# 简单工厂模式

## 背景

考虑一个简单的软件应用场景，一个软件系统可以提供多个外观不同的按钮（如圆形按钮、矩形按钮、菱形按钮等）， 这些按钮都源自同一个基类，不过在继承基类后不同的子类修改了部分属性从而使得它们可以呈现不同的外观，如果我们希望在使用这些按钮时，不需要知道这些具体按钮类的名字，只需要知道表示该按钮类的一个参数，并提供一个调用方便的方法，把该参数传入方法即可返回一个相应的按钮对象，此时，就可以使用简单工厂模式。

## 定义

简单工厂模式(Simple Factory Pattern)：又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，**可以根据参数的不同返回不同类的实例**。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

简单工厂模式，就是说到底要实例化谁，将来会不会增加实例化的对象（即有一些不可预知的变化的地方），应该考虑用一个单独的类来做这个创造实例的过程，这就是工厂。

## 模式结构

简单工厂模式包含如下角色：

1、Factory：工厂角色

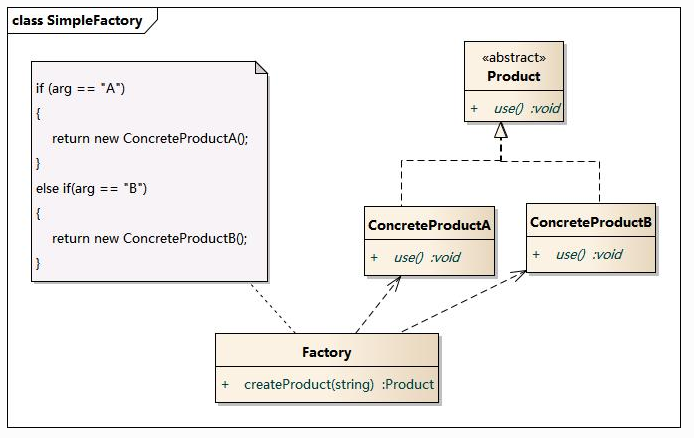
工厂角色负责实现创建所有实例的内部逻辑

2、Product：抽象产品角色

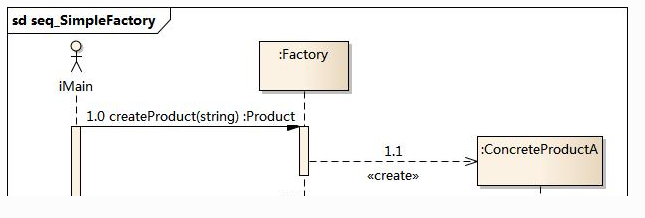
抽象产品角色是所创建的所有对象的父类，负责描述所有实例所共有的公共接口

3、ConcreteProduct：具体产品角色

具体产品角色是创建目标，所有创建的对象都充当这个角色的某个具体类的实例。



## 时序图



## 代码实现

具体代码：

public class OperationFactory

{

public static Operation createOperation(string operation)

{

Operation oper = null;

switch(operation){

case “+”：

oper = new OperationAdd();

break;

case "=":

oper = new OperationSub();

break;

case "\*":

oper = new OperationMul();

break;

case "/":

oper = new OperationDiv();

break;

}

return oper;

}

}

//客户端代码：

Operation oper;

oper = OperationFactory.createOperation("+");

oper.number1 = 1;

oper.number2 = 2;

double result = oper.GetResult()

另外一种代码：

#include "Factory.h"

#include "ConcreteProductA.h"

#include "ConcreteProductB.h"

Product\* Factory::createProduct(string proname){

if ( "A" == proname )

{

return new ConcreteProductA();

}

else if("B" == proname)

{

return new ConcreteProductB();

}

return NULL;

}

## 分析

将对象的创建和对象本身业务处理分离可以降低系统的耦合度，使得两者修改起来都相对容易。

在调用工厂类的工厂方法时，由于工厂方法是静态方法，使用起来很方便，可通过类名直接调用，而且只需要传入一个简单的参数即可，在实际开发中，还可以在调用时将所传入的参数保存在XML等格式的配置文件中，修改参数时无须修改任何源代码。

简单工厂模式最大的问题在于**工厂类的职责相对过重，增加新的产品需要修改工厂类的判断逻辑**（修改源代码），这一点与开闭原则是相违背的。

简单工厂模式的要点在于：当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须知道其创建细节。

## 特点

简单工厂模式的最大优点在于工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体产品的依赖。

但是，如果我们需要求解M的N次方运算，就应该在case语句中增加一个判断，修改原来的class类，这显然是不合理的，违背了开放-封闭原则，于是引入了下面的工厂方法模式。

### 优点

工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅“消费”产品；简单工厂模式通过这种做法实现了对责任的分割，它提供了专门的工厂类用于创建对象。

客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以减少使用者的记忆量。

通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性。

### 缺点

工厂类职责过重，由于工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都要受到影响。

使用简单工厂模式将会增加系统中类的个数，在一定程序上增加了系统的复杂度和理解难度。

系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。

违反了“开闭原则”，添加新产品需要修改工厂逻辑，工厂越来越复杂。

简单工厂模式由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。

## 适用环境

在以下情况下可以使用简单工厂模式：

工厂类负责创建的对象比较少：由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。

客户端只知道传入工厂类的参数，对于如何创建对象不关心：客户端既不需要关心创建细节，甚至连类名都不需要记住，只需要知道类型所对应的参数。

## 模式应用

1、JDK类库中广泛使用了简单工厂模式，如工具类java.text.DateFormat，它用于格式化一个本地日期或者时间。

public final static DateFormat getDateInstance();

public final static DateFormat getDateInstance(int style);

public final static DateFormat getDateInstance(int style,Localelocale);

2、Java加密技术

获取不同加密算法的密钥生成器:

KeyGenerator keyGen=KeyGenerator.getInstance("DESede");

创建密码器:

Cipher cp=Cipher.getInstance("DESede");

注：在我们的分布式数据库中，具体实例化PlanNode的哪一个子类，SQLNode，UnionNode，JoinNode则可以使用工厂模式。

## 总结

创建型模式对类的实例化过程进行了抽象，能够将对象的创建与对象的使用过程分离。

简单工厂模式又称为静态工厂方法模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

简单工厂模式包含三个角色：工厂角色负责实现创建所有实例的内部逻辑；抽象产品角色是所创建的所有对象的父类，负责描述所有实例所共有的公共接口；具体产品角色是创建目标，所有创建的对象都充当这个角色的某个具体类的实例。

简单工厂模式的要点在于：当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须知道其创建细节。

简单工厂模式最大的优点在于实现对象的创建和对象的使用分离，将对象的创建交给专门的工厂类负责，但是其最大的缺点在于工厂类不够灵活，增加新的具体产品需要修改工厂类的判断逻辑代码，而且产品较多时，工厂方法代码将会非常复杂。

简单工厂模式适用情况包括：工厂类负责创建的对象比较少；客户端只知道传入工厂类的参数，对于如何创建对象不关心。

# 工厂方法模式

## 背景

现在对该系统进行修改，不再设计一个按钮工厂类来统一负责所有产品的创建，而是将具体按钮的创建过程交给专门的工厂子类去完成，我们先定义一个抽象的按钮工厂类，再定义具体的工厂类来生成圆形按钮、矩形按钮、菱形按钮等，它们实现在抽象按钮工厂类中定义的方法。这种抽象化的结果使这种结构可以在不修改具体工厂类的情况下引进新的产品，如果出现新的按钮类型，只需要为这种新类型的按钮创建一个具体的工厂类就可以获得该新按钮的实例，这一特点无疑使得**工厂方法模式具有超越简单工厂模式的优越性，更加符合“开闭原则”**。

## 定义

工厂方法模式(Factory Method Pattern)又称为**工厂模式**，也叫虚拟构造器(Virtual Constructor)模式或者多态工厂(Polymorphic Factory)模式，它属于类创建型模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

工厂方法模式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，工厂方法使一个类的实例化延迟到其子类。

注：**简单工厂模式+开闭原则=工厂方法模式**

## 模式结构

工厂方法模式包含如下角色：

1、Product：抽象产品

2、ConcreteProduct：具体产品

3、Factory：抽象工厂

4、ConcreteFactory：具体工厂

## 时序图

## 代码实现

#include "ConcreteFactory.h"

#include "ConcreteProduct.h"

Product\* ConcreteFactory::factoryMethod(){

return new ConcreteProduct();

}

#include "Factory.h"

#include "ConcreteFactory.h"

#include "Product.h"

#include <iostream>

using namespace std;

int main(int argc, char \*argv[])

{

Factory \* fc = new ConcreteFactory();

Product \* prod = fc->factoryMethod();

prod->use();

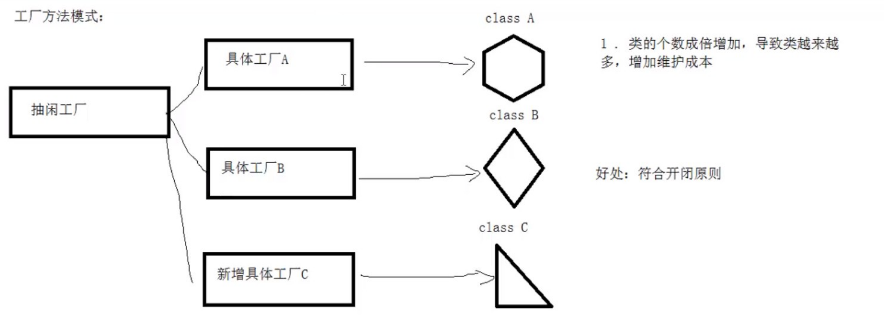
delete fc;

delete prod;

return 0;

}

**另外一个例子：**



#define CRT SECURE NO WARNINGS

#include<iostream>

#include<string>

using namespace std;

//抽象水果

class AbstractFruit{

public:

virtual void ShowName() = 0;

};

//苹果

class Apple :public AbstractFruit{

public:

virtual void ShowName(){

cout << "我是苹果!" << endl;

}

};

//香蕉

class Banana :public AbstractFruit{

public:

virtual void ShowName(){

cout << "我是香蕉!" << endl;

}

};

//鸭梨

class Pear :public AbstractFruit{

public:

virtual void ShowName(){

cout << "我是鸭梨!" <<endl;

}

};

//抽象工厂

class AbstractFruitFactory{

public:

virtual AbstractFruit\*CreateFruit() = 0;

};

// 苹果工厂

class AppleFactory :public AbstractFruitFactory {

public:

virtual AbstractFruit\*CreateFruit(){

return new Apple;

}

};

//香蕉工厂

class BananaFactory :public AbstractFruitFactory{

public:

virtual AbstractFruit\*CreateFruit(){

return new Banana;

}

};

//鸭梨

class PearFactory :public AbstractFruitFactory {

public:

virtual AbstractFruit\*CreateFruit(){

return new Pear;

}

};

void test01() {

AbstractFruitFactory\*factory = NULL;

AbstractFruit\*fruit = NULL;

//创建一个苹果工厂

factory = new AppleFactory;

//创建苹果

fruit = factory->CreateFruit();

fruit->ShowName();

delete fruit;

delete factory;

//创建一个香蕉工厂

factory = new BananaFactory;

//创建香蕉

fruit = factory->CreateFruit();

fruit->ShowName();

delete fruit;

delete factory;

//创建一个鸭梨工厂

factory = new PearFactory;

//创建鸭梨

fruit = factory->CreateFruit();

fruit->ShowName();

delete fruit;

delete factory;

}

int main()

{

test01();

return 0;

}

## 分析

工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责哪一个产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

## 实例

日志记录器

某系统日志记录器要求支持多种日志记录方式，如文件记录、数据库记录等，且用户可以根据要求动态选择日志记录方式，现使用工厂方法模式设计该系统。

## 特点

### 优点

在工厂方法模式中，工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节，用户只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名。不需要记住具体类名，甚至连具体参数都不用记忆。实现了对象创建和使用的分离。

基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够使工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式之所以又被称为多态工厂模式，是因为所有的具体工厂类都具有同一抽象父类。

使用工厂方法模式的另一个优点是在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，无需修改接口和原类，完全符合“开闭原则”。

注：工厂方法模式的优点就是符合开闭原则，但是很难维护。

### 缺点

在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。

由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到DOM、反射等技术，增加了系统的实现难度。

## 适用环境

在以下情况下可以使用工厂方法模式：

一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。

一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

## 模式应用

JDBC中的工厂方法:

Connection conn=DriverManager.getConnection("jdbc:microsoft:sqlserver://loc

alhost:1433; DatabaseName=DB;user=sa;password=");

Statement statement=conn.createStatement();

ResultSet rs=statement.executeQuery("select \* from UserInfo");

## 拓展

使用多个工厂方法：在抽象工厂角色中可以定义多个工厂方法，从而使具体工厂角色实现这些不同的工厂方法，这些方法可以包含不同的业务逻辑，以满足对不同的产品对象的需求。

产品对象的重复使用：工厂对象将已经创建过的产品保存到一个集合（如数组、List等）中，然后根据客户对产品的请求，对集合进行查询。如果有满足要求的产品对象，就直接将该产品返回客户端；如果集合中没有这样的产品对象，那么就创建一个新的满足要求的产品对象，然后将这个对象在增加到集合中，再返回给客户端。

多态性的丧失和模式的退化：如果工厂仅仅返回一个具体产品对象，便违背了工厂方法的用意，发生退化，此时就不再是工厂方法模式了。一般来说，工厂对象应当有一个抽象的父类型，如果工厂等级结构中只有一个具体工厂类的话，抽象工厂就可以省略，也将发生了退化。当只有一个具体工厂，在具体工厂中可以创建所有的产品对象，并且工厂方法设计为静态方法时，工厂方法模式就退化成简单工厂模式。

## 总结

工厂方法模式又称为工厂模式，它属于类创建型模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

工厂方法模式包含四个角色：抽象产品是定义产品的接口，是工厂方法模式所创建对象的超类型，即产品对象的共同父类或接口；具体产品实现了抽象产品接口，某种类型的具体产品由专门的具体工厂创建，它们之间往往一一对应；抽象工厂中声明了工厂方法，用于返回一个产品，它是工厂方法模式的核心，任何在模式中创建对象的工厂类都必须实现该接口；具体工厂是抽象工厂类的子类，实现了抽象工厂中定义的工厂方法，并可由客户调用，返回一个具体产品类的实例。

工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

工厂方法模式的主要优点是增加新的产品类时无须修改现有系统，并封装了产品对象的创建细节，系统具有良好的灵活性和可扩展性；其缺点在于增加新产品的同时需要增加新的工厂，导致系统类的个数成对增加，在一定程度上增加了系统的复杂性。

工厂方法模式适用情况包括：一个类不知道它所需要的对象的类；一个类通过其子类来指定创建哪个对象；将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定。

# 抽象工厂模式

## 背景

在工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。但是有时候我们需要一个工厂可以提供多个产品对象，而不是单一的产品对象。

为了更清晰地理解工厂方法模式，需要先引入两个概念：

产品等级结构 ：产品等级结构即产品的继承结构，如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。

产品族 ：在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中。

当系统所提供的工厂所需生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构中属于不同类型的具体产品时需要使用抽象工厂模式。

抽象工厂模式针对的是产品族，而不是产品等级结构。

抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。

抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构，一个工厂等级结构可以负责多个不同产品等级结构中的产品对象的创建。当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象时，抽象工厂模式比工厂方法模式更为简单、有效率。

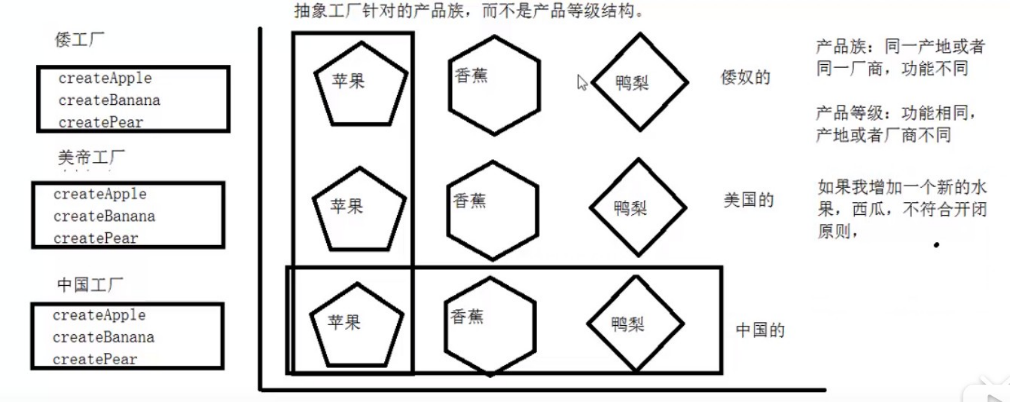
## 定义

抽象工厂模式(Abstract Factory Pattern)：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为Kit模式，属于对象创建型模式。

## 模式结构

## 时序图

## 代码实现



注：横着看是产品族，竖着是产品等级结构。

#define CRT SECURE NO WARNINGS

#include<iostream>

#include<string>

using namespace std;

//抽象苹果

class AbstractApple{

public:

virtual void ShowName() = 0;

};

//中国苹果

class ChinaApple :public AbstractApple{

public:

virtual void ShowName(){

cout << "中国苹果!" << endl;

}

};

//美国苹果

class USAApple :public AbstractApple{

public:

virtual void ShowName(){

cout << "美国苹果!" << endl;

}

};

//倭国苹果

class JapanApple :public AbstractApple{

public:

virtual void ShowName(){

cout << "倭国苹果!" << endl;

}

};

//抽象香蕉

class AbsrtactBanana{

public:

virtual void ShowName() = 0;

};

//中国香蕉

class ChinaBanana :public AbsrtactBanana{

public:

virtual void ShowName(){

cout << "中国香蕉!" << endl;

}

};

//美国香蕉

class USABanana :public AbsrtactBanana {

public:

virtual void ShowName(){

cout << "美国香蕉!" << endl;

}

};

//倭国香蕉

class JapanBanana :public AbsrtactBanana{

public:

virtual void ShowName(){

cout << "倭国香蕉!" << endl;

}

};

/////////////////////////////////////////////////////////

// 抽象鸭梨

class AbstractPear{

public:

virtual void ShowName() = 0;

};

//中国鸭梨

class ChinaPear :public AbstractPear{

public:

virtual void ShowName(){

cout << "中国鸭梨!" << endl;

}

};

//美国鸭梨

class USAPear :public AbstractPear {

public:

virtual void ShowName(){

cout << "美国鸭梨!" << endl;

}

};

//倭国鸭梨

class JapanPear :public AbstractPear{

public:

virtual void ShowName(){

cout << "倭国鸭梨!" << endl;

}

};

//抽象工厂针对产品族

class AbstracFactory{

public:

virtual AbstractApple\*CreateApple() = 0;

virtual AbsrtactBanana\*CreateBanana() = 0;

virtual AbstractPear\*CreatePear() = 0;

};

//中国工厂

class ChinaFactory:public AbstracFactory{

public:

virtual AbstractApple\*CreateApple() {

return new ChinaApple;

}

virtual AbsrtactBanana\*CreateBanana() {

return new ChinaBanana;

}

virtual AbstractPear\*CreatePear() {

return new ChinaPear;

}

};

//美国工厂

class USAFactory :public AbstracFactory{

public:

virtual AbstractApple\*CreateApple() {

return new USAApple;

}

virtual AbsrtactBanana\*CreateBanana() {

return new USABanana;

}

virtual AbstractPear\*CreatePear() {

return new USAPear;

}

};

//倭国工厂

class JapanFactory :public AbstracFactory{

public:

virtual AbstractApple\*CreateApple() {

return new JapanApple;

}

virtual AbsrtactBanana\*CreateBanana() {

return new JapanBanana;

}

virtual AbstractPear\*CreatePear() {

return new JapanPear;

}

};

void test01() {

AbstracFactory\* factory = NULL;

AbstractApple\* apple = NULL;

AbsrtactBanana\* banana = NULL;

AbstractPear\* pear = NULL;

//中国工厂

factory = new ChinaFactory;

apple = factory->CreateApple();

banana = factory->CreateBanana();

pear = factory->CreatePear();

apple->ShowName();

banana->ShowName ();

pear->ShowName();

delete pear;

delete banana;

delete apple;

delete factory;

}

int main()

{

test01();

return 0;

}

## 分析

## 实例

## 特点

### 优点

抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。另外，应用抽象工厂模式可以实现高内聚低耦合的设计目的，因此抽象工厂模式得到了广泛的应用。

当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象。这对一些需要根据当前环境来决定其行为的软件系统来说，是一种非常实用的设计模式。

增加新的具体工厂和产品族很方便，无须修改已有系统，符合“开闭原则”。

### 缺点

在添加新的产品对象时，难以扩展抽象工厂来生产新种类的产品，这是因为在抽象工厂角色中规定了所有可能被创建的产品集合，要支持新种类的产品就意味着要对该接口进行扩展，而这将涉及到对抽象工厂角色及其所有子类的修改，显然会带来较大的不便。

开闭原则的倾斜性（增加新的工厂和产品族容易，增加新的产品等级结构麻烦）。

## 适用环境

在以下情况下可以使用抽象工厂模式：

一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是重要的。

系统中有多于一个的产品族，而每次只使用其中某一产品族。

属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。

系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。

## 模式应用

在很多软件系统中需要更换界面主题，要求界面中的按钮、文本框、背景色等一起发生改变时，可以使用抽象工厂模式进行设计。

## 模式扩展

### “开闭原则”的倾斜性

### 工厂模式的退化

## 总结

抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为Kit模式，属于对象创建型模式。

抽象工厂模式包含四个角色：抽象工厂用于声明生成抽象产品的方法；具体工厂实现了抽象工厂声明的生成抽象产品的方法，生成一组具体产品，这些产品构成了一个产品族，每一个产品都位于某个产品等级结构中；抽象产品为每种产品声明接口，在抽象产品中定义了产品的抽象业务方法；具体产品定义具体工厂生产的具体产品对象，实现抽象产品接口中定义的业务方法。

抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构。

抽象工厂模式的主要优点是隔离了具体类的生成，使得客户并不需要知道什么被创建，而且每次可以通过具体工厂类创建一个产品族中的多个对象，增加或者替换产品族比较方便，增加新的具体工厂和产品族很方便；主要缺点在于增加新的产品等级结构很复杂，需要修改抽象工厂和所有的具体工厂类，对“开闭原则”的支持呈现倾斜性。

抽象工厂模式适用情况包括：一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节；系统中有多于一个的产品族，而每次只使用其中某一产品族；属于同一个产品族的产品将在一起使用；系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。