王陆

那答案就是：我可以“保证”麻衣同学有幸福的未来，我们两个今后会得到充实的“保障”，对吗？

随笔 - 655 文章 - 0 评论 - 438
博主考研复习中，加油，奥利给！

C++设计模式——简单工厂模式与策略模式比较

简单工厂模式本应该放到工厂模式那篇博客中去介绍的，因为与策略模式有一定的相似性，这里摘出来单独成章。

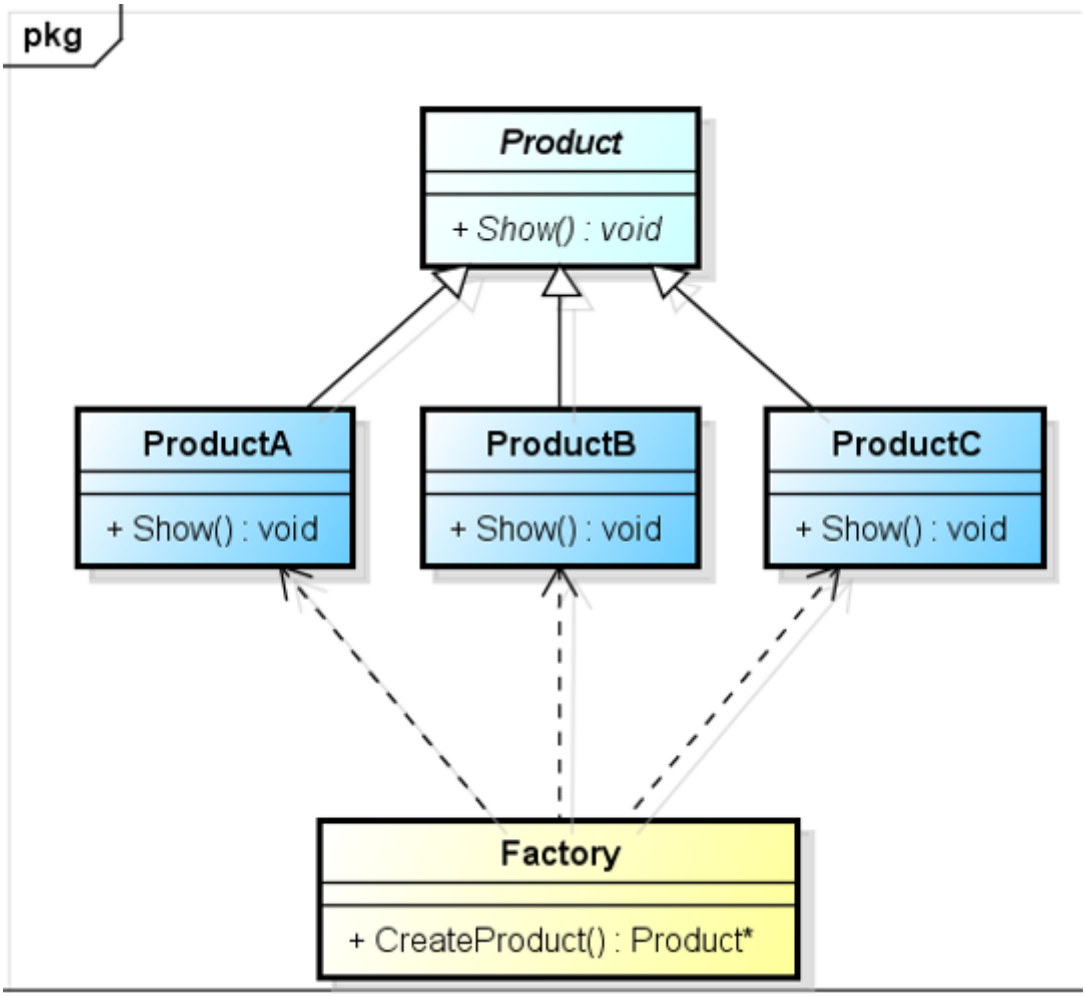
简单工厂模式

用一个单独的类来做创造实例的过程，就是工厂。

工厂模式有一种非常形象的描述：**建立对象的类就如一个工厂，而需要被建立的对象就是一个个产品；在工厂中加工产品，使用产品的人，不用在乎产品是如何生产出来的。**从软件开发的角度来说，这样就有效的降低了模块之间的耦合。

适用场合

在程序中，需要创建的对象很多，导致对象的new操作多且杂时，需要使用简单工厂模式；
由于对象的创建过程是我们不需要去关心的，而我们注重的是对象的实际操作，所以，我们需要分离对象的创建和操作两部分，如此，方便后期的程序扩展和维护。



基本代码

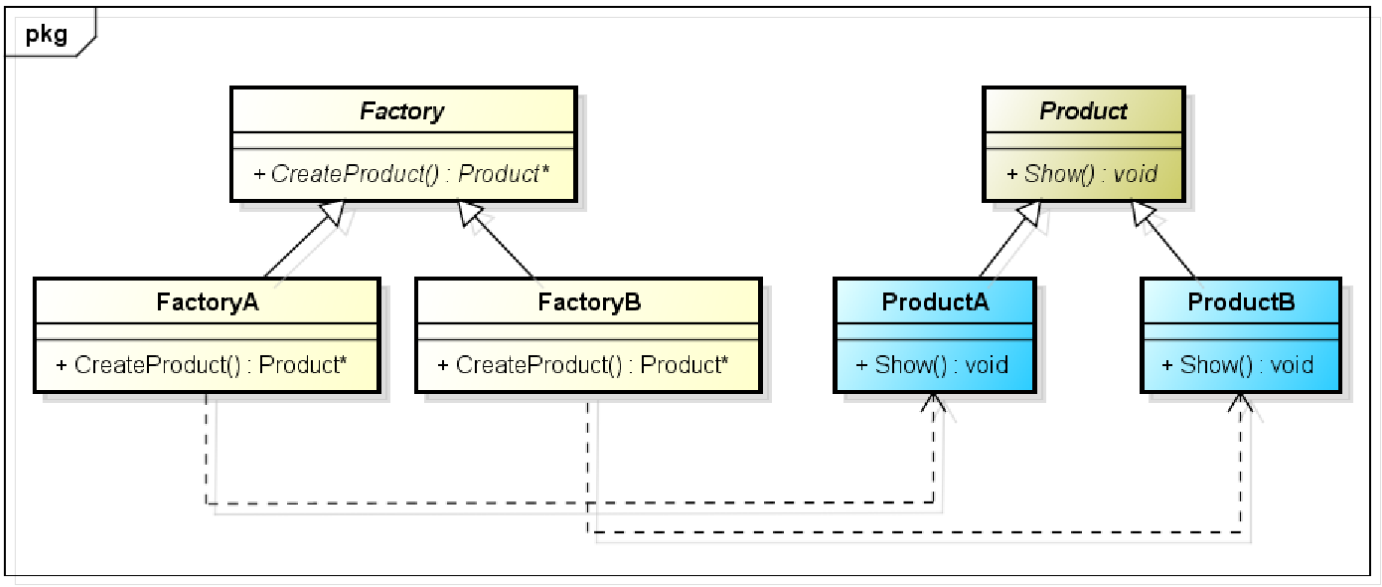
```
typedef enum ProductTypeTag
{
    TypeA,
    TypeB,
    TypeC
} PRODUCTTYPE;

class Product
{
public:
    virtual void Show() = 0;
};

class ProductA : public Product
{
public:
    void Show()
    {
        cout<<"I'm ProductA"<<endl;
    }
};
class ProductB : public Product
{
public:
    void Show()
```

```
    {
        cout<<"I'm ProductB"<<endl;
    }
};

class ProductC : public Product
{
public:
    void Show()
    {
        cout<<"I'm ProductC"<<endl;
    }
};
class Factory
{
public:
    Product* CreateProduct(PRODUCTTYPE type)
    {
        switch (type)
        {
        case TypeA:
            return new ProductA();
        case TypeB:
            return new ProductB();
        case TypeC:
            return new ProductC();
        default:
            return NULL;
        }
    }
};
int main(int argc, char *argv[])
{
    Factory *ProductFactory = new Factory();
    Product *productObjA = ProductFactory->CreateProduct(TypeA);
    if (productObjA != NULL)
        productObjA->Show();
    Product *productObjB = ProductFactory->CreateProduct(TypeB);
    if (productObjB != NULL)
        productObjB->Show();
    Product *productObjC = ProductFactory->CreateProduct(TypeC);
    if (productObjC != NULL)
        productObjC->Show();
    return 0;
}
```

简单工厂模式的局限性

若工厂现在能生产ProductA、ProductB和ProductC三种产品了，此时，需要增加生产ProductD产品；那么，首先是不是需要在产品枚举类型中添加新的产品类型标识，然后，修改Factory类中的switch结构代码。是的，这种对代码的修改，对原有代码的改动量较大，易产生编码上的错误（虽然很简单，如果工程大了，出错也是在所难免的！！！）。同时，由于对已经存在的函数进行了修改，那么以前进行过的测试，都将是无效的，所有的测试，都将需要重新进行，所有的代码都需要进行重新覆盖。

工厂模式的引入



FactoryA专心负责生产ProductA，FactoryB专心负责生产ProductB，FactoryA和FactoryB之间没有关系；如果到了后期，如果需要生产ProductC时，我们则可以创建一个FactoryC工厂类，该类专心负责生产ProductC类产品。由于FactoryA、FactoryB和FactoryC之间没有关系，当加入FactoryC加入时，对FactoryA和FactoryB的工作没有产生任何影响，那么对代码进行测试时，只需要单独对FactoryC和ProductC进行单元测试，而FactoryA和FactoryB则不用进行测试，可省去大量无趣无味的测试工作。

```
class Product
{
public:
    virtual void Show() = 0;
};

class ProductA : public Product
{
public:
    void Show()
    {
        cout<< "I'm ProductA"<<endl;
    }
};
class ProductB : public Product
{
public:
    void Show()
    {
        cout<< "I'm ProductB"<<endl;
    }
};
class Factory
{
public:
    virtual Product *CreateProduct() = 0;
```

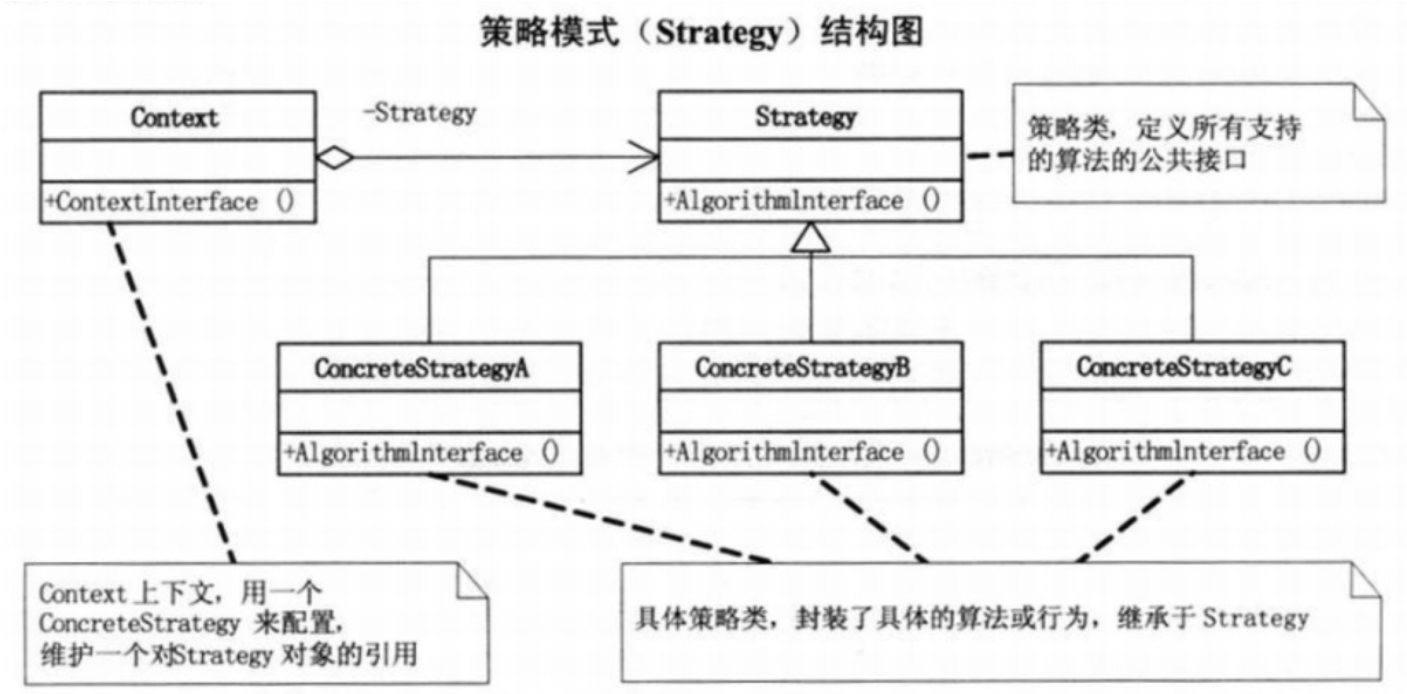
```
};
class FactoryA : public Factory
{
public:
    Product *CreateProduct()
    {
        return new ProductA ();
    }
};
class FactoryB : public Factory
{
public:
    Product *CreateProduct()
    {
        return new ProductB ();
    }
};
int main(int argc, char *argv [])
{
    Factory *factoryA = new FactoryA ();
    Product *productA = factoryA->CreateProduct();
    productA->Show();
    Factory *factoryB = new FactoryB ();
    Product *productB = factoryB->CreateProduct();
    productB->Show();

    return 0;
}
```

详细内容参见上一篇博客**C++设计模式——工厂模式Factory Method**

简单工厂模式和策略模式的比较

看到这个UML图回想一下 之前学习过的**策略模式**

二者好像差不多？



唯一不同的就是 简单工厂类 和 Context类。

转载自https://blog.csdn.net/zwj_jyzl/article/details/80869905
原博客是用Java实现的，由于并不影响阅读与理解，在这里直接拷贝过来，没有做修改

简单工厂类和Context类中代码的区别

简单工厂类：

```
public class OperationFactory
{
    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;
            default:
                oper = new Operation();
                break;
        }
```

```
        return oper;
    }
}
```

策略模式中的Context类：

```
class Context
{
    CashSuper csuper;
    public Context(CashSuper cs)
    {
        this.csuper = cs;
    }
    public double GetResult(double money)
    {
        //调用具体策略类的收费方法
        return csuper.acceptCash(money);
    }
}
```

1.首先看一下接收的参数：简单工厂类中的 CreateOperate 方法接收的是字符串，返回的是一个 Operation 对象；而 Context 类初始化时需要接收一个 CashSuper 对象。
2.简单工厂类中是根据接收的条件创建一个相应的对象，而 Context 类接收的是一个对象，可以调用方法去执行此对象的方法。
总结简单工厂模式和策略模式

1.从类型上说：简单工厂模式属于创建型模式，而策略模式属于行为型模式。
2.接下来，看一个小例子：

斧子有很多种，有一个工厂专门负责生产各种需求的斧子。

工厂模式：
1）根据你给出的目的来生产不同用途的斧子，例如要砍人，那么工厂生产砍人斧子，要伐木就生产伐木斧子。
2）即根据你给出一些属性来生产不同行为的一类对象返回给你。
3）关注对象创建
策略模式：
1）用工厂生产的斧子来做对应的事情，例如用砍人的斧子来砍人，用伐木的斧子来伐木。
2）即根据你给出对应的对象来执行对应的方法。
3）关注行为的选择

3.简单工厂模式：根据客户选择的条件，来帮客户创建一个对象。
策略模式：客户给它一个创建好的对象，它来帮客户做相应的事。
两种模式的优缺点

首先来看一下两种模式的客户端代码：

```
//简单工厂模式的客户端：
Operation op;
//交给简单工厂类创建对象
op = OperationFactory.CreateOperate("+");
op.StrNumberA = 10;
op.StrNumberB = 20;
//调用生成对象的方法
double result = op.GetResult();
Console.WriteLine(result);
```

```
//策略模式的客户端：
double total = 0;
private void btnOk_Click(object sender, EventArgs e)
{
    CashContext cc = null;
    //客户端自己创建对象
    switch(cbxType.SelectedItem.ToString())
    {
        case: "正常收费":
            cc = new CashContext(new CashNormal());
            break;
        case: "满300返100":
            cc = new CashContext(new CashReturn());
            break;
        case: "打8折":
            cc = new CashContext(new CashRebate());
            break;
    }
    //计算具体策略收取的费用，交给context类执行相应的方法，客户端只需要接收返回的值就可以
    double acceptMoney = cc.GetResult(Convert.ToDouble(txtPrice.Text) * Convert.ToDouble(txtNum.
    //计算总费用
    total += acceptMoney;
    listBox1.Items.Add("单价：" + txtPrice.Text + " 数量：" + txtNum.Text + " " + comboBox1.Select
    lblResult.Text = total.ToString();
}
```

通过比较客户端的代码发现：

简单工厂模式：将对象的选择创建交给了简单工厂类，客户端只需要输入相应的条件就可以，不用负责对象的创建，但是需要客户端自己调用算法类的方法。但是一旦需要增加新的运算类，比如开根运算，就要去修改简单工厂类。

策略模式：对象的选择创建仍需要自己来做，但是将调用方法的职责交给了Context类。一旦需要增加新的策略需要修改客户端。

因此，简单工厂模式的缺点就是当有新的需求增加时，需要频繁的修改工厂类。只用策略模式，当有新的需求增加时需要修改的是客户端，客户端仍然承担着创建对象的职责，并没有减轻客户端的压力。而将这两种模式结合起来使用，则需要修改 Context 类，总之不是完美的。

分类: C++设计模式

« 上一篇： C++设计模式——工厂模式Factory Method
» 下一篇： 基于Servlet实现简单系统登录

posted @ 2020-04-14 09:41 王陆 阅读(315) 评论(0) 编辑 收藏

导航目录