**Factory模式**

问题：

在面向对象系统设计中经常可以遇到以下的两类问题：

1）为了提高内聚（Cohesion）和松耦合（Coupling），我们经常会抽象出一些类的公共接口以形成抽象基类或者接口。这样我们可以通过声明一个指向基类的指针来指向实际的子类实现，达到了多态的目的。这里很容易出现的一个问题n多的子类继承自抽象基类，我们不得不在每次要用到子类的地方就编写诸如new xxx; 的代码。这里带来两个问题1）客户程序员必须知道实际子类的名称（当系统复杂后，命名将是一个很不好处理的问题，为了处理可能的名字冲突，有的命名可能并不是具有很好的可读性和可记忆性，就姑且不论不同程序员千奇百怪的个人偏好了。），2）程序的扩展性和维护变得越来越困难。

2）还有一种情况就是在父类中并不知道具体要实例化哪一个具体的子类。这里的意思为：假设我们在类A中要使用到类B，B是一个抽象父类，在A中并不知道具体要实例化那一个B的子类，但是在类A的子类D中是可以知道的。在A中我们没有办法直接使用类似于new xxx; 的语句，因为根本就不知道xxx是什么。

以上两个问题也就引出了Factory模式的两个最重要的功能：

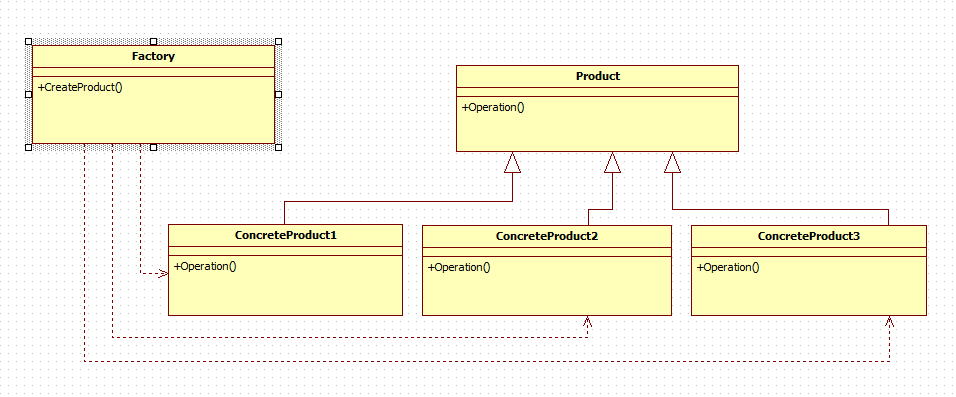
1）定义创建对象的接口，封装了对象的创建；

2）使得具体化类的工作延迟到了子类中。

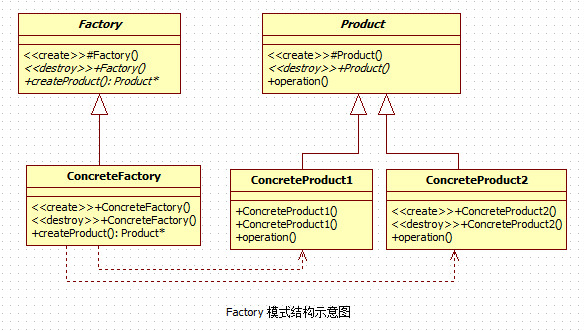
**模式选择**

我们通常使用Factory模式来解决上面给出的两个问题。在第一个问题中，我们经常就是声明一个创建对象的接口，并封装了对象的创建过程。Factory这里类似于一个真正意义上的工厂（生产对象）。在第二个问题中，我们需要提供一个对象创建对象的接口，并在子类中提供其具体实现（因为只有在子类中可以决定到底实例化哪一个类）。

第一中情况的Factory的结构示意图为：



以上Factory模式经常在系统开发中用到，但是这并不是Factory模式的最大威力所在（因为这可以通过其他方式解决这个问题）。Factory模式不单是提供了创建对象的接口，其最重要的是延迟了子类的实例化（第二个问题），以下是这种情况的一个Factory的结构示意图：



Factory模式的应用并不是只是为了封装对象的创建，而是要把对象的创建放到子类中实现：Factory中只是提供了对象创建的接口，其实现将放在Factory的子类ConcreteFactory中进行。这是图2和图1的区别所在。

**实现：**

完整代码示例（code）

// File Name : Product.h

#if !defined(\_PRODUCT\_H)

#define \_PRODUCT\_H

class Product

{

public:

virtual ~*Product*() = 0;

void operation();

protected:

Product();

};

#endif //\_PRODUCT\_H

File Name : Product.cpp

#include "Product.h"

#include <iostream>

using namespace std;

Product::Product()

{

cout << "Product..." << endl;

}

Product::~*Product*()

{

}

void Product::operation()

{

}

// File Name : ConcreteProduct.h

#if !defined(\_CONCRETEPRODUCT\_H)

#define \_CONCRETEPRODUCT\_H

#include "Product.h"

class ConcreteProduct : public Product

{

public:

ConcreteProduct();

~*ConcreteProduct*();

void operation();

};

#endif //\_CONCRETEPRODUCT\_H

// File Name : ConcreteProduct.cpp

#include "ConcreteProduct.h"

#include <iostream>

using namespace std;

ConcreteProduct::ConcreteProduct1()

{

cout << "ConcreteProduct..." << endl;

}

ConcreteProduct::~*ConcreteProduct*()

{

}

void ConcreteProduct::operation()

{

cout << "operation in ConcreteProduct" << endl;

}

// @ File Name : Factory.h

#if !defined(\_FACTORY\_H)

#define \_FACTORY\_H

class Product;

class Factory

{

public:

virtual ~*Factory*() = 0;

virtual Product\* *createProduct*() = 0;

protected:

Factory();

};

#endif //\_FACTORY\_H

// @ File Name : Factory.cpp

#include "Factory.h"

#include "Product.h"

Factory::~*Factory*()

{

}

Factory::Factory()

{

}

// @ File Name : ConcreteFactory.h

#if !defined(\_CONCRETEFACTORY\_H)

#define \_CONCRETEFACTORY\_H

#include "Factory.h"

class ConcreteFactory : public Factory

{

public:

ConcreteFactory();

~*ConcreteFactory*();

Product\* *createProduct*();

};

#endif //\_CONCRETEFACTORY\_H

// @ File Name : ConcreteFactory.cpp

#include "ConcreteFactory.h"

#include "ConcreteProduct1.h"

#include <iostream>

using namespace std;

ConcreteFactory::ConcreteFactory()

{

cout << "ConcreteFactory..." << endl;

}

ConcreteFactory::~*ConcreteFactory*()

{

}

Product\* ConcreteFactory::*createProduct*()

{

return new ConcreteProduct1();

}

// @ File Name : main.cpp

#include "ConcreteFactory.h"

#include "ConcreteProduct1.h"

#include <iostream>

using namespace std;

int main()

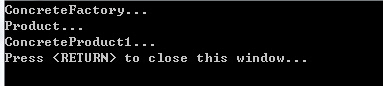
{

Factory\* factory = new ConcreteFactory();

Product\* product = factory->*createProduct*();

return 0;

}



**代码说明：**

示例代码中给出的是Factory模式解决父类中并不知道具体要实例化哪一个具体的子类的问题，至于为创建对象提供接口问题，可以由Factory 中附加相应的创建操作例如Create\*\*\*Product（）即可。

Factory模式在实际开发中应用非常广泛，面向对象的系统经常面临着对象创建问题：要创建的类实在是太多了。而Factory提供的创建对象的接口封装（第一个功能），以及其将类的实例化推迟到子类（第二个功能）都部分地解决了实际问题。

**Factory模式也带来至少以下两个问题：**

1）如果为每一个具体的ConcreteProduct类的实例化提供一个函数体，那么我们可能不得不在系统中添加了一个方法来处理这个新建的ConcreteProduct，这样Factory的接口永远就不肯能封闭（Close）。当然我们可以通过创建一个Factory的子类来通过多态实现这一点，但是这也是以新建一个类作为代价的。

2）在实现中我们可以通过参数化工厂方法，即给FactoryMethod（）传递一个参数用以决定是创建具体哪一个具体的Product（实际上笔者在VisualCMCS中也正是这样做的）。当然也可以通过模板化避免1）中的子类创建子类，其方法就是将具体Product 类作为模板参数，实现起来也很简单。

可以看出，Factory模式对于对象的创建给予开发人员提供了很好的实现策略，但是Factory模式仅仅局限于一类类（就是说Product是一类，有一个共同的基类），如果我们要为不同类的类提供一个对象创建的接口，那就要用AbstractFactory了。