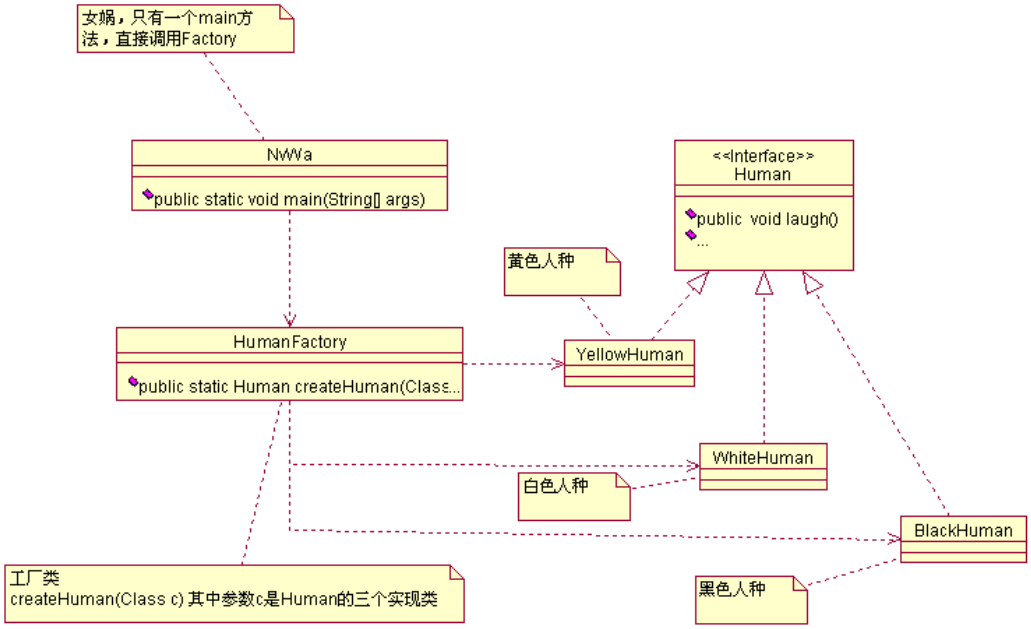
别忘了是神仙呀，没有办不到的事情，于是女娲就架起了八卦炉（技术术语：建立工厂）开始创建人，具体过程是这样的：先是泥巴捏，然后放八卦炉里烤，再扔到地上成长，但是意外总是会产生的：

第一次烤泥人，兹兹兹兹~~，感觉应该熟了，往地上一扔，biu~，一个白人诞生了，没烤熟！

第二次烤泥人，兹兹兹兹兹兹兹兹~~，上次都没烤熟，这次多烤会儿，往地上一扔，嘿，熟过头了，黑人哪！

第三次烤泥人，兹~兹~兹~，一边烤一边看着，嘿，正正好，Perfect，优品，黄色人种！【备注:RB 人不再此列】

这个过程还是比较有意思的，先看看类图：（之前在论坛上有兄弟建议加类图和源文件，以后的模式都会加上去，之前的会一个一个的补充，目的是让大家看着舒服，看着愉悦，看着就想要，就像是看色情小说一样，目标，目标而已，能不能实现就看大家给我的信心了）



那这个过程我们就用程序来表现，首先定义一个人类的总称：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个人类的统称
 */
public interface Human {
    //首先定义什么是人类

     //人是愉快的，会笑的，本来是想用smile表示，想了一下laugh更合适，好长时间没有大笑了；
    public  void laugh();

    //人类还会哭，代表痛苦
    public void cry();

    //人类会说话
    public void talk();

}
```

然后定义具体的人种：

```java
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 黄色人种，这个翻译的不准确，将就点吧
 */
public class YellowHuman implements Human {


    public void cry() {
        System.out.println("黄色人种会哭");
    }


    public void laugh() {
        System.out.println("黄色人种会大笑，幸福呀！");
    }


    public void talk() {
        System.out.println("黄色人种会说话，一般说的都是双字节");

    }

}
```

白色人种：

```java
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 白色人种
 */
public class WhiteHuman implements Human {

    public void cry() {
        System.out.println("白色人种会哭");
    }

    public void laugh() {
        System.out.println("白色人种会大笑，侵略的笑声");
```

```
    }

    public void talk() {
        System.out.println("白色人种会说话，一般都是但是单字节！");
    }

}
```

黑色人种：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 黑色人种，记得中学学英语，老师说black man是侮辱人的意思，不懂，没跟老外说
话
 */
public class BlackHuman implements Human {

    public void cry() {
        System.out.println("黑人会哭");
    }

    public void laugh() {
        System.out.println("黑人会笑");
    }

    public void talk() {
        System.out.println("黑人可以说话，一般人听不懂");

    }
}
```

人种也定义完毕了，那我们把八卦炉定义出来：

```
package com.cbf4life;

import java.util.List;
import java.util.Random;

/**
 * @author cbf4Life cbf4life@126.com
```

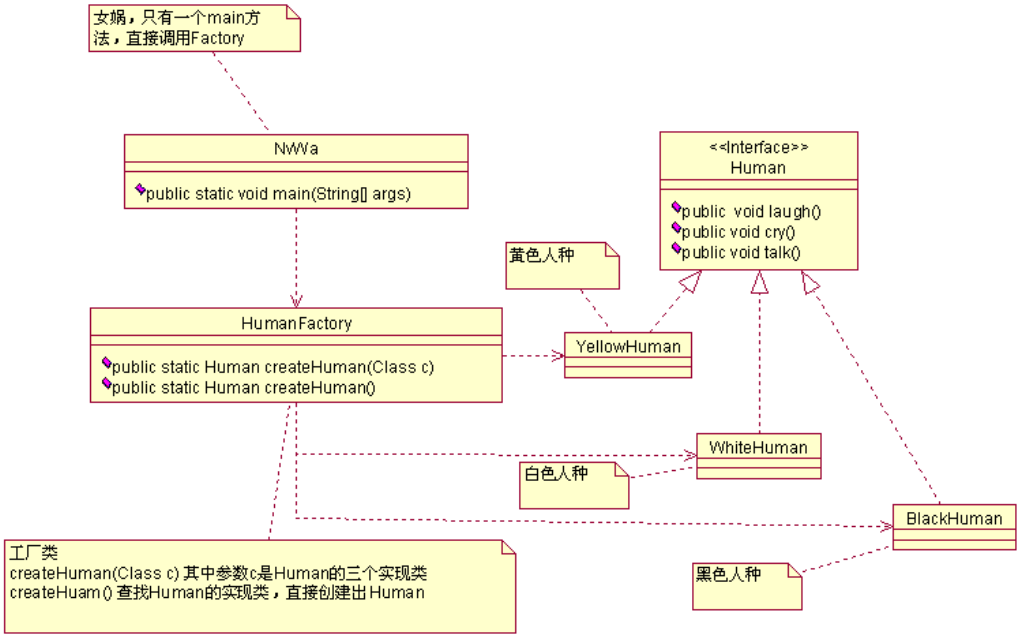
```
 * I'm glad to share my knowledge with you all.
 * 今天讲女娲造人的故事，这个故事梗概是这样的：
 * 很久很久以前，盘古开辟了天地，用身躯造出日月星辰、山川草木，天地一片繁华
 * One day，女娲下界走了一遭，哎！太寂寞，太孤独了，没个会笑的、会哭的、会说话的东东
 * 那怎么办呢？不用愁，女娲，神仙呀，造出来呀，然后捏泥巴，放八卦炉（后来这个成了太白金星的宝贝）中烤，于是就有了人：
 * 我们把这个生产人的过程用Java程序表现出来：
 */
public class HumanFactory {

    //定一个烤箱，泥巴塞进去，人就出来，这个太先进了
    public static Human createHuman(Class c){
        Human human=null;  //定义一个类型的人类

        try {
            human = (Human)Class.forName(c.getName()).newInstance();
//产生一个人种

        } catch (InstantiationException e) {//你要是不说个人种颜色的话，没法烤，要白的黑，你说话了才好烤

            System.out.println("必须指定人种的颜色");
        } catch (IllegalAccessException e) { //定义的人种有问题，那就烤不出来了，这是...

            System.out.println("人种定义错误！");
        } catch (ClassNotFoundException e) { //你随便说个人种，我到哪里给你制造去？！

            System.out.println("混蛋，你指定的人种找不到！");
        }
        return human;
    }

}
```

然后我们再把女娲声明出来：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
```

```
 * 首先定义女娲，这真是额的神呀
 */
public class NvWa {

    public static void main(String[] args) {

        //女娲第一次造人，试验性质，少造点，火候不足，缺陷产品
        System.out.println("------------造出的第一批人是这样的：白人
----------------");
        Human whiteHuman =
HumanFactory.createHuman(WhiteHuman.class);
        whiteHuman.cry();
        whiteHuman.laugh();
        whiteHuman.talk();

        //女娲第二次造人，火候加足点，然后又出了个次品，黑人
        System.out.println("\n\n------------造出的第二批人是这样的：黑人
----------------");
        Human blackHuman =
HumanFactory.createHuman(BlackHuman.class);
        blackHuman.cry();
        blackHuman.laugh();
        blackHuman.talk();

        //第三批人了，这次火候掌握的正好，黄色人种（不写黄人，免得引起歧义），
备注：RB人不属于此列
        System.out.println("\n\n------------造出的第三批人是这样的：黄色
人种----------------");
        Human yellowHuman =
HumanFactory.createHuman(YellowHuman.class);
        yellowHuman.cry();
        yellowHuman.laugh();
        yellowHuman.talk()

    }

}
```

这样这个世界就热闹起来了，人也有了，但是这样创建太累了，神仙也会累的，那怎么办？神仙就想了：我塞进去一团泥巴，随机出来一群人，管他是黑人、白人、黄人，只要是人就成（你看看，神仙都偷懒，何况是我们人），先修改类图：



然后看我们的程序修改，先修改 HumanFactory.java，增加了 createHuman()方法：

```
package com.cbf4life;

import java.util.List;
import java.util.Random;
public class HumanFactory {

    //定一个烤箱，泥巴塞进去，人就出来，这个太先进了
    public static Human createHuman(Class c){
        Human human=null;  //定义一个类型的人类

        try {
            human = (Human)Class.forName(c.getName()).newInstance();
//产生一个人种

        } catch (InstantiationException e) {//你要是不说个人种颜色的话，
没法烤，要白的黑，你说话了才好烤

            System.out.println("必须指定人种的颜色");
        } catch (IllegalAccessException e) { //定义的人种有问题，那就烤
不出来了，这是...

            System.out.println("人种定义错误！");
        } catch (ClassNotFoundException e) { //你随便说个人种，我到哪里
给你制造去？！
```

```
            System.out.println("混蛋，你指定的人种找不到！");
        }
        return human;
    }

    //女娲生气了，把一团泥巴塞到八卦炉，哎产生啥人种就啥人种
    public static Human createHuman(){
        Human human=null;  //定义一个类型的人类

        //首先是获得有多少个实现类，多少个人种
        List<Class> concreteHumanList =
ClassUtils.getAllClassByInterface(Human.class);  //定义了多少人种
        //八卦炉自己开始想烧出什么人就什么人
        Random random = new Random();
        int rand = random.nextInt(concreteHumanList.size());

        human = createHuman(concreteHumanList.get(rand));

        return human;
    }
}
```

然后看女娲是如何做的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 首先定义女娲，这真是额的神呀
 */
public class NvWa {

    public static void main(String[] args) {

        //女娲第一次造人，试验性质，少造点，火候不足，缺陷产品
        System.out.println("------------造出的第一批人是这样的：白人
-----------------");
        Human whiteHuman =
HumanFactory.createHuman(WhiteHuman.class);
        whiteHuman.cry();
        whiteHuman.laugh();
        whiteHuman.talk();
```

```
        //女娲第二次造人，火候加足点，然后又出了个次品，黑人
        System.out.println("\n\n------------造出的第二批人是这样的：黑人
-----------------");
        Human blackHuman =
HumanFactory.createHuman(BlackHuman.class);
        blackHuman.cry();
        blackHuman.laugh();
        blackHuman.talk();

        //第三批人了，这次火候掌握的正好，黄色人种（不写黄人，免得引起歧义），
备注：RB人不属于此列
        System.out.println("\n\n------------造出的第三批人是这样的：黄色
人种-----------------");
        Human yellowHuman =
HumanFactory.createHuman(YellowHuman.class);
        yellowHuman.cry();
        yellowHuman.laugh();
        yellowHuman.talk();

        //女娲烦躁了，爱是啥人种就是啥人种，烧吧

        for(int i=0;i<10000000000;i++){
            System.out.println("\n\n------------随机产生人种了
-----------------" + i);
            Human human = HumanFactory.createHuman();
            human.cry();
            human.laugh();
            human.talk();
        }

    }

}
```

哇，这个世界热闹了!，不过还没有完毕，这个程序你跑不起来，还要有这个类：

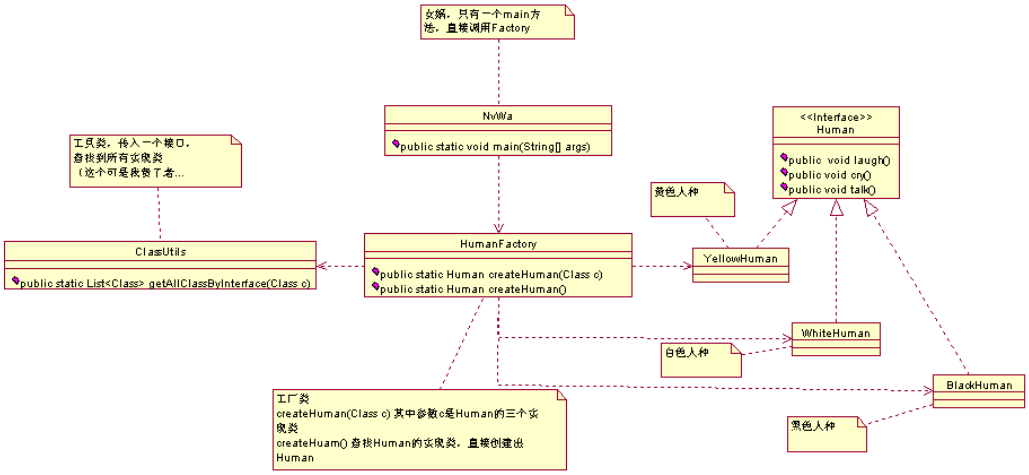
```
package com.cbf4life;

import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Enumeration;
```

```
import java.util.List;

/**
 * @author cbf4Life cbf4life@126.com I'm glad to share my knowledge
with you
 *        all.
 *
 */
@SuppressWarnings("all")
public class ClassUtils {


    //给一个接口，返回这个接口的所有实现类
    public static List<Class> getAllClassByInterface(Class c){
        List<Class> returnClassList = new ArrayList<Class>();  //返
回结果

        //如果不是一个接口，则不做处理
        if(c.isInterface()){
            String packageName = c.getPackage().getName();  //获得当
前的包名
            try {
                List<Class>  allClass = getClasses(packageName);  //
获得当前包下以及子包下的所有类

                //判断是否是同一个接口
                for(int i=0;i<allClass.size();i++){
                    if(c.isAssignableFrom(allClass.get(i))){  //判断
是不是一个接口
                        if(!c.equals(allClass.get(i))){   //本身不加进
去
                            returnClassList.add(allClass.get(i));
                        }
                    }
                }
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        return returnClassList;
    }
```

```
    //从一个包中查找出所有的类，在jar包中不能查找
    private static List<Class> getClasses(String packageName)
            throws ClassNotFoundException, IOException {
        ClassLoader classLoader = Thread.currentThread()
                .getContextClassLoader();
        String path = packageName.replace('.', '/');
        Enumeration<URL> resources = classLoader.getResources(path);
        List<File> dirs = new ArrayList<File>();
        while (resources.hasMoreElements()) {
            URL resource = resources.nextElement();
            dirs.add(new File(resource.getFile()));
        }
        ArrayList<Class> classes = new ArrayList<Class>();
        for (File directory : dirs) {
            classes.addAll(findClasses(directory, packageName));
        }
        return classes;
    }


    private static List<Class> findClasses(File directory, String
packageName) throws ClassNotFoundException {
        List<Class> classes = new ArrayList<Class>();
        if (!directory.exists()) {
            return classes;
        }
        File[] files = directory.listFiles();
        for (File file : files) {
            if (file.isDirectory()) {
                assert !file.getName().contains(".");
                classes.addAll(findClasses(file, packageName + "." +
file.getName()));
            } else if (file.getName().endsWith(".class")) {
                classes.add(Class.forName(packageName + '.' +
file.getName().substring(0, file.getName().length() - 6)));
            }
        }
        return classes;
    }

}
```

告诉你了，这个 ClassUtils 可是个宝，用处可大了去了，可以由一个接口查找到所有

的实现类，也可以由父类查找到所有的子类，这个要自己修改一下，动脑筋想下，简单的很！完整的类图如下：



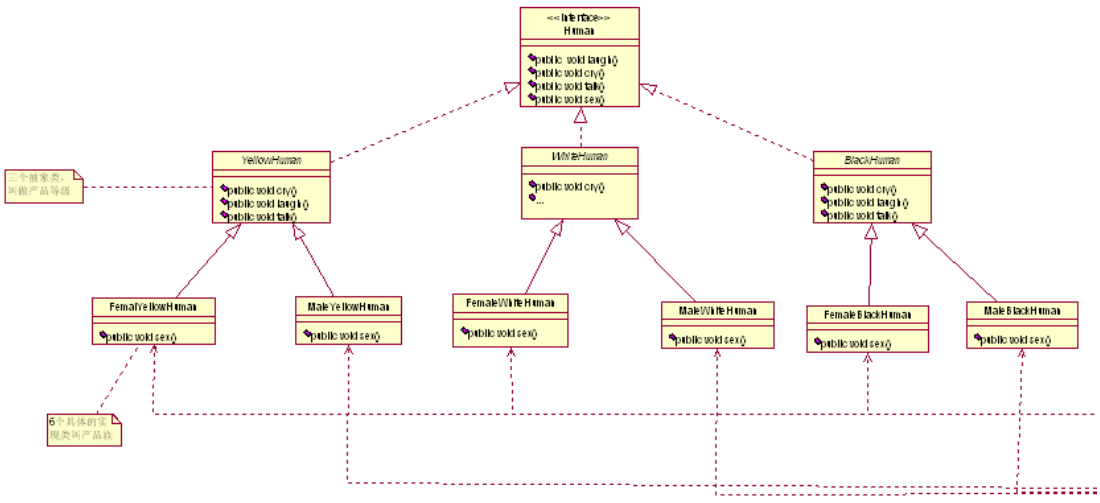
我们来总结一下，特别是增加了 createHuman()后，是不是这个工厂的扩展性更好了？你看你要再加一个人种，只要你继续集成 Human 接口成了，然后啥都不用修改就可以生产了，具体产多少，那要八卦炉说了算，简单工厂模式就是这么简单，那我们再引入一个问题：人是有性别的呀，有男有女，你这怎么没区别，别急，这个且听下回分解！

6. 抽象工厂模式【Abstract Factory Pattern】

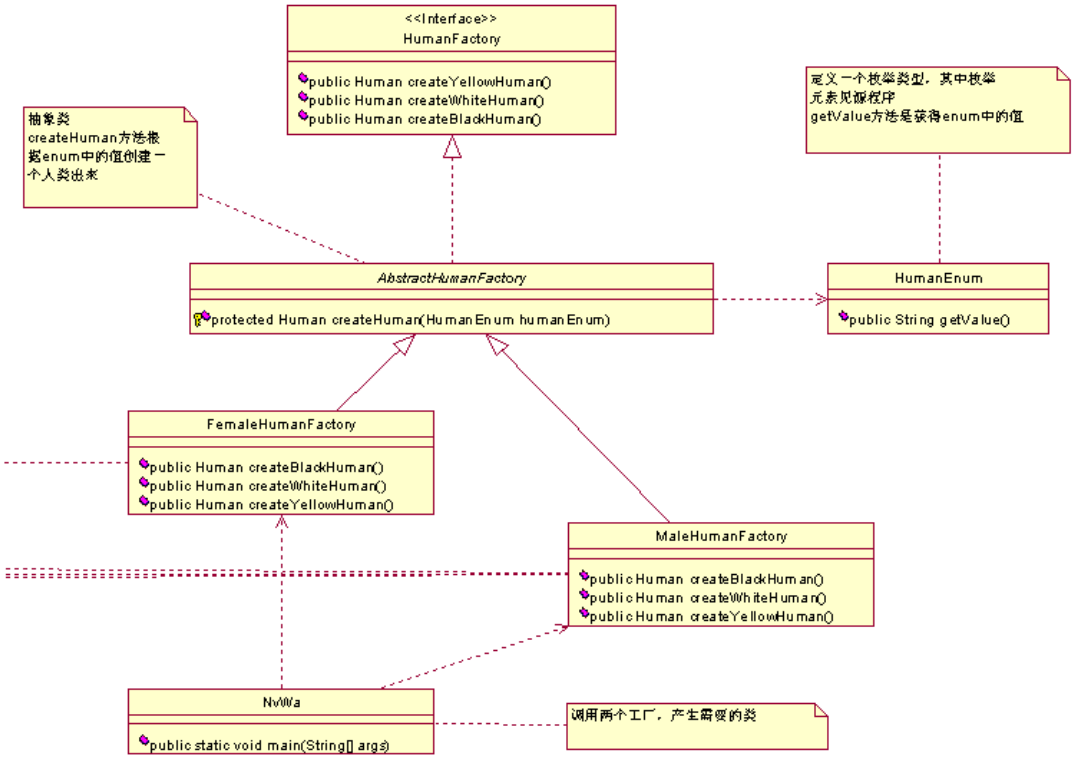
好了，我们继续上一节课，上一节讲到女娲造人，人是造出来了，世界时热闹了，可是低头一看，都是清一色的类型，缺少关爱、仇恨、喜怒哀乐等情绪，人类的生命太平淡了，女娲一想，猛然一拍脑袋，Shit！忘记给人类定义性别了，那怎么办？抹掉重来，然后就把人类重新洗牌，准备重新开始制造人类。

由于先前的工作已经花费了很大的精力做为铺垫，也不想从头开始了，那先说人类（Product 产品类）怎么改吧，好，有了，给每个人类都加一个性别，然后再重新制造，这个问题解决了，那八卦炉怎么办？只有一个呀，要么生产出全都是男性，要不都是女性，那不行呀，有了，把已经有了一条生产线——八卦炉（工厂模式中的 Concrete Factory）拆开，于是女娲就使用了“八卦拷贝术”，把原先的八卦炉一个变两个，并且略加修改，就成了女性八卦炉（只生产女性，一个具体工厂的实现类）和男性八卦炉（只生产男性，又一个具体工厂的实现类），这个过程的类图如下：

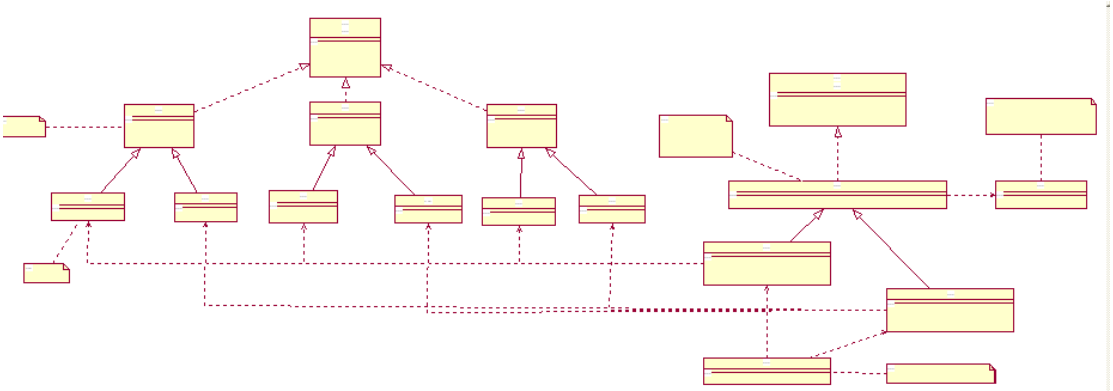
先看人类（也就是产品）的类图：



这个类图也比较简单，Java 的典型类图，一个接口，几个抽象类，然后是几个实现类，没啥多说的，其中三个抽象类在抽象工厂模式中是叫做产品等级，六个实现类是叫做产品族，这个也比较好理解，实现类嘛是真实的产品，一个叫产品，多了就叫产品族，然后再看工厂类：



其中抽象工厂只实现了一个 createHuman 的方法，目的是简化实现类的代码工作量，这个在讲代码的时候会说。这里还使用了 Jdk 1.5 的一个新特性 Enum 类型，其实这个完全可以类的静态变量来实现，但我想既然是学习就应该学有所获得，即使你对这个模式非常了解，也可能没用过 Enum 类型，也算是一个不同的知识点吧，我希望给大家讲解，每次都有新的技术点提出来，每个人都会有一点的收获就足够了，然后在具体的项目中使用时，知道有这个技术点，然后上 baidu 狗狗一下就能解决问题。话题扯远了，我们继续类图，完整的类图如下，这个看不大清楚，其实就是上面那两个类图加起来，大家可以看源码中的那个类图文件：



然后类图讲解完毕，我们来看程序实现：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个人类的统称，问题出来了，刚刚定义的时候忘记定义性别了
 * 这个重要的问题非修改不可，否则这个世界上太多太多的东西不存在了
 */
public interface Human {
    //首先定义什么是人类

     //人是愉快的，会笑的，本来是想用smile表示，想了一下laugh更合适，好长时间没有大笑了；
    public  void laugh();

    //人类还会哭，代表痛苦
    public void cry();

    //人类会说话
    public void talk();

    //定义性别
    public void sex();
}
```

人类的接口定义好，然后根据接口创建三个抽象类，也就是三个产品等级，实现laugh()、cry()、talk()三个方法，以 AbstractYellowHuman 为例：

```
package com.cbf4life.yellowHuman;

import com.cbf4life.Human;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 为什么要修改成抽象类呢？要定义性别呀
 */
public abstract class AbstractYellowHuman implements Human {


    public void cry() {
        System.out.println("黄色人种会哭");
```

```
    }


    public void laugh() {
        System.out.println("黄色人种会大笑，幸福呀！");
    }


    public void talk() {
        System.out.println("黄色人种会说话，一般说的都是双字节");

    }


}
```

其他的两个抽象类AbstractWhiteHuman和AbstractgBlackHuman与此类似的事项方法，不再通篇拷贝代码，大家可以看一下源代码。算了，还是拷贝，反正是电子档的，不想看就往下翻页，也成就了部分“懒人”，不用启动Eclipse，还要把源码拷贝进来：

白种人的抽象类：

```
package com.cbf4life.whiteHuman;

import com.cbf4life.Human;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 白色人人种
 * 为了代码整洁，新建一个包，这里是白种人的天下了
 */
public abstract class AbstractWhiteHuman implements Human {


    public void cry() {
        System.out.println("白色人种会哭");

    }


    public void laugh() {
        System.out.println("白色人种会大笑，侵略的笑声");
```

```
    }


    public void talk() {
        System.out.println("白色人种会说话，一般都是但是单字节！");

    }

}
```

黑种人的抽象类：

```
package com.cbf4life.blackHuman;

import com.cbf4life.Human;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 黑色人种，记得中学学英语，老师说black man是侮辱人的意思，不懂，没跟老外说
话
 */
public abstract class AbstractBlackHuman implements Human {

    public void cry() {
        System.out.println("黑人会哭");
    }

    public void laugh() {
        System.out.println("黑人会笑");
    }

    public void talk() {
        System.out.println("黑人可以说话，一般人听不懂");

    }

}
```

三个抽象类都实现完毕了，然后就是些实现类了。其实，你说抽象类放这里有什么意义吗？就是不允许你 new 出来一个抽象的对象呗，使用非抽象类完全就可以代替，呵呵，杀猪杀尾巴，各有各的杀法，不过既然进了 Java 这个门就要遵守 Java 这个规矩，我们看实现类：

女性黄种人的实现类：

```
package com.cbf4life.yellowHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 女性黄种人
 */
public class YellowFemaleHuman extends AbstractYellowHuman {

    public void sex() {
        System.out.println("该黄种人的性别为女...");
    }

}
```

男性黄种人的实现类：

```
package com.cbf4life.yellowHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 男性黄种人
 */
public class YellowMaleHuman extends AbstractYellowHuman {

    public void sex() {
        System.out.println("该黄种人的性别为男....");
    }

}
```

女性白种人的实现类：

```
package com.cbf4life.whiteHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.\
 * 女性白种人
 */
public class WhiteFemaleHuman extends AbstractWhiteHuman {
```

```
    public void sex() {
        System.out.println("该白种人的性别为女....");

    }

}
```

男性白种人的实现类：

```
package com.cbf4life.whiteHuman;


/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 男性白种人
 */
public class WhiteMaleHuman extends AbstractWhiteHuman {

    public void sex() {
        System.out.println("该白种人的性别为男....");
    }

}
```

女性黑种人的实现类：

```
package com.cbf4life.blackHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 女性黑种人
 */
public class BlackFemaleHuman extends AbstractBlackHuman {

    public void sex() {
        System.out.println("该黑种人的性别为女...");
    }

}
```

男性黑种人的实现类：

```
package com.cbf4life.blackHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 男性黑种人
 */
public class BlackMaleHuman extends AbstractBlackHuman {

    public void sex() {
        System.out.println("该黑种人的性别为男...");
    }

}
```

抽象工厂模式下的产品等级和产品族都已经完成，也就是人类以及产生出的人类是什么样子的都已经定义好了，下一步就等着工厂开工创建了，那我们来看工厂类。

在看工厂类之前我们先看那个枚举类型，这个是很有意思的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 世界上有哪些类型的人，列出来
 * JDK 1.5开始引入enum类型也是目的的，吸引C程序员转过来
 */
public enum HumanEnum {

    //把世界上所有人类型都定义出来
    YelloMaleHuman("com.cbf4life.yellowHuman.YellowMaleHuman"),

    YelloFemaleHuman("com.cbf4life.yellowHuman.YellowFemaleHuman")
,

    WhiteFemaleHuman("com.cbf4life.whiteHuman.WhiteFemaleHuman"),

    WhiteMaleHuman("com.cbf4life.whiteHuman.WhiteMaleHuman"),

    BlackFemaleHuman("com.cbf4life.blackHuman.BlackFemaleHuman"),
```

```
    BlackMaleHuman("com.cbf4life.blackHuman.BlackMaleHuman");


    private String value = "";
    //定义构造函数，目的是Data(value)类型的相匹配
    private HumanEnum(String value){
        this.value = value;
    }

    public String getValue(){
        return this.value;
    }


    /*
     * java enum类型尽量简单使用，尽量不要使用多态、继承等方法
     * 毕竟用Clas完全可以代替enum
     */


}
```

我之所以引入 Enum 这个类型，是想让大家在看这本书的时候能够随时随地的学到点什么，你如果看不懂设计模式，你可以从我的程序中学到一些新的技术点，不用像我以前报着砖头似的书在那里啃，看一遍不懂，再看第二遍，然后翻了英文原本才知道，哦~，原来是这样滴，只能说有些翻译家实在不懂技术。我在讲解技术的时候，尽量少用专业术语，尽量使用大部分人类都能理解的语言。

Enum 以前我也很少用，最近在一个项目中偶然使用上了，然后才发觉它的好处，Enum 类型作为一个参数传递到一个方法中时，在 Junit 进行单元测试的时候，不用判断输入参数是否为空、长度为 0 的边界异常条件，如果方法传入的参数不是 Enum 类型的话，根本就传递不进来，你说定义一个类，定义一堆的静态变量，这也可以呀，这个不和你抬杠，上面的代码我解释一下，构造函数没啥好说的，然后是 getValue()方法，就是获得枚举类型中一个元素的值，枚举类型中的元素也是有名称和值的，这个和 HashMap 有点类似。

然后，我们看我们的工厂类，先看接口：

```
package com.cbf4life;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这次定一个接口，应该要造不同性别的人，需要不同的生产线
 * 那这个八卦炉必须可以制造男人和女人
 */
public interface HumanFactory {

    //制造黄色人种
    public Human createYellowHuman();

    //制造一个白色人种
    public Human createWhiteHuman();

    //制造一个黑色人种
    public Human createBlackHuman();
}
```

然后看抽象类：

```
package com.cbf4life.humanFactory;

import com.cbf4life.Human;
import com.cbf4life.HumanEnum;
import com.cbf4life.HumanFactory;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 编写一个抽象类，根据enum创建一个人类出来
 */
public abstract class AbstractHumanFactory implements HumanFactory
{

    /*
     * 给定一个性别人种，创建一个人类出来 专业术语是产生产品等级
     */
    protected Human createHuman(HumanEnum humanEnum) {
        Human human = null;

        //如果传递进来不是一个Enum中具体的一个Element的话，则不处理
        if (!humanEnum.getValue().equals("")) {
            try {
                //直接产生一个实例
```

```
                human = (Human)
Class.forName(humanEnum.getValue()).newInstance();
            } catch (Exception e) {
                //因为使用了enum，这个种异常情况不会产生了，除非你的enum有问
题；
                e.printStackTrace();
            }
        }
        return human;
    }

}
```

看到没，这就是引入 enum 的好处，createHuman(HumanEnum humanEnum)这个方法定义了输入参数必须是 HumanEnum 类型，然后直接使用 humanEnum.getValue()方法就能获得具体传递进来的值，这个不多说了，大家自己看程序领会，没多大难度，这个抽象类的目的就是减少下边实现类的代码量，我们看实现类：

男性工厂，只创建男性：

```
package com.cbf4life.humanFactory;

import com.cbf4life.Human;
import com.cbf4life.HumanEnum;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 男性创建工厂
 */
public class MaleHumanFactory extends AbstractHumanFactory {

    //创建一个男性黑种人
    public Human createBlackHuman() {
        return super.createHuman(HumanEnum.BlackMaleHuman);
    }

    //创建一个男性白种人
    public Human createWhiteHuman() {
        return super.createHuman(HumanEnum.WhiteMaleHuman);
    }

    //创建一个男性黄种人
```

```
    public Human createYellowHuman() {
        return super.createHuman(HumanEnum.YelloMaleHuman);
    }

}
```

女性工厂，只创建女性：

```
package com.cbf4life.humanFactory;

import com.cbf4life.Human;
import com.cbf4life.HumanEnum;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.\
 * 女性创建工厂
 */
public class FemaleHumanFactory extends AbstractHumanFactory {

    //创建一个女性黑种人
    public Human createBlackHuman() {
        return super.createHuman(HumanEnum.BlackFemaleHuman);
    }

    //创建一个女性白种人
    public Human createWhiteHuman() {
        return super.createHuman(HumanEnum.WhiteFemaleHuman);
    }

    //创建一个女性黄种人
    public Human createYellowHuman() {
        return super.createHuman(HumanEnum.YelloFemaleHuman);
    }

}
```

产品定义好了，工厂也定义好了，万事俱备只欠东风，那咱就开始造吧，哦，不对，女娲开始造人了：

```
package com.cbf4life;
```

```
import com.cbf4life.humanFactory.FemaleHumanFactory;
import com.cbf4life.humanFactory.MaleHumanFactory;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 女娲建立起了两条生产线，分别是：
 * 男性生产线
 * 女性生产线
 */
public class NvWa {

    public static void main(String[] args) {

        //第一条生产线，男性生产线
        HumanFactory maleHumanFactory = new MaleHumanFactory();

        //第二条生产线，女性生产线
        HumanFactory femaleHumanFactory = new FemaleHumanFactory();

        //生产线建立完毕，开始生产人了：
        Human maleYellowHuman =
maleHumanFactory.createYellowHuman();

        Human femaleYellowHuman =
femaleHumanFactory.createYellowHuman();

        maleYellowHuman.cry();
        maleYellowHuman.laugh();
        femaleYellowHuman.sex();
        /*
         * .....
         * 后面你可以续了
         */
    }
}
```

两个八卦炉，一个造女的，一个造男的，开足马力，一直造到这个世界到现在这个模式为止。

抽象工厂模式讲完了，那我们再思考一些问题：工厂模式有哪些优缺点？先说优点，我这人一般先看人优点，非常重要的有点就是，工厂模式符合 OCP 原则，也就是开闭原则，怎么说呢，比如就性别的问题，这个世界上还存在双性人，是男也是女的人，那这个就是要在

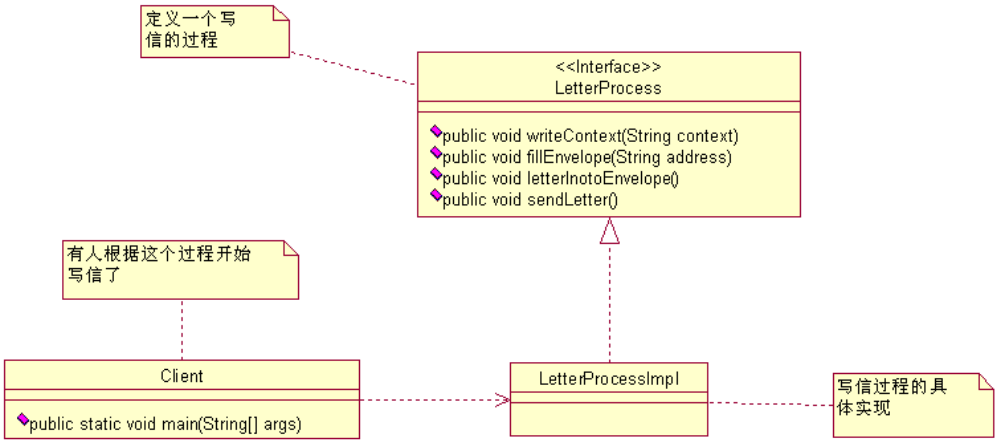
我们的产品族中增加一类产品，同时再增加一个工厂就可以解决这个问题，不需要我再来实现了吧，很简单的大家自己画下类图，然后实现下。

那还有没有其他好处呢？抽象工厂模式，还有一个非常大的有点，高内聚，低耦合，在一个较大的项目组，产品是由一批人定义开发的，但是提供其他成员访问的时候，只有工厂方法和产品的接口，也就是说只需要提供 Product Interface 和 Concrete Factory 就可以产生自己需要的对象和方法，Java 的高内聚低耦合的特性表现的一览无遗，哈哈。

7. 门面模式【Facade Pattern】

好，我们继续讲课。大家都是高智商的人，都写过纸质的信件吧，比如给女朋友写情书什么的，写信的过程大家都还记得吧，先写信的内容，然后写信封，然后把信放到信封中，封好，投递到信箱中进行邮递，这个过程还是比较简单的，虽然简单，这四个步骤都是要跑的呀，信多了还是麻烦，比如到了情人节，为了大海捞针，给十个女孩子发情书，都要这样跑一遍，你不要累死，更别说你要发个广告信啥的，一下子发 1 千万封邮件，那不就完蛋了？那怎么办呢？还好，现在邮局开发了一个新业务，你只要把信件的必要信息高速我，我给你发，我来做这四个过程，你就不要管了，只要把信件交给我就成了。

我们的类图还是从最原始的状态开始：



在这中环境下，最累的是写信的人，为了发送一封信出去要有四个步骤，而且这四个步骤还不能颠倒，你不可能没写信就把信放到信封吧，写信的人要知道这四个步骤，而且还要知道这四个步骤的顺序，恐怖吧，我们先看看这个过程如何表现出来的：

先看写信的过程接口，定义了写信的四个步骤：

```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个写信的过程
```

```
 */
public interface LetterProcess {

    //首先要写信的内容
    public void writeContext(String context);

    //其次写信封
    public void fillEnvelope(String address);

    //把信放到信封里
    public void letterInotoEnvelope();

    //然后邮递
    public void sendLetter();

}
```

写信过程的具体实现：

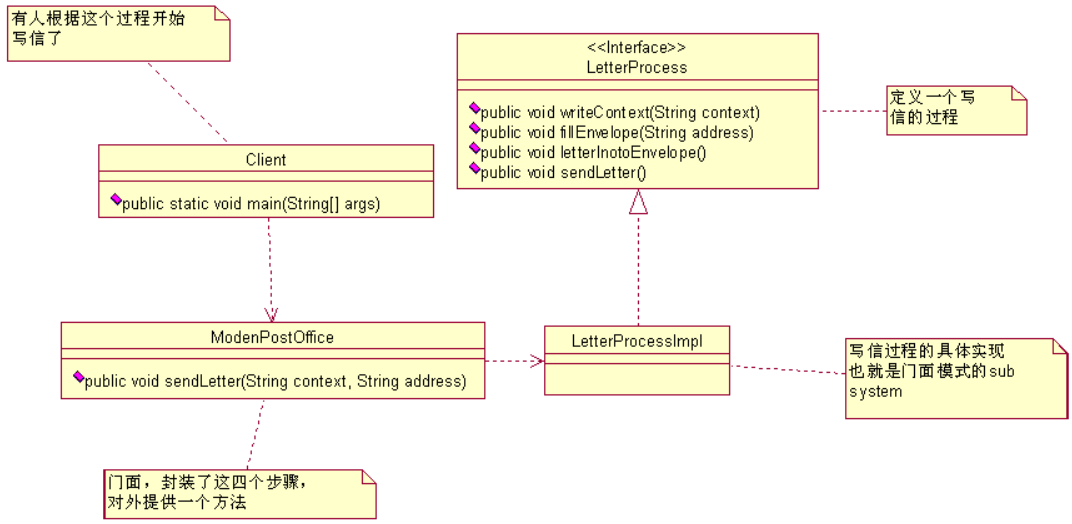
```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 写信的具体实现了
 */
public class LetterProcessImpl implements LetterProcess {

    //写信
    public void writeContext(String context) {
        System.out.println("填写信的内容...." + context);
    }

    //在信封上填写必要的信息
    public void fillEnvelope(String address) {
        System.out.println("填写收件人地址及姓名...." + address);
    }

    //把信放到信封中，并封好
    public void letterInotoEnvelope() {
        System.out.println("把信放到信封中....");
    }

    //塞到邮箱中，邮递
```

```
    public void sendLetter() {
        System.out.println("邮递信件...");
    }

}
```

然后就有人开始用这个过程写信了：

```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我开始给朋友写信了
 */
public class Client {

    public static void main(String[] args) {

        //创建一个处理信件的过程
        LetterProcess letterProcess = new LetterProcessImpl();

        //开始写信
        letterProcess.writeContext("Hello,It's me,do you know who I
am? I'm your old lover. I'd like to....");

        //开始写信封
        letterProcess.fillEnvelope("Happy Road No. 666,God
Province,Heaven");

        //把信放到信封里，并封装好
        letterProcess.letterInotoEnvelope();

        //跑到邮局把信塞到邮箱，投递
        letterProcess.sendLetter();


    }

}
```

那这个过程与高内聚的要求相差甚远，你想，你要知道这四个步骤，而且还要知道这四个步骤的顺序，一旦出错，信就不可能邮寄出去，那我们如何来改进呢？先看类图：



这就是门面模式，还是比较简单的，Sub System 比较复杂，为了让调用者更方便的调用，就对 Sub System 进行了封装，增加了一个门面，Client 调用时，直接调用门面的方法就可以了，不用了解具体的实现方法以及相关的业务顺序，我们来看程序的改变，LetterProcess 接口和实现类都没有改变，只是增加了一个 ModenPostOffice 类，我们这个 java 程序清单如下：

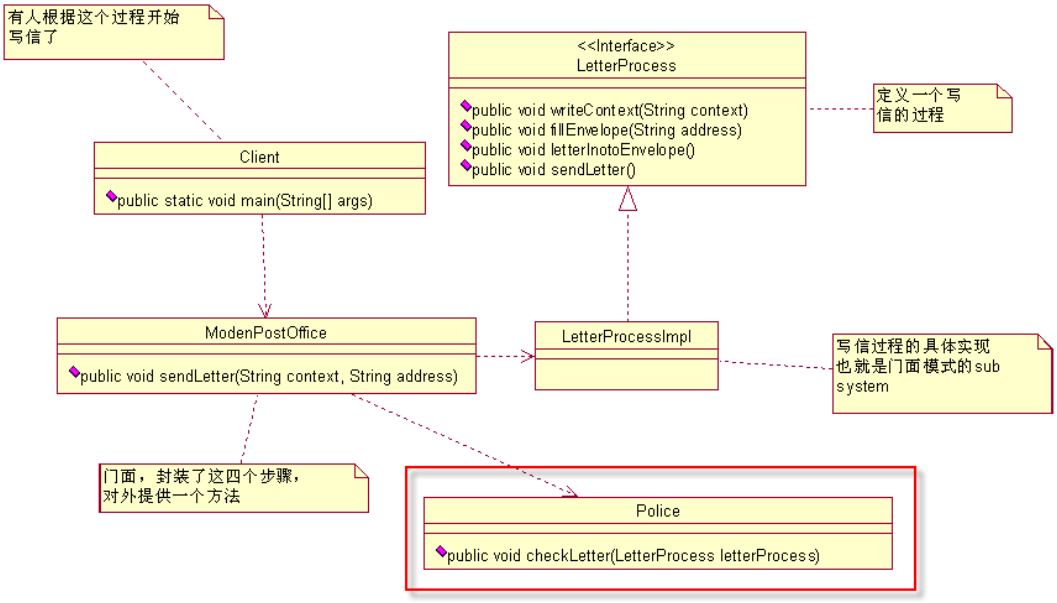
```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class ModenPostOffice {
    private LetterProcess letterProcess = new LetterProcessImpl();

    //写信，封装，投递，一体化了
    public void sendLetter(String context,String address){

        //帮你写信
        letterProcess.writeContext(context);

        //写好信封
        letterProcess.fillEnvelope(address);

        //把信放到信封中
        letterProcess.letterInotoEnvelope();
```

```
        //邮递信件
        letterProcess.sendLetter();

    }
}
```

这个类是什么意思呢，就是说现在又一个叫 Hell Road PostOffice（地狱路邮局）提供了一种新型的服务，客户只要把信的内容以及收信地址给他们，他们就会把信写好，封好，并发送出去，这种服务提出时大受欢迎呀，这简单呀，客户减少了很多工作，那我们看看客户是怎么调用的，Client.java 的程序清单如下：

```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我开始给朋友写信了
 */
public class Client {

    public static void main(String[] args) {
        //现代化的邮局，有这项服务，邮局名称叫Hell Road
        ModenPostOffice hellRoadPostOffice = new ModenPostOffice();

        //你只要把信的内容和收信人地址给他，他会帮你完成一系列的工作；
        String address = "Happy Road No. 666,God Province,Heaven"; //
定义一个地址
        String context = "Hello,It's me,do you know who I am? I'm your
old lover. I'd like to....";
        hellRoadPostOffice.sendLetter(context, address);

    }

}
```

看到没，客户简单了很多，提供这种模式后，系统的扩展性也有了很大的提高，突然一个非常时期，寄往 God Province（上帝省）的邮件都必须进行安全检查，那我们这个就很好处理了，看类图：



看这个红色的框，只增加了这一部分，其他部分在类图上都不需要改动，那我们来看源码：

```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class ModenPostOffice {
    private LetterProcess letterProcess = new LetterProcessImpl();
    private Police letterPolice = new Police();

    //写信，封装，投递，一体化了
    public void sendLetter(String context,String address){

        //帮你写信
        letterProcess.writeContext(context);

        //写好信封
        letterProcess.fillEnvelope(address);

        //警察要检查信件了
        letterPolice.checkLetter(letterProcess);

        //把信放到信封中
        letterProcess.letterInotoEnvelope();
```

```
            //邮递信件
            letterProcess.sendLetter();

        }
    }
```

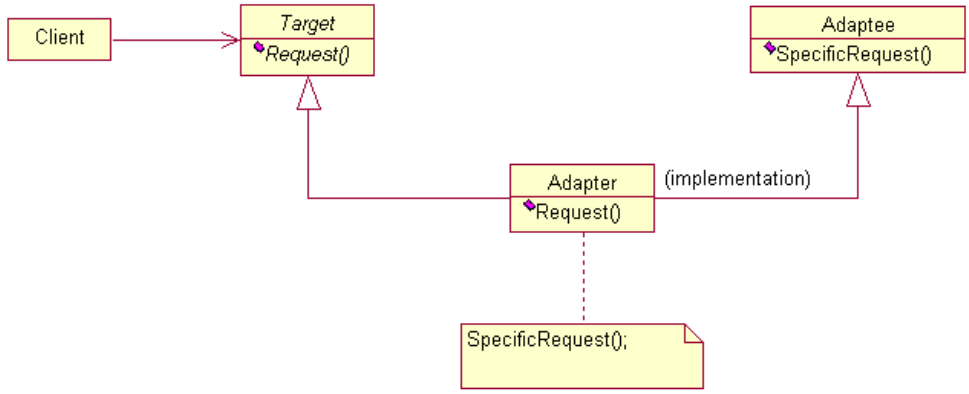
只是增加了一个 letterPolice 变量的声明以及一个方法的调用，那这个写信的过程就变成了这样：先写信，然后写信封，然后警察开始检查，然后才把信放到信封，然后发送出去，那这个变更对客户来说，是透明的，他根本就看不到有人在检查他的邮件，他也不用了解，反正现代化的邮件都帮他做了，这也是他乐意的地方。

门面模式讲解完毕，这是一个很好的封装方法，一个子系统比较复杂的实话，比如算法或者业务比较复杂，就可以封装出一个或多个门面出来，项目的结构简单，而且扩展性非常好。还有，在一个较大项目中的时候，为了避免人员带来的风险，也可以使用这个模式，技术水平比较差的成员，尽量安排独立的模块（Sub System），然后把他写的程序封装到一个门面里，尽量让其他项目成员不用看到这些烂人的代码，看也看不懂，我也遇到过一个“高人”写的代码，private 方法、构造函数、常量基本都不用，你要一个 public 方法，好，一个类里就一个 public 方法，所有代码都在里面，然后你就看吧，一大坨的程序，看着能把人逼疯，使用门面模式后，对门面进行单元测试，约束项目成员的代码质量，对项目整体质量的提升也是一个比较好的帮助。

8. 适配器模式【Adapter Pattern】

好，请安静，后排聊天的同学别吵醒前排睡觉的同学了，大家要相互理解嘛。今天讲适配器模式，这个模式也很简单，你笔记本上的那个拖在外面的黑盒子就是个适配器，一般你在中国能用，在日本也能用，虽然两个国家的的电源电压不同，中国是 220V，日本是 110V，但是这个适配器能够把这些不同的电压转换为你需要的 36V 电压，保证你的笔记本能够正常运行，那我们在设计模式中引入这个适配器模式是不是也是这个意思呢？是的，一样的作用，两个不同接口，有不同的实现，但是某一天突然上帝命令你把 B 接口转换为 A 接口，怎么办？继承，能解决，但是比较傻，而且还违背了 OCP 原则，怎么办？好在我们还有适配器模式。

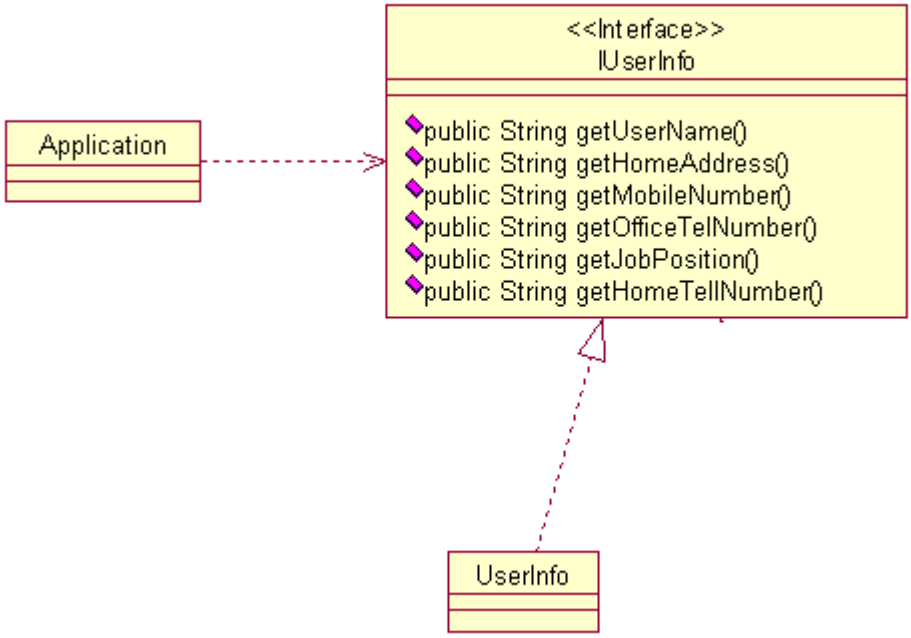
适配器的通用类图是这个样子滴：



首先声明，这个不是我画的，这是从 Rational Rose 的帮助文件中截取的，这个类图也很容易理解，Target 是一个类（接口），Adaptee 是一个接口，通过 Adapter 把 Adaptee 包装成 Target 的一个子类（实现类），注意了这里用了个名词包装（Wrapper），那其实这个模式也叫做包装模式(Wrapper)，但是包装模式可不知一个，还包括了以后要讲的装饰模式。我来讲个自己的一个经验，让大家可以更容易了解这个适配器模式，否则纯讲技术太枯燥了，技术也可以有娱乐的嘛。

我在 2004 年的时候带了一个项目，做一个人力资源管理，该项目是我们总公司发起的项目，公司一共有 700 多号人，包括子公司，这个项目还是比较简单的，分为三大模块：人员信息管理，薪酬管理，职位管理，其中人员管理这块就用到了适配器模式，是怎么回事呢？当时开发时明确的指明：人员信息简管理的对象是所有员工的所有信息，然后我们就这样设

计了一个类图：

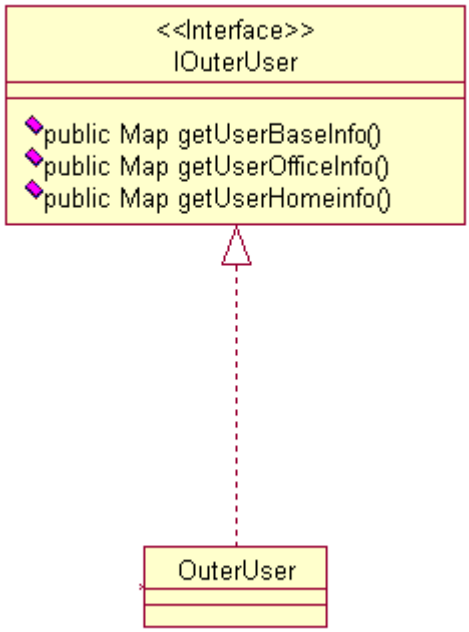


还是比较简单的，有一个对象 UserInfo 存储用户的所有信息（实际系统上还有很多子类，不多说了），也就是 BO（Business Object），这个对象设计为贫血对象（Thin Business Object），不需要存储状态以及相关的关系，而且我也是反对使用充血对象（Rich Business Object），这里说了两个名词贫血对象和充血对象，这两个名词很简单，在领域模型中分别叫做贫血领域模型和充血领域模型，有什么区别呢？在一个对象中不存储实体状态以及对象之间的关系的就叫做贫血对象，上升到领域模型中就是贫血领域模型，有实体状态和对象关系的模型的就是充血领域模型，是不是太技术化了？这个看不懂没关系，都是糊弄人的东西，属于专用名词，这本书写完了，我再折腾本领域模型的文章，揭露领域模型中糊弄人的专有名词，这个绝对是专有名词的堆砌，呵呵。扯远了，我们继续说适配器模式，这个 UserInfo 对象，在系统中很多地方使用，你可以查看自己的信息，也可以做修改，当然这个对象是有 setter 方法的，我们这里用不到就隐藏掉了。

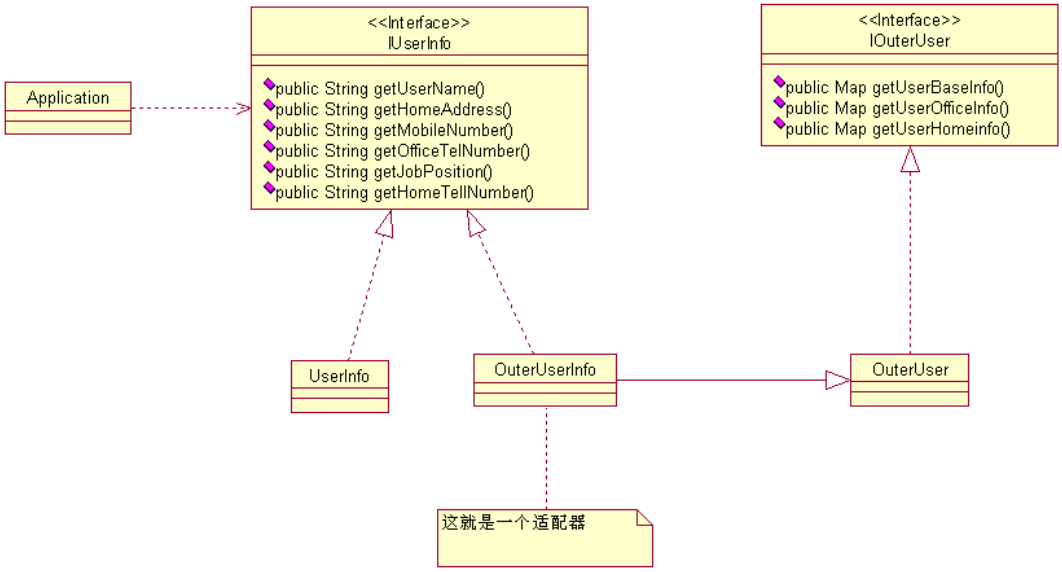
这个项目是 04 年年底投产的，运行到 05 年年底还是比较平稳的，中间修修补补也很正常，05 年年底不知道是那股风吹的，很多公司开始使用借聘人员的方式招聘人员，我们公司也不例外，从一个人力资源公司借用了一大批的低技术、低工资的人员，分配到各个子公司，总共有将近 200 号人，然后就找我们部门老大谈判，说要增加一个功能借用人员管理，老大一看有钱赚呀，一拍大腿，做！

我带人过去一调研，不是这么简单，人力资源公司有一套自己的人员管理系统，我们公

司需要把我们使用到的人员信息传输到我们的系统中，系统之间的传输使用 RMI（Remote Method Invocation，远程对象调用）的方式，但是有一个问题人力资源公司的人员对象和我们系统的对象不相同呀，他们的对象是这样的：



人员资源公司是把人的信息分为了三部分：基本信息，办公信息和个人家庭信息，并且都放到了 HashMap 中，比如人员的姓名放到 BaseInfo 信息中，家庭地址放到 HomeInfo 中，这咱不好说他们系统设计的不好，那问题是咱的系统要和他们系统有交互，怎么办？使用适配器模式，类图如下：



大家可能会问，这两个对象都不在一个系统中，你如何使用呢？简单！RMI 已经帮我们做了这件事情，只要有接口，就可以把远程的对象当成本地的对象使用，这个大家有时间可以去看一下 RMI 文档，不多说了。通过适配器，把 OuterUser 伪装成我们系统中一个 IUserInfo 对象，这样，我们的系统基本不用修改什么程序员，所有的人员查询、调用跟本地一样样的，说的口干舌燥，那下边我们来看具体的代码实现：

首先看 IUserInfo.java 的代码：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 用户信息对象
 */
public interface IUserInfo {

    //获得用户姓名
    public String getUserName();

    //获得家庭地址
    public String getHomeAddress();

    //手机号码，这个太重要，手机泛滥呀
    public String getMobileNumber();

    //办公电话，一般式座机
    public String getOfficeTelNumber();

    //这个人的职位是啥
    public String getJobPosition();

    //获得家庭电话，这个有点缺德，我是不喜欢打家庭电话讨论工作
    public String getHomeTelNumber();
}
```

然后看这个接口的实现类：

```
package com.cbf4life;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class UserInfo implements IUserInfo {

    /*
     * 获得家庭地址，下属送礼也可以找到地方
     */
    public String getHomeAddress() {
        System.out.println("这里是员工的家庭地址....");
        return null;
    }

    /*
     * 获得家庭电话号码
     */
    public String getHomeTelNumber() {
        System.out.println("员工的家庭电话是....");
        return null;
    }

    /*
     * 员工的职位，是部门经理还是小兵
     */
    public String getJobPosition() {
        System.out.println("这个人的职位是BOSS....");
        return null;
    }

    /*
     * 手机号码
     */
    public String getMobileNumber() {
        System.out.println("这个人的手机号码是0000....");
        return null;
    }

    /*
     * 办公室电话，烦躁的时候最好“不小心”把电话线踢掉，我经常这么干，对己对人都有好处
     */
    public String getOfficeTelNumber() {
        System.out.println("办公室电话是....");
```

```
        return null;
    }

    /*
     * 姓名了，这个老重要了
     */
    public String getUserName() {
        System.out.println("姓名叫做...");
        return null;
    }

}
```

可能有人要问了，为什么要把电话号码、手机号码都设置成 String 类型，而不是 int 类型，大家觉的呢？题外话，这个绝对应该是 String 类型，包括数据库也应该是 varchar 类型的，手机号码有小灵通带区号的，比如 02100001，这个你用数字怎么表示？有些人要在手机号码前加上 0086 后再保存，比如我们公司的印度阿三就是这样，喜欢在手机号码前 0086 保存下来，呵呵，我是想到啥就说啥，啰嗦了点。继续看我们的代码，下面看我们系统的应用如何调用 UserInfo 的信息：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这就是我们具体的应用了，比如老板要查所有的20-30的女性信息
 */
public class App {


    public static void main(String[] args) {
        //没有与外系统连接的时候，是这样写的
        IUserInfo youngGirl = new UserInfo();
        //从数据库中查到101个
        for(int i=0;i<101;i++){
            youngGirl.getMobileNumber();
        }


    }
```

```
}
```

这老板，比较那个，为什么是 101，是男生都应该知道吧，111 代表男生，101 代表女生，呵呵，是不是比较色呀。从数据库中生成了 101 个 UserInfo 对象，直接打印出来就成了。那然后增加了外系统的人员信息，怎么处理呢？下面是 IOuterUser.java 的源代码：

```
package com.cbf4life;

import java.util.Map;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 外系统的人员信息
 */
@SuppressWarnings("all")
public interface IOuterUser {

    //基本信息，比如名称，性别，手机号码了等
    public Map getUserBaseInfo();

    //工作区域信息
    public Map getUserOfficeInfo();

    //用户的家庭信息
    public Map getUserHomeInfo();

}
```

我们再看看外系统的用户信息的具体实现类：

```
package com.cbf4life;

import java.util.HashMap;
import java.util.Map;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 外系统的用户信息的实现类
 */
@SuppressWarnings("all")
```

```
public class OuterUser implements IOuterUser {

    /*
     * 用户的基本信息
     */
    public Map getUserBaseInfo() {
        HashMap baseInfoMap = new HashMap();

        baseInfoMap.put("userName", "这个员工叫混世魔王....");
        baseInfoMap.put("mobileNumber", "这个员工电话是....");

        return baseInfoMap;
    }

    /*
     * 员工的家庭信息
     */
    public Map getUserHomeInfo() {
        HashMap homeInfo = new HashMap();

        homeInfo.put("homeTelNumbner", "员工的家庭电话是....");
        homeInfo.put("homeAddress", "员工的家庭地址是....");

        return homeInfo;
    }

    /*
     * 员工的工作信息，比如职位了等
     */
    public Map getUserOfficeInfo() {
        HashMap officeInfo = new HashMap();

        officeInfo.put("jobPosition","这个人的职位是BOSS...");
        officeInfo.put("officeTelNumber", "员工的办公电话是....");

        return officeInfo;
    }

}
```

那怎么把外系统的用户信息包装成我们公司的人员信息呢？看下面的 OuterUserInfo 类源码，也就是我们的适配器：

```
package com.cbf4life;

import java.util.Map;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 把OuterUser包装成UserInfo
 */
@SuppressWarnings("all")
public class OuterUserInfo extends OuterUser implements IUserInfo {

    private Map baseInfo = super.getUserBaseInfo();  //员工的基本信息
    private Map homeInfo = super.getUserHomeInfo(); //员工的家庭 信息
    private Map officeInfo = super.getUserOfficeInfo(); //工作信息

    /*
     * 家庭地址
     */
    public String getHomeAddress() {
        String homeAddress =
(String)this.homeInfo.get("homeAddress");
        System.out.println(homeAddress);
        return homeAddress;
    }

    /*
     * 家庭电话号码
     */
    public String getHomeTelNumber() {
        String homeTelNumber =
(String)this.homeInfo.get("homeTelNumber");
        System.out.println(homeTelNumber);
        return homeTelNumber;
    }

    /*
     *职位信息
     */
    public String getJobPosition() {
        String jobPosition =
(String)this.officeInfo.get("jobPosition");
        System.out.println(jobPosition);
        return jobPosition;
```

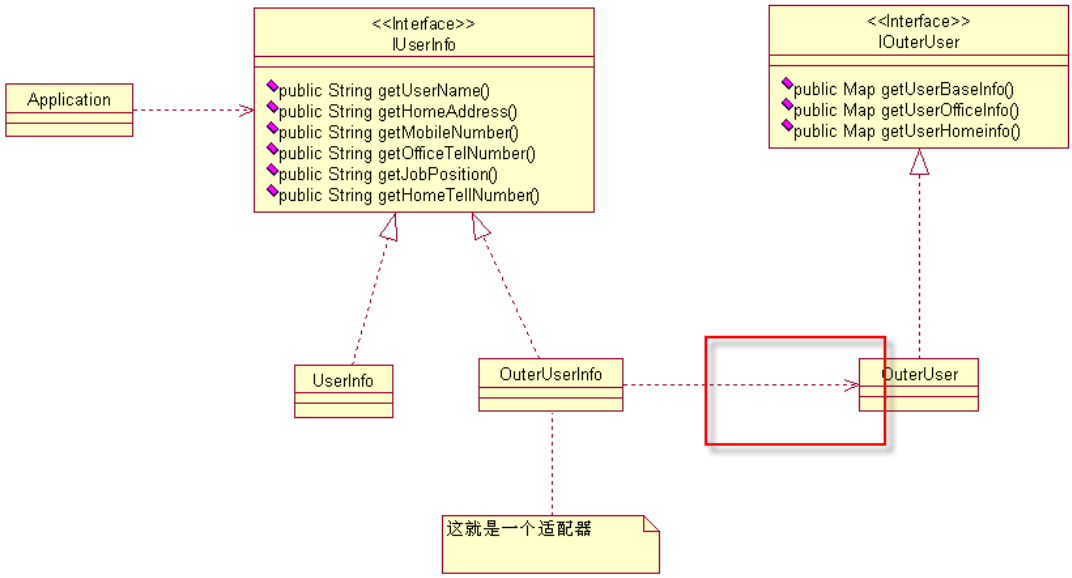
```
    }

    /*
     * 手机号码
     */
    public String getMobileNumber() {
        String mobileNumber =
(String)this.baseInfo.get("mobileNumber");
        System.out.println(mobileNumber);
        return mobileNumber;
    }

    /*
     * 办公电话
     */
    public String getOfficeTelNumber() {
        String officeTelNumber =
(String)this.officeInfo.get("officeTelNumber");
        System.out.println(officeTelNumber);
        return officeTelNumber;
    }

    /*
     * 员工的名称
     */
    public String getUserName() {
        String userName = (String)this.baseInfo.get("userName");
        System.out.println(userName);
        return userName;
    }

}
```

大家看到没？这里有很多的强制类型转换，就是(String)这个东西，如果使用泛型的话，完全就可以避免这个转化，这节课啰嗦的太多就不再讲了，下次找个时间再讲。这个适配器的作用就是做接口的转换，那然后我们再来看看我们的业务是怎么调用的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这就是我们具体的应用了，比如老板要查所有的20-30的女性信息
 */
public class App {


    public static void main(String[] args) {
        //没有与外系统连接的时候，是这样写的
        //IUserInfo youngGirl = new UserInfo();

        //老板一想不对呀，兔子不吃窝边草，还是找人力资源的员工好点
        IUserInfo youngGirl = new OuterUserInfo();  //我们只修改了这一
句好
        //从数据库中查到101个
        for(int i=0;i<101;i++){
            youngGirl.getMobileNumber();
        }

    }

}
```

大家看，使用了适配器模式只修改了一句话，其他的业务逻辑都不用修改就解决了系统对接的问题，而且在我们实际系统中只是增加了一个业务类的继承，就实现了可以查本公司的员工信息，也可以查人力资源公司的员工信息，尽量少的修改，通过扩展的方式解决了该问题。

适配器模式分为类适配器和对象适配器，这个区别不大，上边的例子就是类适配器，那对象适配器是什么样子呢？对象适配器的类图是这个样子滴：



看到没？和上边的类图就一个箭头的图形的差异，一个是继承，一个是关联，就这么多区别，只要把我们上面的程序稍微修改一下就成了类适配器，这个大家自己考虑一下，简单的很。

适配器模式不适合在系统设计阶段采用，没有一个系统分析师会在做详设的时候考虑使用适配器模式，这个模式使用的主要场景是扩展应用中，就像我们上面的那个例子一样，系统扩展了，不符合原有设计的时候才考虑通过适配器模式减少代码修改带来的风险。

在论坛上有网友建议我加上模式的优缺点，这个我是不建议加上去，你先掌握了怎么使用，然后再想怎么更好的使用，而且我想你既然想到了你要使用这 23 个模式，肯定是有权衡的吧，那这个模式的优缺点是不是由你自己来总结会更好的呢？

9. 模板方法模式【Template Method Pattern】

周三，9：00，我刚刚坐到位置，打开电脑准备开始干活。

“小三，小三，叫一下其它同事，到会议室，开会”老大跑过来吼，带着淫笑。还不等大家坐稳，老大就开讲了，

“告诉大家一个好消息，昨天终于把牛叉模型公司的口子打开了，要我们做悍马模型，虽然是第一个车辆模型，但是我们有能力，有信心做好，我们一定要…（中间省略 20 分钟的讲话，如果你听过领导人的讲话，这个你应该能够续上）”

动员工作做完了，那就开始压任务了，“这次时间是非常紧张的，只有一个星期的时间，小三，你负责在一个星期的时间把这批 10 万车模(注：车模是车辆模型的意思，不是香车美女那个车模)建设完成…”

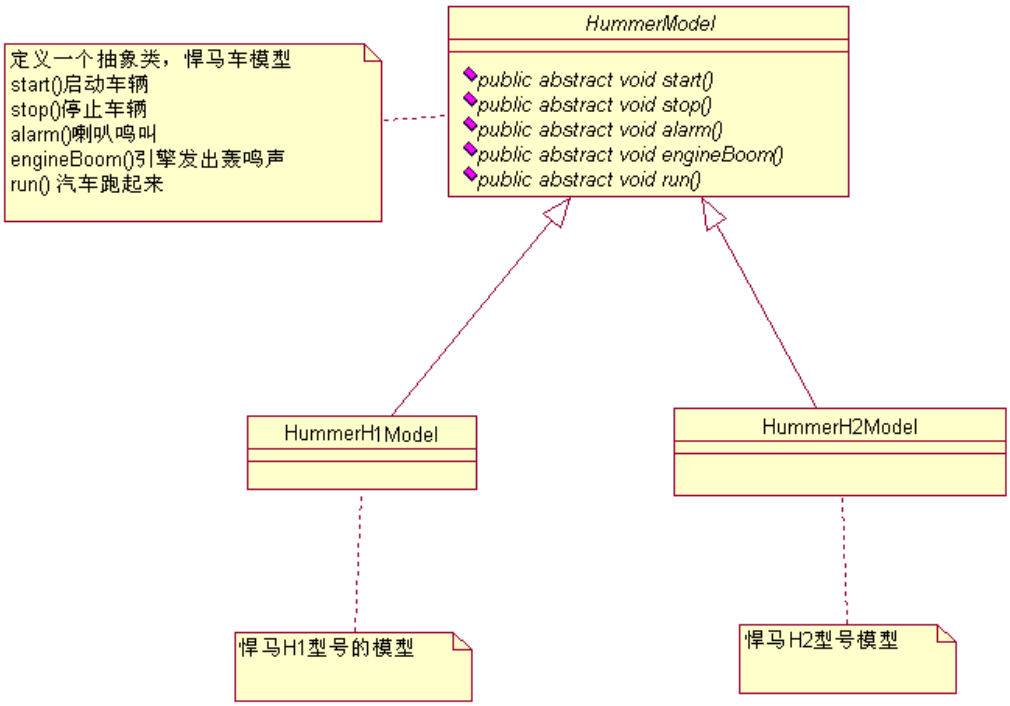
“一个星期？这个…，是真做不完，要做分析，做模板，做测试，还要考虑扩展性、稳定性、健壮性等，时间实在是太少了”还没等老大说完，我就急了，再不急我的小命就折在上面了！

“那这样，你只做实现，不考虑使用设计模式，扩展性等都不用考虑”老大又把我压回去了。

“不考虑设计模式？那…”

哎，领导已经布置任务了，那就开始死命的做吧，命苦不能怨政府，点背不能怪社会呀，然后就开始准备动手做，在做之前先介绍一下我们公司的背景，我们公司是做模型生产的，做过桥梁模型、建筑模型、机械模型，甚至是一些政府、军事的机密模型，这个不能说，就是把真实的实物按照一定的比例缩小或放大，用于试验、分析、量化或者是销售等等，上面提到的牛叉模型公司专门销售车辆模型的公司，自己不生产，我们公司是第一次从牛叉模型公司接单，那我怎么着也要把活干好，可时间很紧张呀，怎么办？

既然领导都说了，不考虑扩展性，那好办，我先设计个类图：



非常简单的实现，你要悍马模型，我就给你悍马模型，先写个抽象类，然后两个不同型号的模型实现类，那我们把这个程序实现出来：

HummerModel 抽象类的程序清单如下：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * Hummer Model是悍马车辆模型的意思，不是悍马美女车模
 */
public abstract class HummerModel {

    /*
     * 首先，这个模型要能够被发动起来，别管是手摇发动，还是电力发动，反正
     * 是要能够发动起来，那这个实现要在实现类里了
     */
    public abstract void start();

    //能发动，那还要能停下来，那才是真本事
    public abstract void stop();
```

```
    //喇叭会出声音，是滴滴叫，还是哔哔叫
    public abstract void alarm();

    //引擎会轰隆隆的响，不响那是假的
    public abstract void engineBoom();

    //那模型应该会跑吧，别管是人推的，还是电力驱动，总之要会跑
    public abstract void run();
}
```

H1 型号悍马的定义如下：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 悍马车是每个越野者的最爱，其中H1最接近军用系列
 */
public class HummerH1Model extends HummerModel {

    @Override
    public void alarm() {
        System.out.println("悍马H1鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H1引擎声音是这样在...");
    }

    @Override
    public void start() {
        System.out.println("悍马H1发动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H1停车...");
    }

    /*
```

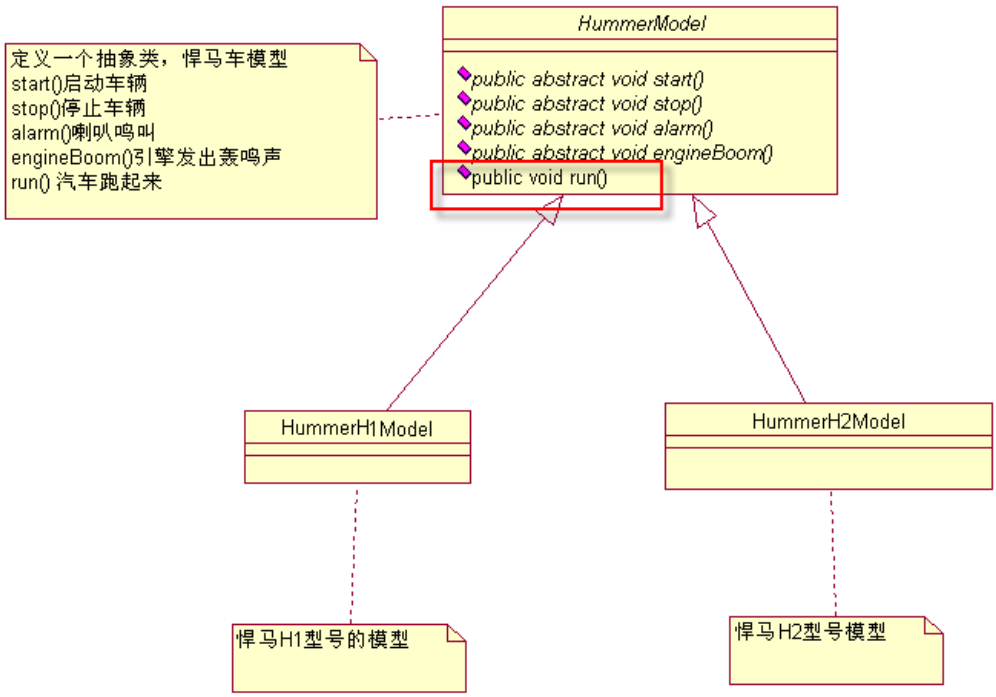
```
     * 这个方法是很有意思的，它要跑，那肯定要启动，停止了等，也就是要调其他方法
     */
    @Override
    public void run() {

        //先发动汽车
        this.start();

        //引擎开始轰鸣
        this.engineBoom();

        //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭
        this.alarm();

        //到达目的地就停车
        this.stop();
    }

}
```

然后看悍马 H2 型号的实现：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * H1和H2有什么差别，还真不知道，真没接触过悍马
 */
public class HummerH2Model extends HummerModel {

    @Override
    public void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H2引擎声音是这样在...");
    }

    @Override
    public void start() {
```

```
        System.out.println("悍马H2发动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H1停车...");
    }

    /*
     * H2要跑，那肯定要启动，停止了等，也就是要调其他方法
     */
    @Override
    public void run() {

        //先发动汽车
        this.start();

        //引擎开始轰鸣
        this.engineBoom();

        //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭
        this.alarm();

        //到达目的地就停车
        this.stop();
    }

}
```

然后程序写到这里，你就看到问题了，run 方法的实现应该在抽象类上，不应该在实现类上，好，我们修改一下类图和实现：



就把 run 方法放到了抽象类中，那代码也相应的改变一下，先看 HummerModel.java:

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * Hummer Model是悍马车辆模型的意思，不是悍马美女车模
 */
public abstract class HummerModel {

    /*
     * 首先，这个模型要能够被发动起来，别管是手摇发动，还是电力发动，反正
     * 是要能够发动起来，那这个实现要在实现类里了
     */
    public abstract void start();

    //能发动，那还要能停下来，那才是真本事
    public abstract void stop();

    //喇叭会出声音，是滴滴叫，还是哔哔叫
    public abstract void alarm();

    //引擎会轰隆隆的响，不响那是假的
```

```
    public abstract void engineBoom();

    //那模型应该会跑吧，别管是人退的，还是电力驱动，总之要会跑
    public void run() {

        //先发动汽车
        this.start();

        //引擎开始轰鸣
        this.engineBoom();

        //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭
        this.alarm();

        //到达目的地就停车
        this.stop();
    }
}
```

下面是 HummerH1Model.java 程序清单：

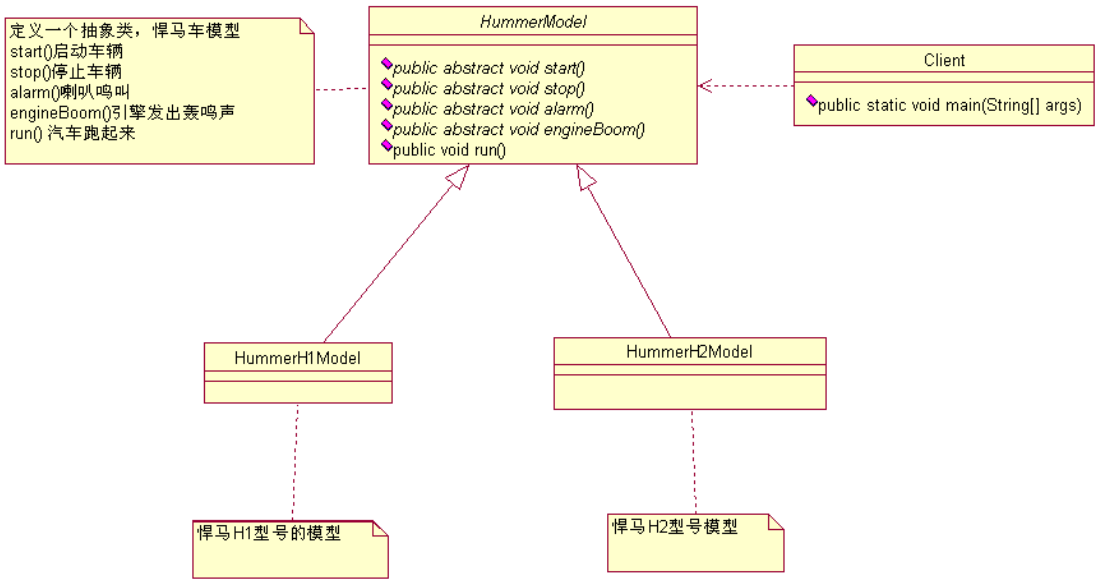
```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 悍马车是每个越野者的最爱，其中H1最接近军用系列
 */
public class HummerH1Model extends HummerModel {

    @Override
    public void alarm() {
        System.out.println("悍马H1鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H1引擎声音是这样在...");
    }

    @Override
    public void start() {
        System.out.println("悍马H1发动...");
```

```
    }

    @Override
    public void stop() {
        System.out.println("悍马H1停车...");
    }

}
```

下面是 HummerH2Model.java 的程序清单：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * H1和H2有什么差别，还真不知道，真没接触过悍马
 */
public class HummerH2Model extends HummerModel {

    @Override
    public void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H2引擎声音是这样在...");
    }

    @Override
    public void start() {
        System.out.println("悍马H2发动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H2停车...");
    }

}
```

类图修改完毕了，程序也该好了，提交给老大，老大一看，挺好，就开始生产了，并提交给客户使用了，那客户是如何使用的呢？类图上增加一个 Client 类，就是客户，我们这个是用 main 函数来代替他使用，类图如下：



然后看增加的 Client.java 程序，非常的简单：

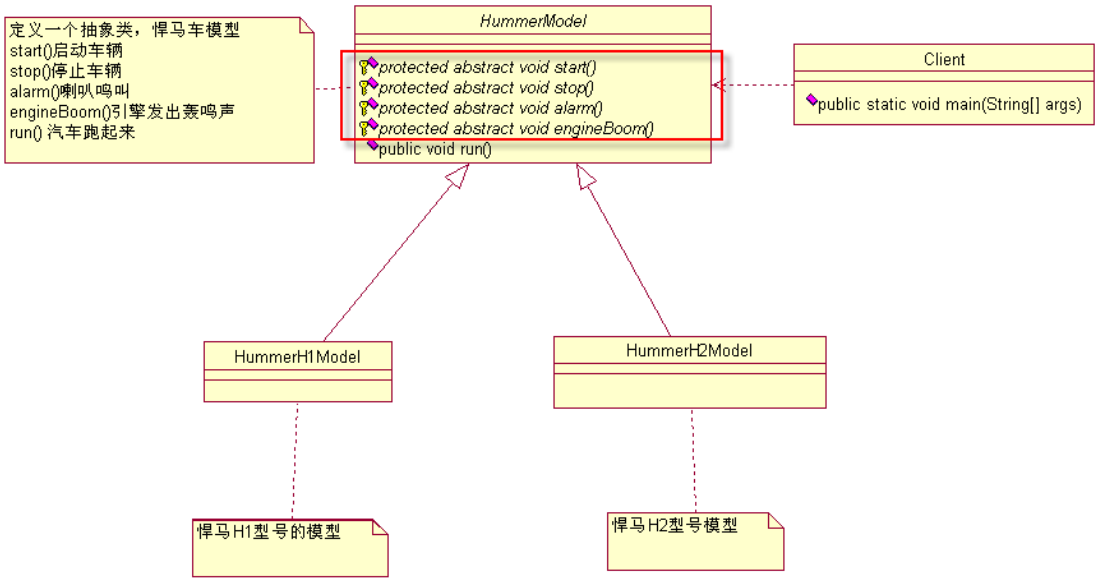
```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        //客户开着H1型号，出去遛弯了
        HummerModel h1 = new HummerH1Model();
        h1.run(); //汽车跑起来了；

        //客户开H2型号，出去玩耍了
        HummerModel h2 = new HummerH2Model();
        h2.run();
    }

}
```

运行的结果如下：

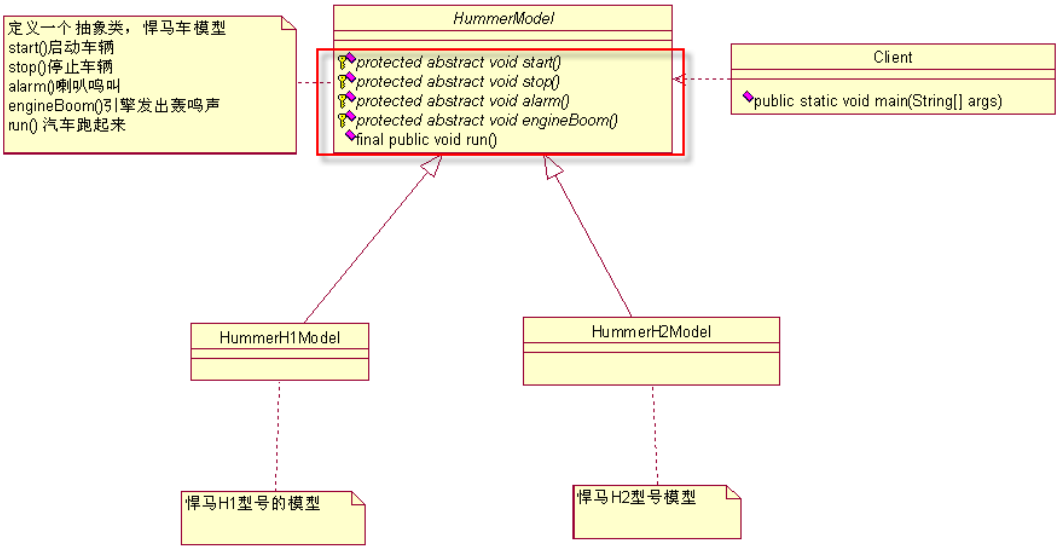
```
悍马H1发动...
悍马H1引擎声音是这样在...
悍马H1鸣笛...
悍马H1停车...
悍马H2发动...
悍马H2引擎声音是这样在...
悍马H2鸣笛...
悍马H2停车...
```

非常非常的简单，那如果我告诉这就是模板方法模式你会不会很不屑呢？就这模式，太简单了，我一直在使用呀，是的，你经常在使用，但你不知道这是模板方法模式，那些所谓的高手就可以很牛 X 的说“用模板方法模式就可以实现…”，你还要很崇拜的看着，哇，牛人，模板方法模式是什么呀?

然后我们继续回顾我们这个模型，回头一想，不对呀，需求分析的有点问题，客户要关心模型的启动，停止，鸣笛，引擎声音吗？他只要在 run 的过程中，听到或看都成了呀，暴露那么多的方法干啥？好了，我们重新修改一下类图：



把抽象类上的四个方法设置为 protected 访问权限，好了，既然客户不关心这几个方法，而且这四个方法都是由子类来实现的，那就设置成 protected 模式。咦~，那还有个缺陷，run 方法既然子类都不修改，那是不是可以设置成 final 类型呢？是滴是滴，类图如下：



好了，这才是模板方法模式，就是这个样子，我们只要修改抽象类代码就可以了，HummerModel.java 程序清单如下：

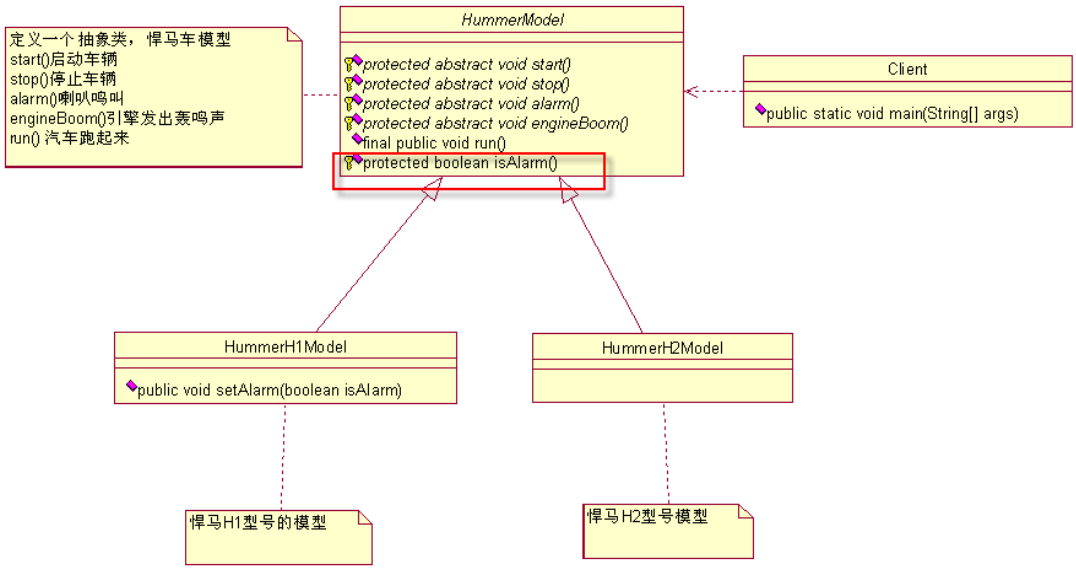
```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * Hummer Model是悍马车辆模型的意思，不是悍马美女车模
 */
public abstract class HummerModel {

    /*
     * 首先，这个模型要能够被发动起来，别管是手摇发动，还是电力发动，反正
     * 是要能够发动起来，那这个实现要在实现类里了
     */
    protected abstract void start();

    //能发动，那还要能停下来，那才是真本事
    protected abstract void stop();

    //喇叭会出声音，是滴滴叫，还是哔哔叫
    protected abstract void alarm();

    //引擎会轰隆隆的响，不响那是假的
    protected abstract void engineBoom();

    //那模型应该会跑吧，别管是人退的，还是电力驱动，总之要会跑
```

```
    final public void run() {

        //先发动汽车
        this.start();

        //引擎开始轰鸣
        this.engineBoom();

        //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭
        this.alarm();

        //到达目的地就停车
        this.stop();
    }
}
```

其他的子类都不用修改(如果要修改，就是把四个方法的访问权限由 public 修改 protected)，大家请看这个 run 方法，他定义了调用其他方法的顺序，并且子类是不能修改的，这个叫做模板方法；start、stop、alarm、engineBoom 这四个方法是子类必须实现的，而且这四个方法的修改对应了不同的类，这个叫做基本方法，基本方法又分为三种：在抽象类中实现了的基本方法叫做具体方法;在抽象类中没有实现,在子类中实现了叫做抽象方法，我们这四个基本方法都是抽象方法，由子类来实现的；还有一种叫做钩子方法，这个等会讲。

到目前为止，这两个模型都稳定的运行，突然有一天，老大又找到了我，

“客户提出新要求了，那个喇叭想让它响就响，你看你设计的模型，车子一启动，喇叭就狂响，赶快修改一下”，确实是设计缺陷，呵呵，不过是我故意的，那我们怎么修改呢？看修改后的类图：



增加一个方法，isAlarm()，喇嘛要不要响，这就是钩子方法(Hook Method)，那我们只要修改一下抽象类就可以了：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * Hummer Model是悍马车辆模型的意思，不是悍马美女车模
 */
public abstract class HummerModel {

    /*
     * 首先，这个模型要能够被发动起来，别管是手摇发动，还是电力发动，反正
     * 是要能够发动起来，那这个实现要在实现类里了
     */
    protected abstract void start();

    //能发动，那还要能停下来，那才是真本事
    protected abstract void stop();

    //喇叭会出声音，是滴滴叫，还是哔哔叫
    protected abstract void alarm();

    //引擎会轰隆隆的响，不响那是假的
    protected abstract void engineBoom();

    //那模型应该会跑吧，别管是人推的，还是电力驱动，总之要会跑
```

```
    final public void run() {

        //先发动汽车
        this.start();

        //引擎开始轰鸣
        this.engineBoom();

        //喇嘛想让它响就响，不想让它响就不响
        if(this.isAlarm()){
            this.alarm();
        }

        //到达目的地就停车
        this.stop();
    }

    //钩子方法，默认喇叭是会响的
    protected  boolean isAlarm(){
        return true;
    }
}
```

钩子方法模式是由抽象类来实现的，子类可以重写的，H2 型号的悍马是不会叫的，喇叭是个摆设，看 HummerH2Model.java 代码：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * H1和H2有什么差别，还真不知道，真没接触过悍马
 */
public class HummerH2Model extends HummerModel {

    @Override
    protected void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

    @Override
    protected void engineBoom() {
```

```
        System.out.println("悍马H2引擎声音是这样在...");
    }

    @Override
    protected void start() {
        System.out.println("悍马H2发动...");
    }

    @Override
    protected void stop() {
        System.out.println("悍马H1停车...");
    }


    //默认没有喇叭的
    @Override
    protected boolean isAlarm() {
        return false;
    }

}
```

那 H2 型号的模型都没有喇叭，就是按了喇叭也没有声音，那客户端这边的调用没有任何修改，出来的结果就不同，我们先看 Client.java 程序：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        HummerH2Model h2 = new HummerH2Model();
        h2.run();  //H2型号的悍马跑起来
    }

}
```

运行的出来的结果是这样的：

```
悍马H2发动...
悍马H2引擎声音是这样在...
悍马H1停车...
```

那 H1 又有所不同了，它的喇叭要不要响是由客户来决定，其实在类图上已经标明了 setAlarm 这个方法，我们看 HummerH1Model.java 的代码：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 悍马车是每个越野者的最爱，其中H1最接近军用系列
 */
public class HummerH1Model extends HummerModel {
    private boolean alarmFlag = true;  //是否要响喇叭

    @Override
    protected void alarm() {
        System.out.println("悍马H1鸣笛...");
    }

    @Override
    protected void engineBoom() {
        System.out.println("悍马H1引擎声音是这样在...");
    }

    @Override
    protected void start() {
        System.out.println("悍马H1发动...");
    }

    @Override
    protected void stop() {
        System.out.println("悍马H1停车...");
    }
```

```
    @Override
    protected boolean isAlarm() {
        return this.alarmFlag;
    }

    //要不要响喇叭，是有客户的来决定的
    public void setAlarm(boolean isAlarm){
        this.alarmFlag = isAlarm;
    }

}
```

这段代码呢修改了两个地方，一是重写了父类的 isAlarm()方法，一是增加了一个 setAlarm 方法，由调用者去决定是否要这个功能，也就是喇叭要不要滴滴答答的响，哈哈，那我们看看 Client.java 的修改：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        //客户开着H1型号，出去遛弯了
        HummerH1Model h1 = new HummerH1Model();
        h1.setAlarm(true);
        h1.run(); //汽车跑起来了；

    }

}
```

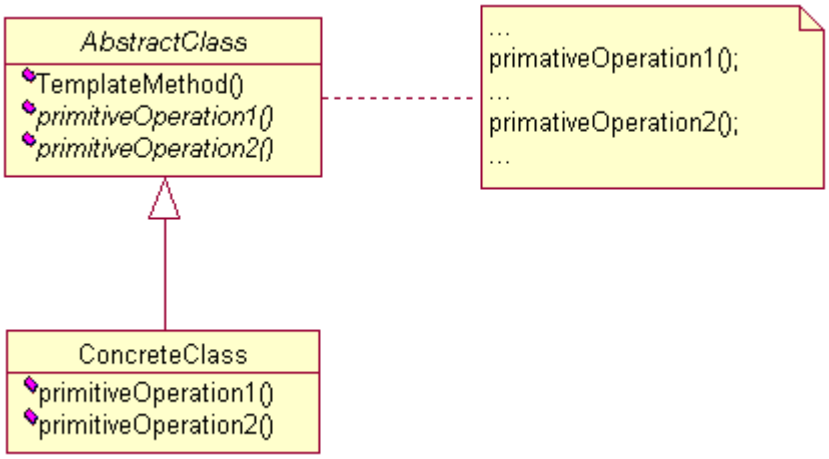
运行的结果如下：

```
悍马H1发动...
悍马H1引擎声音是这样在...
悍马H1鸣笛...
```

悍马 H1 停车...

看到没，这个模型 run 起来就有声音了，那当然把 h1.setAlarm(false)运行起来喇叭就没有声音了，钩子方法的作用就是这样滴。

那我们总结一下模板方法模式，模板方法模式就是在模板方法中按照一个的规则和顺序调用基本方法，具体到我们上面那个例子就是 run 方法按照规定的顺序(先调用 start,然后再调用 engineBoom，再调用 alarm，最后调用 stop)调用本类的其他方法，并且由 isAlarm 方法的返回值确定 run 中的执行顺序变更，通用类图如下：



其中 TemplateMethod 就是模板方法，operation1 和 operation2 就是基本方法，模板方法是通过汇总或排序基本方法而产生的结果集。模板方法在一些开源框架中应用很多，它提供了一个抽象类，然后开源框架写了一堆子类，在《XXX In Action》中就说明了，如果你需要扩展功能，可以继承了这个抽象类，然后修改 protected 方法，再然后就是调用一个类似 execute 方法，就完成你的扩展开发，确实是一种简单的模式。

初级程序员在写程序的时候经常会问高手“父类怎么调用子类的方法”，这个问题很有普遍性，反正我是被问过好几回，那么父类是否可以调用子类的方法呢？我的回答是能，但强烈的、极度的不建议，怎么做呢?

- *把子类传递到父类的有参构造中，然后调用;*
- *使用反射的方式调用，你使用了反射还有谁不能调用的？！*
- *父类调用子类的静态方法。*

这三种都是父类直接调用子类的方法，好用不？好用！解决问题了吗？解决了！项目中

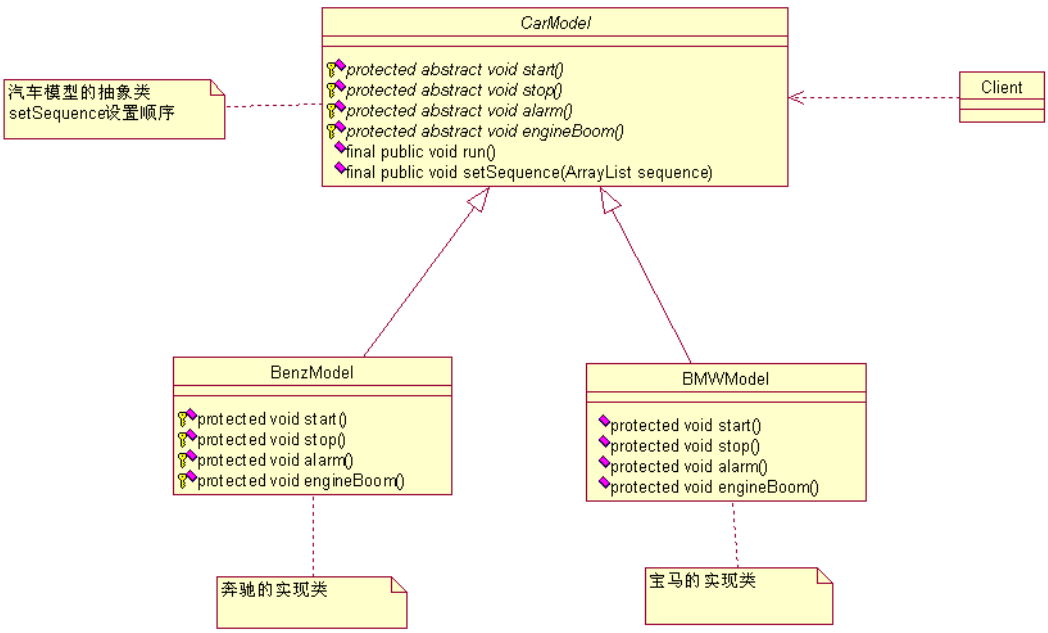
允许使用不？不允许！我就一直没有搞懂为什么要父类调用子类的方法，如果一定要调用子类，那为什么要继承它呢？搞不懂。其实这个问题可以换个角度去理解，在重写了父类部分方法后，再调用从父类继承的方法，产生不同的结果（而这正是模板方法模式），这是不是也可以理解为父类调用了子类的方法呢？你修改了子类，影响了父类的结果，模板方法模式就是这样效果。

10.建造者模式【Builder Pattern】

又是一个周三，快要下班了，老大突然又拉住我，喜滋滋的告诉我“牛叉公司很满意我们做的模型，又签订了一个合同，把奔驰、宝马的车辆模型都交给我我们公司制作了，不过这次又额外增加了一个新需求：汽车的启动、停止、喇叭声音、引擎声音都有客户自己控制，他想什么顺序就什么顺序，这个没问题吧?”。

看着老大殷切的目光，我还能说啥，肯定的点头，“没问题!”，加班加点做呗，“再苦再累就当自己二百五 再难再险就当自己二皮脸 与君共勉!”这句话说出了我的心声。那任务是接下来，我们怎么做实现呢？

首先我们想了，奔驰、宝马都是一个产品，他们有共有的属性，牛叉公司关心的是单个模型，奔驰模型 A 是先有引擎声音，然后再启动；奔驰模型 B 呢是先启动起来，然后再有引擎声音，这才是牛叉公司要关关心的，那到我们老大这边呢，就是满足人家的要求，要什么顺序就立马能产生什么顺序的模型出来，我呢就负责把老大的要求实现掉，而且还要是批量的，看不懂？没关系，继续看下去，首先由我生产出 N 多个奔驰和宝马车辆模型，这些车辆模型的都有 run 方法,但是具体到每一个模型的 run 方法可能中间的执行任务的顺序是不同的，老大说要啥顺序，我就给啥顺序，最终客户买走后只能是既定的模型，还是没听明白，我们继续，我们先把我们最基本的对象 Product 在类图中表明出来：



我们定义了一个 CarModel 的抽象类，其中 run 和 setSequence 是由抽象类实现的，其他都是子类自己实现，那这个是否可以解决这个问题呢?应该可以，我们把代码实现出来，先看 CarModel.java 程序：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个车辆模型的抽象类，所有的车辆模型都继承这里类
 */
public abstract class CarModel {

    //这个参数是各个基本方法执行的顺序
    private ArrayList<String> sequence = new ArrayList<String>();

    /*
     * 模型是启动开始跑了
     */
    protected abstract void start();

    //能发动，那还要能停下来，那才是真本事
    protected abstract void stop();

    //喇叭会出声音，是滴滴叫，还是哔哔叫
    protected abstract void alarm();

    //引擎会轰隆隆的响，不响那是假的
    protected abstract void engineBoom();

    //那模型应该会跑吧，别管是人推的，还是电力驱动，总之要会跑
    final public void run() {

        //循环一遍，谁在前，就先执行谁
        for(int i=0;i<this.sequence.size();i++){
            String actionName = this.sequence.get(i);

            if(actionName.equalsIgnoreCase("start")){  //如果是start
关键字，
                this.start();  //开启汽车
```

```
            }else if(actionName.equalsIgnoreCase("stop")){ //如果是
stop关键字
                this.stop(); //停止汽车
            }else if(actionName.equalsIgnoreCase("alarm")){ //如果是
alarm关键字
                this.alarm(); //喇叭开始叫了
            }else if(actionName.equalsIgnoreCase("engine
boom")){  //如果是engine boom关键字
                this.engineBoom();  //引擎开始轰鸣
            }

        }

    }

    //把传递过来的值传递到类内
    final public void setSequence(ArrayList<String> sequence){
        this.sequence = sequence;
    }
}
```

其中 setSequence 方法是允许客户自己设置一个顺序，是要先跑起来在有引擎声音还是先有引擎声音再跑起来，还是说那个喇叭就不要响，对于一个具体的模型永远都固定的，那这个事由牛叉告诉我们，有 1W 件事这样的，1W 件事那样的顺序，目前的设计都能满足这个要求。

run 方法使用了一个数组的遍历，确定那个是先执行，程序比较简单，不多说了，我们继续看两个实现类，先看 BenzModel.java 程序：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 奔驰车模型
 */
public class BenzModel extends CarModel {


    @Override
    protected void alarm() {
        System.out.println("奔驰车的喇叭声音是这个样子的...");
    }
```

```
    @Override
    protected void engineBoom() {
        System.out.println("奔驰车的引擎是这个声音的...");
    }


    @Override
    protected void start() {
        System.out.println("奔驰车跑起来是这个样子的...");
    }


    @Override
    protected void stop() {
        System.out.println("奔驰车应该这样停车...");
    }

}
```

没啥特别的，不多说，再看 BMWModel.java 程序：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 宝马车模型
 */
public class BMWModel extends CarModel {


    @Override
    protected void alarm() {
        System.out.println("宝马车的喇叭声音是这个样子的...");
    }


    @Override
    protected void engineBoom() {
        System.out.println("宝马车的引擎是这个声音的...");
    }
```

```
    @Override
    protected void start() {
        System.out.println("宝马车跑起来是这个样子的...");
    }


    @Override
    protected void stop() {
        System.out.println("宝马车应该这样停车...");
    }

}
```

两个实现类都完成，我们再来看牛叉公司要的要求，我先要 1 个奔驰的模型，这个模型的要求是跑的时候，先发动引擎，然后再挂档启动，然后停下来，不需要喇叭，那怎么实现呢：

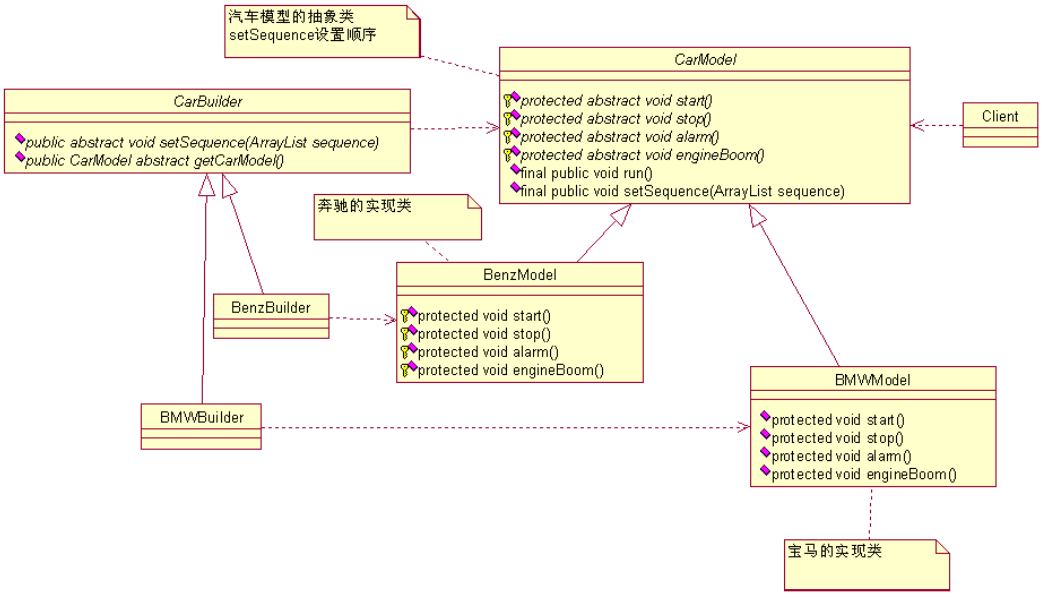
```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 最终客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        /*
         * 客户告诉牛叉公司，我要这样一个模型，然后牛叉公司就告诉我老大
         * 说要这样一个模型，这样一个顺序，然后我就来制造
         */
        BenzModel benz = new BenzModel();
        ArrayList<String> sequence  = new ArrayList<String>();  //存放run的顺序

        sequence.add("engine boom");  //客户要求，run的时候时候先发动引擎
        sequence.add("start");  //启动起来
        sequence.add("stop");  //开了一段就停下来
```

```
        //然后我们把这个顺序给奔驰车：
        benz.setSequence(sequence);
        benz.run();

    }

}
```

运行结果是这样的：

```
奔驰车的引擎是这个声音的...
奔驰车跑起来是这个样子的...
奔驰车应该这样停车...
```

看，满足了牛叉公司的需求了，满足完了，还要下一个需求呀，然后是第 2 件宝马模型，只要启动，停止，其他的什么都不要，第 3 件模型，先喇叭，然后启动，然后停止，第 4 件...，直到把你逼疯为止，那怎么办？我们修改程序，满足这种变态需求，好，看我如何修改，先修改类图：



增加了一个 CarBuilder 的抽象类，以及两个实现类，其目的是你要什么排列顺序的车，我就给你什么顺序的车，那我们先看 CarBuilder.java 抽象类的程序：

```
package com.cbf4life;
```

```
import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 要什么顺序的车，你说，我给建造出来
 */
public abstract class CarBuilder {

    //建造一个模型，你要给我一个顺序要，就是组装顺序
    public abstract void setSequence(ArrayList<String> sequence);

    //设置完毕顺序后，就可以直接拿到这个车辆模型
    public abstract CarModel getCarModel();
}
```

这个抽象类比较简单，程序上也以后注释说明，不多说，我们看两个实现类，先看 BenzBuilder.java 的程序代码：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 各种设施都给了，我们按照一定的顺序制造一个奔驰车
 */
public class BenzBuilder extends CarBuilder {
    private BenzModel benz = new BenzModel();

    @Override
    public CarModel getCarModel() {
        return this.benz;
    }

    @Override
    public void setSequence(ArrayList<String> sequence) {
        this.benz.setSequence(sequence);
    }
```

```
}
```

下面是 BMWBuilder.java 程序代码：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 给定一个顺序，返回一个宝马车
 */
public class BMWBuilder extends CarBuilder {
    private BMWModel bmw = new BMWModel();

    @Override
    public CarModel getCarModel() {
        return this.bmw;
    }

    @Override
    public void setSequence(ArrayList<String> sequence) {
        this.bmw.setSequence(sequence);
    }

}
```

程序很简单，很实用，这就是我最希望的，简单而又实用，我欣赏这样的程序员。那我们再来看我们 Client.java 程序的修改：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 最终客户开始使用这个模型
 */
public class Client {
```

```
    public static void main(String[] args) {
        /*
         * 客户告诉牛叉公司，我要这样一个模型，然后牛叉公司就告诉我老大
         * 说要这样一个模型，这样一个顺序，然后我就来制造
         */
        ArrayList<String> sequence  = new ArrayList<String>();  //存放run的顺序
        sequence.add("engine boom");  //客户要求，run的时候时候先发动引擎
        sequence.add("start");  //启动起来
        sequence.add("stop");  //开了一段就停下来

        //要一个奔驰车：
        BenzBuilder benzBuilder = new BenzBuilder();
        //把顺序给这个builder类，制造出这样一个车出来
        benzBuilder.setSequence(sequence);
        //制造出一个奔驰车
        BenzModel benz = (BenzModel)benzBuilder.getCarModel();
        //奔驰车跑一下看看
        benz.run();

    }

}
```

运行结果如下：

```
奔驰车的引擎是这个声音的...
奔驰车跑起来是这个样子的...
奔驰车应该这样停车...
```

那如果我再想要个同样顺序的宝马车呢？很简单，Client.java 程序如下：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
```

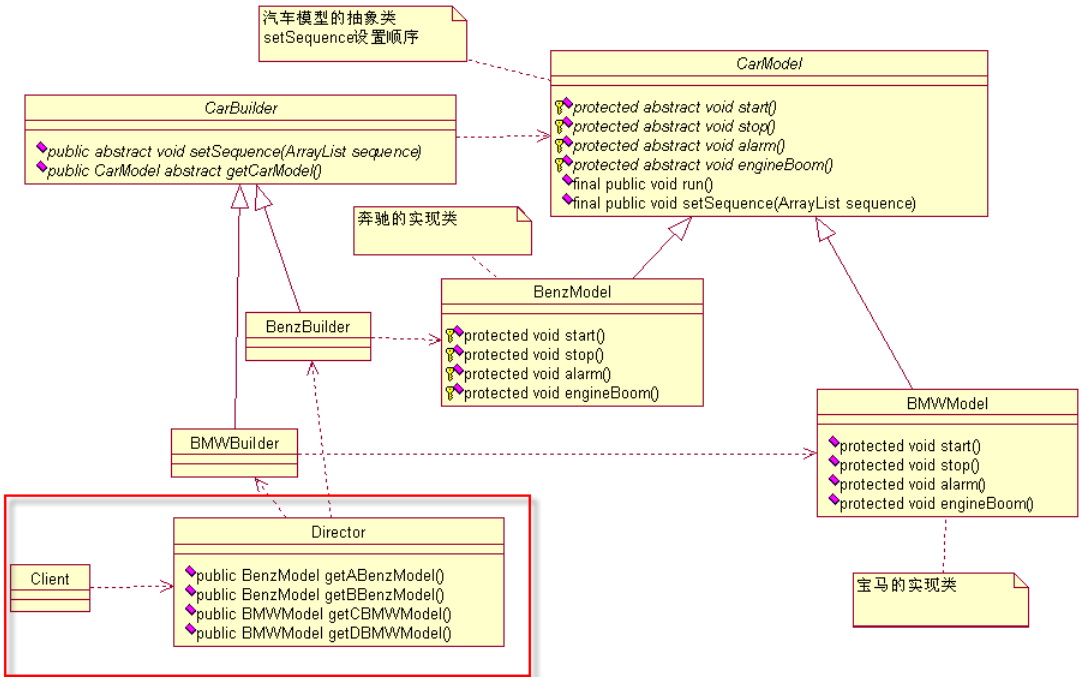
```
 * 最终客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        ArrayList<String> sequence  = new ArrayList<String>();  //存放run的顺序
        sequence.add("engine boom");  //客户要求，run的时候时候先发动引擎
        sequence.add("start");  //启动起来
        sequence.add("stop");  //开了一段就挺下来

        //要一个奔驰车：
        BenzBuilder benzBuilder = new BenzBuilder();
        //把顺序给这个builder类，制造出这样一个车出来
        benzBuilder.setSequence(sequence);
        //制造出一个奔驰车
        BenzModel benz = (BenzModel)benzBuilder.getCarModel();
        //奔驰车跑一下看看
        benz.run();

        //按照同样的顺序，我再要一个宝马
        BMWBuilder bmwBuilder = new BMWBuilder();
        bmwBuilder.setSequence(sequence);
        BMWModel bmw = (BMWModel)bmwBuilder.getCarModel();
        bmw.run();

    }

}
```

运行结果如下：

```
奔驰车的引擎是这个声音的...
奔驰车跑起来是这个样子的...
奔驰车应该这样停车...
宝马车的引擎是这个声音的...
宝马车跑起来是这个样子的...
宝马车应该这样停车...
```

看，是不是比刚开始直接访问产品类(Procuct)简单了很多吧，那这有个这样的需求，这四个过程（start,stop,alarm,engineBoom）按照排列组合有很多种，那我们怎么满足这

种需求呢？也就是要有个类来安排这几个方法组合，就像导演安排演员一样，那个先出场那个后出场，那个不出场，我们这个也叫导演类，那我们修改一下类图：



增加了 Director 类，这个类是做了层封装，类中的方法说明如下：

A 型号的奔驰车辆模型是只有启动（start）、停止(stop)方法，其他的引擎声音、喇叭都没有;

B 型号的奔驰车是先发动引擎（engine boom）,然后启动(star),再然后停车(stop),没有喇叭;

C 型号的宝马车是先喇叭叫一下（alarm），然后（start）,再然后是停车(stop)，引擎不轰鸣;

D 型号的宝马车就一个启动(start)，然后一路跑到黑，永动机，没有停止方法，没有喇叭，没有引擎轰鸣;

E 型号、F 型号…等等，可以有很多，启动(start)、停止(stop)、喇叭(alarm)、引擎轰鸣(engine boom)这四个方法在这个类中可以随意的自由组合,有几种呢？好像是排列组合，这个不会算，高中数学没学好，反正有很多种了，这里都可以实现。

我们看一下代码实现，Director.java 代码如下：

```
package com.cbf4life;

import java.util.ArrayList;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 导演安排顺序，生产车辆模型
 */
public class Director {
    private ArrayList<String> sequence = new ArrayList();
    private BenzBuilder benzBuilder = new BenzBuilder();
    private BMWBuilder bmwBuilder = new BMWBuilder();

    /*
     * A类型的奔驰车模型，先start,然后stop,其他什么引擎了，喇叭一概没有
     */
    public BenzModel getABenzModel(){
        //清理场景，这里是一些初级程序员不注意的地方
        this.sequence.clear();

        //这只ABenzModel的执行顺序
        this.sequence.add("start");
        this.sequence.add("stop");

        //按照顺序返回一个奔驰车
        this.benzBuilder.setSequence(this.sequence);
        return (BenzModel)this.benzBuilder.getCarModel();

    }

    /*
     * B型号的奔驰车模型，是先发动引擎，然后启动，然后停止，没有喇叭
     */
    public BenzModel getBBenzModel(){
        this.sequence.clear();

        this.sequence.add("engine boom");
        this.sequence.add("start");
        this.sequence.add("stop");

        this.benzBuilder.setSequence(this.sequence);
        return (BenzModel)this.benzBuilder.getCarModel();
    }

    /*
     * C型号的宝马车是先按下喇叭（炫耀嘛），然后启动，然后停止
```

```
     */
    public BMWModel getCBMWModel(){
        this.sequence.clear();

        this.sequence.add("alarm");
        this.sequence.add("start");
        this.sequence.add("stop");

        this.bmwBuilder.setSequence(this.sequence);
        return (BMWModel)this.bmwBuilder.getCarModel();
    }

    /*
     * D类型的宝马车只有一个功能，就是跑，启动起来就跑，永远不停止，牛叉
     */
    public BMWModel getDBMWModel(){
        this.sequence.clear();

        this.sequence.add("start");

        this.bmwBuilder.setSequence(this.sequence);
        return (BMWModel)this.benzBuilder.getCarModel();
    }

    /*
     * 这里还可以有很多方法，你可以先停止，然后再启动，或者一直停着不动，静态的
嘛
     * 导演类嘛，按照什么顺序是导演说了算
     */
}
```

大家看一下程序中有很多 this 调用，这个我一般是这样要求项目组成员的，如果你要调用类中的成员变量或方法，需要在前面加上 this 关键字，不加也能正常的跑起来，但是不清晰，加上 this 关键字，我就是要调用本类中成员变量或方法，而不是本方法的中的一个变量，还有 super 方法也是一样，是调用父类的的成员变量或者方法，那就加上这个关键字，不要省略，这要靠约束，还有就是程序员的自觉性，他要是死不悔改，那咱也没招。

上面每个方法都一个 this.sequence.clear()，这个估计你一看就明白，但是做为一个系统分析师或是技术经理一定要告诉告诉项目成员，ArrayList 和 HashMap 如果定义成类的成员变量，那你在方法中调用一定要做一个 clear 的动作，防止数据混乱，这个如果你发生过一次问题的话，比如 ArrayList 中出现一个“出乎意料”的数据，而你又花费了几个通宵才解

决这个问题，那你会有很深刻的印象。

然后 Client 程序就只与 Director 打交道了，牛叉公司要 A 类型的奔驰车 1W 辆，B 类型的奔驰车 100W 辆，C 类型的宝马车 1000W 辆，D 类型的我不要：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这里是牛叉公司的天下，他要啥我们给啥
 */
public class Client {

    public static void main(String[] args) {
        Director director = new Director();

        //1W辆A类型的奔驰车
        for(int i=0;i<10000;i++){
            director.getABenzModel().run();
        }

        //100W辆B类型的奔驰车
        for(int i=0;i<1000000;i++){
            director.getBBenzModel().run();
        }

        //1000W量C类型的宝马车
        for(int i=0;i<10000000;i++){
            director.getCBMWModel().run();
        }
    }

}
```

清晰，简单吧，我们写程序重构的最终目的就是这个，简单，清晰，代码是让人看的，不是写完就完事了，我一直在教育我带的团队，Java 程序不是像我们前辈写那个二进制代码、汇编一样，写完基本上就自己能看懂，别人看就跟看天书一样，现在的高级语言，要像写中文汉字一样，你写的，别人能看懂。

整个程序编写完毕，而且简洁明了，这就是建造者模式，中间有几个角色需要说明一下：

Client 就是牛叉公司，这个到具体的应用中就是其他的模块或者页面；

CarModel 以及两个实现类 BenzModel 和 BMWModel 叫做产品类(Product Class)，这个产品类实现了模板方法模式，也就是有模板方法和基本方法，这个参考上一节的模板方法模式；

CarBuilder 以及两个实现类 BenzBuilder 和 BMWBuilder 叫做建造者(Builder Class)，在上面的那个例子中就是我和我的团队，负责建造 Benz 和 BMW 车模，按照指定的顺序；

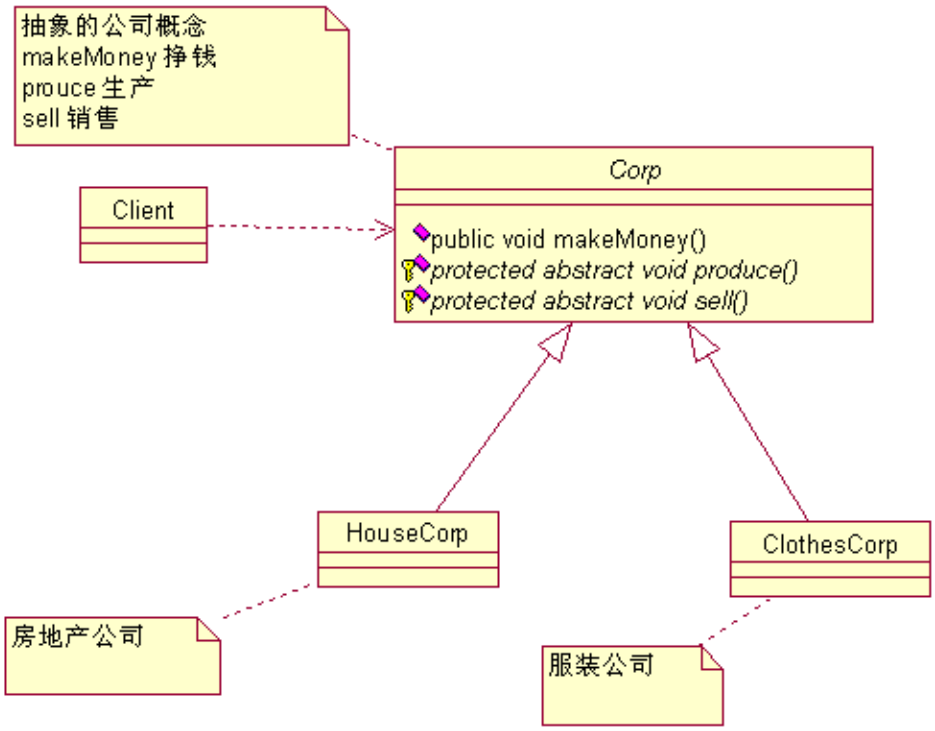
Director 类叫做导演类(Director Class),负责安排已有模块的顺序，然后告诉 Builder 开始建造，在上面的例子中就是我们的老大，Client 找到老大，说我要这个，这个，那个类型的车辆模型，然后老大就把命令传递给我，我和我的团队就开始拼命的建造，于是一个项目建设完毕了。

大家看到这里估计就开始犯嘀咕了，这个建造者模式和工厂模式非常相似呀，Yes，是的，是非常相似，但是记住一点你就可以游刃有余的使用了：建造者模式最主要功能是基本方法的调用顺序安排，也就是这些基本方法已经实现了；而工厂方法则重点是创建，你要什么对象我创造一个对象出来，组装顺序则不是他关心的。

建造者模式使用的场景，一是产品类非常的复杂，或者产品类中的调用顺序不同产生了不同的效能,这个时候使用建造者模式是非常合适,我曾在一个银行交易类项目中遇到了这个问题，一个产品的定价计算模型有 N 多种，每个模型有固定的计算步骤，计算非常复杂，项目中就使用了建造者模式；二是“ 在对象创建过程中会使用到系统中的一些其它对象，这些对象在产品对象的创建过程中不易得到”，这个是我没有遇到过的，创建过程中不易得到？那为什么在设计阶段不修正这个问题，创建的时候都不易得到耶！

11.桥梁模式【Bridge Pattern】

今天我要说说我自己，梦想中的我自己，我身价过亿，有两个大公司，一个是房地产公司，一个是服装制造业，这两个公司都很赚钱，天天帮我在累加财富，其实是什么公司我倒是不关心，我关心的是是不是在赚钱，赚了多少，这才是我关心的，我是商人呀，唯利是图是我的本性，偷税漏税是我的方法，欺上瞒下、压榨员工血汗我是的手段嘛，我先用类图表示一下我这两个公司：



类图很简单，声明了一个 Corp 抽象类，定义一个公司的抽象模型，公司首要是赚钱的，不赚钱谁开公司，做义务或善举那也是有背后利益支撑的，我还是赞成这句话“天下熙熙，皆为利来；天下壤壤，皆为利往”，那我们先看 Corp 类的源代码：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个公司的抽象类
 */
public abstract class Corp {
```

```
    /*
     * 是公司就应该有生产把，甭管是什么软件公司还是制造业公司
     * 那每个公司的生产的东西都不一样，所以由实现类来完成
     */
    protected abstract void  produce();

    /*
     * 有产品了，那肯定要销售呀，不销售你公司怎么生存
     */
    protected abstract void sell();

    //公司是干什么的？赚钱的呀，不赚钱傻子才干
    public void makeMoney(){

        //每个公司都是一样，先生产
        this.produce();

        //然后销售
        this.sell();

    }

}
```

合适的方法存在合适的类中，这个基本上是每本 Java 基础书上都会讲的，但是到实际的项目中应用的时候就不是这么回事儿了，正常的很。我们继续看两个实现类如何实现的，先看 HouseCorp 类，这个是我最赚钱的公司：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 房地产公司，按照翻译来说应该叫realty corp，这个是比较准确的翻译
 * 但是我问你房地产公司翻译成英文，你第一反应什么？对嘛还是house corp!
 */
public class HouseCorp extends Corp {

    //房地产公司就是盖房子
    protected void produce() {
        System.out.println("房地产公司盖房子...");
```

```
    }

    //房地产卖房子，自己住那可不赚钱
    protected void sell() {
        System.out.println("房地产公司出售房子...");
    }

    //房地产公司很High了，赚钱，计算利润
    public void makeMoney(){
        super.makeMoney();
        System.out.println("房地产公司赚大钱了...");
    }

}
```

然后看服装公司，虽然不景气，但好歹也是赚钱的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 服装公司，这个行当现在不怎么样
 */
public class ClothesCorp extends Corp {

    //服装公司生产的就是衣服了
    protected void produce() {
        System.out.println("服装公司生产衣服...");
    }

    //服装公司买服装，可只卖服装，不买穿衣服的模特
    protected void sell() {
        System.out.println("服装公司出售衣服...");
    }

    //服装公司不景气，但怎么说也是赚钱行业也
    public void makeMoney(){
        super.makeMoney();
        System.out.println("服装公司赚小钱...");
    }

}
```

两个公司都有了，那肯定有人会关心两个公司的运营情况呀，我自己怎么也要知道它是生产什么的，赚多少钱吧，那看看这个 Client.java 是什么样子的：

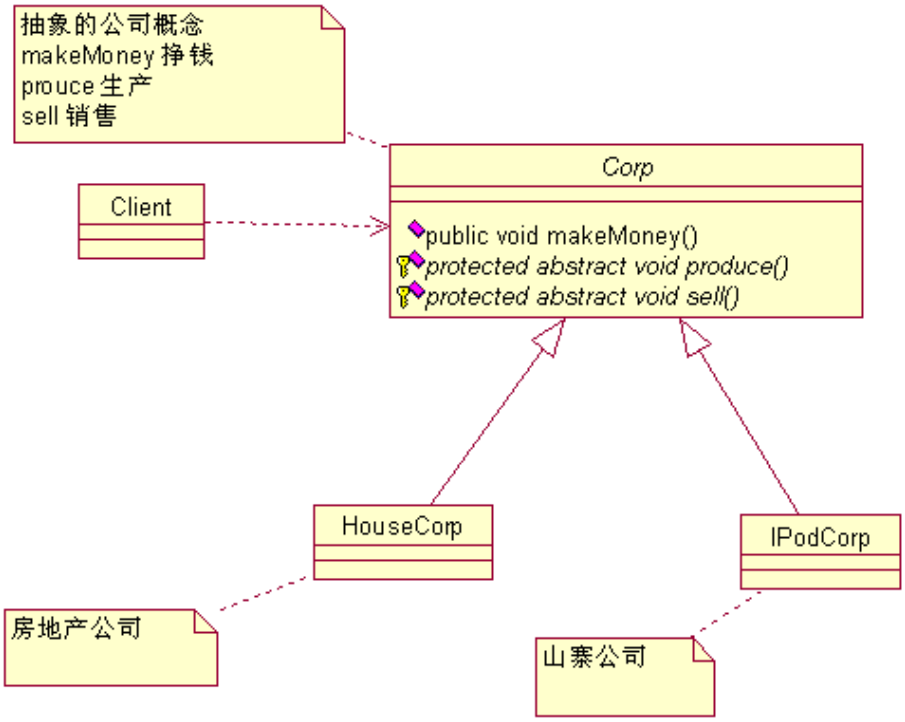
```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要关心我自己的公司了
 */
public class Client {

    public static void main(String[] args) {
        System.out.println("-------房地产公司是这个样子运行的-------");
        //先找到我的公司
        HouseCorp houseCorp =new HouseCorp();
        //看我怎么挣钱
        houseCorp.makeMoney();
        System.out.println("\n");

        System.out.println("-------服装公司是这样运行的-------");
        ClothesCorp clothesCorp = new ClothesCorp();
        clothesCorp.makeMoney();

    }
}
```

看，这个代码很简单，运行结果如下：

```
-------房地产公司是这个样子运行的-------
房地产公司盖房子...
房地产公司出售房子...
房地产公司赚大钱了...


-------服装公司是这样运行的-------
服装公司生产衣服...
服装公司出售衣服...
服装公司赚小钱...
```

上述代码完全可以描述我现在的公司，但是你要知道万物都是运动的，你要用运动的眼光看问题，我公司也会发展，终于在有一天我觉得赚钱速度太慢，于是我上下疏通，左右打关系，终于开辟了一条赚钱的康庄大道：生产山寨产品，什么产品呢？就是市场上什么牌子的东西火爆我生产什么牌子的东西，甭管是打火机还是电脑，只要它火爆，我就生产，赚过了高峰期就换个产品，打一枪换一个牌子，不承担售后成本、也不担心销路问题，我只有正品的十分之一的价格，你买不买？哈哈，赚钱呀！

企业的方向定下来了，通过调查，市场上前段时间比较火爆的是苹果 iPod，那咱就生产这个，把服装厂该成 iPod 生产厂，看类图的变化：



好，我的企业改头换面了，开始生产 iPod 产品了，看我 IPodCorp 类的实现：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我是山寨老大，你流行啥我就生产啥
 * 现在流行iPod
 */
```

```
public class IPodCorp extends Corp {

    //我开始生产iPod了
    protected void produce() {
        System.out.println("我生产iPod...");
    }

    //山寨的iPod很畅销，便宜呀
    protected void sell() {
        System.out.println("iPod畅销...");
    }

    //狂赚钱
    public void makeMoney(){
        super.makeMoney();
        System.out.println("我赚钱呀...");
    }

}
```

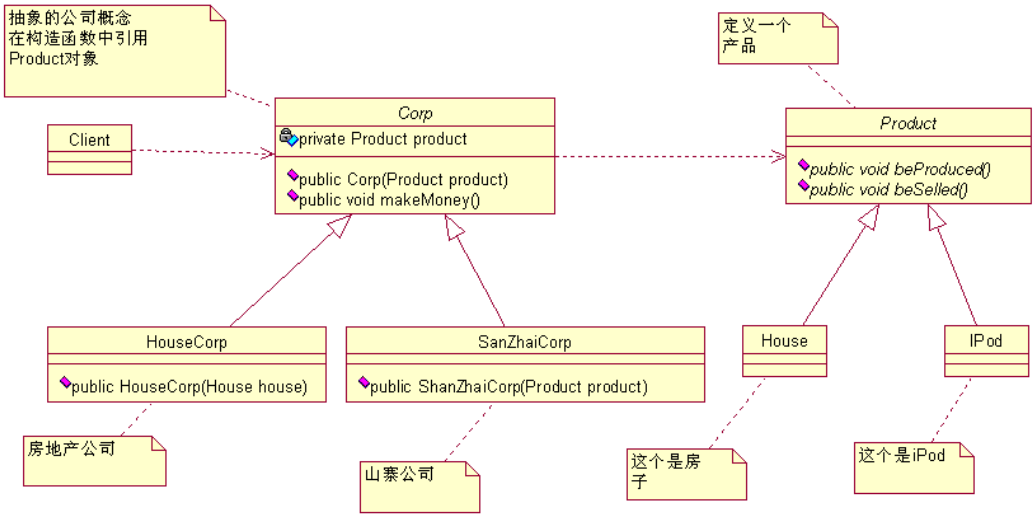
这个厂子修改完毕了，我这个董事长还要去看看到底生产什么的，看这个 Client.java 程序：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要关心我自己的公司了
 */
public class Client {

    public static void main(String[] args) {
        System.out.println("-------房地产公司是这个样子运行的-------");
        //先找到我的公司
        HouseCorp houseCorp =new HouseCorp();
        //看我怎么挣钱
        houseCorp.makeMoney();
        System.out.println("\n");

        System.out.println("-------山寨公司是这样运行的-------");
        IPodCorp iPodCorp = new IPodCorp();
```

```
        iPodCorp.makeMoney();

    }
}
```

确实，只用修改了几句话，我的服装厂就开始变成山寨 iPod 生产车间，然后就看我的财富在积累积累，你想呀山寨的东西不需要特别的销售渠道（正品到哪里我就到哪里），不需要维修成本（大不了给你换个，你还想咋地，过了高峰期我就改头换面了你找谁维修去，投诉？投诉谁呢？），不承担广告成本（正品在打广告，我还需要吗？需要吗？），但是我也有犯愁的时候，我这是个山寨工厂，要及时的生产出市场上流行产品，转型要快，要灵活，今天从生产 iPod 转为生产 MP4,明天再转为生产上网本，这个都需要灵活的变化，不要限制的太死，那问题来了，每次我的厂房，我的工人，我的设备都在，不可能每次我换个山寨产品我的厂子就彻底不要了，这不行，成本忒高了点，那怎么办？

Thinking，Thinking…..，I got an idea!，这样设计：



Corp 类和 Product 类建立一个关联关系，可以彻底解决我以后山寨公司生产产品的问题，看程序说话，先看 Product 抽象类：

```
package com.cbf4life.implementor;

/**
 * @author cbf4Life cbf4life@126.com
```

```
 * I'm glad to share my knowledge with you all.
 * 这是我整个集团公司的产品类
 */
public abstract class Product {

    //甭管是什么产品它总要是能被生产出来
    public abstract void beProducted();

    //生产出来的东西，一定要销售出去，否则扩本呀
    public abstract void beSelled();

}
```

简单，忒简单了，看 House 产品类：

```
package com.cbf4life.implementor;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这是我集团公司盖的房子
 */
public class House extends Product {

    //豆腐渣就豆腐渣呗，好歹也是个房子
    public void beProducted() {
        System.out.println("生产出的房子是这个样子的...");
    }

    //虽然是豆腐渣，也是能够销售出去的
    public void beSelled() {
        System.out.println("生产出的房子卖出去了...");
    }

}
```

不多说，看 Clothes 产品类：

```
package com.cbf4life.implementor;

/**
```

```
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我集团公司生产的衣服
 */
public class Clothes extends Product {

    public void beProducted() {
        System.out.println("生产出的衣服是这个样子的...");
    }

    public void beSelled() {
        System.out.println("生产出的衣服卖出去了...");
    }

}
```

下面是 iPod 产品类：

```
package com.cbf4life.implementor;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 生产iPod了
 */
public class IPod extends Product {

    public void beProducted() {
        System.out.println("生产出的iPod是这个样子的...");
    }

    public void beSelled() {
        System.out.println("生产出的iPod卖出去了...");
    }

}
```

产品类是有了，那我们再看 Corp 抽象类：

```
package com.cbf4life.abstraction;
```

```
import com.cbf4life.implementor.Product;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个公司的抽象类
 */
public abstract class Corp {

    //定义一个产品对象，抽象的了，不知道具体是什么产品
    private Product product;

    //构造函数，由子类定义传递具体的产品进来
    public Corp(Product product){
        this.product = product;
    }

    //公司是干什么的？赚钱的呀，不赚钱傻子才干
    public void makeMoney(){

        //每个公司都是一样，先生产
        this.product.beProducted();

        //然后销售
        this.product.beSelled();

    }


}
```

这里多了个有参构造，其目的是要继承的子类都必选重写自己的有参构造函数，把产品类传递进来，再看子类 HouseCorp 的实现：

```
package com.cbf4life.abstraction;

import com.cbf4life.implementor.House;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 房地产公司，按照翻译来说应该叫realty corp，这个是比较准确的翻译
```

```
 * 但是我问你房地产公司翻译成英文，你第一反应什么？对嘛还是house corp!
 */
public class HouseCorp extends Corp {

    //定义传递一个House产品进来
    public HouseCorp(House house){
        super(house);
    }

    //房地产公司很High了，赚钱，计算利润
    public void makeMoney(){
        super.makeMoney();
        System.out.println("房地产公司赚大钱了...");
    }

}
```

理解上没有多少难度，不多说，继续看山寨公司的实现：

```
package com.cbf4life.abstraction;

import com.cbf4life.implementor.Product;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我是山寨老大，你流行啥我就生产啥
 */
public class ShanZhaiCorp extends Corp {

     //产什么产品，不知道，等被调用的才知道
    public ShanZhaiCorp(Product product){
        super(product);
    }

    //狂赚钱
    public void makeMoney(){
        super.makeMoney();
        System.out.println("我赚钱呀...");
    }

}
```

HouseCorp 类和 ShanZhaiCorp 类的区别是在有参构造的参数类型上，HouseCorp 类比较明确，我就是只要 House 类，所以直接定义传递进来的必须是 House 类， 一个类尽可能少的承担职责，那方法也是一样，既然 HouseCorp 类已经非常明确只生产 House 产品，那为什么不定义成 House 类型呢？ShanZhaiCorp 就不同了，它是确定不了生产什么类型。

好了，两大对应的阵营都已经产生了，那我们再看 Client 程序：

```
package com.cbf4life;

import com.cbf4life.abstraction.HouseCorp;
import com.cbf4life.abstraction.ShanZhaiCorp;
import com.cbf4life.implementor.Clothes;
import com.cbf4life.implementor.House;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要关心我自己的公司了
 */
public class Client {

    public static void main(String[] args) {
        House house = new House();
        System.out.println("-------房地产公司是这个样子运行的-------");
        //先找到我的公司
        HouseCorp houseCorp =new HouseCorp(house);
        //看我怎么挣钱
        houseCorp.makeMoney();
        System.out.println("\n");

        //山寨公司生产的产品很多，不过我只要指定产品就成了
        System.out.println("-------山寨公司是这样运行的-------");
        ShanZhaiCorp shanZhaiCorp = new ShanZhaiCorp(new Clothes());
        shanZhaiCorp.makeMoney();

    }
}
```

运行结果如下：

```
-------房地产公司是这个样子运行的-------
```

```
生产出的房子是这个样子的...
生产出的房子卖出去了...
房地产公司赚大钱了...



-------山寨公司是这样运行的-------
生产出的衣服是这个样子的...
生产出的衣服卖出去了...
我赚钱呀...
```

这个山寨公司的前身是生产衣服的，那我现在要修改一下，生产 iPod，看如下的变化：

```
package com.cbf4life;

import com.cbf4life.abstraction.HouseCorp;
import com.cbf4life.abstraction.ShanZhaiCorp;
import com.cbf4life.implementor.House;
import com.cbf4life.implementor.IPod;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要关心我自己的公司了
 */
public class Client {

    public static void main(String[] args) {
        House house = new House();
        System.out.println("-------房地产公司是这个样子运行的-------");
        //先找到我的公司
        HouseCorp houseCorp =new HouseCorp(house);
        //看我怎么挣钱
        houseCorp.makeMoney();
        System.out.println("\n");

        //山寨公司生产的产品很多，不过我只要制定产品就成了
        System.out.println("-------山寨公司是这样运行的-------");
        //ShanZhaiCorp shanZhaiCorp = new ShanZhaiCorp(new
Clothes());
        ShanZhaiCorp shanZhaiCorp = new ShanZhaiCorp(new IPod());
        shanZhaiCorp.makeMoney();

    }
```

```
}
```

运行结果如下：

```
-------房地产公司是这个样子运行的-------
生产出的房子是这个样子的...
生产出的房子卖出去了...
房地产公司赚大钱了...



-------山寨公司是这样运行的-------
生产出的iPod是这个样子的...
生产出的iPod卖出去了...
我赚钱呀...
```

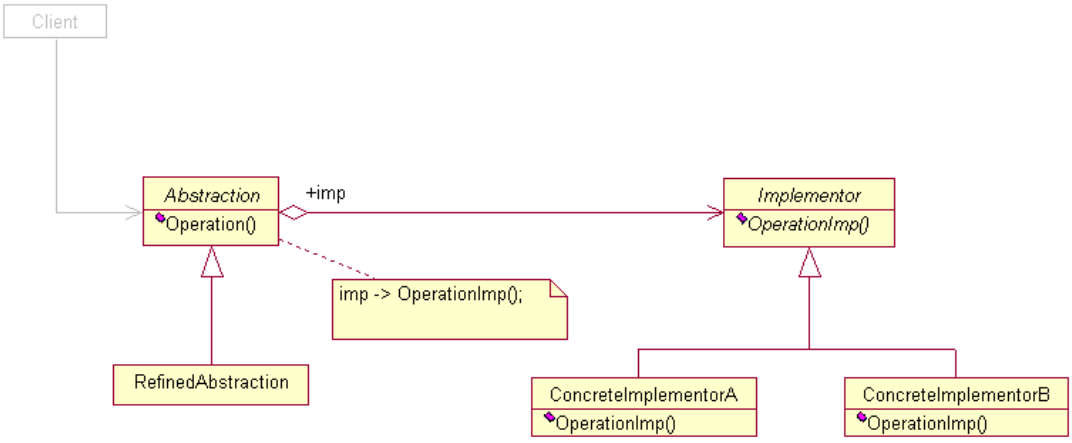
看代码上的黄色底色的代码，就修改了这一一句话代码就完成了生产产品的转换。那我们深入的思考一下，既然万物都是运动的，我现在只有房地产公司和山寨公司，那以后我会不会增加一些其他的公司呢？或者房地产公司会不会对业务进行细化，比如分为公寓房公司，别墅公司，以及商业房公司等等呢？那我告诉你，会的，绝对的会的，但是你发觉没，这种变化对我们上面的类图没有大的修改，充其量是扩展，你看呀：

增加公司，你要么继承 Corp 类，要么继承 HouseCorp 或 ShanZhaiCorp，不用再修改原有的类了；

增加产品，继承 Product 类，或者继承 House 类，你要把房子分为公寓房、别墅、商业用房等等；

你都是在扩展，唯一你要修改的就是 Client 类，你类都增加了哪能不修改调用呢，也就是说 Corp 类和 Product 类都可以自由的扩展，而不会对整个应用产生太的变更，这就是桥梁模式。

为什么叫桥梁模式？我们看一下桥梁模式的通用类图：



看到中间那根带箭头的线了吗？是不是类似一个桥，连接了两个类？所以就叫桥梁模式。我们再把桥梁模式的几个概念熟悉一下，大家有没有注意到我把 Corp 类以及它的两个实现类放到了 Abstraction 包中，把 House 以及相关的三个实现类放到了 Implementor 包中，这两个包分别对应了桥梁模式的业务抽象角色（Abstraction）和业务实现角色（Implementor），这两个角色我估计没几个人能说的明白，特别是看了“四人帮”的书或者是那本非常有名的、比砖头还要厚的书，你会越看越糊涂，忒专业化，有点像看政府的红头文件，什么都说了，可好像又什么都没有说。这两个角色大家只要记住一句话就成：业务抽象角色引用业务实现角色，或者说业务抽象角色的部分实现是由业务实现角色完成的，很简单，别想那么复杂了。

桥梁模式的优点就是类间解耦，我们上面已经提到，两个角色都可以自己的扩展下去，不会相互影响，这个也符合 OCP 原则。

今天说到桥梁模式，那就多扯几句，大家对类的继承有什么看法吗？继承的优点有很多，可以把公共的方法或属性抽取，父类封装共性，子类实现特性，这是继承的基本功能，缺点有没有？有，强关联关系，父类有个方法，你子类也必须有这个方法，是不可选择的，那这会带来扩展性的问题，我举个简单的例子来说明这个问题：Father 类有一个方法 A，Son 继承了这个方法，然后 GrandSon 也继承了这个方法，问题是突然有一天 Son 要重写父类的这个方法，他敢做吗？绝对不敢！GrandSon 可是要用从 Father 继承过来的方法 A，你修改了，那就要修改 Son 和 GrandSon 之间的关系，那这个风险就大了去。

今天讲的这个桥梁模式就是这一问题的解决方法，桥梁模式描述了类间弱关联关系，还说上面的那个例子，Fater 类完全可以把可能会变化的方法放出去，Son 子类要有这个方法很简答，桥梁搭过去，获得这个方法，GrandSon 也一样，即使你 Son 子类不想使用这个方法了，

也没关系，对GrandSon 不产生影响，他不是从你Son 中继承来的方法！

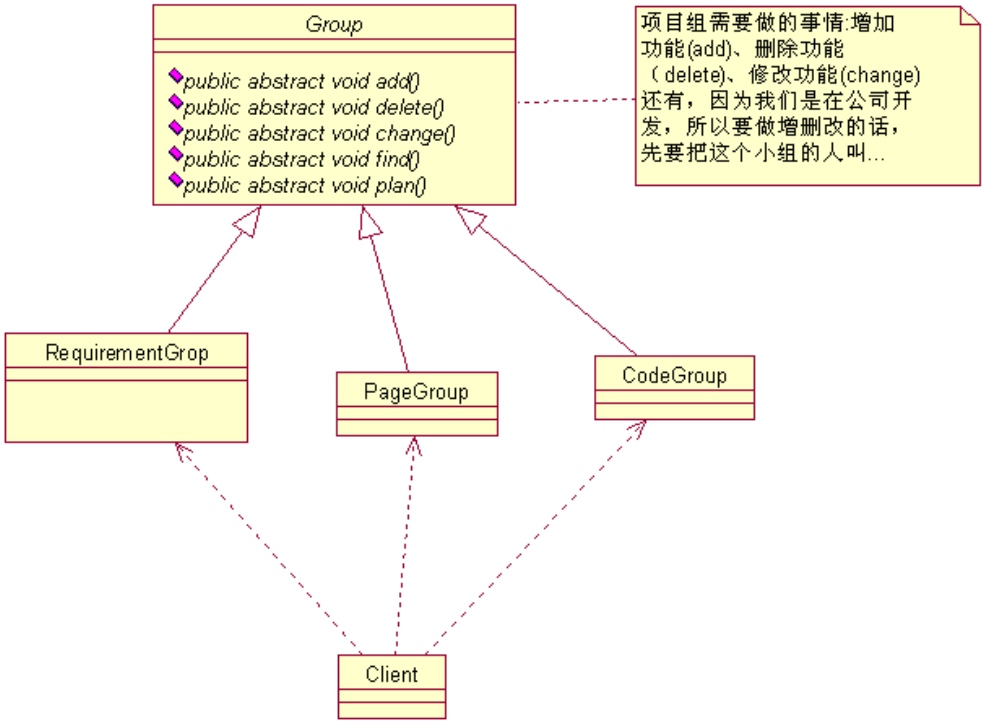
继承不能说它不好，非常好，但是有缺点的，我们可以扬长避短，对于比较明确不发生变化的，则通过继承来完成，若不能确定是否会发生变化的，那就认为是会发生变化，则通过桥梁模式来解决，这才是一个完美的世界。

12.命令模式【Command Pattern】

今天讲命令模式，这个模式从名字上看就很简单，命令嘛，老大发命令，小兵执行就是了，确实是这个意思，但是更深化了，用模式来描述真是是世界的命令情况。正在看这本书的你，我猜测分为两类：已经工作的和没有工作的，先说没有工作的，那你为啥要看这本书，为了以后工作呗，只要你参见工作，你肯定会待在项目组，那今天我们就以项目组为例子来讲述命令模式。

我是我们部门的项目经理，就是一个项目的头，在中国做项目，项目经理就是什么都要懂，什么都要管，做好了项目经理能分到一杯羹，做不好都是你项目经理的责任，这个是绝对的，我带过太多的项目，行政命令一压下来，那就一条道，做完做好！我们虽然是一个集团公司，但是我们部门是独立核算的，就是说呀，我们部门不仅仅为我们集团服务，还可以为其他甲方服务，赚取更多的外快，所以俺们的工资才能是中上等。在 2007 年我带领了一个项目，比较小，但是钱可不少，是做什么的呢？为一家旅行社建立一套内部管理系统，管理他的客户、旅游资源、票务以及内部管理，整体上类似一个小型的 ERP 系统，门店比较多，员工也比较多，但是需求比较明确，因为他们之前有一套自己购买的内部管理系统，这次变动部分模块基本上是翻版，而且旅行社有自己的 IT 部门，比较好相处，都是技术人员，没有交流鸿沟嘛。

这个项目的成员分工也是采用了常规的分工方式，分为需求组（Requirement Group，简称 RG）、美工组（Page Group,简称 PG）、代码组（我们内部还有一个比较优雅的名字：逻辑实现组，这里使用大家经常称呼的名称吧，英文缩写叫 Code Group，简称 CG），总共加上我这个项目经理正好十个人，刚开始的时候客户（也就是旅行社，甲方）还是很乐意和我们每个组探讨，比如和需求组讨论需求，和美工讨论页面，和代码组讨论实现，告诉他们修改这里，删除这里，增加这些等等，这是一种比较常见的甲乙方合作模式，甲方深入到乙方的项目开发中，我们把这个模式用类图表示一下：



这个类图很简单，客户和三个组都有交流，这也合情合理，那我们看看这个的实现，首先看抽象类，我们是面向接口或抽象类编程的嘛：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 项目组分成了三个组，每个组还是要接受增删改的命令
 */
public abstract class Group {

    //甲乙双方分开办公，你要和那个组讨论，你首先要找到这个组
    public abstract void find();

    //被要求增加功能
    public abstract void add();

    //被要求删除功能
    public abstract void delete();

    //被要求修改功能
    public abstract void change();
```

```
    //被要求给出所有的变更计划
    public abstract void plan();


}
```

大家看抽象类中的每个方法，是不是每个都是一个命令？找到它，增加，删除，给我计划！是不是命令，这也就是命令模式中的命令接收者角色(Receiver)，等会细讲。我们再看三个实现类，需求组最重要，没有需求你还设计个 P 呀，看 RequirmentGroup 类的实现：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 需求组的职责是和客户谈定需求，这个组的人应该都是业务领域专家
 */
public class RequirementGroup extends Group {

    //客户要求需求组过去和他们谈
    public void find() {
        System.out.println("找到需求组...");
    }

    //客户要求增加一项需求
    public void add() {
        System.out.println("客户要求增加一项需求...");
    }

    //客户要求修改一项需求
    public void change() {
        System.out.println("客户要求修改一项需求...");
    }

    //客户要求删除一项需求
    public void delete() {
        System.out.println("客户要求删除一项需求...");
    }

    //客户要求出变更计划
    public void plan() {
        System.out.println("客户要求需求变更计划...");
```

```
        }

    }
```

需求组有了，我们再看美工组，美工组也很重要，是项目的脸面，客户最终接触到的还是界面，这个非常重要，看 PageGroup 的实现：

```
    package com.cbf4life;

    /**
     * @author cbf4Life cbf4life@126.com
     * I'm glad to share my knowledge with you all.
     * 美工组的职责是设计出一套漂亮、简单、便捷的界面
     */
    public class PageGroup extends Group {

        //首先这个美工组应该被找到吧，要不你跟谁谈？
        public void find() {
            System.out.println("找到美工组...");
        }

        //美工被要求增加一个页面
        public void add() {
            System.out.println("客户要求增加一个页面...");
        }

        //客户要求对现有界面做修改
        public void change() {
            System.out.println("客户要求修改一个页面...");
        }

        //甲方是老大，要求删除一些页面
        public void delete() {
            System.out.println("客户要求删除一个页面...");
        }

        //所有的增删改那要给出计划呀
        public void plan() {
            System.out.println("客户要求页面变更计划...");
        }
    }
```

最后看代码组，这个组的成员一般都是比较闷骚型的，不多说话，但多做事儿，比较沉闷，这是这个组的典型特点，我们来看看这个 CodeGroup 类：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 代码组的职责是实现业务逻辑，当然包括数据库设计了
 */
public class CodeGroup extends Group {

    //客户要求代码组过去和他们谈
    public void find() {
        System.out.println("找到代码组...");
    }


    //客户要求增加一项功能
    public void add() {
        System.out.println("客户要求增加一项功能...");
    }

    //客户要求修改一项功能
    public void change() {
        System.out.println("客户要求修改一项功能...");
    }

    //客户要求删除一项功能
    public void delete() {
        System.out.println("客户要求删除一项功能...");
    }

    //客户要求出变更计划
    public void plan() {
        System.out.println("客户要求代码变更计划...");
    }

}
```

整个项目的三个支柱都已经产生了，那看客户怎么和我们谈。客户刚刚开始给了我们一份他们自己写的一份需求，还是比较完整的，需求组根据这份需求写了一份分析说明书，客户一看，

不对，要增加点需求，看程序：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户就是甲方，给我们钱的一方，是老大
 */
public class Client {

    public static void main(String[] args) {

        //首先客户找到需求组说，过来谈需求，并修改
        System.out.println("-------------客户要求增加一个需求
-----------------");
        Group rg = new RequirementGroup();
        //找到需求组
        rg.find();

        //增加一个需求
        rg.add();

        //要求变更计划
        rg.plan();

    }
}
```

运行的结果如下：

```
-------------客户要求增加一个需求-----------------
找到需求组...
客户要求增加一项需求...
客户要求需求变更计划...
```

好的，客户的需求达到了，需求刚开始没考虑周全，增加需求是在所难免的嘛，理解理解。然后又过了段时间，客户说“界面多画了一个，过来谈谈”，于是：

```
package com.cbf4life;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户就是甲方，给我们钱的一方，是老大
 */
public class Client {

    public static void main(String[] args) {

        //首先客户找到美工组说，过来谈页面，并修改
        System.out.println("-------------客户要求删除一个页面
-----------------");
        Group pg = new PageGroup();
        //找到需求组
        pg.find();

        //增加一个需求
        pg.delete();

        //要求变更计划
        pg.plan();

    }
}
```

运行结果如下：

```
-------------客户要求增加一个页面-----------------
找到美工组...
客户要求删除一个页面...
客户要求页面变更计划...
```

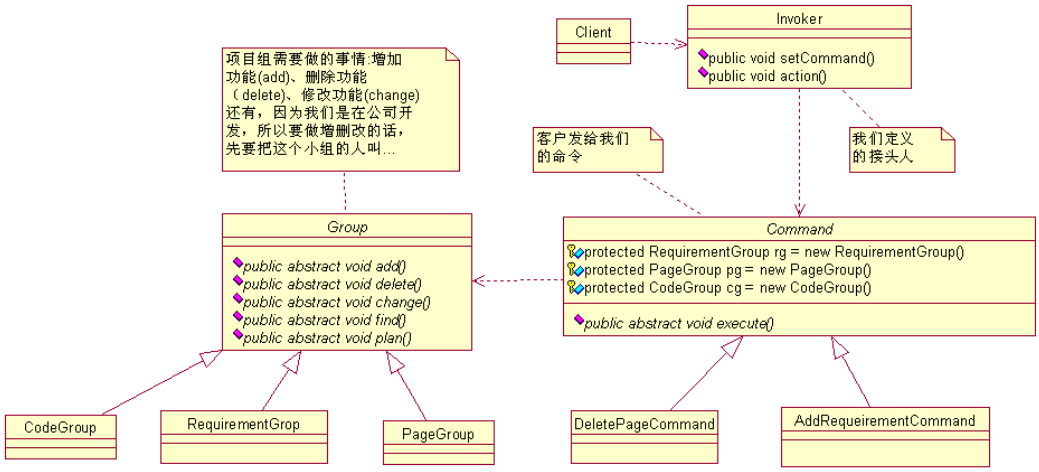
好了，界面也谈过了，应该没什么大问题了吧。过了一天后，客户又让代码组过去，说是数据库设计问题，然后又叫美工过去，布置了一堆命令，这个我就不一一写了，大家应该能够体会到，你做过项目的话，这种体会更深，客户让修改你不修改？项目不想做了你！

但是问题来了，我们修改可以，但是每次都是叫一个组去，布置个任务，然后出计划，次次都这样，如果让你当甲方也就是客户，你烦不烦？而且这种方式很容易出错误呀，而且还真发生过，客户把美工叫过去了，要删除，可美工说需求是这么写的，然后客户又命令需求组过去，

一次次的折腾，客户也烦躁了，于是直接抓住我这个项目经理说：

“我不管你们内部怎么安排，你就给我找个接头人，我告诉他怎么做，删除页面了，增加功能了，你们内部怎么处理，我就告诉他我要干什么就成了…”

我一听，好呀，这也正是我想要的，我项目组的兄弟们也已经受不了了，于是我改变了一下我的处理方式，看看类图：



类图中增加了不少，看着也比较清晰，比较简单的（Command 抽象类与 Group 抽象类是没有关联关系的，与 Group 的三个实现类是有关联关系，为了线条不交叉就直接画上父类有关系），增加的几个类说明如下：

Command 抽象类：客户发给我们的命令，定义三个工作组的成员变量，供子类使用；定义一个抽象方法 execute，由子类来实现；

Invoker 实现类:项目接头人，setComand 接受客户发给我我们的命令，action 方法是执行客户的命令（方法名写成是 action 是与 command 的 execute 区分开，避免混淆）

我们先看 Command 抽象类代码：

```
package com.cbf4life.command;

import com.cbf4life.receiver.CodeGroup;
import com.cbf4life.receiver.PageGroup;
import com.cbf4life.receiver.RequirementGroup;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 命令的抽象类，我们把客户发出的命令定义成一个一个的对象
```

```
 */
public abstract class Command {

    //把三个组都定义好，子类可以直接使用
    private RequirementGroup rg = new RequirementGroup();  //需求组
    private PageGroup pg = new PageGroup();  //美工组
    private CodeGroup cg = new CodeGroup();  //代码组;

    //只要一个方法，你要我做什么事情
    public abstract void execute();

}
```

这个简单，看两个具体的实现类，先看 AddRequeirementCommand 类，这个类的作用就是增加一项需求。

```
package com.cbf4life.command;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 增加一项需求
 */
public class AddRequirementCommand extends Command {
    //执行增加一项需求的命令
    public void execute() {
        //找到需求组
        super.rg.find();

        //增加一份需求
        super.rg.add();

        //给出计划
        super.rg.plan();
    }

}
```

看删除一个页面的命令,DeletePageCommand 类：

```
package com.cbf4life.command;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 删除一个页面的命令
 */
public class DeletePageCommand extends Command {

    //执行删除一个页面的命令
    public void execute() {
        //找到页面组
        super.pg.find();

        //删除一个页面
        super.rg.delete();

        //给出计划
        super.rg.plan();
    }

}
```

Command 抽象类还可以有很多的子类，比如增加一个功能命令（AddCodeCommand），删除一份需求命令（DeleteRequirementCommand）等等，这里就不用描述了，都很简单。

我们再看我们的接头人，就是 Invoker 类的实现：

```
package com.cbf4life.invoker;

import com.cbf4life.command.Command;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 接头人的职责就是接收命令，并执行
 */
public class Invoker {
    //什么命令
    private Command command;

    //客户发出命令
    public void setCommand(Command command){
        this.command = command;
```

```
    }

    //执行客户的命令
    public void action(){
        this.command.execute();
    }
}
```

这个是更简单了，简单是简单，可以帮我们解决很多问题，我们再看一下客户提出变更的过程：

```
package com.cbf4life;

import com.cbf4life.command.AddRequirementCommand;
import com.cbf4life.command.Command;
import com.cbf4life.invoker.Invoker;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户就是甲方，给我们钱的一方，是老大
 */
public class Client {

    public static void main(String[] args) {
        //定义我们的接头人
        Invoker xiaoSan = new Invoker();  //接头人就是我小三

        //客户要求增加一项需求
        System.out.println("-------------客户要求增加一项需求
-----------------");
        //客户给我们下命令来
        Command command = new AddRequirementCommand();

        //接头人接收到命令
        xiaoSan.setCommand(command);

        //接头人执行命令
        xiaoSan.action();

    }
}
```

运行结果如下：

```
-------------客户要求增加一项需求-----------------
找到需求组...
客户要求增加一项需求...
客户要求需求变更计划...
```

那我们看看，如果客户要求删除一个页面，那我们的修改有多大呢？想想，Look：

```
package com.cbf4life;

import com.cbf4life.command.Command;
import com.cbf4life.command.DeletePageCommand;
import com.cbf4life.invoker.Invoker;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户就是甲方，给我们钱的一方，是老大
 */
public class Client {

    public static void main(String[] args) {
        //定义我们的接头人
        Invoker xiaoSan = new Invoker();  //接头人就是我小三

        //客户要求增加一项需求
        System.out.println("-------------客户要求删除一个页面
-----------------");
        //客户给我们下命令来
        //Command command = new AddRequirementCommand();
        Command command = new DeletePageCommand();

        //接头人接收到命令
        xiaoSan.setCommand(command);

        //接头人执行命令
        xiaoSan.action();

    }
```

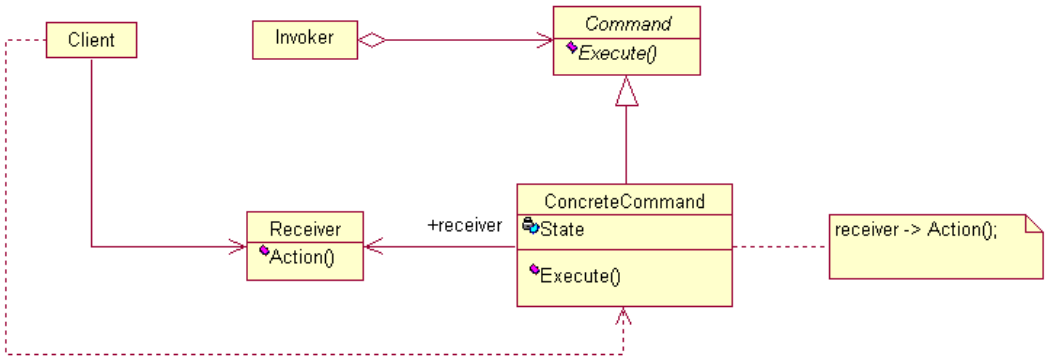
```
}
```

运行结果如下：

```
-------------客户要求删除一个页面-----------------
找到美工组...
客户要求删除一项需求...
客户要求需求变更计划...
```

看到上面打黄色的代码来吗？就修改了这么多，就完成了一个命令的，是不是很简单，而且客户也不用知道到底要谁来修改，这个他不需要知道的，高内聚的要求体现出来了，这就是命令模式。

命令模式的通用类图如下：



在这个类图中，我们看到三个角色：

Receiver 角色：这个就是干活的角色，命令传递到这里是应该被执行的，具体到上面我们的例子中就是 Group 的三个实现类；

Command 角色：就是命令，需要我执行的所有命令都这里声明；

Invoker 角色：调用者，接收到命令，并执行命令，例子中我这里项目经理就是这个角色；

命令模式比较简单，但是在项目中使用是非常频繁的，封装性非常好，因为它把请求方（Invoker）和执行方（Receiver）分开了，扩展性也有很好的保障。但是，命令模式也是有缺点的，你看 Command 的子类没有，那个如果我要写下去的可不是几个，而是几十个，这个类膨胀的非常多，这个就需要大家在项目中自己考虑使用了。

上面的例子我还没有说完，我们想想，客户要求增加一项需求，那是不是页面也增加，同时功能也要增加呢？如果不使用命令模式，客户就需要先找需求组，然后找美工组，然后找代码

组，这个…，你想让客户跳楼呀！使用命令模式后，客户只管发命令模式，你增加一项需求，没问题，我内部调动三个组通力合作，然后反馈你结果，这也正是客户需要的，那这个要怎么修改呢？想想看，很简单的，在 AddRequirementCommand 类的 execute 方法中增加对 PageGroup 和 CodePage 的调用就成了，修改后的代码如下：

```
package com.cbf4life.command;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 增加一项需求
 */
public class AddRequirementCommand extends Command {
    //执行增加一项需求的命令
    public void execute() {
        //找到需求组
        super.rg.find();

        //增加一份需求
        super.rg.add();

        //页面也要增加
        super.pg.add();

        //功能也要增加
        super.cg.add();

        //给出计划
        super.rg.plan();
    }

}
```

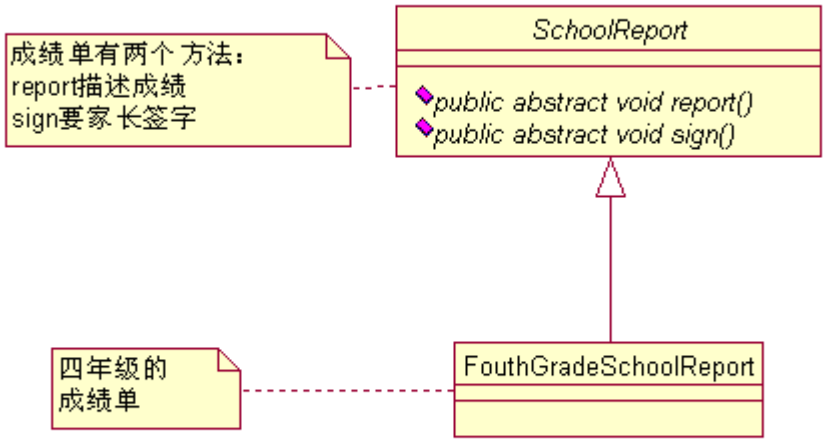
看看，是不是就解决问题了？命令模式做了一层非常好的封装。那还有一个问题需要大家考虑：客户发出命令，那要是撤回怎么办？就类似你使用 Ctl+Z 组合键（undo 功能），发出一个命令，在没有执行或执行后撤回（执行后撤回是状态变更）该怎么实现呢?想想看，简单，非常简单，undo 也是一个命令嘛！

13.装饰模式【Decorator Pattern】

Ladies and gentlemen,May I get your attention,Please?,Now I’m going to talk about decorator pattern.装饰模式在中国使用的那实在是多，中国的文化是中庸文化，说话或做事情都不能太直接，需要有技巧的，比如说话吧，你要批评一个人，你不能一上来就说你这个做的不对，那个做的不对，你要先肯定他的成绩，表扬一下优点，然后再指出瑕疵，指出错误的地方，最后再来个激励，你修改了这些缺点后有那些好处，比如你能带更多的小兵，到个小头目等等，否则你一上来就是一顿批评，你瞅瞅看，肯定是不服气，顶撞甚至是直接“此处不养爷，自有养爷处”开溜哇。这是说话，那做事情也有很多，在山寨产品流行之前，假货很是比较盛行的，我在 2002 年买了个手机，当时老板吹的是天花乱坠，承诺这个手机是最新的，我看着也像，壳子是崭新的，包装是崭新的，没有任何瑕疵，就是比正品便宜了一大截，然后我买了，缺钱哪，用来 3 个月，坏了，一送修，检查，说这是个新壳装旧机，我晕！拿一个旧手机的线路板，找个新的外壳、屏幕、包装就成了新手机，装饰模式害人不浅呀！

我们不说不开心的事情，今天举一个什么例子呢？就说说我上小学的的糗事吧。我上小学的时候学习成绩非常的差，班级上 40 多个同学，我基本上都是在排名 45 名以后，按照老师给我的定义就是“不是读书的料”，但是我老爸管的很严格，明知道我不是这块料，还是往赶鸭子上架，每次考试完毕我都是战战兢兢的，“竹笋炒肉”是肯定少不了的，能少点就少点吧，肉可是自己的呀。四年级期末考试考完，学校出来个很损的招儿（这招儿现在很流行的），打印出成绩单，要家长签字，然后才能上五年级，我那个恐惧呀，不过也就是几秒钟的时间，玩起来什么都忘记了。

我们先看看这个成绩单的类图：



成绩单的抽象类，然后有一个四年级的成绩单实现类，先看抽象类：

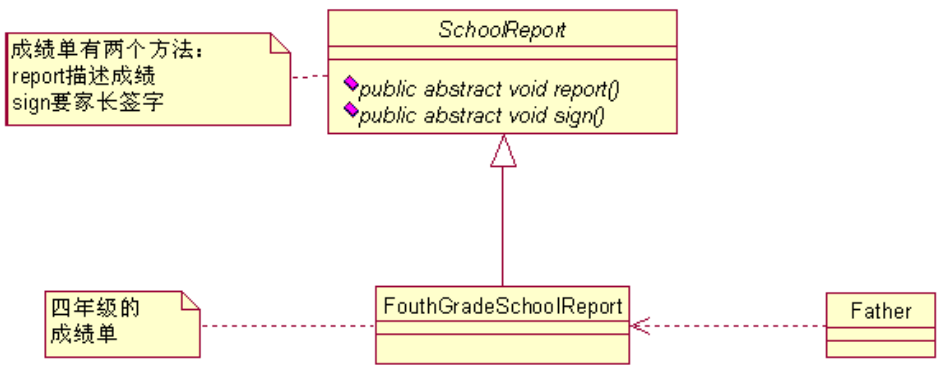
```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 成绩单的抽象类
 */
public abstract class SchoolReport {

    //成绩单的主要展示的就是你的成绩情况
    public abstract void report();

    //成绩单要家长签字，这个是最要命的
    public abstract void sign();
}
```

然后看我们的实现类 FouthGradSchoolReport：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
```

```
 * 四年级的成绩单，这个是我们学校第一次实施，以前没有干过
 * 这种“缺德”事。
 */
public class FouthGradeSchoolReport extends SchoolReport {

    //我的成绩单
    public void report() {
        //成绩单的格式是这个样子的
        System.out.println("尊敬的XXX家长:");
        System.out.println("  ......");
        System.out.println("  语文 62  数学65 体育 98  自然  63");
        System.out.println("  .......");
        System.out.println("              家长签名：       ");
    }

    //家长签名
    public void sign(String name) {
        System.out.println("家长签名为: "+name);
    }

}
```

成绩单出来，你别看什么 62，65 之类的成绩，你要知道在小学低于 90 分基本上就是中下等了，唉，爱学习的人太多了！怎么着，那我把这个成绩单给老爸看看？好，我们修改一下类图，成绩单给老爸看：



老爸开始看成绩单，这个成绩单可是最真实的，啥都没有动过，原装，看 Father 类：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 老爸看成绩单了
 */
public class Father {

    public static void main(String[] args) {
        //成绩单拿过来
        SchoolReport sr = new FouthGradeSchoolReport();

        //看成绩单
        sr.report();

        //签名？休想！
    }
}
```

运行结果如下：

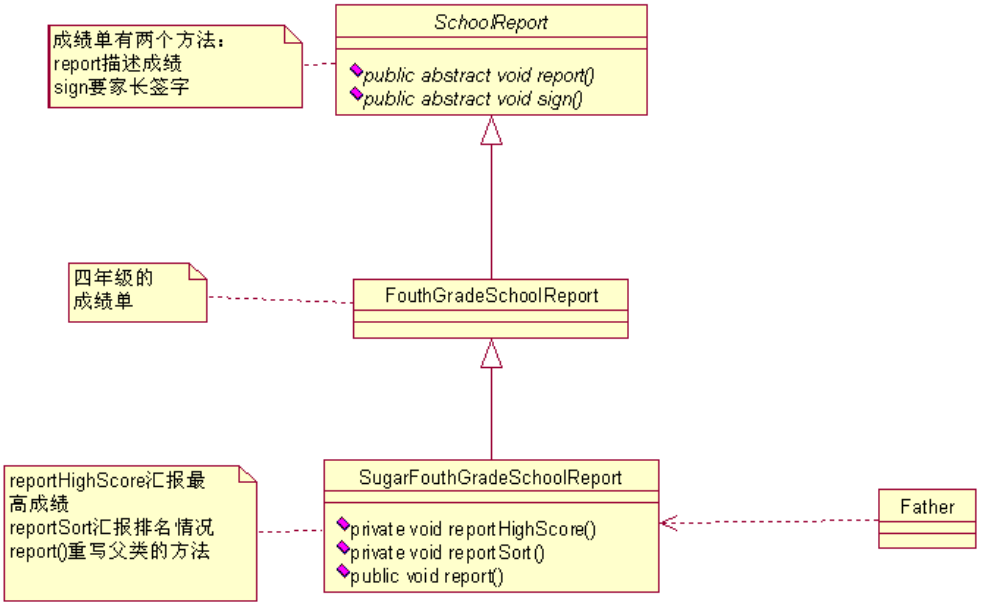
```
尊敬的XXX家长:
  ......
  语文 62  数学65 体育 98  自然  63
  .......
              家长签名:
```

就这成绩还要我签字？！老爸就开始找笤帚，我的屁股已经做好了准备，肌肉要绷紧，要不那个太疼了！哈哈，幸运的是，这个不是当时的真实情况，我没有直接把成绩单交给老爸，而是在交给他之前做了点技术工作，我要把成绩单封装一下，封装分类两步走：

第一步:跟老爸说各个科目的最高分，语文最高是 75，数学是 78，自然是 80，然老爸觉的我成绩与最高分数相差不多，这个是实情，但是不知道是什么原因，反正期末考试都考的不怎么样，但是基本上都集中在 70 分以上，我这 60 多分基本上还是垫底的角色；

第二步：在老爸看成绩单后，告诉他我是排名第 38 名，全班，这个也是实情，为啥呢？有将近十个同学退学了！这个情况我是不说的。不知道是不是当时第一次发成绩单，学校没有考虑清楚，没有写上总共有多少同学，排名第几名等等，反正是被我钻了个空子。

那修饰是说完了，我们看看类图如何修改：



我想这是你最容易想到的类图，通过直接增加了一个子类，重写 report 方法，很容易的解决了这个问题，是不是这样？是的，确实是，确实是一个很好的办法，我们来看具体的实现：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 对这个成绩单进行美化
 * Sugar这个词太好了，名词是糖的意思，动词就是美化
 * 给你颗糖你还不美去
 */
public class SugarFouthGradeSchoolReport extends
FouthGradeSchoolReport {

    //首先要定义你要美化的方法，先给老爸说学校最高成绩
    private void reportHighScore(){
        System.out.println("这次考试语文最高是75，数学是78，自然是80");
    }

    //在老爸看完毕成绩单后，我再汇报学校的排名情况
    private void reportSort(){
        System.out.println("我是排名第38名...");
    }
```

```
    //由于汇报的内容已经发生变更，那所以要重写父类
    @Override
    public void report(){
        this.reportHighScore();  //先说最高成绩
        super.report();  //然后老爸看成绩单
        this.reportSort(); //然后告诉老爸学习学校排名
    }

}
```

然后 Father 类稍做修改就可以看到美化后的成绩单，看代码如下：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 老爸看成绩单了
 */
public class Father {

    public static void main(String[] args) {
        //美化过的成绩单拿过来
        SchoolReport sr= new SugarFouthGradeSchoolReport();

        //看成绩单
        sr.report();

        //然后老爸，一看，很开心，就签名了
        sr.sign("老三");  //我叫小三，老爸当然叫老三
    }
}
```

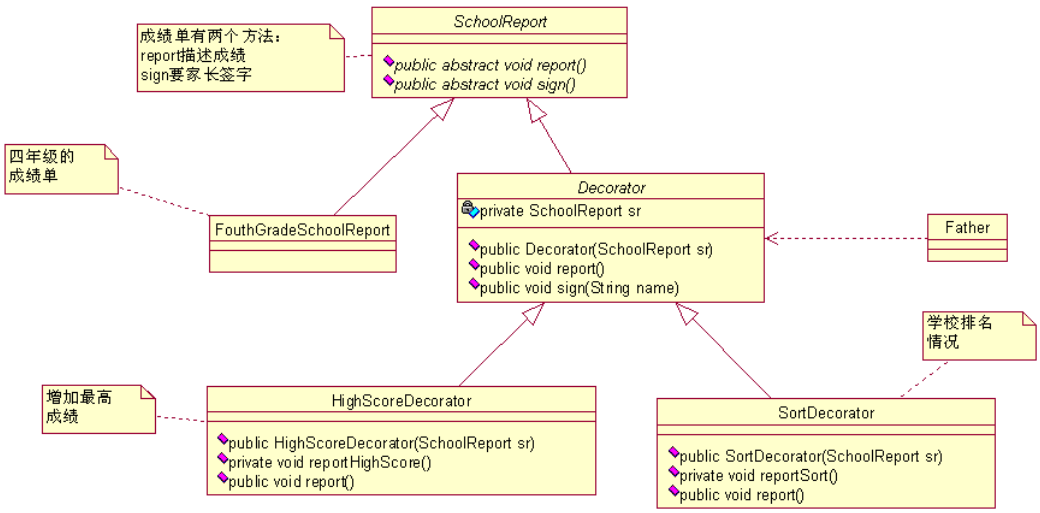
运行结果如下：

```
这次考试语文最高是75，数学是78，自然是80
尊敬的XXX家长:
  ......
  语文 62  数学65 体育 98  自然  63
  .......
              家长签名:
```

```
我是排名第38名...
家长签名为：老三
```

通过继承确实能够解决这个问题，老爸看成绩单很开心，然后就给签字了，但是现实的情况很复杂的，可能老爸听我汇报最高成绩后，就直接乐开花了，直接签名了，后面的排名就没必要了，或者老爸要先听排名情况，那怎么办？继续扩展类？你能扩展多少个类？这还是一个比较简单的场景，一旦需要装饰的条件非常的多，比如 20 个，你还通过继承来解决，你想想的子类有多少个？你是不是马上就要崩溃了！

好，你也看到通过继承情况确实出现了问题，类爆炸，类的数量激增，光写这些类不累死你才怪，而且还要想想以后维护怎么办，谁愿意接收这么一大堆类的维护哪？并且在面向对象的设计中，如果超过 2 层继承，你就应该想想是不是出设计问题了，是不是应该重新找一条道了，这是经验值，不是什么绝对的，继承层次越多你以后的维护成本越多，问题这么多，那怎么办？好办，装饰模式出场来解决这些问题，我们先来看类图：



增加一个抽象类和两个实现类，其中 Decorator 的作用是封装 SchoolReport 类，看源代码：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 装饰类，我要把我的成绩单装饰一下
```

```
 */
public abstract class Decorator extends SchoolReport{

    //首先我要知道是那个成绩单
    private SchoolReport sr;

    //构造函数，传递成绩单过来
    public Decorator(SchoolReport sr){
        this.sr = sr;
    }

    //成绩单还是要被看到的
    public void report(){
        this.sr.report();
    }

    //看完毕还是要签名的
    public void sign(String name){
        this.sr.sign(name);
    }

}
```

Decorator 抽象类的目的很简单，就是要让子类来对封装 SchoolReport 的子类，怎么封装？重写 report 方法！先看 HighScoreDecorator 实现类：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要把我学校的最高成绩告诉老爸
 */
public class HighScoreDecorator extends Decorator {

    //构造函数
    public HighScoreDecorator(SchoolReport sr){
        super(sr);
    }

    //我要汇报最高成绩
    private void reportHighScore(){
```

```
        System.out.println("这次考试语文最高是75，数学是78，自然是80");
    }

    //最高成绩我要做老爸看成绩单前告诉他，否则等他一看，就抡起笤帚有揍我，我那还有机会说呀
    @Override
    public void report(){
        this.reportHighScore();
        super.report();
    }
}
```

重写了 report 方法，先调用具体装饰类的装饰方法 reportHighScore，然后再调用具体构件的方法，我们再来看怎么回报学校排序情况 SortDecorator 代码：

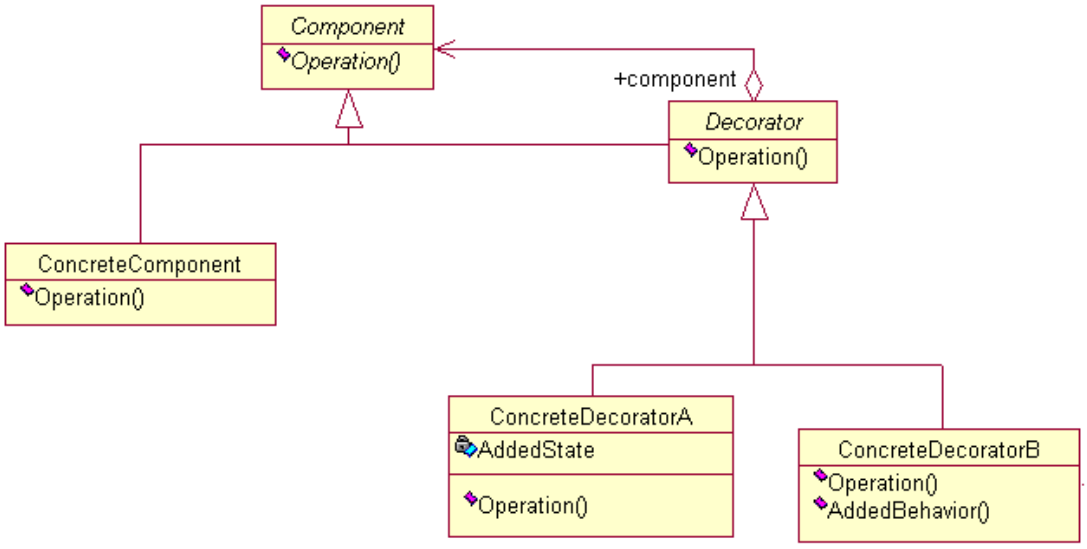
```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 学校排名的情况汇报
 */
public class SortDecorator extends Decorator {

    //构造函数
    public SortDecorator(SchoolReport sr){
        super(sr);
    }

    //告诉老爸学校的排名情况
    private void reportSort(){
        System.out.println("我是排名第38名...");
    }

    //老爸看完成绩单后再告诉他，加强作用
    @Override
    public void report(){
        super.report();
        this.reportSort();
    }
}
```

然后看看我老爸怎么看成绩单的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 老爸看成绩单了
 */
public class Father {

    public static void main(String[] args) {
        //成绩单拿过来
        SchoolReport sr;
        sr = new FouthGradeSchoolReport();  //原装的成绩单

        //加 了最高分说明的成绩单
        sr = new HighScoreDecorator(sr);

        //又加了成绩排名的说明
        sr = new SortDecorator(sr);

        //看成绩单
        sr.report();

        //然后老爸，一看，很开心，就签名了
        sr.sign("老三");  //我叫小三，老爸当然叫老三
    }
}
```

老爸一看成绩单，听我这么一说，非常开心，儿子有进步呀，从 40 多名进步到 30 多名，进步很大，躲过了一顿海扁。

这就是装饰模式，装饰模式的通用类图如下：



看类图，Component 是一个接口或者是抽象类，就是定义我们最核心的对象，也就是最原始的对象，比如上面的成绩单，记住在装饰模式中，必然有一个被提取出来最核心、最原始、最基本的接口或抽象类，就是 Component。

ConcreteComponent 这个事最核心、最原始、最基本的接口或抽象类的实现，你要装饰的就是这个东东。

Decorator 一般是一个抽象类，做什么用呢？实现接口或者抽象方法，它里面可不一定有抽象的方法呀，在它的属性里必然有一个 private 变量指向 Component。

ConcreteDecoratorA 和 ConcreteDecoratorB 是两个具体的装饰类，你要把你最核心的、最原始的、最基本的东西装饰城啥东西，上面的例子就是把一个比较平庸的成绩单装饰成家长认可的成绩单。

装饰模式是对继承的有力补充，你要知道继承可不是万能的，继承可以解决实际的问题，但是在项目中你要考虑诸如易维护、易扩展、易复用等，而且在一些情况下（比如上面那个成绩单例子）你要是用继承就会增加很多了类，而且灵活性非常的差，那当然维护也不容易了，也就是说装饰模式可以替代继承，解决我们类膨胀的问题，你要知道继承是静态的给类增加功能，而装饰模式则是动态的给增加功能，你看上面的那个例子，我不想要 SortDecorator 这层的封装也很简单呀，直接在 Father 中去掉就可以了，如果你用继承就必须修改程序。

装饰模式还有一个非常好的优点，扩展性非常好，在一个项目中，你会有非常多因素考虑不到，特别是业务的变更，时不时的冒出一个需求，特别是提出一个令项目大量延迟的需求时候，那种心情是…，真想骂娘！装饰模式可以给我们很好的帮助，通过装饰模式重新封

装一个类，而不是通过继承来完成，简单点说，三个继承关系 Father,Son,GrandSon 三个类，我要再 Son 类上增强一些功能怎么办？我想你会坚决的顶回去！不允许，对了，为什么呢？你增强的功能是修改 Son 类中的方法吗？增加方法吗 ？对 GrandSon 的影响哪？特别是 GrandSon 有多个的情况，你怎么办？这个评估的工作量就是够你受的，所以这个是不允许的，那还是要解决问题的呀，怎么办？通过建立 SonDecorator 类来修饰 Son，等于说是创建了一个新的类，这个对原有程序没有变更，通过扩充很好的完成了这次变更。

14.组合模式【Composite Pattern】

计划完成日期：2009 年 6 月 14 日

15.观察者模式【Observer Pattern】

计划完成时间：2009 年 6 月 21 日

16.访问者模式【Visitor Pattern】

计划完成日期：2009 年 6 月 28 日

17.状态模式【State Pattern】

计划完成日期：2009 年 7 月 5 日

18.责任链模式【Chain of Responsibility Pattern】

计划完成日期：2009 年 7 月 5 日

19.原型模式【Prototype Pattern】

计划完成日期：2009 年 7 月 12 日

20.中介者模式【Mediator Pattern】

计划完成日期：2009 年 7 月 19 日

21.迭代器模式【Iterator Pattern】

计划完成日期：2009 年 7 月 19 日

22.解释器模式【Interpreter Pattern】

计划完成日期：2009 年 7 月 19 日

23.亨元模式【Flyweight Pattern】

计划完成日期：2009 年 7 月 26 日

24.备忘录模式【Memento Pattern】

计划完成日期：2009 年 7 月 26 日

25.模式大 PK

计划完成时间：2009 年 8 月 31 日

26.混编模式讲解

计划完成时间：2009 年 8 月 31 日

27. 更新记录：

2009 年 4 月 22 日　完成策略模式和代理模式的编写；并发布到论坛上；

2009 年 4 月 24 日　完成单例模式和多例模式的编写；

2009 年 4 月 29 日　完成工厂方法编写；

2009 年 5 月 2 日　完成抽象工厂模式的编写；

2009 年 5 月 2 日　完成了门面模式的编写；增加封面、编写计划、后序以及部分类图

2009 年 5 月 10 日　完成适配器模式的编写；

2009 年 5 月 17 日　完成模板方法模式和建造者模式；

2009 年 5 月 24 日　完成桥梁模式；

2009 年 5 月 30 日　完成命令模式和装饰模式；

28.相关说明

软件环境：JDK 1.5 MyEclipse 7.0

类图设计软件：Rational Rose

博客地址：

http://hi.baidu.com/cbf4life/blog/item/e1ff58f849a8ea51242df219.html

本文下载地址：

http:// cbf4life.66ip.com/设计模式.pdf

源代码及类图下载地址：

http:// cbf4life.66ip.com /source-code.rar

论坛：

http://www.javaeye.com/topic/372233

29.后序

首先，要说的是非常感谢大家的支持，让我有勇气续写下去，我为什么要写这本书？真的不是炫耀自己，从 2000 年毕业，到现在 9 年的光影，在 IT 技术这块基本上都做过，从程序员起步，高级程序员，系统分析师，项目经理，测试经理，质量经理，项目维护，IT 老师，呵呵，接触的语言也比较多，什么 C 了汇编了 Ruby 了这些边角的东西都做过项目，更别说 Java 了，Java 项目做了 6 年，金融交易类的，OA 管理类的，政府类的等都做过，这几年基本上都是项目经理和技术经理一块兼职，搞的很累，带过最少 3 个人的团队，也带过 40 多人的研发团队，可以说自己比较失败，9 年了，始终还是个做技术的，不过技术还是我最热衷的，我喜欢看简单清晰明了易懂的代码，一看代码一团糟，那这人肯定不怎么样。

9 年了，整天忙的跟疯子一样，不知道这样忙下去什么时候是个头，而且自己的年龄也不小了，不能跟着这批 80 后甚至是 90 后一起疯狂加班了，并且最近在工作上也发生了一些事情，让我根本看不清楚未来的路，单位是好，福利应该是同行业的中上等吧，可我们这小兵的路在何方？40 岁了还写代码，中国可行吗？跟 90 后甚至是 00 后一起加班？这不是我想要的工作模式，很失落，所以想找个精神激励，于是就想把自己的这几年的工作经验整理成一本书，让大家尽量能看懂的书，大家的每一个回帖、邮件、评价都给了我莫大的鼓励！由于是第一次写书，可能确实有部分考虑不周，请大家指正。这本书能出版更好，不能出版也无所谓，我只是想找个精神鼓励。

大家一看目录可能就发怵了，怎么是 24 个模式呀，一般书上都是 23 个模式，呵呵，确实是，我增加了多例模式，这个一般都是融合在单例模式中讲的，我是拆出来了。

设计模式这块，我是从 04 年开始系统学习的，《J2EE 核心模式》、《Java 与模式》、《Head First 设计模式》等几本经典的书都拜读过，也确实学到了不少东西，但是始终觉的这些书有缺陷，前两本都是一副孔老夫子假正经的摸样，板着脸一本正经的在讲技术，真的是研究，实话说我是忍着看下去的，技术是为了使用服务的，你说那么多用不到的优缺点、场景干什么，干嘛不说个大家接触到的看的懂例子？！《Head First 设计模式》实话说，我不喜欢，西式的幽默不合我的胃口，看过一遍就不想看第二遍了。

这个序是我想到那里就写到那里，没有个重点，也没个理个头绪出来，大家将就吧。

2009 年 5 月 2 日