创建型模式（二）

2020年3月23日 22:12

创建型模式（二）

作业要求：

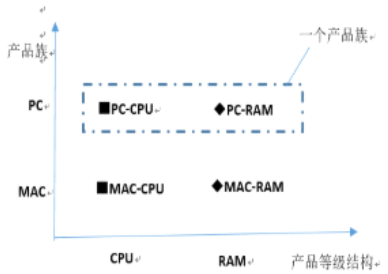
作业要求：

1、试画出抽象工厂模式的模式结构图，并对模式进行分析。

2、试画出建造者模式的模式结构图，并对模式进行分析。

3、计算机包括内存（RAM）,CPU等硬件设备，根据图中的"产品等级结构--产品族"示意图，使用抽象工厂模式实现计算机设备创建过程并绘制出相应的类图。

（画出模式结构图，并进行解析）

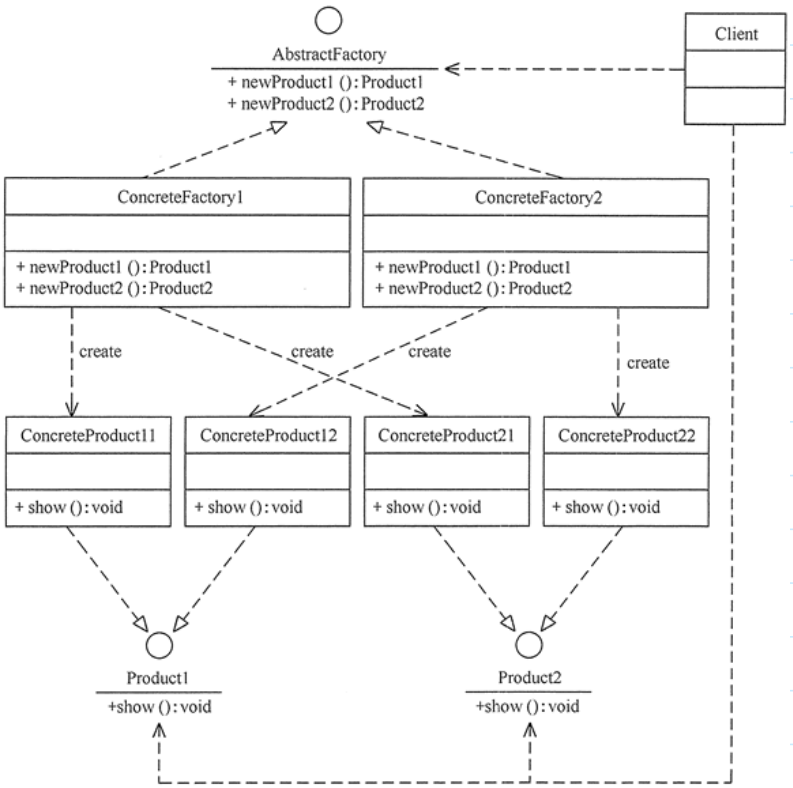


- 作业与笔记github地址：
  https://github.com/baobaotql/CCNU_DesignPattern

- **Q1：试画出抽象工厂模式的模式结构图，并对模式进行分析。**
- **A1:**
  - **抽象工厂模式概述**
    - **定义：**抽象工厂模式是提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。抽象工厂又称为Kit模式，属于对象创建型工厂。
    - **理解：**定义中说了，我们是要创建一个接口， 而这个接口是干嘛的呢，前面说了，是为了创建一组相关或者相互依赖的对象，而且还有一点就是，我们创建的对象不是具体的类，也就是说我们创建的是一个接口或者一个抽象类。

  - **抽象工厂模式的结构与模式结构图**
    - **抽象工厂：**声明一组用于创建一族产品的方法，每个方法对应一种对象；在抽象工厂中声明了多个工厂方法， 用于创建不同类型的对象，抽象工厂可以是接口， 也可以是抽象类或者具体类
    - **具体工厂：**具体工厂实现了抽象工厂，每个工厂方法返回一个具体对象，一个具体工厂所创建的具体对象构成一个族
    - **抽象产品：**定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。
    - **具体产品：**实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间是多对一的关系。

- **模式结构图**



- **模式的实现**
  - **抽象工厂：**提供产品的生成方法

```
interface AbstractFactory
{
    public Product1 newProduct1();
    public Product2 newProduct2();
}
```

  - **具体工厂：**实现了产品的生成方法

```
class ConcreteFactory1 implements AbstractFactory
{
    public Product1 newProduct1()
    {
        System.out.println("具体工厂 1 生成-->具体产品 11...");
        return new ConcreteProduct11();
    }
    public Product2 newProduct2()
    {
        System.out.println("具体工厂 1 生成-->具体产品 21...");
        return new ConcreteProduct21();
    }
}
```

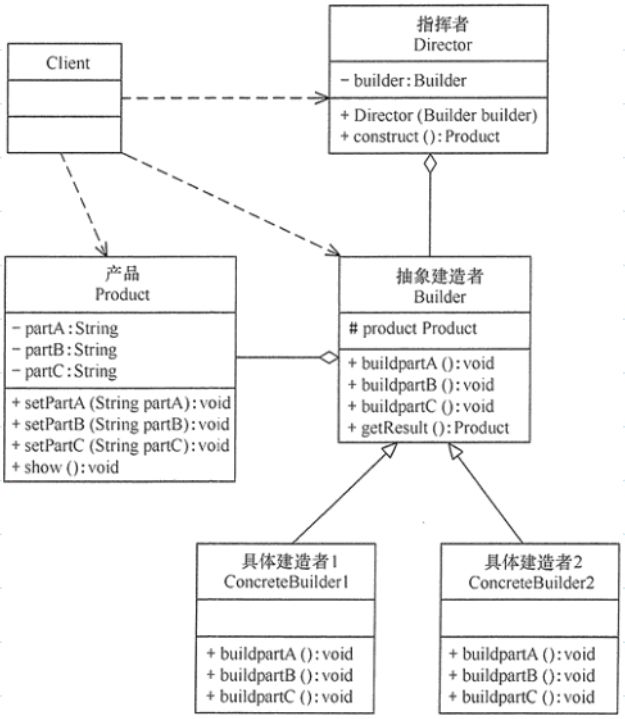
- **抽象工厂模式优缺点与模式分析**
  - **优点**（除了具有工厂方法模式的优点外）
    - 可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理。
    - 当增加一个新的产品族时不需要修改原代码，满足开闭原则。
  - **缺点**

        - 当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。
    - **使用场景**
        - 一个系统不应当依赖于具体类实例如何被创建、 组合和表达的细节， 这对于所有类型的工厂模式都是很重要的， 用户无须关心对象的创建过程， 将对象的创建和使用解耦；
        - 系统中有多于一个的族， 而每次只使用其中某一族。 可以通过配置文件等方式来使得用户可以动态改变族， 也可以很方便地增加新的族。
        - 等级结构稳定， 设计完成之后， 不会向系统中增加新的等级结构或者删除已有的等级结构。
    - **模式分析**
        - 抽象工厂模式的扩展有一定的“开闭原则”倾斜性：
            - 当增加一个新的产品族时只需增加一个新的具体工厂，不需要修改原代码，满足开闭原则。
            - 当产品族中需要增加一个新种类的产品时，则所有的工厂类都需要进行修改，不满足开闭原则。
        - 另一方面，当系统中只存在一个等级结构的产品时，抽象工厂模式将退化到工厂方法模式。

- **Q2：试画出建造者模式的模式结构图，并对模式进行分析。**
- **A2：**
    - **建造者模式概述：**
        - **定义：**建造者模式讲一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示；
          建造者模式是一步一步创建一个复杂的对象，它允许用户只铜鼓欧指定复杂对象的类型和内容就可以构建他们，用户不需要知道内部的具体构建细节。建造者模式术语对象创建模式
        - **理解：**建造者（Builder）模式和工厂模式的关注点不同：建造者模式注重零部件的组装过程，而工厂方法模式更注重零部件的创建过程，但两者可以结合使用。

    - **建造者模式的结构与模式结构图**
        - **抽象建造者：**它是一个包含创建产品各个子部件的抽象方法的接口，通常还包含一个返回复杂产品的方法
        - **具体建造者：**实现 Builder 接口，完成复杂产品的各个部件的具体创建方法
        - **指挥者：**它调用建造者对象中的部件构造与装配方法完成复杂对象的创建，在指挥者中不涉及具体产品的信息
        - **产品角色：**它是包含多个组成部件的复杂对象，由具体建造者来创建其各个部件

        - **模式结构图**



- **模式实现**
  - **产品角色：**包含多个组成部分的复杂对象

```
class Product
{
    private String partA;
    private String partB;
    private String partC;
    public void setPartA(String partA)
    {
        this.partA=partA;
    }
    public void setPartB(String partB)
    {
        this.partB=partB;
    }
    public void setPartC(String partC)
    {
        this.partC=partC;
    }
    public void show()
    {
        //显示产品的特性
    }
}
```

  - **抽象建造者：**包含创建产品各个子部件的抽象方法

```
abstract class Builder
{
    //创建产品对象
    protected Product product=new Product();
    public abstract void buildPartA();
    public abstract void buildPartB();
    public abstract void buildPartC();
    //返回产品对象
    public Product getResult()
```

```
        {
            return product;
        }
    }
```

- **具体建造者：**实现了抽象创建者接口

```
public class ConcreteBuilder extends Builder
{
    public void buildPartA()
    {
        product.setPartA("建造 PartA");
    }
    public void buildPartB()
    {
        product.setPartA("建造 PartB");
    }
    public void buildPartC()
    {
        product.setPartA("建造 PartC");
    }
}
```

- **指挥者：**调用建造者中的方法完成复杂对象的创建

```
class Director
{
    private Builder builder;
    public Director(Builder builder)
    {
        this.builder=builder;
    }
    //产品构建与组装方法
    public Product construct()
    {
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();
        return builder.getResult();
    }
}
```

- **客户类**

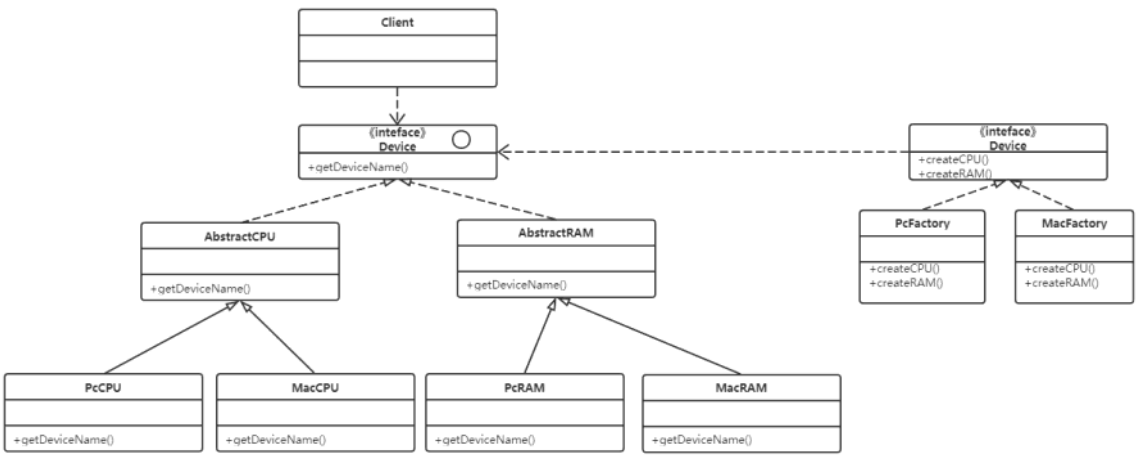
```
public class Client
{
    public static void main(String[] args)
    {
        Builder builder=new ConcreteBuilder();
        Director director=new Director(builder);
        Product product=director.construct();
        product.show();
    }
}
```

- **建造者模式优缺点与模式分析**
  - **优点**
    - 各个具体的建造者相互独立，有利于系统的扩展
    - 客户端不必知道产品内部组成的细节，便于控制细节风险
  - **缺点**
    - 产品的组成部分必须相同，这限制了其使用范围

- 如果产品的内部变化复杂，该模式会增加很多的建造者类

- **建造者模式和工厂模式的关注点不同：**建造者模式注重零部件的组装过程，而工厂方法模式更注重零部件的创建过程，但两者可以结合使用。
- **应用场景**
  - 创建的对象较复杂，由多个部件构成，各部件面临着复杂的变化，但构件间的建造顺序是稳定的
  - 创建复杂对象的算法独立于该对象的组成部分以及它们的装配方式，即产品的构建过程和最终的表示是独立的
- **模式分析**
  - 与抽象工厂模式相比，建造者工厂模式返回一个组装好的完整产品，而抽象工厂模式返回一系列相关的产品，这些产品谓语不同产品等级结构，构成了一个产品族。
  - 建造者模式在应用过程中可以根据需要改变，如果创建的产品种类只有一种，只需要一个具体建造者，这时可以省略掉抽象建造者，甚至可以省略掉指挥者角色。

- **Q3：计算机包括内存（RAM）,CPU等硬件设备，根据图中的”产品等级结构--产品族“示意图，使用抽象工厂模式实现计算机设备创建过程并绘制出相应的类图。**
  （画出模式结构图，并进行解析）
- **A3：**



  - **模式实现及注释解析**

```
//定义抽象产品类CPU
public interface CPU {
        void create();
    }
//定义抽象产品类RAM
public interface RAM {
        void create();
}
//定义具体产品类PcCPU
public class PcCPU implements CPU{
    public void create() {
```

```
            System.out.println(" Win CPU is working");
        }
    }
    //定义具体产品类PcRAM
    public class PcRAM implements RAM {
        public void create() {
            System.out.println("Win RAM is working");
        }
    }
    //定义具体产品类MacCPU
    public class MacCPU implements CPU{
        public void create() {
            System.out.println("Mac CPU is working");
        }
    }
    //定义具体产品类MacRAM
    public class MacRAM implements RAM{
        public void create() {
            System.out.println("Mac RAM is working");
        }
    }
    //定义抽象工厂接口ComputerFactory
    interface ComputerFactory {
        CPU produceCPU();

        RAM produceRAM();
    }
    //定义客户端类ComputerPartsClient
    public class ComputerPartsClient {
        public static void main(String args[]) {
            ComputerFactory factory;
            CPU cpu;
            RAM ram;
            factory = new PcFactory();
            cpu = factory.produceCPU();
            cpu.create();
            ram = factory.produceRAM();
            ram.create();
            ComputerFactory factory1;
            factory1 = new MacFactory();
            cpu = factory1.produceCPU();
            cpu.create();
            ram = factory1.produceRAM();
            ram.create();
        }
    }
    //定义具体工厂类PcFactory
    public class PcFactory implements ComputerFactory {
        public PcCPU produceCPU() {
            System.out.println("请使用PC产品族的CPU");
            return new PcCPU();
        }

        public PcRAM produceRAM() {
            System.out.println("请使用PC产品族的RAM");
            return new PcRAM();
```

```
    }
}
//定义具体工厂类MacFactory
public class MacFactory implements ComputerFactory{
    public MacCPU produceCPU() {
        System.out.println("请使用Mac产品族的CPU");
        return new MacCPU();
    }

    public MacRAM produceRAM() {
        System.out.println("请使用Mac产品族的RAM");
        return new MacRAM();
    }
}
```