

创建型模式

创建型模式(Creational Pattern)对类的实例化过程进行了抽象，能够将软件模块中对象的创建和对象的使用分离。为了使软件的结构更加清晰，外界对于这些对象只需要知道它们共同的接口，而不清楚其具体的实现细节，使整个系统的设计更加符合单一职责原则。

创建型模式在创建什么(What)，由谁创建(Who)，何时创建(When)等方面都为软件设计者提供了尽可能大的灵活性。创建型模式隐藏了类的实例的创建细节，通过隐藏对象如何被创建和组合在一起达到使整个系统独立的目的。

包含模式

- 简单工厂模式 (**Simple Factory**)
重要程度：4 (5为满分)
- 工厂方法模式 (**Factory Method**)
重要程度：5
- 抽象工厂模式 (**Abstract Factory**)
重要程度：5
- 建造者模式 (**Builder**)
重要程度：2
- 原型模式 (**Prototype**)
重要程度：3
- 单例模式 (**Singleton**)
重要程度：4

目录

- 1. 简单工厂模式(Simple Factory Pattern)
 - [1.1. 模式动机](#)
 - [1.2. 模式定义](#)
 - [1.3. 模式结构](#)
 - [1.4. 时序图](#)
 - [1.5. 代码分析](#)
 - [1.6. 模式分析](#)
 - [1.7. 实例](#)

- 1.8. 简单工厂模式的优点
- 1.9. 简单工厂模式的缺点
- 1.10. 适用环境
- 1.11. 模式应用
- 1.12. 总结
- 2. 工厂方法模式(Factory Method Pattern)
 - 2.1. 模式动机
 - 2.2. 模式定义
 - 2.3. 模式结构
 - 2.4. 时序图
 - 2.5. 代码分析
 - 2.6. 模式分析
 - 2.7. 实例
 - 2.8. 工厂方法模式的优点
 - 2.9. 工厂方法模式的缺点
 - 2.10. 适用环境
 - 2.11. 模式应用
 - 2.12. 模式扩展
 - 2.13. 总结
- 3. 抽象工厂模式(Abstract Factory)
 - 3.1. 模式动机
 - 3.2. 模式定义
 - 3.3. 模式结构
 - 3.4. 时序图
 - 3.5. 代码分析
 - 3.6. 模式分析
 - 3.7. 实例
 - 3.8. 优点
 - 3.9. 缺点
 - 3.10. 适用环境
 - 3.11. 模式应用
 - 3.12. 模式扩展
 - 3.13. 总结
- 4. 建造者模式
 - 4.1. 模式动机
 - 4.2. 模式定义
 - 4.3. 模式结构
 - 4.4. 时序图
 - 4.5. 代码分析
 - 4.6. 模式分析
 - 4.7. 实例
 - 4.8. 优点
 - 4.9. 缺点
 - 4.10. 适用环境
 - 4.11. 模式应用

- 4.12. 模式扩展
 - 4.13. 总结
- 5. 单例模式
 - 5.1. 模式动机
 - 5.2. 模式定义
 - 5.3. 模式结构
 - 5.4. 时序图
 - 5.5. 代码分析
 - 5.6. 模式分析
 - 5.7. 实例
 - 5.8. 优点
 - 5.9. 缺点
 - 5.10. 适用环境
 - 5.11. 模式应用
 - 5.12. 模式扩展
 - 5.13. 总结

1. 简单工厂模式(Simple Factory Pattern)

目录

- 简单工厂模式(Simple Factory Pattern)
 - 模式动机
 - 模式定义
 - 模式结构
 - 时序图
 - 代码分析
 - 模式分析
 - 实例
 - 简单工厂模式的优点
 - 简单工厂模式的缺点
 - 适用环境
 - 模式应用
 - 总结

1.1. 模式动机

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。

1.2. 模式定义

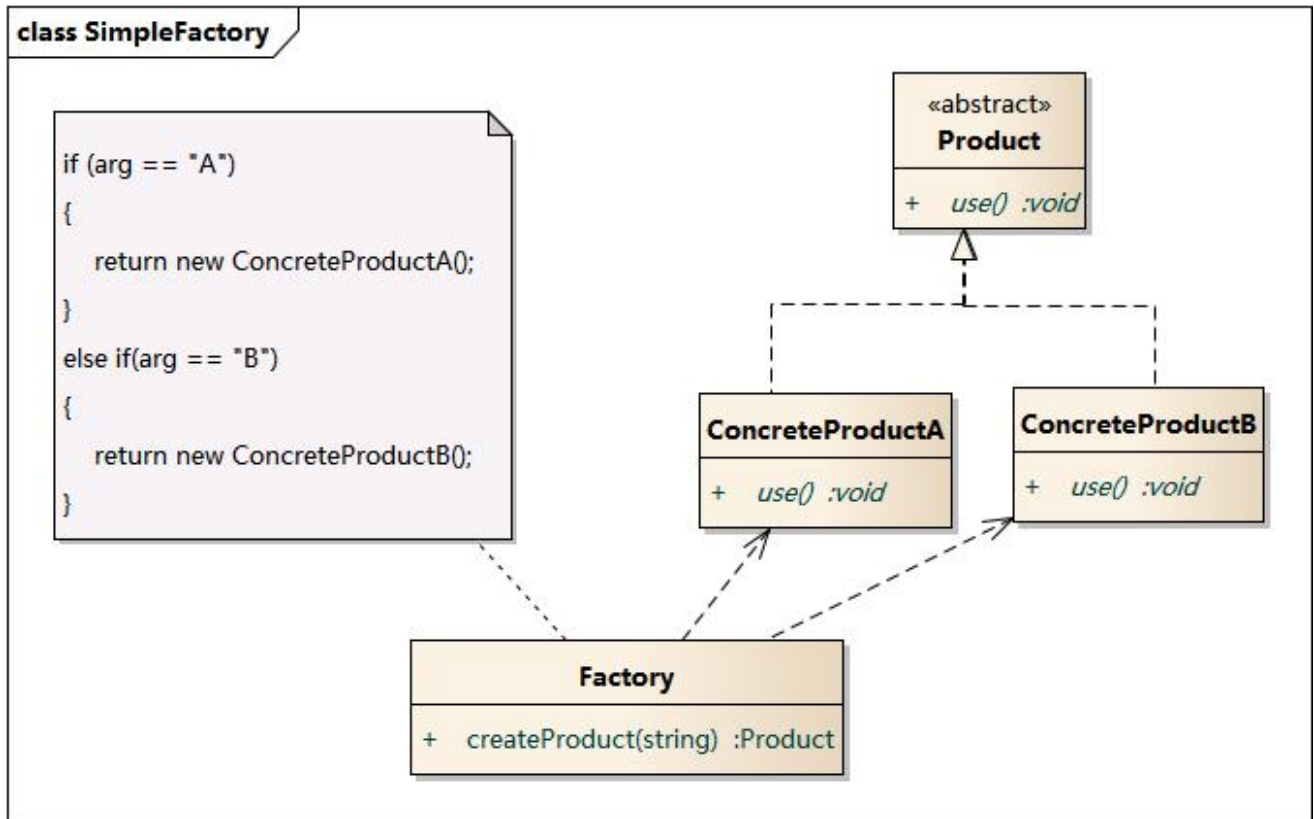
简单工厂模式(Simple Factory Pattern)：又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

1.3. 模式结构

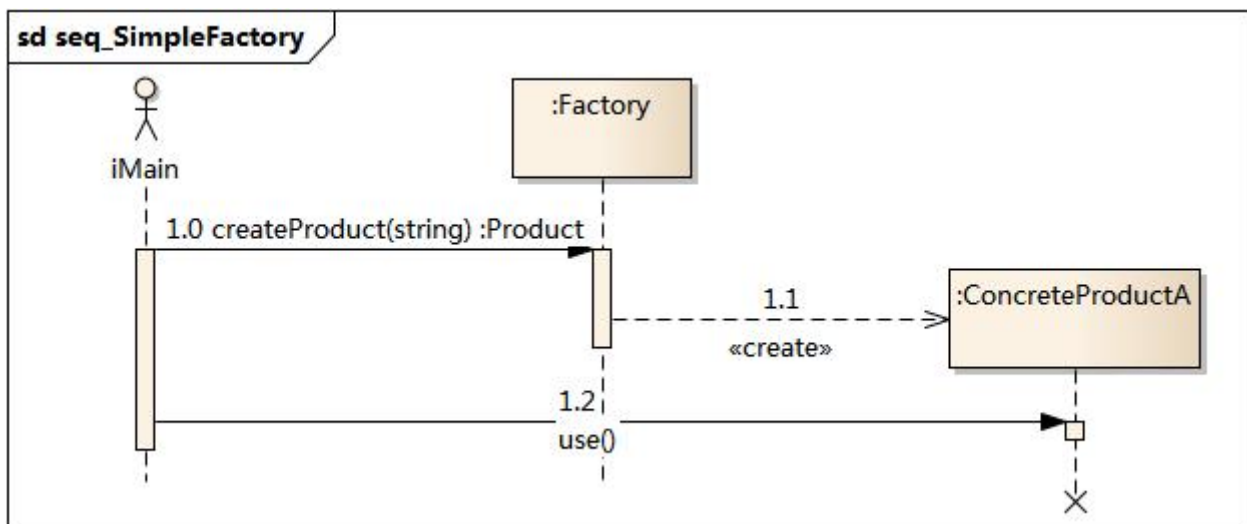
简单工厂模式包含如下角色：

- **Factory**：工厂角色
工厂角色负责实现创建所有实例的内部逻辑

- **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
- **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。



1.4. 时序图



1.5. 代码分析

```

1  //////////////////////////////////////
2  //  Factory.cpp
3  //  Implementation of the Class Factory
4  //  Created on:      01-十月-2014 18:41:33
5  //  Original author: colin
6  //////////////////////////////////////
7
8  #include "Factory.h"
9  #include "ConcreteProductA.h"
10 #include "ConcreteProductB.h"
11 Product* Factory::createProduct(string proname){
12     if ( "A" == proname )
13     {
14         return new ConcreteProductA();
15     }
16     else if("B" == proname)
17     {
18         return new ConcreteProductB();
19     }
20     return NULL;
21 }

```

1.6. 模式分析

- 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，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。

1.7. 实例

(略)

1.8. 简单工厂模式的优点

- 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。

1.9. 简单工厂模式的缺点

- 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，在一定程度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。
- 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。

1.10. 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。

1.11. 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,Locale  
locale);
```

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");
```

1.12. 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。

2. 工厂方法模式(Factory Method Pattern)

目录

- [工厂方法模式\(Factory Method Pattern\)](#)

- [模式动机](#)
- [模式定义](#)
- [模式结构](#)
- [时序图](#)
- [代码分析](#)
- [模式分析](#)
- [实例](#)
- [工厂方法模式的优点](#)
- [工厂方法模式的缺点](#)
- [适用环境](#)
- [模式应用](#)
- [模式扩展](#)
- [总结](#)

2.1. 模式动机

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”。

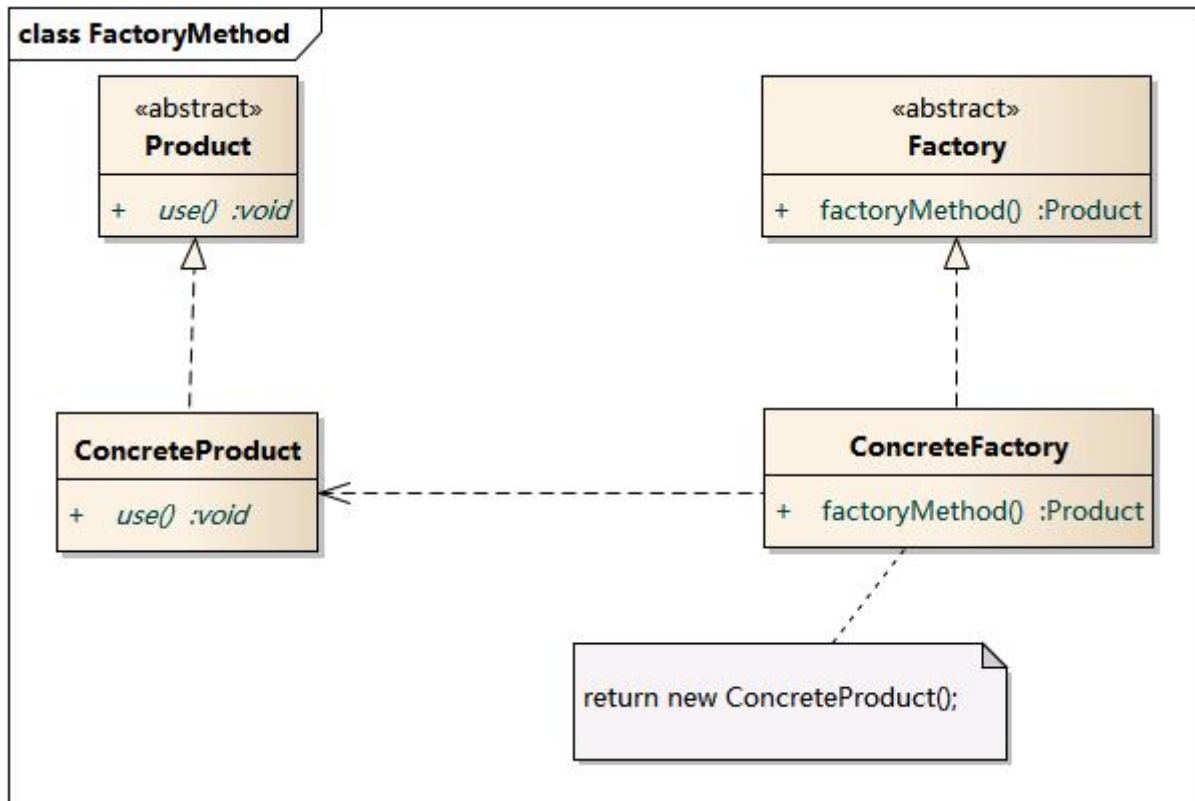
2.2. 模式定义

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。

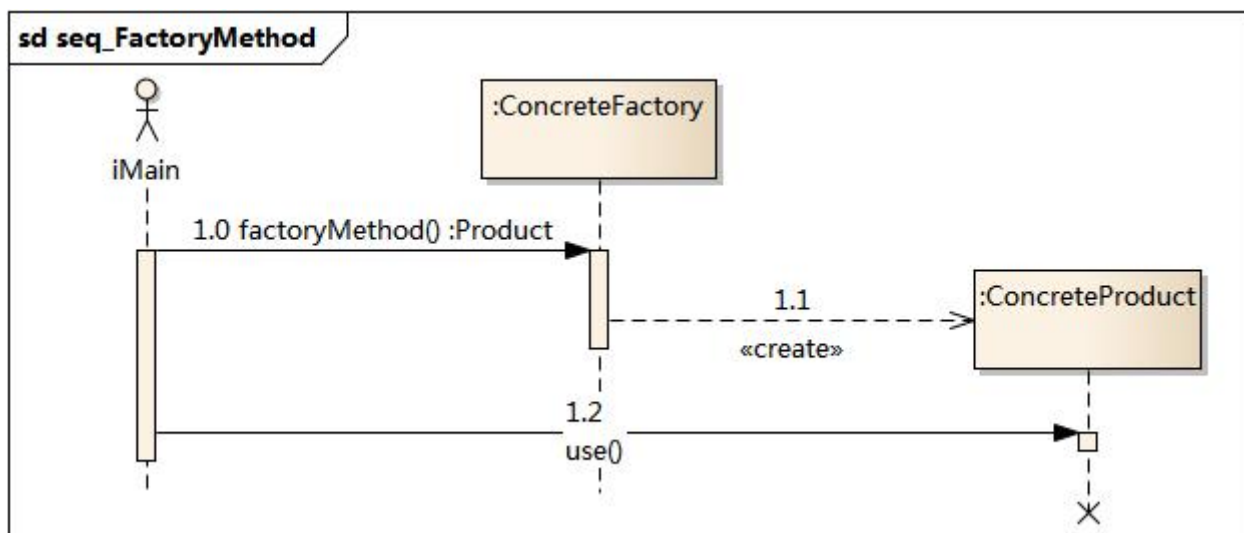
2.3. 模式结构

工厂方法模式包含如下角色：

- Product：抽象产品
- ConcreteProduct：具体产品
- Factory：抽象工厂
- ConcreteFactory：具体工厂



2.4. 时序图



2.5. 代码分析

```

1  //////////////////////////////////////
2  //  ConcreteFactory.cpp
3  //  Implementation of the Class ConcreteFactory
4  //  Created on:      02-十月-2014 10:18:58
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "ConcreteFactory.h"
8  #include "ConcreteProduct.h"
9
10 Product* ConcreteFactory::factoryMethod(){
11     return new ConcreteProduct();
12 }
13
14

```

```

1  #include "Factory.h"
2  #include "ConcreteFactory.h"
3  #include "Product.h"
4  #include <iostream>
5  using namespace std;
6
7  int main(int argc, char *argv[])
8  {
9      Factory * fc = new ConcreteFactory();
10     Product * prod = fc->factoryMethod();
11     prod->use();
12
13     delete fc;
14     delete prod;
15
16     return 0;
17 }

```

2.6. 模式分析

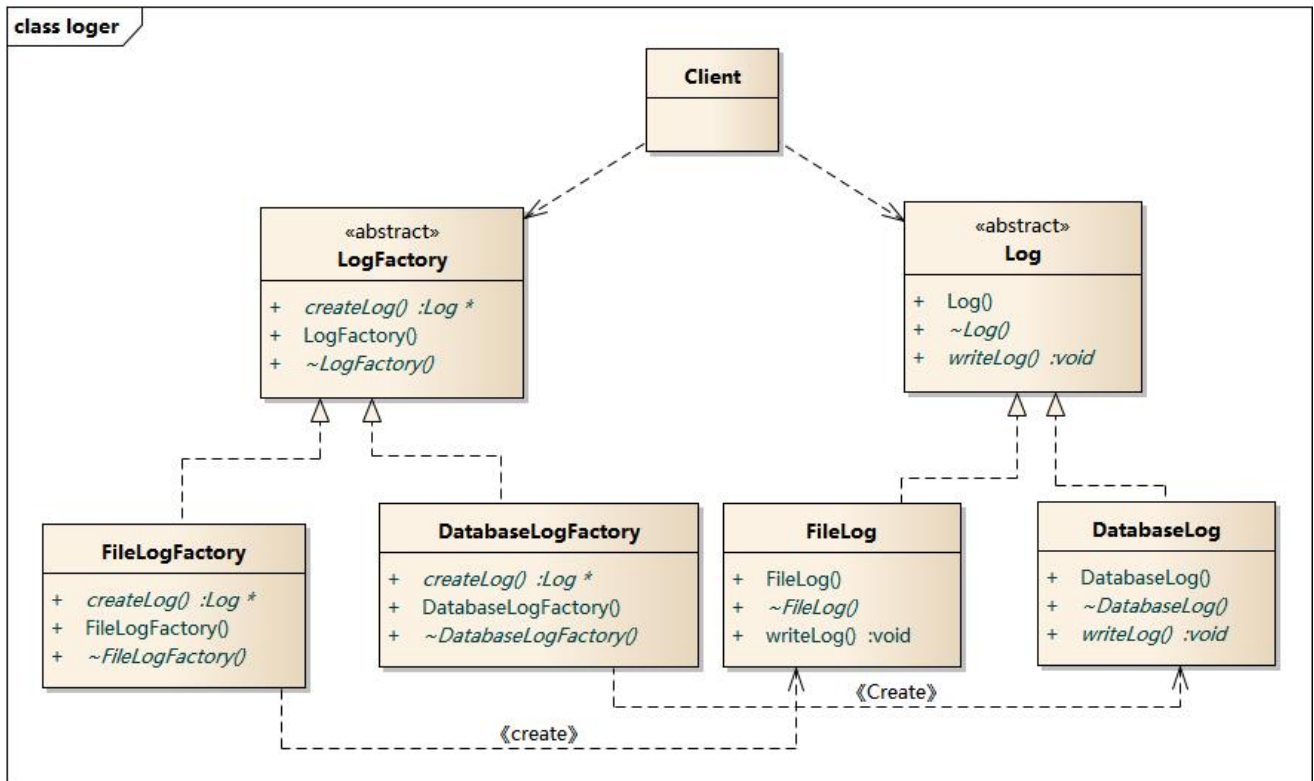
工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不承担哪一个产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

2.7. 实例

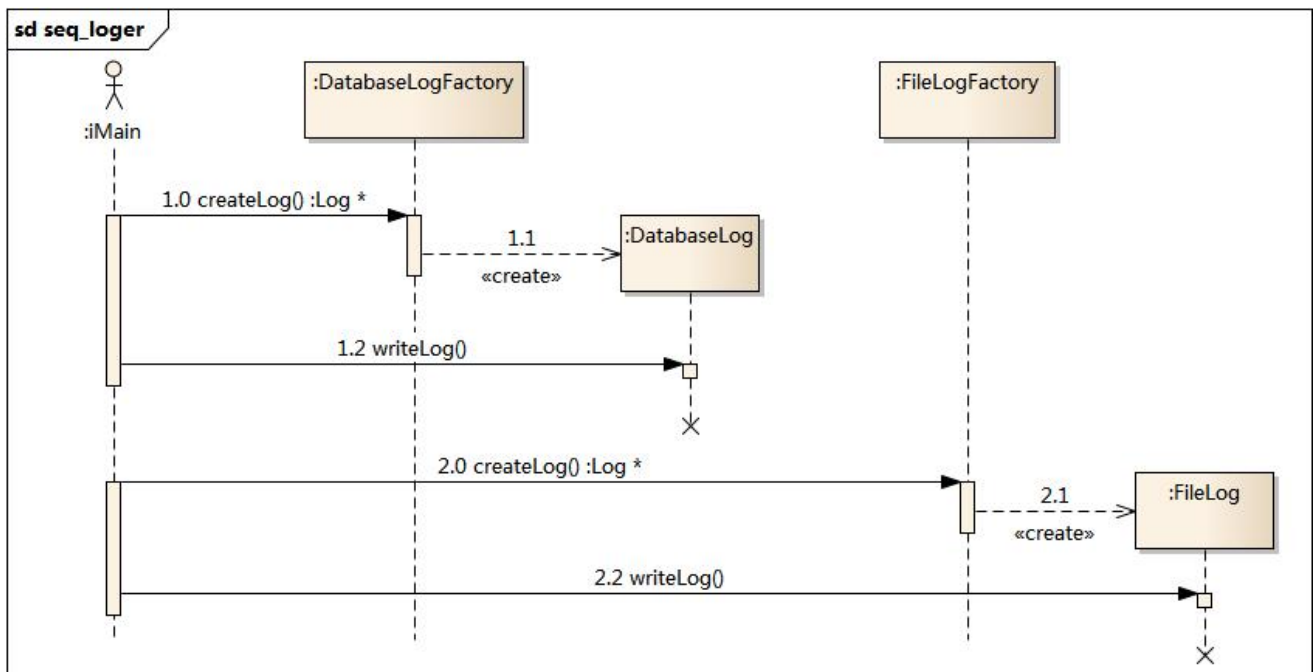
日志记录器

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。

结构图：



时序图：



2.8. 工厂方法模式的优点

- 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。
- 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，完全符合“开闭原则”。

2.9. 工厂方法模式的缺点

- 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。

2.10. 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。

2.11. 模式应用

JDBC中的工厂方法:

```
Connection conn=DriverManager.getConnection("jdbc:microsoft:sqlserver://localhost:1433; DatabaseName=DB;user=sa;password=");
Statement statement=conn.createStatement();
ResultSet rs=statement.executeQuery("select * from UserInfo");
```

2.12. 模式扩展

- 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。

2.13. 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。

3. 抽象工厂模式(Abstract Factory)

目录

- 抽象工厂模式(Abstract Factory)
 - 模式动机
 - 模式定义
 - 模式结构
 - 时序图
 - 代码分析
 - 模式分析
 - 实例
 - 优点
 - 缺点
 - 适用环境
 - 模式应用
 - 模式扩展
 - “开闭原则”的倾斜性
 - 工厂模式的退化
 - 总结

3.1. 模式动机

- 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。

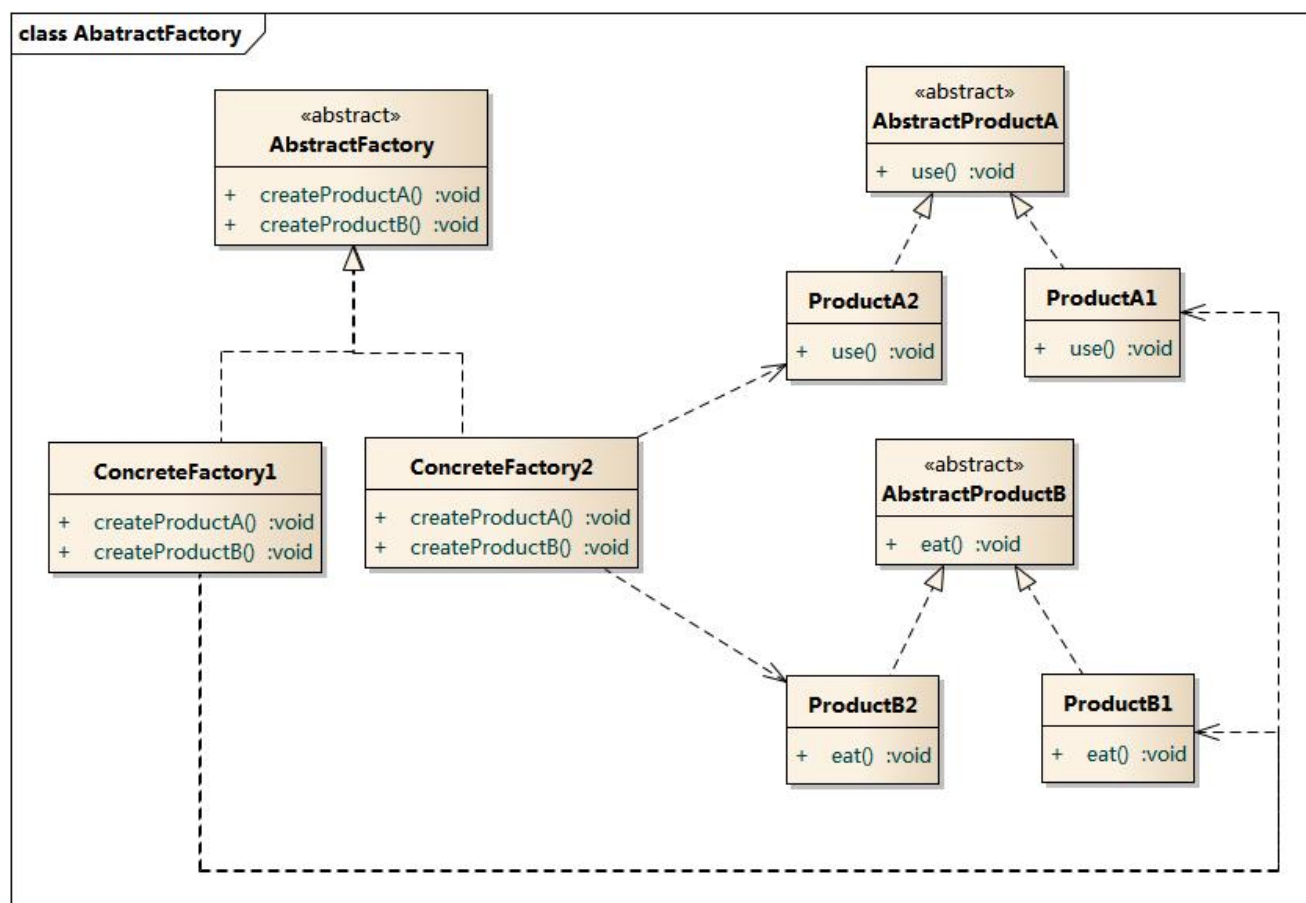
3.2. 模式定义

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。

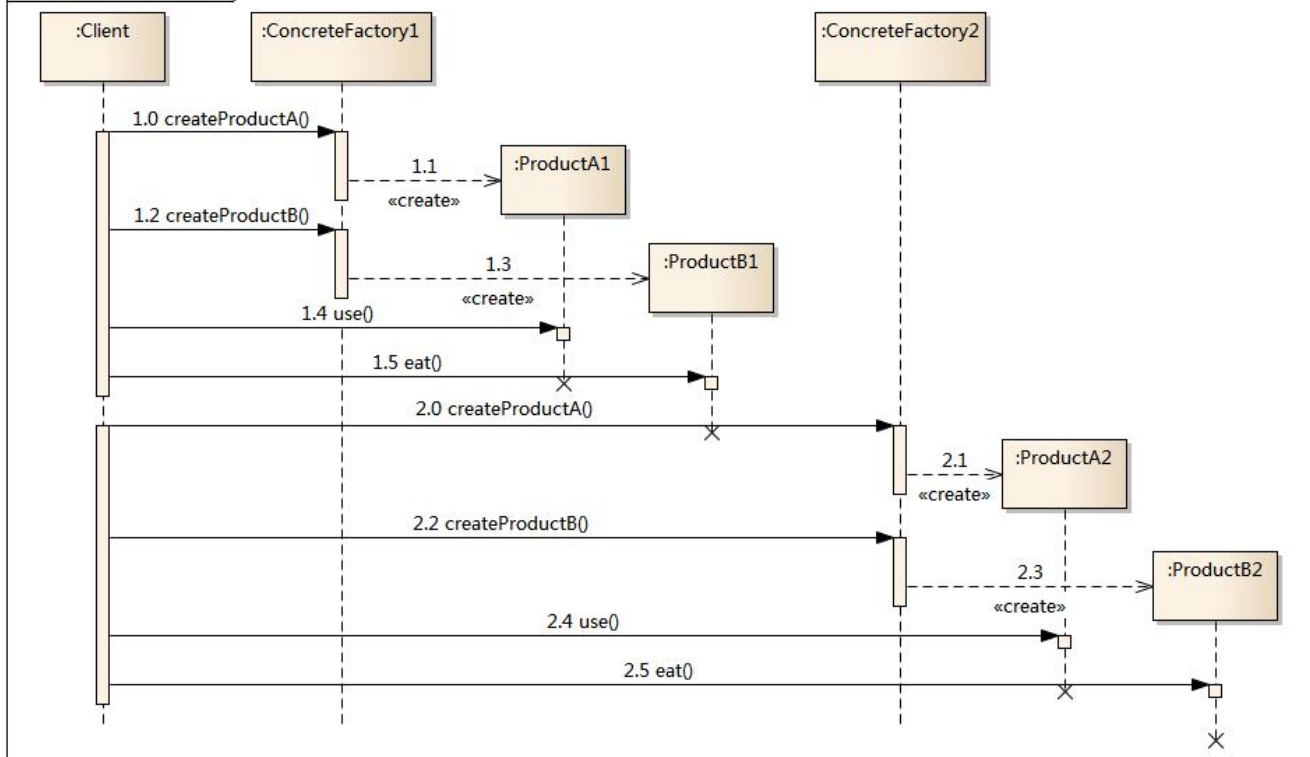
3.3. 模式结构

抽象工厂模式包含如下角色：

- AbstractFactory：抽象工厂
- ConcreteFactory：具体工厂
- AbstractProduct：抽象产品
- Product：具体产品



3.4. 时序图



3.5. 代码分析

```

1  #include <iostream>
2  #include "AbstractFactory.h"
3  #include "AbstractProductA.h"
4  #include "AbstractProductB.h"
5  #include "ConcreteFactory1.h"
6  #include "ConcreteFactory2.h"
7  using namespace std;
8
9  int main(int argc, char *argv[])
10 {
11     AbstractFactory * fc = new ConcreteFactory1();
12     AbstractProductA * pa = fc->createProductA();
13     AbstractProductB * pb = fc->createProductB();
14     pa->use();
15     pb->eat();
16
17     AbstractFactory * fc2 = new ConcreteFactory2();
18     AbstractProductA * pa2 = fc2->createProductA();
19     AbstractProductB * pb2 = fc2->createProductB();
20     pa2->use();
21     pb2->eat();
  
```

```

1  //////////////////////////////////////
2  //  ConcreteFactory1.cpp
3  //  Implementation of the Class ConcreteFactory1
4  //  Created on:      02-十月-2014 15:04:11
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "ConcreteFactory1.h"
8  #include "ProductA1.h"
9  #include "ProductB1.h"
10 AbstractProductA * ConcreteFactory1::createProductA(){
11     return new ProductA1();
12 }
13
14 AbstractProductB * ConcreteFactory1::createProductB(){
15     return new ProductB1();
16 }
17
18

```

```

1  //////////////////////////////////////
2  //  ProductA1.cpp
3  //  Implementation of the Class ProductA1
4  //  Created on:      02-十月-2014 15:04:17
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "ProductA1.h"
8  #include <iostream>
9  using namespace std;
10 void ProductA1::use(){
11     cout << "use Product A1" << endl;
12 }
13

```

运行结果：

```

"C:\Users\cl\Documents\GitHub\'
use Product A1
eat Product B1
use Product A2
eat Product B2
请按任意键继续. . .

```

3.6. 模式分析

3.7. 实例

3.8. 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”。

3.9. 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）。

3.10. 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。

3.11. 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。

3.12. 模式扩展

“开闭原则”的倾斜性

- “开闭原则”要求系统对扩展开放，对修改封闭，通过扩展达到增强其功能的目的。对于涉及到多个产品族与多个产品等级结构的系统，其功能增强包括两方面：
 1. 增加产品族：对于增加新的产品族，工厂方法模式很好的支持了“开闭原则”，对于新增加的产品族，只需要对应增加一个新的具体工厂即可，对已有代码无须做任何修改。

2. 增加新的产品等级结构：对于增加新的产品等级结构，需要修改所有的工厂角色，包括抽象工厂类，在所有的工厂类中都需要增加生产新产品的方法，不能很好地支持“开闭原则”。

- 抽象工厂模式的这种性质称为“开闭原则”的倾斜性，抽象工厂模式以一种倾斜的方式支持增加新的产品，它为新产品族的增加提供方便，但不能为新的产品等级结构的增加提供这样的方便。

工厂模式的退化

- 当抽象工厂模式中每一个具体工厂类只创建一个产品对象，也就是只存在一个产品等级结构时，抽象工厂模式退化成工厂方法模式；当工厂方法模式中抽象工厂与具体工厂合并，提供一个统一的工厂来创建产品对象，并将创建对象的工厂方法设计为静态方法时，工厂方法模式退化成简单工厂模式。

3.13. 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。

4. 建造者模式

目录

- [建造者模式](#)
 - [模式动机](#)
 - [模式定义](#)
 - [模式结构](#)
 - [时序图](#)
 - [代码分析](#)
 - [模式分析](#)
 - [实例](#)
 - [优点](#)
 - [缺点](#)
 - [适用环境](#)
 - [模式应用](#)
 - [模式扩展](#)
 - [总结](#)

4.1. 模式动机

无论是在现实世界中还是在软件系统中，都存在一些复杂的对象，它们拥有多个组成部分，如汽车，它包括车轮、方向盘、发送机等各种部件。而对于大多数用户而言，无须知道这些部件的装配细节，也几乎不会使用单独某个部件，而是使用一辆完整的汽车，可以通过建造者模式对其进行设计与描述，建造者模式可以将部件和其组装过程分开，一步一步创建一个复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，而无须知道其内部的具体构造细节。

在软件开发中，也存在大量类似汽车一样的复杂对象，它们拥有一系列成员属性，这些成员属性中有些是引用类型的成员对象。而且在这些复杂对象中，还可能存在一些限制条件，如某些属性没有赋值则复杂对象不能作为一个完整的产品使用；有些属性的赋值必须按照某个顺序，一个属性没有赋值之前，另一个属性可能无法赋值等。

复杂对象相当于一辆有待建造的汽车，而对象的属性相当于汽车的部件，建造产品的过程就相当于组合部件的过程。由于组合部件的过程很复杂，因此，这些部件的组合过程往往被“外部化”到一个称作建造者的对象里，建造者返还给客户端的是一个已经建造完毕的完整产品对象，而用户无须关心该对象所包含的属性以及它们的组装方式，这就是建造者模式的模式动机。

4.2. 模式定义

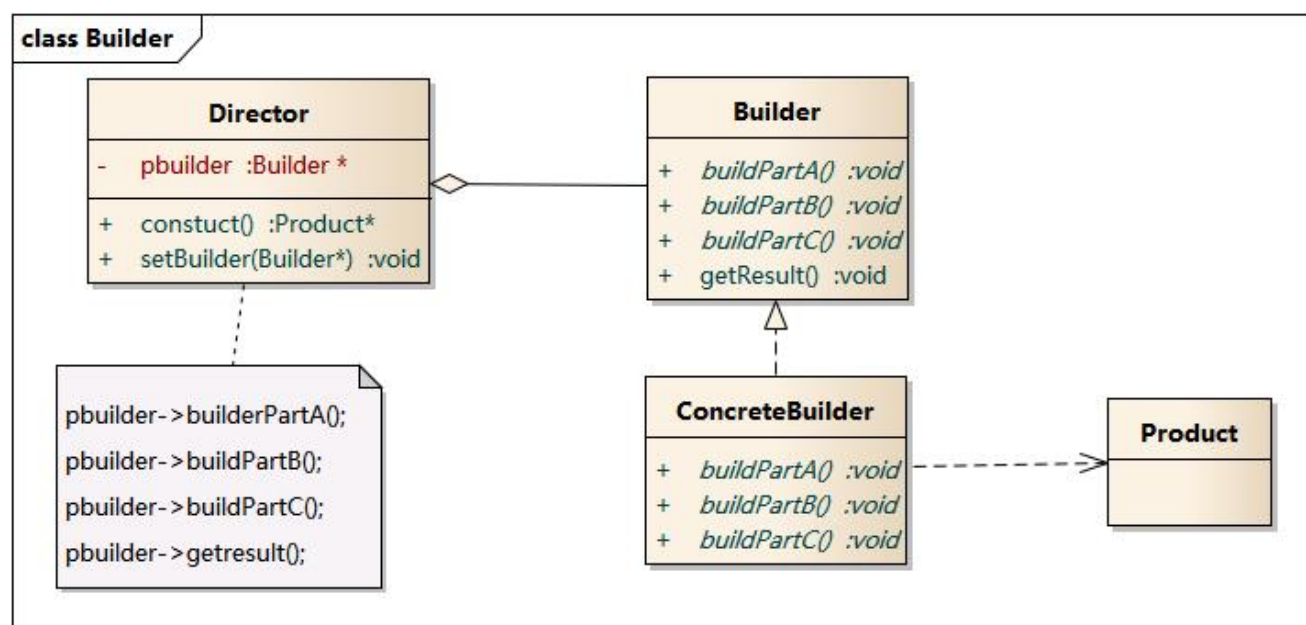
造者模式(Builder Pattern): 将一个复杂对象的构建与它的表示分离, 使得同样的构建过程可以创建不同的表示。

建造者模式是一步一步创建一个复杂的对象, 它允许用户只通过指定复杂对象的类型和内容就可以构建它们, 用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式。根据中文翻译的不同, 建造者模式又可以称为生成器模式。

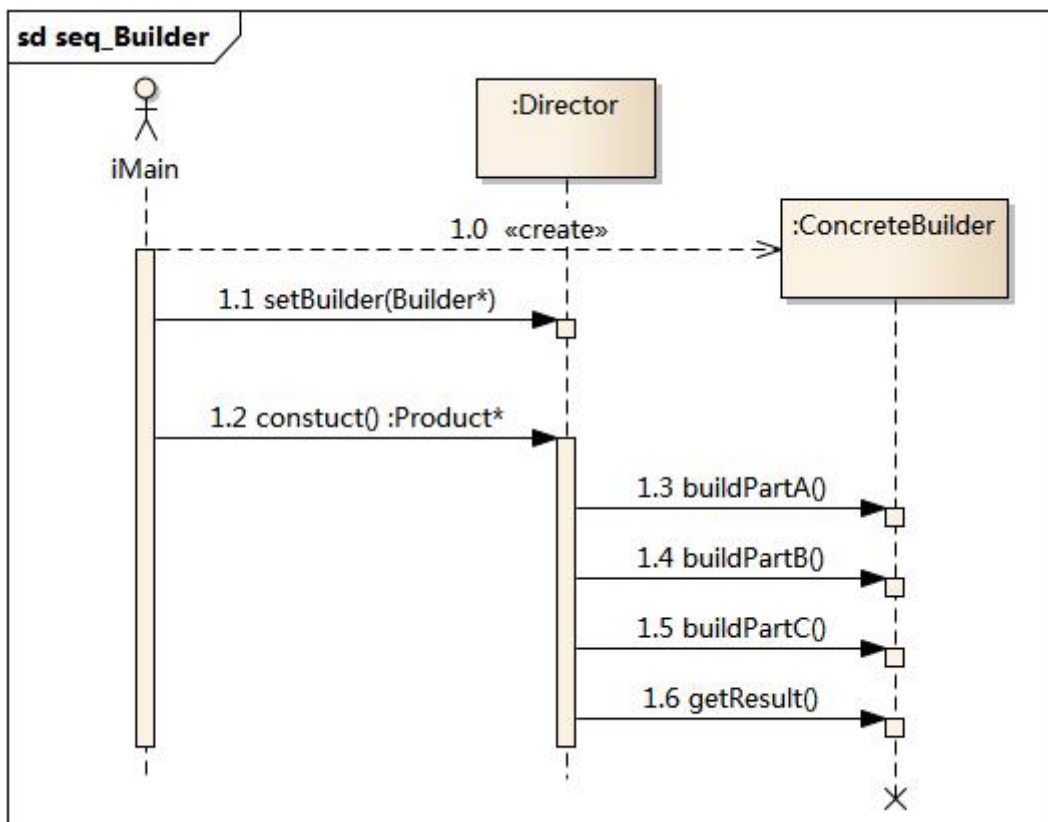
4.3. 模式结构

建造者模式包含如下角色:

- Builder: 抽象建造者
- ConcreteBuilder: 具体建造者
- Director: 指挥者
- Product: 产品角色



4.4. 时序图



4.5. 代码分析

```

1  #include <iostream>
2  #include "ConcreteBuilder.h"
3  #include "Director.h"
4  #include "Builder.h"
5  #include "Product.h"
6
7  using namespace std;
8
9  int main(int argc, char *argv[])
10 {
11     ConcreteBuilder * builder = new ConcreteBuilder();
12     Director director;
13     director.setBuilder(builder);
14     Product * pd = director.constuct();
15     pd->show();
16
17     delete builder;
18     delete pd;
19     return 0;
20 }
  
```

```
1  //////////////////////////////////////
2  //  ConcreteBuilder.cpp
3  //  Implementation of the Class ConcreteBuilder
4  //  Created on:      02-十月-2014 15:57:03
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "ConcreteBuilder.h"
8
9  ConcreteBuilder::ConcreteBuilder(){
10 }
11
12
13 ConcreteBuilder::~ConcreteBuilder(){
14 }
15
16 void ConcreteBuilder::buildPartA(){
17     m_prod->setA("A Style "); // 不同的建造者，可以实现不同产品的建造
18 }
19
20 void ConcreteBuilder::buildPartB(){
21     m_prod->setB("B Style ");
22 }
23
24 void ConcreteBuilder::buildPartC(){
25     m_prod->setC("C style ");
26 }
27
28
29
30
31
32
33
```

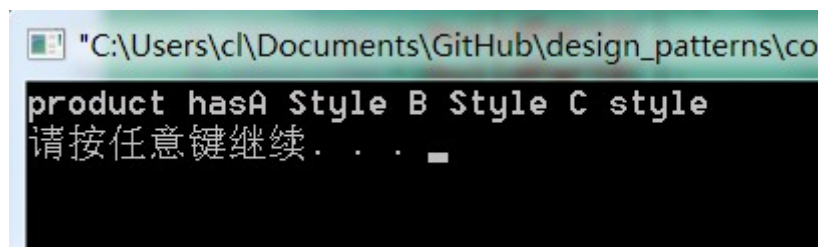


```

1  //////////////////////////////////////
2  //  Director.cpp
3  //  Implementation of the Class Director
4  //  Created on:      02-十月-2014 15:57:01
5  //  Original author: colin
6  //////////////////////////////////////
7
8  #include "Director.h"
9
10 Director::Director(){
11 }
12
13 Director::~Director(){
14 }
15
16 Product* Director::constuct(){
17     m_pbuilder->buildPartA();
18     m_pbuilder->buildPartB();
19     m_pbuilder->buildPartC();
20
21     return m_pbuilder->getResult();
22 }
23
24 void Director::setBuilder(Builder* buider){
25     m_pbuilder = buider;
26 }
27

```

运行结果：



4.6. 模式分析

抽象建造者类中定义了产品的创建方法和返回方法;

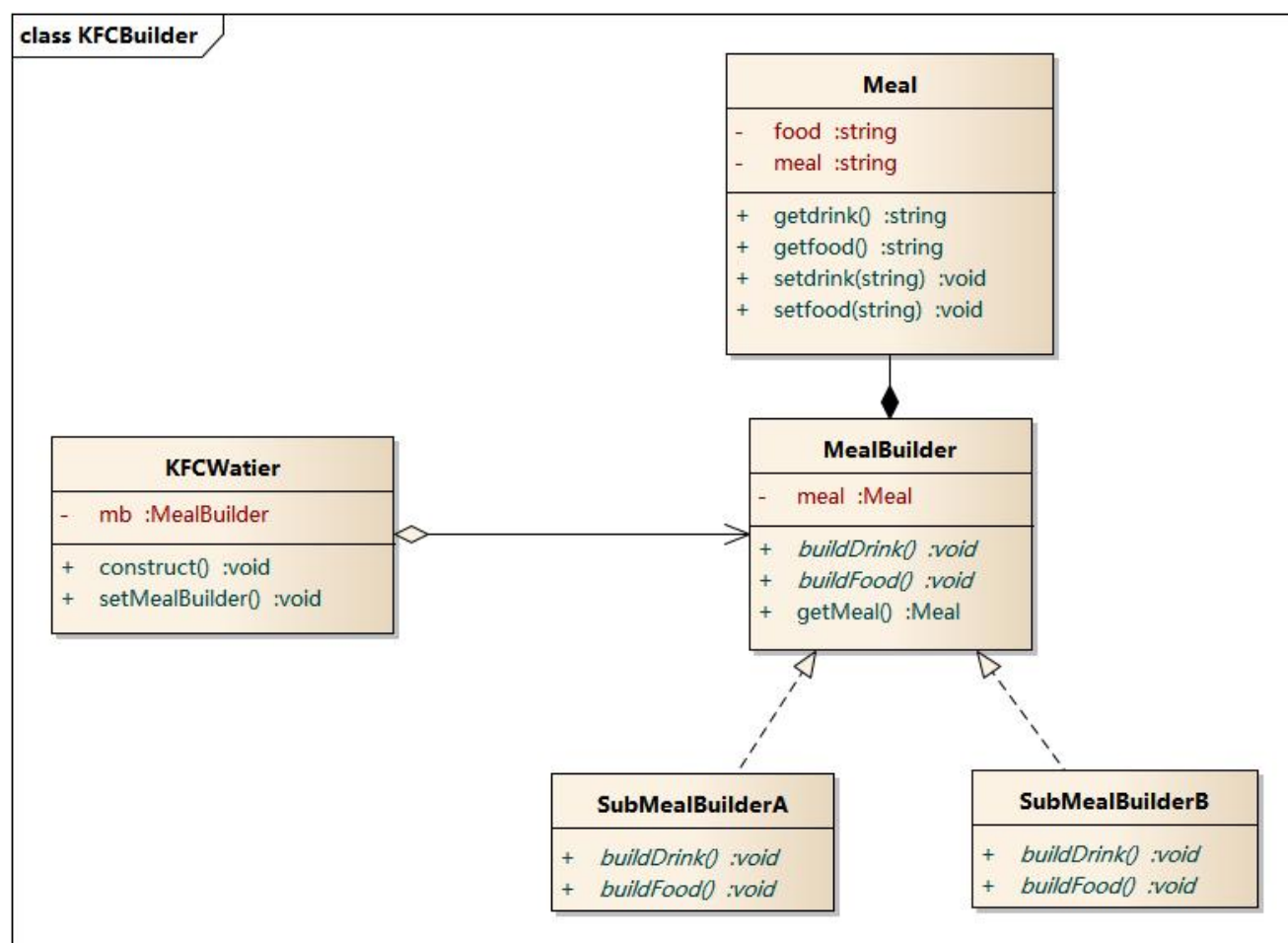
建造者模式的结构中还引入了一个指挥者类Director，该类的作用主要有两个：一方面它隔离了客户与生产过程；另一方面它负责控制产品的生成过程。指挥者针对抽象建造者编程，客户端只需要知道具体建造者的类型，即可通过指挥者类调用建造者的相关方法，返回一个完整的产品对象

在客户端代码中，无须关心产品对象的具体组装过程，只需确定具体建造者的类型即可，建造者模式将复杂对象的构建与对象的表现分离开来，这样使得同样的构建过程可以创建出不同的表现。

4.7. 实例

实例：KFC套餐

建造者模式可以用于描述KFC如何创建套餐：套餐是一个复杂对象，它一般包含主食（如汉堡、鸡肉卷等）和饮料（如果汁、可乐等）等组成部分，不同的套餐有不同的组成部分，而KFC的服务员可以根据顾客的要求，一步一步装配这些组成部分，构造一份完整的套餐，然后返回给顾客。



4.8. 优点

- 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。
- 每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，用户使用不同的具体建造者即可得到不同的产品对象。
- 可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。
- 增加新的具体建造者无须修改原有类库的代码，指挥者类针对抽象建造者类编程，系统扩展方便，符合“开闭原则”。

4.9. 缺点

- 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。
- 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。

4.10. 适用环境

在以下情况下可以使用建造者模式：

- 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性。
- 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。
- 对象的创建过程独立于创建该对象的类。在建造者模式中引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类中。
- 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品。

4.11. 模式应用

在很多游戏软件中，地图包括天空、地面、背景等组成部分，人物角色包括人体、服装、装备等组成部分，可以使用建造者模式对其进行设计，通过不同的具体建造者创建不同类型的地图或人物。

4.12. 模式扩展

建造者模式的简化：

- 省略抽象建造者角色：如果系统中只需要一个具体建造者的话，可以省略掉抽象建造者。
- 省略指挥者角色：在具体建造者只有一个的情况下，如果抽象建造者角色已经被省略掉，那么还可以省略指挥者角色，让

Builder角色扮演指挥者与建造者双重角色。

建造者模式与抽象工厂模式的比较：

- 与抽象工厂模式相比，建造者模式返回一个组装好的完整产品，而抽象工厂模式返回一系列相关的产品，这些产品位于不同的产品等级结构，构成了一个产品族。
- 在抽象工厂模式中，客户端实例化工厂类，然后调用工厂方法获取所需产品对象，而在建造者模式中，客户端可以不直接调用建造者的相关方法，而是通过指挥者类来指导如何生成对象，包括对象的组装过程和建造步骤，它侧重于一步步构造一个复杂对象，返回一个完整的对象。
- 如果将抽象工厂模式看成汽车配件生产工厂，生产一个产品族的产品，那么建造者模式就是一个汽车组装工厂，通过对部件的组装可以返回一辆完整的汽车。

4.13. 总结

- 建造者模式将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式。
- 建造者模式包含如下四个角色：抽象建造者为创建一个产品对象的各个部件指定抽象接口；具体建造者实现了抽象建造者接口，实现各个部件的构造和装配方法，定义并明确它所创建的复杂对象，也可以提供一个方法返回创建好的复杂产品对象；产品角色是被构建的复杂对象，包含多个组成部件；指挥者负责安排复杂对象的建造次序，指挥者与抽象建造者之间存在关联关系，可以在其construct()建造方法中调用建造者对象的部件构造与装配方法，完成复杂对象的建造
- 在建造者模式的结构中引入了一个指挥者类，该类的作用主要有两个：一方面它隔离了客户与生产过程；另一方面它负责控制产品的生成过程。指挥者针对抽象建造者编程，客户端只需要知道具体建造者的类型，即可通过指挥者类调用建造者的相关方法，返回一个完整的产品对象。
- 建造者模式的主要优点在于客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象，每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，符合“开闭原则”，还可以更加精细地控制产品的创建过程；其主要缺点在于由于建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，因此其使用范围受到一定的限制，如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。
- 建造者模式适用情况包括：需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性；需要生成的产品对象的属性相互依赖，需要指定其生成顺序；对象的创建过程独立于创建该对象的类；隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同类型的产品。

5. 单例模式

目录

- 单例模式
 - 模式动机
 - 模式定义
 - 模式结构
 - 时序图
 - 代码分析
 - 模式分析
 - 实例
 - 优点
 - 缺点
 - 适用环境
 - 模式应用
 - 模式扩展
 - 总结

5.1. 模式动机

对于系统中的某些类来说，只有一个实例很重要，例如，一个系统中可以存在多个打印任务，但是只能有一个正在工作的任务；一个系统只能有一个窗口管理器或文件系统；一个系统只能有一个计时工具或ID（序号）生成器。

如何保证一个类只有一个实例并且这个实例易于被访问呢？定义一个全局变量可以确保对象随时都可以被访问，但不能防止我们实例化多个对象。

一个更好的解决办法是让类自身负责保存它的唯一实例。这个类可以保证没有其他实例被创建，并且它可以提供一个访问该实例的方法。这就是单例模式的模式动机。

5.2. 模式定义

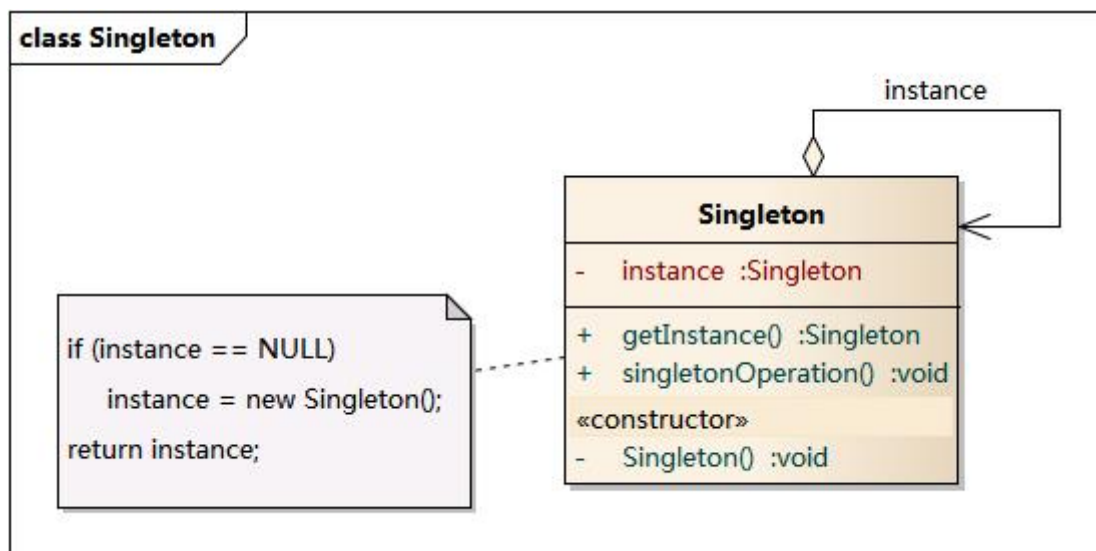
单例模式(Singleton Pattern)：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。

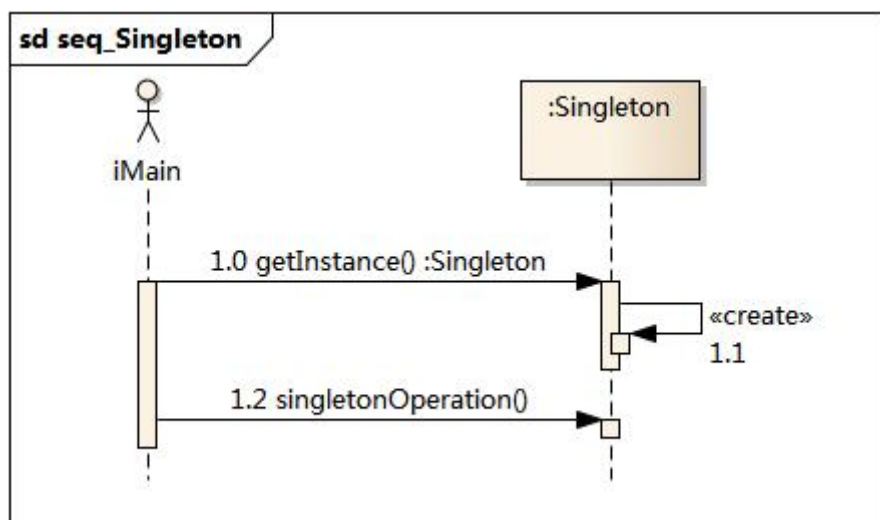
5.3. 模式结构

单例模式包含如下角色：

- Singleton：单例



5.4. 时序图



5.5. 代码分析

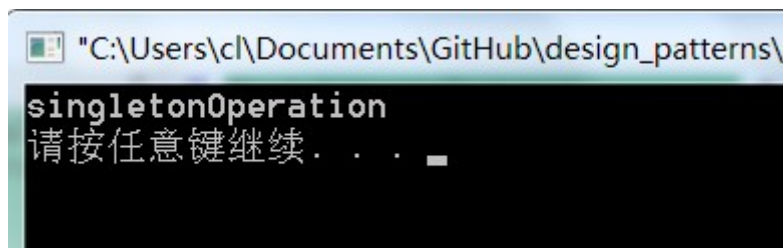
```
1  #include <iostream>
2  #include "Singleton.h"
3  using namespace std;
4  int main(int argc, char *argv[])
5  {
6      Singleton * sg = Singleton::getInstance();
7      sg->singletonOperation();
8      return 0;
9  }
10
11
```

```

1  //////////////////////////////////////
2  // Singleton.cpp
3  // Implementation of the Class Singleton
4  // Created on:      02-十月-2014 17:24:46
5  // Original author: colin
6  //////////////////////////////////////
7  #include "Singleton.h"
8  #include <iostream>
9  using namespace std;
10 Singleton * Singleton::instance = NULL;
11 Singleton::Singleton(){
12 }
13 Singleton::~Singleton(){
14     delete instance;
15 }
16 Singleton* Singleton::getInstance(){
17     if (instance == NULL)
18     {
19         instance = new Singleton();
20     }
21     return instance;
22 }
23
24 void Singleton::singletonOperation(){
25     cout << "singletonOperation" << endl;
26 }
27
28
29
30
31
32
33

```

运行结果：



5.6. 模式分析

单例模式的目的是保证一个类仅有一个实例，并提供一个访问它的全局访问点。单例模式包含的角色只有一个，就是单例类——Singleton。单例类拥有一个私有构造函数，确保用户无法通过new关键字直接实例化它。除此之外，该模式中包含一个静态私有成员变量与静态公有的工厂方法，该工厂方法负责检验实例的存在性并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。

在单例模式的实现过程中，需要注意如下三点：

- 单例类的构造函数为私有；
- 提供一个自身的静态私有成员变量；
- 提供一个公有的静态工厂方法。

5.7. 实例

在操作系统中，打印池(Print Spooler)是一个用于管理打印任务的应用程序，通过打印池用户可以删除、中止或者改变打印任务的优先级，在一个系统中只允许运行一个打印池对象，如果重复创建打印池则抛出异常。现使用单例模式来模拟实现打印池的设计。

5.8. 优点

- 提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它，并为设计及开发团队提供了共享的概念。
- 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象，单例模式无疑可以提高系统的性能。
- 允许可变数目的实例。我们可以基于单例模式进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例。

5.9. 缺点

- 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。
- 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品的本身的功能融合到一起。
- 滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；现在很多面向对象语言(如Java、C#)的运行环境都提供了自动垃圾回收的技术，因此，如果实例化的对象长时间不被利用，系统会认为它是垃圾，会自动销毁并回收资源，下次利用时又将重新实例化，这将导致对象状态的丢失。

5.10. 适用环境

在以下情况下可以使用单例模式：

- 系统只需要一个实例对象，如系统要求提供一个唯一的序列号生成器，或者需要考虑资源消耗太大而只允许创建一个对象。
- 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例。
- 在一个系统中要求一个类只有一个实例时才应当使用单例模式。反过来，如果一个类可以有几个实例共存，就需要对单例模式进行改进，使之成为多例模式

5.11. 模式应用

一个具有自动编号主键的表可以有多个用户同时使用，但数据库中只能有一个地方分配下一个主键编号，否则会出现主键重复，因此该主键编号生成器必须具备唯一性，可以通过单例模式来实现。

5.12. 模式扩展

5.13. 总结

- 单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。
- 单例模式只包含一个单例角色：在单例类的内部实现只生成一个实例，同时它提供一个静态的工厂方法，让客户可以使用它的唯一实例；为了防止在外部对其实例化，将其构造函数设计为私有。
- 单例模式的目的是保证一个类仅有一个实例，并提供一个访问它的全局访问点。单例类拥有一个私有构造函数，确保用户无法通过new关键字直接实例化它。除此之外，该模式中包含一个静态私有成员变量与静态公有的工厂方法。该工厂方法负责检验实例的存在性并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。
- 单例模式的主要优点在于提供了对唯一实例的受控访问并可以节约系统资源；其主要缺点在于因为缺少抽象层而难以扩展，且单例类职责过重。
- 单例模式适用情况包括：系统只需要一个实例对象；客户调用类的单个实例只允许使用一个公共访问点。