# 1 简单工程模式

## 1.1 描述

简单工厂模式是工厂模式中最简单的一种，他可以用比较简单的方式隐藏创建对象的细节，一般只需要告诉工厂类所需要的类型，工厂类就会返回需要的产品类，但客户端看到的只是产品的抽象对象，无需关心到底是返回了哪个子类。客户端唯一需要知道的具体子类就是工厂子类。除了这点，基本是达到了依赖倒转原则的要求。

假如，我们不用工厂类，只用AbstractProduct和它的子类，那客户端每次使用不同的子类的时候都需要知道到底是用哪一个子类，当类比较少的时候还没什么问题，但是当类比较多的时候，管理起来就非常的麻烦了，就必须要做大量的替换，一个不小心就会发生错误。

而使用了工厂类之后，就不会有这样的问题，不管里面多少个类，我只需要知道类型号即可。不过，这里还有一个疑问，那就是如果我每次用工厂类创建的类型都不相同，这样修改起来的时候还是会出现问题，还是需要大量的替换。所以简单工厂模式一般应该于程序中大部分地方都只使用其中一种产品，工厂类也不用频繁创建产品类的情况。这样修改的时候只需要修改有限的几个地方即可。

客户只需要知道SimpleFactory就可以了，使用的时候也是使用的AbstractFactory，这样客户端只在第一次创建工厂的时候是知道具体的细节的，其他时候它都只知道AbstractFactory，这样就完美的达到了依赖倒转的原则。

## 1.2 常用场景

例如部署多种数据库的情况，可能在不同的地方要使用不同的数据库，此时只需要在配置文件中设定数据库的类型，每次再根据类型生成实例，这样，不管下面的数据库类型怎么变化，在客户端看来都是只有一个AbstractProduct，使用的时候根本无需修改代码。提供的类型也可以用比较便于识别的字符串，这样不用记很长的类名，还可以保存为配置文件。

这样，每次只需要修改配置文件和添加新的产品子类即可。

所以简单工厂模式一般应用于多种同类型类的情况，将这些类隐藏起来，再提供统一的接口，便于维护和修改。

## 1.3 优缺点

### 1.3.1 优点

1.隐藏了对象创建的细节，将产品的实例化推迟到子类中实现。

2.客户端基本不用关心使用的是哪个产品，只需要知道用哪个工厂就行了，提供的类型也可以用比较便于识别的字符串。

3.方便添加新的产品子类，每次只需要修改工厂类传递的类型值就行了。

4.遵循了依赖倒转原则。

### 1.3.2 缺点

1.要求产品子类的类型差不多，使用的方法名都相同，如果类比较多，而所有的类又必须要添加一种方法，则会是非常麻烦的事情。或者是一种类另一种类有几种方法不相同，客户端无法知道是哪一个产品子类，也就无法调用这几个不相同的方法。

2.每添加一个产品子类，都必须在工厂类中添加一个判断分支，这违背了开放-封闭原则。

## 1.4 C++代码实现

1. AbstractProduct.h

|  |
| --- |
| #ifndef \_ABSTRACTPRODUCT\_H\_  #define \_ABSTRACTPRODUCT\_H\_  #include <stdio.h>  /\*  \* 产品抽象类  \*/  class AbstractProduct  {  public:  AbstractProduct();  virtual ~AbstractProduct();  public:  virtual void operation() = 0;  };  /\*  \* 产品子类A  \*/  class ProductA : public AbstractProduct  {  public:  ProductA();  virtual ~ProductA();  public:  void operation();  };  /\*  \* 产品子类B  \*/  class ProductB : public AbstractProduct  {  public:  ProductB();  virtual ~ProductB();  public:  void operation();  };  #endif |

2. AbstractProduct.cpp

|  |
| --- |
| #include "AbstractProduct.h"  /\*  \* 产品抽象类-构造函数  \*/  AbstractProduct::AbstractProduct()  { }  /\*  \* 产品抽象类-析构函数  \*/  AbstractProduct::~AbstractProduct()  { }  /\*  \* 产品子类A-构造函数  \*/  ProductA::ProductA()  { }  /\*  \* 产品子类A-析构函数  \*/  ProductA::~ProductA()  { }  /\*  \* 产品子类A-方法1  \*/  void ProductA::operation()  {  fprintf(stderr,"ProductA operation!\n");  }  /\*  \* 产品子类B-构造函数  \*/  ProductB::ProductB()  { }  /\*  \* 产品子类B-析构函数  \*/  ProductB::~ProductB()  { }  /\*  \* 产品子类B-方法1  \*/  void ProductB::operation()  {  fprintf(stderr,"ProductB operation!\n");  } |

3. SimpleFactory.h

|  |
| --- |
| #ifndef \_SIMPLE\_FACTORY\_H\_  #define \_SIMPLE\_FACTROY\_H\_  #include <stdio.h>  #include "AbstractProduct.h"  class SimpleFactory  {  public:  SimpleFactory();  ~SimpleFactory();  public:  AbstractProduct\* createProduct(int type);  };  #endif |

4. SimpleFactory.cpp

|  |
| --- |
| #include "SimpleFactory.h"  SimpleFactory::SimpleFactory()  { }  SimpleFactory::~SimpleFactory()  { }  AbstractProduct\* SimpleFactory::createProduct(int type)  {  AbstractProduct\* temp = NULL;  switch(type)  {  case 1:  temp = new ProductA();  break;  case 2:  temp = new ProductB();  break;  default:  break;  }  return temp;  } |

5. main.cpp

|  |
| --- |
| #include "SimpleFactory.h"  int main(){  SimpleFactory\* factory = new SimpleFactory();  AbstractProduct\* product = factory->createProduct(1);  product->operation();  delete product;  product = NULL;    product = factory->createProduct(2);  product->operation();  delete product;  product = NULL;  return 0;  } |

# 2 工厂模式

## 2.1 描述

工厂模式基本与简单工厂模式差不多，上面也说了，每次添加一个产品子类都必须在工厂类中添加一个判断分支，这样违背了开放-封闭原则，因此，工厂模式就是为了解决这个问题而产生的。

既然每次都要判断，那我就把这些判断都生成一个工厂子类，这样，每次添加产品子类的时候，只需再添加一个工厂子类就可以了。这样就完美的遵循了开放-封闭原则。但这其实也有问题，如果产品数量足够多，要维护的量就会增加，好在一般工厂子类只用来生成产品类，只要产品子类的名称不发生变化，那么基本工厂子类就不需要修改，每次只需要修改产品子类就可以了。

同样工厂模式一般应该于程序中大部分地方都只使用其中一种产品，工厂类也不用频繁创建产品类的情况。这样修改的时候只需要修改有限的几个地方即可。

## 2.2 常用场景

基本与简单工厂模式一致，只不过是改进了简单工厂模式中的开放-封闭原则的缺陷，使得模式更具有弹性。将实例化的过程推迟到子类中，由子类来决定实例化哪个。

## 2.3 优缺点

### 2.3.1优点

基本与简单工厂模式一致，多的一点优点就是遵循了开放-封闭原则，使得模式的灵活性更强。

### 2.3.2 缺点

与简单工厂模式差不多。

## 2.4 C++代码实现

1. AbstractProduct.h同上

2. AbstractProduct.cpp同上

3. AbstractFactory.h

|  |
| --- |
| #ifndef \_ABSTRACT\_FACTROY\_H\_  #define \_ABSTRACT\_FACTROY\_H\_  #include <stdio.h>  #include "AbstractProduct.h"  class AbstractFactory{  public:  AbstractFactory();  virtual ~AbstractFactory();  public:  virtual AbstractProduct\* createProduct() = 0;  };  class FactoryA:public AbstractFactory{  public:  FactoryA();  ~FactoryA();  public:  AbstractProduct\* createProduct();  };  class FactoryB:public AbstractFactory{  public:  FactoryB();  ~FactoryB();  public:  AbstractProduct\* createProduct();  };  #endif |

4. AbstractFactory.cpp

|  |
| --- |
| #include "AbstractFactory.h"  AbstractFactory::AbstractFactory()  {}  AbstractFactory::~AbstractFactory()  {}  FactoryA::FactoryA()  {}  FactoryA::~FactoryA()  {}  AbstractProduct\* FactoryA::createProduct(){  AbstractProduct\* temp = NULL;  temp = new ProductA();  return temp;  }  FactoryB::FactoryB()  {}  FactoryB::~FactoryB()  {}  AbstractProduct\* FactoryB::createProduct(){  AbstractProduct\* temp = NULL;  temp = new ProductB();  return temp;  } |

5.main.cpp

|  |
| --- |
| #include "AbstractFactory.h"  int main(){  AbstractFactory\* factory = new FactoryA();  AbstractProduct\* product = factory->createProduct();  product->operation();  delete product;  product = NULL;  delete factory;  factory = NULL;    factory = new FactoryB();  product = factory->createProduct();  product->operation();  delete product;  product = NULL;  delete factory;  factory = NULL;  return 0;  } |

# 3 抽象工厂模式

抽象工厂模式就变得比工厂模式更为复杂，就像上面提到的缺点一样，工厂模式和简单工厂模式要求产品子类必须要是同一类型的，拥有共同的方法，这就限制了产品子类的扩展。于是为了更加方便的扩展，抽象工厂模式就将同一类的产品子类归为一类，让他们继承同一个抽象子类，我们可以把他们一起视作一组，然后好几组产品构成一族。

此时，客户端要使用时必须知道是哪一个工厂并且是哪一组的产品抽象类。每一个工厂子类负责产生一族产品，而子类的一种方法产生一种类型的产品。在客户端看来只有AbstractProductA和AbstractProductB两种产品，使用的时候也是直接使用这两种产品。而通过工厂来识别是属于哪一族产品。

产品ProductA\_1和ProductB\_1构成一族产品，对应于有Factory1来创建，也就是说Factory1总是创建的ProductA\_1和ProductB\_1的产品，在客户端看来只需要知道是哪一类工厂和产品组就可以了。一般来说， ProductA\_1和ProductB\_1都是适应同一种环境的，所以他们会被归为一族。

## 3.1 常用场景

例如Linux和windows两种操作系统下，有2个挂件A和B，他们在Linux和Windows下面的实现方式不同，Factory1负责产生能在Linux下运行的挂件A和B，Factory2负责产生能在Windows下运行的挂件A和B，这样如果系统环境发生变化了，我们只需要修改工厂就行了。

## 3.2 优缺点

### 3.2.1优点

1.封装了产品的创建，使得不需要知道具体是哪种产品，只需要知道是哪个工厂就行了。

2.可以支持不同类型的产品，使得模式灵活性更强。

3.可以非常方便的使用一族中间的不同类型的产品。

### 3.2.2缺点

1.结构太过臃肿，如果产品类型比较多，或者产品族类比较多，就会非常难于管理。

2.每次如果添加一组产品，那么所有的工厂类都必须添加一个方法，这样违背了开放-封闭原则。所以一般适用于产品组合产品族变化不大的情况。

## 3.3 C++实现代码

1. AbstractProductA.h

|  |
| --- |
| #ifndef \_ABSTRACTPRODUCTA\_H\_  #define \_ABSTRACTPRODUCTA\_H\_  #include <stdio.h>  class AbstractProductA{  public:  AbstractProductA();  virtual ~AbstractProductA();  public:  virtual void operationA() = 0;  };  class ProductA\_1:public AbstractProductA{  public:  ProductA\_1();  virtual ~ProductA\_1();  public:  void operationA();  };  class ProductA\_2:public AbstractProductA{  public:  ProductA\_2();  ~ProductA\_2();  public:  void operationA();  };  #endif |

2. AbstractProductA.cpp

|  |
| --- |
| #include "AbstractProductA.h"  AbstractProductA::AbstractProductA(){  }  AbstractProductA::~AbstractProductA(){  }  ProductA\_1::ProductA\_1(){  }  ProductA\_1::~ProductA\_1(){  }  void ProductA\_1::operationA(){  fprintf(stderr,"productA\_1 operation!\n");  }  ProductA\_2::ProductA\_2(){  }  ProductA\_2::~ProductA\_2(){  }  void ProductA\_2::operationA(){  fprintf(stderr,"productA\_2 operation!\n");  } |

3. AbstractProductB.h

|  |
| --- |
| #ifndef \_ABSTRACTPRODUCTB\_H\_  #define \_ABSTRACTPRODUCTB\_H\_  #include <stdio.h>  class AbstractProductB{  public:  AbstractProductB();  virtual ~AbstractProductB();  public:  virtual void operationB() = 0;  };  class ProductB\_1:public AbstractProductB{  public:  ProductB\_1();  virtual ~ProductB\_1();  public:  void operationB();  };  class ProductB\_2:public AbstractProductB{  public:  ProductB\_2();  ~ProductB\_2();  public:  void operationB();  };  #endif |

4. AbstractProductB.cpp

|  |
| --- |
| #include "AbstractProductB.h"  AbstractProductB::AbstractProductB(){  }  AbstractProductB::~AbstractProductB(){  }  ProductB\_1::ProductB\_1(){  }  ProductB\_1::~ProductB\_1(){  }  void ProductB\_1::operationB(){  fprintf(stderr,"productB\_1 operation!\n");  }  ProductB\_2::ProductB\_2(){  }  ProductB\_2::~ProductB\_2(){  }  void ProductB\_2::operationB(){  fprintf(stderr,"productB\_2 operation!\n");  } |

5. AbstractFactory.h

|  |
| --- |
| #ifndef \_SIMPLEFACTORY\_H\_  #define \_SIMPLEFACTROY\_H\_  #include <stdio.h>  #include "AbstractProductA.h"  #include "AbstractProductB.h"  class AbstractFactory{  public:  AbstractFactory();  virtual ~AbstractFactory();  public:  virtual AbstractProductA\* createProductA() = 0;  virtual AbstractProductB\* createProductB() = 0;  };  class Factory1:public AbstractFactory{  public:  Factory1();  ~Factory1();  public:  AbstractProductA\* createProductA();  AbstractProductB\* createProductB();  };  class Factory2:public AbstractFactory{  public:  Factory2();  ~Factory2();  public:  AbstractProductA\* createProductA();  AbstractProductB\* createProductB();  };  #endif |

6. AbstractFactory.cpp

|  |
| --- |
| #include "AbstractFactory.h"  AbstractFactory::AbstractFactory(){  }  AbstractFactory::~AbstractFactory(){  }  Factory1::Factory1(){  }  Factory1::~Factory1(){  }  AbstractProductA\* Factory1::createProductA(){  AbstractProductA\* temp = NULL;  temp = new ProductA\_1();  return temp;  }  AbstractProductB\* Factory1::createProductB(){  AbstractProductB\* temp = NULL;  temp = new ProductB\_1();  return temp;  }  Factory2::Factory2(){  }  Factory2::~Factory2(){  }  AbstractProductA\* Factory2::createProductA(){  AbstractProductA\* temp = NULL;  temp = new ProductA\_2();  return temp;  }  AbstractProductB\* Factory2::createProductB(){  AbstractProductB\* temp = NULL;  temp = new ProductB\_2();  return temp;  } |

7. main.cpp

|  |
| --- |
| #include "AbstractFactory.h"  int main(){  AbstractFactory\* factory = new Factory1();  AbstractProductA\* productA = factory->createProductA();  AbstractProductB\* productB = factory->createProductB();  productA->operationA();  productB->operationB();    delete factory;  factory = NULL;  delete productA;  productA = NULL;  delete productB;  productB = NULL;    factory = new Factory2();  productA = factory->createProductA();  productB = factory->createProductB();  productA->operationA();  productB->operationB();    delete factory;  factory = NULL;  delete productA;  productA = NULL;  delete productB;  productB = NULL;  return 0;  } |