笔者在《如何成为 Android 高手》一文和视频中曾提出，成为一名真正的 Android 高手必须掌握和遵循的一些准则：

1，学会懒惰

2，精通 Android 体系架构、MVC、常见的设计模式、控制反转（IoC）

3，编写可重用、可扩展、可维护、灵活性高的代码

4，高效的编写高效的代码

5，学会至少一门服务器端开发技术

上面的几条准则非常明确的指出：熟练掌握设计模式以及设计模式相关的内容是在成为 Android 高手的道路上必修的课程。

Android 号称是首个为移动终端打造的真正开放和完整的移动软件。作为一个气象万千的平台，设计原则、设计模式、IoC 以及相关思想的应用是是导致 Android 之所以能够取得今日的 Android 的成功的核心因素之一。

为了让国内的 Android 爱好者们从浩如烟海的设计模式相关的系列书籍和文档中解脱出来，本着一种方便国内 Android 开发者更好、更快、更轻松的对 Android 的设计原则、设计模式、IoC(控制反转)理解和掌握 的心态，国士工作室成员在百忙之中编写了《Android 之大话设计模式》一书，该书涵盖了 6 中设计原则、主要的设计模式、UML 建模语言和 StarUML 建模工具的使用等，主要内容如下：

- 前言(已发布)
- 针对接口编程---问世间情为何物 直教人生死相许(已发布)
- 单一职责原则 乔峰 VS 慕容复(已发布)
- 开放封闭原则 孙悟空任弼马温一职(已发布)
- 里氏代换原则 法海捉拿白蛇新解(已发布)
- 迪米特法则 慈禧太后为何不和陌生人说话(已发布)
- 合成聚合复用原则 刘邦 VS 韩信（已发布）
- 简单工厂模式 一见钟情的代价（已发布）
- 工厂方法法模式 让麦当劳的汉堡适合不同 MM 的不同口味（已发布）
- 抽象工厂模式 MM 的生日
- 单例模式 你是我的唯一
- 原型模式 肉麻情话
- 建造者模式 让我们同居吧！
- 装饰模式 见 MM 的家长
- 外观模式 MM 也迷恋炒股？
- 享元模式 短信可以这样发
- 适配器模式 笔记本电脑的适配器
- 代理模式 QQ 聊天机器人
- 桥接模式 最重要的是要有一颗让 MM 快乐 的心
- 组合模式 MM 的生日礼物
- 模板方法模式 人的一生应该这样度过
- 观察者模式 GG 在 MM 身边有两个妹妹
- 状态模式 在一天的不同时间要给 MM 发不通的短信
- 策略模式 帮助 MM 选择商场打折策略
- 职责链模式 帮助 MM 选择商场打折策略
- 统一建模语言 UML 简介和 StarUML 使用

本着开放、分享、交流的原则，现免费开放该书，希望能够为推动国内 Android 的发展贡献力量。

前言

Alexander 在《建筑的永恒之道》中给出的模式的经典定义是：每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心。通过这种方式，你可以无数次地使用那些已有的解决方案，无需在重复相同的工作。

一般意义上讲，模式包括架构模式、设计模式、编码模式或者语言惯例。

本书的关注核心在于设计原则和设计模式。

何谓设计模式？设计模式是在某种情境下，针对某种问题的某种典型、通用的解决方案。这里的关键词如下：

情境：是在特定情境下反复出现的情况，这要要求使用模式必须分析清楚事实。

问题：问题一般就是你要实现目标或要解决的目标。

解决方案：典型的、通用的解决方案。能够引人深思和举一反三的解决方案。

设计模式是被发现的，不是被创造的。
设计模式被发现了、被发现着、将被发现！

设计模式来自哪里？当然是来自人类和宇宙相互的作用。是人类长期为追求更加美好生活经验和智慧的结晶。

设计模式来自人类与宇宙的相互作用是不是太抽象、是人类追求大脑与宇宙相互作用的永恒之道的过程是不是让你想晕死呢？！

Take it easy!

其实模式理论的基本思想起源于中国。每一个炎黄子孙的血液中都留着模式的血液。模式充斥于中国历史和现实生活中各个方面，无孔不入！下面我们简要的分享一下。

《孙子兵法》到处都是模式，当然也包括设计原则。“置于死地而后生”就是其中的一种一种模式，三十六计中的“美人计”、“欲擒故纵”都是模式。现在这些模式都成了古今中外各行各业研究学习的对象。听说美军攻打伊拉克的时候就使用了《孙子兵法》中的很多战争模式。

商业中是最讲究模式又是最不讲究模式的。每次我们谈到创业或者某种商业现象时我们都会重点思考商业模式的问题，当然也包括产品的生命周期的问题。例如现在互联网的以广告为主的商业模式、以服务为主的移动增值模式、电子商务模式等。不过这又是一个不讲模式的领域，昨天就有一位哥们壮志豪情的对我我说：我要和你比谁跑的快的，不过我不和你比技术，因为和你比技术的话，正如大多数人一样，这辈子都 没赢你的机会，我要在商业中和你比谁更成功、谁更有钱！但是我只能说多多指教啊。不知道在哥们是否已经精通了一切模式，到了可以乱来的地步呢？

中国古代的法律中的很多部分都是遵循案例法的，无独有偶，现在欧洲国家一般也采用案例法的形式。所谓案例法基本意思就是说在解决现实问题前先研究一起的案例，然后举一反三，根据现实情况举一反，快速而准确的解决现实问题。

医学中也是如此，例如扁鹊的"望闻问切"。望，指观气色；闻，指听声息；问；指询问症状；切；指摸脉象。合称四诊(four diagnostic methods)。"望闻问切"是一般的中医的看病的一半步骤。这是不是很像模板模式呢？

现实世界中每个国家和地区都在追求自己的最佳发展模式，喔，模式！又是模式！中国一直在探求中国特色的自己特色的发展道路。到时中国要收回香港主权的时候，小平同志就创造性的提出了一国两制的构想，丰富了中国特色模式的道路。

爱情作为一个古老而常青的话题，被人们创建出了无数的模式，不必说卓文君带着司马相如私奔，不必说梁山伯与祝英台的可歌可泣，更不必说现在各种感情剧的的流行，但就说无数的人迷恋牵涉情爱的网络游戏就可见一斑了，是非几乎所有的人都想成为她的救世主。大家更熟悉的就是郎才女貌模式、英雄救美模式等等。而且歌手阿杜为了找到爱情的永恒模式，竟说自己翻破了爱情的秘籍，最后找到了四个字"坚持到底"！不知道这个"坚持到底"是否真的的有用。但是我知道在实现梦想中"坚持到底"的模式是绝对重要的，深度决定高度和广度嘛！只用你能坚持到底，甚至"跪着生"，最后可能也会向马云一样"拿望远镜都找不到对手"。

比尔盖茨曾说，他对员工的控制只会加强，永远的掌控，让员工在图灵面前哭泣吧。而在网络中听到关于微软软件开发过程的介绍中，其中非常重要的就是遵循绝对严格的步骤，这些都是模式。当然微软对创新是比较宽容的，这或许是未来寻求新的更好的模式吧。

既然我们生活中的各个方面都存在着模式，或者说我们生活的一切都是按照模式运行，我们是否可以到处套用模式，模式不就是为了套用的吗？呵呵，当然不能随便套用！必须在合适的情境下、针对合适的问题才能采用。就是要实事求是。如果不考虑使用场合，随便套用看病的模式，那医院就没有存在的必要了，生病了去 Google 一下不就 OK 了吗？可以是事实去不是如此，要懂得辨症用药！三国演义中的马谡随便套用了"置之死地而后生"的模式，签下了军令状，导致几乎全军覆没。结果自己被砍头，如果不是诸葛亮仁慈，唯恐他妻儿也死罪难逃。

不过也感谢这些对模式不合适应用的现象，这产生了反模式。

反模式描述"对产生绝对负面结果的问题的一种常用解决方案" — 旨在通过向人们展示如何避免常见的陷阱来解决问题的另一半。

当然诸多模式也可以联手，例如开发中如日中天的 MVC 模式就是设计模式联合优化的一种模式。当然，准确的说，MVC 不能说是设计模式，因为 MVC 划分的维度过大，MVC 属于架构模式。

在模式中，在一些比较重要的场合或者一个比较重量级的书中，你可能时不时的听到"力"这个关键字，不要被"力"搞糊涂了！按照笔者的理解，"力"是实现目标的约束条件，这种约束条件可能把你带向光明的一面-实现目标，也可能带入痛苦的深渊-远离目标。

其实每一种事物都是在在各种力的相互作用下存在和发展或者毁灭的。正如地球的外貌是各种相互作用的结果一样，软件分析、设计、开发、测试以及维护的各个阶段也都是各种力相互平衡的结果。

这个世界没有偶然。愿力与您同在！

针对接口编程---问世间情为何物 直教人生死相许

应用场景举例：

“十六年后 在此重会；夫妻情深 勿失信约”，悲痛欲绝的杨过跑到断肠崖，看到小龙女亲手留在石壁上的文字，即惊喜又痛苦不欲生：“十六年！为什么要等到十六年？！”。

但是信约已定，痴情的杨过也只能等十六年了。

离开断肠崖后，杨过一边开始了自己的苦苦的等待与思恋，一边寄情练功，当然开始时 候也忘不了吃那疗伤的草药。后来杨过巧遇了千年神雕，和神雕一见如故，从此便开始修炼独孤求败的武功。无事可做，寄情练剑倒也不失为人生的一大快事。“相思无用,唯别而已。别期若有定,千般煎熬又何如?莫道黯然销魂,何处柳暗花明？！”，惊天地泣鬼神的黯然销魂掌就这样诞生了。时 光飞逝，恍惚间快过了十六年。此时，杨过的神功已成，想象着十六年约期就将来临，心中想象着自己一生的挚爱，不免感慨和激动万分！在祭拜过求败他老人家之后，杨过和神雕一起开始去赴那场长达十六年之久的约会。令众生激动和艳羡。

再出江湖的杨过惩奸除恶、帮扶弱小，很快就侠名远播，被人尊称为“神雕侠”。自己心中想象着小龙女过往的一笑一颦，想象着她是怎么度过这十六年的，不禁催生了更加浓烈的相思和相见之情。

千呼万唤，终于，这一天来到！

断肠崖边，佳人芳踪迹未现，过儿万念俱灰，纵身跳下悬崖...

幸好悬崖下面是深渊，杨过并没有死，被水冲到了岸边的杨过苏醒过来后，看到了很多小蜜蜂，他一眼就认出了这是小龙女样的蜜蜂，莫非龙儿就在附近？最后在深潭水下，杨过找到了自己苦苦等待了十六年的挚爱。原来小龙女得知自己无药可救，也纵身跳下断肠崖，十六年之期只不过是为了让杨过不要轻生。但是跳崖后的小龙女并没有死掉，接着就在谷底一个世外桃源的地方慢慢的疗伤，竟然完全康复了。真是有情人终成眷属，有情人终成名人。

定义：

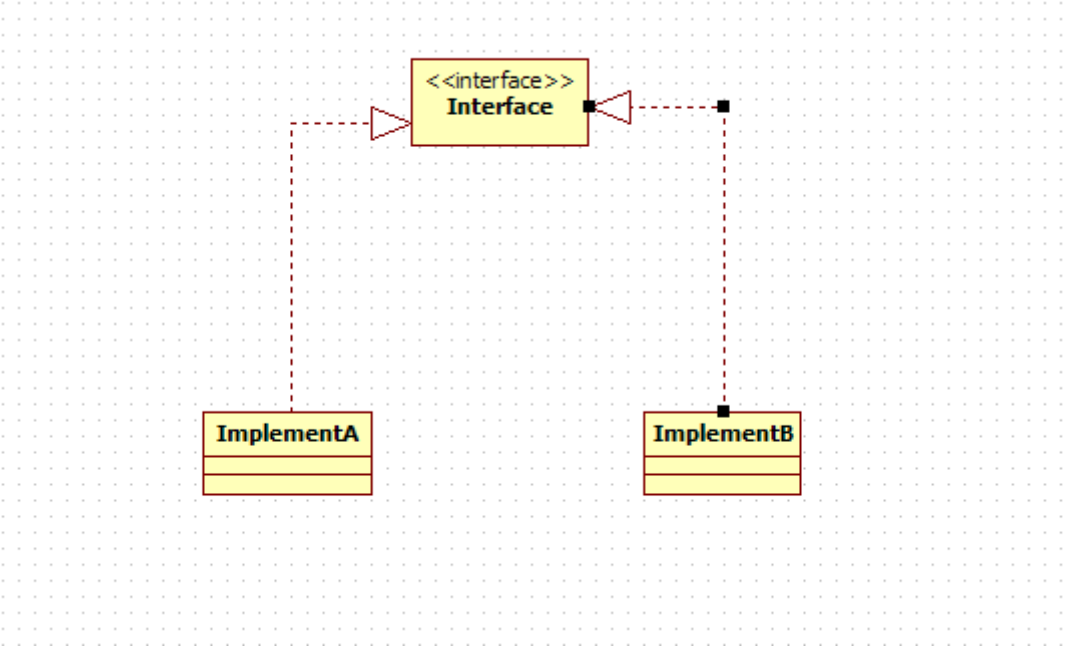
接口是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现，因此 这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为(功能)。

接口是对抽象的抽象。

接口就是标准，就是承诺。

针对接口编程，不要针对具体编程是依赖倒转原则的另外一种表述。

针对接口编程又称为面向接口编程，针对接口编程就是要先设计一系列的接口，把设计和实现分离开，使用时只需引用接口即可，也由于系统各部分的解耦合。如下图所示：

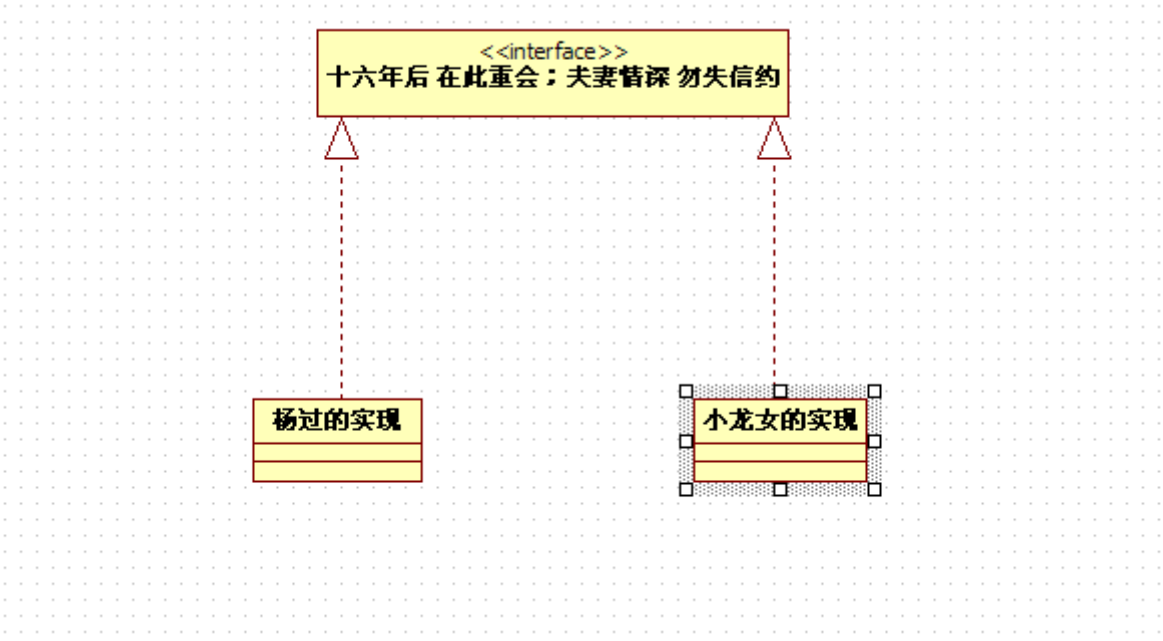


故事分析：

“十六年后 在此重会；夫妻情深 勿失信约”就是针对接口编程的一个绝妙的例子。而且最后还加了“信约”一次。言外之意就是说我们说好了要十六年在此地重逢，我们俩都要遵照此约定。根据上面的故事，小龙女和杨过制定好接口后，就纵身跳下了悬崖，不管自己的过儿了，因为她此时已经不用关心过儿怎么去再十六年后河自己相见，只要十六年后在此地相见即可，也就是说小龙女针对和使用都是接口，至于杨过怎么实现，她此时已经身患绝症而无法顾及了。而杨过看过此约定后，虽然无奈，但也只好照办。具体实现如下：回去吃断肠草调理自己，然后以神雕为伴练习武功，知道练成了黯然销魂掌而神功大成，然后就是在江湖上做侠义之事，然后就是按照信约与十六年后来 到断肠崖边；而小龙女就在谷底慢慢的调养，十六年后身体早已康复，而且越发迷人了。当杨过没有见到小龙女时纵身跳崖，遵照了“问世间情为何物，直教人生死相许”标准。

总结一下:小龙女和杨过定下接口，然后各自针对接口各自独立的做事，最终得以相见。

针对接口编程是未来提高程序的可维护性、可伸缩性和可复用性。如果你在一个类中直接使用另外的一个，这样就把两个类紧密的联系在了一起，以后如果想做出改变就很难了。如果针对接口编程，当业务变化时我们只需要用一个新的类实现接口即可，而客户端依旧可以使用接口引用新的类的，同时也保证了客户端的不变性。这样客户端和实现端互不影响，保持了各自的相对独立性。正如小龙女和杨过的，他们树立了十六年制约后，就不用关心彼此的如何去赴这场约定，只需要按照约定做事就 OK 了。互不影响，自由在在。如下图所示：



Java 代码实现：

新建一个"信约"的接口，这个接口是杨过和小龙女都必须通过自己的方式实现的。代码如下：

```
package com.diermeng.designPattern.dating;
/*
 * 杨过和小龙女定下的约定接口
 */
public interface Dating {
    /*
     * 约定的接口
     */
    public void dating();
}
```

然后分别建立杨过和小龙女的实现类，分别实现上面的接口。代码依次如下：

```
package com.diermeng.designPattern.dating.impl;


import com.diermeng.designPattern.dating.Dating;
/*
 * 杨过对接口的实现
 */
public class Yangguo implements Dating {
    /*
     * 姓名
     */
    String name;


    /*
     * 默认空构造方法
     */
    public Yangguo(){}
    /*
     * 传入 name 参数的构造方法
     */
    public Yangguo(String name) {
        this.name = name;
    }
```

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    /*
     * (non-Javadoc)
     * @see com.diermeng.dating.inter.Dating#dating()
     * 杨过对约定的实现
     */
    public void dating() {
        if(this.getName()!=null){
            System.out.println(this.getName()+" : "+"十六年后 在此重会；夫妻情深 勿失信约");
        }
        else{
            System.out.println("十六年后 在此重会；夫妻情深 勿失信约");
        }
    }
}
```

```
package com.diermeng.designPattern.dating.impl;

import com.diermeng.designPattern.dating.Dating;
/*
 * 小龙女对接口的实现
 */
public class XiaoLongnv implements Dating{
    /*
     * 姓名
     */
    String name;

    /*
```

```
    * 默认空构造方法
    */
   public XiaoLongnv(){}

   /*
    * 传入 name 参数的构造方法
    */
   public XiaoLongnv(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }

   /*
    * (non-Javadoc)
    * @see com.diermeng.dating.inter.Dating#dating()
    * 小龙女对约定的实现
    */
   public void dating() {
      if(this.getName()!=null){
         System.out.println(this.getName()+" : "+"十六年后 在此重会；夫妻情深 勿失信约");
       }
      else{
         System.out.println("十六年后 在此重会；夫妻情深 勿失信约");
       }
   }

}
```

建立一个测试类，代码如下：

```
package com.diermeng.designPattern.dating.client;
```

```
import com.diermeng.designPattern.dating.Dating;
import com.diermeng.designPattern.dating.impl.XiaoLongnv;
import com.diermeng.designPattern.dating.impl.Yangguo;
/*
 * 对杨过和小龙女约定进行测试的客户端
 */
public class DatingTest {

   public static void main(String[] args)
   {
      //分别实例化实例化
      Dating yangguo = new Yangguo("过儿");
      Dating xiaoLongnv = new XiaoLongnv("龙儿");

      //调用各自的方法
      yangguo.dating();
      xiaoLongnv.dating();
   }
}
```

程序运行结果如下：

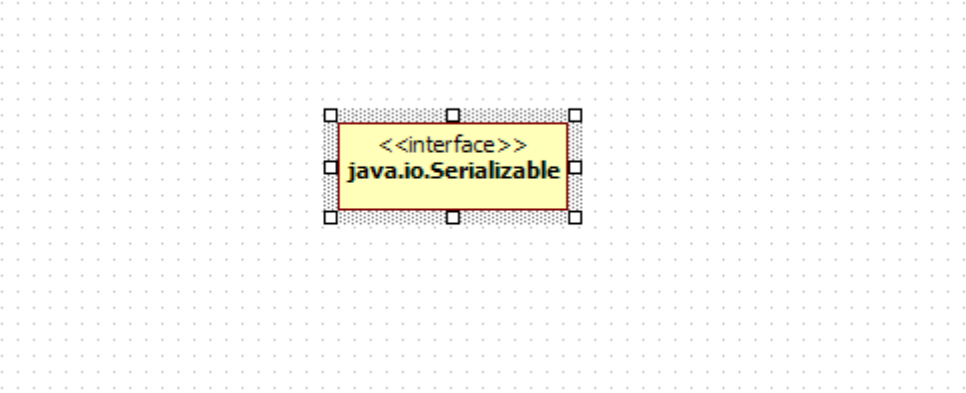
```
过儿 ： 十六年后 在此重会；夫妻情深 勿失信约
龙儿 ： 十六年后 在此重会；夫妻情深 勿失信约
```

已有应用简介：

Java 是相面对象编程的语言，而面向对象编程的核心之一就是要针对针对接口编程、不要针对实现编程，在 Java API 中的标志接口 java.io.Serializable 和 java.rmi.Remote 等就是我们经常遇到的，下面以 java.io.Serializable 为例说明一下，源代码如下：

```
package java.io;
public interface Serializable{}
```

UML 图形如下图所示：



当然在 J2EE 框架的使用中到处都是针对接口编程的身影。例如在 www.babasport.com 中几乎每一处都是针对接口编程的，令人印象非常深刻的一点就是巴巴运动网把对数据库的 CRUD 等基本操作封装在了一个统一接口中，这给以后的代码的编写和数据库的操作带来了极大的方便，当然这里也使用了 Java 5 的泛形技术。有兴趣的读者可以去学习巴巴运动网的源代码。

温馨提示：

许下的承诺就是欠下的债。所以不要轻易做出承诺。

杨过和小龙女为了承诺而付出了十六年的努力。

在软件设计和编码中，如果确立了接口，也就对客户做出了承诺，这种承诺几乎没有改变的机会，时间越长越是如此，因为那是别人对你接口的使用已经遍布世界各地，当然前提是你的借口很出色，这样才能取得很多人的信赖和消费。

不要轻易说："我爱你"，因为这是一生的承诺。

单一职责原则 乔峰 VS 慕容复

应用场景举例：

江湖盛传：北乔峰，南慕容。

少室山下，天下群雄云集。

“你们一起来吧，我萧峰何惧！”，一声豪情和怒吼，乔峰卷入了和慕容复、游坦之、丁春秋一决生死之战。乔峰果然不愧是天下第一豪侠，以一敌三，你来我往，打得不可开交。乔峰使出了降龙十八掌中的“亢龙有悔”，此时慕容复忙往后退，情急之下，使出了自己的绝技“游龙引凤”来化解乔峰如此强烈的进攻，此时慕容复双腿选在了亭子的柱子上，王语嫣心想：“表哥使用游龙引凤自然是不会败的了，可是面对天下众英雄的面，竟然和游坦之、丁春秋等这样的人联手对付大英雄乔峰，这也有些太不公平了，表哥，你在想些什么啊？”。此时被乔峰手下保护的段誉心急如焚，心想，这个不行，唯恐大哥有什么意外，毕竟总是乔峰神功盖世，眼前面对也是江湖上一流的高手，这样下去如何是好？突然，段誉挣脱众人的包括冲了上来，对慕容复使用激将法说：“你们两个打一个算什么英雄好汉，慕容复，有本事来和我单打啊！”。慕容复本来就对段誉一直以来对王语嫣的爱慕之意深表不满，又被他这么一激怒，怒声到：“找死！”，就像段誉扑来。此时段誉以从神仙姐姐那里学来的“凌波微步”的奇功和慕容复纠缠。此时，虚竹正在和收拾丁春秋。而乔峰正在迎战游坦之，游坦之虽然练成了很毒辣的武功，但也绝非是乔峰的对手，数招之内便被乔峰从塔顶扔了下来，结果弄到一个腿部骨折而惨败的下场！段誉白在慕容复的剑下，宁死不屈。慕容复正欲一剑了解了段誉，段正淳喝道：“休伤我儿！”挡住了慕容复的剑。慕容复非常恼怒，数招之内就把慕容复打伤在地。眼看慕容复就要杀死自己父亲，躺在地上的段誉“啊”的一声使出了大理国的绝学“六脉神剑”,几招之内，慕容复惨败，正要进一步进攻时，此时他的梦中情人王语嫣柔声叫到：“段公子手下留情啊”，这段誉，一听到王语嫣的声音，就神魂颠倒、不知所以了，慕容复乘此机会偷袭段誉，乔峰见状，一边喝道：“三弟小心”，一边瞬间出手，粉碎了慕容复的武器，慕容复还没反应过来，就已经被乔峰高高举起在了半空中，乔峰喝道：“我萧峰大好男儿，岂能与你这等小人齐名”，随即把慕容复恨恨的抛了出去！慕容复在天下英雄面前惨败，自觉颜面无常，欲拔剑自刎，被灰衣人阻拦...

定义：

单一职责原则(Single Responsibility Principle):就一个类而言，应该仅有一个引起它变化的原因。换句话说，一个类的功能要单一，只做与它相关的事情。

如果一个完成额外的不太相关的功能或者完成其它类的功能，这就会使得一个引起一个类变化的因素太多，如果类的一处需要修改，其它和它相关连的代码都会受到影响，这就直接导致一旦系统出现了问题就难以调试困境，同时这样也非常不利于维护。

遵循单一职责原则也会给测试带来极大的方便。

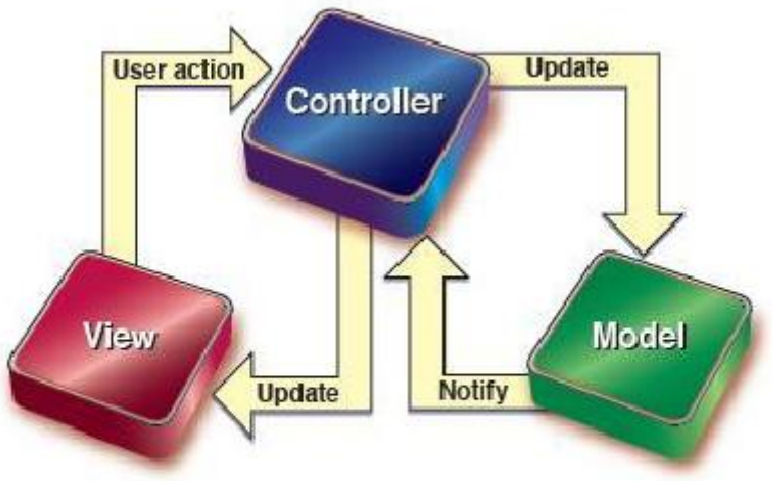
违背单一职责原则会降低类的内聚性、增强类的耦合性。

违背单一职责原则会导致错误呈现几何级数的增长，因为类之间的关联性太强，每一个类都会对其他类有影响，一个类出现错误极可能会导致其他相关联的类出现错误，而且关联类联合起来还有可能产生新的错误。

在软件开发中，人们越来越意识到单一职责原则的重要性，美工只需要负责美工界面，业务层的人员只需写好业务代码，而数据层的人员只需关注数据层的工作即可。这样每个人都以自己专程协同工作，工作效率就得到了大大的提高了。

现在软件开发的经典模式 MVC 模式，也非常好的体现了单一职责原则。MVC(Model-View-Control)就是模型、视图、控制器三层架构模式，其中 M 是指数据模型、V 是指用户界面、C 则是控制器。采用 MVC 模式使得数据和表现相分离，同一个数据层可以有不同的显示层。数据层和显示层的改变互不影响。这就非常有利于提高软件的可维护性和可复用性，同时也方便了软件的管理工作和提高软件开发效率。

如下图所示：



故事分析：

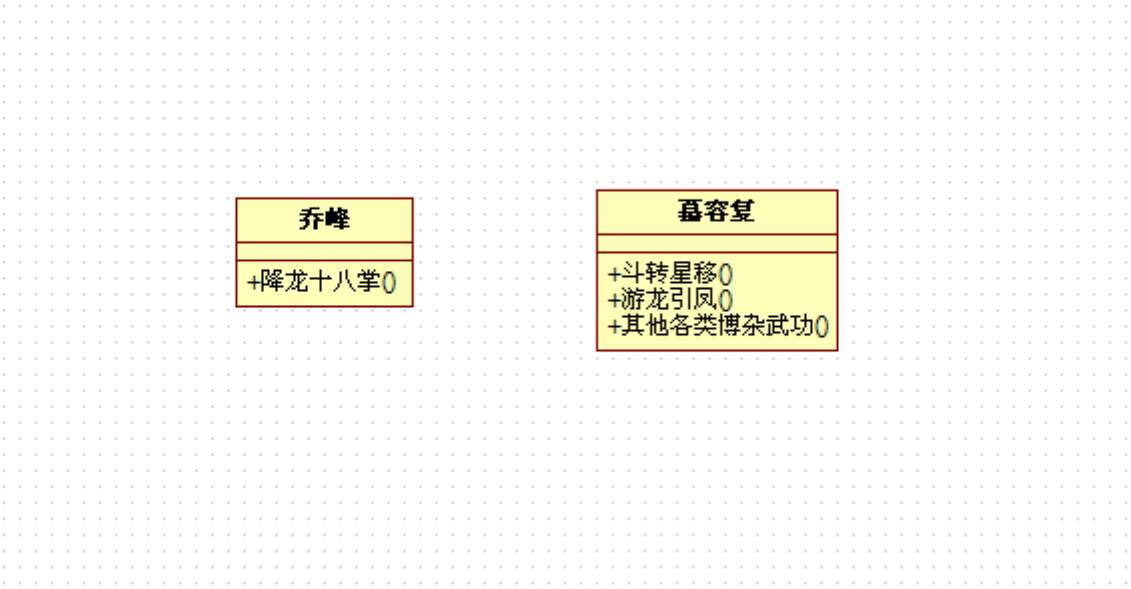
有王语嫣这个武学博士相助，同时集天下几乎所有武学与一身"以彼之道，还施彼身"的慕容复却输了，输给了懂得六脉神剑的段誉， 也输给了用降龙十八掌打遍天下的乔峰。抛开其它的原因不谈，慕容复之所以会输，是因为他虽然懂得很多武功，甚至懂得天下几乎百分之八十以上的武功，但是因为复国心切，确很少做到能够精通；从另外一个角度讲，也是修炼这么多武功害了慕容复。你想啊，一个年纪轻轻的人修炼这么多武功，怎么可能有时间去把每种武功都精通呢，而且还会累的要死。其实，慕容复就像一匹狼一样，面对着一大群样，想把每一个都吃掉！结果是可能基本咬死了每一只羊，却无暇去好好品尝和消化。而段誉和乔峰就不一样了。段誉会六脉神剑。这里我们主要分析一下乔峰。乔峰自幼被少林寺收养，刻苦勤奋，潜心修炼降龙十八掌，直至到达无招胜有招的地步。当然乔峰这样的真英雄、大豪杰自然是实战经验丰富的。就这样，当"以彼之道，还施彼身"的慕容复遇到了用降龙十八掌打遍天下无敌手的乔峰是，慕容复输了。

从设计原则的角度考虑，慕容复输是因为他违背了单一职责原则，他把太多的精力分散于各种武学之中，他迷失在了武学之中。当然，不可否认，慕容复确实是年轻有为的武学天才。但是临阵实战的时候讲究的是你真正的实力，而不是以懂得的武功的多少决定胜负的。修炼太多的武功种类必然让人太累，而且非常容易走火入魔。这就像吃饭一样，吃了太多未必是什么好事，而且几乎肯定不是好事。这是因为，一方面吃了太多，对自己的消化系统和身体都不好，容易发胖等；另外一方面讲，吃的多不一定消化的多，而且很多时候反而削弱消化能力，因为吃的太多带来了太多的负担，影响了消化系统的功能。而乔峰就不一样了，他在打好基础的情况，专注于降龙十八掌，直到能够随心所欲、运用自如。当然乔峰这样的盖世豪侠对武功本质的和武功精髓的理解也是高于慕容复的。而且，乔峰还有一个特定，就是遇强则强、越战越勇。

单一职责原则告诉我们，一个类应该只有一个引起该类变化的原因。这样有利提高类的可维护性和可复用性。如果把乔峰和慕容复比作两个类的话，引起乔峰变化的就是降龙十八掌，而且乔峰把降龙十八掌做到了极致，无论是敌是友，

乔峰都可以用降龙十八掌化解。而慕容复呢，引起他变化的原因就太多了，虽然"以彼之道，还施彼身"几乎令所有江湖人士都敬畏三分，但是因为多而不精，遇到像乔峰这样的高手后，恐怕就无法"以彼之道，还施彼身"了。

这里，我们把乔峰的降龙十八掌看做是乔峰的方法，而慕容复懂的武功也看作慕容复拥有的方法，建模的图形如下：



Java 代码实现：

乔峰的类：

```
package com.diermeng.designPattern.SRP;
/*
 * 乔峰
 */
public class Qiaofeng {
    /*
     * 姓名
     */
    String name;

    /*
     * 无参构造方法
     */
    public Qiaofeng() {}
    /*
     * 带参数的构造方法
     */
    public Qiaofeng(String name) {
        this.name = name;
    }
```

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    /*
     * 乔峰的功夫绝技展现
     */
    public void gongfu()
    {
        if(this.getName()!=null){
            System.out.println("我是"+this.getName()+" 使用降龙十八掌，无论是敌是友我都可以应对自如,这都要感谢我遵循了单一职责原则！");
        }
        else{
            System.out.println("使用降龙十八掌，无论是敌是友我都可以应对自如，这都要感谢我遵循了单一职责原则！");
        }
    }
}
```

慕容复的类图：

```
package com.diermeng.designPattern.SRP;
/*
 * 慕容复
 */
public class MuRongfu {
    /*
     * 姓名
     */
    String name;

    /*
```

```
    * 无参构造方法
    */
   public MuRongfu() {}
   /*
    * 带参数的构造方法
    */
   public MuRongfu(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }

   /*
    * 慕容复的功夫展现
    */
   public void gongfu()
   {
      if(this.getName()!=null){
         System.out.println("我是"+this.getName()+" 我会很多武功，对绝大多数人来说我都可以做到以彼之道还施彼身，但是如果遇到乔峰这样的高手，我就 Over 了，这都是没有遵循单一职责原则惹的祸！");
      }
      else{
         System.out.println(" 我会很多武功，对绝大多数人来说我都可以做到以彼之道还施彼身，但是如果遇到乔峰这样的高手，我就 Over 了，这都是没有遵循单一职责原则惹的祸！");
      }
   }
}
```

建立一个测试类，代码如下：

```
package com.diermeng.designPattern.SRP.client;
```

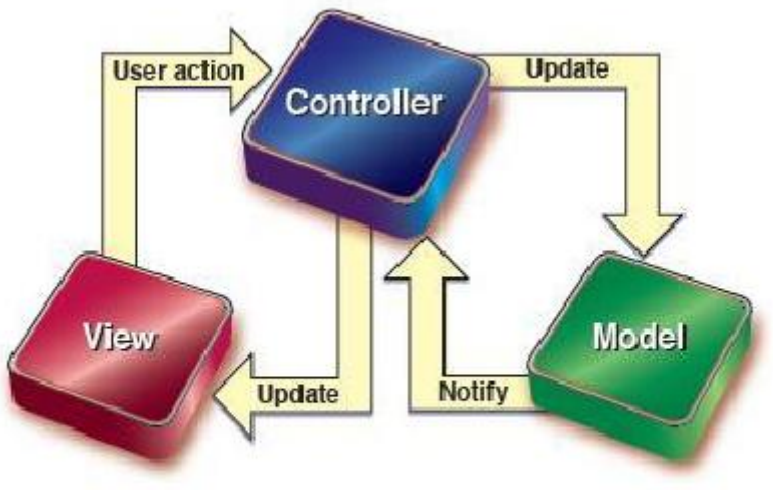
```
import com.diermeng.designPattern.SRP.MuRongfu;
import com.diermeng.designPattern.SRP.Qiaofeng;

public class GongfuTest {

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

        Qiaofeng qiaofeng = new Qiaofeng("乔峰");
        MuRongfu muRongfu = new MuRongfu("慕容复");

        qiaofeng.gongfu();
        muRongfu.gongfu();
    }

}
```

程序运行结果如下：

```
我是乔峰 使用降龙十八掌，无论是敌是友我都可以应对自如,这都要感谢我遵循了单一职责原则!
我是慕容复 我会很多武功，对绝大多数人来说我都可以做到以彼之道还施彼身，但是如果遇到乔峰这样的高手，我就 Over 了，这都是没有遵循单一职责原则惹的祸!
```

已有应用简介：

我们这里已经以 MVC 模式为例来分析单一职责原则的应用。



模型车、视图层、控制层各司其责、相互独立，一个模型可以有多个视图，一个视图可以有多个控制器，同样的一个控制器也可以由多个模型。MVC 基本的处理流程如下： 用户与视图交互，视图接受并反馈用户的动作；视图把用户的请求传给相应的控制器，有控制器决定调用哪个模型，然后由模型调用相应的业务逻辑对用户的请求进行加工处理，如果需要返回数据，模型会把相应的数据返回给控制，由控制器调用相应的视图，最终由视图格式化和渲染返回的数据，以一种对用户尽可能友好的方式展现给用户。

温馨提示：

专注于一件事情，专注于一切事情。

如果这个世界每个人都专注于做一件事情，并把这件事情做到极致，世界会是怎样的一种和谐美好呢？

遵循单一职责原则的乔峰使少林寺的扫地僧也不禁发出感叹：“降龙十八掌果然天下第一！”

树立一个高远的目标，然后拼尽一切力量的去实现这个目标。

愿单一职责原则保佑您！

开放封闭原则 孙悟空任弼马温一职

应用场景举例：

孙悟空从东海龙宫拿到定海神针如意金箍棒后回到花果山，和自己的部下过着自由自在的生活。那只好景不长，因为他在地狱删除了自己和花果山所有猴子的名单，同时又拿走了定海神针，不久便被阎王和龙王告上了天庭。玉帝正要下旨去捉拿妖猴问罪。忙被龙王劝止，龙王说孙悟空神通广大，阎王也深表赞同。玉帝有些迟疑，问众仙该如何是好，太白金星说不如封他一个天宫中的官职去做，这样明为封官，实际上在暗地里确进行压制。玉帝深表赞同。但是要封孙悟空一个什么官好呢？玉帝也一时想不出什么号的职位，于是就宣仙卿百官入朝，共同商讨此事。玉帝问道："众爱卿，现在天庭什么地方可以空缺的职位啊？给那妖猴一个官去做。"众人都说现在天庭的各个职位人说爆满，无任何空缺职位。大家一时之间不知道该如何是好，此事太白金星灵机一动，向玉帝禀报道："禀报玉帝，鉴于天庭个职位人员爆满，不如封他一个弼马温的职位。"玉帝问道："弼马温？是何等职位啊？"太白金星说，弼马温就是用来管理天马的，反正现在天马无人管理，在天庭新设立一个弼马温的职位一方面有利于管理天马，另一方面可以不影响天庭现有的职位，最后还可以安抚妖猴。

之所以会出现上面职位难以安排的情况，这还要从天庭的官吏机制说起，是整个天庭的官吏制度导致了这种情况。在天庭中，法力越大的位置越高，相应的法力越低的位置就越低。而且法力高的由于可以得到各种相应的更多的仙桃啊、太上老君的金丹啊等就变的法力越来越大，也就导致了位置越来越高；另一方面，因为法力越来越大，所以寿命也就越来越长。这导致了什么结果你？导致了终身制！一个职位几乎由相应的仙人一直掌管，永远没有空缺的时候。在天庭是一个萝卜一个坑，官职一旦确定就很难更改！我们说托塔李天王，他就永远是李天王，没人能够取代他的位置！那现在如何安排孙悟空呢？要做到既不影响天庭既有秩序和众仙的利益，又能够安抚孙悟空，那就只有新设立一个职位啦！

玉帝一听太白金星的想法，大悦。立刻派人把孙悟空请了上来。孙悟空早就听说天庭好玩，而且在天庭还有官职，喜出望外，欢欢喜喜的赴任去了。

定义：

开放封闭原则(Open-Closed Principle)：一个软件实体应当对扩展开放，则修改关闭。对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况；对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。

开放封闭原则是所有面向对象原则的核心。

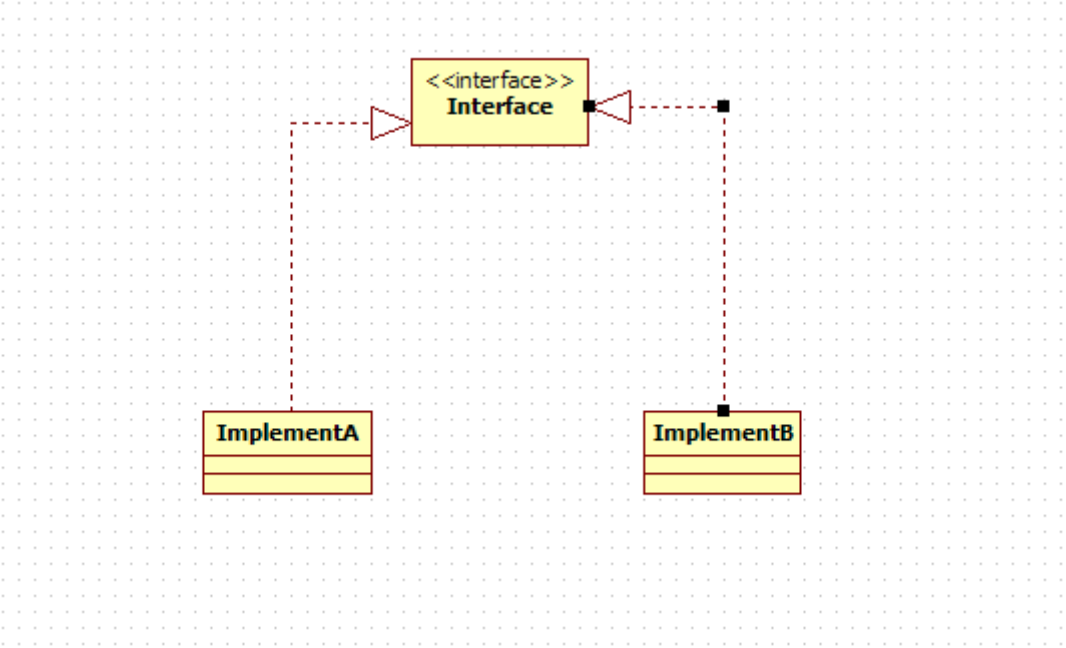
“需求总是在变化”。

“世界没有一个软件是不变的”。

如何做到开放封闭原则呢？答案是：封装变化，依赖接口和抽象类，而不要依赖具体实现类。要针对接口和抽象类编程，不要针对具体实现编程。因为接口和抽象类是稳定的，他们是一种对客户端的一种承诺，是相对不变的。如果以后需要扩展或者开发新的功能，只需要实现或者继承接口或者抽象类即可覆盖或者扩展新的功能，这样做同时也不回影响新的功能。这就很好的做到了对扩展开放、对修改关闭。

实际上讲，绝对封闭的系统式不存在的。无论我们怎么保持封闭，一个系统总会有要变化的地方，“世界上没有一个不边的软件”、“需求总是在改变”。我们要做的不是消灭变化，而是把变化隔离开来，并对其进行封装。我们无法控制变化，但是我们可以预则哪里会发生变法。把要变化的地方抽象起来，这样以后再面临变化的时候我们就可以尽量的扩展，而无须改变以后的代码。

如下图所示：



故事分析：

太白金星献计任孙悟空为弼马温一职就很好的体现了开放封闭原则。

为什么这么说呢？

这还要从天庭的官吏机制说起，在天庭中，法力越大的位置越高，相应的法力越低的位置就越低。而且法力高的由于可以得到各种相应的更多的仙桃啊、太上老君的金丹啊等就变的法力越来越大，也就导致了位置越来越高；另一方面，因为法力越来越大，所以寿命也就越来越长。这导致了什么结果你？导致了终身制！一个职位几乎由相应的仙人一直掌管，永远没有空缺的时候。在天庭是一个萝卜一个坑，官职一旦确定就很难更改！我们说托塔李天王，他就永远是李天王，没人能够取代他的位置！那现在如何安排孙悟空呢？要做到既不影响天庭既有秩序和众仙的利益，又能够安抚孙悟空，那就只有新设立一个职位啦！

总结一下：天庭既有秩序不变，扩展一个弼马温的位置给孙悟空。而且这种扩张不会影响到天庭的其它秩序。这不就是对修改关闭，对扩展开放吗？！

同时，可能不久又出现一个新的“孙悟空第二”，龙王可能又要告到天庭，“需求总是变法的”！这时候只需要按照针对孙悟空同样的思路就可以很好解决此类变化。

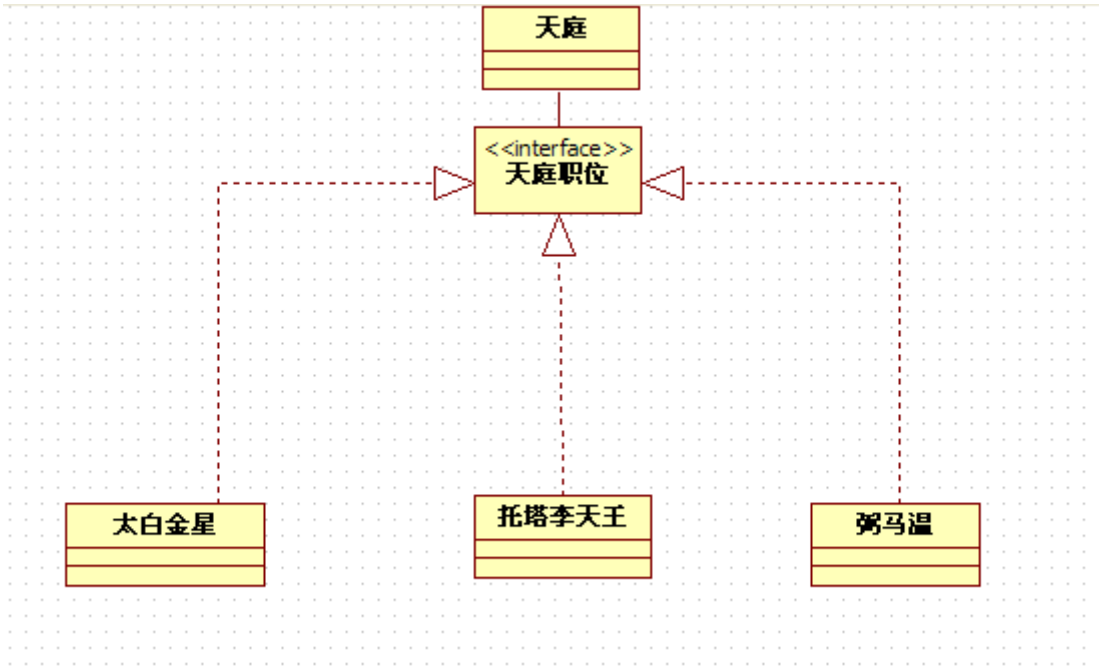
再次重温一下面的话：

“需求总是在变化。”

“世界上没有一个软件是不变的。”

“针对抽象编程，不要针对实现编程。”

如下图所示：



Java 代码实现：

新建一个职位的接口：

```
package com.dieremng.designPattern.OCP;
/*
 * 职位的接口
 */
public interface Position {
    /*
     * 定义职位的职责
     */
    public void duty();
}
```

太白金星、托塔李天王、弼马温分别实现上面的接口。代码依次如下：

```java
package com.dieremng.designPattern.OCP.impl;

import com.dieremng.designPattern.OCP.Position;
/*
 * 太白金星对职位的实现
 */
public class Taibaijinxin implements Position {

    public void duty() {
        System.out.println("这里是太白金星");

    }

}
```

```java
package com.dieremng.designPattern.OCP.impl;

import com.dieremng.designPattern.OCP.Position;
/*
 * 托塔李天王对职位的实现
 */
public class Tuotalitianwang implements Position{

    public void duty() {
        System.out.println("这里是托塔李天王");
    }

}
```

```java
package com.dieremng.designPattern.OCP.impl;
```

```
import com.dieremng.designPattern.OCP.Position;
/*
 * 弼马温对职位的实现
 */
public class Bimawen implements Position {

    public void duty() {
        System.out.println("这里是弼马温");
    }

}
```

建立一个测试类，代码如下：

```
package com.dieremng.designPattern.OCP.client;

import com.dieremng.designPattern.OCP.Position;
import com.dieremng.designPattern.OCP.impl.Bimawen;
import com.dieremng.designPattern.OCP.impl.Taibaijinxin;
import com.dieremng.designPattern.OCP.impl.Tuotalitianwang;

public class PositionTest {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Position taibaijinxin = new Taibaijinxin();
        Position tuotalitianwang = new Tuotalitianwang();
        Position bimawen = new Bimawen();

        taibaijinxin.duty();
        tuotalitianwang.duty();
        bimawen.duty();
    }

}
```

程序运行结果如下：

```
这里是太白金星
这里是托塔李天王
这里是弼马温
```

已有应用简介：

开放封闭原则是所有面向对象原则的核心。

软件的分析、设计、编码、维护等生命周期的各个阶段总是力求做到对修改关闭、对扩展开放。

著名的巴巴运动网生命周期的各个阶段就遵循了开放封闭原则。它把基本的 CRUD 操作做成了一个接口，同时采用了 JDK 5 引入的泛型技术，这样就可以保证以后做基本的添删改查操作时只需要实现该类即可。但由于引入了泛型技术，同时在后台提供了对接口的抽象实现，你甚至不用写一行代码，就可以自如的操作数据库。如果以后又需要扩展的地方，只需要扩展继承扩展自己的特有的操作即可，大大提高了生产效率。

里氏代换原则 法海捉拿白蛇新解

应用场景举例：

《白蛇传》是中国四大民间传说之一，妇孺皆知。

在大多数人的感觉和印象中，白蛇是一个善良痴情、知恩图报、温柔友善、美貌绝伦、冰雪聪明、明辨是非、救苦救难的活菩萨；而法海却是一个仗着自己的法力高强、打着降妖除魔的口号而恶意拆散许仙和白娘子这对恩爱夫妻负面形象。大多说人之所以觉得如此，主要是因为影视中的白蛇善良的无以复加。试想，如果传说和影视中的白蛇不是表现的善良，恐怕人们恨不得早些让法海去把白蛇收服呢。

但是，法海真的那么坏吗？法海真的一无是处吗？

法海不坏。法海是一个非常敬业的人！

法海多年来苦心钻研佛法、潜心修炼是为了降妖除魔、解救世人。法海的原则是凡是妖魔都要被降服。尽管你可能说，妖魔也有善良的啊。首先这个善良本身就已经加入了你主观意志。是你想象中的善良，是你强加的。另外，人和妖的世界时不同的世界，不同的世界做事和行为方式肯定是有极大的碰撞，对一个世界好的事情，对另外一个世界可能就是彻头彻尾的坏了。一句话：两个世界语言沟通有障碍。就按照你说的白蛇善良，你就在你的妖界善良就行了，干嘛非要跑到人间来。为了报恩？报恩非要嫁给许仙才算是报恩吗？报恩的方式有无数种，如果每一个收到许仙恩惠的女子都要嫁给许仙，你白蛇怎么办？人和妖界既然是两个不同的世界，就肯定有很大的不同，就肯定会有很大的摩擦。法海在发誓要收服白蛇前肯定也想过白蛇的善良，但更多的是：你在妖界善良也就罢了，为何来到人间？怎么保证你不危害世人？所以凡事出来的妖魔，法海就必然收服了。

其实法海说到底也只是非常敬业而已。逻辑步骤很简单：法海的任务是是降妖除魔、造福于众生；白蛇是妖；所以要除掉。如此敬业的人，怎能不赢的我们的掌声呢？

定义：

里氏代换原则（Liskov Substitution Principle）是指：一个软件实体如果使用的是基类的话，那么也一定适用于其子类，而且它根本觉察不错使用的是基类对象还是子类对象；反过来的代换这是不成立的，即：如果一个软件实体使用一个类的子类对象，那么它不能够适用于基类对象。

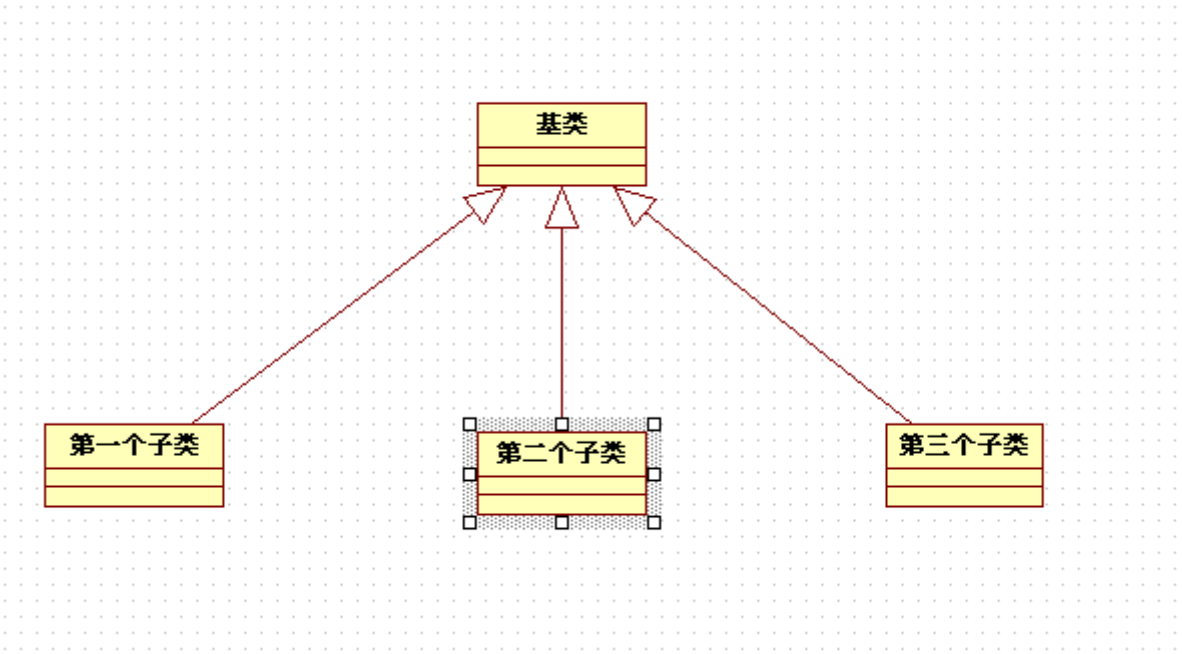
里氏代换原则是由麻省理工学院（MIT）计算机科学实验室的 Liskov 女士，在 1987 年的 OOPSLA 大会上发表的一篇文章《Data Abstraction and Hierarchy》里面提出来的，主要阐述了有关继承的一些原则，也就是什么时候应该使用继承，什么时候不应该使用继承，以及其中的蕴涵的原理。2002 年，软件工程大师 Robert C. Martin，出版了一本《Agile

Software DevelopmentPrinciples Patterns and Practices》，在文中他把里氏代换原则最终简化为一句话：“Subtypes must besubstitutable for their base types”。也就是，子类必须能够替换成它们的基类。

里氏代换原则讲的是基类和子类的关系，只有这种关系存在的时候里氏代换原则才能够成立。

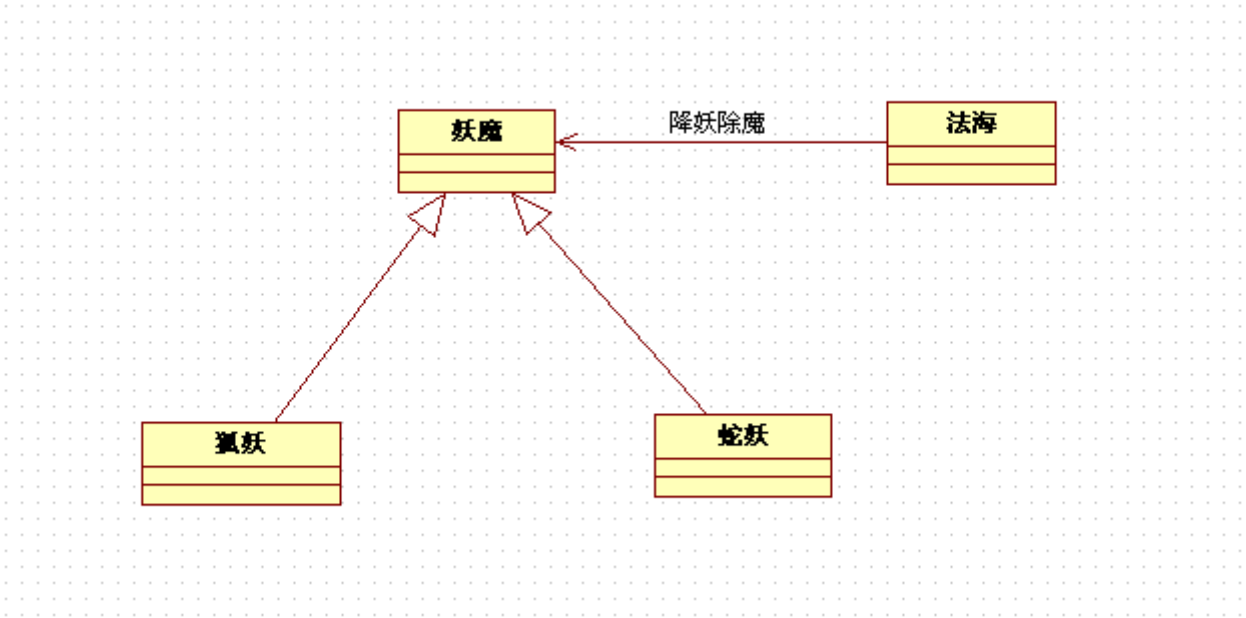
里氏代换原则是实现开放封闭原则的具体规范。这是因为：实现开放封闭原则的关键是进行抽象，而继承关系又是抽象的一种具体实现，这样 LSP 就可以确保基类和子类关系的正确性，进而为实现开放封闭原则服务。

如下图所示：

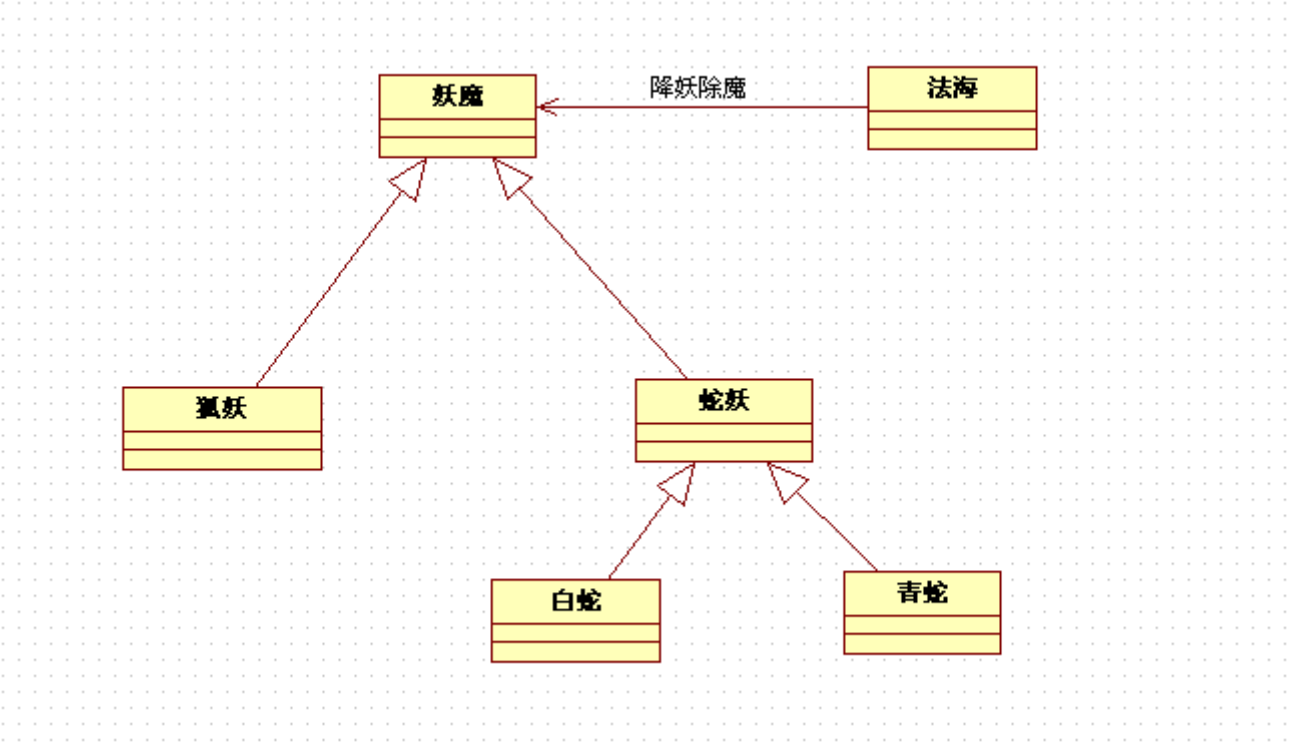


故事分析：

法海的任务是降妖除魔、造福众生。凡是妖魔现身，法海必定全力诛灭之。



而白蛇是修炼千年的蛇妖，蛇妖也是妖。



法海要做的就是不管你是什么妖魔，只要是妖魔我就要把你收服，以防再次出来危害人间。他并不用区分你是蛇妖还是狐妖，更不用管你蛇妖中的青蛇还是白色。

凡是对妖魔这个基类适用的 白蛇、青蛇就全部适用。

Java 代码实现：

妖魔的基类：

```
package com.diermeng.DesignPattern.LSP;


/*
 * 妖魔的基类
 */
public abstract class Spirit {
    /*
     * 妖魔的行为方法
     */
    public abstract void say();
}
```

蛇妖的基类

```
package com.diermeng.DesignPattern.LSP;
/*
 * 蛇妖的基类，继承妖魔，可以扩展方法属性
```

```
 */
public abstract class Snake extends Spirit {


    @Override
    public abstract void say();


}
```

白蛇类：

```
package com.diermeng.DesignPattern.LSP;
/*
 * 白蛇对蛇妖的继承实现
 */
public class WhiteSnake extends Snake {


    @Override
    public void say() {
        System.out.println("我是白蛇");
    }


}
```

建立一个测试类，代码如下：

```
package com.diermeng.DesignPattern.LSP.client;


import com.diermeng.DesignPattern.LSP.Spirit;
import com.diermeng.DesignPattern.LSP.WhiteSnake;


public class LSPTest {


    /**
     * @param args
     */
    public static void main(String[] args) {
        Spirit spirit = new WhiteSnake();
```

```
        spirit.say();
    }


}
```

程序运行结果如下：

```
我是白蛇
```

已有应用简介：

这里主要分析一下 Java 编译器对里氏代换原则的支持机制。

在 Java 中如果子类覆盖了基类的方法，子类是该方法的访问权限必须是不能低于它在父类中的访问权限的。这样在才能保证在客户端程序调用父类相应方法而需要使用子类的相应的方法代替时（调用父类的方法时需要使用子类相应的方法是普遍的），不会因为子类的方法降低了访问权限而导致客户端不能在继续调用。

温馨提示：

里氏代换原则是很多其它设计模式的基础。

它和开放封闭原则的联系尤其紧密。违背了里氏代换原则就一定不符合开放封闭原则。

迪米特法则 慈禧太后为何不和陌生人说话

应用场景举例：

在《投名状》这部轰动一时的影片中有这么一个片段，慈禧太后召见庞青龙，带路的太监说，从门口到见到老佛爷（也就是慈禧太后）这条短短的路他花了大半辈子才走完，而很多人一辈子也走不完，感叹道：“你倒好，这么短的时间里就走了别人花费一生才能走完的道路”。

定义：

迪米特法则(Law of Demeter，简写 LoD)又叫做最少知识原则（LeastKnowledge Principle 简写 LKP），也就是说，一个对象应当对其他对象尽可能少的了解，不和陌生人说话。

迪米特法则最初是用来作为面向对象的系统设计风格的一种法则，于 1987 年秋天由 lanholland 在美国东北大学为一个叫做迪米特的项目设计提出的。被 UML 的创始者之一 Booch 等普及。后来，因为在经典著作《 The PragmaticProgrammer》阐述而广为人知。

狭义的迪米特法则是指：如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一类的某一个方法的话，可以通过第三者转发这个调用。

广义的迪米特法则是指：一个模块设计的好坏的一个重要标志就是该模块在多大程度上讲自己的内部数据与实现的有关细节隐藏起来。

一个软件实体应当尽可能少的与其他实体发生相互作用。

每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。

迪米特法则的目的在于降低类与类之间的耦合。由于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块功能独立，是的相互间存在尽可能少的依赖关系。

在运用迪米特法则到系统的设计中时,要注意以下几点：

第一：在类的划分上，应当创建弱耦合的类，类与类之间的耦合越弱，就越有利于实现可复用的目标。

第二：在类的结构设计上，每个类都应该降低成员的访问权限。

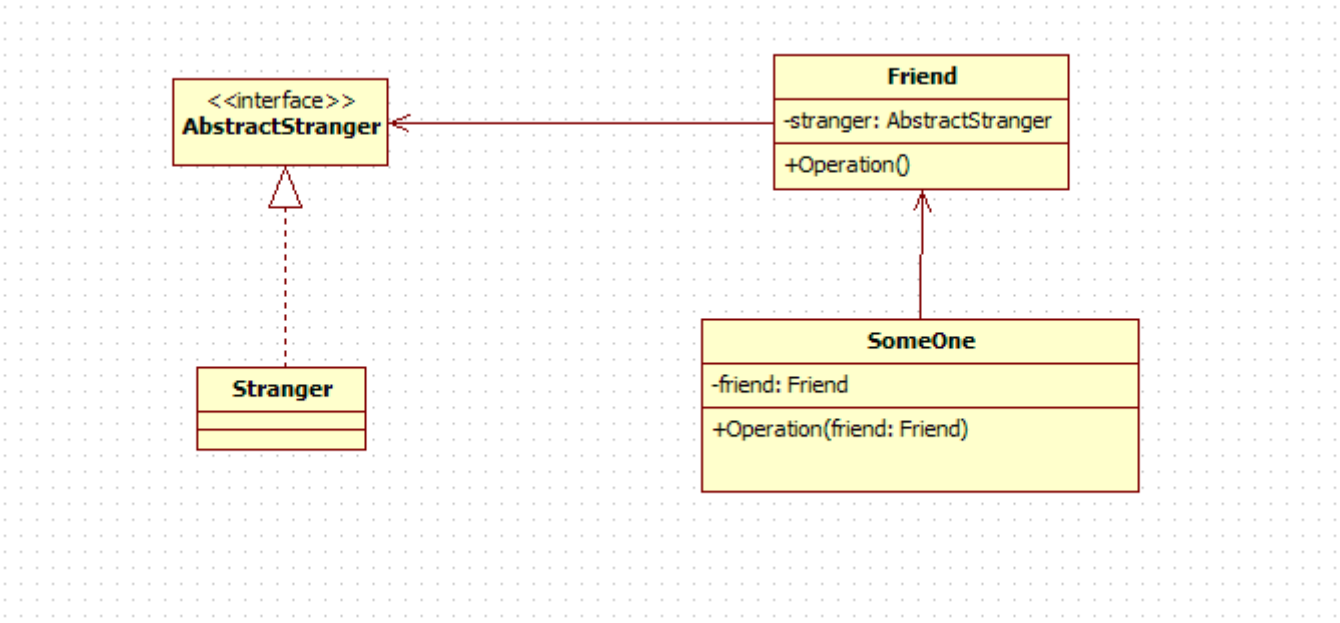
第三：在类的设计上，只要有可能，一个类应当设计成不变的类。

第四：在对其他类的应用上，一个对象对其他类的对象的应用应该降到最低。

第五：尽量限制局部变量的有效范围。

但是过度使用迪米特法则，也会造成系统的不同模块之间的通信效率降低，使系统的不同模块之间不容易协调等缺点。同时，因为迪米特法则要求类与类之间尽量不直接通信，如果类之间需要通信就通过第三方转发的方式，这就直接导致了系统中存在大量的中介类，这些类存在的唯一原因是为了传递类与类之间的相互调用关系，这就毫无疑问的增加了系统的复杂度。解决这个问题的方式是：使用依赖倒转原则（通俗的讲就是要针对接口编程，不要针对具体编程），这要就可以是调用方和被调用方之间有了一个抽象层，被调用方在遵循抽象层的前提下就可以自由的变化，此时抽象层成了调用方的朋友。

如下图所示：



故事分析：

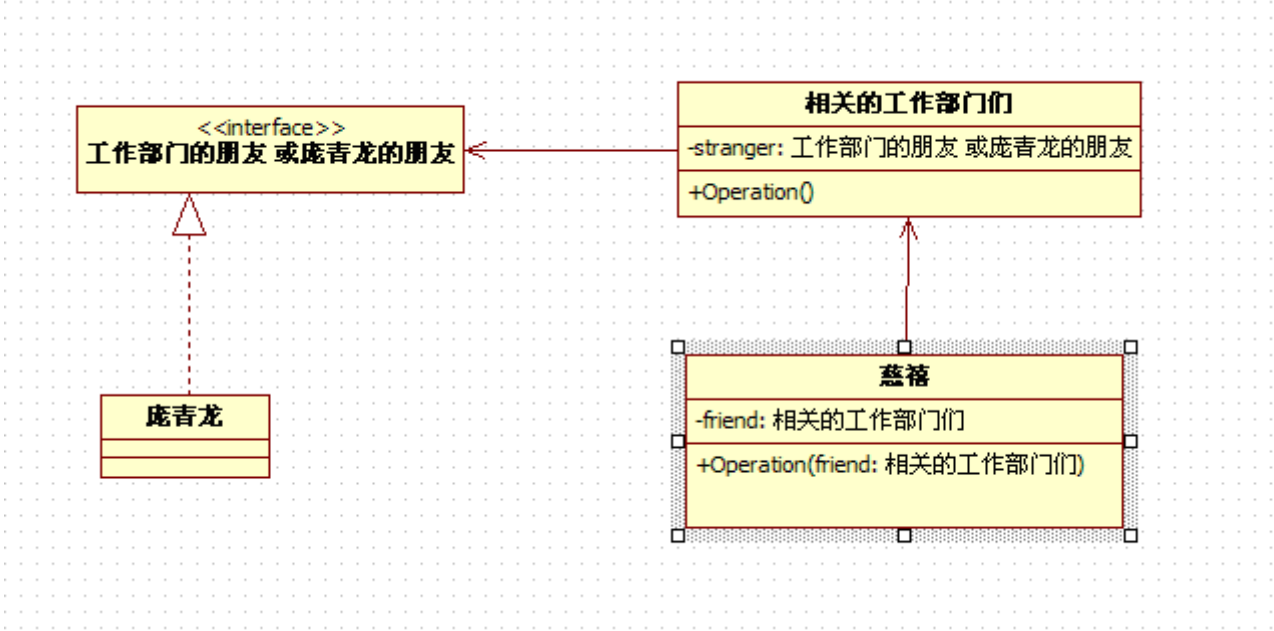
慈禧太后要召见庞青龙。庞青龙在见到慈禧太后前经历了那些过程呢？首先，当然是有人通知庞青龙要被召见，通知庞青龙的人当然不会是慈禧本人！慈禧只是下达旨意，然后又相关的只能部门传达旨意，相关部门的领导人也不会亲自去通知庞青龙，这些领导人会派遣信得过的人去，而这个被派遣的人也不是说想见庞青龙就能见得了的，他也必须通过和庞青龙熟悉的人，最后才能见到庞青龙，从而才能成功的传达旨意；第二：在进宫前，庞青龙必须卸掉自己随身携带的任何武器；第三：会有专门的只能部门对庞青龙进行全身彻底的检查，以防有任何可以伤害人的东西携带在身上，当然这个过程可能非常的复杂和繁琐。最后，由一个太监带路到慈禧面前。当然，见到慈禧的时候，庞青龙不是和慈禧坐在一起的，要报仇距离！慈禧也深深的懂得保持距离的重要性！

见到慈禧太后以后慈禧也没有和庞青龙直接说话，因为慈禧不和陌生人说话！而是同时身边的人传达自己的话，慈禧只需颐指气使即可。

从上面的过程中我们可以看出处处体现了迪米特法则的应用，慈禧知道庞青龙这个人肯定是通过一层又一层的关系得知的，就是迪米特法则中的第三者转发而且这里面说不定还有若干个第三者的转发！而从慈溪下旨召见庞青龙到庞青龙收到旨意，这中间又是完美的提现了迪米特法则，这中间经历无数的第三者！就连庞青龙面见到慈禧后，慈禧也不和他直接说话，而是通过身边的人传话，这慈禧是不是太傻了，直接和他说不就行了吗?慈禧当然不傻，因为她深知迪米特法则的重要。两个类的对象之间如果不发生直接的联系就不直接发生关系！

不过这也产生了一个问题，这中间经历这么多的转发，需要机构和人啊？或许这就是为什么当时的清政府机构那么庞大、财政开支惊人的原因之一吧^_^

如下图所示：



Java 代码实现：

新建一个陌生人的抽象父类，其他的陌生人继承这个接口：

```
package com.diermeng.designPattern.LoD;
/*
 * 抽象的陌生人类
 */
public abstract class Stranger {
    /*
     * 抽象的行为方法
     */
    public abstract void operation();
}
```

庞青龙实现继承实现陌生人抽象类：

```
package com.diermeng.designPattern.LoD.impl;


import com.diermeng.designPattern.LoD.Stranger;
/*
 * 庞青龙对抽象类 Stranger 的实现
 */
public class PangQingyong extends Stranger{
    /*
```

```
    * 操作方法
    * @see com.diermeng.designPattern.LoD.Stranger#operation()
    */
    public void operation(){
        System.out.println("禀报太后：我是庞青龙，我擅长用兵打仗！");
    }
}
```

朋友类，这里指太监类

```
package com.diermeng.designPattern.LoD.impl;

import com.diermeng.designPattern.LoD.Stranger;
/*
 * 太监类
 */
public class Taijian {
    /*
     * 太监类的操作方法
     */
    public void operation(){
        System.out.println("friends paly");
    }

    /*
     * 由太监类提供 Cixi 需要的方法
     */
    public void findStranger() {
        //创建一个 Stranger
        Stranger stranger = new PangQingyong();
        //执行相应的方法
        stranger.operation();
    }
}
```

调用者类的代码，在这里是慈禧太后类：

```
package com.diermeng.designPattern.LoD.impl;

/*
 * 慈禧类
 */
public class Cixi {
    //拥有对太监的引用，即对“朋友”的引用
    private Taijian taijian;

    //得到一个太监对象
    public Taijian getTaijian() {
        return taijian;
    }

    //设置一个太监对象
    public void setTaijian(Taijian taijian) {
        this.taijian = taijian;
    }

    //操作方法
    public void operation(){
        System.out.println("someone play");
    }


}
```

建立一个测试类，代码如下：

```
package com.diermeng.designPattern.LoD.client;

import com.diermeng.designPattern.LoD.impl.Taijian;
import com.diermeng.designPattern.LoD.impl.Cixi;

/*
 * 测试客户端
 */
public class LoDTest {
```

```
    public static void main(String[] args) {
        //声明并实例化慈禧类
        Cixi zhangsan = new Cixi();

        //设置一个太监实例化对象，即找到一个"朋友"帮忙做事
        zhangsan.setTaijian(new Taijian());

        //慈禧通过宫中太监传话给陌生人
        zhangsan.getTaijian().findStranger();
    }
}
```

程序运行结果如下：

```
禀报太后：我是庞青龙，我擅长用兵打仗！
```

已有应用简介：

迪米特法则或者最少知识原则作为面向对象设计风格的一种法则，也是很多著名软件设计系统的指导原则，比如火星登陆软件系统、木星的欧罗巴卫星轨道飞船软件系统。

温馨提示：

迪米特法则是一种面向对象系统设计风格的一种法则，尤其适合做大型复杂系统设计指导原则。但是也会造成系统的不同模块之间的通信效率降低，使系统的不同模块之间不容易协调等缺点。同时，因为迪米特法则要求类与类之间尽量不直接通信，如果类之间需要通信就通过第三方转发的方式，这就直接导致了系统中存在大量的中介类，这些类存在的唯一原因是为了传递类与类之间的相互调用关系，这就毫无疑问的增加了系统的复杂度。解决这个问题的方式是：使用依赖倒转原则（通俗的讲就是要针对接口编程，不要针对具体编程），这要就可以是调用方和被调用方之间有了一个抽象层，被调用方在遵循抽象层的前提下就可以自由的变化，此时抽象层成了调用方的朋友。

合成聚合复用原则 刘邦 VS 韩信

应用场景举例：


VS


一次，刘邦闲着没事，又想起了故人韩信，于是打开 QQ 想和韩信聊天，正巧韩信也在，开始寒暄了几句，就在不自觉中进入严肃的问题，刘邦问："韩兄觉得如果是我带兵，最多能够带多少呢？"，韩信立即回复说："十万"，刘邦心想本王竟然只能带十万，那你韩信又能够带多少呢，于是暂时按捺了心中的郁闷，很客气的问道："那请问韩兄最多能够带多少呢"，韩信又立即回复道："我啊，那自然是越多越好啦"，刘邦看到此言顿时大怒，而且语气还极其的傲慢，还"啊，啦"的，刘邦心想："虽然你韩信带兵打仗有道，天下皆知，但是这样对本王说话也太过分了吧"，刘邦正要发飙，随即停了一下，深谙世道的刘邦问了一句："将军神勇盖世，带兵百万，却为何会在我领导下呢？"，刘邦想："好你个韩信，叫了你几声韩兄你就不知道自己是谁了，如果回答不上来，或是回答不好，看我如何收拾你！"，等了大约三秒钟，QQ 闪了一下，只见上面赫然写道："陛下虽不善统兵，却善御将"。刘邦大悦！

定义：

合成聚合复用原则（Composite Aggregate Reuse Principle,简称为 CARP）经常又被人们称为合成复用原则（Composite Reuse Principle,简称为 CRP）.合成聚合复用原则是指在一个新的对象中使用原来已经存在的一些对象，是这些原来已经存在的对象称为新对象的一部分，新的对象通过向这些原来已经具有的对象委派相应的动作或者命令达到复用已有功能的目的。

合成复用原则跟简洁的表述是：要尽量使用合成和聚合，尽量不要使用继承。

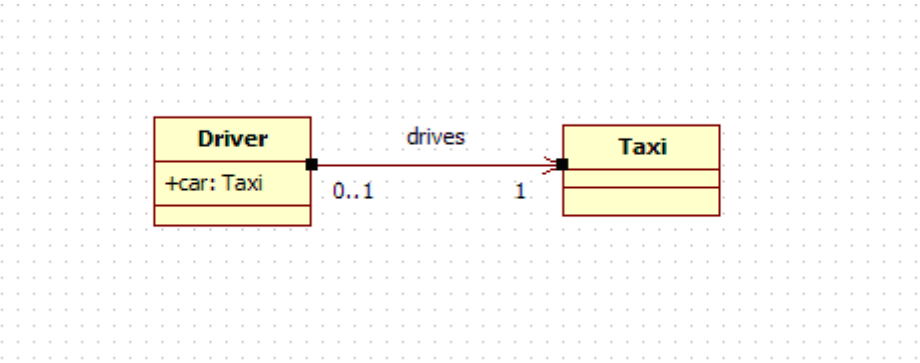
聚合（Aggregation）是关联关系的一种，用来表示一种整体和部分的拥有关系。整体持有对部分的引用，可以调用部分的能够被访问的方法和属性等，当然这种访问往往是对接口和抽象类的访问。作为部分可以可以同时被多个新的对象引用，同时为多个新的对象提供服务。

合成（Composition）也是关联关系的一种，但合成是一种比聚合强得多的一种关联关系。在合成关系里面，部分和整体的生命周期是一样的。作为整体的新对象完全拥有对作为部分的支配权，包括负责和支配部分的创建和销毁等，即要负责作为部分的内存的分配和内存释放等。从这里也可以看出来，一个合成关系中的成员对象是不能喝另外的一个合成关系共享的。

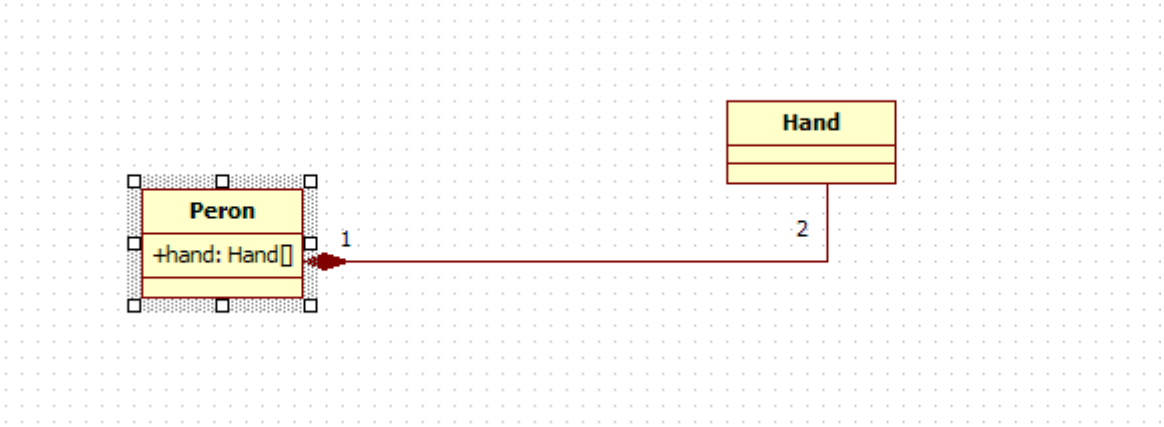
为何"要尽量使用合成和聚合，尽量不要使用继承"呢？这是因为：第一，继承复用破坏包装，它把超类的实现细节直接暴露给了子类，这违背了信息隐藏的原则；第二：如果超类发生了改变，那么子类也要发生相应的改变，这就直接导致

了类与类之间的高耦合，不利于类的扩展、复用、维护等，也带来了系统僵硬和脆弱的设计。而是用合成和聚合的时候新对象和已有对象的交互往往是通过接口或者抽象类进行的，就可以很好的避免上面的不足，而且这也可以让每一个新的类专注于实现自己的任务，符合单一职责原则。

聚合关系的示意图如下所示：



聚合关系的示意图如下所示：

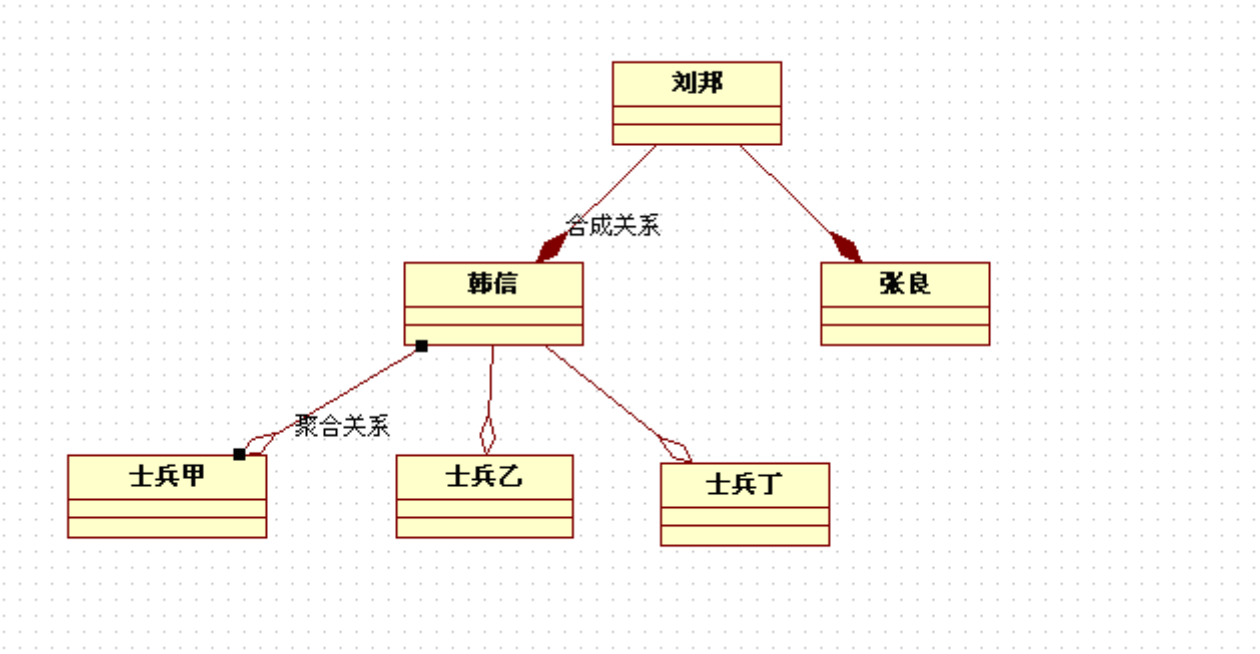


故事分析：

“韩信带兵，多多益善”，韩信之所以有很大的影响力，一方面归功他建立了很大的功勋；另外一方面是因为他手握重兵。建立很大的功勋甚至是功高盖主，这就直接导致韩信在军队中深得人心，毕竟，作为士兵，很少有不愿意追随一个在战场上无往不胜的领袖的；手握重兵，有合成聚合的原则来说就是对自己统领的士兵将士保持有引用，就是聚合。韩信手中聚合有一大批能征善战的人，随时听从韩信的调遣，这是不能不让刘邦戒备的。而现在，又说了一个自己带兵多多益善的话，这怎能不让刘邦感觉生气呢？

而“普天之下，莫非王土；四海之内，莫非王臣”，韩信拥有士兵，刘邦却拥有天下，何况韩信也并非拥有全部士兵。“君叫臣死，臣不得不死”，刘邦拥有对天下苍生的生杀大权。韩信很清楚，“陛下虽不善统兵，却善御将”。刘邦是拥有一种比韩信更强的“拥有”关系，可以主宰生死，同时也可以主宰韩信的生死。这就相当于合成关系。更重要的是韩信也是合成关系的一个成员，是刘邦的将领。所以说刘邦比韩信更加强悍！

如下图所示：



Java 代码实现：

新建大臣的接口：

```
package com.diermeng.designPattern.CARP;
/*
 * 大臣的接口
 */
public interface Minister {
    /*
     * 大臣能够执行的动作
     */
    public void duty();
}
```

士兵的接口

```
package com.diermeng.designPattern.CARP;
/*
 * 士兵的接口
 */
public interface Soldier {
    /*
     * 士兵能够执行的动作
     */
    public void duty();
```

```
}
```

韩信对大臣接口的实现

```
package com.diermeng.designPattern.CARP.impl;

import com.diermeng.designPattern.CARP.Minister;
import com.diermeng.designPattern.CARP.Soldier;
/*
 * 韩信对大臣接口的实现
 */
public class Hanxin implements Minister {
    //对士兵的聚合关系
    Soldier[] soldiers;

    /*
     * 无参构造方法
     */
    public Hanxin() {}
    /*
     * 有士兵参数的构造方法
     */
    public Hanxin(Soldier[] soldiers) {
        super();
        this.soldiers = soldiers;
    }

    /*
     * 获取士兵的集合
     */
    public Soldier[] getSoldiers() {
        return soldiers;
    }

    /*
     * 设置士兵的集合
     */
```

```
    public void setSoldiers(Soldier[] soldiers) {
        this.soldiers = soldiers;
    }
    /*
     * 韩信的职能
     * @see com.diermeng.designPattern.CARP.Minister#duty()
     */
    public void duty() {
        System.out.println("我是刘邦的大臣，永远忠实于刘邦");

    }

}
```

士兵 A 对士兵接口的实现

```
package com.diermeng.designPattern.CARP.impl;

import com.diermeng.designPattern.CARP.Soldier;
/*
 * 士兵 A
 */
public class SoldierA implements Soldier {
    /*
     * 士兵 A 的职责
     * @see com.diermeng.designPattern.CARP.Soldier#duty()
     */
    public void duty() {
        System.out.println("我是韩信的士兵 A");

    }

}
```

士兵 B 对士兵接口的实现

```
package com.diermeng.designPattern.CARP.impl;

import com.diermeng.designPattern.CARP.Soldier;
```

```
/*
 * 士兵 B
 */
public class SoldierB implements Soldier {
    /*
     * 士兵 B 的职责
     * @see com.diermeng.designPattern.CARP.Soldier#duty()
     */
    public void duty() {
        System.out.println("我是韩信的士兵 B");

    }

}
```

刘邦类

```
package com.diermeng.designPattern.CARP.impl;

import com.diermeng.designPattern.CARP.Minister;

/*
 * 刘邦类
 */
public class Liubang {
    //拥有大臣
    Minister[] minister;

    /*
     * 无参构造方法
     */
    public Liubang() {}
    /*
     * 把大臣的数组作为参数传入构造方法
     */
    public Liubang(Minister[] minister) {
        super();
        this.minister = minister;
    }
```

```
    /*
     * 获取大臣的集合
     */
    public Minister[] getMinister() {
        return minister;
    }

    /*
     * 设置大臣的集合
     */
    public void setMinister(Minister[] minister) {
        this.minister = minister;
    }

    /*
     * 刘邦的职能
     */
    public void duty()
    {
        System.out.println("我是皇帝，普天之下，莫非王土；四海之内，莫非王臣");
    }
}
```

建立一个测试类，代码如下：

```
package com.diermeng.designPattern.CARP.client;

import com.diermeng.designPattern.CARP.Minister;
import com.diermeng.designPattern.CARP.Soldier;
import com.diermeng.designPattern.CARP.impl.Hanxin;
import com.diermeng.designPattern.CARP.impl.Liubang;
import com.diermeng.designPattern.CARP.impl.SoldierA;
import com.diermeng.designPattern.CARP.impl.SoldierB;

/*
 * 测试类的客户端
 */
public class CARPClient {
```

```
    public static void main(String[] args)
    {
        //声明并实例化士兵 A
        Soldier soldierA = new SoldierA();
        //声明并实例化士兵 B
        Soldier soldierB = new SoldierB();
        //构造士兵数组
        Soldier[] soldiers = {soldierA,soldierB};

        //声明并实例化韩信，同时传入士兵数组
        Minister hanxin = new Hanxin(soldiers);

        //构造大臣数组
        Minister[] minister = {hanxin};
        //声明并实例化刘邦，同时传入大臣数组
        Liubang liubang = new Liubang(minister);

        liubang.duty();

        //循环输出大臣
       for(Minister aminister:liubang.getMinister()){
           aminister.duty();
        }

    }
}
```

程序运行结果如下：

```
我是皇帝，普天之下，莫非王土；四海之内，莫非王臣
我是刘邦的大臣，永远忠实于刘邦
```

已有应用简介：

对于面向对象的软件系统而言，如何提高软件的可维护性和可复用性始终是一个核心问题。合成聚合复用原则的合理而充分的使用时非常有利于构建可维护、可复用、可扩展和灵活性好的软件系统。合成聚合复用原则作为一种构造优质系统的手段几乎可以应用到任何环境中去。对于已有的应用这里就不在赘述啦。

温馨提示：

合成聚合复用原则虽然几乎可以应用到任何环境中去，但是这个原则也有自己的缺点。因为此原则鼓励使用已有的类和对象来构建新的类的对象，这就导致了系统中会有很多的类和对象需要管理和维护，从而增加系统的复杂性。

同时，也不是说在任何环境下使用合成聚合复用原则就是最好的，如果两个类之间在符合分类学的前提下有明显的“IS-A”的关系，而且基类能够抽象出子类的共有的属性和方法，而此时子类有能通过增加父类的属性和方法来扩展基类，那么此时使用继承将是一种更好的选择。

简单工厂模式 一见钟情的代价

简单工厂模式应用场景举例：

“你知不知道大学的规矩啊？”，MM 有些不满的问道。“什么规矩？当然不知道了啊。”，GG 傻傻的说道，很明显这个 MM 已经对 GG 的不懂事和不主动有些不满了。“在大学里，当两个人确定恋爱关系时，都是要请女朋友同寝室的人去吃饭的”，MM 带着一些不满又有一些撒娇的口气说道。“啊？我不知道哎，请众美女吃饭我还求之不得呢，什么时候有时间啊，确定是时间和地点，我随叫随到！”GG 很激动很爽快的答应道。MM 笑着抬头看了一样这个傻 GG,“那好，让我想想，我们...我们下周六下午有时间，要么这样，你带我们去麦当劳吧”，“一言为定”，“那我们在下周六下午五点在中心商业街南边的麦当劳分店见，听说那边的口味还不错:-O”，“好的，只要你开心就好，不见不散”GG 回答道，“不见不散！”MM 就这样嫣然一笑的欢天喜地的离开了。

想想前几天 GG 和 MM 因为非常偶然的因素相见的情景，GG 再次涌起了一种无法言喻的幸福和激动。那一天，GG 见到了 MM，仿若晴天霹雳，整个地球在颤抖，她甜美而柔和的声音、她极具古典气息的是秀发、她超棒的身材、她恰到好处的着装、她极尽秀美而恬静的娇容、她似音乐般的举止顿时令他彻底的迷醉了，仿佛整个世界只有她一人，仿佛一切都是为她而生的，突然，两人目光交错，眼神相遇...就这么一见钟情！GG 想，到麦当劳也好，反正我不会做饭，再说了，即使会做也不能去做啊，众口难调啊，更何况是一群美女，到麦当劳让她们自个儿去挑吧^_^不过我这一个月的生活费怕是要泡汤了，难怪别人说大学里最高的消费是花费的女朋友身上的消费~~~~(>_<)~~~~

千呼万唤，终于到了周六下午。被感情冲昏大脑的 GG 突然间变的不再那么笨了，这次他提前预定了座位，是一个可以容纳 8 个人的座位。而且具体告诉了 MM 座位的位置，这样大家都清楚位置是比较好的，避免了到时候没有位置的尴尬。赶往麦当劳路上的 GG 心潮澎湃但是有些担心，毕竟要面对六个美女，而且女朋友也是刚认识几天。“亲爱的，现在到哪了？”手机中 MM 发过来了一条短信，GG 一看时间，天啊，光顾着去傻想，还有几分钟就五点了，第一次如果都迟到那就太不好了，于是立即回复到，“宝贝儿，我就到了！”，因为麦当劳就在对面，抬头就可以看到的。GG 跑上了麦当劳的二楼的用餐处，见到诸位美女，紧张的还没说不话来，“这是我男朋友”MM 拽着 GG 的手臂说，“大家好，大家好”，GG 紧张的说道。忙又补充到：“我们先点餐，大家自便，都不要客气啊”，“我要吃鸡翅”，“我要麦香鱼套餐”，我要“板烧鸡腿套餐”，我要“奶昔”，我要“薯条”，...，大家都点好自己的喜欢的食品，然后 GG 和 MM 分别又加了几份食品，有 GG 把订单拿到前台交给了服务员，服务员清算了一下所有花费，GG 当即晕倒^_^。看来一个月的生活费是确实的泡汤了，不过还是故作振作，微笑着来到众美女中，和众美女坐在那里等着慢慢享用美食，而剩下的一切就交给服务员了...

简单工厂模式解释：

简单工厂模式（Simple Factory Pattern）属于类的创新型模式，又叫静态工厂方法模式（Static FactoryMethod Pattern）,是通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

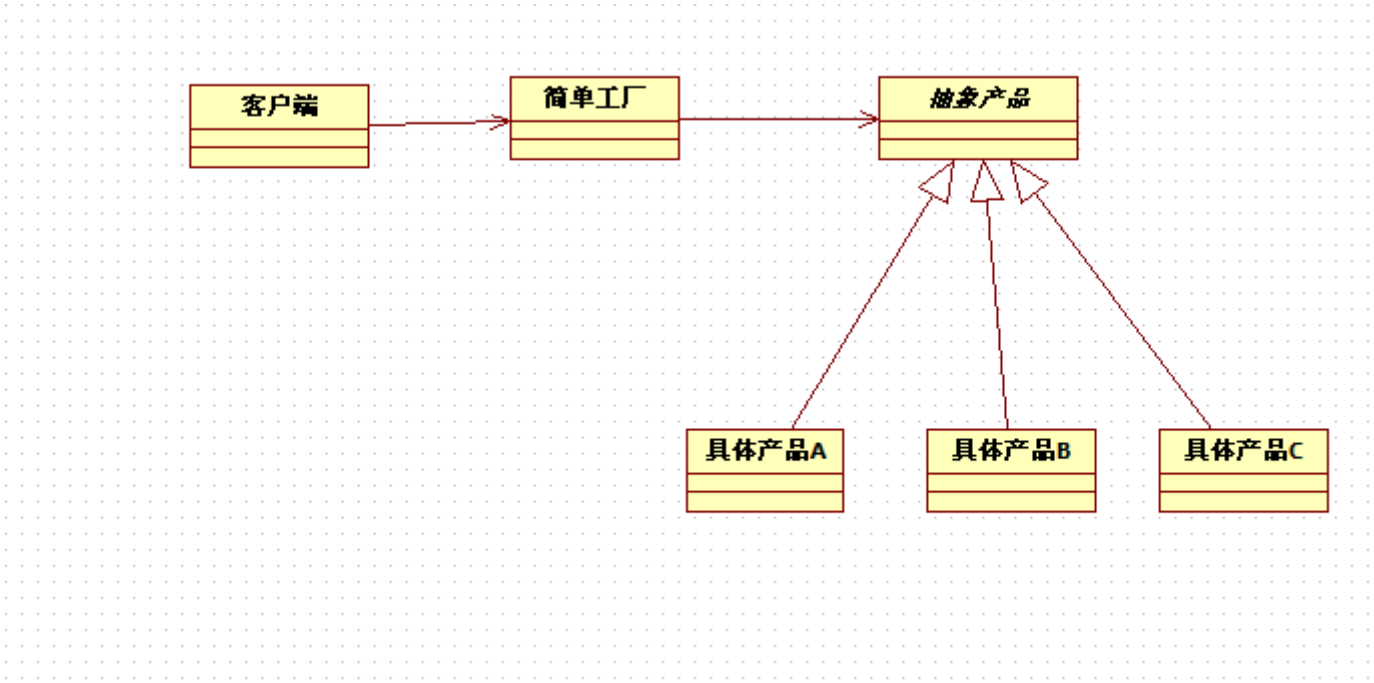
简单工厂模式的 UML 图：

简单工厂模式中包含的角色及其相应的职责如下：

工厂角色（Creator）：这是简单工厂模式的核心，由它负责创建所有的类的内部逻辑。当然工厂类必须能够被外界调用，创建所需要的产品对象。

抽象（Product）产品角色：简单工厂模式所创建的所有对象的父类，注意，这里的父类可以是接口也可以是抽象类，它负责描述所有实例所共有的公共接口。

具体产品（Concrete Product）角色：简单工厂所创建的具体实例对象，这些具体的产品往往都拥有共同的父类。



简单工厂模式深入分析：

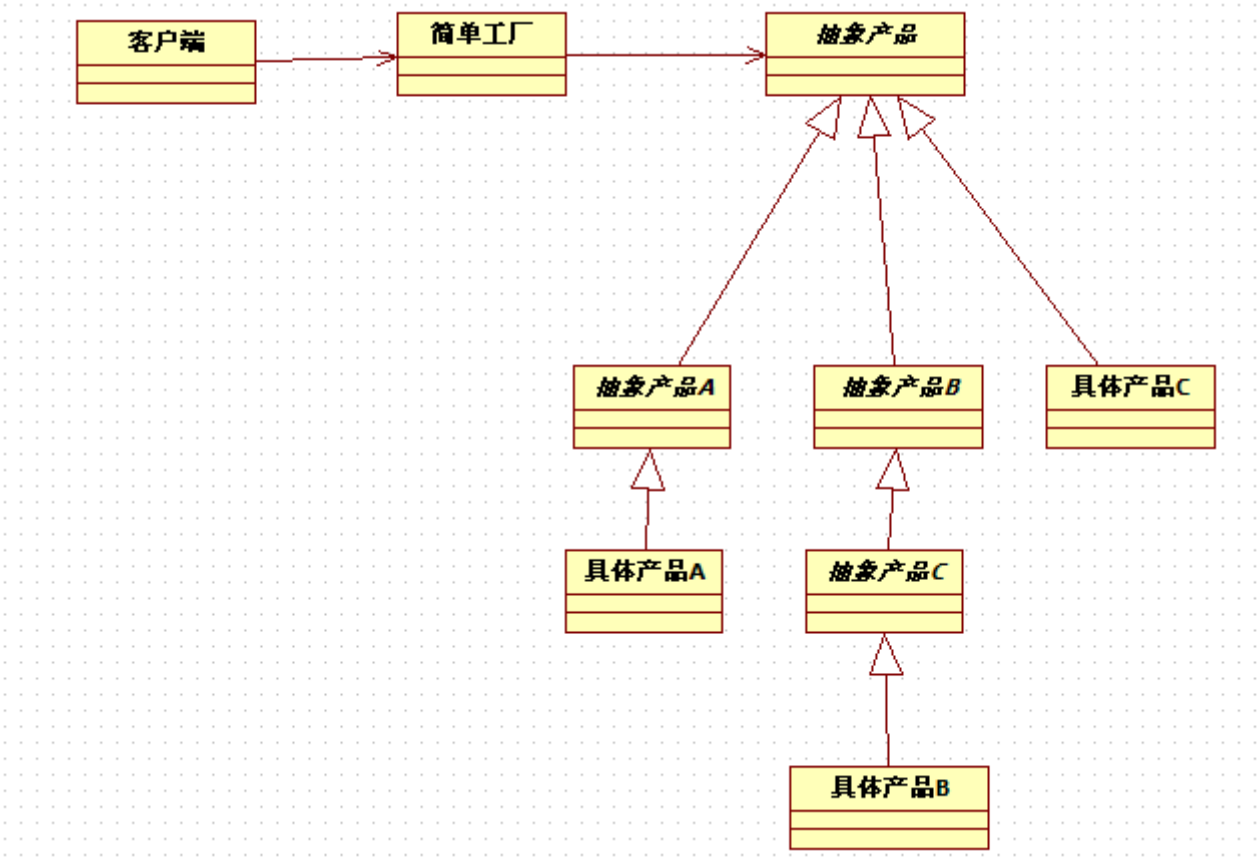
简单工厂模式解决的问题是如何去实例化一个合适的对象。

简单工厂模式的核心思想就是：有一个专门的类来负责创建实例的过程。

具体来说，把产品看着是一系列的类的集合，这些类是由某个抽象类或者接口派生出来的一个对象树。而工厂类用来产生一个合适的对象来满足客户的要求。

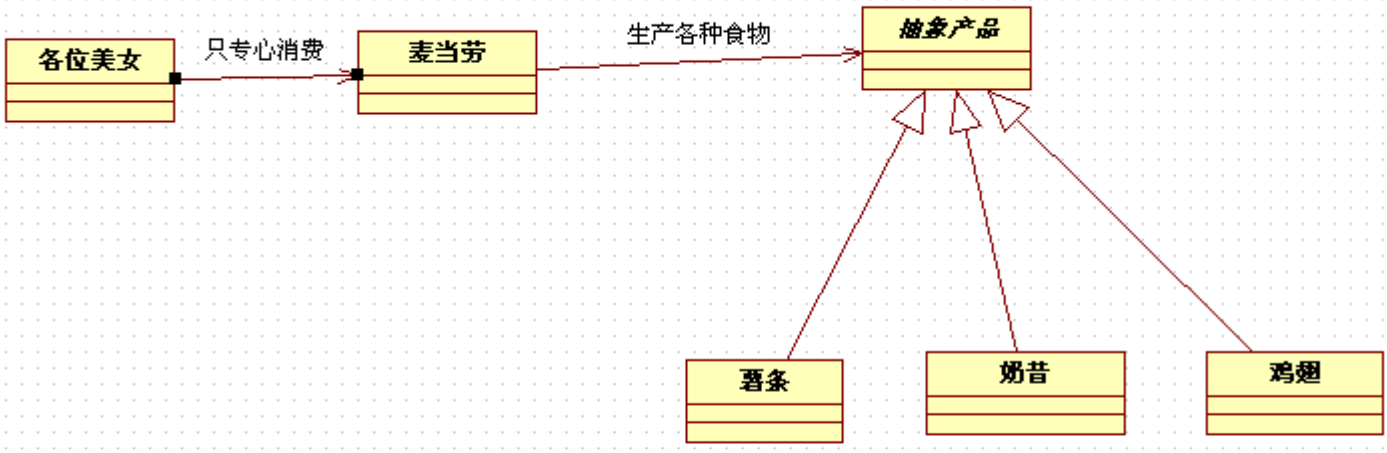
如果简单工厂模式所涉及到的具体产品之间没有共同的逻辑，那么我们就可以使用接口来扮演抽象产品的角色；如果具体产品之间有功能的逻辑或，我们就必须把这些共同的东西提取出来，放在一个抽象类中，然后让具体产品继承抽象类。为实现更好复用的目的，共同的东西总是应该抽象出来的。

在实际的的使用中，抽闲产品和具体产品之间往往是多层次的产品结构，如下图所示：



简单工厂模式使用场景分析及代码实现：

GG 请自己的女朋友和众多美女吃饭，但是 GG 自己是不会做饭的或者做的饭很不好，这说明 GG 不用自己去创建各种食物的对象；各个美女都有各自的爱好，到麦当劳后她们喜欢吃什么直接去点就行了，麦当劳就是生产各种食物的工厂，这时候 GG 不用自己动手，也可以请这么多美女吃饭，所要做的就是买单 O(∩_∩)O 哈哈~,其 UML 图如下所示：



实现代码如下：

新建立一个食物的接口：

```
package com.diermeng.designPattern.SimpleFactory;


/*
 * 产品的抽象接口
 */
public interface Food {
    /*
     * 获得相应的食物
     */
    public void get();
}
```

接下来建立具体的产品：麦香鸡和薯条

```
package com.diermeng.designPattern.SimpleFactory.impl;
import com.diermeng.designPattern.SimpleFactory.Food;


/*
 * 麦香鸡对抽象产品接口的实现
 */
public class McChicken implements Food{
    /*
     * 获取一份麦香鸡
     */
    public void get(){
        System.out.println("我要一份麦香鸡");
    }
}
```

```
package com.diermeng.designPattern.SimpleFactory.impl;
import com.diermeng.designPattern.SimpleFactory.Food;


/*
 * 薯条对抽象产品接口的实现
 */
public class Chips implements Food{
```

```
    /*
     * 获取一份薯条
     */
    public void get(){
        System.out.println("我要一份薯条");
    }
}
```

现在建立一个食物加工工厂：

```
package com.diermeng.designPattern.SimpleFactory.impl;
import com.diermeng.designPattern.SimpleFactory.Food;



public class FoodFactory {

    public static Food getFood(String type) throws InstantiationException,
IllegalAccessException, ClassNotFoundException {
        if(type.equalsIgnoreCase("mcchicken")) {
            return McChicken.class.newInstance();

        } else if(type.equalsIgnoreCase("chips")) {
            return Chips.class.newInstance();
        } else {
            System.out.println("哎呀！找不到相应的实例化类啦！");
            return null;
        }


    }
}
```

最后我们建立测试客户端：

```
package com.diermeng.designPattern.SimpleFactory.client;
import com.diermeng.designPattern.SimpleFactory.Food;
import com.diermeng.designPattern.SimpleFactory.impl.FoodFactory;
```

```
/*
 * 测试客户端
 */
public class SimpleFactoryTest {
   public static void main(String[] args) throws InstantiationException,
IllegalAccessException, ClassNotFoundException {

       //实例化各种食物
       Food mcChicken = FoodFactory.getFood("McChicken");
       Food chips = FoodFactory.getFood("Chips");
       Food eggs = FoodFactory.getFood("Eggs");

       //获取食物
       if(mcChicken!=null){
           mcChicken.get();
       }
       if(chips!=null){
           chips.get();
       }
       if(eggs!=null){
           eggs.get();
       }


   }
}
```

输出的结果如下：

```
哎呀！找不到相应的实例化类啦！
我要一份麦香鸡
我要一份薯条
```

简单工厂模式的优缺点分析：

优点：工厂类是整个模式的关键所在。它包含必要的判断逻辑，能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。用户在使用时可以直接根据工厂类去创建所需的实例，而无需了解这些对象是如何创建以及如何组织的。有利于整个软件体系结构的优化。

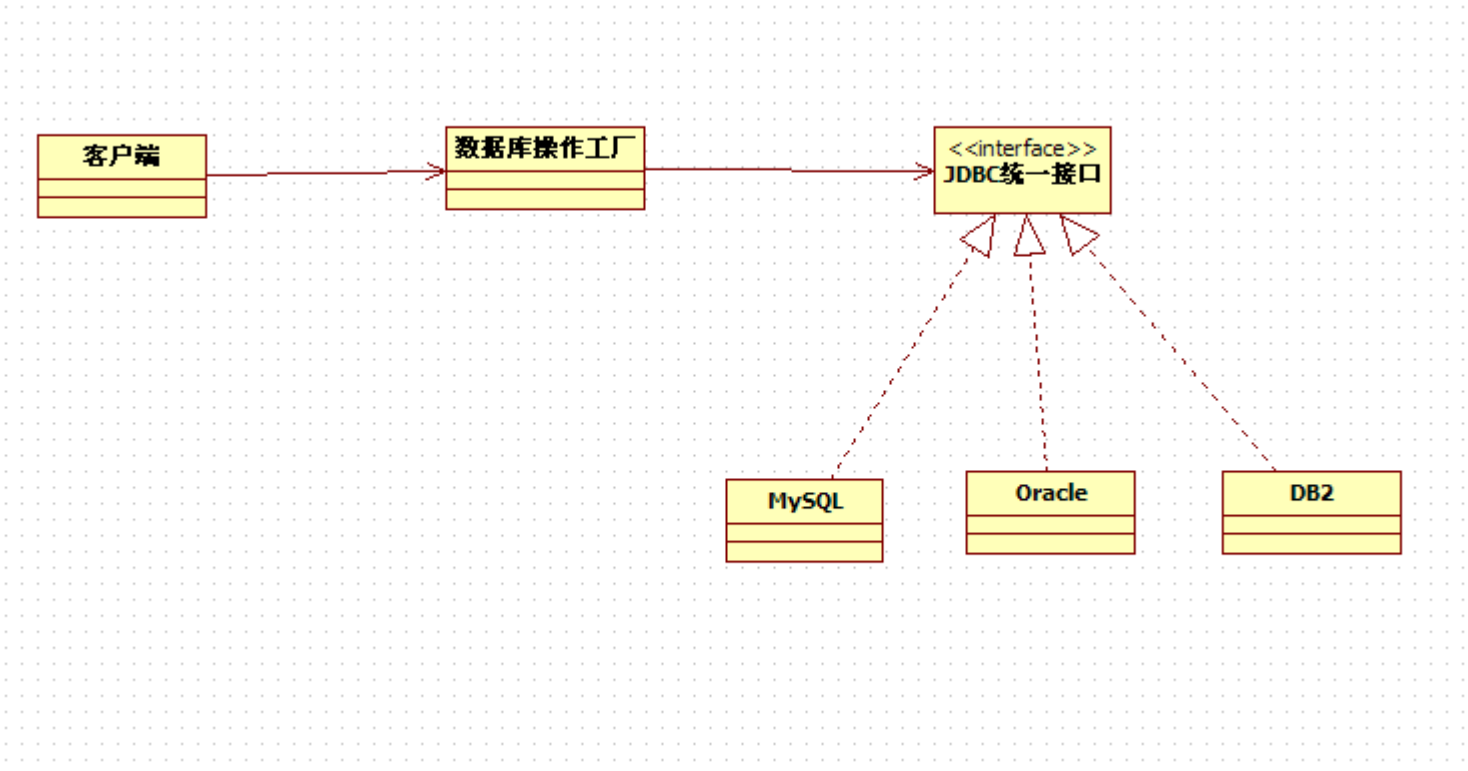
缺点：由于工厂类集中了所有实例的创建逻辑，这就直接导致一旦这个工厂出了问题，所有的客户端都会受到牵连；而且由于简单工厂模式的产品室基于一个共同的抽象类或者接口，这样一来，但产品的种类增加的时候，即有不同的产品接口或者抽象类的时候，工厂类就需要判断何时创建何种种类的产品，这就和创建何种种类产品的产品相互混淆在了一起，违背了单一职责，导致系统丧失灵活性和可维护性。而且更重要的是，简单工厂模式违背了“开放封闭原则”，就是违背了“系统对扩展开放，对修改关闭”的原则，因为当我新增加一个产品的时候必须修改工厂类，相应的工厂类就需要重新编译一遍。

总结一下：简单工厂模式分离产品的创建者和消费者，有利于软件系统结构的优化；但是由于一切逻辑都集中在一个工厂类中，导致了没有很高的内聚性，同时也违背了“开放封闭原则”。另外，简单工厂模式的方法一般都是静态的，而静态工厂方法是无法让子类继承的，因此，简单工厂模式无法形成基于基类的继承树结构。

简单工厂模式的实际应用简介：

作为一个最基本和最简单的设计模式，简单工厂模式却有很非常广泛的应用，我们这里以 Java 中的 JDBC 操作数据库为例来说明。

JDBC 是 SUN 公司提供的一套数据库编程接口 API，它利用 Java 语言提供简单、一致的方式来访问各种关系型数据库。Java 程序通过 JDBC 可以执行 SQL 语句，对获取的数据进行处理，并将变化了的数据存回数据库，因此，JDBC 是 Java 应用程序与各种关系数据进行对话的一种机制。用 JDBC 进行数据库访问时，要使用数据库厂商提供的驱动程序接口与数据库管理系统进行数据交互。



客户端要使用使用数据时，只需要和工厂进行交互即可，这就导致操作步骤得到极大的简化，操作步骤按照顺序依次为：注册并加载数据库驱动，一般使用 Class.forName();创建与数据库的链接 Connection 对象；创建 SQL 语句对象 preparedStatement(sql)；提交 SQL 语句，根据实际情况使用 executeQuery()或者 executeUpdate()；显示相应的结果；关闭数据库。

温馨提示：

严格意义上讲，简单工厂模式并不算是一种设计模式，简单工厂模式更像是一种编程习惯，而这被广泛的应用。但是因为简单工厂模式在“高内聚”方面的欠缺，同时更致命的是违背了严格意义上的“开放封闭原则”，或者说只对“开放封

闭原则”提供某种程度上的支持，这就使得每次新增加一个产品的时候是非常麻烦的，因为每当增加一种新的产品的时候，工厂角色必须知道这个产品，同时必须知道如何创建这个产品，还要以一种对客户端有好的方式提供给客户端使用。简而言之，就是每增加一个新的产品就必须修改工厂角色的源代码。所以简单工厂模式是不利于构建容易发生变化的系统的。而“需求总是在变化”，“世界上没有一个软件是不变的”。所以使用简单工厂模式的时候必须慎重考虑。

让麦当劳的汉堡适合不同 MM 的不同口味

工厂方法模式应用场景举例：

“你知不知道大学的规矩啊？”，MM 有些不满的问道。“什么规矩？当然不知道了啊。”，GG 傻傻的说道，很明显这个 MM 已经对 GG 的不懂事和不主动有些不满了。“在大学里，当两个人确定恋爱关系时，都是要请女朋友同寝室的人去吃饭的”，MM 带着一些不满又有一些撒娇的口气说道。“啊？我不知道哎，请众美女吃饭我还求之不得呢，什么时候有时间啊，确定是时间和地点，我随叫随到！”GG 很激动很爽快的答应道。MM 笑着抬头看了一样这个傻 GG,“那好，让我想想，我们...我们下周六下午有时间，要么这样，你带我们去麦当劳吧”，“一言为定”，“那我们在下周六下午五点在中心商业街南边的麦当劳分店见，听说那边的口味还不错:-O”，“好的，只要你开心就好，不见不散”GG 回答道，“不见不散！”MM 就这样嫣然一笑的欢天喜地的离开了。

想想前几天 GG 和 MM 因为非常偶然的因素相见的情景，GG 再次涌起了一种无法言喻的幸福和激动。那一天，GG 见到了 MM，仿若晴天霹雳，整个地球在颤抖，她甜美而柔和的声音、她极具古典气息的是秀发、她超棒的身材、她恰到好处的着装、她极尽秀美而恬静的娇容、她似音乐般的举止顿时令他彻底的迷醉了，仿佛整个世界只有她一人，仿佛一切都是为她而生的，突然，两人目光交错，眼神相遇...就这么一见钟情！GG 想，到麦当劳也好，反正我不会做饭，再说了，即使会做也不能去做啊，众口难调啊，更何况是一群美女，到麦当劳让她们自个儿去挑吧^_^不过我这一个月的生活费怕是要泡汤了，难怪别人说大学里最高的消费是花费的女朋友身上的消费~~~~(>_<)~~~~

千呼万唤，终于到了周六下午。被感情冲昏大脑的 GG 突然间变的不再那么笨了，这次他提前预定了座位，是一个可以容纳 8 个人的座位。而且具体告诉了 MM 座位的位置，这样大家都清楚位置是比较好的，避免了到时候没有位置的尴尬。赶往麦当劳路上的 GG 心潮澎湃但是有些担心，毕竟要面对六个美女，而且女朋友也是刚认识几天。“亲爱的，现在到哪了？”手机中 MM 发过来了一条短信，GG 一看时间，天啊，光顾着去傻想，还有几分钟就五点了，第一次如果都迟到那就太不好了，于是立即回复到，“宝贝儿，我就到了！”，因为麦当劳就在对面，抬头就可以看到的。GG 跑上了麦当劳的二楼的用餐处，见到诸位美女，紧张的还没说不话来，“这是我男朋友”MM 拽着 GG 的手臂说，“大家好，大家好”，GG 紧张的说道。忙又补充到：“我们先点餐，大家自便，都不要客气啊”，“我要吃鸡翅”，“我要麦香鱼套餐”，我要“板烧鸡腿套餐”，我要“奶昔”，我要“薯条”，“我要汉堡”，“我也要汉堡”，“我和她们一眼都要要汉堡”，无语啊，GG 当然不可能知道每个人的口味，无奈之下，只好把这三个美女带到服务台前，和服务员说，除了要订单上的东西外，在要三个汉堡，至于是什么味道的汉堡，就直接让这三个美女自己和服务员交流了

工厂方法法模式模式解释：

工厂方法模式（Factory Method Pattern）同样属于类的创建型模式又被称为多态工厂模式(Polymorphic) 。工厂方法模式的意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。

英文定义为：Define an interface for creating an object, but let subclassesdecide which class to instantiate. Factory Method lets a class deferinstantiation to subclasses.

工厂方法模式的 UML 图：

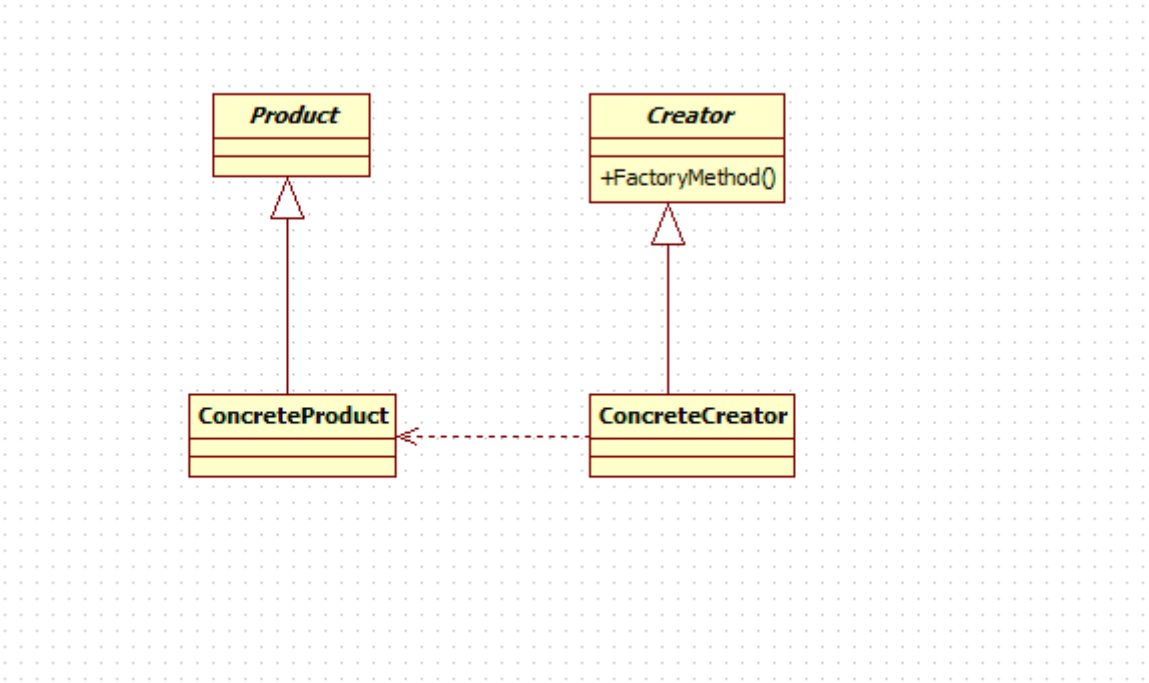
工厂方法模式中包含的角色及其相应的职责如下：

抽象工厂（Creator）角色：工厂方法模式的核心，任何工厂类都必须实现这个接口。

具体工厂（ Concrete Creator）角色：具体工厂类是抽象工厂的一个实现，负责实例化产品对象。

抽象（Product）产品角色：简单工厂模式所创建的所有对象的父类，注意，这里的父类可以是接口也可以是抽象类，它负责描述所有实例所共有的公共接口。

具体产品（Concrete Product）角色：简单工厂所创建的具体实例对象，这些具体的产品往往都拥有共同的父类。

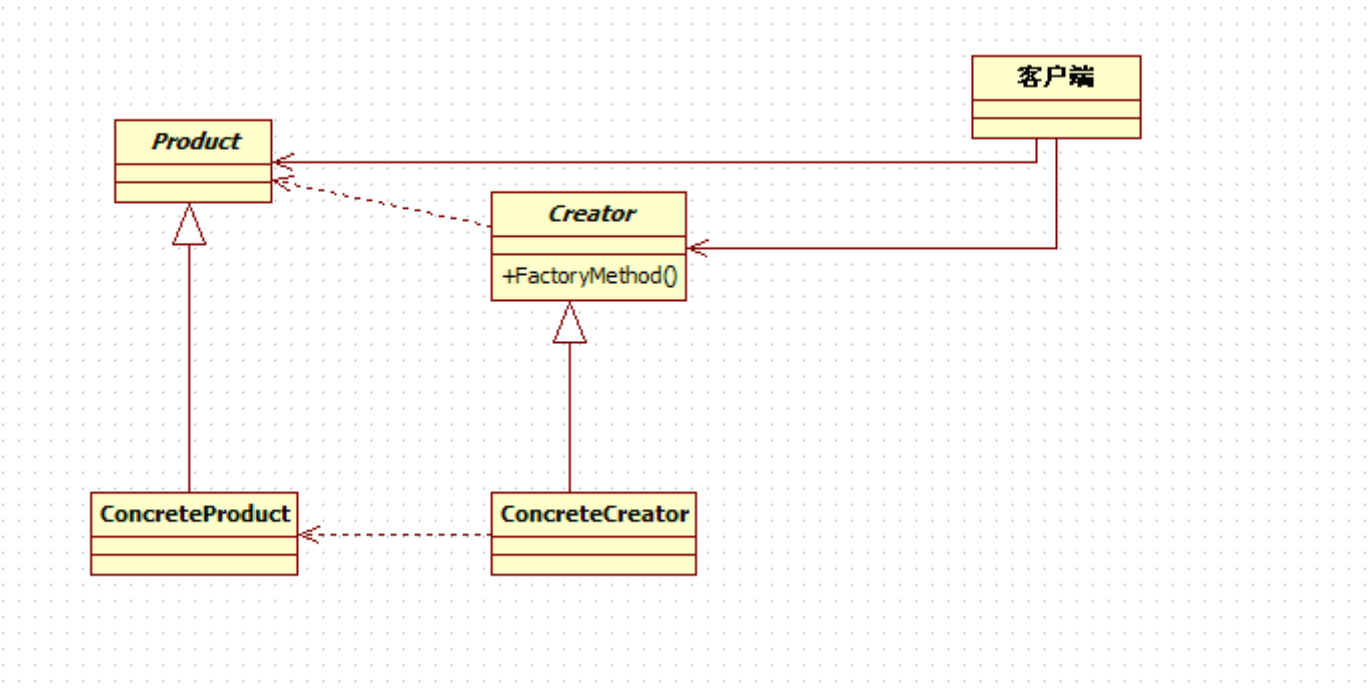


工厂方法法模式深入分析：

简单工厂模式使用一个类负责所有产品的创建，虽然使得客服端和服务端相互分离，使得客服端不用关心产品的具体创建过程，客户端唯一所要做的就是调用简单工厂的静态方法获得想要的产品即可。但是，简单工厂模式违背了严格意义上的"开放封闭原则"，这就使得一旦有一个新的产品增加就必须修改工厂类的源代码，从而将新的产品的创建逻辑加入简单工厂中供客户端调用。工厂方法法模式正是在简单工厂模式的基础上进一步抽象而来的。由于工厂方法法模式的核心是抽象工厂角色，使用了面向对象的多态性，这就使得工厂方法法模式即保持了简单工厂模式的优点，又克服了简单工厂模式违背"开放封闭原则"的缺点。

工厂方法法模式中核心工厂类不再提供所有的产品的创建工作，而是将具体的产品创建工作交给了子类去做。这时候的核心工厂类做什么呢？做标准！核心工厂类只需要负责制定具体工厂需要实现的接口即可，至于具体的工作就留给了子类去创建了。

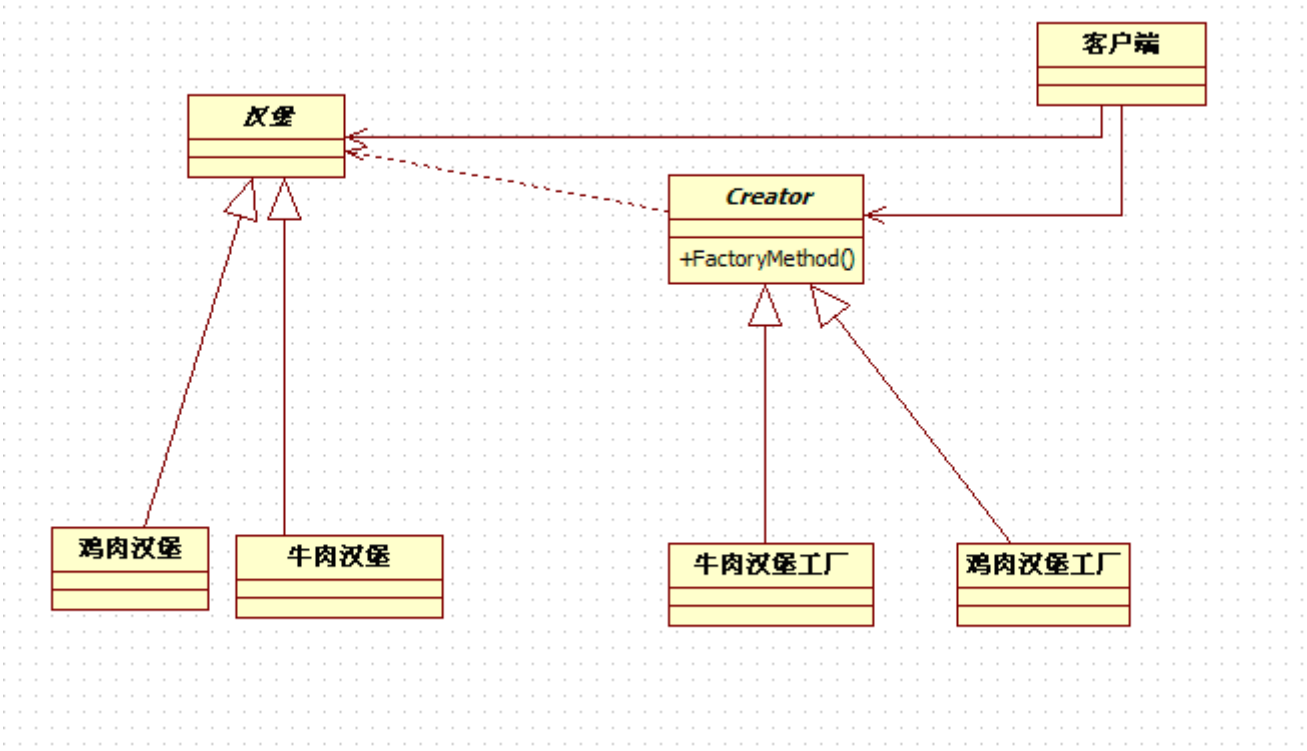
在实际的的使用中，抽闲产品和具体产品之间往往是多层次的产品结构，如下图所示：



工厂方法模式使用场景分析及代码实现：

不同的美女同时要吃汉堡，但是不同的美女的有不同的口味，这是 GG 是不知道每个要吃汉堡美女的口味的，而且也没有必要知道，如果真的知道了，自己的女朋友肯定会很不高兴的~~~~(>_<)~~~~ 。这时候 GG 只需要带着这些美女到服务台那里，让这些美女直接和服务人员讲自己具体需要什么口味汉堡就 OK 了。但是，不管这些妹妹需要的具体是什么汉堡，结果一定还是一个汉堡。

UML 模型图如下所示：



具体实现代码如下：

新建立一个食物的接口：

```
package com.diermeng.designPattern.FatoryMethod;
/*
 * 食物的抽象接口
 */
public interface Food {
    /*
     * 获取相应的食物
     */
    public void get();
}
```

新建一个食物工厂的接口：

```
package com.diermeng.designPattern.FatoryMethod;

/*
 * 食物的生产工厂的抽象接口
 */
public interface FoodFactory {
      /*
       * 生产工厂生产出相应的食物
       */
      public Food getFood();
}
```

接下来建立具体的产品：牛肉汉堡和鸡肉汉堡

```
package com.diermeng.designPattern.FatoryMethod.impl;
import com.diermeng.designPattern.FatoryMethod.Food;

/*
 * 牛肉汉堡
 */
public class BeefBurger implements Food{
    /*
     * 获取牛肉汉堡
     */
    public void get(){
        System.out.println("获得一份牛肉汉堡");
    }
}
```

```
package com.diermeng.designPattern.FatoryMethod.impl;
import com.diermeng.designPattern.FatoryMethod.Food;

/*
 * 鸡肉汉堡
 */
public class ChickenBurger implements Food{
    /*
     * 获取鸡肉汉堡
```

```
    */
    public void get(){
        System.out.println("获取一份鸡肉汉堡");
    }
}
```

新建一个牛肉汉堡工厂：

```
package com.diermeng.designPattern.FatoryMethod.impl;
import com.diermeng.designPattern.FatoryMethod.Food;
import com.diermeng.designPattern.FatoryMethod.FoodFactory;

/*
 *牛肉汉堡工厂
 */
public class BeefBurgerFactory implements FoodFactory {
    /*
     * 生产出一份牛肉汉堡
     * @see com.diermeng.designPattern.FatoryMethod.FoodFactory#getFood()
     */
    public Food getFood() {
        return new BeefBurger();
    }

}
```

新建一个鸡肉汉堡工厂：

```
package com.diermeng.designPattern.FatoryMethod.impl;
import com.diermeng.designPattern.FatoryMethod.Food;
import com.diermeng.designPattern.FatoryMethod.FoodFactory;

/*
 * 肌肉汉堡工厂
 */
public class ChickenBurgerFactory implements FoodFactory {
    /*
     * 生产出一份肌肉汉堡
```

```
     * @see com.diermeng.designPattern.FatoryMethod.FoodFactory#getFood()
     */
    public Food getFood() {
        return new ChickenBurger();
    }

}
```

最后我们建立测试客户端：

```
package com.diermeng.designPattern.FatoryMethod.client;
import com.diermeng.designPattern.FatoryMethod.Food;
import com.diermeng.designPattern.FatoryMethod.FoodFactory;
import com.diermeng.designPattern.FatoryMethod.impl.BeefBurgerFactory;
import com.diermeng.designPattern.FatoryMethod.impl.ChickenBurgerFactory;



/*
 * 测试客户端
 */
public class FactoryMethodTest {
    public static void main(String[] args) {
        //获得 ChickenBurgerFactory
        FoodFactory chickenBurgerFactory = new ChickenBurgerFactory();

        //通过 ChickenBurgerFactory 来获得 ChickenBurger 实例对象
        Food ChickenBurger = chickenBurgerFactory.getFood();
        ChickenBurger.get();



        //获得 BeefBurgerFactory
        FoodFactory beefBurgerFactory = new BeefBurgerFactory();

        //通过 BeefBurgerFactory 来获得 BeefBurger 实例对象
        Food beefBurger = beefBurgerFactory.getFood();
        beefBurger.get();

    }
}
```

```

```

输出的结果如下：

```
获取一份鸡肉汉堡
获得一份牛肉汉堡
```

工厂方法模式的优缺点分析：

优点：工厂方法类的核心是一个抽象工厂类，所有具体的工厂类都必须实现这个接口。当系统扩展需要添加新的产品对象时，仅仅需要添加一个具体对象以及一个具体工厂对象，原有工厂对象不需要进行任何修改，也不需要修改客户端，这就很好的符合了"开放－封闭"原则。

缺点：使用工厂方法模式的时候，客户端需要决定实例化哪一个具体的工厂。也就是说工厂方法法模式把简单工厂模式的内部判断逻辑转移到了客户端代码。而且使用该模式需要增加额外的代码，这就导致工作量的增加。

工厂方法模式的实际应用简介：

工厂方法模式解决的是同一系列的产品的创建问题，而且很好的满足了"开放封闭原则"，这需要创建同一系列产品的时候，使用工厂方法模式往往比使用简单工厂模式更好，尤其是对大型复杂的系统而言。

温馨提示：

工厂方法法模式解决了简单工厂模式的不足。在增加一个产品时，只需要一个具体的产品和创建产品的工厂类就可以，具有非常好的可维护性。但是也正因为如此，增加了代码量和工作量。不过瑕不掩瑜，因为增加的代码量不是很大的。

抽象工厂模式 MM 的生日

抽象工厂模式应用场景举例：

时光甜蜜的飞逝，GG 和 MM 过着童话般的王子和公主的浪漫的生活。眼看 MM 生日就要到了，GG 着急了。毕竟，这是自己的第一个女朋友的第一个生日啊。想了千万种方法，问了身边很多朋友，这个傻 GG 最终还是没有确定最终该如何去做~~~~(>_<)~~~~

哎！爱，总是想到太多做的太少^_^

都快夜里十二点了，GG 还在 Google 和百度上面查询如何给自己的 Sweatheart 过生日。此时，突然手机短信铃声响了，打开一看，上面写道："亲爱的，我知道这些天你一直在想我们如何一切过生日，其实，一切都很简单的。简单就好。"，看完短信，GG 顿时全身暖流涌动，感觉好幸福^_^，有如此体贴理解人的 MM，夫复何求(⊙ o ⊙)啊！刚要回复短信，手机铃声又响了，上面写道："我们还去麦当劳吧，不过这次使我们俩，要换一个地方，到华联那边的麦当劳吧^_^"，GG 读着短信，感动的无语了。短信回复道："一切惟老婆大人之命是从:-O"。GG 和 MM 都沉浸在甜蜜和幸福中^_^

抽象工厂模式解释：

抽象工厂模式（Abstact Factory Pattern）是所有形态的工厂模式中最为抽象和最其一般性的。抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，能够创建多个产品族的产品对象。

抽象工厂中方法对应产品结构，具体工厂对应产品族

英文定义为：Provide an interface forcreating families of related or dependent objects without specifying theirconcrete classes.

抽象工厂模式的 UML 图：

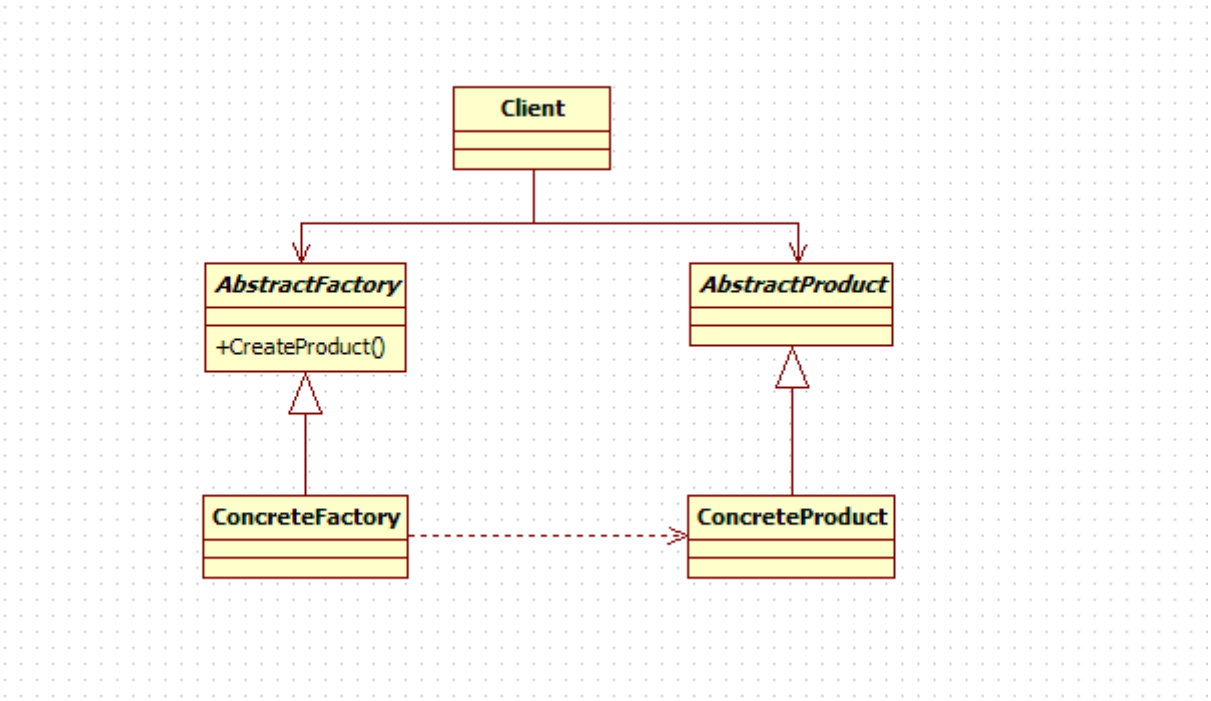
抽象工厂模式模式中包含的角色及其相应的职责如下：

抽象工厂（Creator）角色：抽象工厂模式的核心，包含对多个产品结构的声明，任何工厂类都必须实现这个接口。

具体工厂（Concrete Creator）角色：具体工厂类是抽象工厂的一个实现，负责实例化某个产品族中的产品对象。

抽象（Product）产品角色：抽象模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

具体产品（Concrete Product）角色：抽象模式所创建的具体实例对象。



抽象工厂模式深入分析：

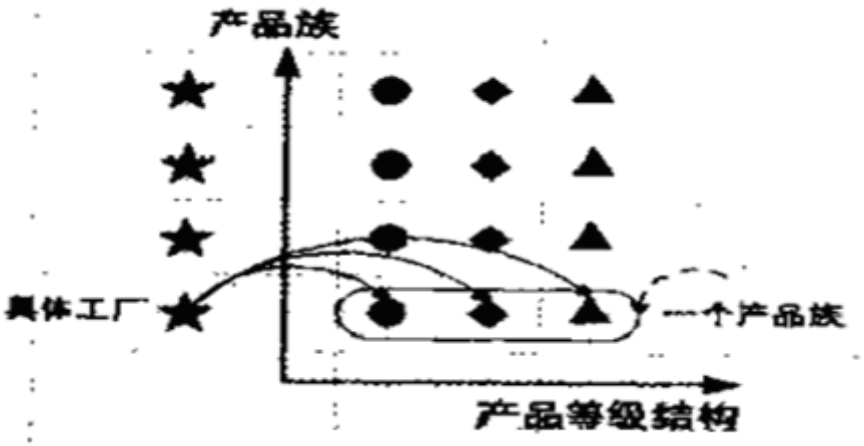
抽象工厂模式是在当产品有多个 抽象角色的时候使用的一种创建型设计模式。

按照里氏代换原则，凡是父类适用的地方，子类也必然适用。而在实际系统中，我们需要的是和父类类型相同的子类的实例对象，而不是父类本身，也就是这些抽象产品的具体子类的实例。具体工厂类就是来负责创建抽象产品的具体子类的实例的。

当每个抽象产品都有多于一个的具体子类的时候，工厂角色是如何确定实例化哪一个子类呢？例如说有两个抽象产品角色，而每个抽象产品角色都有两个具体产品。抽象工厂模式提供两个具体工厂角色，分别对应于这两个具体产品角色，每一个具体工厂角色只负责某一个产品角色的实例化。每一个具体工厂类只负责创建抽象产品的某一个具体子类的实例。

每一个模式都是针对一定问题的解决方案，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式针对的是多个产品等级结构。

何谓产品族？产品族是指位于不同产品等级结构中，功能相关联的产品组成的家族。一般是位于不同的等级结构中的相同位置上。显然，每一个产品族中含有产品的数目，与产品等级结构的数目是相等的，形成一个二维的坐标系，水平坐标是产品等级结构，纵坐标是产品族。

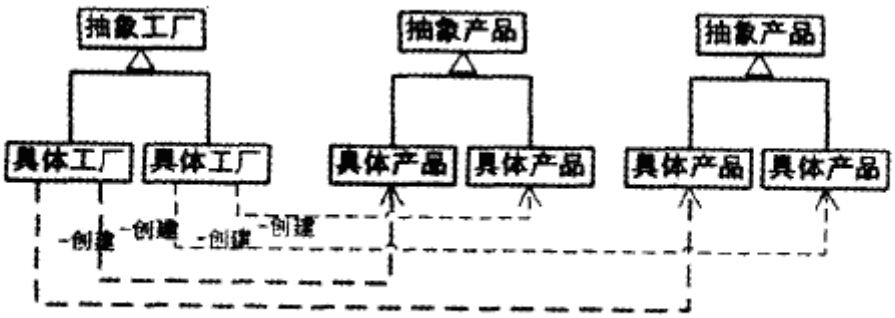


对于每一个产品族，都有一个具体工厂。而每一个具体工厂创建属于同一个产品族，但是分属于不同等级结构的产品。

通过引进抽象工厂模式，可以处理具有相同（或者相似）等级结构的多个产品族中的产品对象的创建问题。

由于每个具体工厂角色都需要负责不同等级结构的产品对象的创建，因此每个工厂角色都需要提供相应数目的工厂方法，分别用于创建相应数目的等级结构的产品。

如下图所示：

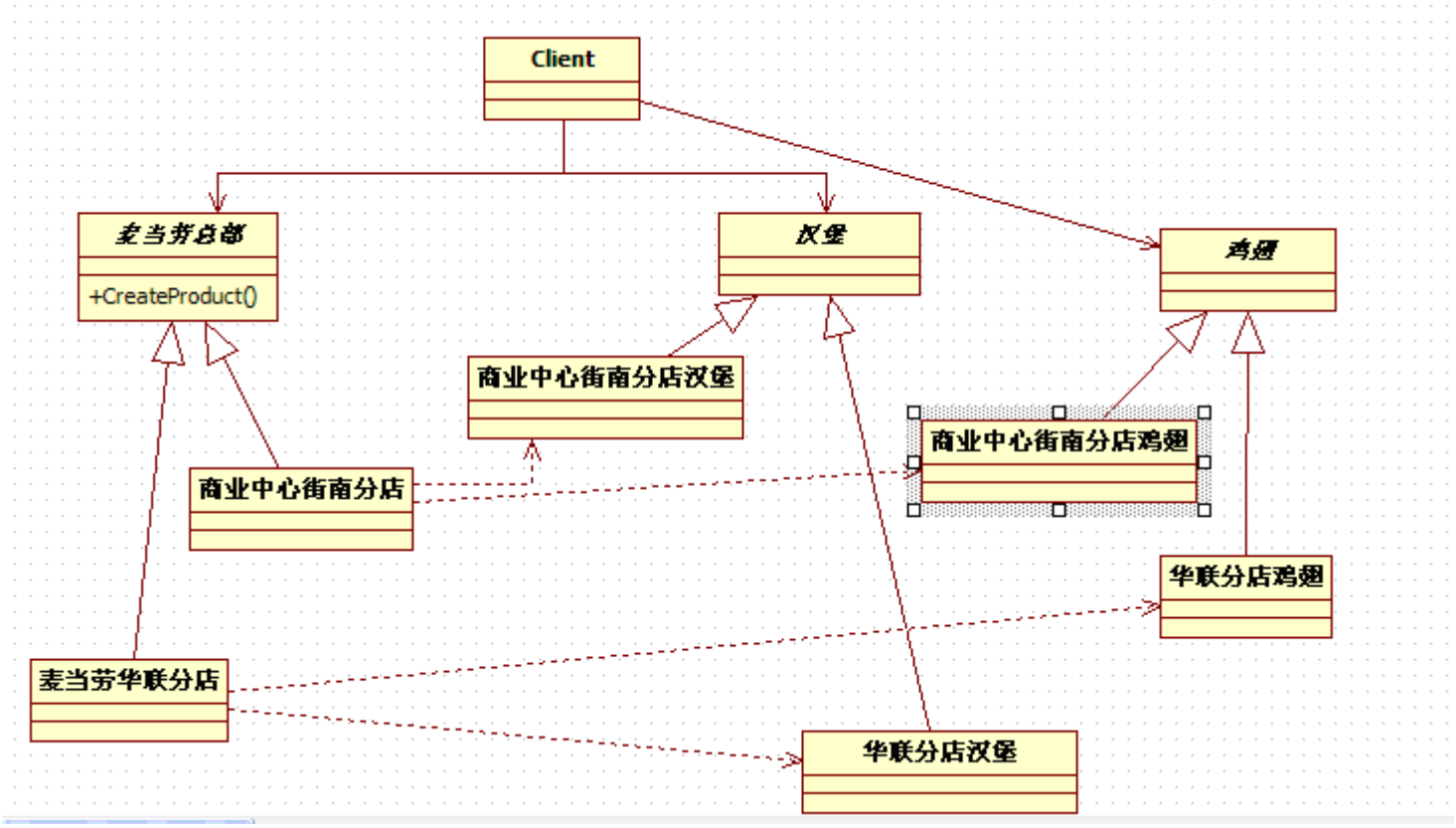


抽象工厂模式使用场景分析及代码实现：

MM 过生日的时候还是要到麦当劳，但是这次要求是到华联那边的麦当劳去，就是地方不同了，要换换口味和心情。这就是抽象工厂模式的一个很好的体现。首先对不同的麦当劳分店而言，每一种产品，例如说汉堡，都是汉堡，但是每个地方的汉堡在遵循统一标准的前提下又会尽力突出自己的特色，这样这样才能更好的吸引和留住顾客，因为不同的地方，随着环境等的不同，人们的喜好和口味等都会有所不同，但是无论怎么不同，始终还是汉堡，具有汉堡的基本功能。同时，每一个分店都有一系列的产品，例如汉堡、鸡翅等等，这就构成了产品的等级结构。

总之：麦当劳总部相当于抽象工厂，每个分店相当于具体工厂，而每种产品又有所不同。这样在既保持了统一性的前提下，又使得各分店的特色有所不同，适合于吸引和留住不同环境下的客户。

UML 模型图如下所示：



具体实现代码如下：

新建立一个食物的接口：

```
package com.diermeng.designPattern.AbstractFactory;
/*
 * 所有食物的接口
 */
public interface Food {
    /*
     * 获取食物的方法
     */
    public void get();
}
```

新建一个麦当劳总店的接口：

```
package com.diermeng.designPattern.AbstractFactory;


/*
 * 麦当劳总店
 */
```

```
public interface FoodFactory {
    //实例化汉堡
    public Food getHamburg();
    //实例化鸡翅
    public Food getChickenWing();
}
```

建立汉堡的抽象基类

```
package com.diermeng.designPattern.AbstractFactory;

/*
 * 汉堡的抽象父类
 */
public abstract class Hamburg implements Food{
    /*
     * 获取汉堡的方法
     */
    public abstract void get();
}
```

建立鸡翅的抽象基类

```
package com.diermeng.designPattern.AbstractFactory;

/*
 * 鸡翅的抽象类
 */
public abstract class ChickenWing implements Food{
    /*
     * 获取鸡翅的方法
     */
    public abstract void get();
}
```

建立中心商业街南部的麦当劳分店

```
package com.diermeng.designPattern.AbstractFactory.impl;
import com.diermeng.designPattern.AbstractFactory.Food;
```

```
import com.diermeng.designPattern.AbstractFactory.FoodFactory;


/*
 * 中心商业街南边的麦当劳分店
 */
public class SouthMacDonald implements FoodFactory {
    /*
     * 获取汉堡
     * @see com.diermeng.designPattern.AbstractFactory.FoodFactory#getHamburg()
     */
    public Food getHamburg() {
        return new SouthMacDonaldHamburg();
    }
    /*
     * 获取鸡翅
     * @see com.diermeng.designPattern.AbstractFactory.FoodFactory#getChickenWing()
     */
    public Food getChickenWing() {
        return new SouthMacDonaldChickenWing();
    }

}
```

建立华联那边麦当劳分店

```
package com.diermeng.designPattern.AbstractFactory.impl;
import com.diermeng.designPattern.AbstractFactory.Food;
import com.diermeng.designPattern.AbstractFactory.FoodFactory;


/*
 * 麦当劳的华联分店
 */
public class HualianMacDonald implements FoodFactory {
    /*
     * 获取汉堡
     * @see com.diermeng.designPattern.AbstractFactory.FoodFactory#getHamburg()
     */
    public Food getHamburg() {
        return new HualianMacDonaldHamburg();
```

```
    }

    /*
     * 获取鸡翅
     * @see com.diermeng.designPattern.AbstractFactory.FoodFactory#getChickenWing()
     */
    public Food getChickenWing() {
        return new HualianMacDonaldChickenWing();
    }

}
```

建立中心商业街南边的麦当劳的汉堡：

```
package com.diermeng.designPattern.AbstractFactory.impl;
import com.diermeng.designPattern.AbstractFactory.Hamburg;

/*
 * 中心商业街南边的的麦当劳分店的汉堡
 */
public class SouthMacDonaldHamburg extends Hamburg {
    /*
     * 获取汉堡
     * @see com.diermeng.designPattern.AbstractFactory.Hamburg#get()
     */
    public void get() {
        System.out.println("获取中心商业街南边的的麦当劳分店的汉堡");
    }

}
```

建立华联那边的麦当劳的汉堡：

```
package com.diermeng.designPattern.AbstractFactory.impl;
import com.diermeng.designPattern.AbstractFactory.Hamburg;

/*
 * 华联那边的麦当劳分店的汉堡
 */
```

```
public class HualianMacDonaldHamburg extends Hamburg {
    /*
     * 获取汉堡
     * @see com.diermeng.designPattern.AbstractFactory.Hamburg#get()
     */
    public void get() {
        System.out.println("获取华联那边的麦当劳分店的汉堡");
    }

}
```

建立中心商业街南边的麦当劳鸡翅

```
package com.diermeng.designPattern.AbstractFactory.impl;
import com.diermeng.designPattern.AbstractFactory.ChickenWing;

/*
 * 中心商业街南边的的麦当劳分店的鸡翅
 */
public class SouthMacDonaldChickenWing extends ChickenWing {
    /*
     * 获取鸡翅
     * @see com.diermeng.designPattern.AbstractFactory.ChickenWing#get()
     */
    public void get() {
        System.out.println("获取中心商业街南边的的麦当劳分店的鸡翅");
    }

}
```

建立华联那边的麦当劳的鸡翅

```
package com.diermeng.designPattern.AbstractFactory.impl;
import com.diermeng.designPattern.AbstractFactory.ChickenWing;

/*
 * 华联那边的麦当劳分店的鸡翅
 */
```

```
public class HualianMacDonaldChickenWing extends ChickenWing {
    /*
     * 获取鸡翅
     * @see com.diermeng.designPattern.AbstractFactory.ChickenWing#get()
     */
    public void get() {
        System.out.println("获取华联那边的麦当劳分店的鸡翅");
    }

}
```

最后我们建立测试客户端：

```
package com.diermeng.designPattern.AbstractFactory.client;

import com.diermeng.designPattern.AbstractFactory.Food;
import com.diermeng.designPattern.AbstractFactory.FoodFactory;
import com.diermeng.designPattern.AbstractFactory.impl.HualianMacDonald;
import com.diermeng.designPattern.AbstractFactory.impl.SouthMacDonald;

/*
 * 测试客户端
 */
public class AbstractFactoryTest {
    public static void main(String[] args) {



        //声明并实例化中心商业街南边的的麦当劳分店
        FoodFactory southMacDonald= new SouthMacDonald();
        //获取中心商业街南边的的麦当劳分店的汉堡
        Food southMacDonaldHamburg = southMacDonald.getHamburg();
        southMacDonaldHamburg.get();
        //获取中心商业街南边的的麦当劳分店的鸡翅
        Food southMacDonaldChickenWing = southMacDonald.getChickenWing();
        southMacDonaldChickenWing.get();

        //声明并实例化华联那边的麦当劳分店
```

```
        FoodFactory hualianMacDonald = new HualianMacDonald();
        //获取华联那边的麦当劳分店的汉堡
        Food hualianMacDonaldHamburg =hualianMacDonald.getHamburg();
        hualianMacDonaldHamburg.get();
        //获取华联那边的麦当劳分店的鸡翅
        Food hualianMacDonaldChickenWing = hualianMacDonald.getChickenWing();
        hualianMacDonaldChickenWing.get();
    }
}
```

输出的结果如下：

```
获取中心商业街南边的的麦当劳分店的汉堡
获取中心商业街南边的的麦当劳分店的鸡翅
获取华联那边的麦当劳分店的汉堡
获取华联那边的麦当劳分店的鸡翅
```

抽象工厂模式的优缺点分析：

优点：客户端不再负责对象的具体创建，而是把这个责任交给了具体的工厂类，客户端之负责对对象的调用。当具有产品家族性质的产品被涉及到一个工厂类中后，对客户端而言是非常友好的，更重要的是如果想要更换为另外一产品家族，所要做的只是需要增加相应的产品家族成员和增加一个具体的产品工厂而已。

缺点：当有新的产品加入的时候，也就是当产品的结构发生改变时，修要修改抽象工厂类的设计，这就导致了必须修改所有的具体工厂类，导致很客观的工作量的增加。

抽象工厂模式的实际应用简介：

抽象工厂模式是针对多个产品系列的创建的问题，这在持久化层的设计很实现中有很大的指导意义。由于 Java 的跨平台性，一般而言，持久化层都要考虑到都肿数据库的问题，例如 MySQL、Oracle 等，每一个数据库就相当于一个产品系列，持久化层必须设计好好不同产品系列的共同接口，这样才便于使用者操作数据库，同时也有利于数据库的移植。大名鼎鼎的 Hibernate 就很好的借鉴了抽象工厂模式的设计方法。

温馨提示：

抽象工厂模式考虑的是不同产品系列的创建的问题，并非能到处使用。另外在新增加产品的时候，需要改变抽象工厂的设计，这会导致很大的工作量，所以在规划之初必须考虑好产品的结构，力求降低参加产品的可能性，是抽象工厂比较稳定。

单例模式 你是我的唯一

单例模式应用场景举例：

“曾经沧海难为水，除却巫山不是云”，这句话用现在的语言解释就是“你是我的唯一”。

GG 和 MM 都是初次恋爱，都把对方视为自己此生的唯一。而且 GG 和 MM 都在不断的向对方学习，不断的完善自己。GG 和 MM 的甜蜜和幸福很快就轰动了整个院系。男生一般都拿 GG 的女朋友教育自己的女朋友说别人怎么怎么样，而女生也经常拿 MM 的男朋友说男生该如何如何做。而且，年级辅导员还在年级会上表扬了 GG 和 MM，说男生都应该想 MM 的男朋友学习，女生都应该向 GG 的女朋友学习！呵呵，很显然，大家都知道，辅导员说 GG 的女朋友就是指 MM，而说 MM 的男朋友时就是指 GG。

单例模式解释：

GoF 对单例模式(Singleton Pattern)的定义是：保证一个类、只有一个实例存在，同时提供能对该实例加以访问的全局访问方法。

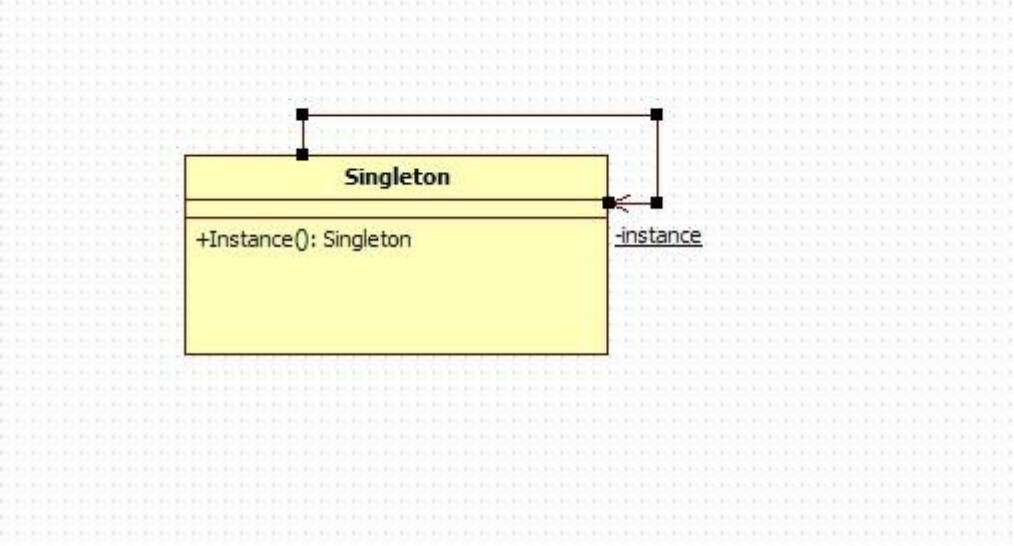
单例模式是一种对象创建型模式，使用单例模式，可以保证为一个类只生成唯一的实例对象。也就是说，在整个程序空间中，该类只存在一个实例对象。

单例模式的要点有三个；一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

英文定义为：Ensure a class only has one instance, and provide a global point of access to it.

单例模式的 UML 图：

单例模式比较的单纯，其 UML 图如下所示：



单例模式深入分析：

单例模式的要点有三个；一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

单例模式适合于一个类只有一个实例的情况，比如窗口管理器，打印缓冲池和文件系统，它们都是原型的例子。典型的情况是，那些对象的类型被遍及一个软件系统的不同对象访问，因此需要一个全局的访问指针，这便是众所周知的单例模式的应用。当然这只有在你确信你不再需要任何多于一个的实例的情况下

在计算机系统中，需要管理的资源包括软件外部资源，譬如每台计算机可以有若干个打印机，但只能有一个 Printer Spooler， 以避免两个打印作业同时输出到打印机中。每台计算机可以有若干传真卡，但是只应该有一个软件负责管理

传真卡，以避免出现两份传真作业同时传到传真卡中的情况。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。

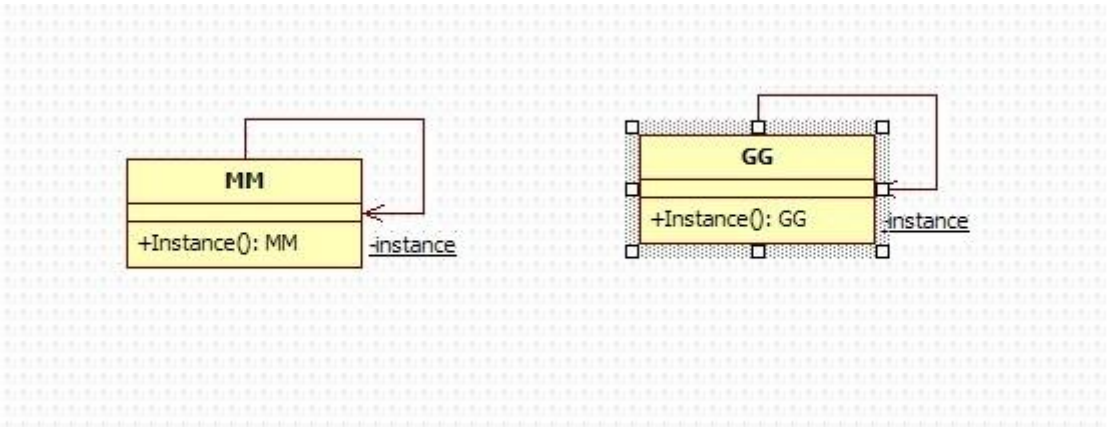
需要管理的资源也包括软件内部资源，譬如，大多数的软件都有一个（甚至多个）属性（properties）文件存放系统配置。这样的系统应当由一个对象来管理一个属性文件。

需要管理的软件内部资源也包括譬如负责记录网站来访人数的部件，记录软件系统内部事件、出错信息的部件，或是对系统的表现进行检查的部件等。这些部件都必须集中管理。

单例模式使用场景分析及代码实现：

在上面的使用场景中，无论是谁叫 GG 的女朋友，大家都知道只的是 MM；而相应的，无论是谁说 MM 的男朋友，大家都知道是 GG。GG 和 MM 分别都是对方单例 O(∩_∩)O 哈哈~

UML 模型图如下所示：



笔者在这里以 MM 的男朋友 GG 为例进行单例模式的说明。

GG 单例模式的第一个版本，采用的是"饿汉式"，也就是当类加载进来的就立即实例化 GG 对象，但是这种方式比较的消耗计算机资源。具体实现代码如下：

```
package com.diermeng.designPattern.Singleton;
/*
 * GG 单例模式的第一个版本 为"饿汉式"
 */
public class GGVersionOne {
    //在类被加载进入内存的时候就创建单一的 GG 对象
    public static final GGVersionOne gGVersionOne = new GGVersionOne();
    //名称属性
    private String name;


    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    //构造函数私有化
    private GGVersionOne() {
    }

    //提供一个全局的静态方法
    public static GGVersionOne getGG() {
        return gGVersionOne;
    }
}
```

GG 单例模式的第二个版本：“懒汉式”，在单线程下能够非常好的工作，但是在多线程下存在线程安全问题，具体代码如下：

```
package com.diermeng.designPattern.Singleton;
/*
 * GG 单例模式的第二个版本 采用“懒汉式” 在需要使用的时候才实例化 GG
 */
public class GGVersionTwo {
    //GG 的姓名
    private String name;
    //对单例本身引用的名称
    private static GGVersionTwo gGVersionTwo;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    //构造函数私有化
    private GGVersionTwo() {
    }

    //提供一个全局的静态方法
    public static GGVersionTwo getGG() {
        if(gGVersionTwo == null) {
```

```
            gGVersionTwo = new GGVersionTwo();
        }
        return gGVersionTwo;
    }
}
```

GG 单例模式的第三个版本，为解决多线程问题，采用了对函数进行同步的方式，但是比较浪费资源，因为每次都要进行同步检查，而实际中真正需要检查只是第一次实例化的时候，具体代码如下所示：

```
package com.diermeng.designPattern.Singleton;
/*
 * GG 单例模式的第三个版本 对函数进行同步
 */
public class GGVersionThree {
    //GG 的姓名
    private String name;
    //对单例本身引用的名称
    private static GGVersionThree gGVersionThree;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    //构造函数私有化
    private GGVersionThree() {
    }

    //提供一个全局的静态方法，使用同步方法
    public static synchronized GGVersionThree getGG() {
        if(gGVersionThree == null) {
            gGVersionThree = new GGVersionThree();
        }
        return gGVersionThree;
    }
}
```

GG 单例模式第四个版本，既解决了“懒汉式的”多线程问题，又解决了资源浪费的现象，看上去是一种不错的选择，具体代码如下所示：

```
package com.diermeng.designPattern.Singleton;
/*
 * GG 单例模式的第四个版本，既解决了“懒汉式的”多线程问题，又解决了资源浪费的现象，看上去是一种不错的选择
 */
public class GGVersionFour {
    //GG 的姓名
    private String name;
    //对单例本身引用的名称
    private static GGVersionFour gGVersionFour;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    //构造函数私有化
    private GGVersionFour() {
    }

    //提供一个全局的静态方法
    public static GGVersionFour getGG() {
        if(gGVersionFour == null) {
            synchronized (GGVersionFour.class) {
                if(gGVersionFour == null) {
                    gGVersionFour = new GGVersionFour();
                }
            }

        }
        return gGVersionFour;
    }
}
```

最后我们建立测试客户端测试一下版本四：

```
package com.diermeng.designPattern.Singleton.client;
import com.diermeng.designPattern.Singleton.GGVersionFour;

/*
 * 测试客户端
 */
public class SingletonTest {
    public static void main(String[] args) {
        //实例化
        GGVersionFour gG1 = GGVersionFour.getGG();
        GGVersionFour gG2 = GGVersionFour.getGG();
        //设值
        gG1.setName("GGAlias");
        gG2.setName("GG");

        System.out.println(gG1.getName());
        System.out.println(gG2.getName());


    }
}
```

输出的结果如下：

```
GG
GG
```

单例模式的优缺点分析：

优点：客户端使用单例模式类的实例的时候，只需要调用一个单一的方法即可生成一个唯一的实例，有利于节约资源。

缺点：首先单例模式很难实现序列化，这就导致采用单例模式的类很难被持久化，当然也很难通过网络传输；其次由于单例采用静态方法，无法在继承结构中使用。最后如果在分布式集群的环境中存在多个 Java 虚拟机的情况下，具体确定哪个单例在运行也是很困难的事情。

单例模式的实际应用简介：

单例模式一般会出现在以下情况下：

在多个线程之间，比如 servlet 环境，共享同一个资源或者操作同一个对象

在整个程序空间使用全局变量，共享资源

大规模系统中，为了性能的考虑，需要节省对象的创建时间等等。

温馨提示：

细心的读者可能会发现，笔者在写单例模式的双重检查方式的使用了"看上去是一种不错的选择"之语，之所以样说，是因为：Java 的线程工作顺序是不确定的，这就会导致在多线程的情况没有实例化就使用的现象，进而导致程序崩溃。不过双重检查在 C 语言中并没有问题。因为大师说：双重检查对 Java 语言并不是成立的。尽管如此，双重检查仍然不失为解决多线程情况下单例模式的一种理想的方案。

原型模式 肉麻情话

原型模式应用场景举例：

GG 和 MM 经常在 QQ 上聊天，但是 GG 打字的速度慢如蜗牛爬行，每次 MM 在瞬间完成恢复或者问候是，GG 都会很紧张的去尽力快速打字，尽管如此，还是让 MM 有些不高心，MM 说回复信息这么慢，显然是用心不专，不在乎她。哎，GG 也是百口难辩啊，不过也确实是没有办法。

有一天，GG 想自己的密友 K 倾诉了自己的苦衷。K 顿生大笑。说道：“傻瓜，你怎么不去网上收集一些肉麻的情话以及一些你们经常说话会涉及到主题，把这些东西拷贝下来保存在自己的电脑或者 U 盘里面，这样一来如果下次在聊天就可以借用过来了！”，“K 就是 K，我怎么没有想到呢~妙极~妙极^_^”，“不过不要太高兴，这些东西是要适当修改的，要不然你把名字都搞错的话，就等着你的 MM 把你踹死吧 O(∩_∩)O 哈哈~”K 补充道，“嗯，说的对，谢谢 K 哥解决了我的心腹之患啊”GG 乐不可支的说道。

这是 MM 由在网上和 GG 聊天，GG 专门复制那些实现准备好的肉麻情话经过稍加修改后发给 MM，MM 都快美死了...

原型模式解释：

原型模式（Prototype Pattern）是一种对象创建型模式，它采取复制原型对象的方法来创建对象的实例。使用 Prototype 模式创建的实例，具有与原型一样的初始化数据

英文定义为：Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

原型模式的 UML 图：

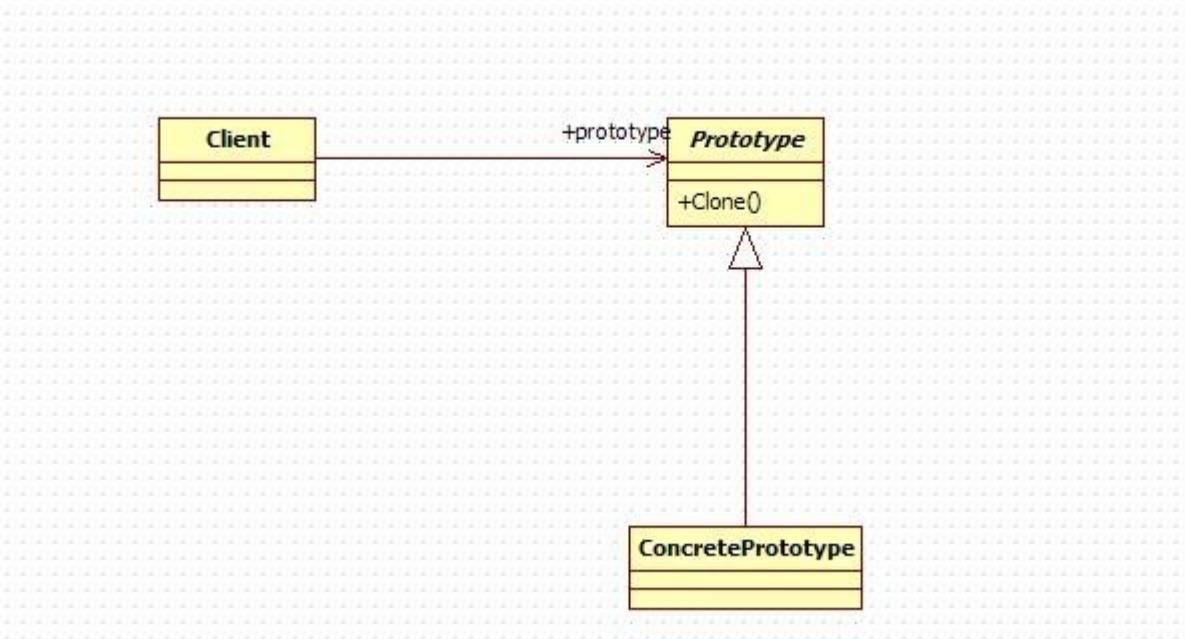
原型模式涉及以下的角色：

客户端（Client）角色：客户端提出创建对象的请求。

抽象原型（Prototype）角色：通常由一个 Java 接口或者 Java 抽象类来实现。从而为具体原型设立好规范。

具体原型（Concrete Prototype）角色：被复制的具体对象，此具体角色实现了抽象原型角色所要求实现的方法。

原型模式的 UML 图如下所示：



原型模式深入分析：

原型模式的工作原理是:通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。

Java 在语言级别是直接支持原型模式的。我们知道，在 java.lang.Object 是一切类和接口的父类，而 java.lang.Object 正好提供了一个 clone（）方法来支持原型模式。当然，一个对象如果想具有被复制的能力，还必须声明自己实现了 Cloneable 接口，如果没有声明，就会在对象被复制的时候抛出 CloneNotSupportedException.

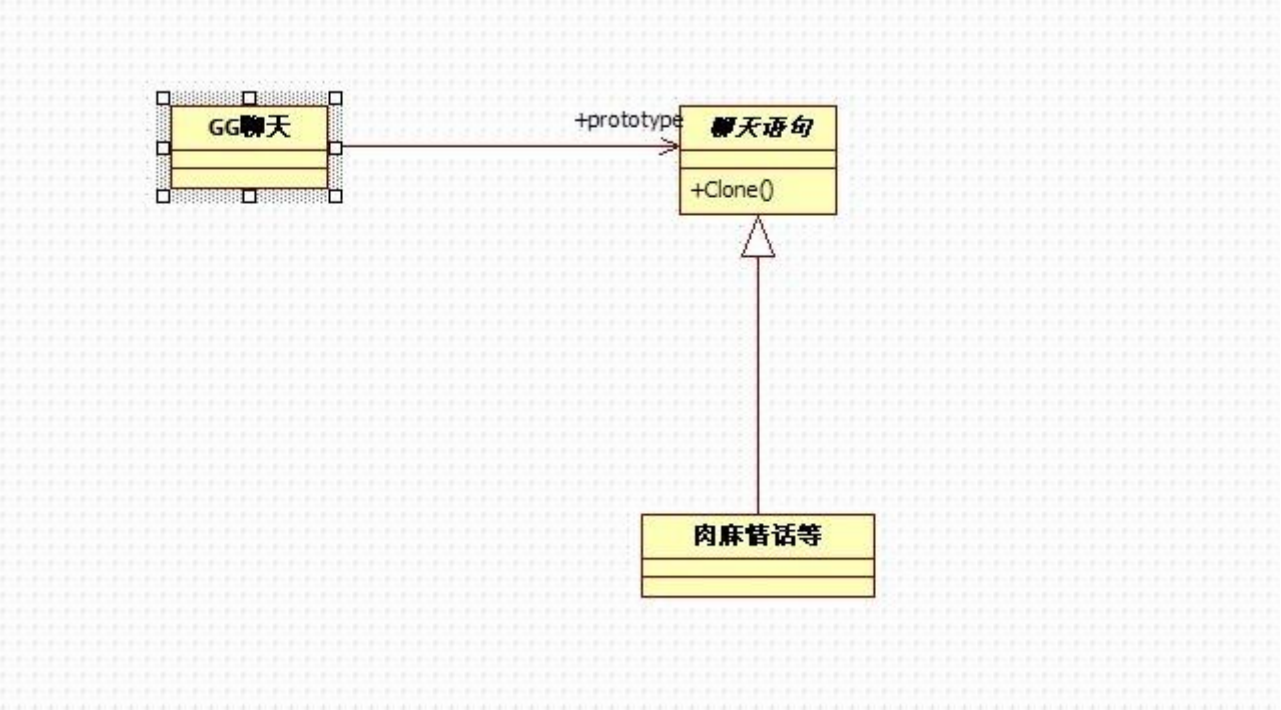
在 java.lang.Object 中提供了一个 protected Object clone()方法来支持对象的克隆，子类可以采用默认的方式进行所有字段的复制，也可以在子类中覆盖 clone()方便，根据实际需要定制自己的复制行为。

复制浅复制和深复制之分，浅复制是对基本数据类型和 String 类型而言的，深复制是对其他引用类型而言的。对于深复制，每一个应用也需要声明 Cloneable 接口。

原型模式使用场景分析及代码实现：

在上面的使用场景中，因为 GG 打字太慢经常被女朋友怪罪，所以有了拷贝网上肉麻情话的和主要聊天话题内容的办法。这样，以后 GG 每次和 MM 聊天的时候只需要把原话拷贝出来，加以适当修改就行，省时省力，而且效果绝佳^_^，这就是设计模式的原型模式的使用的好处 O(∩_∩)O~

UML 模型图如下所示：



建立一个肉麻情话类，类中有非常详细的注释，这里就不在解释了：

```
package com.diermeng.designPattern.Prototype.impl;
import java.util.ArrayList;
import java.util.List;
/*
 * 肉麻情话类
 */
public class SweetWord implements Cloneable{
   //肉麻情话句子
   private String content;
```

```
//肉麻情话句子集合
private List<String> contents;


/*
 * 获取肉麻情话集合
 */
public List<String> getContents() {
    return contents;
}
/*
 * 设置肉麻情话集合
 */
public void setContents(List<String> contents) {
    this.contents = contents;
}



/*
 * 获取肉麻情话
 */
public String getContent() {
    return content;
}
/*
 * 设置肉麻情话
 */
public void setContent(String content) {
    this.content = content;
}


/*
 * 肉麻情话覆盖了 Object 类的 clone()方法，因为这里有 List 引用进行深度复制
 * @see java.lang.Object#clone()
 */
public SweetWord clone() {
    try {
        //新建一个肉麻情话对象，同时复制基本的属性
        SweetWord sweetWord  = (SweetWord)super.clone();
        //新建一个肉麻情话集合
```

```
            List<String> newContents = new ArrayList<String>();
            //把原对象的肉麻情话集合中的肉麻情话集合通过 forEach 循环加入新建的 newContents 中
            for(String friend : this.getContents()) {
                newContents.add(friend);
            }
            //把新的肉麻情话集合设置进新的对象
            sweetWord.setContents(newContents);
            //返回新的的肉麻情话对象
            return  sweetWord;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

}
```

最后我们建立测试客户端：

```
package com.diermeng.designPattern.Prototype.client;

import java.util.ArrayList;
import java.util.List;

import com.diermeng.designPattern.Prototype.impl.SweetWord;

/*
 * 肉麻情话测试客户端
 */
public class PrototypeClient {
    public static void main(String[] args) {

        //新建一个肉麻情话对象并设置相应的属性
        SweetWord content1 = new SweetWord();
        List<String> contents = new ArrayList<String>();
        contents.add("宝贝儿，我爱你");
        contents.add("你是我的唯一");
```

```
        content1.setContents(contents);
        //复制 content1
        SweetWord content2 = content1.clone();
        //分别输入两个对象的内容
        System.out.println(content1.getContents());
        System.out.println(content2.getContents());

        //在原来的肉麻情话对象中加入新的内容并把新的内容设置进去
        contents.add("你是我真命天女");
        content1.setContents(contents);

        //分别输出新的修改后的两个肉麻情话对象
        System.out.println(content1.getContents());
        System.out.println(content2.getContents());
    }
}
```

输出的结果如下：

```
[宝贝儿，我爱你，你是我的唯一]
[宝贝儿，我爱你，你是我的唯一]
[宝贝儿，我爱你，你是我的唯一，你是我真命天女]
[宝贝儿，我爱你，你是我的唯一]
```

原型模式的优缺点分析：

优点：

1.允许动态地增加或减少产品类。由于创建产品类实例的方法是产品类内部具有的，因此增加新产品对整个结构没有影响。

2.提供简化的创建结构。

3.具有给一个应用软件动态加载新功能的能力。

4.产品类不需要非得有任何事先确定的等级结构，因为原型模式适用于任何的等级结构。

缺点：

每一个类都必须配备一个克隆方法，这对于全新的类来说不是很难，而对已有的类来说实现 clone()方法不一定很容易，而且在进行比较深层次的复制的时候也需要编写一定工作量的代码

原型模式的实际应用简介：

原型对象一般在适用于一下场景：

在创建对象的时候，我们不仅希望被创建的对象继承其类的基本机构，而且还希望继承原型对象的数据。

希望对目标对象的修改不影响既有的原型对象（深度克隆的时候可以完全互不影响）。

隐藏克隆操作的细节。很多时候，对对象本身的克隆需要涉及到类本身的数据细节。

温馨提示：

因为使用原型模式的时候每个类都要具备克隆方法。如果在类的设计之初没有很好的规划，等使用很久了才想到克隆，就可能非常的麻烦，尤其是在设计到深层次复制的时候，因为此时牵扯到很多因素，而且工作量非常大。

在给女朋友复制肉麻情话的之前必须充分检查，做适当的修改，别搞的发过去的情话中有参见某某具体网址的情况出现，否则的话，你就死定了 O(∩_∩)O 哈！