抽象工厂模式

> *整理自：《java与模式》之抽象工厂模式*

场景问题

举个生活中常见的例子——组装电脑，我们在组装电脑的时候，通常需要选择一系列的配件，比如CPU、硬盘、内存、主板、电源、机箱等。为讨论使用简单点，只考虑选择CPU和主板的问题。

事实上，在选择CPU的时候，面临一系列的问题，比如品牌、型号、针脚数目、主频等问题，只有把这些问题都确定下来，才能确定具体的CPU。

同样，在选择主板的时候，也有一系列问题，比如品牌、芯片组、集成芯片、总线频率等问题，也只有这些都确定了，才能确定具体的主板。

选择不同的CPU和主板，是每个客户在组装电脑的时候，向装机公司提出的要求，也就是我们每个人自己拟定的装机方案。

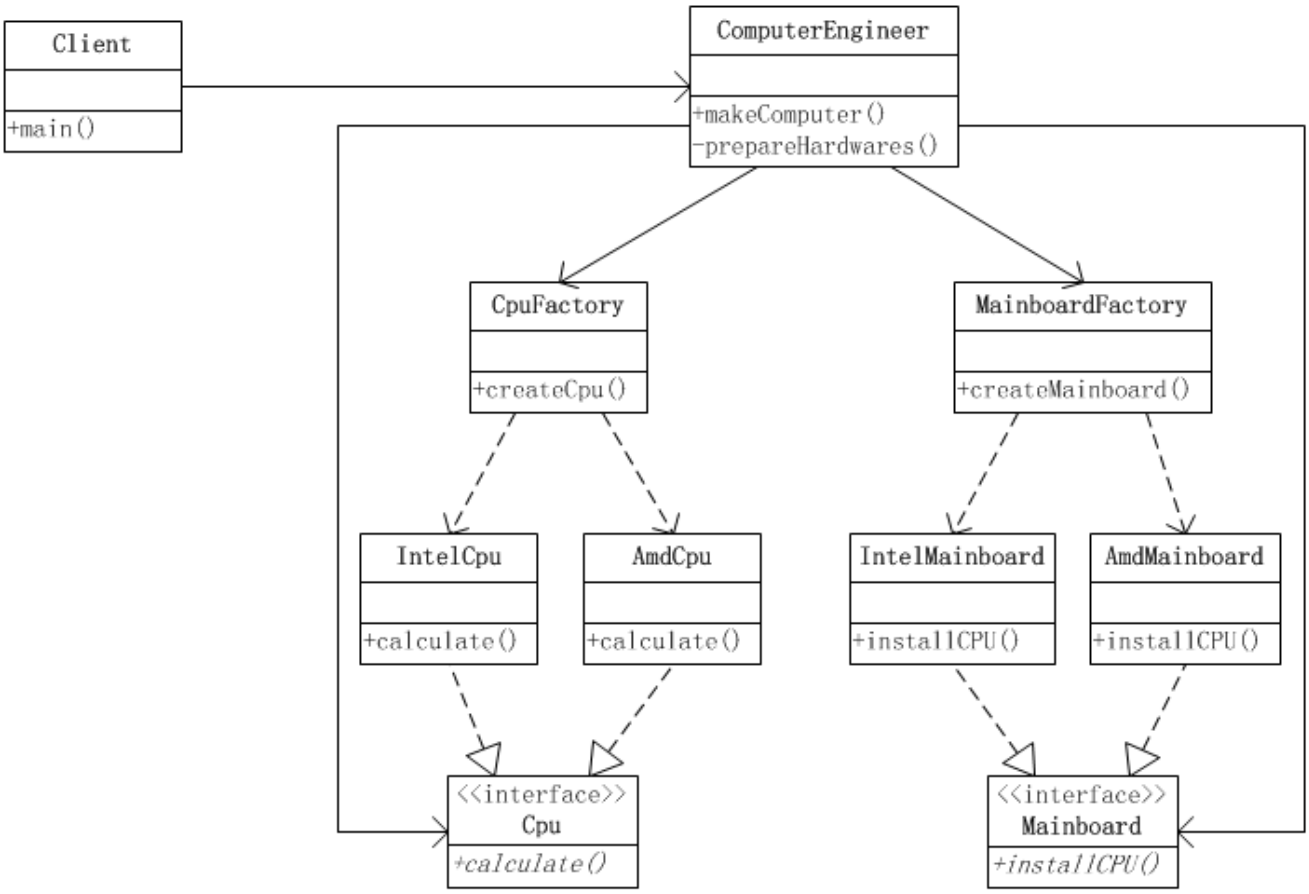
在最终确定这个装机方案之前，还需要整体考虑各个配件之间的兼容性。比如：CPU和主板，如果使用Intel的CPU和AMD的主板是根本无法组装的。因为Intel的CPU针脚数与AMD主板提供的CPU插口不兼容，就是说如果使用Intel的CPU根本就插不到AMD的主板中，所以装机方案是整体性的，里面选择的各个配件之间是有关联的。

对于装机工程师而言，他只知道组装一台电脑，需要相应的配件，但是具体使用什么样的配件，还得由客户说了算。也就是说装机工程师只是负责组装，而客户负责选择装配所需要的具体的配件。因此，当装机工程师为不同的客户组装电脑时，只需要根据客户的装机方案，去获取相应的配件，然后组装即可。

使用简单工厂模式的解决方案

考虑客户的功能，需要选择自己需要的CPU和主板，然后告诉装机工程师自己的选择，接下来就等着装机工程师组装电脑了。

对装机工程师而言，只是知道CPU和主板的接口，而不知道具体实现，很明显可以用上简单工厂模式或工厂方法模式。为了简单，这里选用简单工厂。客户告诉装机工程师自己的选择，然后装机工程师会通过相应的工厂去获取相应的实例对象。



源代码

CPU接口与具体实现:

```
public interface Cpu {
    public void calculate();
}
```

```
public class IntelCpu implements Cpu {
    /**
     * CPU的针脚数
     */
    private int pins = 0;
    public  IntelCpu(int pins){
        this.pins = pins;
    }
    @Override
    public void calculate() {
        // TODO Auto-generated method stub
```

```
        System.out.println("Intel CPU的针脚数: " + pins);
    }

}
```

```
public class AmdCpu implements Cpu {
    /**
     * CPU的针脚数
     */
    private int pins = 0;
    public  AmdCpu(int pins){
        this.pins = pins;
    }
    @Override
    public void calculate() {
        // TODO Auto-generated method stub
        System.out.println("AMD CPU的针脚数: " + pins);
    }
}
```

主板接口与具体实现

```
public interface Mainboard {
    public void installCPU();
}
```

```
public class IntelMainboard implements Mainboard {
    /**
     * CPU插槽的孔数
     */
    private int cpuHoles = 0;
    /**
     * 构造方法，传入CPU插槽的孔数
     * @param cpuHoles
     */
    public IntelMainboard(int cpuHoles){
        this.cpuHoles = cpuHoles;
    }
    @Override
```

```
    public void installCPU() {
        // TODO Auto-generated method stub
        System.out.println("Intel主板的CPU插槽孔数是: " + cpuHoles);
    }

}
```

```
public class AmdMainboard implements Mainboard {
    /**
     * CPU插槽的孔数
     */
    private int cpuHoles = 0;
    /**
     * 构造方法，传入CPU插槽的孔数
     * @param cpuHoles
     */
    public AmdMainboard(int cpuHoles){
        this.cpuHoles = cpuHoles;
    }
    @Override
    public void installCPU() {
        // TODO Auto-generated method stub
        System.out.println("AMD主板的CPU插槽孔数是: " + cpuHoles);
    }
}
```

CPU与主板工厂类

```
public class CpuFactory {
    public static Cpu createCpu(int type){
        Cpu cpu = null;
        if(type == 1){
            cpu = new IntelCpu(755);
        }else if(type == 2){
            cpu = new AmdCpu(938);
        }
        return cpu;
    }
}
```

```
public class MainboardFactory {
    public static Mainboard createMainboard(int type){
        Mainboard mainboard = null;
        if(type == 1){
            mainboard = new IntelMainboard(755);
        }else if(type == 2){
            mainboard = new AmdMainboard(938);
        }
        return mainboard;
    }
}
```

装机工程师类与客户类运行结果如下：

```
public class ComputerEngineer {
    /**
     * 定义组装机需要的CPU
     */
    private Cpu cpu = null;
    /**
     * 定义组装机需要的主板
     */
    private Mainboard mainboard = null;
    public void makeComputer(int cpuType , int mainboard){
        /**
         * 组装机器的基本步骤
         */
        //1:首先准备好装机所需要的配件
        prepareHardwares(cpuType, mainboard);
        //2:组装机器
        //3:测试机器
        //4: 交付客户
    }
    private void prepareHardwares(int cpuType , int mainboard){
        //这里要去准备CPU和主板的具体实现，为了示例简单，这里只准备这两个
        //可是，装机工程师并不知道如何去创建，怎么办呢？

        //直接找相应的工厂获取
        this.cpu = CpuFactory.createCpu(cpuType);
        this.mainboard = MainboardFactory.createMainboard(mainboard);
```

```
        //测试配件是否好用
        this.cpu.calculate();
        this.mainboard.installCPU();
    }
}
```

```
public class Client {
    public static void main(String[]args){
        ComputerEngineer cf = new ComputerEngineer();
        cf.makeComputer(1,1);
    }
}
```

运行结果如下：

上面的实现，虽然通过简单工厂方法解决了：对于装机工程师，只知CPU和主板的接口，而不知道具体实现的问题。**但还有一个问题没有解决，那就是这些CPU对象和主板对象其实是有关系的，需要相互匹配的。**而上面的实现中，并没有维护这种关联关系，CPU和主板是由客户任意选择，这是有问题的。比如在客户端调用makeComputer时，传入参数为(1,2)，运行结果如下：

观察上面结果就会看出问题。客户选择的是Intel的CPU针脚数为755，而选择的主板是AMD，主板上的CPU插孔是938，根本无法组装，这就是没有维护配件之间的关系造成的。该怎么解决这个问题呢？

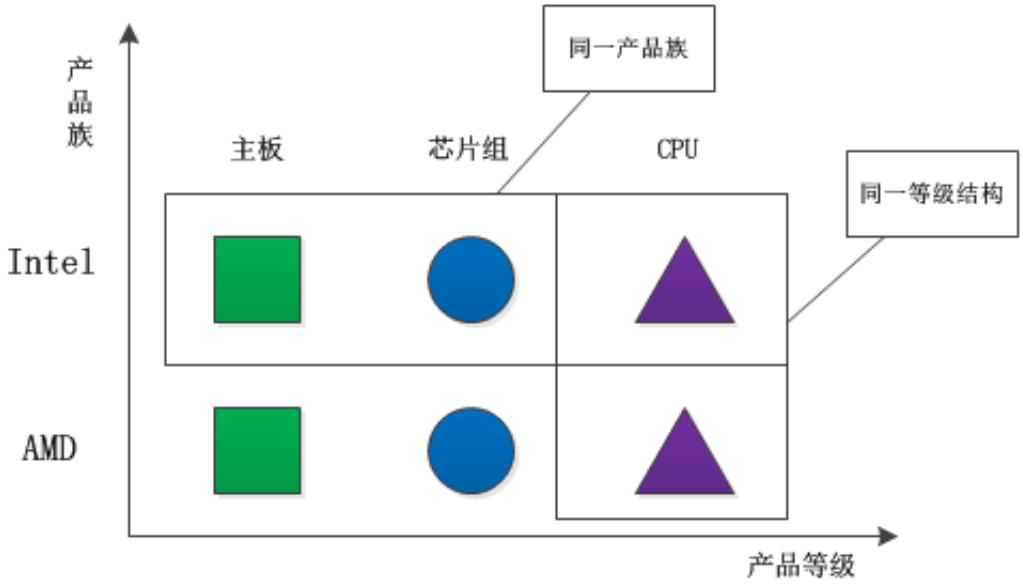
引进抽象工厂模式

每一个模式都是针对一定问题的解决方案。抽象工厂模式与工厂方法模式的最大区别就

在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。

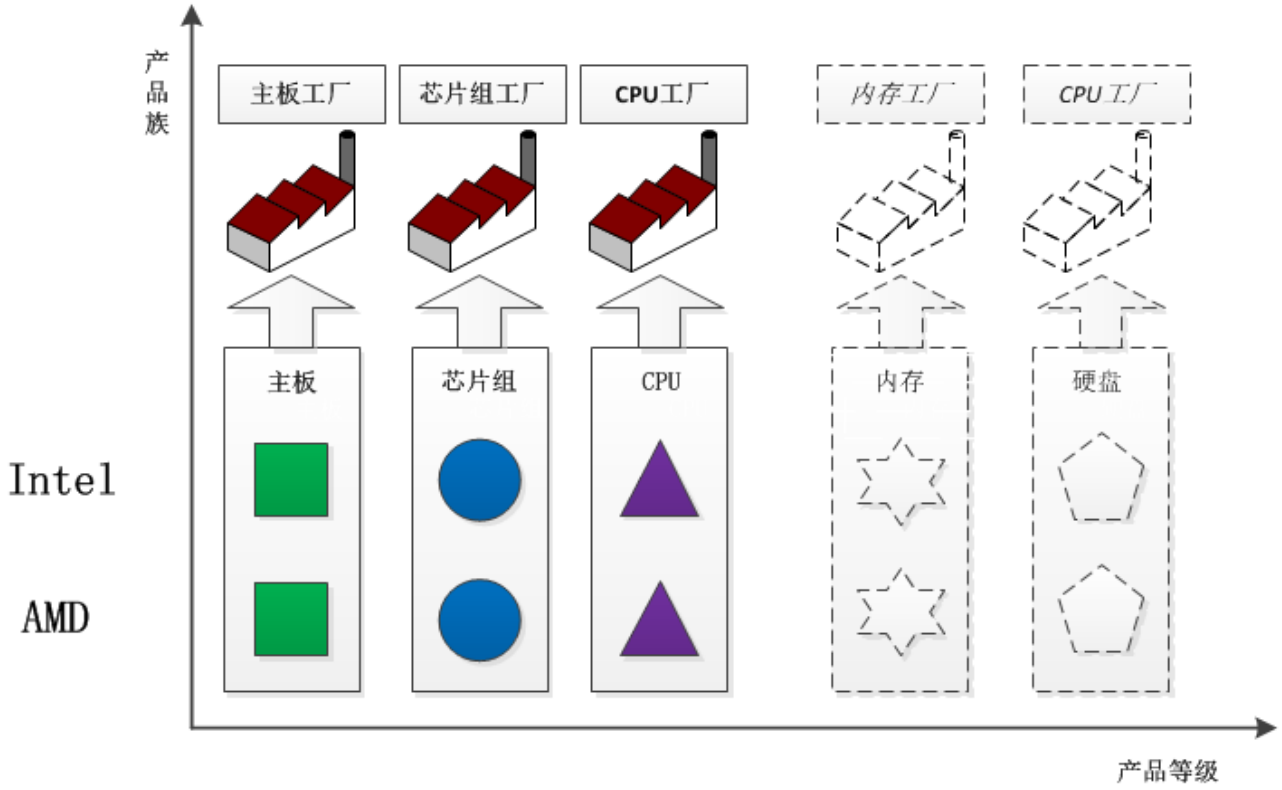
在学习抽象工厂具体实例之前，应该明白两个重要的概念：产品族和产品等级。

所谓产品族，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。比如AMD的主板、芯片组、CPU组成一个家族，Intel的主板、芯片组、CPU组成一个家族。而这两个家族都来自于三个产品等级：主板、芯片组、CPU。一个等级结构是由相同的结构的产品组成，示意图如下：

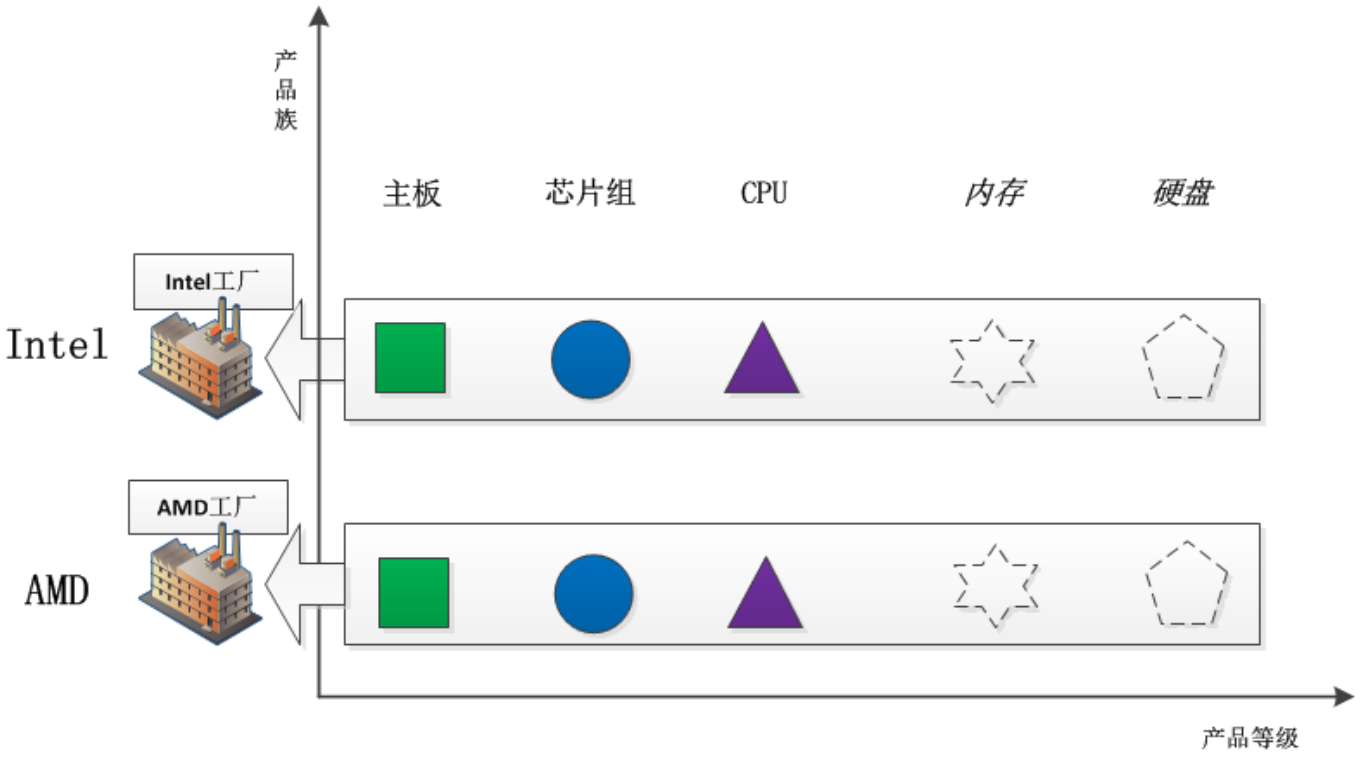


显然，每一个产品族中含有产品的数目，与产品等级结构的数目是相等的。产品的等级结构与产品族将产品按照不同方向划分，形成一个二维的坐标系。横轴表示产品的等级结构，纵轴表示产品族，上图共有两个产品族，分布于三个不同的产品等级结构中。只要指明一个产品所处的产品族以及它所属的等级结构，就可以唯一的确定这个产品。

上面所给出的三个不同的等级结构具有平行的结构。因此，如果采用工厂方法模式，就势必要使用三个独立的工厂等级结构来对付这三个产品等级结构。由于这三个产品等级结构的相似性，会导致三个平行的工厂等级结构。随着产品等级结构的数目的增加，工厂方法模式所给出的工厂等级结构的数目也会随之增加。如下图：



那么，是否可以使用同一个工厂等级结构来对付这些相同或者极为相似的产品等级结构呢？当然可以的，而且这就是抽象工厂模式的好处。同一个工厂等级结构负责三个不同产品等级结构中的产品对象的创建。



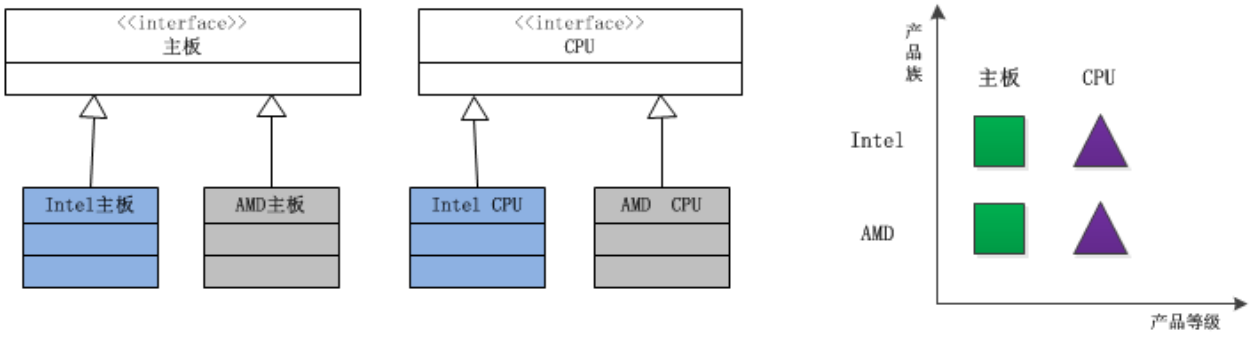
可以看出，一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象。显然，这时候抽象工厂模式比简单工厂模式、工厂方法模式更有效率。对应于每一个产品族都有一个具体工厂。而每一个具体工厂负责创建属于同一个产品族，但是分属于不同等级结构的产品。

抽象工厂模式结构

抽象工厂模式是对象的创建模式，它是工厂方法模式的进一步推广。

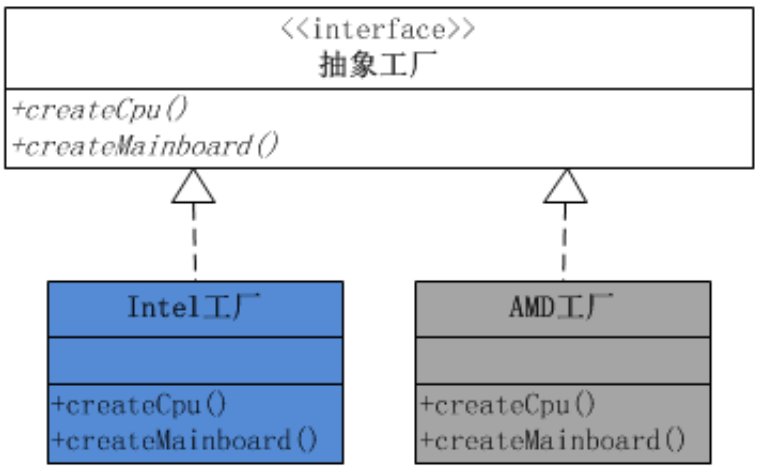
假设一个子系统需要一些产品对象，而这些产品又属于一个以上的产品等级结构。那么为了将消费这些产品对象的责任和创建这些产品对象的责任分割开来，可以引进抽象工厂模式。这样的话，消费产品的一方不需要直接参与产品的创建工作，而只需要向一个公用的工厂接口请求所需要的产品。

通过使用抽象工厂模式，可以处理具有相同（或者相似）等级结构中的多个产品族中的产品对象的创建问题。如下图所示：

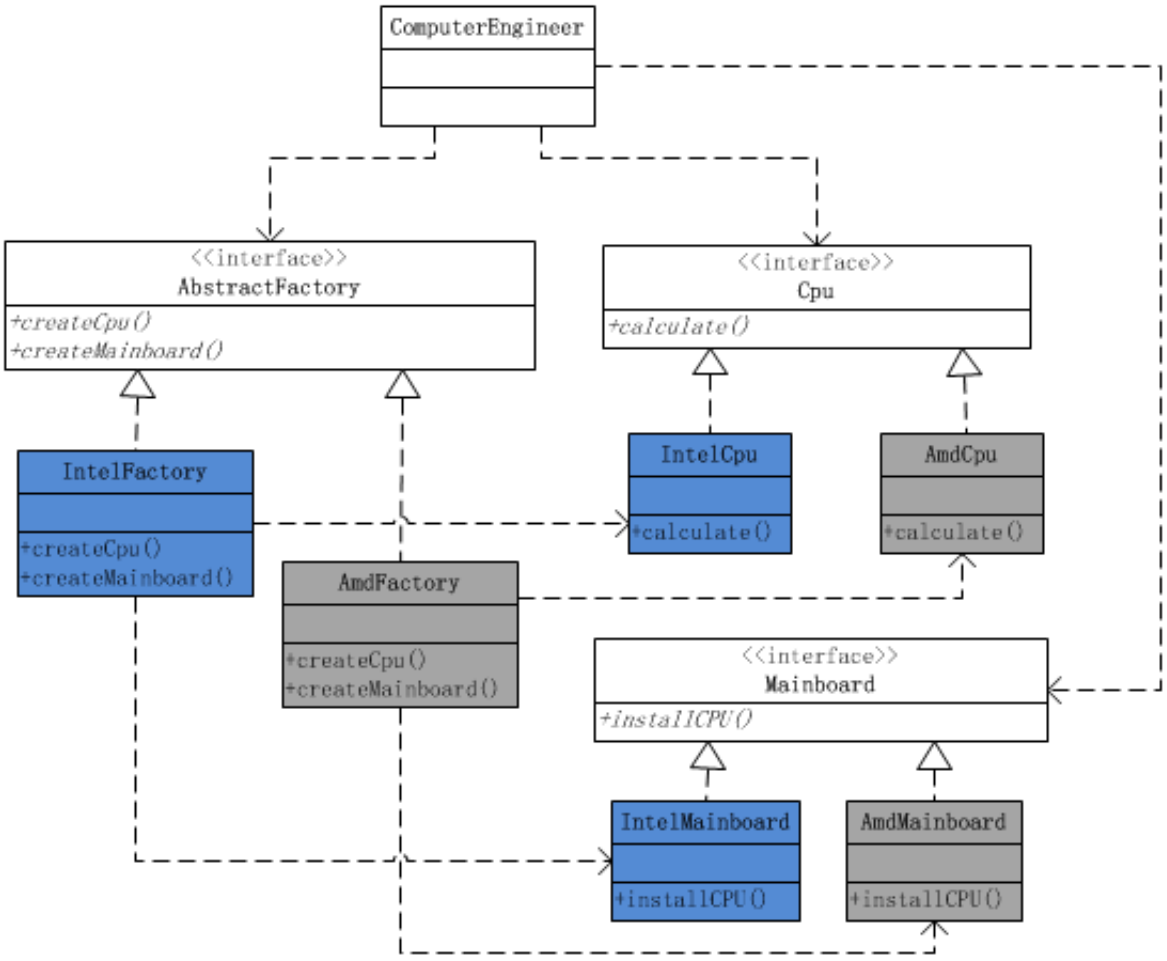


由于这两个产品族的等级结构相同，因此使用同一个工厂族也可以处理这两个产品族的创建问题，这就是抽象工厂模式。

根据产品角色的结构图，就不难给出工厂角色的结构设计图。



可以看出，每一个工厂角色都有两个工厂方法，分别负责创建分属不同产品等级结构的产品对象。



源代码

前面示例实现的CPU接口和CPU实现对象，主板接口和主板实现对象，都不需要变化。

前面示例中创建CPU的简单工厂和创建主板的简单工厂，都不再需要。

新加入的抽象工厂类和实现类：

```
public interface AbstractFactory {
    /**
     * 创建CPU对象
     * @return CPU对象
     */
    public Cpu createCpu();
    /**
     * 创建主板对象
     * @return 主板对象
     */
    public Mainboard createMainboard();
}
```

```
public class IntelFactory implements AbstractFactory {

    @Override
    public Cpu createCpu() {
        // TODO Auto-generated method stub
        return new IntelCpu(755);
    }

    @Override
    public Mainboard createMainboard() {
        // TODO Auto-generated method stub
        return new IntelMainboard(755);
    }

}
```

```
public class AmdFactory implements AbstractFactory {

    @Override
    public Cpu createCpu() {
        // TODO Auto-generated method stub
        return new IntelCpu(938);
```

```
    }

    @Override
    public Mainboard createMainboard() {
        // TODO Auto-generated method stub
        return new IntelMainboard(938);
    }

}
```

装机工程师类跟前面的实现相比，主要的变化是：从客户端不再传入选择CPU和主板的参数，而是直接传入客户已经选择好的产品对象。这样就避免了单独去选择CPU和主板所带来的兼容性问题，客户要选就是一套，就是一个系列。

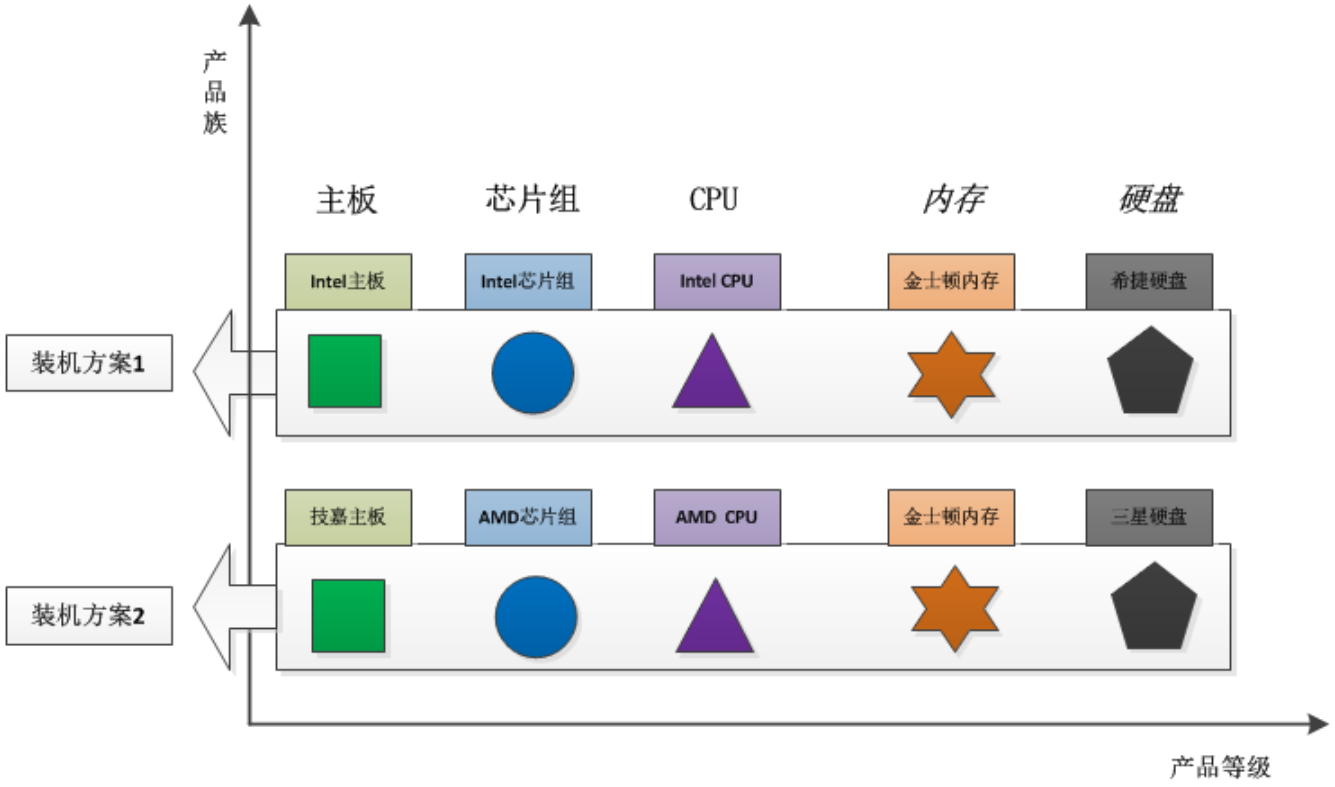
```
public class ComputerEngineer {
    /**
     * 定义组装机需要的CPU
     */
    private Cpu cpu = null;
    /**
     * 定义组装机需要的主板
     */
    private Mainboard mainboard = null;
    public void makeComputer(AbstractFactory af){
        /**
         * 组装机器的基本步骤
         */
        //1:首先准备好装机所需要的配件
        prepareHardwares(af);
        //2:组装机器
        //3:测试机器
        //4: 交付客户
    }
    private void prepareHardwares(AbstractFactory af){
        //这里要去准备CPU和主板的具体实现，为了示例简单，这里只准备这两个
        //可是，装机工程师并不知道如何去创建，怎么办呢？

        //直接找相应的工厂获取
        this.cpu = af.createCpu();
        this.mainboard = af.createMainboard();
```

```
        //测试配件是否好用
        this.cpu.calculate();
        this.mainboard.installCPU();
    }
}
```

客户端代码：

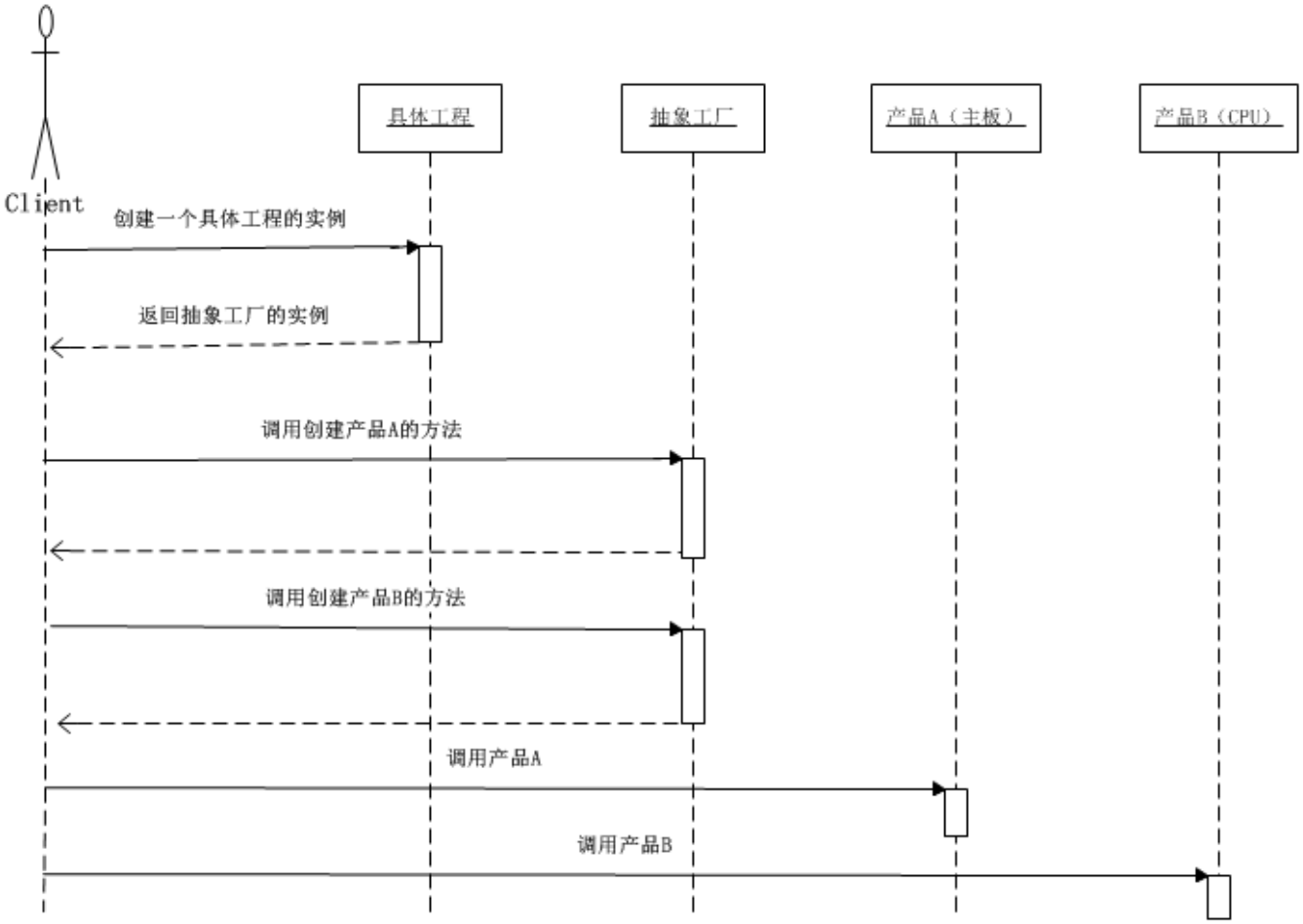
```
public class Client {
    public static void main(String[]args){
        //创建装机工程师对象
        ComputerEngineer cf = new ComputerEngineer();
        //客户选择并创建需要使用的产品对象
        AbstractFactory af = new IntelFactory();
        //告诉装机工程师自己选择的产品，让装机工程师组装电脑
        cf.makeComputer(af);
    }
}
```

抽象工厂的功能是为一系列相关对象或相互依赖的对象创建一个接口。一定要注意，这个接口内的方法不是任意堆砌的，而是一系列相关或相互依赖的方法。比如上面例子中的主板和CPU，都是为了组装一台电脑的相关对象。不同的装机方案，代表一种具体的电脑系列。



由于抽象工厂定义的一系列对象通常是相关或相互依赖的，这些产品对象就构成了一个产品族，也就是抽象工厂定义了一个产品族。

这就带来非常大的灵活性，切换产品族的时候，只要提供不同的抽象工厂实现就可以了，也就是说现在是以一个产品族作为一个整体被切换。



在什么情况下应当使用抽象工厂模式

1.一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。

2.这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

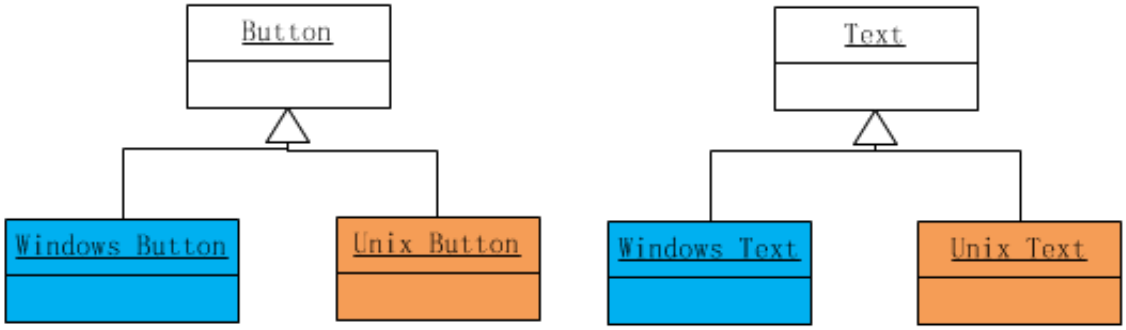
3.同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。（比如：Intel主板必须使用Intel CPU、Intel芯片组）。

4.系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

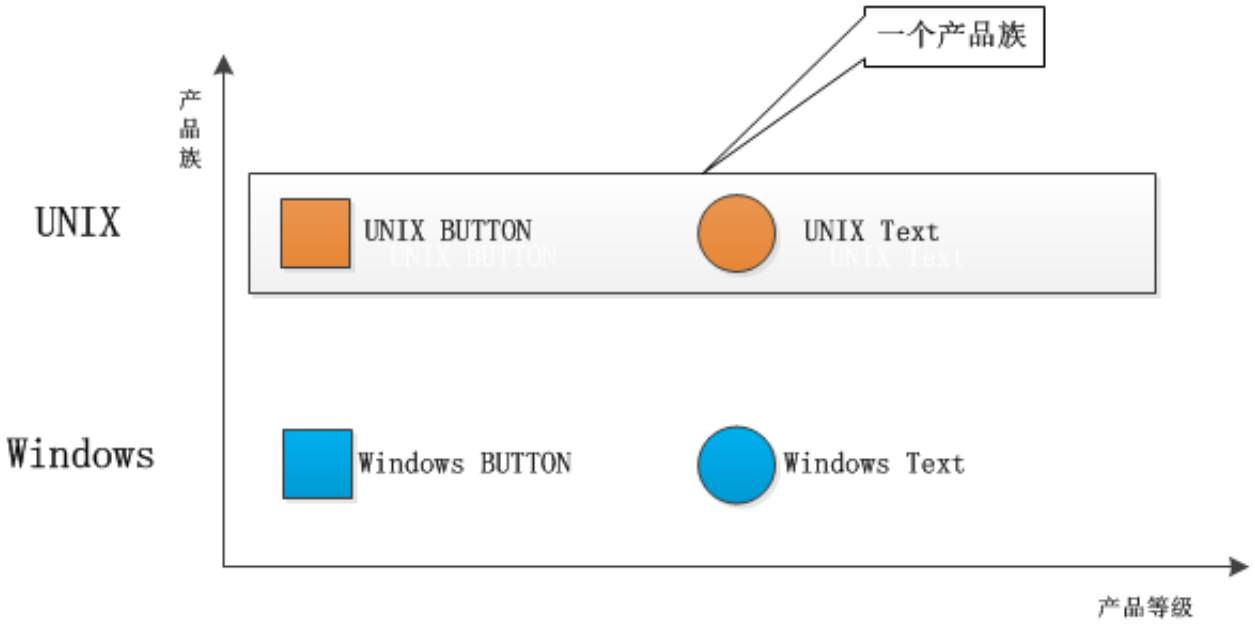
抽象工厂模式的起源

抽象工厂模式的起源或者最早的应用，是用于创建分属于不同操作系统的视窗构建。比如：命令按键（Button）与文字框（Text)都是视窗构建，在UNIX操作系统的视窗环境和Windows操作系统的视窗环境中，这两个构建有不同的本地实现，它们的细节有所不同。

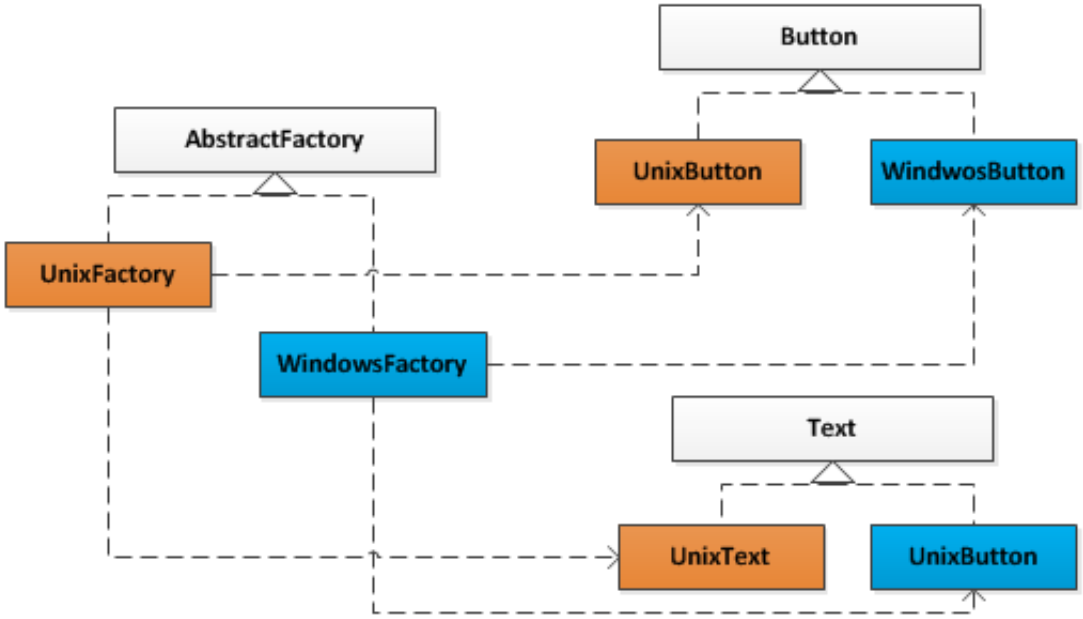
在每一个操作系统中，都有一个视窗构建组成的构建家族。在这里就是Button和Text组成的产品族。而每一个视窗构件都构成自己的等级结构，由一个抽象角色给出抽象的功能描述，而由具体子类给出不同操作系统下的具体实现。



可以发现在上面的产品类图中，有两个产品的等级结构，分别是Button等级结构和Text等级结构。同时有两个产品族，也就是UNIX产品族和Windows产品族。UNIX产品族由UNIX Button和UNIX Text产品构成；而Windows产品族由Windows Button和Windows Text产品构成。



系统对产品对象的创建需求由一个工程的等级结构满足，其中有两个具体工程角色，即UnixFactory和WindowsFactory。UnixFactory对象负责创建Unix产品族中的产品，而WindowsFactory对象负责创建Windows产品族中的产品。这就是抽象工厂模式的应用，抽象工厂模式的解决方案如下图：



显然，一个系统只能够在某一个操作系统的视窗环境下运行，而不能同时在不同的操作系统上运行。所以，系统实际上只能消费属于同一个产品族的产品。

在现代的应用中，抽象工厂模式的使用范围已经大大扩大了，不再要求系统只能消费某一个产品族了。因此，可以不必理会前面所提到的原始用意。

抽象工厂模式的优点

- 分离接口和实现
  客户端使用抽象工厂来创建需要的对象，而客户端根本就不知道具体的实现是谁，客户端只是面向产品的接口编程而已。也就是说，客户端从具体的产品实现中解耦。

- 使切换产品族变得容易
  因为一个具体的工厂实现代表的是一个产品族，比如上面例子的从Intel系列到AMD系列只需要切换一下具体工厂。

抽象工厂模式的缺点

不太容易扩展新的产品

如果需要给整个产品族添加一个新的产品，那么就需要修改抽象工厂，这样就会导致修改所有的工厂实现类。