[1]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 for creating families of related or dependent objects without specifying their concrete classes.

抽象工厂模式的 UML 图：

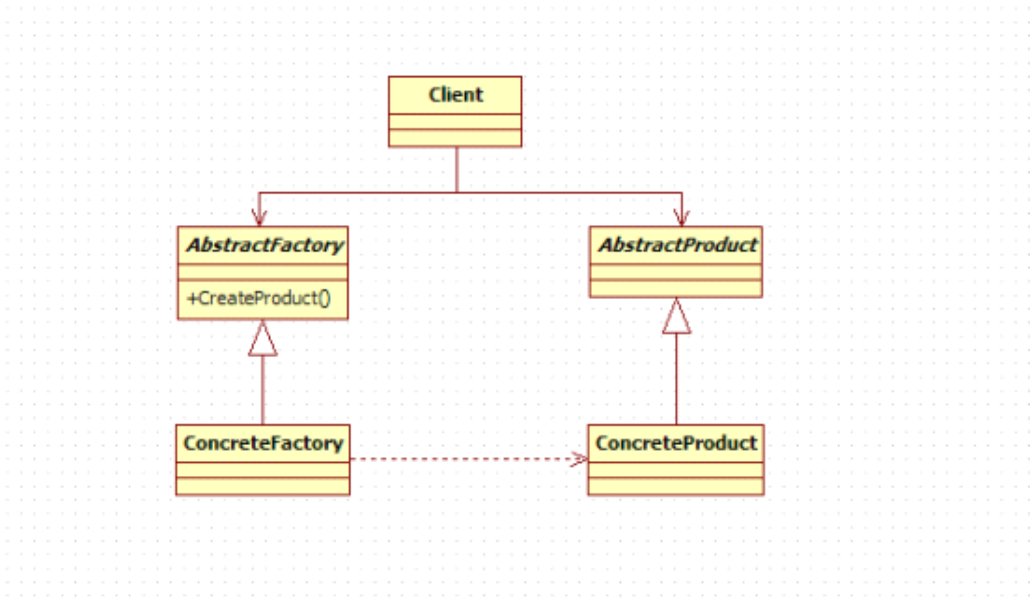
抽象工厂模式模式中包含的角色及其相应的职责如下：

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：

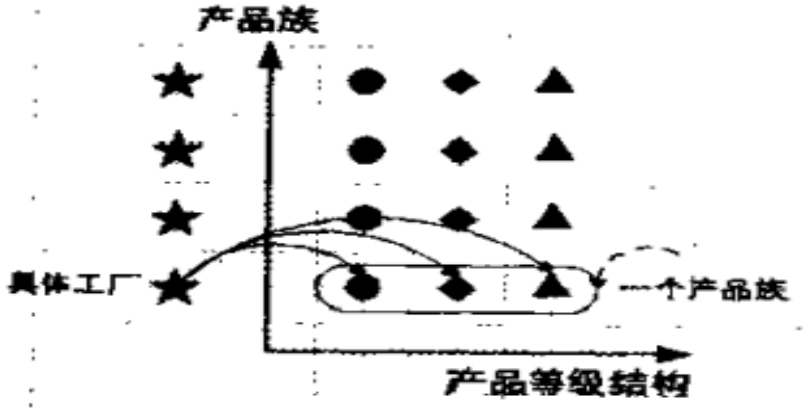
抽象工厂模式是在当产品有多个 抽象角色的时候使用的一种创建型设计模式。

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。

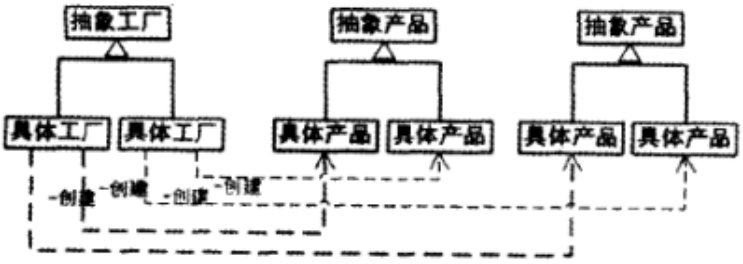


对于每一个产品族，都有一个具体工厂。而每一个具体工厂创建属于同一个产品族，但是分属于不同等级结构的产品。

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：

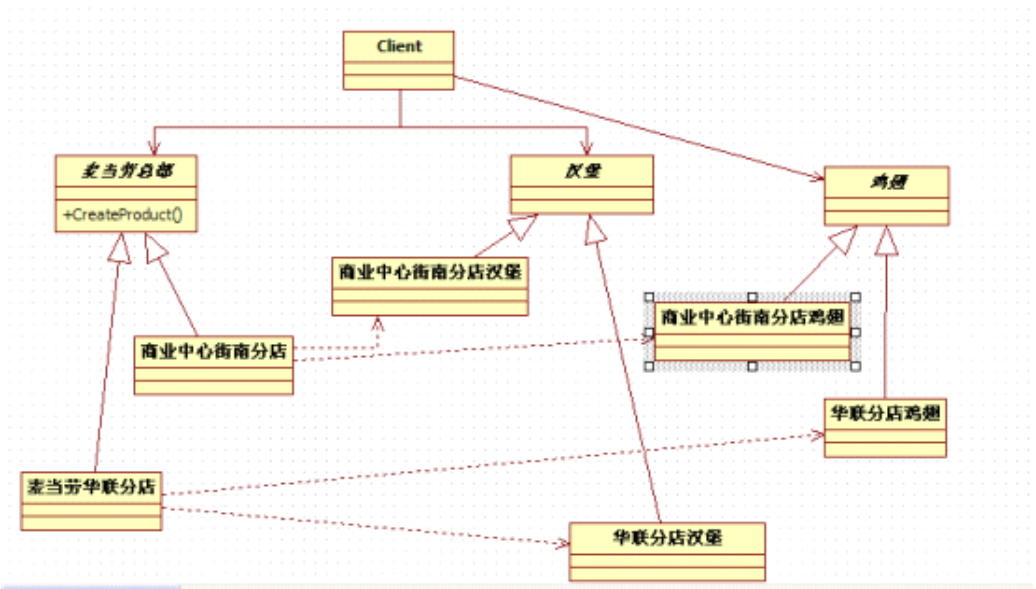


抽象工厂模式使用场景分析及代码实现：

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。