创建型模式（一）

2020年3月16日 13:27

创建型模式（一）

作业要求：

1、试画出简单工厂模式的模式结构图，并对模式进行分析。

2、试画出工厂方法模式的模式结构图，并对模式进行分析。

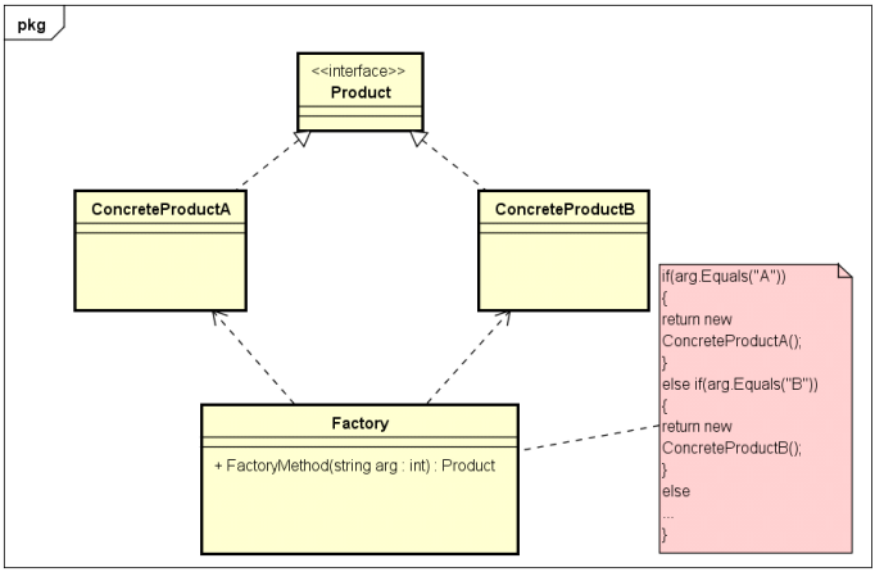
3、现需要设计一个程序来读取多种不同类型的图片格式，针对每一种图片格式都设计个图片读取器(ImageReader)，如GIF图片读取器(GifReader)用于读取GIF格式的图片、JPG图片读取器(Jpgreader)用于读取JPG格式的图片。图片读取器对象通过图片读取器工厂ImageReaderFactory来创建，ImageReaderFactory是一个抽象类，用于创建图片读取器的工厂方法，其子类 GifReaderFactory和 JpgReaderFactory用于创建具体的图片读取器对象。使用工厂方法模式实现该程序的设计。

（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

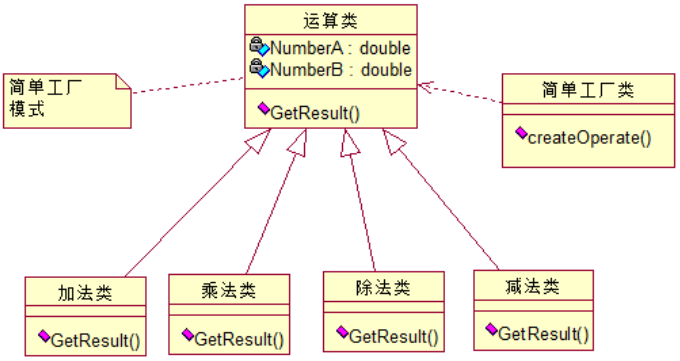
- **Q1：试画出简单工厂模式的模式结构图，并对模式进行分析。**
- **A1:**
  - **简单工程模式概述**
    - 定义：定义一个工厂类，他可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类
    - 在简单工厂模式中用于被创建实例的方法通常为静态(static)方法,因此简单工厂模式又被成为静态工厂方法(Static Factory Method)
    - 需要什么，只需要传入一个正确的参数，就可以获取所需要的对象，而无需知道其实现过程
    - 简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父亲

  - **简单工厂模式的结构与模式结构图**
    - **结构**
      - **Factory(工厂)：** 核心部分，负责实现创建所有产品的内部逻辑，工厂类可以被外界直接调用，创建所需对象
      - **Product(抽象类产品)：** 工厂类所创建的所有对象的父类，封装了产品对象的公共方法，所有的具体产品为其子类对象
      - **ConcreteProduct(具体产品)：** 简单工厂模式的创建目标，所有被创建的对象都是某个具体类的实例。它要实现抽象产品中声明的抽象方法(有关抽象类)



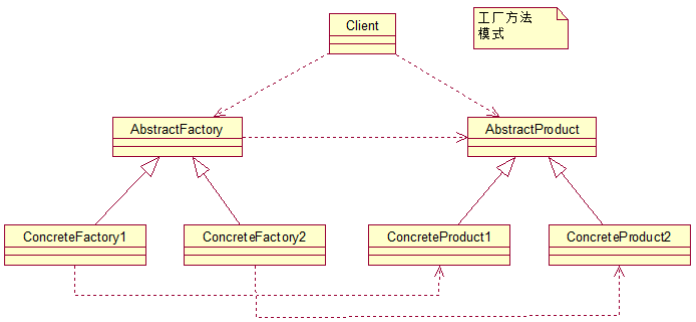
模式代码：

```
public class OperationFactory     ///创建了一个工厂，用户已输入符号，工厂就会判断怎么计算了
    {
        public static Operation createOperate(string operate)
        {
            Operation oper = null;
            switch (operate)
            {
                case "+":
                    oper = new OperationAdd();
                    break;
                case "-":
                    oper = new OperationSub();
                    break;
                case "*":
                    oper = new OperationMul();
                    break;
                case "/":
                    oper = new OperationDiv();
                    break;
            }
            return oper;
        }
    }
}
```



- **简单工厂模式优缺点与模式分析**
  - **优点**
    - 工厂类包含必要的逻辑判断，可以决定在什么时候创建哪一个产品的实例。客户端可以免除直接创建产品对象的职责
    - 客户端无需知道所创建具体产品的类名，只需知道参数即可
    - 也可以引入配置文件，在不修改客户端代码的情况下更换和添加新的具体产品类。
  - **缺点**
    - 工厂类集中了所有产品的创建逻辑，职责过重，一旦异常，整个系统将受影响
    - 使用简单工厂模式会增加系统中类的个数(引入新的工厂类)，增加系统的复杂度和理解难度
    - 系统扩展困难，一旦增加新产品不得不修改工厂逻辑，在产品类型较多时，可能造成逻辑过于复杂
    - 简单工厂模式使用了static工厂方法，造成工厂角色无法形成基于继承的等级结构。
  - **适用环境**
    - 工厂类负责创建的对象比较少，因为不会造成工厂方法中的业务逻辑过于复杂
    - 客户端只知道传入工厂类的参数，对如何创建对象不关心
  - **模式分析**
    - 将对象的创建和对象本身业务处理分离可以降低系统的耦合度，使得两者修改起来都相对容易
    - 在调用工厂类的工厂方法时，由于工厂方法是静态方法，使用起来很方便，可以通过类名直接调用，而且只需要传入一个简单的参数即可
    - 简单工厂模式最大问题就是在于工厂类的职责相对过重，增加新的产品需要工厂类的判断逻辑，这一点与开闭原则是相违背的
    - 简单工厂模式要点在于：当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无需知道其创建细节

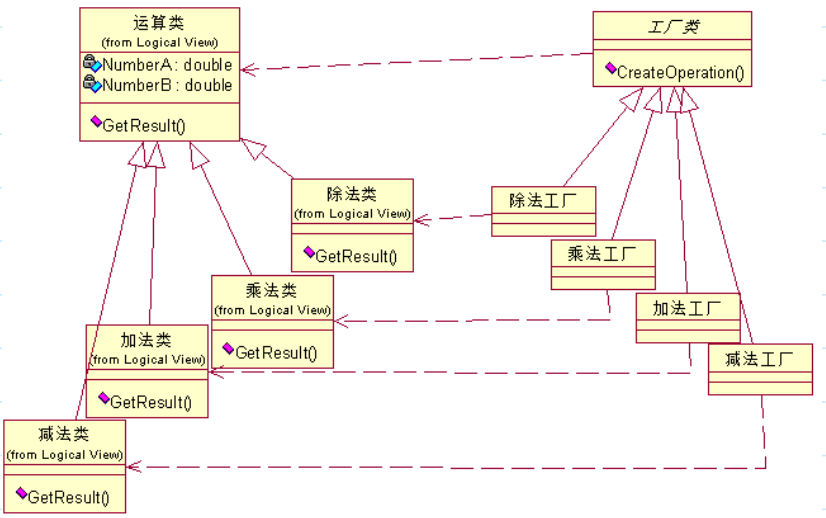
- **Q2：试画出工厂方法模式的模式结构图，并对模式进行分析。**
- **A2：**
  - **工厂模式概述**
    - **定义：**工厂方法模式又叫做工厂模式，也叫作虚拟构造器模式，或者多态工厂模式，它属于创建型模式。
    - 其意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，将核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品
  - **工厂方法模式的结构与模式结构图**
    - **抽象工厂：**是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象的工厂类必须实现这个接口。
    - **具体工厂：**是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象的工厂类必须实现这个接口。
    - **抽象产品：**工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。
    - **具体产品：**这个角色实现了抽象产品角色所定义的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应。



模式代码：

```
interface Ifactory
    {
        Operation CreateOperation();
    }
    class SubFactory  : Ifactory    //减法类工厂
    {
        public Operation CreateOperation()
        {
            return new OperationSub();
        }
    }
    class AddFactory : Ifactory    //加法类工厂
    {
        public Operation CreateOperation()
        {
            return new OperationAdd();
        }
    }
    class MulFactory : Ifactory    //乘法类工厂
    {
        public Operation CreateOperation()
        {
            return new OperationMul();
```

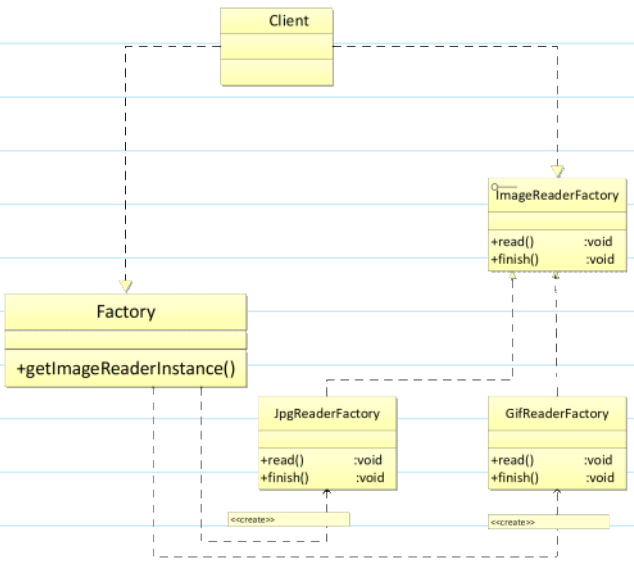
```
        }
    }
    class DivFactory : Ifactory    //除法类工厂
    {
        public Operation CreateOperation()
        {
            return new OperationDiv();
        }
    }
```



- **工厂方法模式优缺点与模式分析**
  - **优点**
    - 符合开闭原则，具有很强的的扩展性、弹性和可维护性。扩展时只要添加一个ConcreteCreator，而无须修改原有的ConcreteCreator，因此维护性也好。解决了简单工厂对修改开放的问题。
    - 使用了依赖倒置原则，依赖抽象而不是具体，使用（客户）和实现（具体类）松耦合。
    - 客户只需要知道所需产品的具体工厂，而无须知道具体工厂的创建产品的过程，甚至不需要知道具体产品的类名
  - **缺点**
    - 一个具体产品对应一个类，当具体产品过多时会使系统类的数目过多，增加系统复杂度。
    - 每增加一个产品时，都需要一个具体类和一个具体创建者，使得类的个数成倍增加，导致系统类数目过多，复杂性增加。
    - 对简单工厂，增加功能修改的是工厂类；对工厂方法，增加功能修改的是客户端。
  - **适用环境**
    - 当需要一个对象时，我们不需要知道该对象所对应的具体类，只要知道哪个具体工厂可以生成该对象，实例化这个具体工厂即可创建该对象。
    - 类的数目不固定，随时有新的子类增加进来，或者是还不知道将来需要实例化哪些具体类。
    - 定义一个创建对象接口，由子类决定要实例化的类是哪一个；客户端可以动态地指定工厂子类创建具体产品。
  - **模式分析**

- 工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。
- 在工厂方法模式中，核心的工厂类不再负责所有产品的创 建，而是将具体创建工作交给子类去做。
- 这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责哪一个产品类被实例化这种细节，这使得工厂 法模式可以允许系统在不修改工厂角色的情况下引进品。
- 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产品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，很好地符合了“开闭原则”。
- 而简单工厂模式在添加新产品对象后不得不修 改工厂方法，扩展性不好。
- 工厂方法模式退化后可以演变成简单工厂模式。

- **Q3：现需要设计一个程序来读取多种不同类型的图片格式，针对每一种图片格式都设计个图片读取器(ImageReader)，如GIF图片读取器(GifReader)用于读取GIF格式的图片、JPG图片读取器(Jpgreader)用于读取JPG格式的图片。图片读取器对象通过图片读取器工厂ImageReaderFactory来创建，ImageReaderFactory是一个抽象类，用于创建图片读取器的工厂方法，其子类GifReaderFactory和 JpgReaderFactory用于创建具体的图片读取器对象。使用工厂方法模式实现该程序的设计。*（画出模式结构图，并进行解析）***
- A2:



ImageReaderFactory为抽象工厂
JpgReaderFactory为具体工厂
GifReaderFactory为具体工厂
图片读取器工厂ImageReaderFactory创建图片读取器，ImageReaderFactory是一个抽象类创建图片读取器的工厂方法，其子类 GifReaderFactory和JpgReaderFactory创建具体的图片读取器对象，代码详解如下：

```
package com;

public interface ImageReaderFactory {
          // 读取
          public void read() ;
          // 完成
          public void finish() ;
}
public class JpgReaderFactory implements ImageReaderFactory{
     public void read()
     {
          System.out.println("读取JPG图片中... ....") ;
     }
     public void finish()
     {
          System.out.println("JPG图片读取完成") ;
     }
}
public class GifReaderFactory implements ImageReaderFactory{
     public void read()
     {
          System.out.println("读取GIF图片中... ... ") ;
     }
     public void finish()
     {
          System.out.println("GIF图片读取完成") ;
     }
}

class Factory
{
     public static ImageReaderFactory getImageReaderInstance(String
     type)
     {
          ImageReaderFactory f = null ;
          if("Jpg".equals(type))
          {
               f = new JpgReaderFactory() ;
          }
          if("Gif".equals(type))
          {
               f = new GifReaderFactory() ;
          }
          if("jpg".equals(type))
          {
               f = new JpgReaderFactory() ;
          }
          if("gif".equals(type))
          {
               f = new GifReaderFactory() ;
          }
          return f ;
     }
};
public class ImageReaderClient {
     public static void main(String[] args)
     {
          if(args.length==0)
          {
               System.out.println("请输入需要读取的图片类型") ;
               /*System.exit(1) ;*/
          }
          ImageReaderFactory f =
          Factory.getImageReaderInstance(args[0]) ;
          /*ImageReaderFactory f =
          Factory.getImageReaderInstance("gif") ;*/
          if(f!=null)
          {
               f.read() ;
               f.finish() ;
          }
          else
          {
               System.out.println("不支持该类型图片。") ;
          }
     }
}
```