工厂方法法模式 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 subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

工厂方法模式的 UML 图：

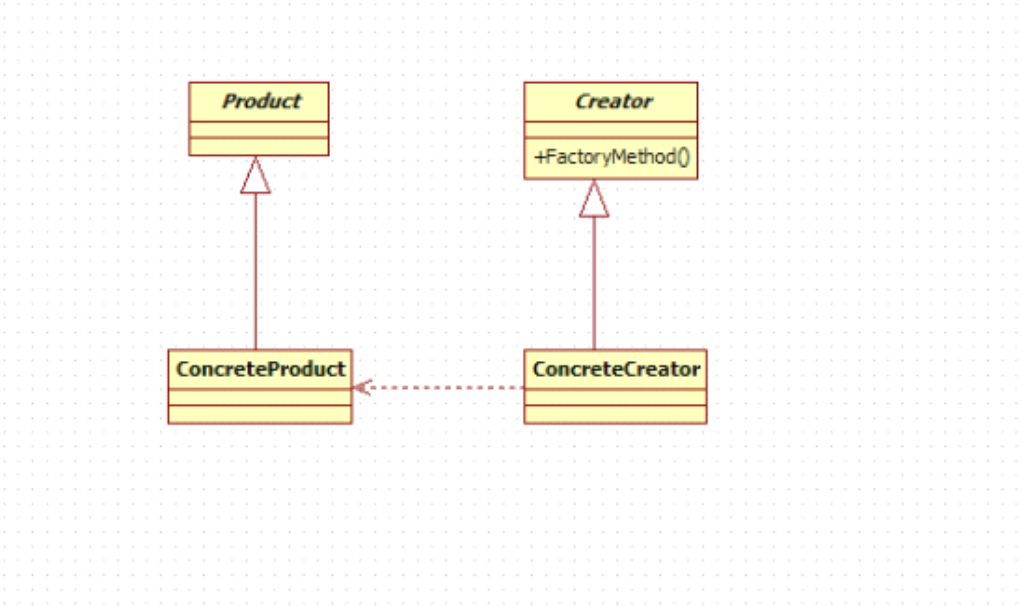
工厂方法模式中包含的角色及其相应的职责如下：

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。

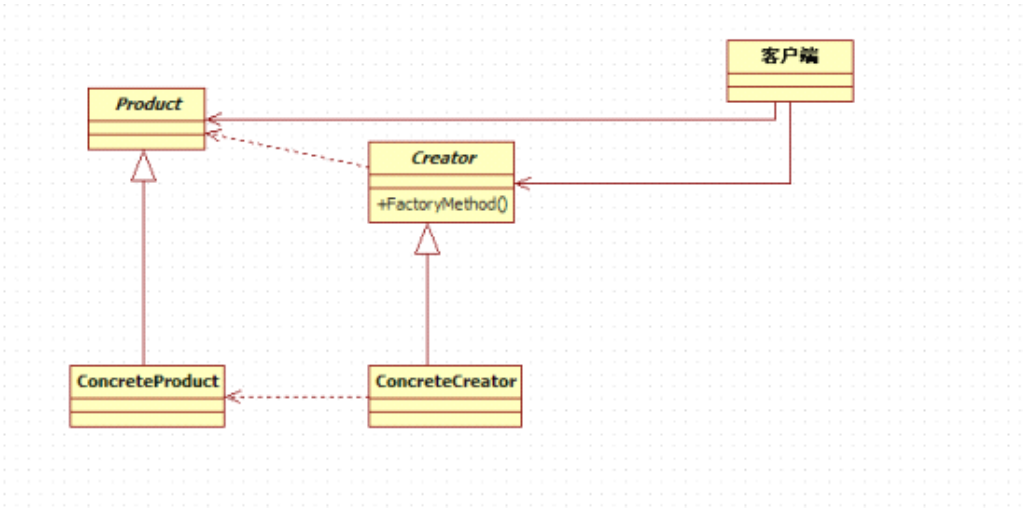


工厂方法法模式深入分析：

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。

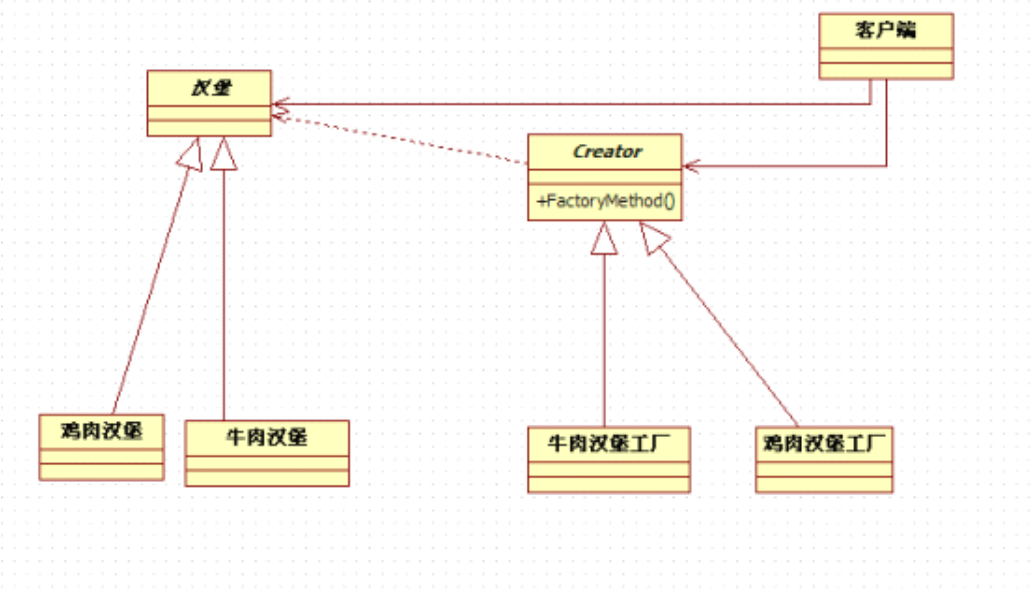
在实际的的使用中，抽闲产品和具体产品之间往往是多层次的产品结构，如下图所示：



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。