

软件领域中的设计模式为开发人员提供了一种使用专家设计经验的有效途径。设计模式中运用了面向对象编程语言的重要特性：封装、继承、多态，真正领悟设计模式的精髓是可能一个漫长的过程，需要大量实践经验的积累。最近看设计模式的书，对于每个模式，用 C++ 写了个小例子，加深一下理解。主要参考《大话设计模式》和《设计模式:可复用面向对象软件的基础》两本书。本文介绍工厂模式的实现。

工厂模式属于创建型模式，大致可以分为三类，简单工厂模式、工厂方法模式、抽象工厂模式。听上去差不多，都是工厂模式。下面一个个介绍，首先介绍简单工厂模式，它的主要特点是需要工厂类中做判断，从而创造相应的产品。当增加新的产品时，就需要修改工厂类。有点抽象，举个例子就明白了。有一家生产处理器核的厂家，它只有一个工厂，能够生产两种型号的处理器核。客户需要什么样的处理器核，一定要显示地告诉生产工厂。下面给出一种实现方案。

[cpp] view plain copy print?

```
1.  enum CTYPE {COREA, COREB};
2.  class SingleCore
3.  {
4.  public:
5.      virtual void Show() = 0;
6.  };
7.  //单核 A
8.  class SingleCoreA: public SingleCore
9.  {
10. public:
11.     void Show() { cout<<"SingleCore A"<<endl; }
12. };
13. //单核 B
14. class SingleCoreB: public SingleCore
15. {
16. public:
17.     void Show() { cout<<"SingleCore B"<<endl; }
18. };
19. //唯一的工厂，可以生产两种型号的处理器核，在内部判断
20. class Factory
21. {
22. public:
23.     SingleCore* CreateSingleCore(enum CTYPE ctype)
24.     {
25.         if(ctype == COREA) //工厂内部判断
26.             return new SingleCoreA(); //生产核 A
27.         else if(ctype == COREB)
28.             return new SingleCoreB(); //生产核 B
29.         else
```

```
30.         return NULL;
31.     }
32. };
```

这样设计的主要缺点之前也提到过，就是要增加新的核类型时，就需要修改工厂类。这就违反了开放封闭原则：软件实体（类、模块、函数）可以扩展，但是不可修改。于是，工厂方法模式出现了。所谓工厂方法模式，是指定义一个用于创建对象的接口，让子类决定实例化哪一个类。**Factory Method** 使一个类的实例化延迟到其子类。

听起来很抽象，还是以刚才的例子解释。这家生产处理器核的产家赚了不少钱，于是决定再开设一个工厂专门用来生产 B 型号的单核，而原来的工厂专门用来生产 A 型号的单核。这时，客户要做的是找好工厂，比如要 A 型号的核，就找 A 工厂要；否则找 B 工厂要，不再需要告诉工厂具体要什么型号的处理核了。下面给出一个实现方案。

[\[cpp\]](#) [view plain](#) [copy](#) [print?](#)

```
1.  class SingleCore
2.  {
3.  public:
4.      virtual void Show() = 0;
5.  };
6.  //单核 A
7.  class SingleCoreA: public SingleCore
8.  {
9.  public:
10.     void Show() { cout<<"SingleCore A"<<endl; }
11. };
12. //单核 B
13. class SingleCoreB: public SingleCore
14. {
15. public:
16.     void Show() { cout<<"SingleCore B"<<endl; }
17. };
18. class Factory
19. {
20. public:
21.     virtual SingleCore* CreateSingleCore() = 0;
22. };
23. //生产 A 核的工厂
24. class FactoryA: public Factory
25. {
26. public:
27.     SingleCoreA* CreateSingleCore() { return new SingleCoreA; }
28. };
29. //生产 B 核的工厂
```

```

30. class FactoryB: public Factory
31. {
32. public:
33.     SingleCoreB* CreateSingleCore() { return new SingleCoreB; }
34. };

```

工厂方法模式也有缺点，每增加一种产品，就需要增加一个对象的工厂。如果这家公司发展迅速，推出了很多新的处理器核，那么就要开设相应的新工厂。在 C++ 实现中，就是要定义一个个的工厂类。显然，相比简单工厂模式，工厂方法模式需要更多的类定义。

既然有了简单工厂模式和工厂方法模式，为什么还要有抽象工厂模式呢？它到底有什么作用呢？还是举这个例子，这家公司的技术不断进步，不仅可以生产单核处理器，也能生产多核处理器。现在简单工厂模式和工厂方法模式都鞭长莫及。抽象工厂模式登场了。它的定义为提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。具体这样应用，这家公司还是开设两个工厂，一个专门用来生产 A 型号的单核多核处理器，而另一个工厂专门用来生产 B 型号的单核多核处理器，下面给出实现的代码。

[\[cpp\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```

1. //单核
2. class SingleCore
3. {
4. public:
5.     virtual void Show() = 0;
6. };
7. class SingleCoreA: public SingleCore
8. {
9. public:
10.     void Show() { cout<<"Single Core A"<<endl; }
11. };
12. class SingleCoreB :public SingleCore
13. {
14. public:
15.     void Show() { cout<<"Single Core B"<<endl; }
16. };
17. //多核
18. class MultiCore
19. {
20. public:
21.     virtual void Show() = 0;
22. };
23. class MultiCoreA : public MultiCore
24. {
25. public:

```

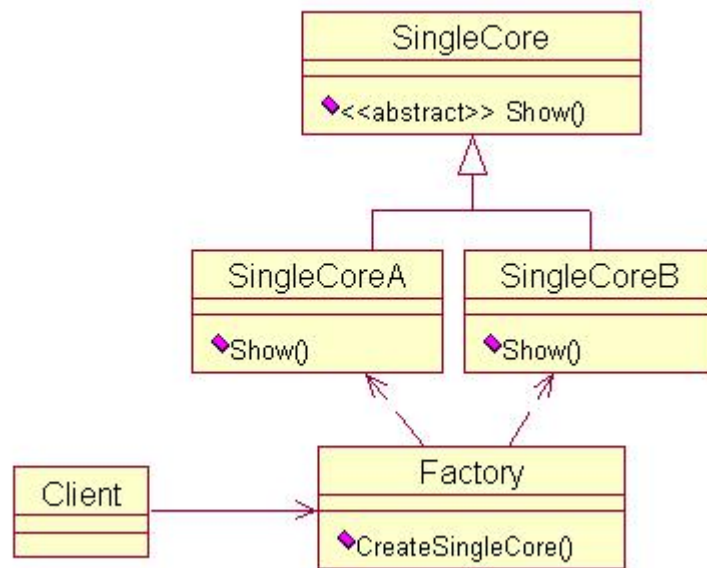
```

26.     void Show() { cout<<"Multi Core A"<<endl; }
27.
28. };
29. class MultiCoreB : public MultiCore
30. {
31. public:
32.     void Show() { cout<<"Multi Core B"<<endl; }
33. };
34. //工厂
35. class CoreFactory
36. {
37. public:
38.     virtual SingleCore* CreateSingleCore() = 0;
39.     virtual MultiCore* CreateMultiCore() = 0;
40. };
41. //工厂 A, 专门用来生产 A 型号的处理
42. class FactoryA :public CoreFactory
43. {
44. public:
45.     SingleCore* CreateSingleCore() { return new SingleCoreA(); }
46.     MultiCore* CreateMultiCore() { return new MultiCoreA(); }
47. };
48. //工厂 B, 专门用来生产 B 型号的处理
49. class FactoryB : public CoreFactory
50. {
51. public:
52.     SingleCore* CreateSingleCore() { return new SingleCoreB(); }
53.     MultiCore* CreateMultiCore() { return new MultiCoreB(); }
54. };

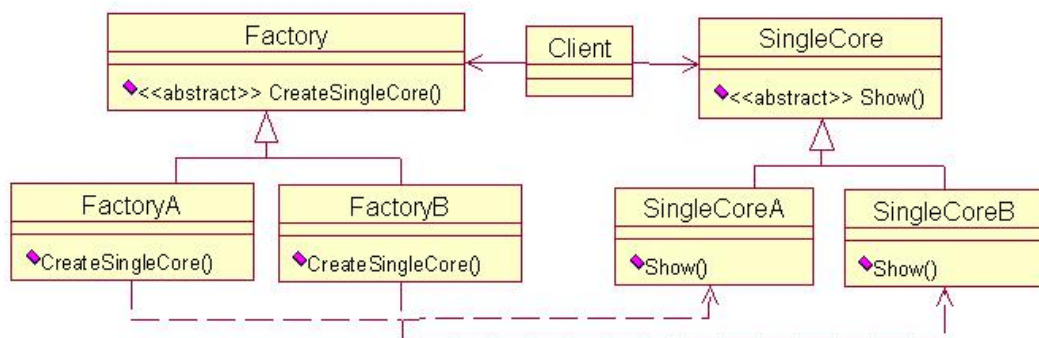
```

至此，工厂模式介绍完了。利用 Rational Rose 2003 软件，给出三种工厂模式的 UML 图，加深印象。

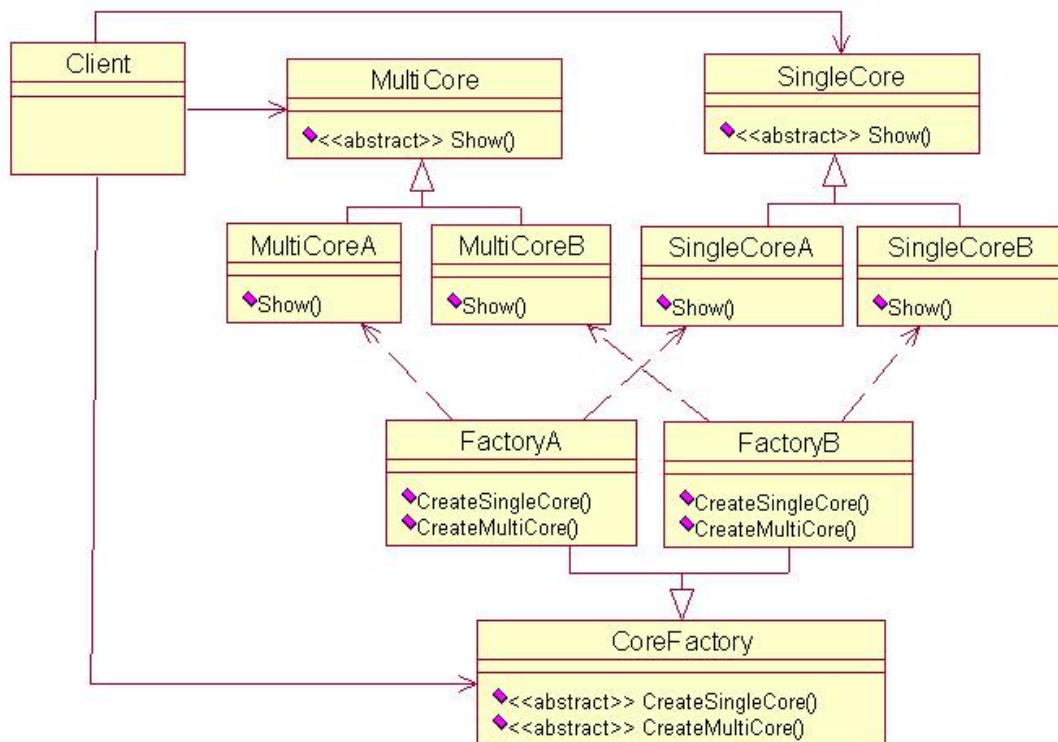
简单工厂模式的 UML 图：



工厂方法的 UML 图:



抽象工厂模式的 UML 图:



本人享有博客文章的版权，转载请标明出处 <http://blog.csdn.net/wuzhekai1985>

三种工厂模式的分析以及 C++实现

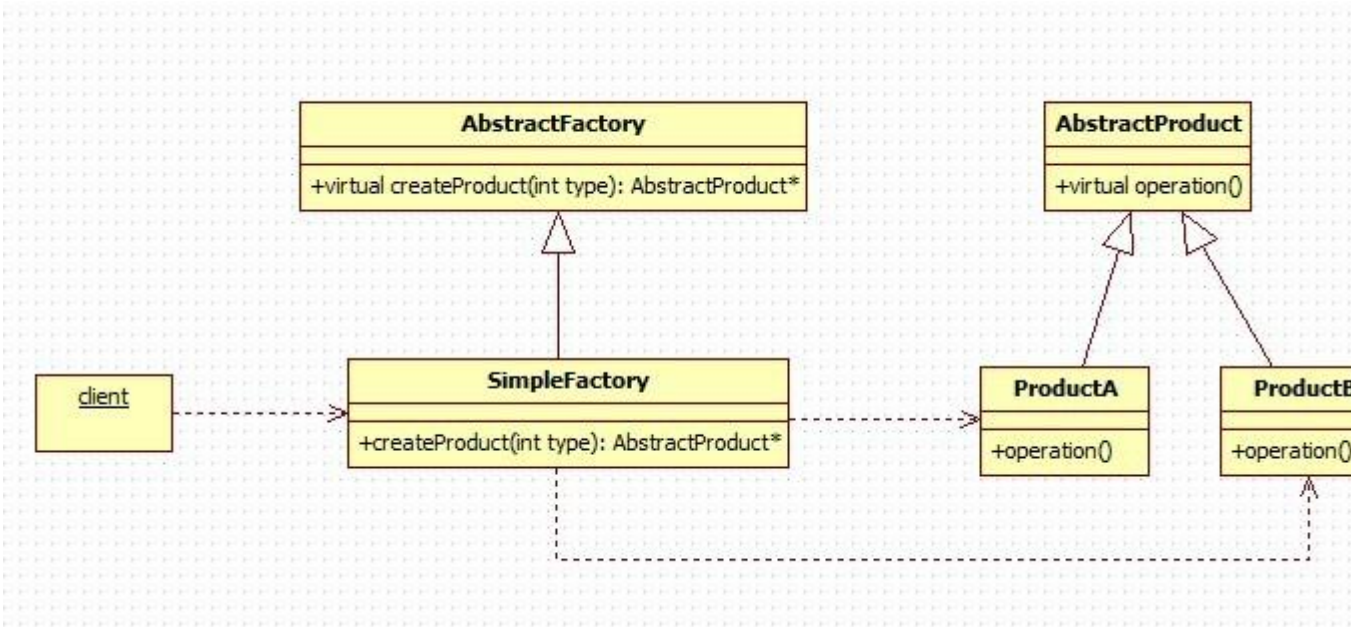
以下是我自己学习设计模式的思考总结。

简单工厂模式

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，但是当类比较多时，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。

常用的场景

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。

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。

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。

C++实现代码

```
1 #ifndef _ABSTRACTPRODUCT_H_ 2 #define _ABSTRACTPRODUCT_H_ 3 4 5 #include
<stdio.h> 6 7 8 class AbstractProduct{ 9 10 public:11     AbstractProduct();12
virtual ~AbstractProduct();13     14 public:15     virtual void operation() =
0;16 };17 18 class ProductA:public AbstractProduct{19 20 public:21
ProductA();22     virtual ~ProductA();23     24 public:25     void
operation();26 };27 28 class ProductB:public AbstractProduct{29 30 public:31
ProductB();32     ~ProductB();33     34 public:35     void operation();36 };37 38
#endif
```




```
1 #include "AbstractProduct.h" 2 3 4 5
AbstractProduct::AbstractProduct() { 6 } 7 8 9
AbstractProduct::~~AbstractProduct() {10 }11 12 13 ProductA::ProductA() {14 }15 16
17 ProductA::~~ProductA() {18 }19 20 21 void ProductA::operation() {22
fprintf(stderr, "productA operation!\n");23 }24 25 26 ProductB::ProductB() {27 }28
29 30 ProductB::~~ProductB() {31 }32 33 34 void ProductB::operation() {35
fprintf(stderr, "productB operation!\n");36 }
```



```
1 #ifndef _SIMPLEFACTORY_H_ 2 #define _SIMPLEFACTROY_H_ 3 4 #include <stdio.h>
5 #include "AbstractProduct.h" 6 7 8 class AbstractFactory{ 9 10 public:11
AbstractFactory();12 virtual ~AbstractFactory();13 14 public:15
virtual AbstractProduct* createProduct(int type) = 0; 16 };17 18 19 class
SimpleFactory:public AbstractFactory{20 21 public:22 SimpleFactory();23
~SimpleFactory();24 25 public:26 AbstractProduct* createProduct(int
type);27 };28 29 #endif
```



```

1 #include "SimpleFactory.h" 2 3 4 AbstractFactory::AbstractFactory(){ 5 } 6
7 8 AbstractFactory::~~AbstractFactory(){ 9 }10 11 12
SimpleFactory::SimpleFactory(){13 }14 15 16
SimpleFactory::~~SimpleFactory(){17 }18 19 20 AbstractProduct*
SimpleFactory::createProduct(int type){21     AbstractProduct* temp = NULL;22
switch(type)23     {24     case 1:25         temp = new ProductA();26
break;27     case 2:28         temp = new ProductB();29         break;30
default:31         break;32     }33     return temp;34 }

```



```

1 #include "SimpleFactory.h" 2 3 4 int main(){ 5     AbstractFactory* factory
= new SimpleFactory(); 6     AbstractProduct* product = factory->createProduct(1);
7     product->operation(); 8     delete product; 9     product = NULL;10     11
product = factory->createProduct(2);12     product->operation();13     delete
product;14     product = NULL;15     return 0;16 }

```



```

1 g++ -o client client.cpp SimpleFactory.cpp AbstractProduct.cpp

```

结果

```

[ubuntu@root test]# ./client
productA operation!
productB operation!

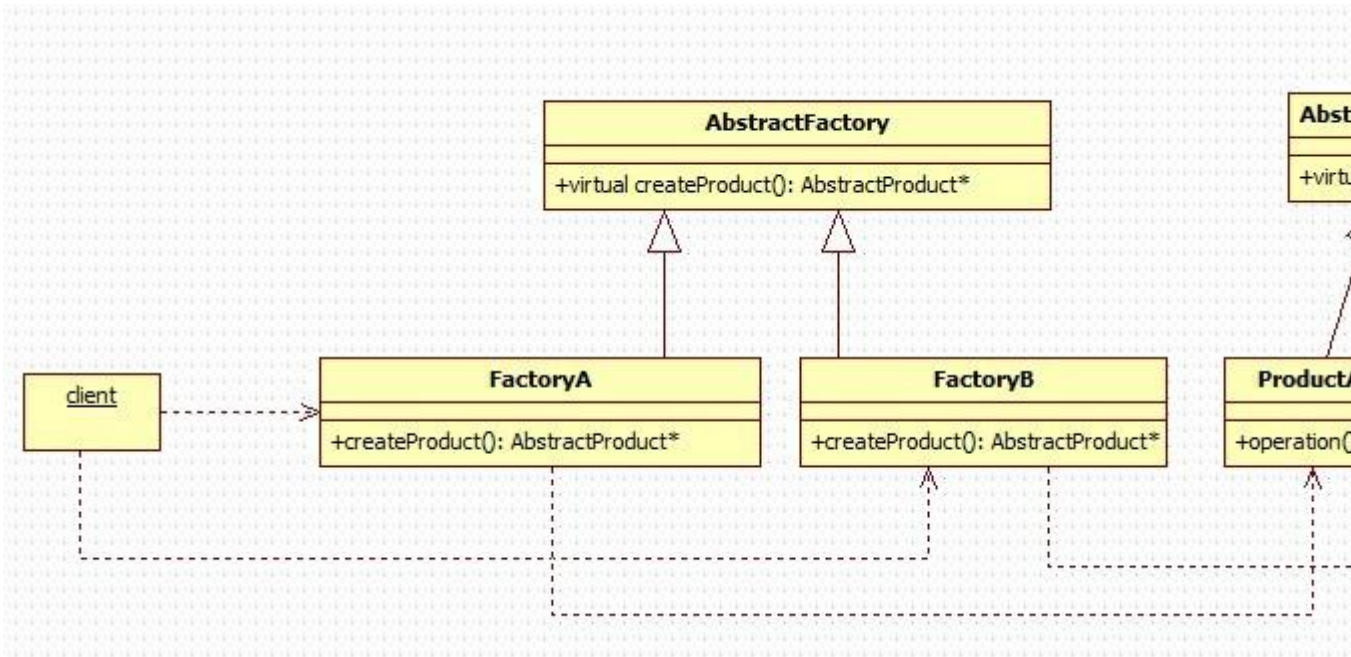
```

工厂模式

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。



常用的场景

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。

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。

缺点

与简单工厂模式差不多。

C++实现代码





```
1 #ifndef _ABSTRACTPRODUCT_H_ 2 #define _ABSTRACTPRODUCT_H_ 3 4 5 #include
<stdio.h> 6 7 8 class AbstractProduct{ 9 10 public:11     AbstractProduct();12
virtual ~AbstractProduct();13     14 public:15     virtual void operation() =
0;16 };17 18 class ProductA:public AbstractProduct{19 20 public:21
ProductA();22     virtual ~ProductA();23     24 public:25     void
operation();26 };27 28 class ProductB:public AbstractProduct{29 30 public:31
ProductB();32     ~ProductB();33     34 public:35     void operation();36 };37 38
#endif
```








```
1 #include "AbstractProduct.h" 2 3 4 5
AbstractProduct::AbstractProduct(){ 6 } 7 8 9
AbstractProduct::~~AbstractProduct(){10 }11 12 13 ProductA::ProductA(){14 }15 16
17 ProductA::~~ProductA(){18 }19 20 21 void ProductA::operation(){22
fprintf(stderr,"productA operation!\n");23 }24 25 26 ProductB::ProductB(){27 }28
29 30 ProductB::~~ProductB(){31 }32 33 34 void ProductB::operation(){35
fprintf(stderr,"productB operation!\n");36 }
```







```
1 #ifndef _SIMPLEFACTORY_H_ 2 #define _SIMPLEFACTROY_H_ 3 4 #include <stdio.h>
5 #include "AbstractProduct.h" 6 7 8 class AbstractFactory{ 9 10 public:11
AbstractFactory();12     virtual ~AbstractFactory();13     14 public:15
virtual AbstractProduct* createProduct() = 0; 16 };17 18 19 class
FactoryA:public AbstractFactory{20 21 public:22     FactoryA();23
~FactoryA();24     25 public:26     AbstractProduct* createProduct();27 };28 29
30 class FactoryB:public AbstractFactory{31 32 public:33     FactoryB();34
~FactoryB();35     36 public:37     AbstractProduct* createProduct();38 };39
#endif
```



```
1 #include "AbstractFactory.h" 2 3 4 AbstractFactory::~AbstractFactory(){ 5 }
6 7 8 AbstractFactory::~~AbstractFactory(){ 9 }10 11 12
FactoryA::FactoryA(){13 }14 15 16 FactoryA::~~FactoryA(){17 }18 19 20
AbstractProduct* FactoryA::createProduct(){21     AbstractProduct* temp =
NULL;22     temp = new ProductA();23     return temp;24 }25 26 27
FactoryB::FactoryB(){28 }29 30 31 FactoryB::~~FactoryB(){32 }33 34 35
AbstractProduct* FactoryB::createProduct(){36     AbstractProduct* temp =
NULL;37     temp = new ProductB();38     return temp;39 }
```



```
1 #include "AbstractFactory.h" 2 3 4 int main(){ 5     AbstractFactory* factory
= new FactoryA(); 6     AbstractProduct* product = factory->createProduct(); 7
product->operation(); 8     delete product; 9     product = NULL;10     delete
factory;11     factory = NULL;12     13     factory = new FactoryB();14     product
```

```
= factory->createProduct();15    product->operation();16    delete product;17  
product = NULL;18    delete factory;19    factory = NULL;20    return 0;21 }
```



```
1 g++ -o client client.cpp AbstractFactory.cpp AbstractProduct.cpp
```

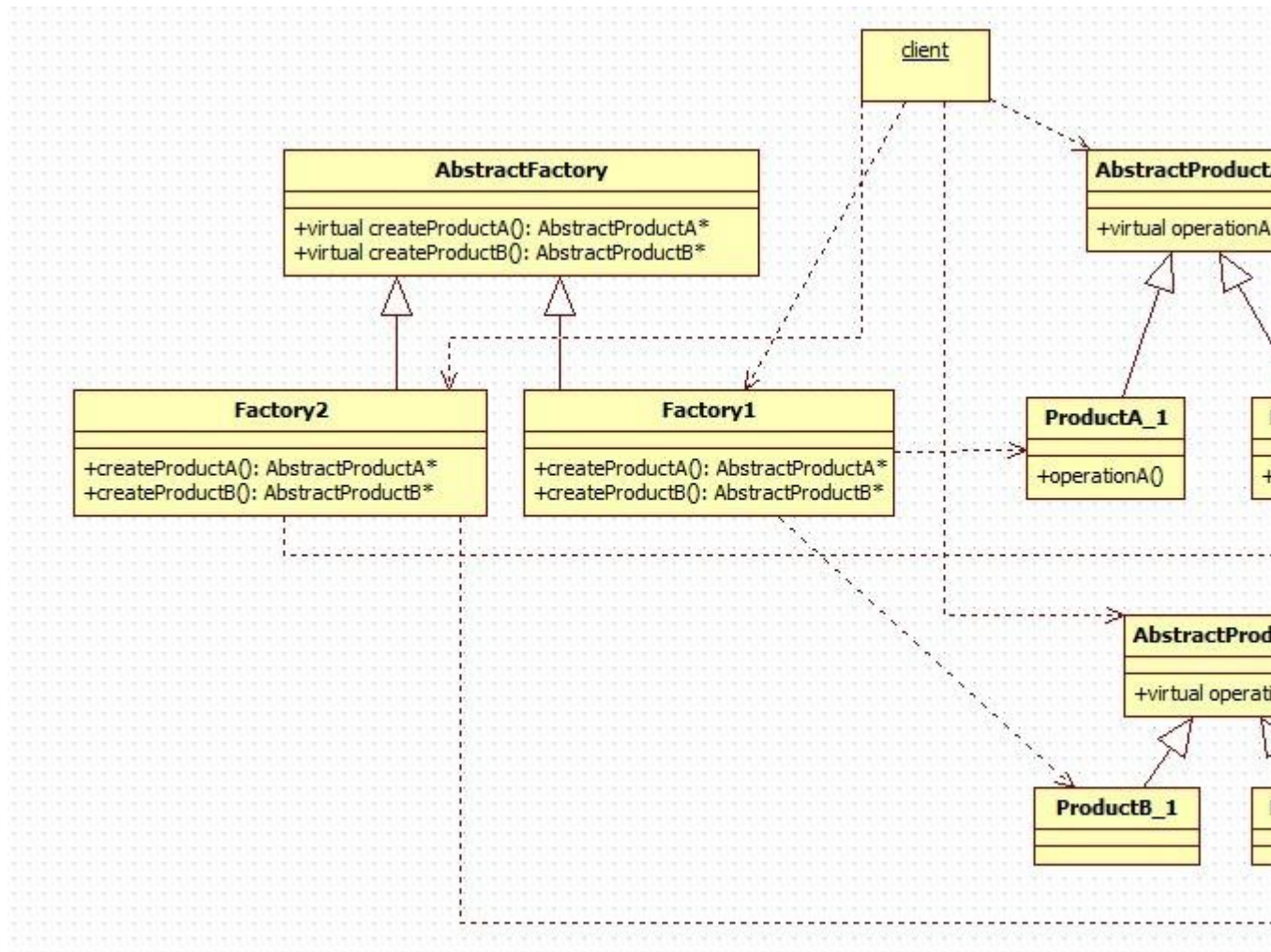
结果

```
[ubuntu@root test]# ./client  
productA operation!  
productB operation!
```

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。

常用的场景

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了变化了，我们只需要修改工厂就行了。

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。

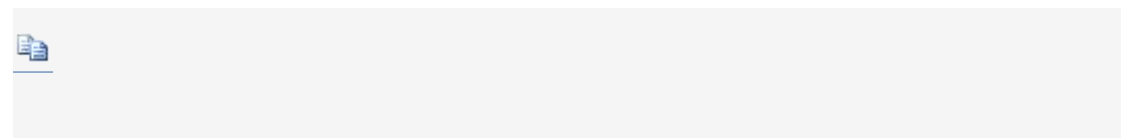
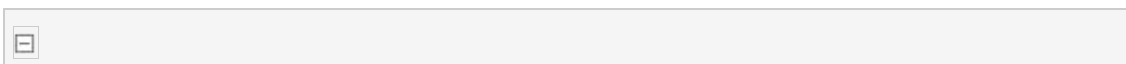
3.可以非常方便的使用一族中间的不同类型的产品。

缺点

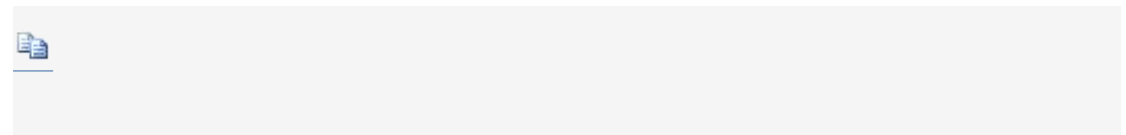
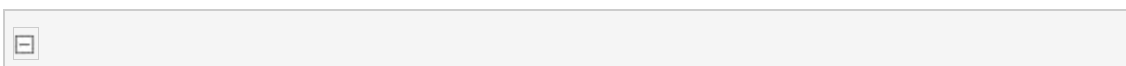
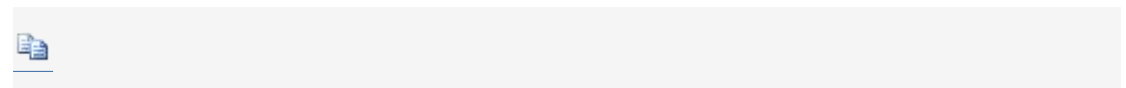
1.结构太过臃肿，如果产品类型比较多，或者产品族类比较多，就会非常难于管理。

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。

C++实现代码



```
1 #ifndef _ABSTRACTPRODUCTA_H_ 2 #define _ABSTRACTPRODUCTA_H_ 3 4 5 #include
<stdio.h> 6 7 8 class AbstractProductA{ 9 10 public:11
AbstractProductA();12     virtual ~AbstractProductA();13     14 public:15
virtual void operationA() = 0;16 };17 18 class ProductA_1:public
AbstractProductA{19 20 public:21     ProductA_1();22     virtual
~ProductA_1();23     24 public:25     void operationA();26 };27 28 class
ProductA_2:public AbstractProductA{29 30 public:31     ProductA_2();32
~ProductA_2();33     34 public:35     void operationA();36 };37 38 #endif
```



```
1 #include "AbstractProductA.h" 2 3 4 5
AbstractProductA::AbstractProductA(){ 6 } 7 8 9
AbstractProductA::~~AbstractProductA(){10 }11 12 13
ProductA_1::ProductA_1(){14 }15 16 17 ProductA_1::~~ProductA_1(){18 }19 20 21 void
```



```

ProductA_1::operationA(){22     fprintf(stderr,"productA_1
operation!\n");23 }24 25 26 ProductA_2::ProductA_2(){27 }28 29 30
ProductA_2::~~ProductA_2(){31 }32 33 34 void ProductA_2::operationA(){35
fprintf(stderr,"productA_2 operation!\n");36 }

```



```

1 #ifndef _ABSTRACTPRODUCTB_H_ 2 #define _ABSTRACTPRODUCTB_H_ 3 4 5 #include
<stdio.h> 6 7 8 class AbstractProductB{ 9 10 public:11
AbstractProductB();12     virtual ~AbstractProductB();13     14 public:15
virtual void operationB() = 0;16 };17 18 class ProductB_1:public
AbstractProductB{19 20 public:21     ProductB_1();22     virtual
~ProductB_1();23     24 public:25     void operationB();26 };27 28 class
ProductB_2:public AbstractProductB{29 30 public:31     ProductB_2();32
~ProductB_2();33     34 public:35     void operationB();36 };37 38 #endif

```



```

1 #include "AbstractProductB.h" 2 3 4 5
AbstractProductB::AbstractProductB(){ 6 } 7 8 9
AbstractProductB::~~AbstractProductB(){10 }11 12 13
ProductB_1::ProductB_1(){14 }15 16 17 ProductB_1::~~ProductB_1(){18 }19 20 21 void
ProductB_1::operationB(){22     fprintf(stderr,"productB_1
operation!\n");23 }24 25 26 ProductB_2::ProductB_2(){27 }28 29 30
ProductB_2::~~ProductB_2(){31 }32 33 34 void ProductB_2::operationB(){35
fprintf(stderr,"productB_2 operation!\n");36 }

```





```
1 #ifndef _SIMPLEFACTORY_H_ 2 #define _SIMPLEFACTROY_H_ 3 4 #include <stdio.h>
5 #include "AbstractProductA.h" 6 #include "AbstractProductB.h" 7 8 9 class
AbstractFactory{10 11 public:12     AbstractFactory();13     virtual
~AbstractFactory();14     15 public:16     virtual AbstractProductA*
createProductA() = 0; 17     virtual AbstractProductB* createProductB() = 0;
18 };19 20 21 class Factory1:public AbstractFactory{22 23 public:24
Factory1();25     ~Factory1();26     27 public:28     AbstractProductA*
createProductA();29     AbstractProductB* createProductB();30 };31 32 33 class
Factory2:public AbstractFactory{34 35 public:36     Factory2();37
~Factory2();38     39 public:40     AbstractProductA* createProductA();41
AbstractProductB* createProductB();42 };43 #endif
```



```
1 #include "AbstractFactory.h" 2 3 4 AbstractFactory::AbstractFactory(){ 5 }
6 7 8 AbstractFactory::~~AbstractFactory(){ 9 }10 11 12
Factory1::Factory1(){13 }14 15 16 Factory1::~~Factory1(){17 }18 19 20
AbstractProductA* Factory1::createProductA(){21     AbstractProductA* temp =
NULL;22     temp = new ProductA_1();23     return temp;24 }25 26 27
AbstractProductB* Factory1::createProductB(){28     AbstractProductB* temp =
NULL;29     temp = new ProductB_1();30     return temp;31 }32 33 34
Factory2::Factory2(){35 }36 37 38 Factory2::~~Factory2(){39 }40 41 42
AbstractProductA* Factory2::createProductA(){43     AbstractProductA* temp =
NULL;44     temp = new ProductA_2();45     return temp;46 }47 48 49
AbstractProductB* Factory2::createProductB(){50     AbstractProductB* temp =
NULL;51     temp = new ProductB_2();52     return temp;53 }
```



```

1 #include "AbstractFactory.h" 2 3 4 int main(){ 5 6     AbstractFactory*
factory = new Factory1(); 7     AbstractProductA* productA =
factory->createProductA(); 8     AbstractProductB* productB =
factory->createProductB(); 9     productA->operationA();10
productB->operationB();11     12     delete factory;13     factory = NULL;14
delete productA;15     productA = NULL;16     delete productB;17     productB =
NULL;18     19     factory = new Factory2();20     productA =
factory->createProductA();21     productB = factory->createProductB();22
productA->operationA();23     productB->operationB();24     25     delete
factory;26     factory = NULL;27     delete productA;28     productA = NULL;29
delete productB;30     productB = NULL;31     return 0;32 }

```



```

1 g++ -o client AbstractProductA.cpp AbstractProductB.cpp AbstractFactory.cpp
client.cpp

```

结果

```

[ubuntu@root test]# ./client
productA_1 operation!
productB_1 operation!
productA_2 operation!
productB_2 operation!

```