

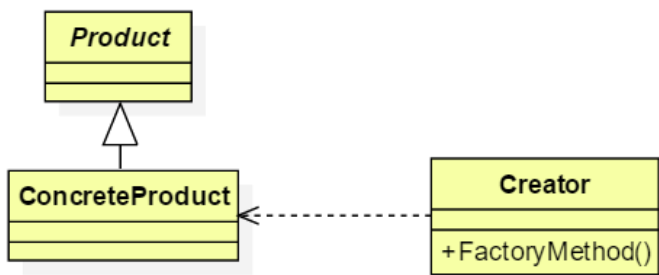
软件架构与设计模式

浙江省高等学校第九届青年教师教学技能竞赛

简单工厂模式 (Simple Factory Pattern)

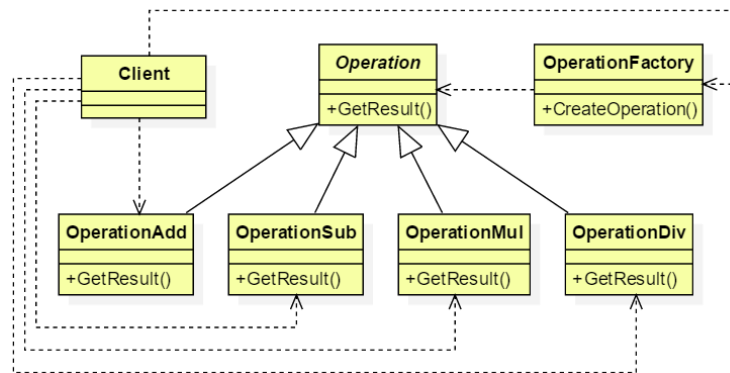
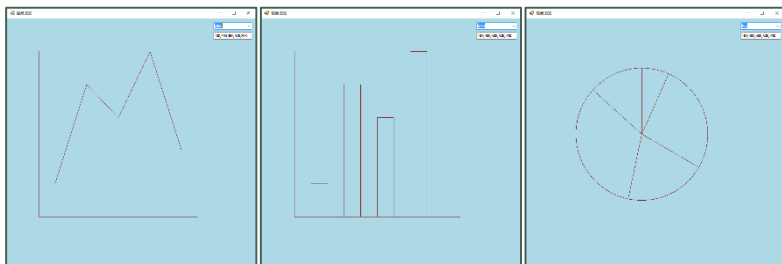
主要内容

- 面向过程的设计到简单工厂模式的**演进过程**
- 简单工厂模式的**结构**及**代码框架**
- 简单工厂模式在**实际工程**中的应用
- 简单工厂模式的**优缺点**及**适用范围**



重点

- 面向过程的设计到简单工厂模式的**演进过程**
- 简单工厂模式的**结构**

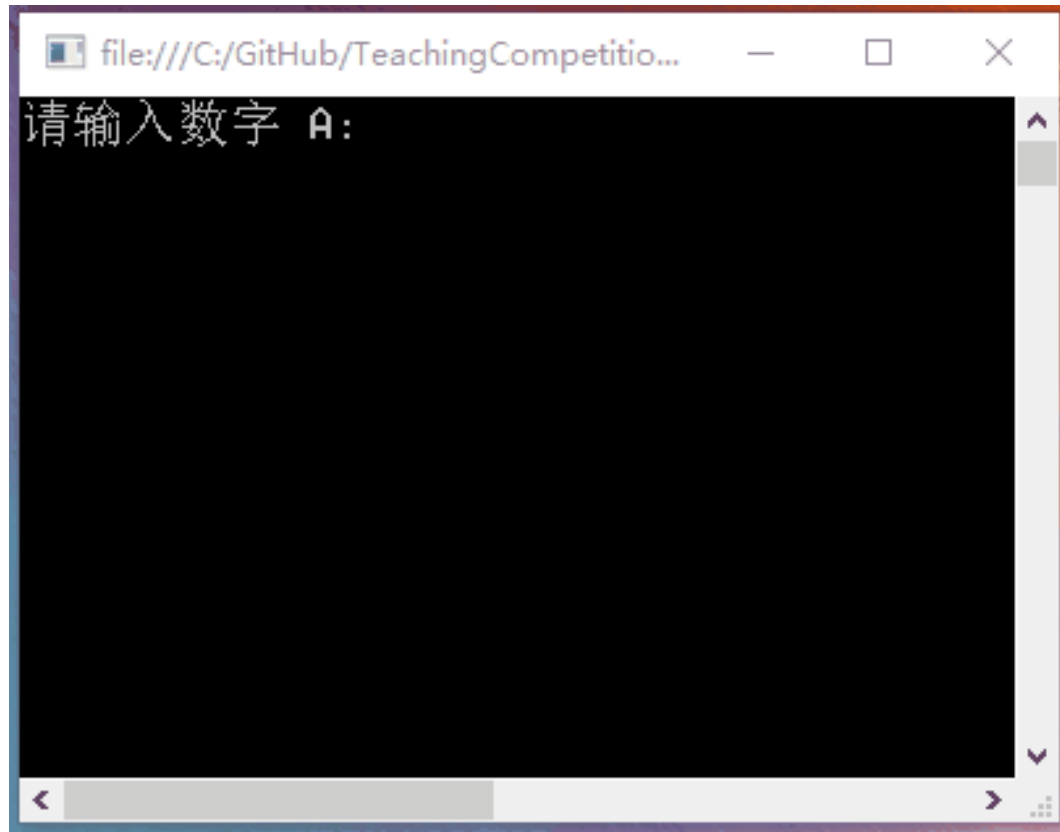


难点

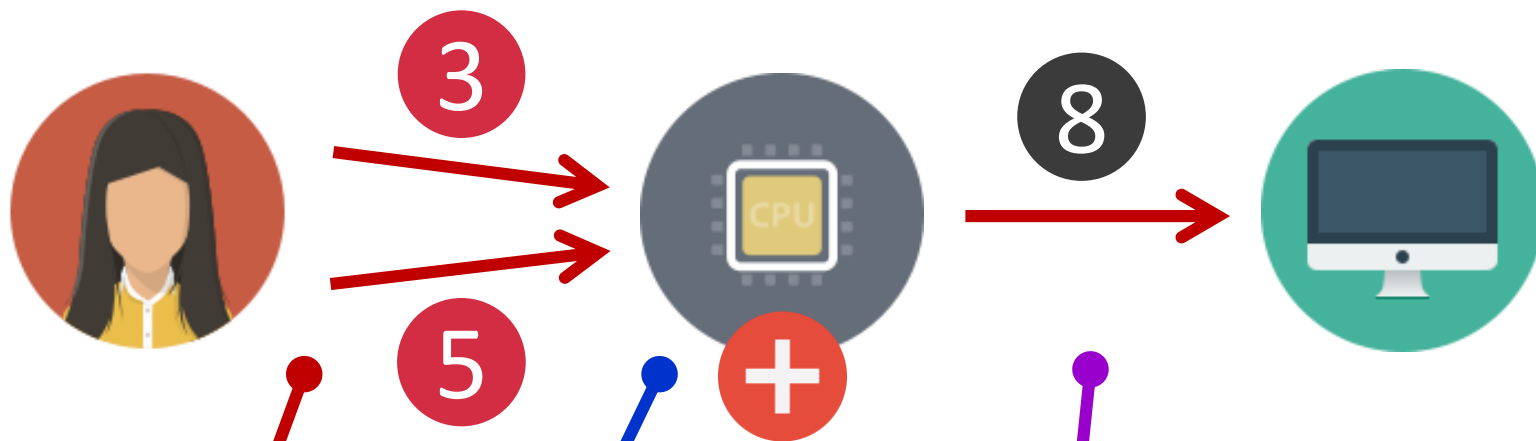
- 简单工厂模式在**实际工程**中的应用

一个很简单的需求

- 实现一个小程序
 - 让用户输入两个数，并求出它们的和。



计算机程序就是一组指令（程序代码）



```
Console.Write("请输入数字 A: ");  
double numberA = Convert.ToDouble(Console.ReadLine());  
Console.Write("请输入数字 B: ");  
double numberB = Convert.ToDouble(Console.ReadLine());
```

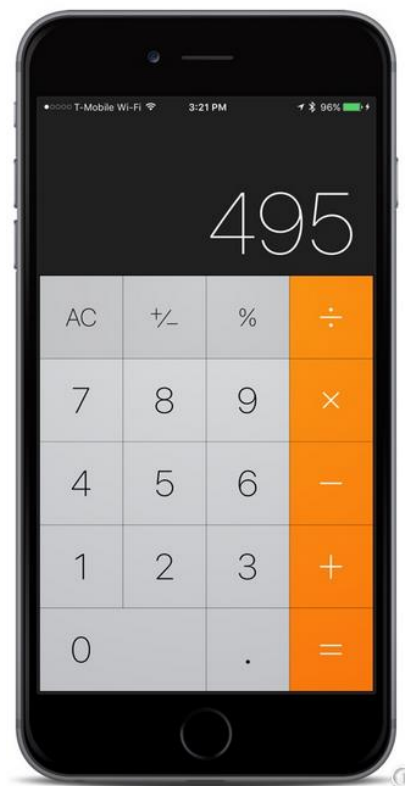
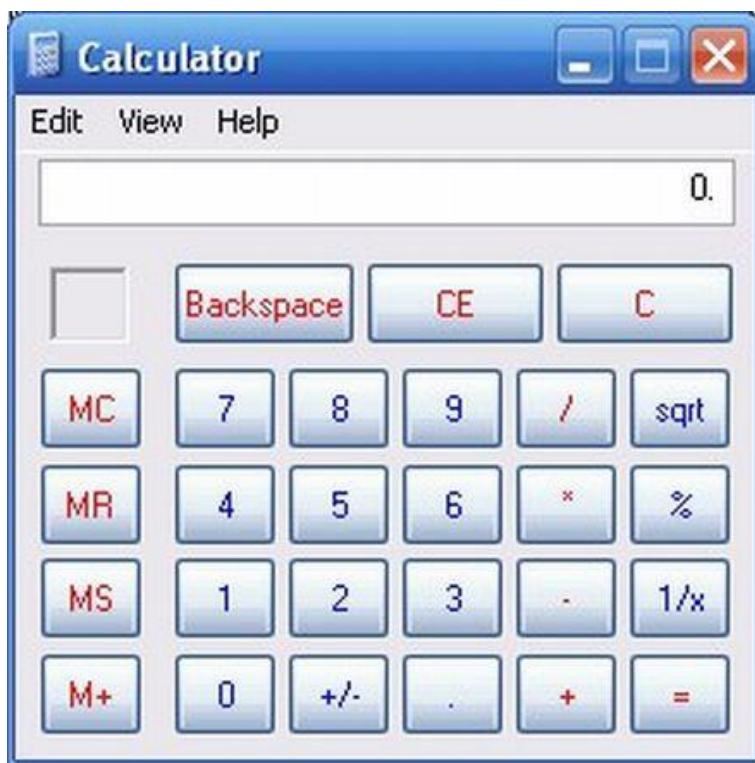
```
double result = numberA + numberB;
```

```
Console.WriteLine("运算结果是: " + result);  
Console.ReadLine();
```

Client

新的需求

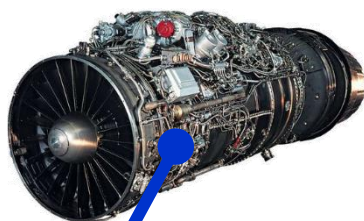
- 现在我们希望不但能在电脑上计算两数之和，还能在手机上计算两数之和。如何尽量重复使用已有的代码？



重复利用已有的代码

□ 软件也是由『零部件』组成的（组件）

□ 封装



```
Console.WriteLine("请输入数字 A: ");  
double numberA = Convert.ToDouble(Console.ReadLine());  
Console.WriteLine("请输入数字 B: ");  
double numberB = Convert.ToDouble(Console.ReadLine());  
double result = numberA + numberB;  
Console.WriteLine("运算结果是: " + result);  
Console.ReadLine();
```

业务代码

OperationAdd
+GetResult()

封装加法运算组件

- 将用于进行加法计算的代码封装为一个加法运算组件（OperationAdd），并作为一个方法（功能）提供给使用者。

类（组件）

```
class OperationAdd
```

```
{
```

```
    public double GetResult(double numberA, double numberB)
```

```
    {
```

```
        return numberA + numberB;
```

```
    }
```

```
}
```

方法（功能）

加法运算代码

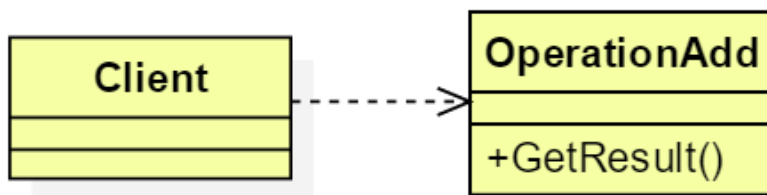
使用加法运算组件

- 需要进行加法运算时，首先创建一个加法组件，然后调用这个组件的 `GetResult()` 方法，得到计算结果。

```
Console.Write("请输入数字 A: ");  
double numberA = Convert.ToDouble(Console.ReadLine());  
Console.Write("请输入数字 B: ");  
double numberB = Convert.ToDouble(Console.ReadLine());  
  
OperationAdd add = new OperationAdd();  
double result = add.GetResult(numberA, numberB);  
  
Console.WriteLine("运算结果是: " + result);  
Console.ReadLine();
```

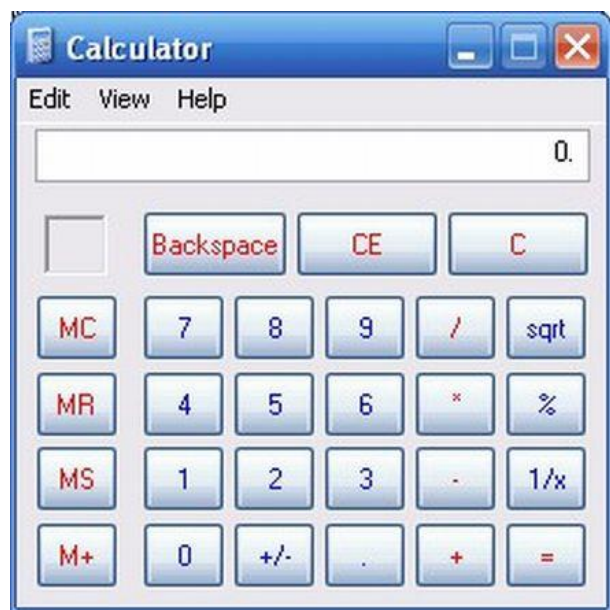
创建
加法
组件

调用
组件
功能

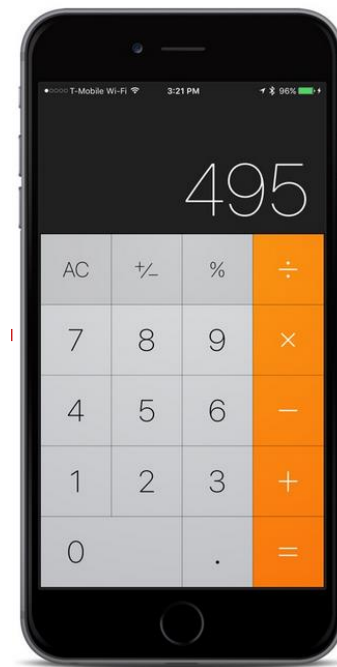


如果 UI 改变了

已有的加法组件**不需要做任何改变**，
就可以被不同的程序所使用

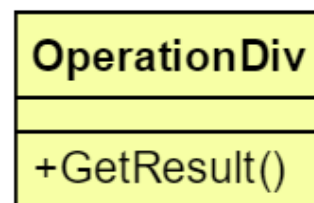
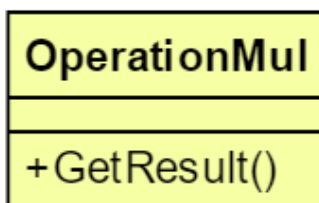
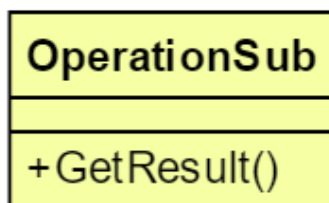
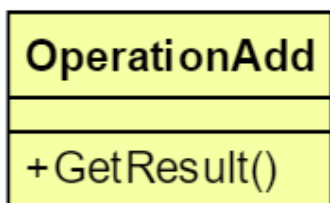


OperationAdd
+GetResult()



通过增加组件为程序添加新的功能

- 通过增加不同的运算组件，为程序添加新的运算类型

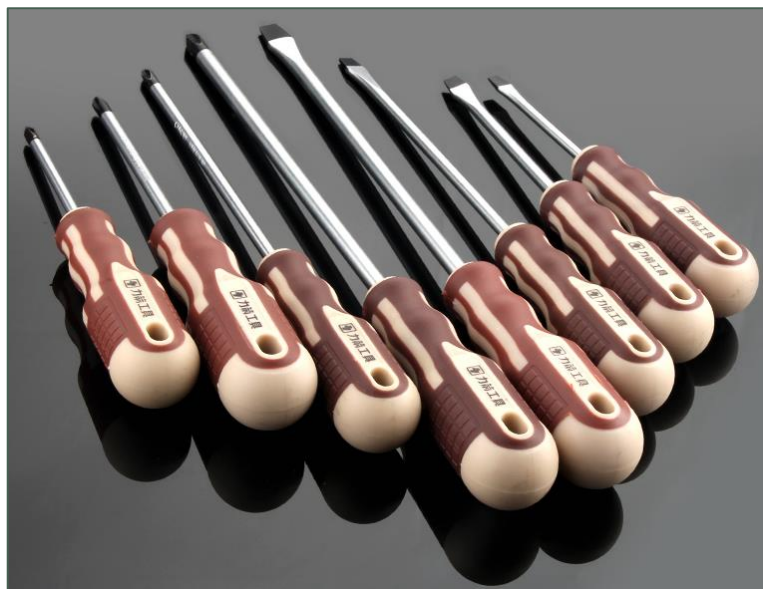


```
class OperationDiv
{
    public double GetResult(double numberA, double numberB)
    {
        return numberA / numberB;
    }
}
```

除法运算

假如未来还需要增加新的运算类型？

□ 如何保证未来增加的组件可以被现有的程序使用？

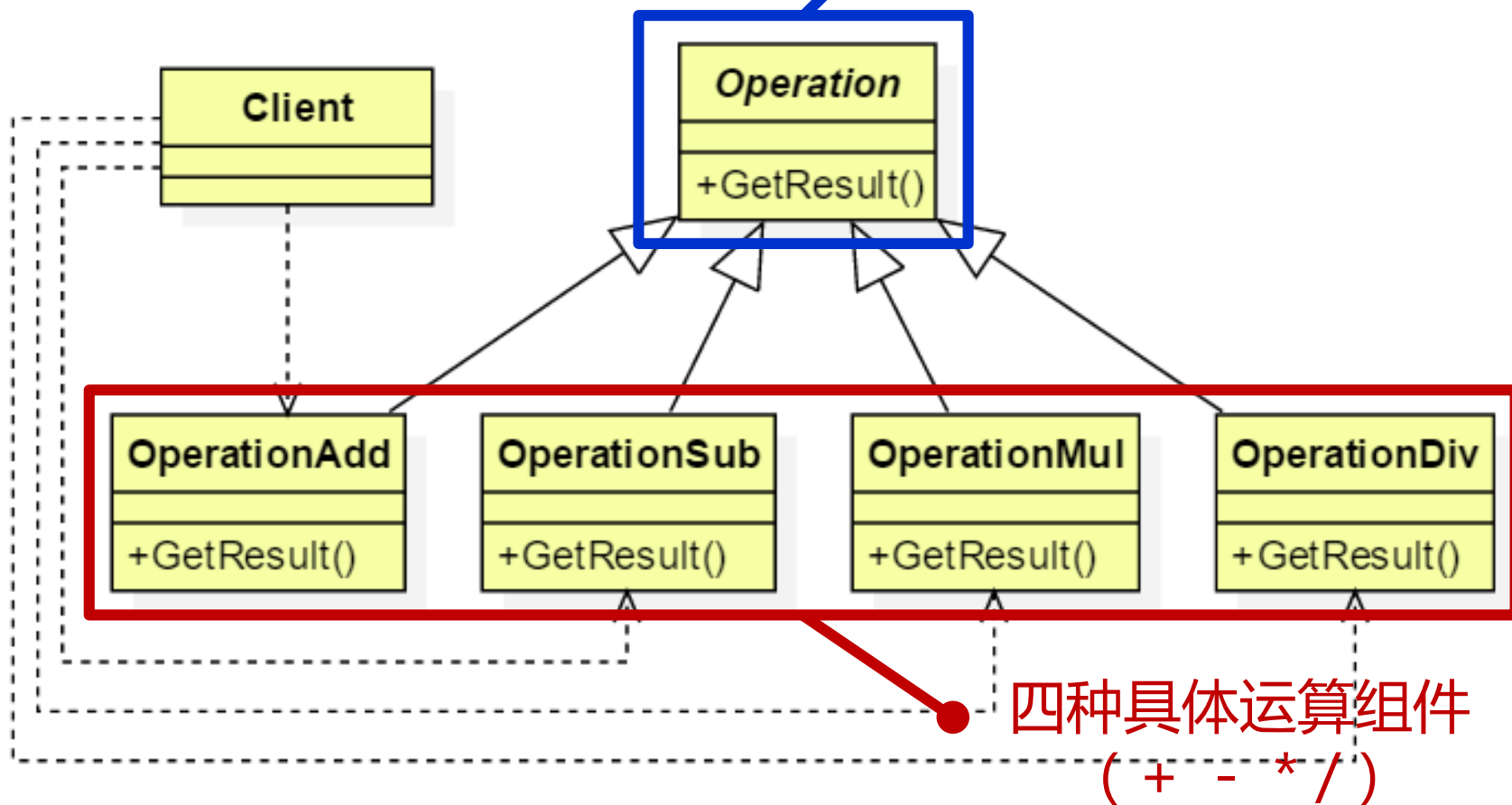


□ 抽取不同组件共同的特性，为不同的组件定义一个统一的接口。无论是组件还是使用者，都遵循这个接口标准。

定义接口之后

□ 用 UML 类图表示

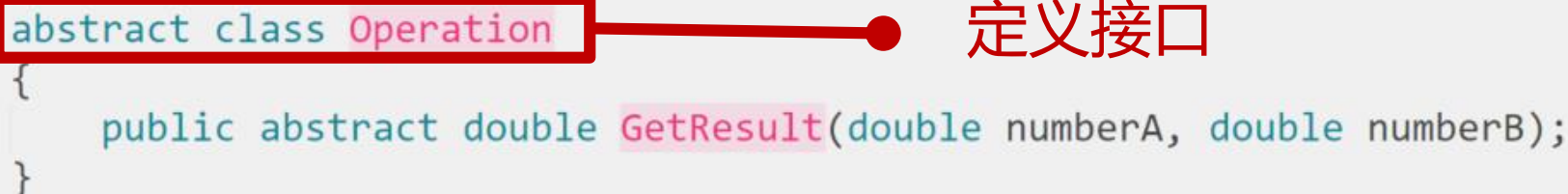
运算接口



使用接口之后

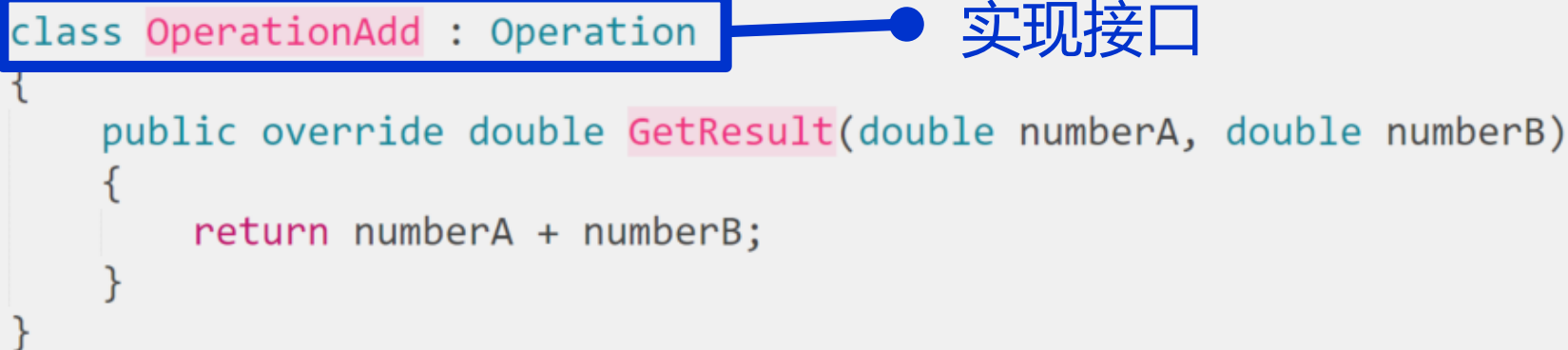
- 为四个运算组件定义统一的接口

```
abstract class Operation
{
    public abstract double GetResult(double numberA, double numberB);
}
```



- 不同的运算组件实现了这个接口

```
class OperationAdd : Operation
{
    public override double GetResult(double numberA, double numberB)
    {
        return numberA + numberB;
    }
}
```



使用接口之后

- 用一致的方式使用四种不同的运算组件

不同的组件

```
Operation add = new OperationAdd();  
double result1 = add.GetResult(numberA, numberB);
```

```
Operation sub = new OperationSub();  
double result2 = sub.GetResult(numberA, numberB);
```

```
Operation mul = new OperationMul();  
double result3 = mul.GetResult(numberA, numberB);
```

```
Operation div = new OperationDiv();  
double result4 = div.GetResult(numberA, numberB);
```

相同的接口

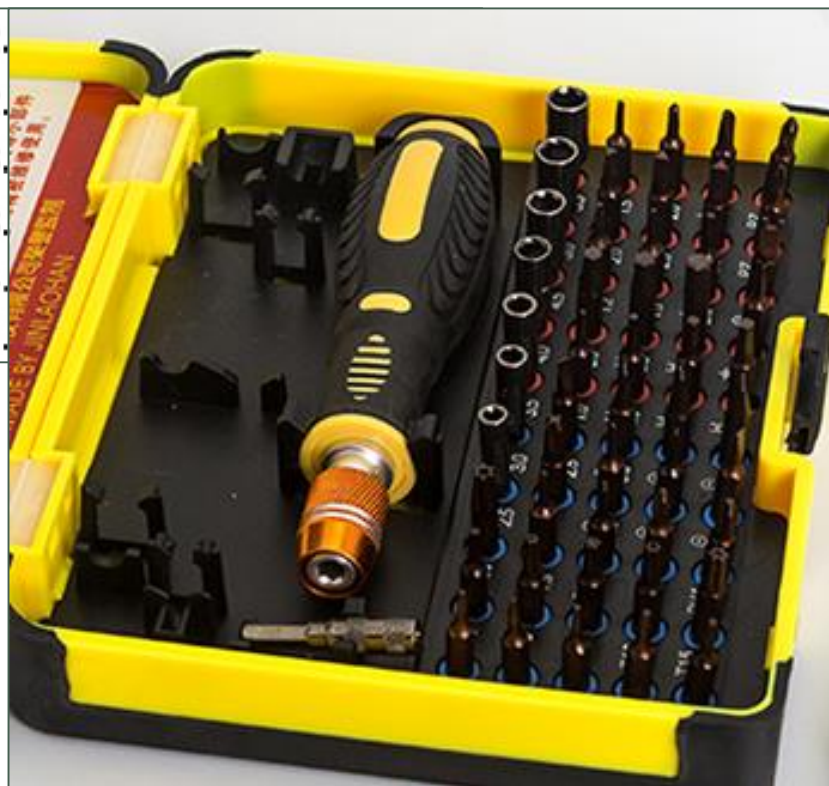
重要的新需求

- 现在用户希望通过输入不同的运算符，进行不同的运算。
- 就好像根据不同的条件，选择不同的工具。

★ T2	★ T7	★ T20	● H3.0	⊖ 2.0
★ T3	★ T8	● H1.0	● H3.5	⊖ 2.5
★ T4	★ T9	● H1.5	● H4.0	⊖ 3.0
★ T5	★ T10	● H2.0	⊖ 1.3	⊖ 3.5
★ T6	★ T15	● H2.5	⊖ 1.5	⊖ 4.0
▲ 2.3	▲ 2.6	⊙ Y1.5	⊙ Y3.0	★ 1.0



如何根据不同条件，得到对应的组件呢？



生活中的例子

□ 如果我们想吃炸鸡腿、汉堡包、鸡米花？



如何从 KFC 得到食物呢？

▫ 让别人帮我们做

- 我们只需要去 KFC，告诉店员我们需要的食物类型，制作食物的这个过程，就由 KFC 来完成，我们只管得到成品就可以了。



▫ 获取产品的思路

- 将创建产品的过程封装到一个专门的组件（工厂类）中，由它专门负责创建产品的操作。

这种思路就是简单工厂模式

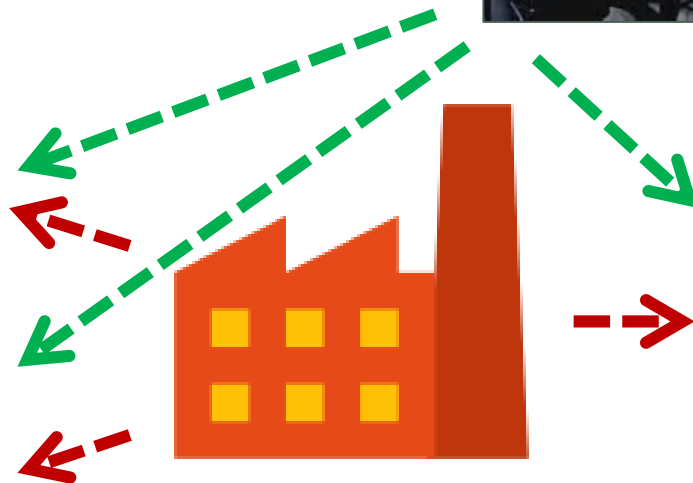
简单工厂模式

□ 定义

- 根据外界提供的条件，创建几种可能的产品中的某一种，这些产品通常都具有相同的接口。

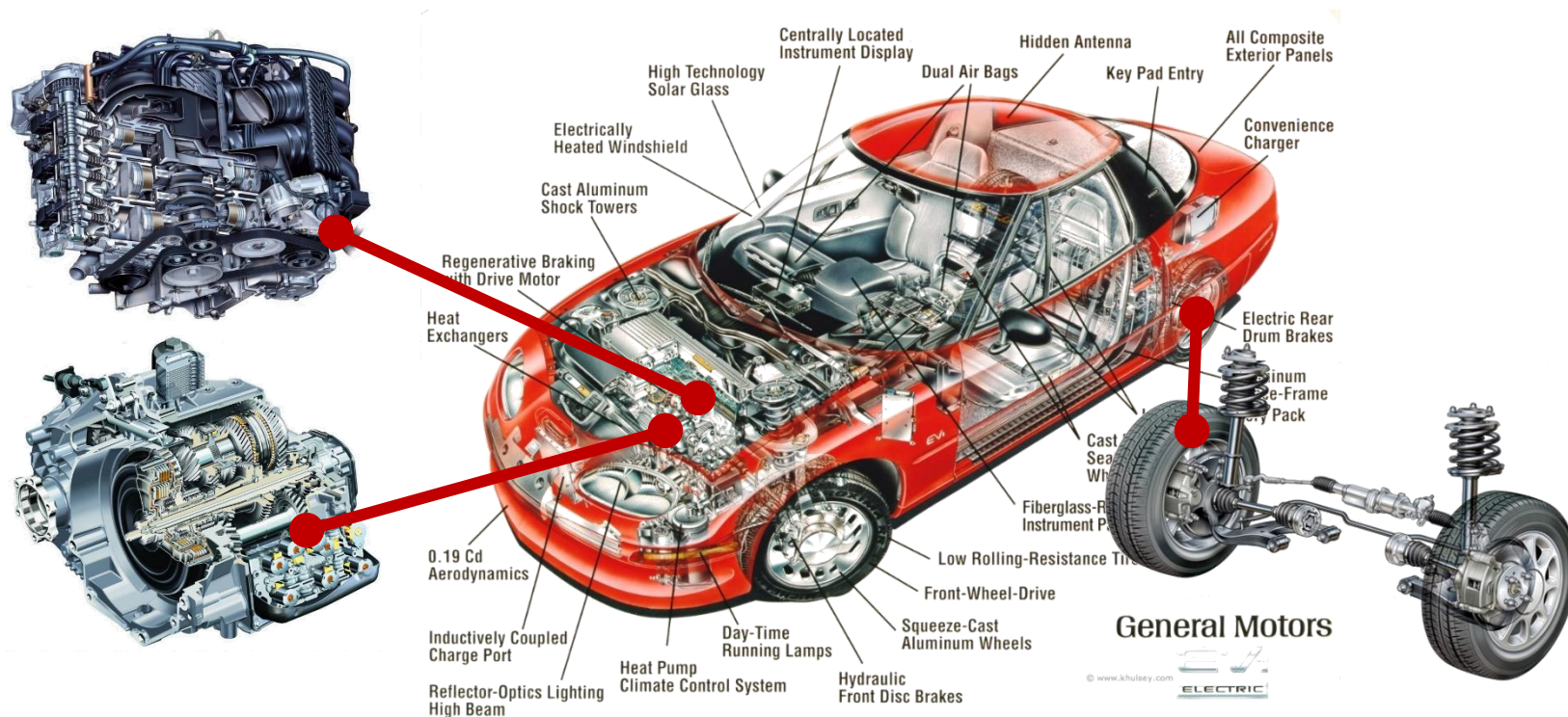
□ 意图

- 将产品的创建过程与使用过程分离。



简单工厂模式

在实际工程中，我们所需要得到的组件可能相当复杂



简单工厂模式能简化得到组件的过程

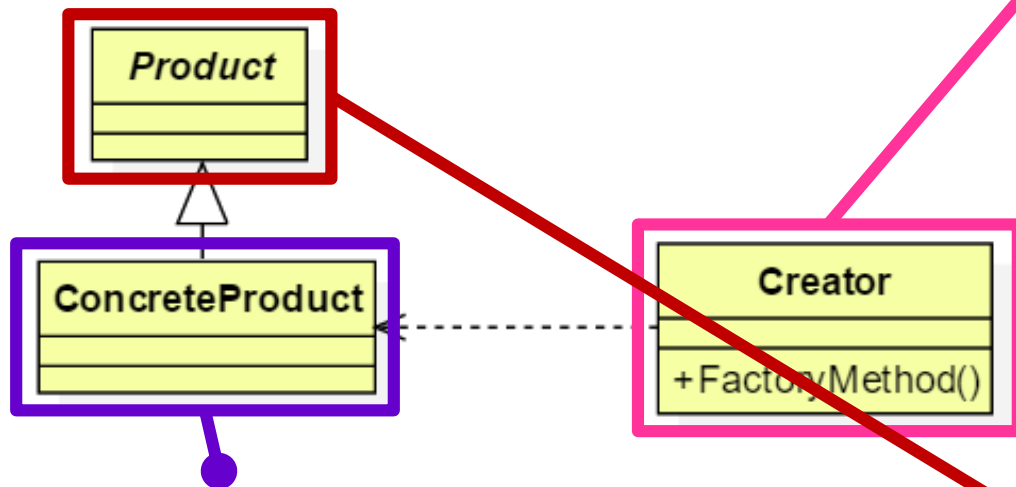
简单工厂模式

□ 比如：简单工厂模式在游戏中应用十分广泛



简单工厂模式的结构

□ 模式结构 UML 图



创建者 (Creator)

- 是简单工厂模式的核心，负责实现创建所有具体产品的实例。创建者可以被外界直接调用，用于创建所需的产品对象。

□ 具体产品 (ConcreteProduct)

- 实现了产品接口，一般为多个，是简单工厂模式的创建目标。创建者返回的都是某一种具体产品。

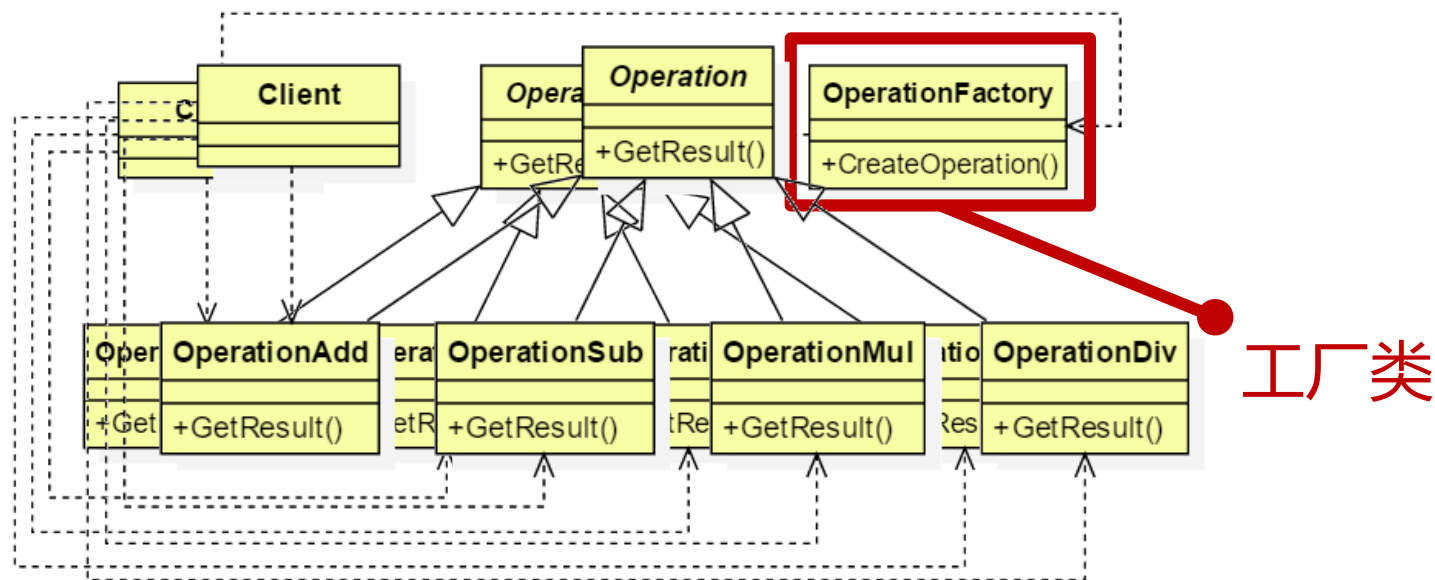
抽象产品 (Product)

- 是所有具体产品的父类，它负责描述所有具体产品所共有的接口。

在四则运算程序中引入简单工厂模式

□ 分析

- 程序中已经有了一个抽象产品（运算接口）与四个具体产品（运算组件）。
现在只需要引入一个工厂类，负责创建具体的运算对象。



如何告诉工厂我们需要哪种产品呢？

工厂类代码实现

□ OperationFactory 类

```
class OperationFactory
{
    public static Operation CreateOperation(string oper)
    {
        switch (oper)
        {
            case "+":
                return new OperationAdd();
            case "-":
                return new OperationSub();
            case "*":
                return new OperationMul();
            case "/":
                return new OperationDiv();
            default:
                return null;
        }
    }
}
```

创建者
(工厂类)

参数
(产品类型)

用于创建产品的功能
(工厂方法)

通过参数值决定创建哪种具体产品

客户端代码实现

客户端代码

根据运算符，得到所需的产品——运算组件

```
Console.Write("请输入数字 A: ");
double numberA = Convert.ToDouble(Console.ReadLine());
Console.Write("请选择运算符(+ - * /): ");
string oper = Console.ReadLine();
Console.Write("请输入数字 B: ");
double numberB = Convert.ToDouble(Console.ReadLine());

Operation operation = OperationFactory.CreateOperation(oper);
double result = operation.GetResult(numberA, numberB);

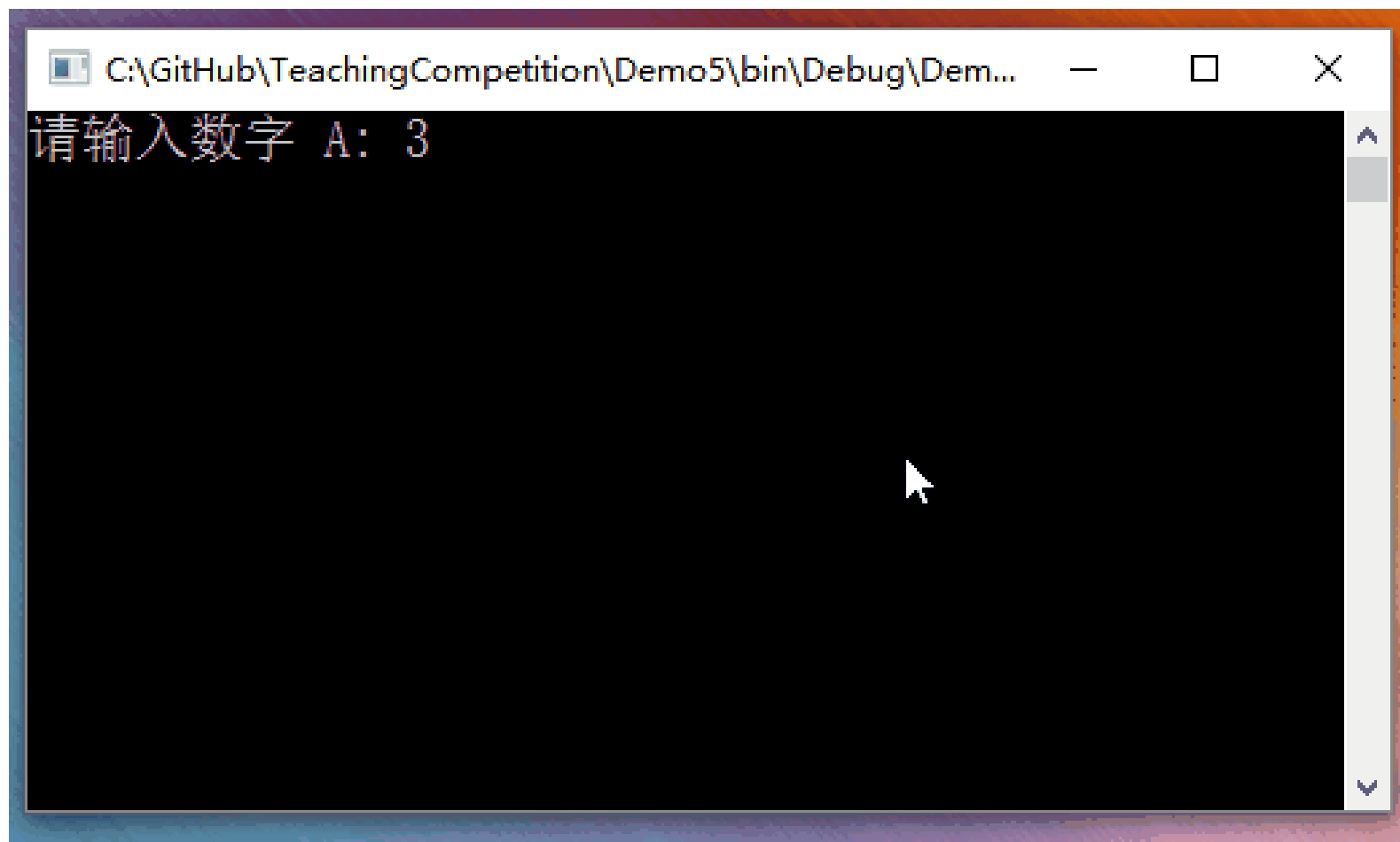
Console.WriteLine("运算结果是: " + result);
Console.ReadLine();
```

传入运算类型

调用运算组件的功能

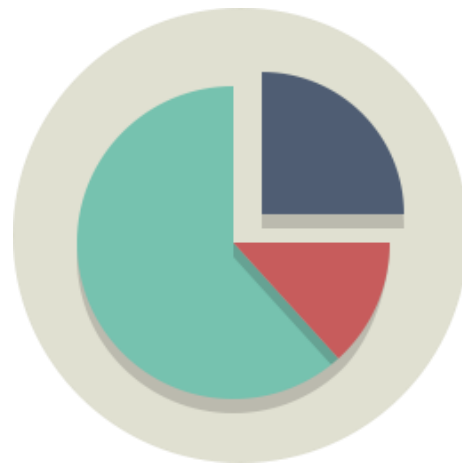
使用了简单工厂模式之后的程序

- 实现了简单工厂模式的程序运行结果



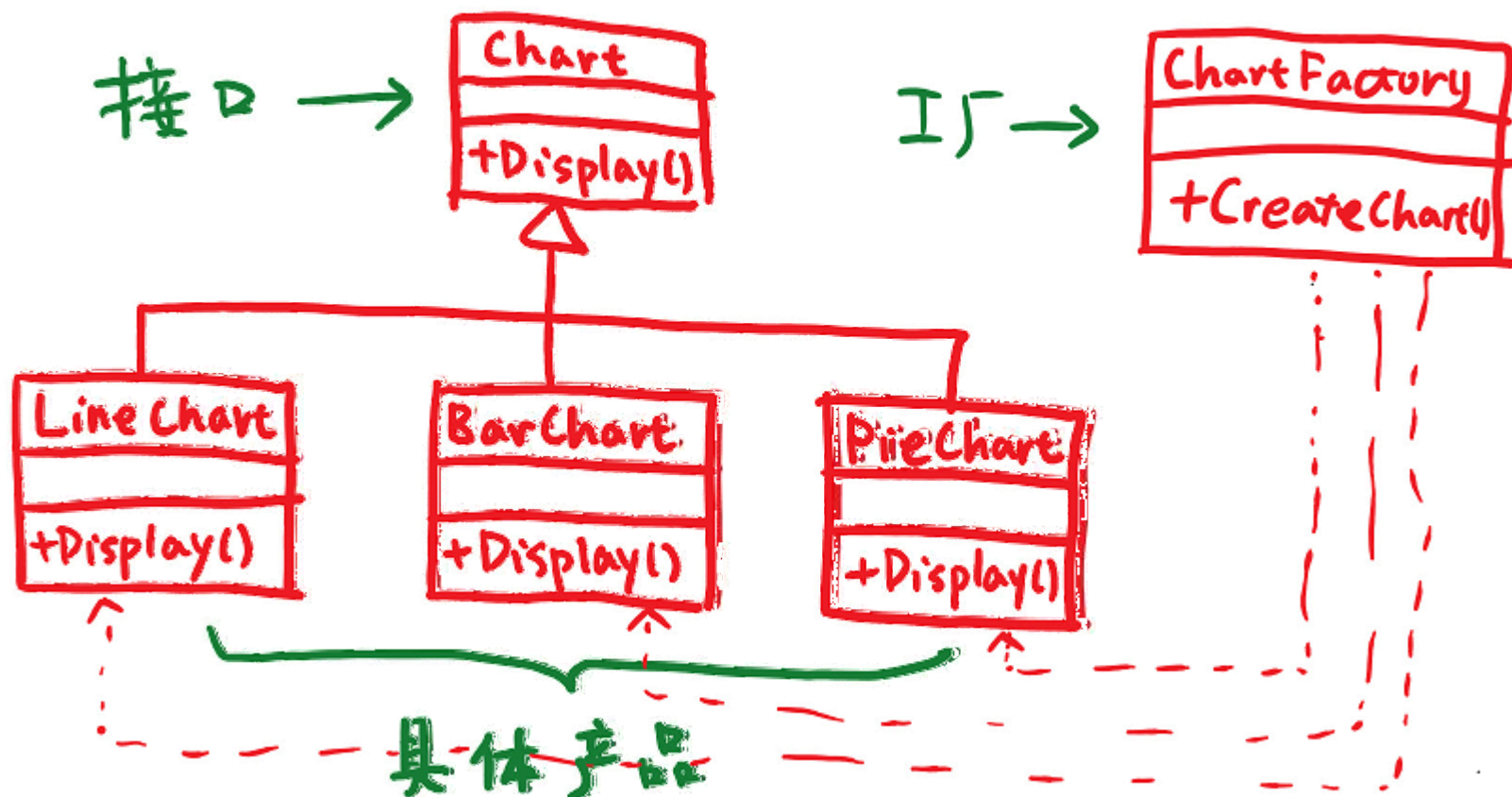
使用了简单工厂模式的简易图表程序

- 功能：可以针对相同的数据，以不同的图表类型加以显示



使用了简单工厂模式的简易图表程序

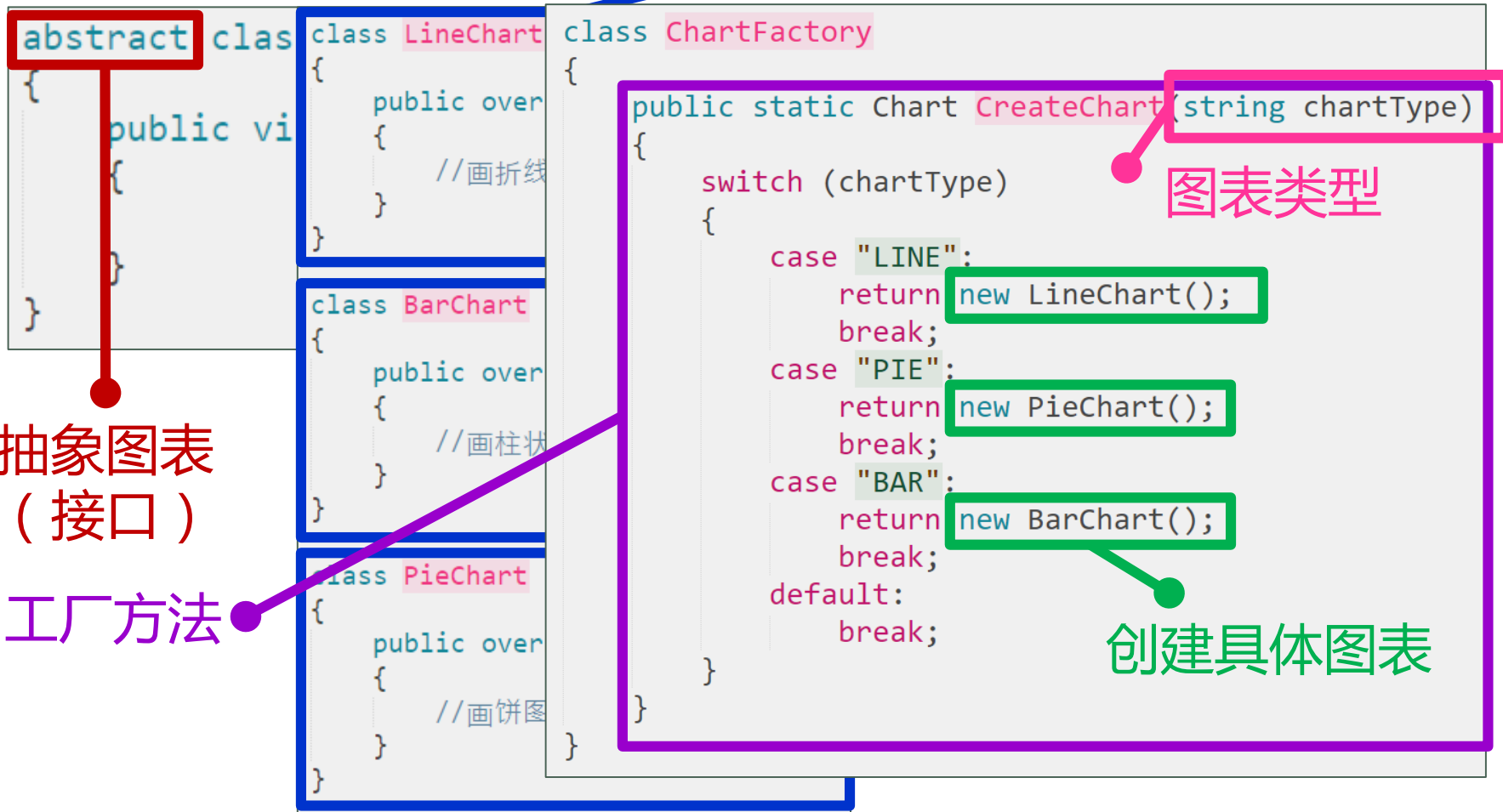
▣ STEP1：手绘出 UML 类图



使用了简单工厂模式的简易图表程序

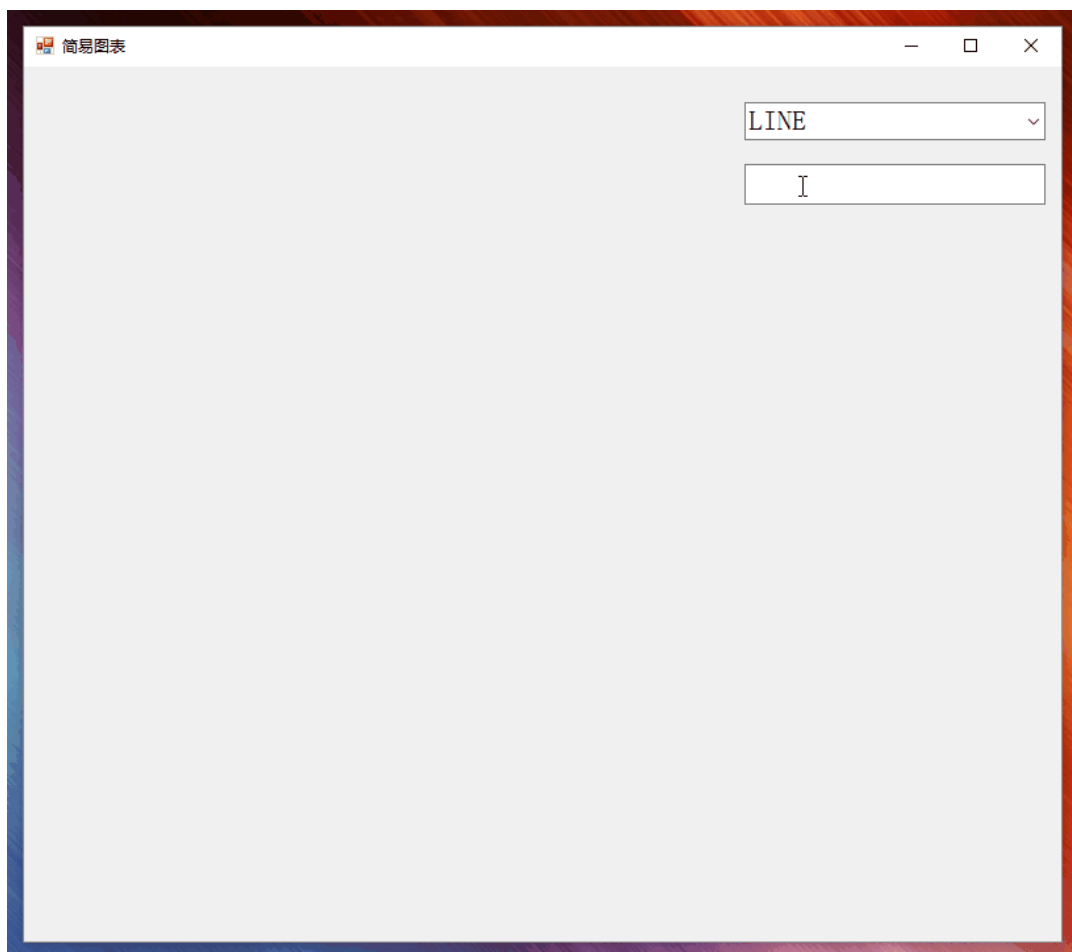
STEP2：写出程序的代码框架

具体图表（组件）



使用了简单工厂模式的简易图表程序

▣ 运行效果



简单工厂模式在:

Java 的 JDK 中的加密

```
//使用DES算法的密码器  
Cipher cp=Cipher.getInstance("DES");  
//使用AES算法的密码器  
Cipher cp=Cipher.getInstance("AES");  
//使用RSA算法的密码器  
Cipher cp=Cipher.getInstance("RSA");
```

内部实现

- Cipher.java
(加密算法工厂类)

根据不同的算法
类型，创建不同的
加密算法组件

```
public static final Cipher getInstance(String transformation)
    throws NoSuchAlgorithmException, NoSuchPaddingException
{
    List<Transform> transforms = getTransforms(transformation);
    List<ServiceId> cipherServices = new ArrayList<>(transforms.size());
    for (Transform transform : transforms) {
        cipherServices.add(new ServiceId("Cipher", transform.transform));
    }
    List<Service> services = GetInstance.getServices(cipherServices);
    // make sure there is at least one service from a signed provider
    // and that it can use the specified mode and padding
    Iterator<Service> t = services.iterator();
    Exception failure = null;
    while (t.hasNext()) {
        Service s = t.next();
        if (JceSecurity.canUseProvider(s.getProvider()) == false) {
            continue;
        }
        Transform tr = getTransform(s, transforms);
        if (tr == null) {
            // should never happen
            continue;
        }
        if (!tr.supportsPadding()) {
            if (canuse == S_NO) {
                // does not support mode or padding we need, ignore
                continue;
            }
            if (canuse == S_YES) {
                return new Cipher(null, s, t, transformation, transforms);
            } else { // S_MAYBE, try out if it works
                try {
                    CipherSpi spi = (CipherSpi)s.newInstance(null);
                    tr.setModePadding(spi);
                    return new Cipher(spi, s, t, transformation, transforms);
                } catch (Exception e) {
                    failure = e;
                }
            }
        }
    }
    throw new NoSuchAlgorithmException
        ("Cannot find any provider supporting " + transformation, failure);
}
```

简单工厂模式总结

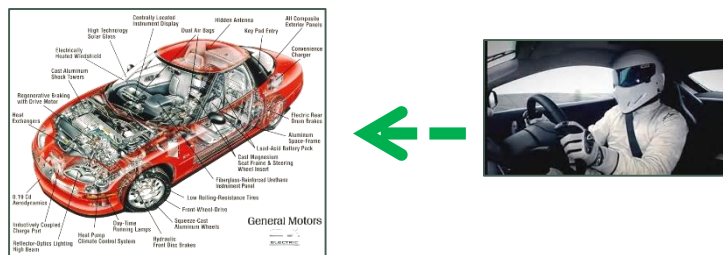
作用

- 根据外部条件，创建不同类型的产品。



优点

- 将产品的创建和产品的使用分离。

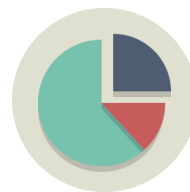


缺点

- 如果产品类型过多，工厂类的代码会变得非常复杂。

适用场景

- 工厂类负责创建的产品类型比较少。



课后作业

- 现有三种僵尸 (Zombie)
 - 普通僵尸 (CommonZombie)
 - 撑杆僵尸 (PoleZombie)
 - 气球僵尸 (BalloonZombie)
- 希望有一个控制台程序，能够：
 1. 随机生成一大波 (200个) 僵尸。
 2. 生成 10 个僵尸，用户可以指定每个僵尸的类型。



使用 C# / Java 或其他 OOP 语言实现此程序。



将 UML 类图及源代码以博客的形式发布在[博客园](#)网站上。



将程序代码提交到代码托管网站 [GitHub](#)。

本节课内容回顾

