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 Factory Method 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。

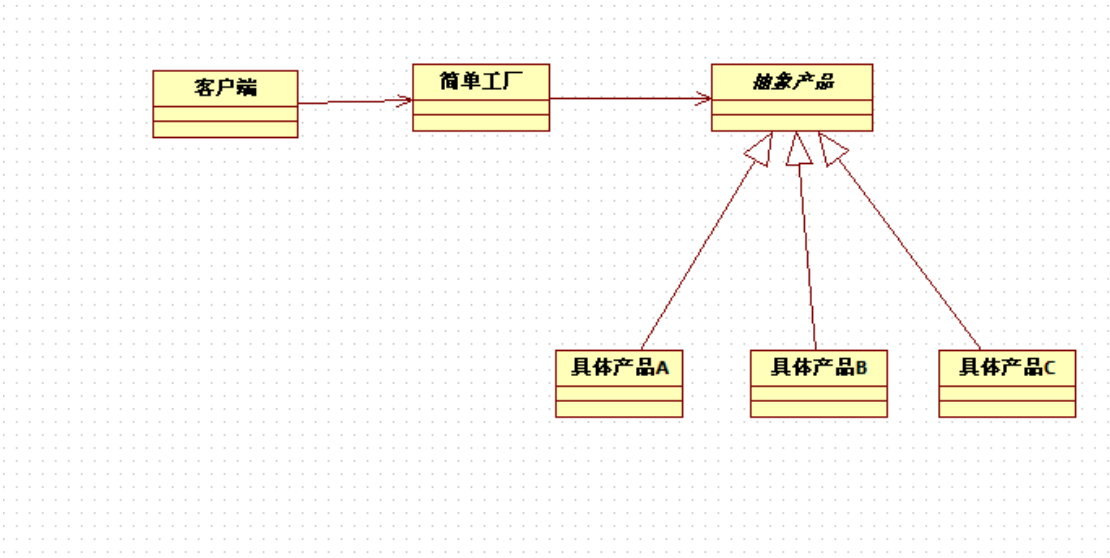
简单工厂模式的 UML 图：

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：

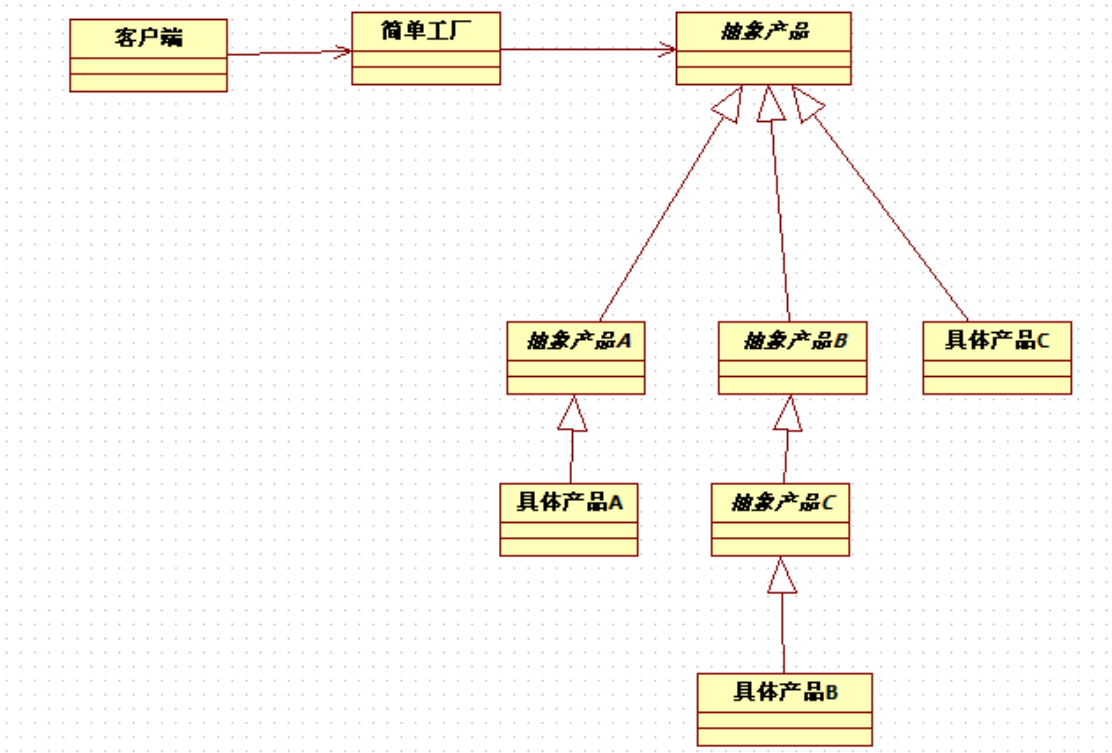
简单工厂模式解决的问题是如何去实例化一个合适的对象。

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。

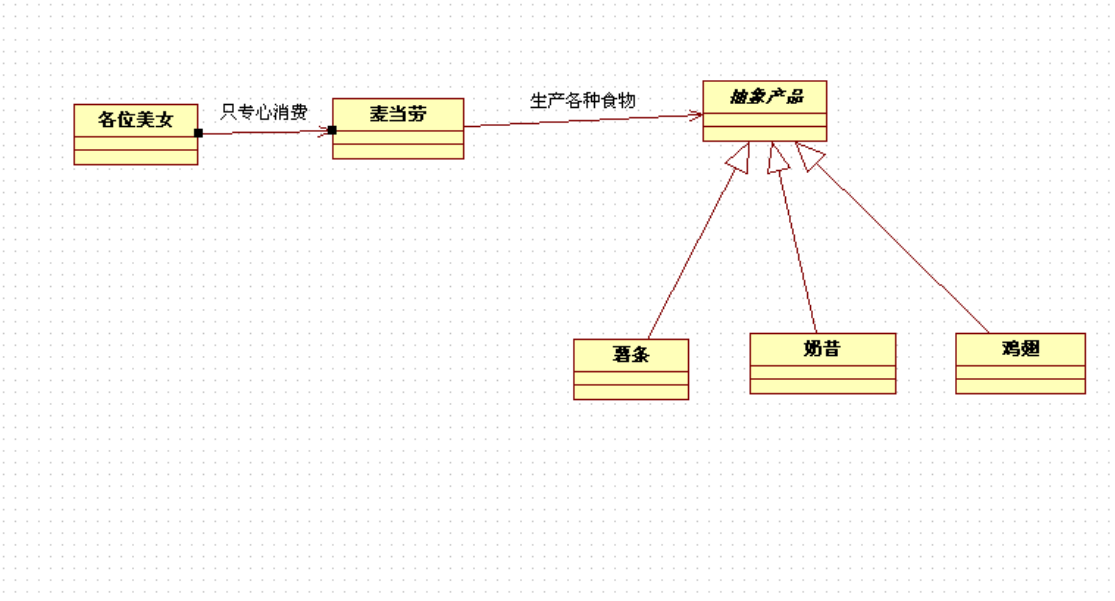
如果简单工厂模式所涉及到的具体产品之间没有共同的逻辑，那么我们就可以使用接口来扮演抽象产品的角色；如果具体产品之间有功能的逻辑或，我们就必须把这些共同的东西提取出来，放在一个抽象类中，然后让具体产品继承抽象类。为实现更好复用的目的，共同的东西总是应该抽象出来的。

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。

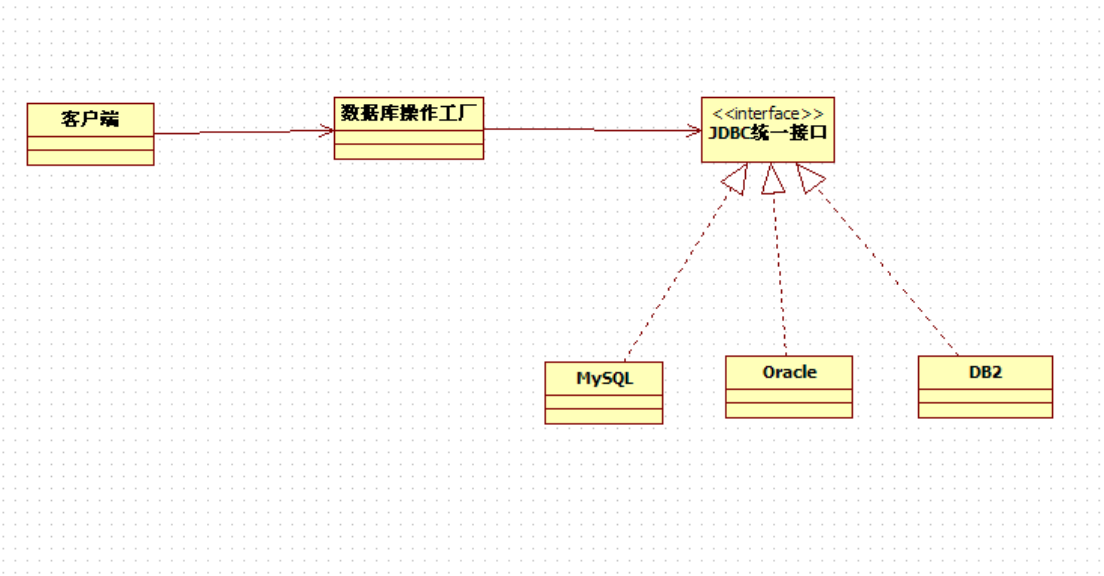
缺点：由于工厂类集中了所有实例的创建逻辑，这就直接导致一旦这个工厂出了问题，所有的客户端都会受到牵连；而且由于简单工厂模式的产品室基于一个共同的抽象类或者接口，这样一来，但产品的种类增加的时候，即有不同的产品接口或者抽象类的时候，工厂类就需要判断何时创建何种种类的产品，这就和创建何种种类产品的产品相互混淆在了一起，违背了单一职责，导致系统丧失灵活性和可维护性。而且更重要的是，简单工厂模式违背了“开放封闭原则”，就是违背了“系统对扩展开放，对修改关闭”的原则，因为当我新增加一个产品的时候必须修改工厂类，相应的工厂类就需要重新编译一遍。

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。