

图说设计模式



软件模式是将模式的一般概念应用于软件开发领域，即软件开发的总体思路或参照样板。软件模式并非仅限于设计模式，还包括架构模式、分析模式和过程模式等，实际上，在软件生存期的每一个阶段都存在着一些...



下载手机APP
畅享精彩阅读

目 录

致谢

介绍

看懂UML类图和时序图

创建型模式

1. 简单工厂模式(Simple Factory Pattern)
2. 工厂方法模式(Factory Method Pattern)
3. 抽象工厂模式(Abstract Factory)
4. 建造者模式
5. 单例模式

结构型模式

1. 适配器模式
2. 桥接模式
3. 装饰模式
4. 外观模式
5. 享元模式
6. 代理模式

行为型模式

1. 命令模式
2. 中介者模式
3. 观察者模式
4. 状态模式
5. 策略模式

致谢

当前文档《图说设计模式》由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建,生成于 2019-11-25。

书栈网仅提供文档编写、整理、归类等功能,以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理,书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候,发现文档内容有不恰当的地方,请向我们反馈,让我们共同携手,将知识准确、高效且有效地传递给每一个人。

同时,如果您在日常工作、生活和学习中遇到有价值有营养的知识文档,欢迎分享到书栈网,为知识的传承献上您的一份力量!

如果当前文档生成时间太久,请到书栈网获取最新的文档,以跟上知识更新换代的步伐。

内容来源: [me115](https://github.com/me115/design_patterns) https://github.com/me115/design_patterns

文档地址: <http://www.bookstack.cn/books/design-patterns>

书栈官网: <https://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享,让知识传承更久远! 感谢知识的创造者,感谢知识的分享者,也感谢每一位阅读到此处的读者,因为我们都将成为知识的传承者。

图说设计模式

软件模式是将模式的一般概念应用于软件开发领域，即软件开发的总体指导思路或参照样板。软件模式并非仅限于设计模式，还包括架构模式、分析模式和过程模式等，实际上，在软件生存期的每一个阶段都存在着一些被认同的模式。

本书使用图形和代码结合的方式来解析设计模式；

每个模式都有相应的对象结构图，同时为了展示对象间的交互细节， 我会用到时序图来介绍其如何运行；（在状态模式中， 还会用到状态图，这种图的使用对于理解状态的转换非常直观）

为了让大家能读懂UML图，在最前面会有一篇文章来介绍UML图形符号；

在系统的学习设计模式之后，我们需要达到3个层次：

- 能在白纸上画出所有的模式结构和时序图；
- 能用代码实现；如果模式的代码都没有实现过，是用不出来的；即所谓，看得懂，不会用；
- 灵活应用到工作中的项目中；
- [看懂UML类图和时序图](#)
 - [从一个示例开始](#)
 - [类之间的关系](#)
 - [时序图](#)
 - [附录](#)
- [创建型模式](#)
 - [1. 简单工厂模式\(Simple Factory Pattern \)](#)
 - [2. 工厂方法模式\(Factory Method Pattern\)](#)
 - [3. 抽象工厂模式\(Abstract Factory\)](#)
 - [4. 建造者模式](#)
 - [5. 单例模式](#)
- [结构型模式](#)
 - [1. 适配器模式](#)
 - [2. 桥接模式](#)
 - [3. 装饰模式](#)
 - [4. 外观模式](#)
 - [5. 享元模式](#)
 - [6. 代理模式](#)
- [行为型模式](#)
 - [1. 命令模式](#)
 - [2. 中介者模式](#)
 - [3. 观察者模式](#)
 - [4. 状态模式](#)
 - [5. 策略模式](#)

Indices and tables

- [索引](#)
- [搜索页面](#)

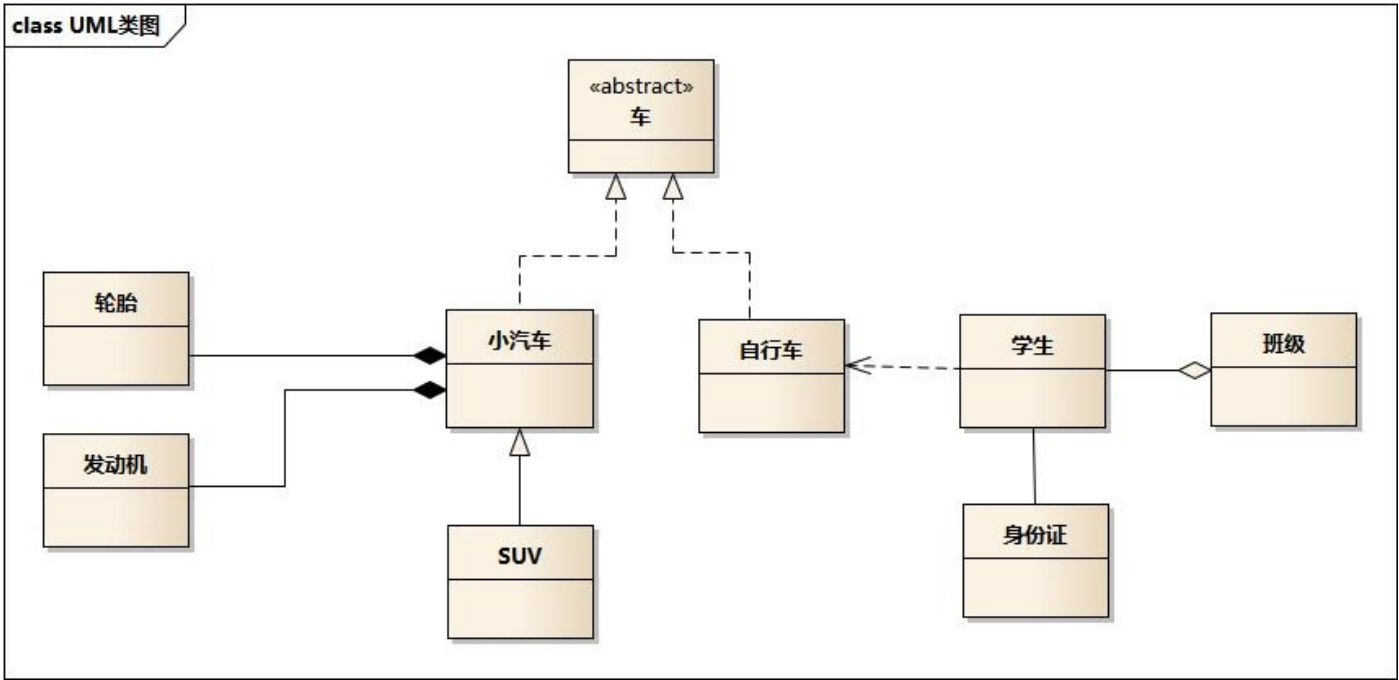
看懂UML类图和时序图

这里不会将UML的各种元素都提到，我只想讲讲类图中各个类之间的关系；能看懂类图中各个类之间的线条、箭头代表什么意思后，也就足够应对日常的工作和交流；同时，我们应该能将类图所表达的含义和最终的代码对应起来；有了这些知识，看后面章节的设计模式结构图就没有什么问题了；

本章所有图形使用Enterprise Architect 9.2来画,所有示例详见根目录下的design_patterns.EAP

从一个示例开始

请看以下这个类图，类之间的关系是我们需要关注的：



- 车的类图结构为<>，表示车是一个抽象类；
- 它有两个继承类：小汽车和自行车；它们之间的关系为实现关系，使用带空心箭头的虚线表示；
- 小汽车为与SUV之间也是继承关系，它们之间的关系为泛化关系，使用带空心箭头的实线表示；
- 小汽车与发动机之间是组合关系，使用带实心箭头的实线表示；
- 学生与班级之间是聚合关系，使用带空心箭头的实线表示；
- 学生与身份证之间为关联关系，使用一根实线表示；
- 学生上学需要用到自行车，与自行车是一种依赖关系，使用带箭头的虚线表示；下面我们将介绍这六种关系；

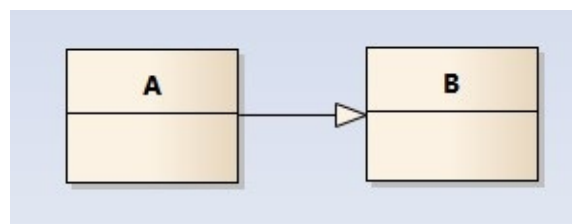
类之间的关系

泛化关系 (generalization)

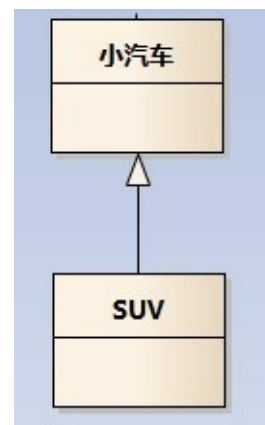
类的继承结构表现在UML中为：泛化 (generalize) 与实现 (realize)：

继承关系为 is-a 的关系；两个对象之间如果可以用 is-a 来表示，就是继承关系：（..是..）

eg: 自行车是车、猫是动物



泛化关系用一条带空心箭头的直接表示；如下图表示（A继承自B）；



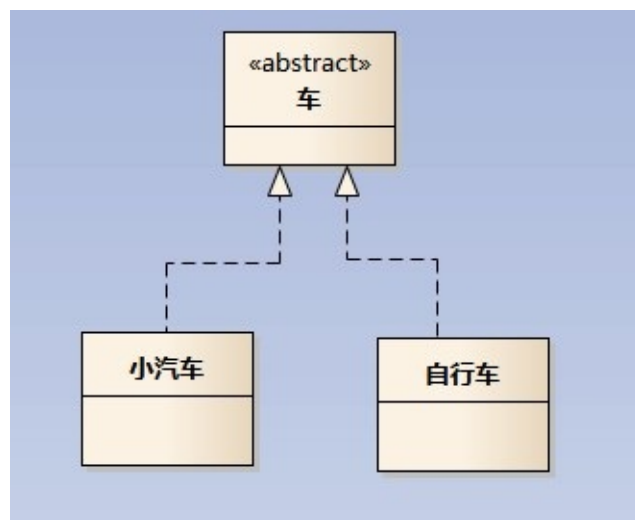
eg: 汽车在现实中有实现，可用汽车定义具体的对象；汽车与SUV之间为泛化关系；
码中，泛化关系表现为继承非抽象类；

注：最终代

实现关系(realize)

实现关系用一条带空心箭头的虚线表示；

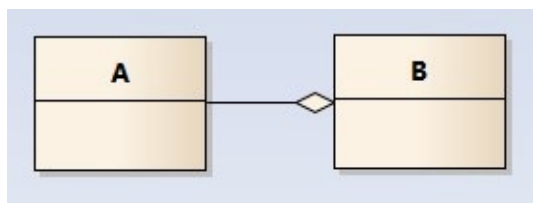
eg: "车"为一个抽象概念，在现实中并无法直接用来定义对象；只有指明具体的子类(汽车还是自行车)，才可以用来定义对象（"车"这个类在C++中用抽象类表示，在JAVA中有接口这个概念，更容易理解）



注：最终代码中，实现关系表现为继承抽象类；

聚合关系(aggregation)

聚合关系用一条带空心菱形箭头的直线表示，如下图表示A聚合到B上，或者说B由A组成；



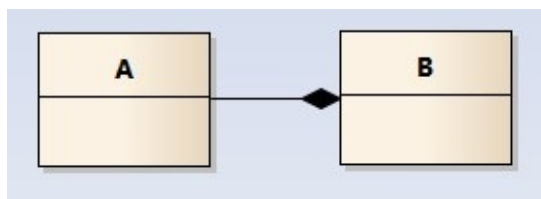
聚合关系用于表示实体对象之间的关系，表示整体由部分构成的语义；例

如一个部门由多个员工组成；

与组合关系不同的是，整体和部分不是强依赖的，即使整体不存在了，部分仍然存在；例如，部门撤销了，人员不会消失，他们依然存在；

组合关系 (composition)

组合关系用一条带实心菱形箭头直线表示，如下图表示A组成B，或者B由A组成；



与聚合关系一样，组合关系同样表示整体由部分构成的语义；比如公司

由多个部门组成；

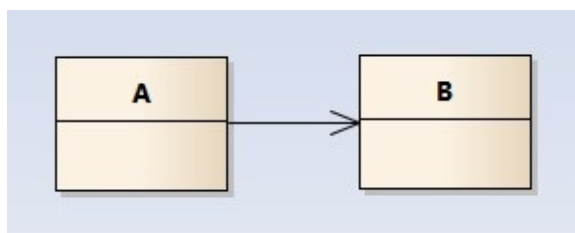
但组合关系是一种强依赖的特殊聚合关系，如果整体不存在了，则部分也不存在了；例如，公司不存在了，部门也将不存在了；

关联关系 (association)

关联关系是用一条直线表示的；它描述不同类的对象之间的结构关系；它是一种静态关系，通常与运行状态无关，一般由常识等因素决定的；它一般用来定义对象之间静态的、天然的结构；所以，关联关系是一种“强关联”的关系；

比如，乘车人和车票之间就是一种关联关系；学生和学校就是一种关联关系；

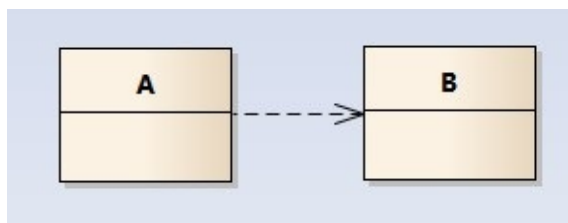
关联关系默认不强调方向，表示对象间相互知道；如果特别强调方向，如下图，表示A知道B，但B不知道A；



注：在最终代码中，关联对象通常是以成员变量的形式实现的；

依赖关系 (dependency)

依赖关系是用一套带箭头的虚线表示的；如下图表示A依赖于B；他描述一个对象在运行期间会用到另一个对象的关



系；与关联关系不同的是，它是一种临时性的关系，通常在运行期间产生，并且随着运行时的变化；依赖关系也可能发生变化；

显然，依赖也有方向，双向依赖是一种非常糟糕的结构，我们总是应该保持单向依赖，杜绝双向依赖的产生；

注：在最终代码中，依赖关系体现为类构造方法及类方法的传入参数，箭头的指向为调用关系；依赖关系除了临时知道对方外，还是“使用”对方的方法和属性；

时序图

为了展示对象之间的交互细节，后续对设计模式解析的章节，都会用到时序图；

时序图（Sequence Diagram）是显示对象之间交互的图，这些对象是按时间顺序排列的。时序图中显示的是参与交互的对象及其对象之间消息交互的顺序。

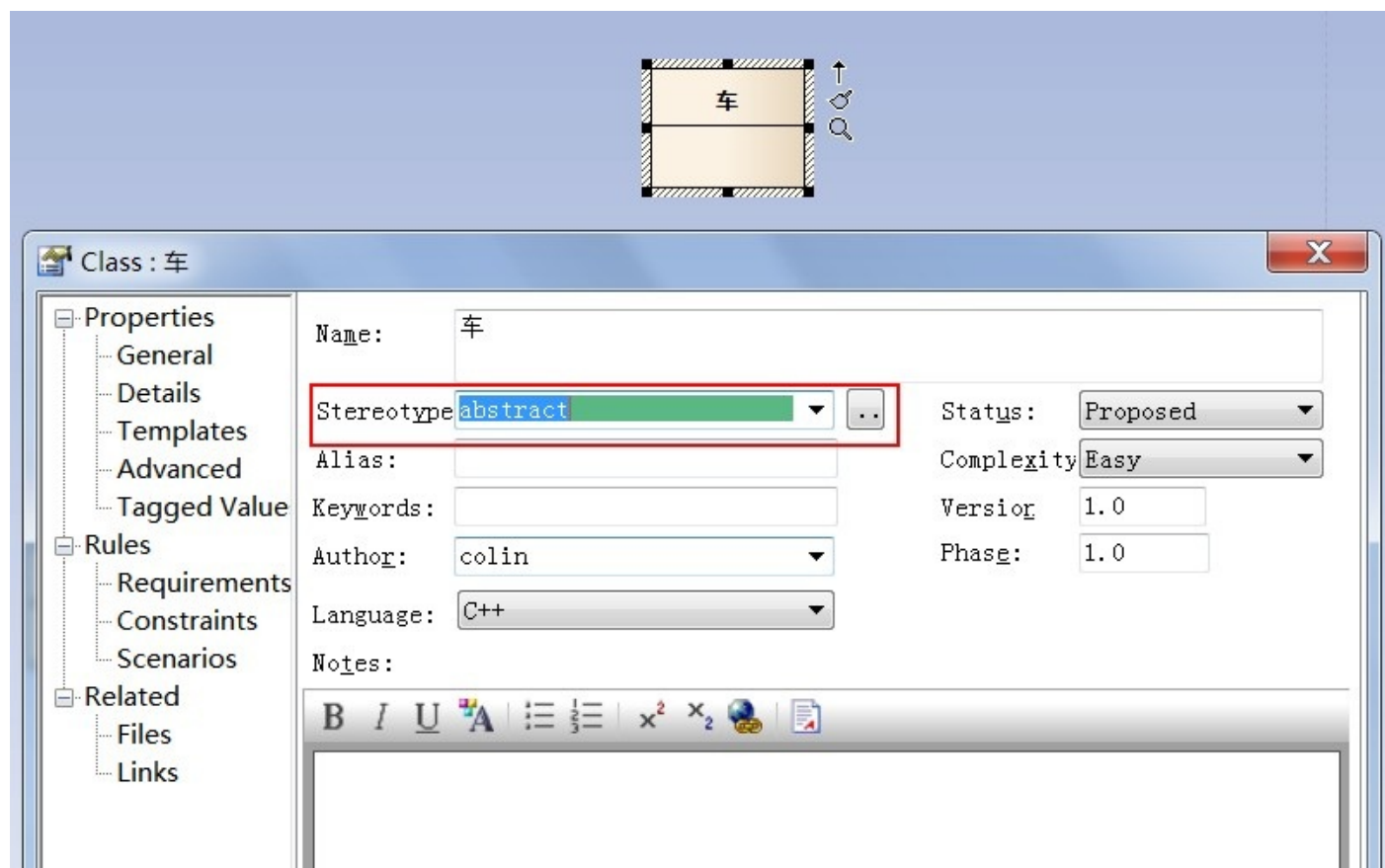
时序图包括的建模元素主要有：对象（Actor）、生命线（Lifeline）、控制焦点（Focus of control）、消息（Message）等等。

关于时序图，以下这篇文章将概念介绍的比较详细；更多实例应用，参见后续章节模式中的时序图；

<http://smartlife.blog.51cto.com/1146871/284874>

附录

在EA中定义一个抽象类（其版型为《abstract》）



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
- 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)

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。

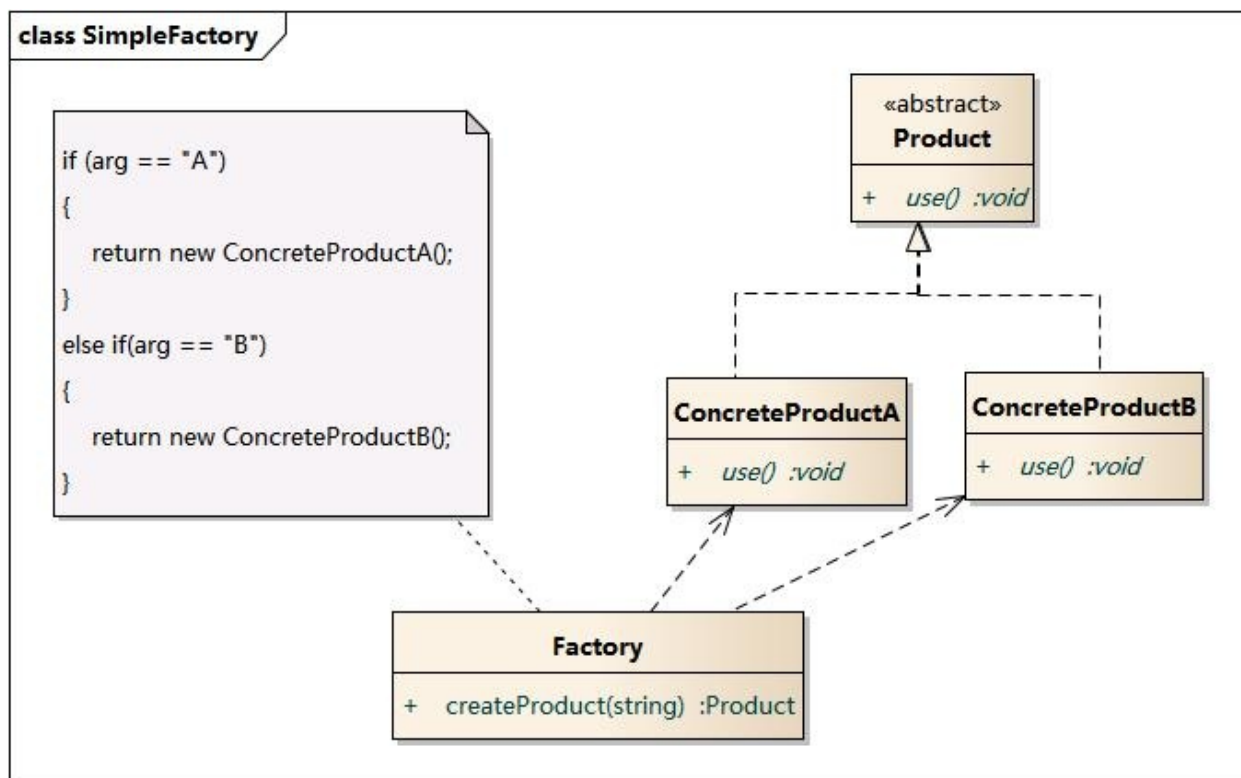
1.2. 模式定义

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。

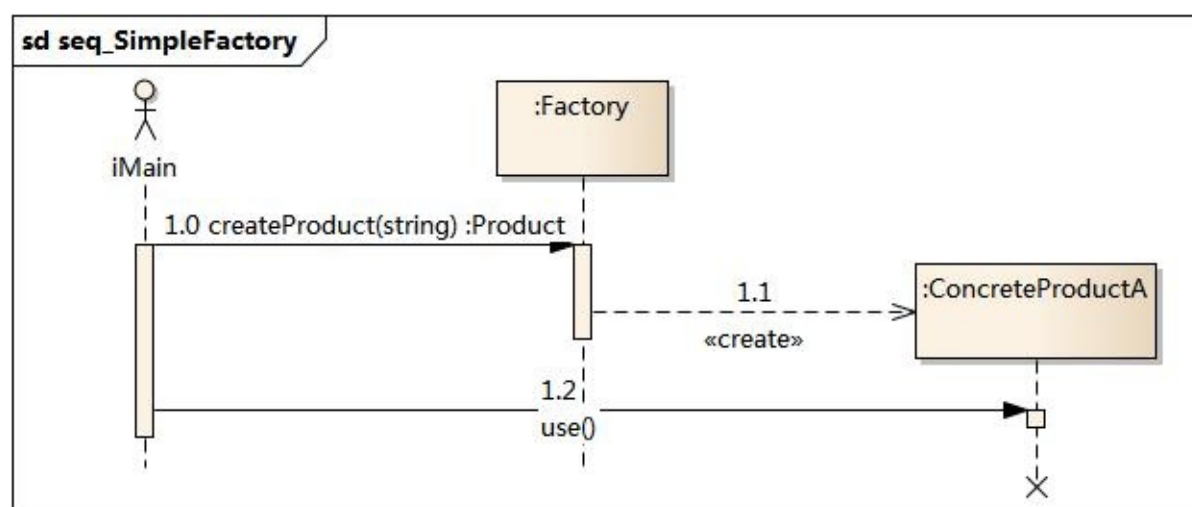
1.3. 模式结构

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 pronomame){
12.     if ( "A" == pronomame )
13.     {
14.         return new ConcreteProductA();
15.     }
16.     else if("B" == pronomame)
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，而无需知道其创建细节。

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

- 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。

```
1. public final static DateFormat getDateInstance();
2. public final static DateFormat getDateInstance(int style);
3. public final static DateFormat getDateInstance(int style,Locale
4. locale);
```

- Java加密技术 获取不同加密算法的密钥生成器：

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

创建密码器：

```
1. 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)

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”。

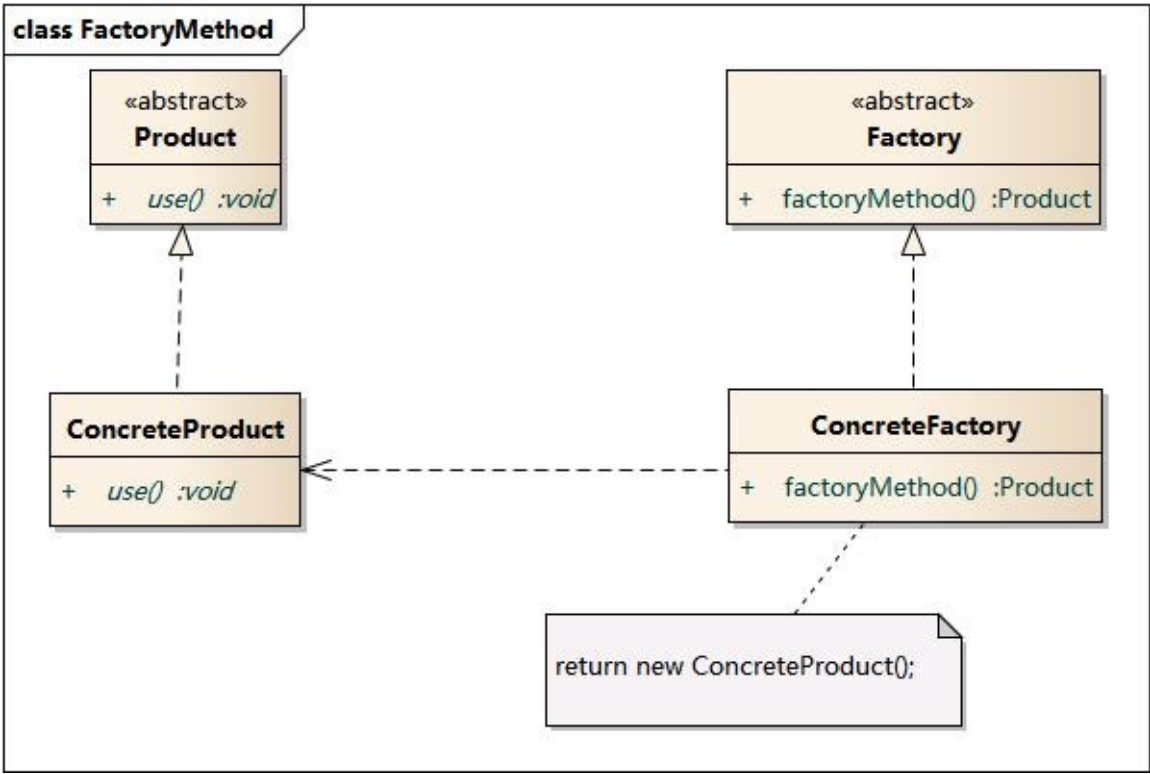
2.2. 模式定义

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。

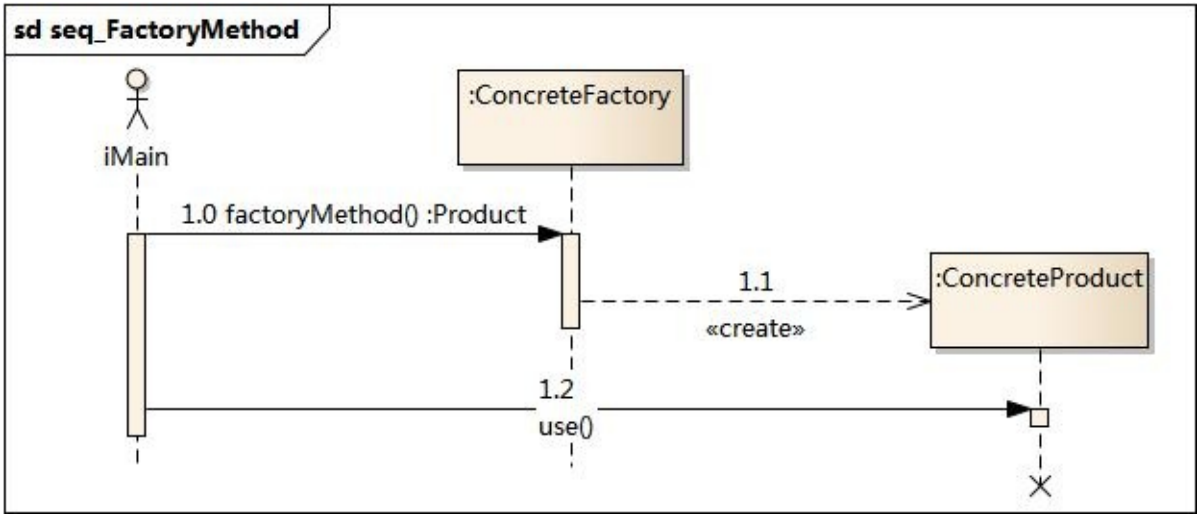
2.3. 模式结构

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.
8. #include "ConcreteFactory.h"
9. #include "ConcreteProduct.h"
```

```
10.  
11. Product* ConcreteFactory::factoryMethod(){  
12.  
13.     return new ConcreteProduct();  
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

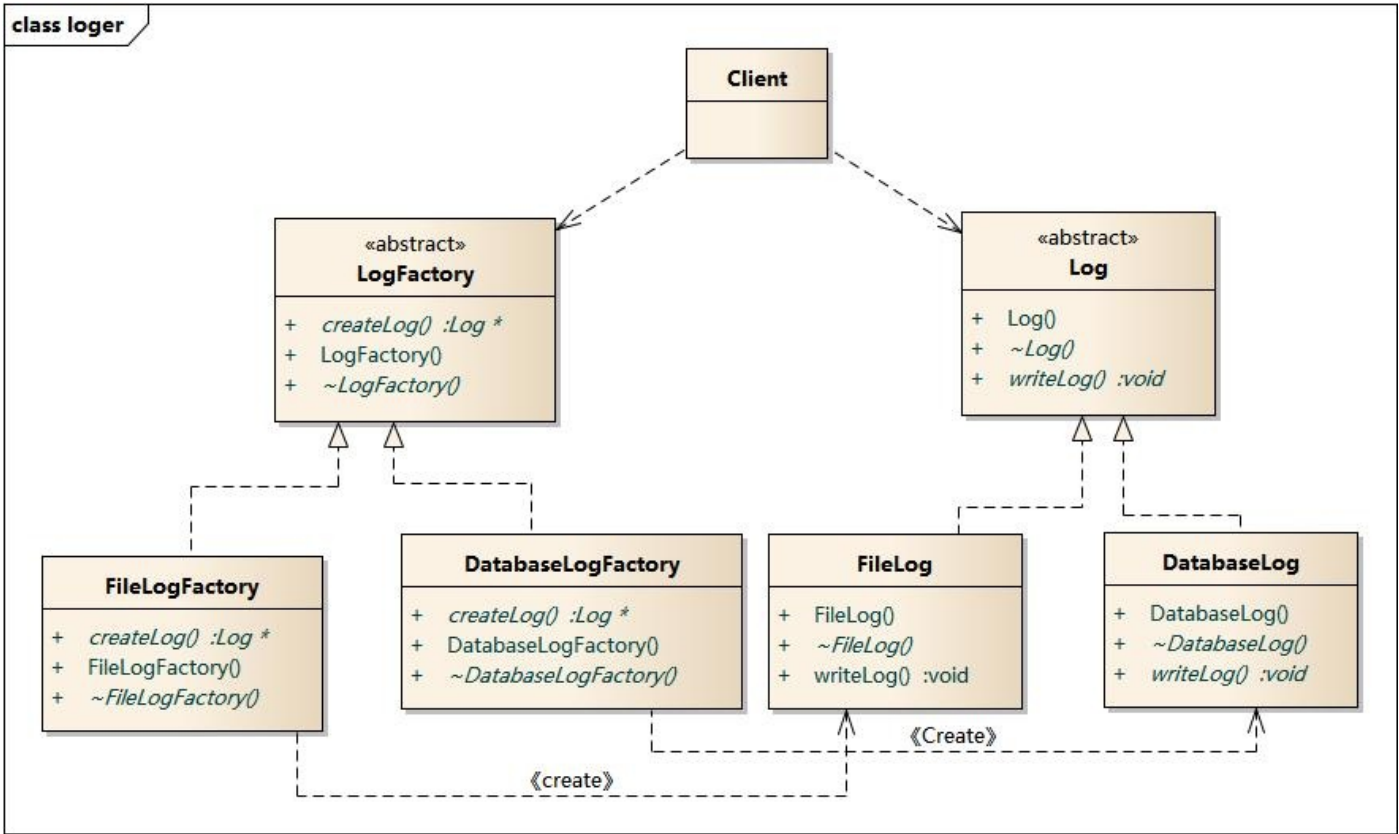
工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责哪一个产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

2.7. 实例

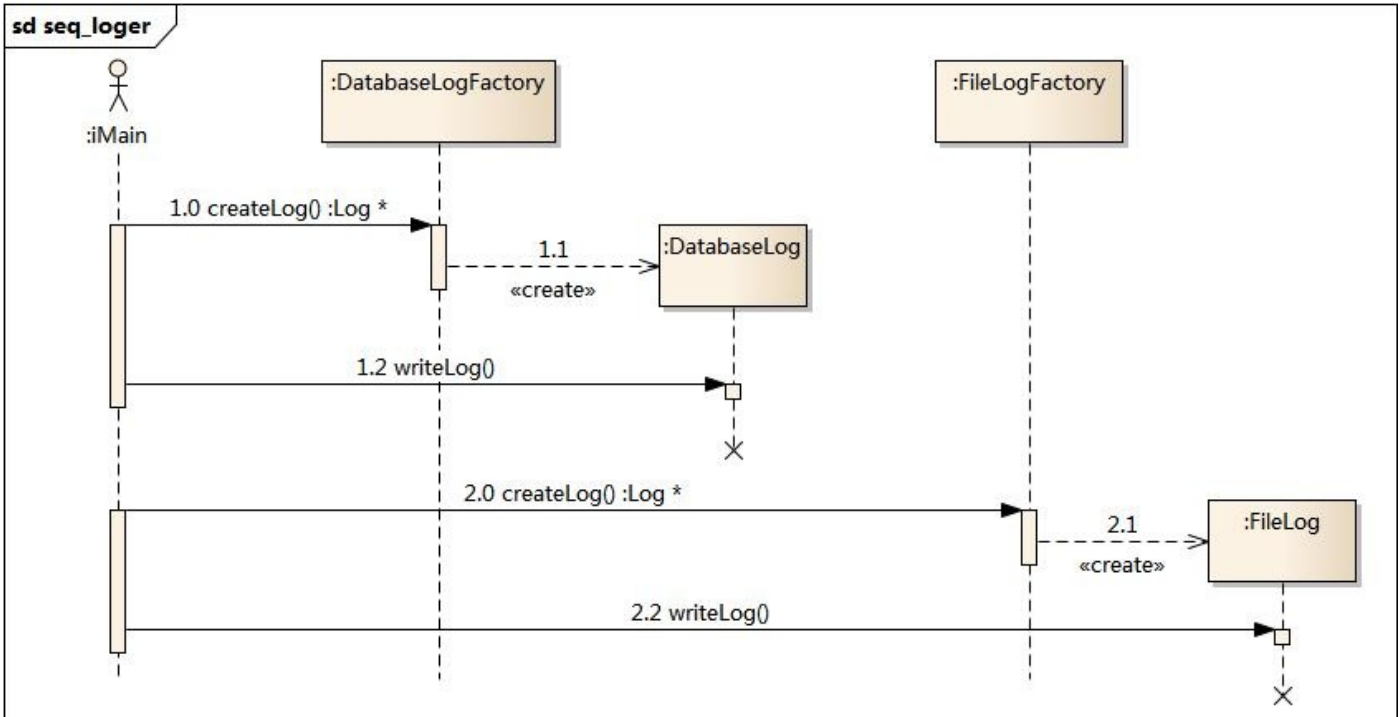
日志记录器

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：

```
1. Connection conn=DriverManager.getConnection("jdbc:microsoft:sqlserver://loc
2. alhost:1433; DatabaseName=DB;user=sa;password=");
3. Statement statement=conn.createStatement();
4. 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)

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产品对象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。

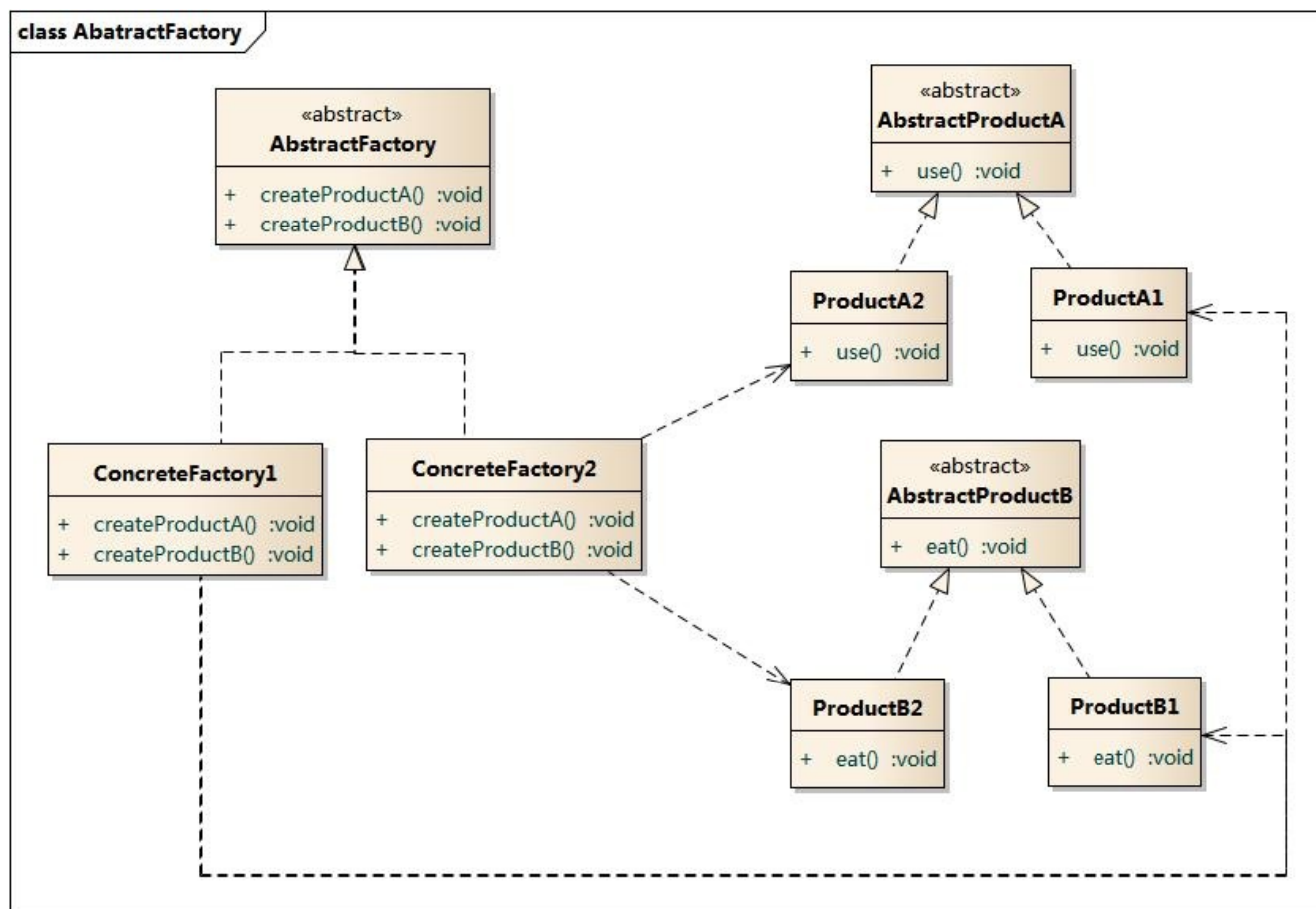
3.2. 模式定义

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。

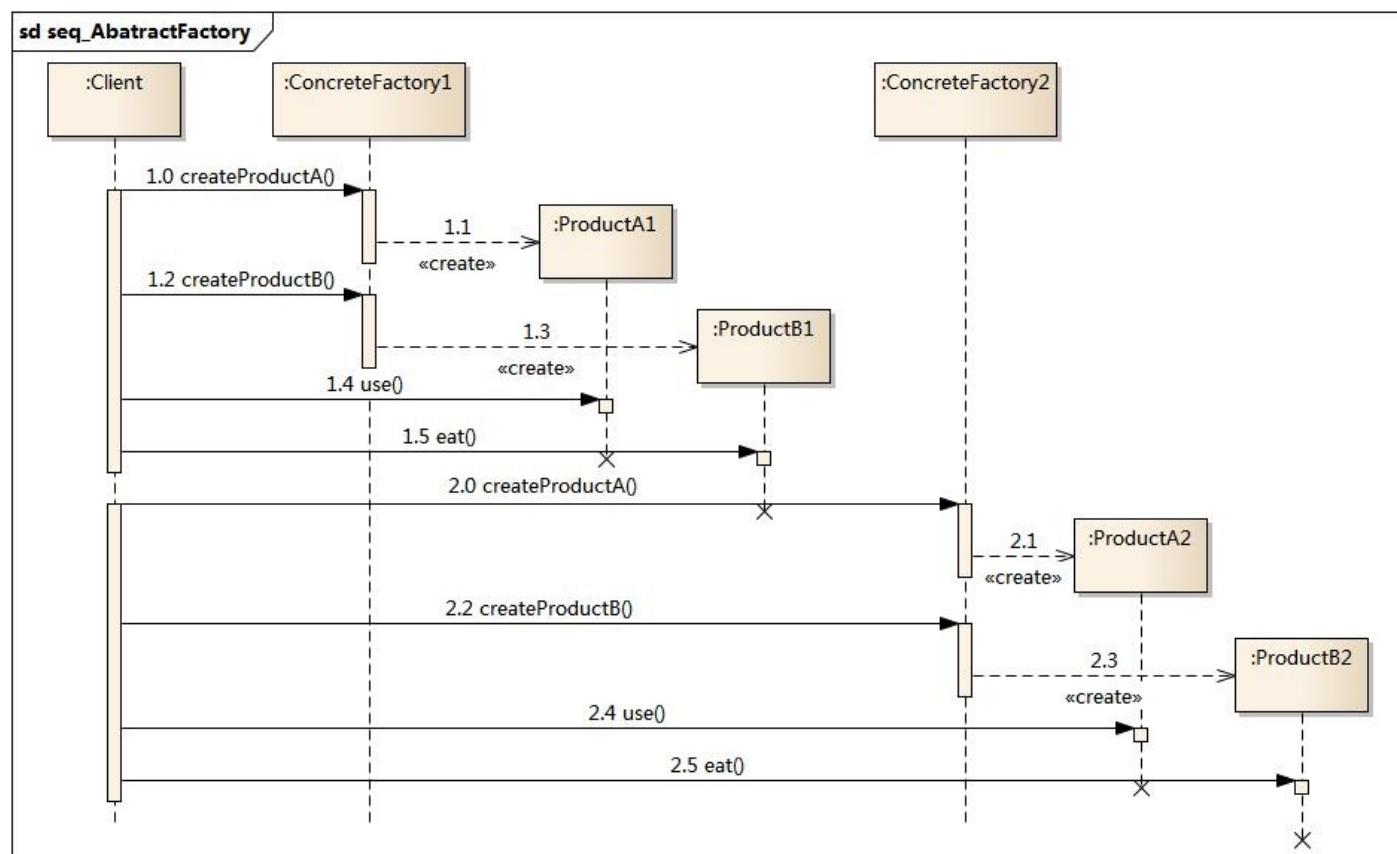
3.3. 模式结构

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.
8. #include "ConcreteFactory1.h"
9. #include "ProductA1.h"
10. #include "ProductB1.h"
11. AbstractProductA * ConcreteFactory1::createProductA(){
12.     return new ProductA1();
13. }
14.
15.
16. AbstractProductB * ConcreteFactory1::createProductB(){
17.     return new ProductB1();
18. }

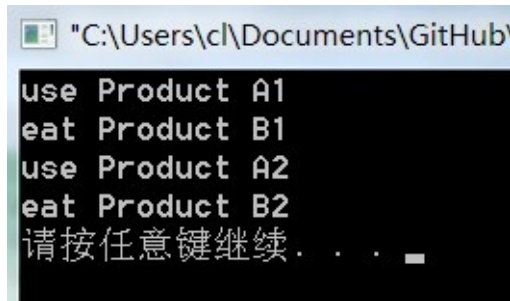
```

```

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.

```

```
8. #include "ProductA1.h"
9. #include <iostream>
10. using namespace std;
11. void ProductA1::use(){
12.     cout << "use Product A1" << endl;
13. }
```



运行结果：

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：
 - 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。
 - 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

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

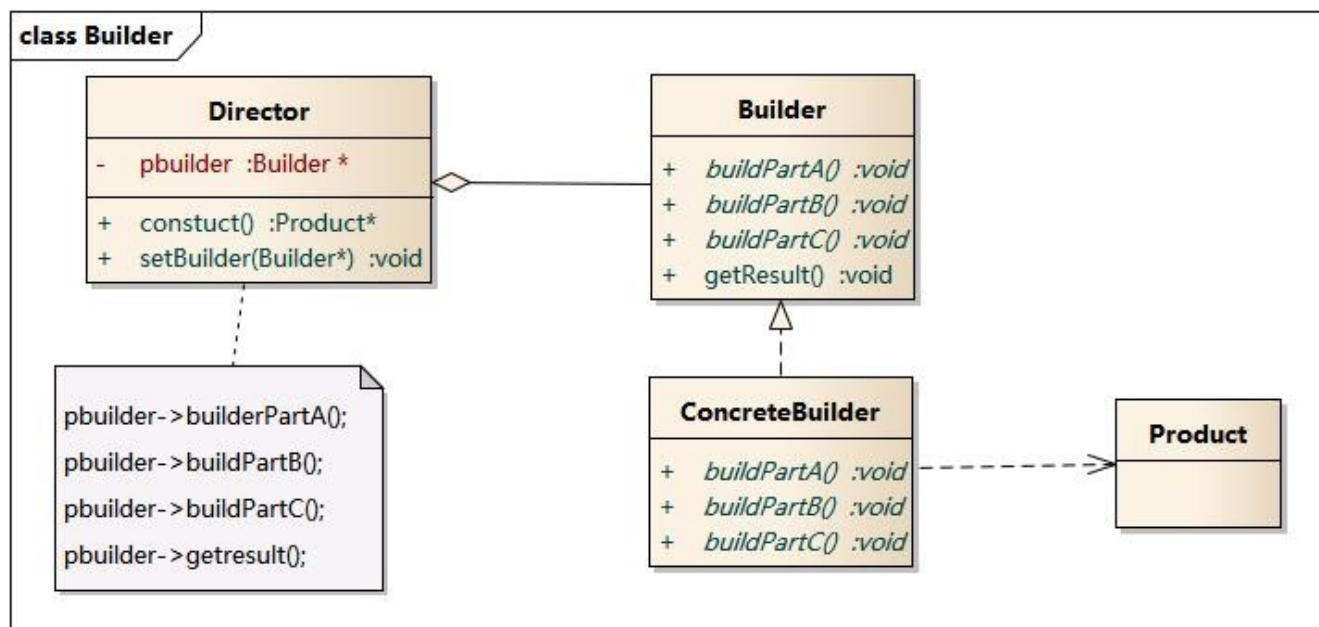
建造者模式(Builder Pattern)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

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。

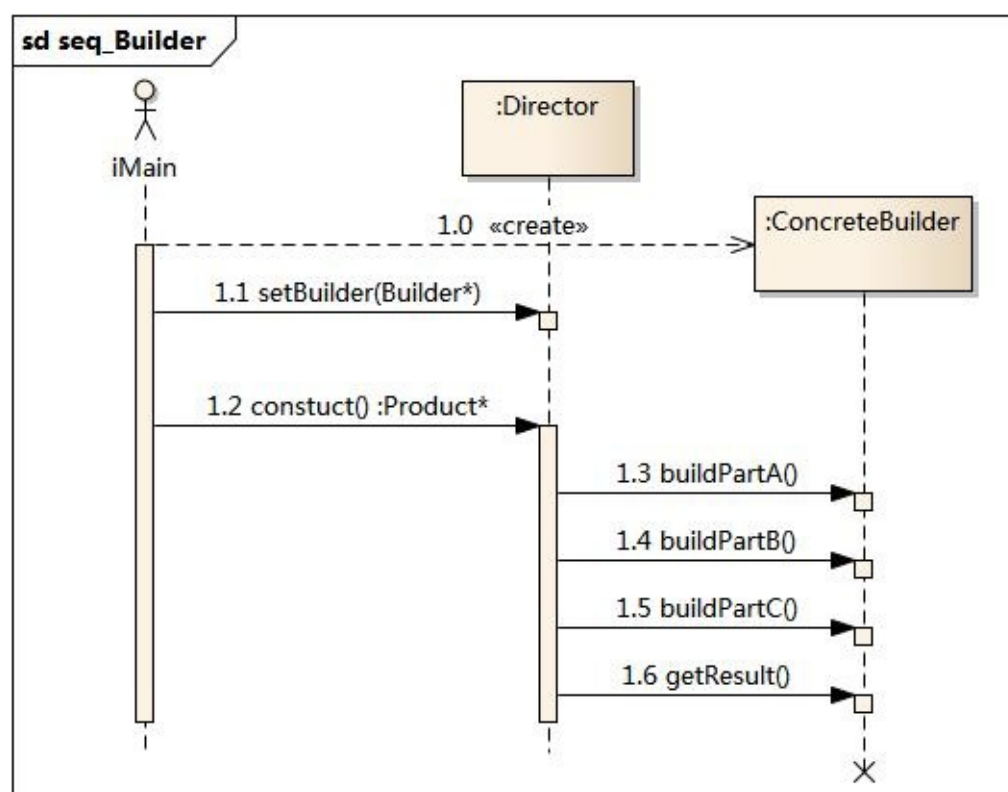
4.3. 模式结构

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.

```

4. 建造者模式

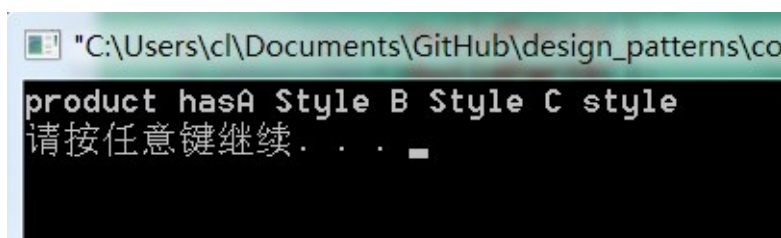
```
7. using namespace std;
8.
9. int main(int argc, char *argv[])
10. {
11.     ConcreteBuilder * builder = new ConcreteBuilder();
12.     Director director;
13.     director.setBuilder(builder);
14.     Product * pd = director.constuct();
15.     pd->show();
16.
17.     delete builder;
18.     delete 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.
8. #include "ConcreteBuilder.h"
9.
10.
11. ConcreteBuilder::ConcreteBuilder(){
12.
13. }
14.
15.
16.
17. ConcreteBuilder::~~ConcreteBuilder(){
18.
19. }
20.
21. void ConcreteBuilder::buildPartA(){
22.     m_prod->setA("A Style "); //不同的建造者，可以实现不同产品的建造
23. }
24.
25.
26. void ConcreteBuilder::buildPartB(){
27.     m_prod->setB("B Style ");
28. }
29.
30.
31. void ConcreteBuilder::buildPartC(){
32.     m_prod->setC("C style ");
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.
25. void Director::setBuilder(Builder* buider){
26.     m_pbuilder = buider;
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

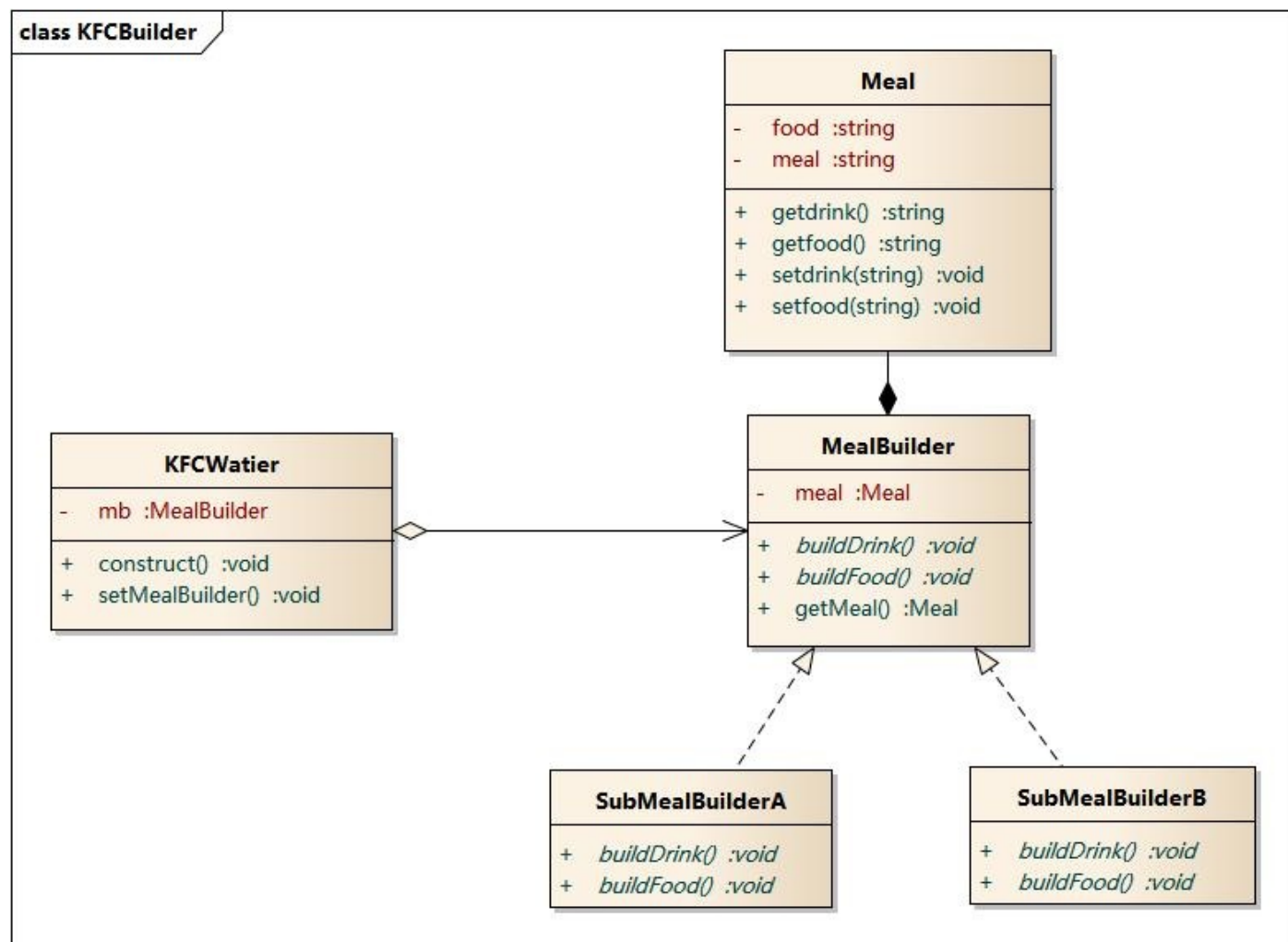
在客户端代码中，无须关心产品对象的具体组装过程，只需确定具体建造者的类型即可，建造者模式将复杂对象的构建与对象的表现分离开来，这样使得同样的构建过程可以创建出不同的表现。

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

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。

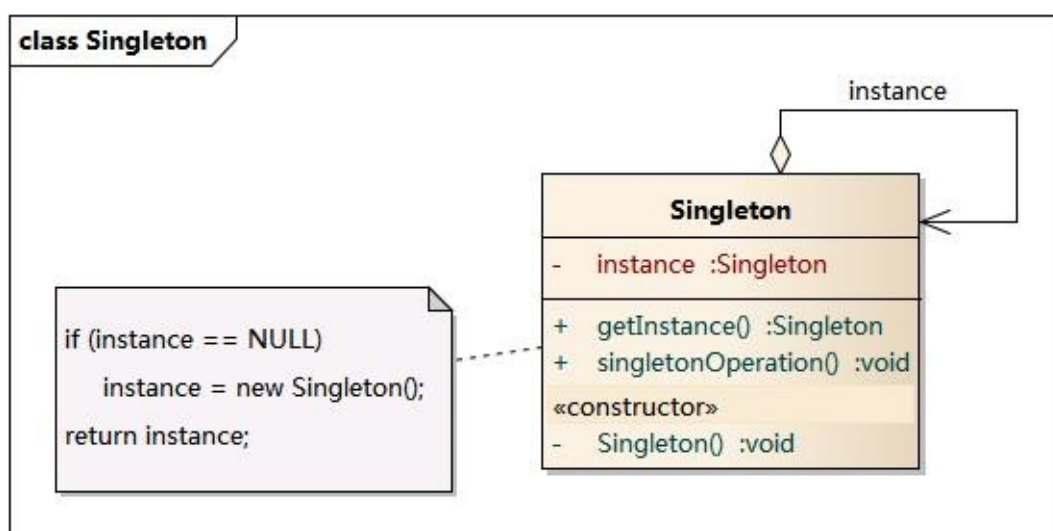
5.2. 模式定义

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。

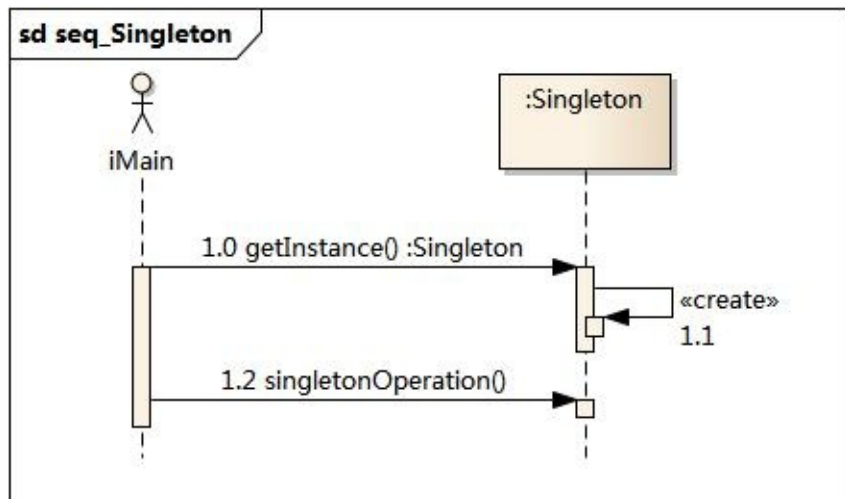
5.3. 模式结构

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.
5. int main(int argc, char *argv[])
6. {
7.     Singleton * sg = Singleton::getInstance();
8.     sg->singletonOperation();
9.
10.    return 0;
11. }

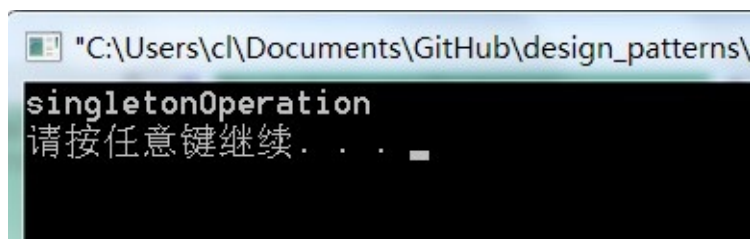
```

```

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.
8. #include "Singleton.h"
9. #include <iostream>
10. using namespace std;
11.
12. Singleton * Singleton::instance = NULL;
13. Singleton::Singleton(){
14.
15. }
16.
17. Singleton::~Singleton(){
18.     delete instance;
19. }
20.
21. Singleton* Singleton::getInstance(){
22.     if (instance == NULL)

```

```
23.     {  
24.         instance = new Singleton();  
25.     }  
26.  
27.     return instance;  
28. }  
29.  
30.  
31. void Singleton::singletonOperation(){  
32.     cout << "singletonOperation" << endl;  
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。

结构型模式

结构型模式(Structural Pattern)描述如何将类或者对象结合在一起形成更大的结构，就像搭积木，可以通过简单积木的组合形成复杂的、功能更为强大的结构。

结构型模式可以分为类结构型模式和对象结构型模式：

- 类结构型模式关心类的组合，由多个类可以组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系。
- 对象结构型模式关心类与对象的组合，通过关联关系使得在一个类中定义另一个类的实例对象，然后通过该对象调用其方法。根据“合成复用原则”，在系统中尽量使用关联关系来替代继承关系，因此大部分结构型模式都是对象结构型模式。

包含模式

- - 适配器模式(Adapter)
 - 重要程度：4
- - 桥接模式(Bridge)
 - 重要程度：3
- - 组合模式(Composite)
 - 重要程度：4
- - 装饰模式(Decorator)
 - 重要程度：3
- - 外观模式(Facade)
 - 重要程度：5
- - 享元模式(Flyweight)
 - 重要程度：1
- - 代理模式(Proxy)
 - 重要程度：4
- 1. 适配器模式
 - 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. 优点
 - 1.9. 缺点

- 1.10. 适用环境
 - 1.11. 模式应用
 - 1.12. 模式扩展
 - 1.13. 总结
- 2. 桥接模式
 - 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. 优点
 - 2.9. 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. 装饰模式
 - 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. 外观模式
 - 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. 享元模式
 - 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
- 6. 代理模式
 - 6.1. 模式动机
 - 6.2. 模式定义
 - 6.3. 模式结构
 - 6.4. 时序图
 - 6.5. 代码分析
 - 6.6. 模式分析
 - 6.7. 实例
 - 6.8. 优点
 - 6.9. 缺点
 - 6.10. 适用环境
 - 6.11. 模式应用
 - 6.12. 模式扩展
 - 6.13. 总结

1. 适配器模式

1.1. 模式动机

- 在软件开发中采用类似于电源适配器的设计和编码技巧被称为适配器模式。
- 通常情况下，客户端可以通过目标类的接口访问它所提供的服务。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是由于现有类中方法名与目标类中定义的方法名不一致等原因所导致的。
- 在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，适配器模式可以完成这样的转化。
- 在适配器模式中定义一个包装类，包装不兼容接口的对象，这个包装类指的就是适配器(Adapter)，它所包装的对象就是适配者(Adaptee)，即被适配的类。
- 适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。也就是说：当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。因此，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。

1.2. 模式定义

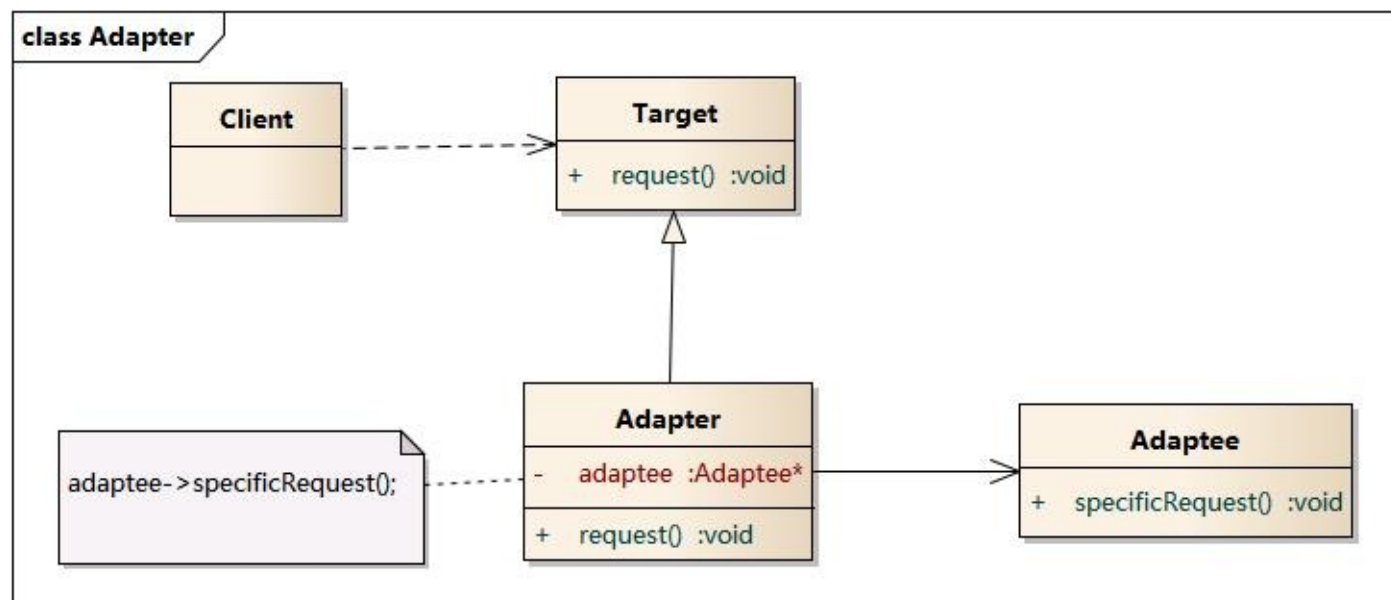
适配器模式(Adapter Pattern)：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。

1.3. 模式结构

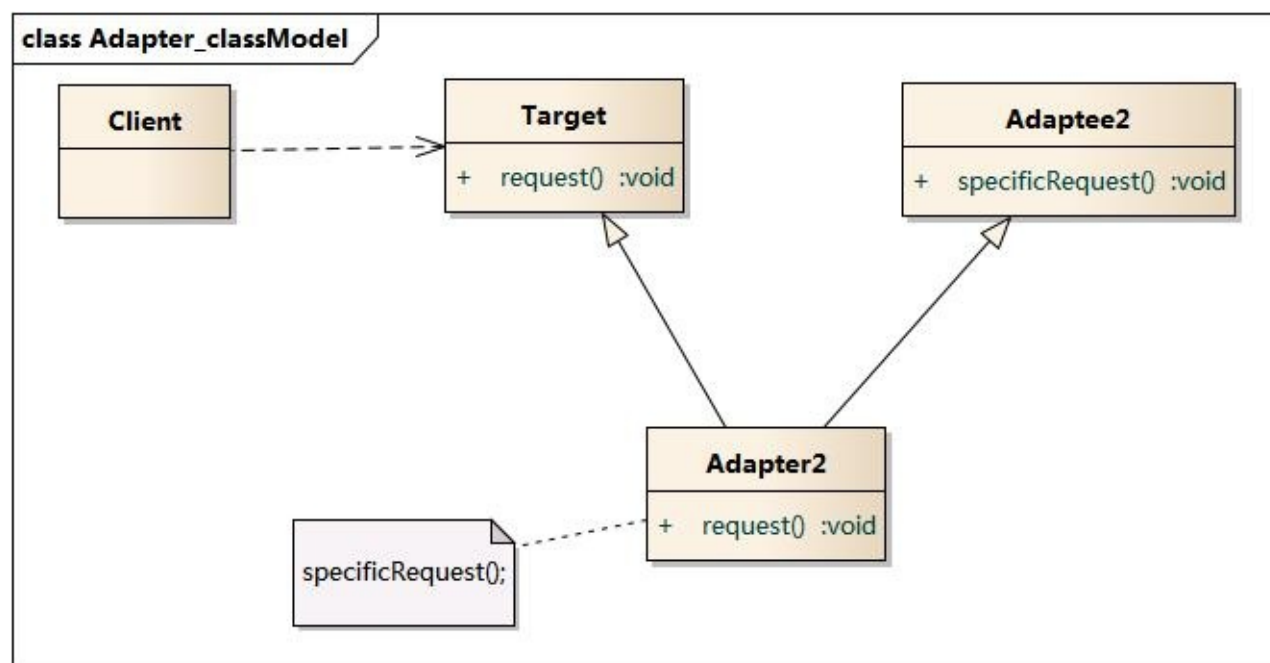
适配器模式包含如下角色：

- Target：目标抽象类
- Adapter：适配器类
- Adaptee：适配者类
- Client：客户类 适配器模式有对象适配器和类适配器两种实现：

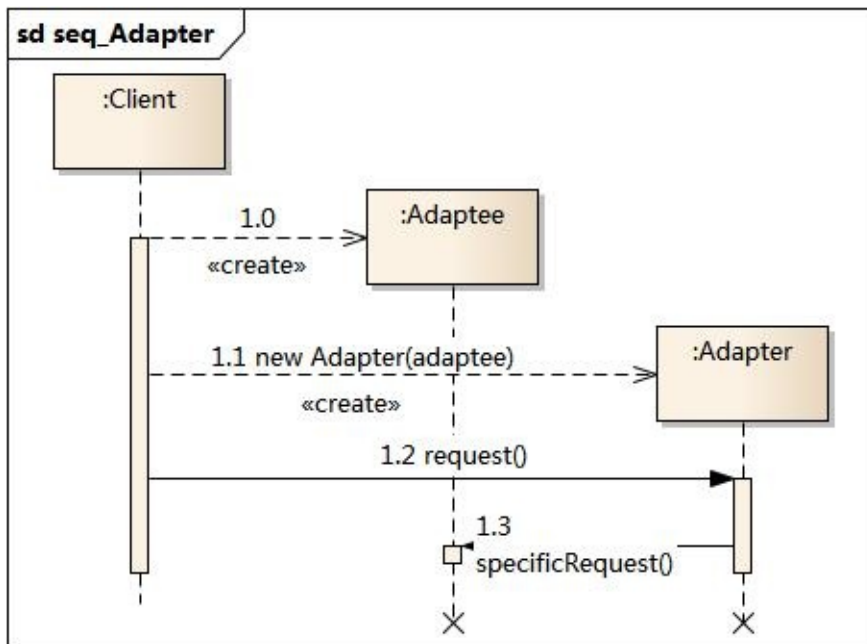
对象适配器：



类适配器:



1.4. 时序图



1.5. 代码分析

```

1. #include <iostream>
2. #include "Adapter.h"
3. #include "Adaptee.h"
4. #include "Target.h"
5.
6. using namespace std;
7.
8. int main(int argc, char *argv[])
9. {
10.     Adaptee * adaptee = new Adaptee();
11.     Target * tar = new Adapter(adaptee);
12.     tar->request();
13.
14.     return 0;
15. }
  
```

```

1. //////////////////////////////////////
2. // Adapter.h
3. // Implementation of the Class Adapter
4. // Created on:      03-十月-2014 17:32:00
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #if !defined(EA_BD766D47_0C69_4131_B7B9_21DF78B1E80D__INCLUDED_)
9. #define EA_BD766D47_0C69_4131_B7B9_21DF78B1E80D__INCLUDED_
10.
11. #include "Target.h"
12. #include "Adaptee.h"
13.
14. class Adapter : public Target
  
```

1. 适配器模式

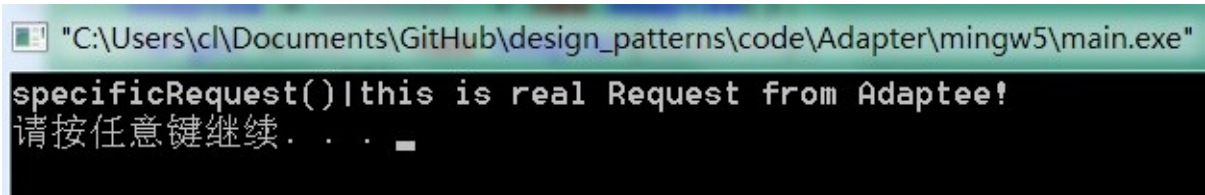
```
15. {
16.
17. public:
18.     Adapter(Adapter *adaptee);
19.     virtual ~Adapter();
20.
21.     virtual void request();
22.
23. private:
24.     Adapter* m_pAdaptee;
25.
26. };
27. #endif // !defined(EA_BD766D47_0C69_4131_B7B9_21DF78B1E80D__INCLUDED_)
```

```
1. //////////////////////////////////////
2. // Adapter.cpp
3. // Implementation of the Class Adapter
4. // Created on:      03-十月-2014 17:32:00
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #include "Adapter.h"
9.
10. Adapter::Adapter(Adapter * adaptee){
11.     m_pAdaptee = adaptee;
12. }
13.
14. Adapter::~Adapter(){
15.
16. }
17.
18. void Adapter::request(){
19.     m_pAdaptee->specificRequest();
20. }
```

```
1. //////////////////////////////////////
2. // Adaptee.h
3. // Implementation of the Class Adaptee
4. // Created on:      03-十月-2014 17:32:00
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #if !defined(EA_826E6B4F_12BE_4609_A0A3_95BD5E657D36__INCLUDED_)
9. #define EA_826E6B4F_12BE_4609_A0A3_95BD5E657D36__INCLUDED_
10.
11. class Adaptee
12. {
13.
14. public:
15.     Adaptee();
16.     virtual ~Adaptee();
```

1. 适配器模式

```
17.  
18.     void specificRequest();  
19.  
20. };  
21. #endif // !defined(EA_826E6B4F_12BE_4609_A0A3_95BD5E657D36__INCLUDED_)
```



运行结果：

1.6. 模式分析

1.7. 实例

1.8. 优点

- 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，而无需修改原有代码。
- 增加了类的透明性和复用性，将具体的实现封装在适配器类中，对于客户端类来说是透明的，而且提高了适配者的复用性。
- 灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”。
- 类适配器模式还具有如下优点：
 - 由于适配器类是适配者类的子类，因此可以在适配器类中置换一些适配者的方法，使得适配器的灵活性更强。
- 对象适配器模式还具有如下优点：
 - 一个对象适配器可以把多个不同的适配者适配到同一个目标，也就是说，同一个适配器可以把适配者类和它的子类都适配到目标接口。

1.9. 缺点

- 类适配器模式的缺点如下：
 - 对于Java、C#等不支持多重继承的语言，一次最多只能适配一个适配者类，而且目标抽象类只能为抽象类，不能为具体类，其使用有一定的局限性，不能将一个适配者类和它的子类都适配到目标接口。
- 对象适配器模式的缺点如下：
 - 与类适配器模式相比，要想置换适配者类的方法就不容易。如果一定要置换掉适配者类的一个或多个方法，就只好先做一个适配者类的子类，将适配者类的方法置换掉，然后再把适配者类的子类当做真正的适配者进行适配，实现过程较为复杂。

1.10. 适用环境

在以下情况下可以使用适配器模式：

- 系统需要使用现有的类，而这些类的接口不符合系统的需要。
- 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

1.11. 模式应用

Sun公司在1996年公开了Java语言的数据库连接工具JDBC，JDBC使得Java语言程序能够与数据库连接，并使用SQL语言来查询和操作数据。JDBC给出一个客户端通用的抽象接口，每一个具体数据库引擎（如SQL Server、Oracle、MySQL等）的JDBC驱动软件都是一个介于JDBC接口和数据库引擎接口之间的适配器软件。抽象的JDBC接口和各个数据库引擎API之间都需要相应的适配器软件，这就是为各个不同数据库引擎准备的驱动程序。

1.12. 模式扩展

- 认适配器模式(Default Adapter Pattern)或缺省适配器模式
- 当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可有选择地覆盖父类的某些方法来实现需求，它适用于一个接口不想使用其所有的方法的情况。因此也称为单接口适配器模式。

1.13. 总结

- 结构型模式描述如何将类或者对象结合在一起形成更大的结构。
- 适配器模式用于将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。
- 适配器模式包含四个角色：目标抽象类定义客户要用的特定领域的接口；适配器类可以调用另一个接口，作为一个转换器，对适配者和抽象目标类进行适配，它是适配器模式的核心；适配者类是被适配的角色，它定义了一个已经存在的接口，这个接口需要适配；在客户类中针对目标抽象类进行编程，调用在目标抽象类中定义的业务方法。
- 在类适配器模式中，适配器类实现了目标抽象类接口并继承了适配者类，并在目标抽象类的实现方法中调用所继承的适配者类的方法；在对象适配器模式中，适配器类继承了目标抽象类并定义了一个适配者类的对象实例，在所继承的目标抽象类方法中调用适配者类的相应业务方法。
- 适配器模式的主要优点是将目标类和适配者类解耦，增加了类的透明性和复用性，同时系统的灵活性和扩展性都非常好，更换适配器或者增加新的适配器都非常方便，符合“开闭原则”；类适配器模式的缺点是适配器类在很多编程语言中不能同时适配多个适配者类，对象适配器模式的缺点是很难置换适配者类的方法。
- 适配器模式适用情况包括：系统需要使用现有的类，而这些类的接口不符合系统的需要；想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类一起工作。

2. 桥接模式

2.1. 模式动机

设想如果要绘制矩形、圆形、椭圆、正方形，我们至少需要4个形状类，但是如果绘制的图形需要具有不同的颜色，如红色、绿色、蓝色等，此时至少有如下两种设计方案：

- 第一种设计方案是为每一种形状都提供一套各种颜色的版本。
- 第二种设计方案是根据实际需要形状和颜色进行组合

对于有两个变化维度（即两个变化的原因）的系统，采用方案二来进行设计系统中类的个数更少，且系统扩展更为方便。设计方案二即是桥接模式的应用。桥接模式将继承关系转换为关联关系，从而降低了类与类之间的耦合，减少了代码编写量。

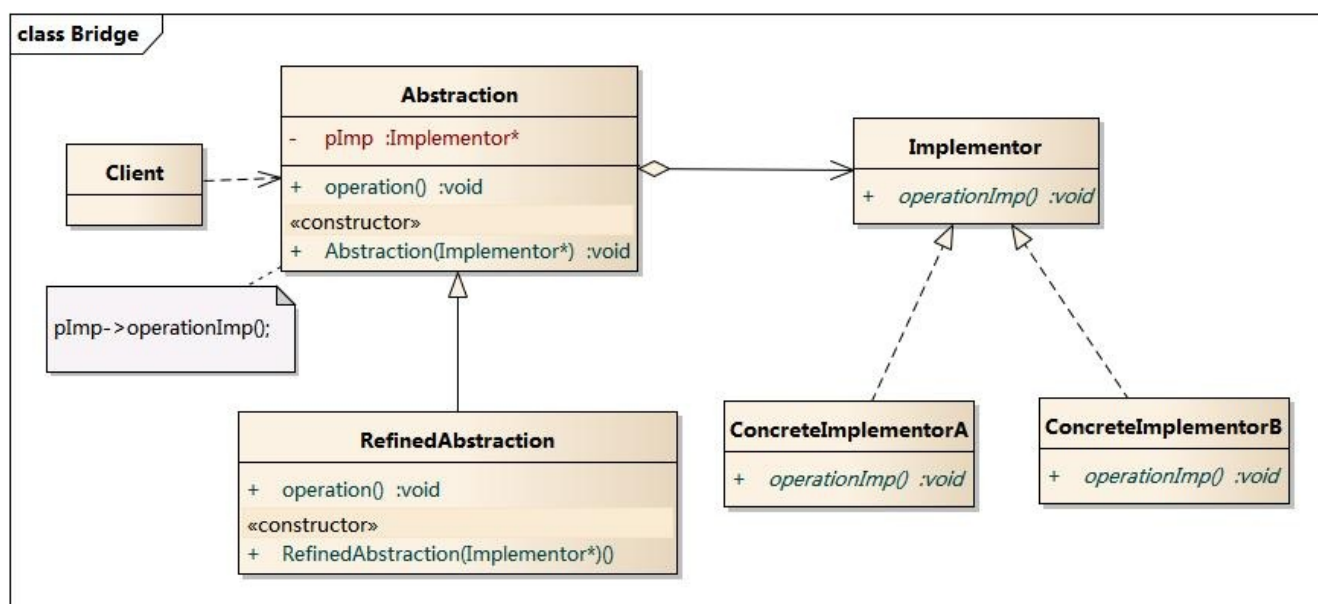
2.2. 模式定义

桥接模式(Bridge Pattern)：将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式。

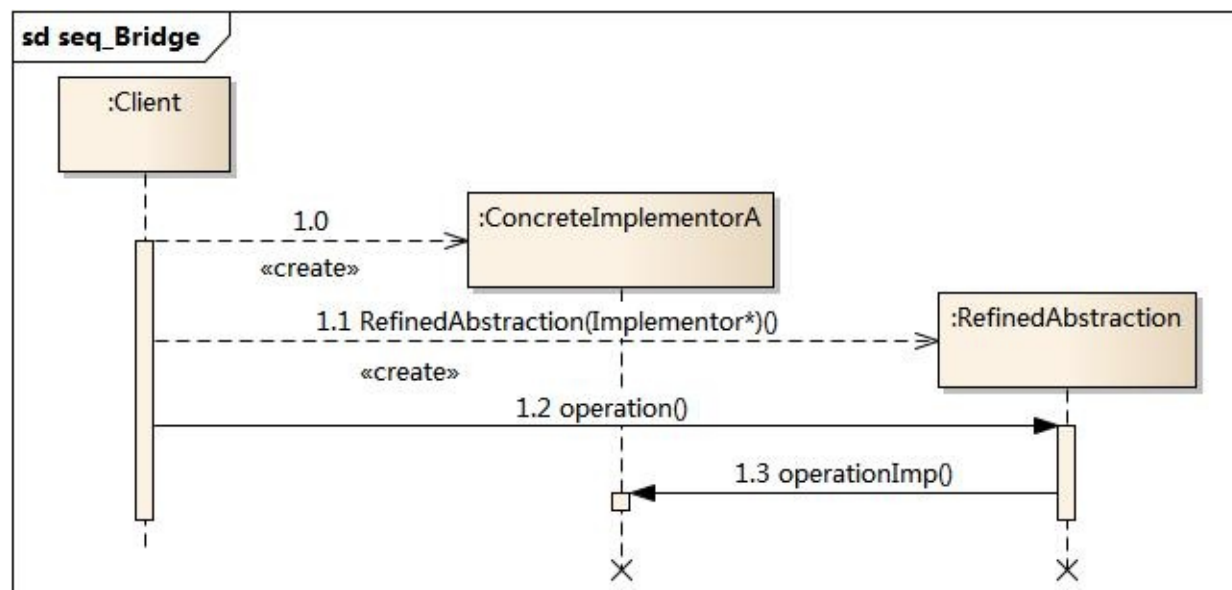
2.3. 模式结构

桥接模式包含如下角色：

- Abstraction：抽象类
- RefinedAbstraction：扩充抽象类
- Implementor：实现类接口
- ConcreteImplementor：具体实现类



2.4. 时序图



2.5. 代码分析

```

1. #include <iostream>
2. #include "ConcreteImplementorA.h"
3. #include "ConcreteImplementorB.h"
4. #include "RefinedAbstraction.h"
5. #include "Abstraction.h"
6.
7. using namespace std;
8.
9. int main(int argc, char *argv[])
10. {
11.
12.     Implementor * pImp = new ConcreteImplementorA();
13.     Abstraction * pa = new RefinedAbstraction(pImp);
14.     pa->operation();
15.
16.     Abstraction * pb = new RefinedAbstraction(new ConcreteImplementorB());
17.     pb->operation();
18.
19.     delete pa;
20.     delete pb;
21.
22.     return 0;
23. }
  
```

```

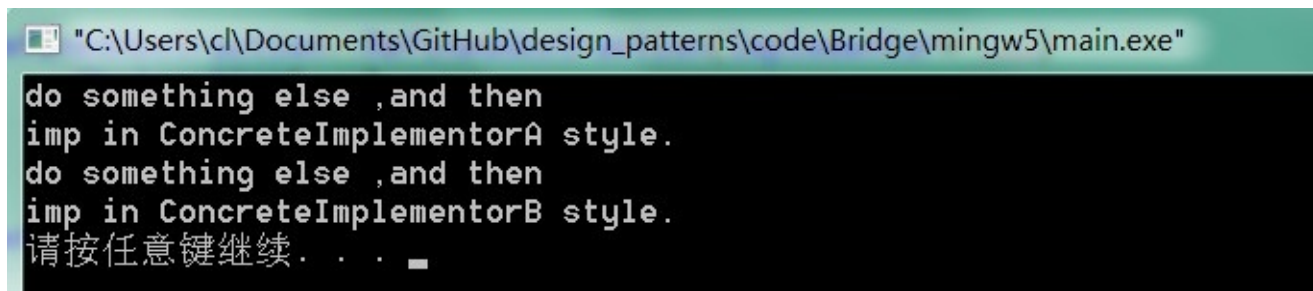
1. //////////////////////////////////////
2. // RefinedAbstraction.h
3. // Implementation of the Class RefinedAbstraction
4. // Created on:      03-十月-2014 18:12:43
5. // Original author: colin
  
```

2. 桥接模式

```
6. //////////////////////////////////////////////////
7.
8. #if !defined(EA_4BA5BE7C_DED5_4236_8362_F2988921CFA7__INCLUDED_)
9. #define EA_4BA5BE7C_DED5_4236_8362_F2988921CFA7__INCLUDED_
10.
11. #include "Abstraction.h"
12.
13. class RefinedAbstraction : public Abstraction
14. {
15.
16. public:
17.     RefinedAbstraction();
18.     RefinedAbstraction(Implementor* imp);
19.     virtual ~RefinedAbstraction();
20.
21.     virtual void operation();
22.
23. };
24. #endif // !defined(EA_4BA5BE7C_DED5_4236_8362_F2988921CFA7__INCLUDED_)
```

```
1. //////////////////////////////////////////////////
2. // RefinedAbstraction.cpp
3. // Implementation of the Class RefinedAbstraction
4. // Created on:      03-十月-2014 18:12:43
5. // Original author: colin
6. //////////////////////////////////////////////////
7.
8. #include "RefinedAbstraction.h"
9. #include <iostream>
10. using namespace std;
11.
12.
13. RefinedAbstraction::RefinedAbstraction(){
14.
15. }
16.
17. RefinedAbstraction::RefinedAbstraction(Implementor* imp)
18.     :Abstraction(imp)
19. {
20. }
21.
22. RefinedAbstraction::~~RefinedAbstraction(){
23.
24. }
25.
26. void RefinedAbstraction::operation(){
27.     cout << "do something else ,and then " << endl;
28.     m_pImp->operationImp();
29. }
30.
31.
```

运行结果：



2.6. 模式分析

理解桥接模式，重点需要理解如何将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化。

- 抽象化：抽象化就是忽略一些信息，把不同的实体当作同样的实体对待。在面向对象中，将对象的共同性质抽取出来形成类的过程即为抽象化的过程。
- 实现化：针对抽象化给出的具体实现，就是实现化，抽象化与实现化是一对互逆的概念，实现化产生的对象比抽象化更具体，是对抽象化事物的进一步具体化的产物。
- 脱耦：脱耦就是将抽象化和实现化之间的耦合解脱开，或者说是将它们之间的强关联改换成弱关联，将两个角色之间的继承关系改为关联关系。桥接模式中的所谓脱耦，就是指在一个软件系统的抽象化和实现化之间使用关联关系（组合或者聚合关系）而不是继承关系，从而使两者可以相对独立地变化，这就是桥接模式的用意。

2.7. 实例

如果需要开发一个跨平台视频播放器，可以在不同操作系统平台（如Windows、Linux、Unix等）上播放多种格式的视频文件，常见的视频格式包括MPEG、RMVB、AVI、WMV等。现使用桥接模式设计该播放器。

2.8. 优点

桥接模式的优点：

- 分离抽象接口及其实现部分。
- 桥接模式有时类似于多继承方案，但是多继承方案违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差，而且多继承结构中类的个数非常庞大，桥接模式是比多继承方案更好的解决方法。
- 桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。
- 实现细节对客户透明，可以对用户隐藏实现细节。

2.9. 缺点

桥接模式的缺点：

- 桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。- 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性。

2.10. 适用环境

在以下情况下可以使用桥接模式：

- 如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。
- 抽象化角色和实现化角色可以以继承的方式独立扩展而互不影响，在程序运行时可以动态将一个抽象化子类的对象和一个实现化子类的对象进行组合，即系统需要对抽象化角色和实现化角色进行动态耦合。
- 一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。
- 虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。
- 对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。

2.11. 模式应用

一个Java桌面软件总是带有所在操作系统的视感(LookAndFeel)，如果一个Java软件是在Unix系统上开发的，那么开发人员看到的是Motif用户界面的视感；在Windows上面使用这个系统的用户看到的是Windows用户界面的视感；而一个在Macintosh上面使用的用户看到的则是Macintosh用户界面的视感，Java语言是通过所谓的Peer架构做到这一点的。Java为AWT中的每一个GUI构件都提供了一个Peer构件，在AWT中的Peer架构就使用了桥接模式

2.12. 模式扩展

适配器模式与桥接模式的联用：

- 桥接模式和适配器模式用于设计的不同阶段，桥接模式用于系统的初步设计，对于存在两个独立变化维度的类可以将其分为抽象化和实现化两个角色，使它们可以分别进行变化；而在初步设计完成之后，当发现系统与已有类无法协同工作时，可以采用适配器模式。但有时候在设计初期也需要考虑适配器模式，特别是那些涉及到大量第三方应用接口的情况。

2.13. 总结

- 桥接模式将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式。
- 桥接模式包含如下四个角色：抽象类中定义了一个实现类接口类型的对象并可以维护该对象；扩充抽象类扩充由抽象类定义的接口，它实现了在抽象类中定义的抽象业务方法，在扩充抽象类中可以调用在实现类接口中定义的业务方法；实现类接口定义了实现类的接口，实现类接口仅提供基本操作，而抽象类定义的接口可能会做更多更复杂的操作；具体实现类实现了实现类接口并且具体实现它，在不同的具体实现类中提供基本操作的不同实现，在程序运行时，具体实现类对象将替换其父类对象，提供给客户端具体的业务操作方法。
- 在桥接模式中，抽象化(Abstraction)与实现化(Implementation)脱耦，它们可以沿着各自的维度独立变化。
- 桥接模式的主要优点是分离抽象接口及其实现部分，是比多继承方案更好的解决方法，桥接模式还提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统，实现细节对客户透明，可以对用户隐藏实现细节；其主要缺点是增加系统的理解与设计难度，且识别出系统中两个独立变化的维度并不是一件容易的事情。
- 桥接模式适用情况包括：需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建

立静态的继承联系；抽象化角色和实现化角色可以以继承的方式独立扩展而互不影响；一个类存在两个独立变化的维度，且这两个维度都需要进行扩展；设计要求需要独立管理抽象化角色和具体化角色；不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统。

3. 装饰模式

3.1. 模式动机

一般有两种方式可以实现给一个类或对象增加行为：

- 继承机制，使用继承机制是给现有类添加功能的一种有效途径，通过继承一个现有类可以使得子类在拥有自身方法的同时还拥有父类的方法。但是这种方法是静态的，用户不能控制增加行为的方式和时机。
- 关联机制，即将一个类的对象嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为以便扩展自己的行为，我们称这个嵌入的对象为装饰器(Decorator)

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任，换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。这就是装饰模式的模式动机。

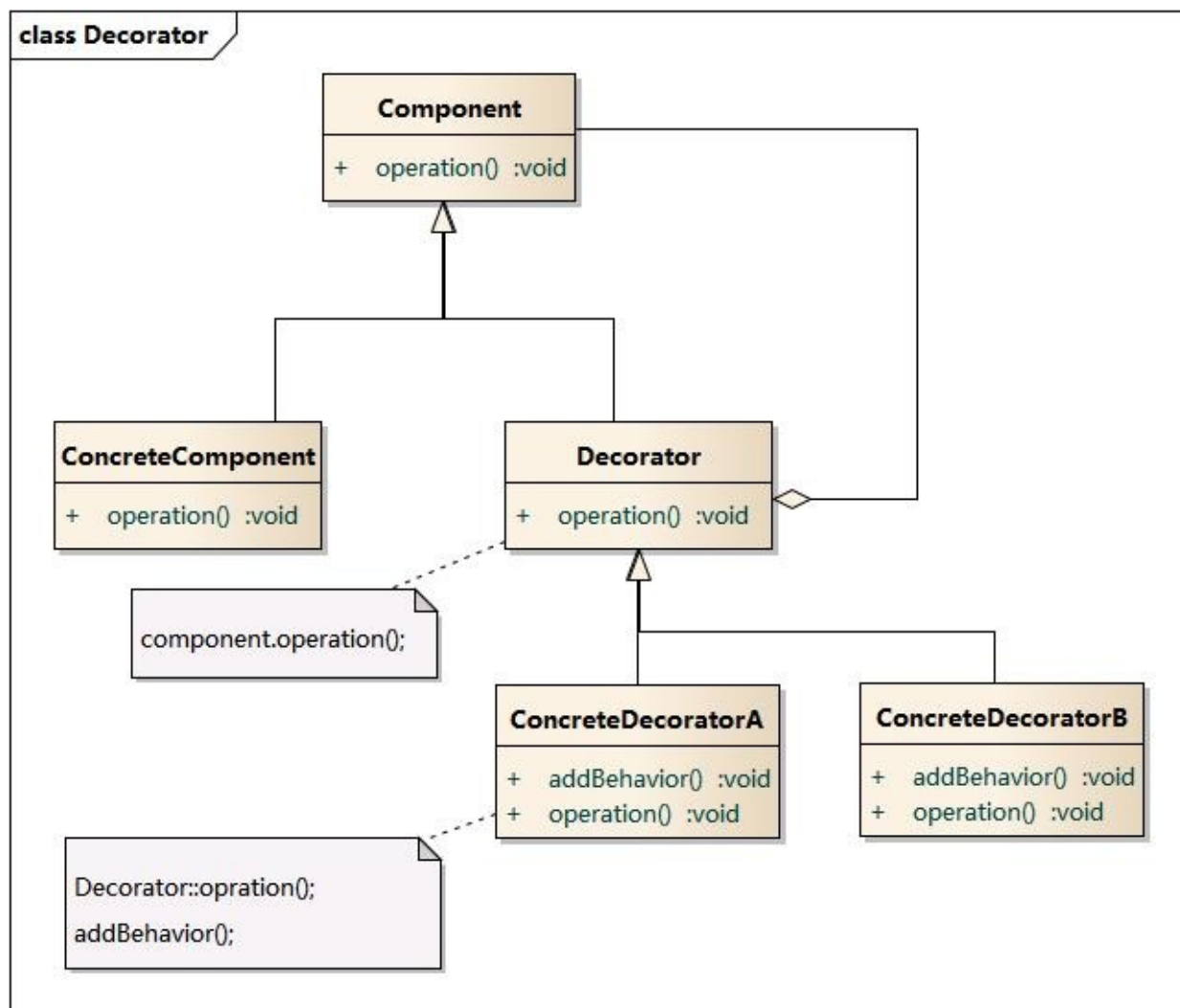
3.2. 模式定义

装饰模式(Decorator Pattern)：动态地给一个对象增加一些额外的职责(Responsibility)，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(wrapper)，与适配器模式的别名相同，但它们适用于不同的场合。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

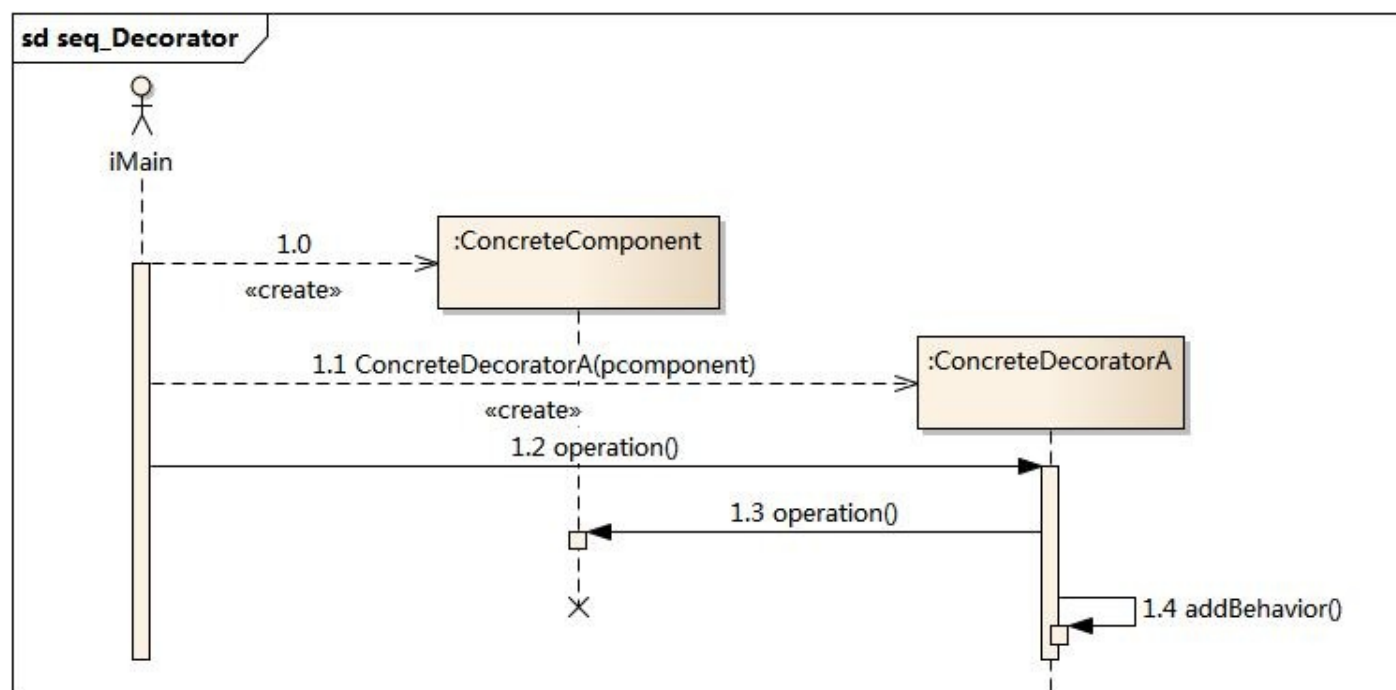
3.3. 模式结构

装饰模式包含如下角色：

- Component：抽象构件
- ConcreteComponent：具体构件
- Decorator：抽象装饰类
- ConcreteDecorator：具体装饰类



3.4. 时序图



3.5. 代码分析

```

1.  //////////////////////////////////
2.  //  ConcreteComponent.cpp
3.  //  Implementation of the Class ConcreteComponent
4.  //  Created on:      03-十月-2014 18:53:00
5.  //  Original author: colin
6.  //////////////////////////////////
7.
8.  #include "ConcreteComponent.h"
9.  #include <iostream>
10. using namespace std;
11.
12.
13. ConcreteComponent::ConcreteComponent(){
14.
15. }
16.
17. ConcreteComponent::~~ConcreteComponent(){
18.
19. }
20.
21. void ConcreteComponent::operation(){
22.     cout << "ConcreteComponent's normal operation!" << endl;
23. }

```

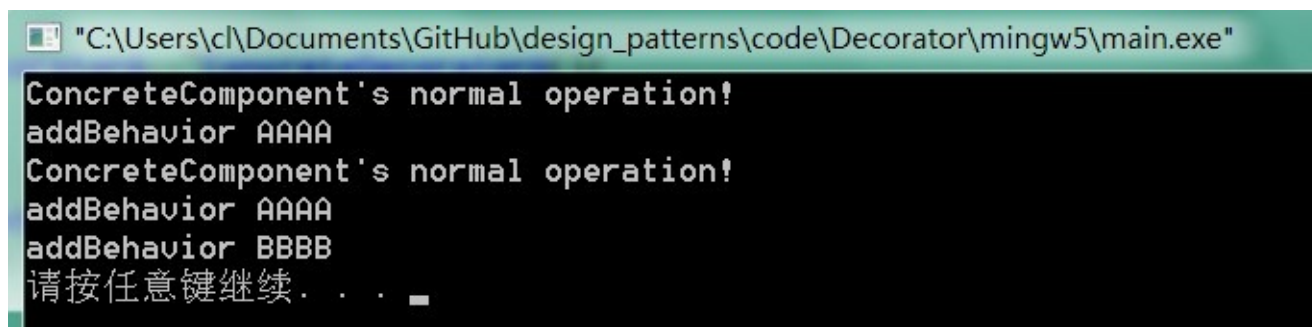
```

1.  //////////////////////////////////
2.  //  ConcreteDecoratorA.h
3.  //  Implementation of the Class ConcreteDecoratorA
4.  //  Created on:      03-十月-2014 18:53:00
5.  //  Original author: colin
6.  //////////////////////////////////
7.
8.  #if !defined(EA_6786B68E_DCE4_44c4_B26D_812F0B3C0382__INCLUDED_)
9.  #define EA_6786B68E_DCE4_44c4_B26D_812F0B3C0382__INCLUDED_
10.
11. #include "Decorator.h"
12. #include "Component.h"
13.
14. class ConcreteDecoratorA : public Decorator
15. {
16.
17. public:
18.     ConcreteDecoratorA(Component* pcmp);
19.     virtual ~ConcreteDecoratorA();
20.
21.     void addBehavior();
22.     virtual void operation();
23.
24. };
25. #endif // !defined(EA_6786B68E_DCE4_44c4_B26D_812F0B3C0382__INCLUDED_)

```

```
1. //////////////////////////////////////
2. // ConcreteDecoratorA.cpp
3. // Implementation of the Class ConcreteDecoratorA
4. // Created on:      03-十月-2014 18:53:00
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #include "ConcreteDecoratorA.h"
9. #include <iostream>
10. using namespace std;
11.
12. ConcreteDecoratorA::ConcreteDecoratorA(Component* pcmp)
13. :Decorator(pcmp)
14. {
15.
16. }
17.
18. ConcreteDecoratorA::~ConcreteDecoratorA(){
19.
20. }
21.
22. void ConcreteDecoratorA::addBehavior(){
23.     cout << "addBehavior AAAA" << endl;
24. }
25.
26.
27. void ConcreteDecoratorA::operation(){
28.     Decorator::operation();
29.     addBehavior();
30. }
```

运行结果：



```
"C:\Users\cl\Documents\GitHub\design_patterns\code\Decorator\mingw5\main.exe"
ConcreteComponent's normal operation!
addBehavior AAAA
ConcreteComponent's normal operation!
addBehavior AAAA
addBehavior BBBB
请按任意键继续. . .
```

3.6. 模式分析

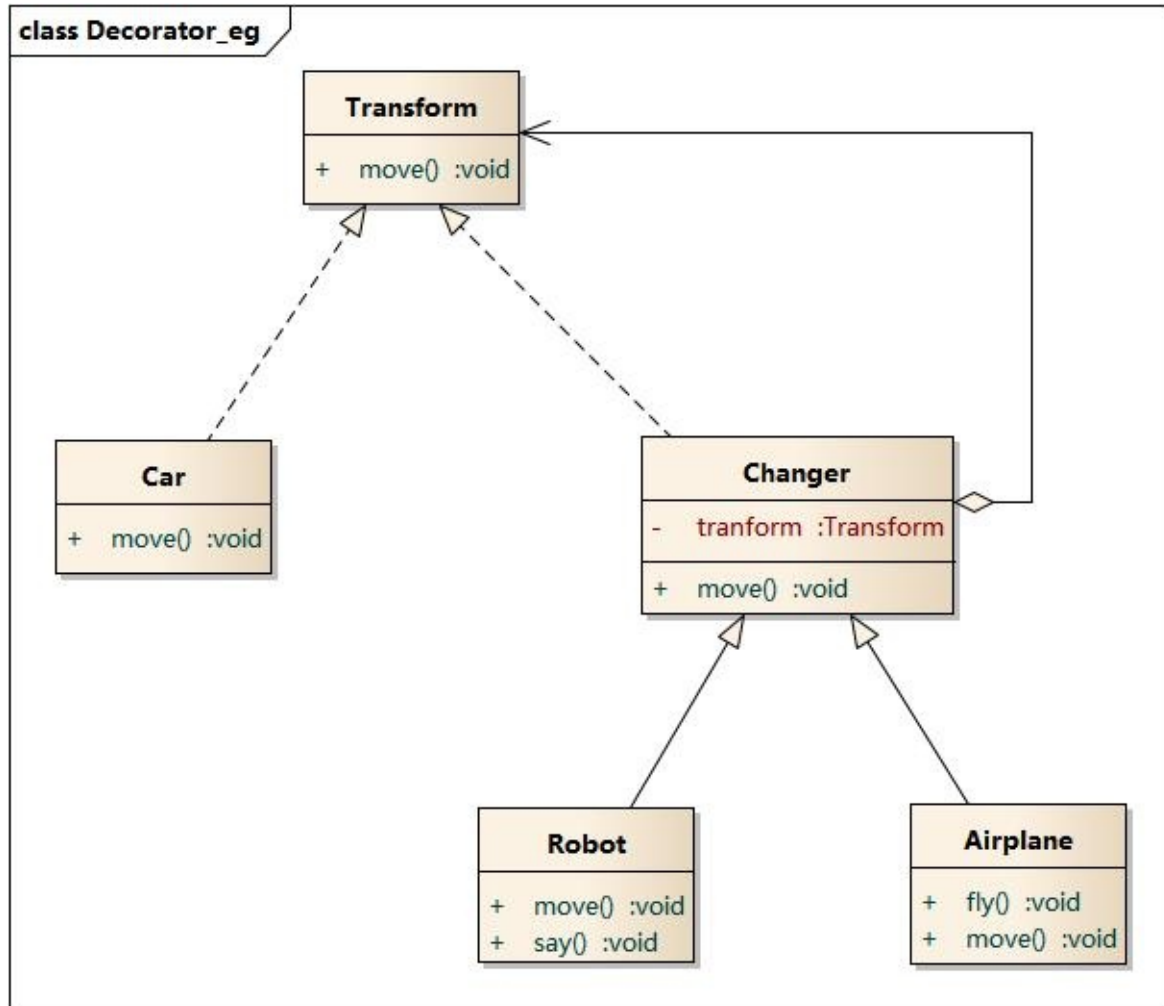
- 与继承关系相比，关联关系的主要优势在于不会破坏类的封装性，而且继承是一种耦合度较大的静态关系，无法在程序运行时动态扩展。在软件开发阶段，关联关系虽然不会比继承关系减少编码量，但是到了软件维护阶段，由于关联关系使系统具有较好的松耦合性，因此使得系统更加容易维护。当然，关联关系的缺点是比继承关系要创建更多的对象。
- 使用装饰模式来实现扩展比继承更加灵活，它以对客户透明的方式动态地给一个对象附加更多的责任。装饰模式

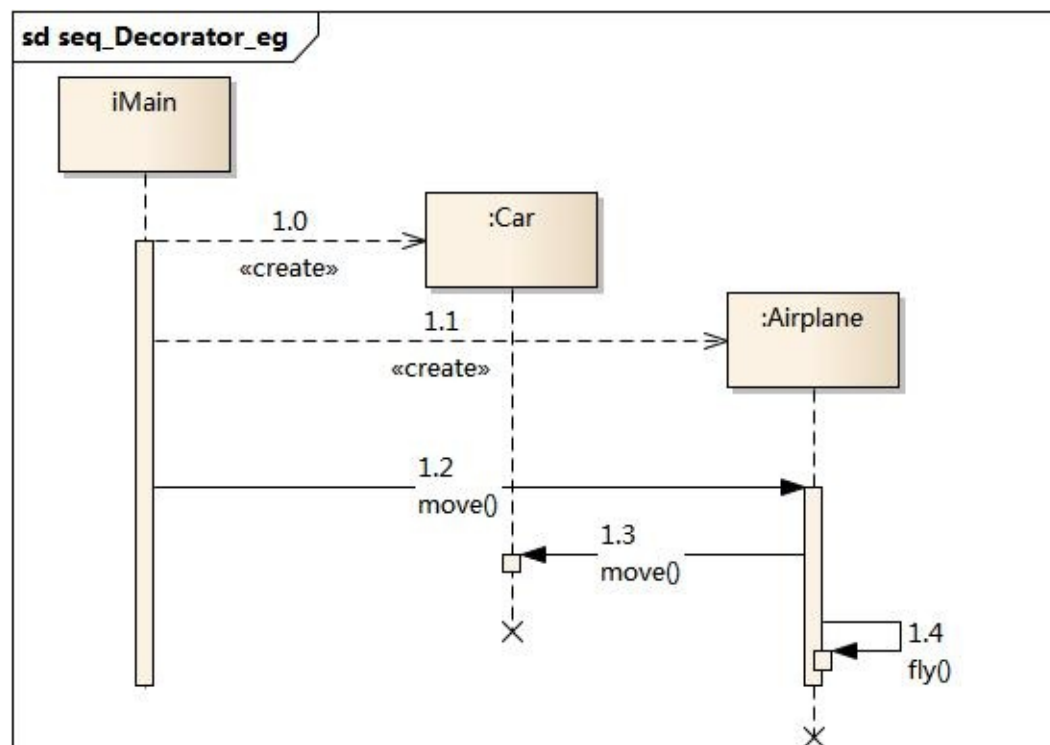
可以在不需要创造更多子类的情况下，将对象的功能加以扩展。

3.7. 实例

实例：变形金刚

变形金刚在变形之前是一辆汽车，它可以在陆地上移动。当它变成机器人之后除了能够在陆地上移动之外，还可以说话；如果需要，它还可以变成飞机，除了能够在陆地上移动还可以在天空中飞翔。





3.8. 优点

装饰模式的优点：

- 装饰模式与继承关系的目都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。
- 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的装饰器，从而实现不同的行为。
- 通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合。可以使用多个具体装饰类来装饰同一对象，得到功能更为强大的对象。
- 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，在使用时再对其进行组合，原有代码无须改变，符合“开闭原则”

3.9. 缺点

装饰模式的缺点：

- 使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同，同时还将产生很多具体装饰类。这些装饰类和小对象的产生将增加系统的复杂度，加大学习与理解的难度。
- 这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。

3.10. 适用环境

在以下情况下可以使用装饰模式：

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

- 需要动态地给一个对象增加功能，这些功能也可以动态地被撤销。
- 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长；第二类是因为类定义不能继承（如final类）。

3.11. 模式应用

3.12. 模式扩展

装饰模式的简化-需要注意的问题：

- 一个装饰类的接口必须与被装饰类的接口保持相同，对于客户端来说无论是装饰之前的对象还是装饰之后的对象都可以一致对待。
- 尽量保持具体构件类Component作为一个“轻”类，也就是说不要把太多的逻辑和状态放在具体构件类中，可以通过装饰类

对其进行扩展。 - 如果只有一个具体构件类而没有抽象构件类，那么抽象装饰类可以作为具体构件类的直接子类。

3.13. 总结

- 装饰模式用于动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。它是一种对象结构型模式。
- 装饰模式包含四个角色：抽象构件定义了对象的接口，可以给这些对象动态增加职责（方法）；具体构件定义了具体的构件对象，实现了在抽象构件中声明的方法，装饰器可以给它增加额外的职责（方法）；抽象装饰类是抽象构件类的子类，用于给具体构件增加职责，但是具体职责在其子类中实现；具体装饰类是抽象装饰类的子类，负责向构件添加新的职责。
- 使用装饰模式来实现扩展比继承更加灵活，它以对客户透明的方式动态地给一个对象附加更多的责任。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。
- 装饰模式的主要优点在于可以提供比继承更多的灵活性，可以通过一种动态的方式来扩展一个对象的功能，并通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合，而且具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类；其主要缺点在于使用装饰模式进行系统设计时将产生很多小对象，而且装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。
- 装饰模式适用情况包括：在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责；需要动态地给一个对象增加功能，这些功能也可以动态地被撤销；当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。
- 装饰模式可分为透明装饰模式和半透明装饰模式：在透明装饰模式中，要求客户端完全针对抽象编程，装饰模式的透明性要求客户端程序不应该声明具体构件类型和具体装饰类型，而应该全部声明为抽象构件类型；半透明装饰模式允许用户在客户端声明具体装饰者类型的对象，调用在具体装饰者中新增的方法。

4. 外观模式

4.1. 模式动机

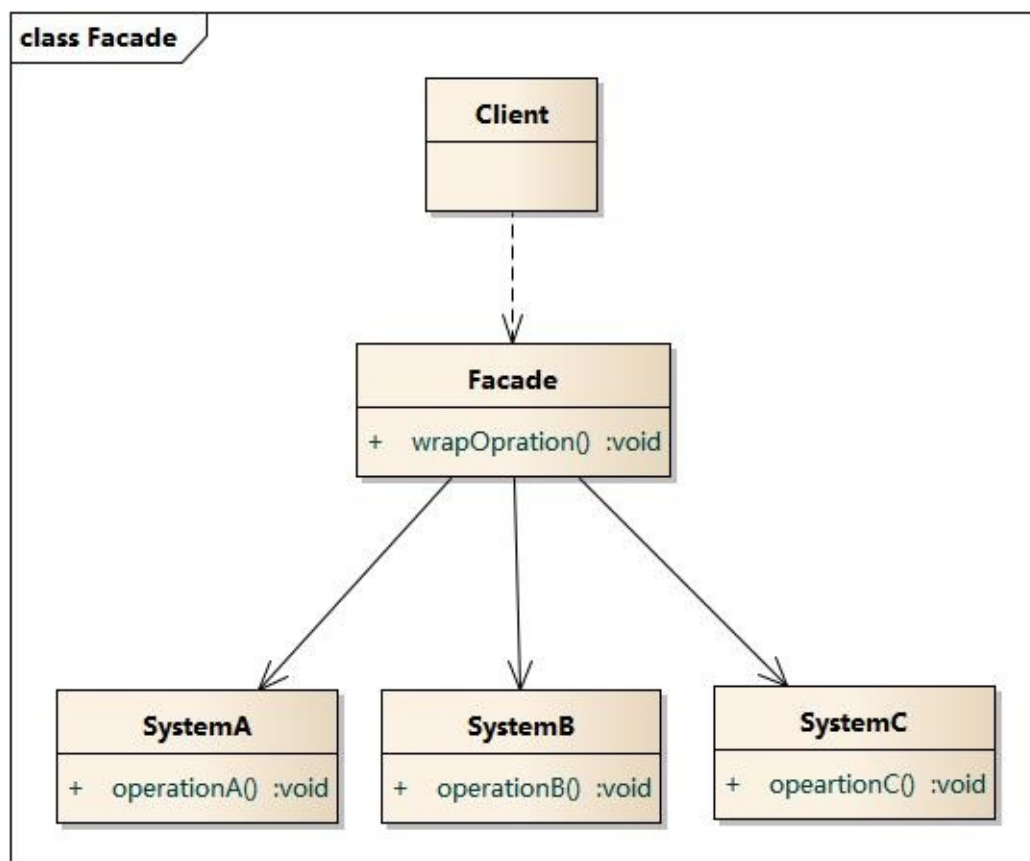
4.2. 模式定义

外观模式(Facade Pattern)：外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。外观模式又称为门面模式，它是一种对象结构型模式。

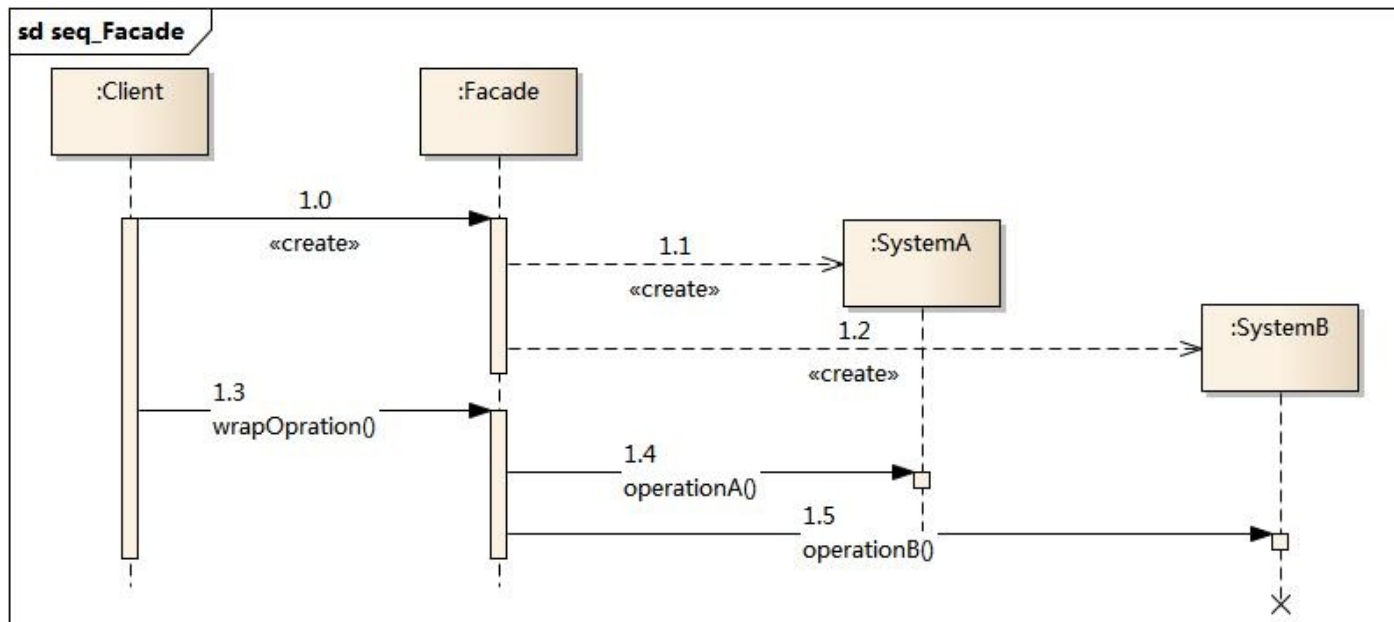
4.3. 模式结构

外观模式包含如下角色：

- Facade：外观角色
- SubSystem：子系统角色



4.4. 时序图



4.5. 代码分析

```

1. #include <iostream>
2. #include "Facade.h"
3. using namespace std;
4.
5. int main(int argc, char *argv[])
6. {
7.     Facade fa;
8.     fa.wrapOperation();
9.
10.    return 0;
11. }
  
```

```

1. //////////////////////////////////////
2. // Facade.h
3. // Implementation of the Class Facade
4. // Created on:      06-十月-2014 19:10:44
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #if !defined(EA_FD130A87_92A9_4168_9B33_7A925C47AFD5__INCLUDED_)
9. #define EA_FD130A87_92A9_4168_9B33_7A925C47AFD5__INCLUDED_
10.
11. #include "SystemC.h"
12. #include "SystemA.h"
13. #include "SystemB.h"
14.
15. class Facade
16. {
17.
18. public:
19.     Facade();
  
```


4. 外观模式

```
20.     virtual ~Facade();
21.
22.     void wrapOperation();
23.
24. private:
25.     SystemC *m_SystemC;
26.     SystemA *m_SystemA;
27.     SystemB *m_SystemB;
28. };
29. #endif // !defined(EA_FD130A87_92A9_4168_9B33_7A925C47AFD5__INCLUDED_)
```

```
1.  //////////////////////////////////////
2.  //  Facade.cpp
3.  //  Implementation of the Class Facade
4.  //  Created on:      06-十月-2014 19:10:44
5.  //  Original author: colin
6.  //////////////////////////////////////
7.
8.  #include "Facade.h"
9.
10.
11. Facade::Facade(){
12.     m_SystemA = new SystemA();
13.     m_SystemB = new SystemB();
14.     m_SystemC = new SystemC();
15. }
16.
17.
18.
19. Facade::~~Facade(){
20.     delete m_SystemA;
21.     delete m_SystemB;
22.     delete m_SystemC;
23. }
24.
25. void Facade::wrapOperation(){
26.     m_SystemA->operationA();
27.     m_SystemB->operationB();
28.     m_SystemC->opeartionC();
29. }
```

运行结果:



```
"C:\Users\cl\Documents\GitHub\design_patterns\code\Facade\mingw5\main.exe"
operationA
operationB
operationC
请按任意键继续...
```

4.6. 模式分析

根据“单一职责原则”，在软件中将一个系统划分为若干个子系统有利于降低整个系统的复杂性，一个常见的设计目标是使子系统间的通信和相互依赖关系达到最小，而达到该目标的途径之一就是引入一个外观对象，它为子系统的访问提供了一个简单而单一的入口。-外观模式也是“迪米特法则”的体现，通过引入一个新的外观类可以降低原有系统的复杂度，同时降低客户类与子系统类的耦合度。- 外观模式要求一个子系统的外部与其内部的通信通过一个统一的外观对象进行，外观类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与外观对象打交道，而不需要与子系统内部的很多对象打交道。-外观模式的目的在于降低系统的复杂程度。-外观模式从很大程度上提高了客户端使用的便捷性，使得客户端无须关心子系统的工作细节，通过外观角色即可调用相关功能。

4.7. 实例

4.8. 优点

外观模式的优点

- 对客户屏蔽子系统组件，减少了客户处理的对象数目并使得子系统使用起来更加容易。通过引入外观模式，客户代码将变得很简单，与之关联的对象也很少。
- 实现了子系统与客户之间的松耦合关系，这使得子系统的组件变化不会影响到调用它的客户类，只需要调整外观类即可。
- 降低了大型软件系统中的编译依赖性，并简化了系统在不同平台之间的移植过程，因为编译一个子系统一般不需要编译所有其他的子系统。一个子系统的修改对其他子系统没有任何影响，而且子系统内部变化也不会影响到外观对象。
- 只是提供了一个访问子系统的统一入口，并不影响用户直接使用子系统类。

4.9. 缺点

外观模式的缺点

- 不能很好地限制客户使用子系统类，如果对客户访问子系统类做太多的限制则减少了可变性和灵活性。
- 在不引入抽象外观类的情况下，增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。

4.10. 适用环境

在以下情况下可以使用外观模式：

- 当要为一个复杂子系统提供一个简单接口时可以使用外观模式。该接口可以满足大多数用户的需求，而且用户也可以越过外观类直接访问子系统。
- 客户程序与多个子系统之间存在很大的依赖性。引入外观类将子系统与客户以及其他子系统解耦，可以提高子系统的独立性和可移植性。
- 在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。

4.11. 模式应用

4.12. 模式扩展

- 一个系统有多个外观类
- 在外观模式中，通常只需要一个外观类，并且此外观类只有一个实例，换言之它是一个单例类。在很多情况下为了节约系统资源，一般将外观类设计为单例类。当然这并不意味着在整个系统里只能有一个外观类，在一个系统中可以设计多个外观类，每个外观类都负责和一些特定的子系统交互，向用户提供相应的业务功能。
- 不要试图通过外观类为子系统增加新行为
- 不要通过继承一个外观类在子系统加入新的行为，这种做法是错误的。外观模式的用意是为子系统提供一个集中化和简化的沟通渠道，而不是向子系统加入新的行为，新的行为的增加应该通过修改原有子系统类或增加新的子系统类来实现，不能通过外观类来实现。
- 外观模式与迪米特法则
- 外观模式创造出外观对象，将客户端所涉及的属于一个子系统的协作伙伴的数量减到最少，使得客户端与子系统内部的对象的相互作用被外观对象所取代。外观类充当了客户类与子系统类之间的“第三者”，降低了客户类与子系统类之间的耦合度，外观模式就是实现代码重构以便达到“迪米特法则”要求的一个强有力的武器。
- 抽象外观类的引入
- 外观模式最大的缺点在于违背了“开闭原则”，当增加新的子系统或者移除子系统时需要修改外观类，可以通过引入抽象外观类在一定程度上解决该问题，客户端针对抽象外观类进行编程。对于新的业务需求，不修改原有外观类，而对应增加一个新的具体外观类，由新的具体外观类来关联新的子系统对象，同时通过修改配置文件来达到不修改源代码并更换外观类的目的。

4.13. 总结

- 在外观模式中，外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。外观模式又称为门面模式，它是一种对象结构型模式。
- 外观模式包含两个角色：外观角色是在客户端直接调用的角色，在外观角色中可以知道相关的(一个或者多个)子系统的功能和责任，它将所有从客户端发来的请求委派到相应的子系统去，传递给相应的子系统对象处理；在软件系统中可以同时有一个或者多个子系统角色，每一个子系统可以不是一个单独的类，而是一个类的集合，它实现子系统的功能。
- 外观模式要求一个子系统的外部与其内部的通信通过一个统一的外观对象进行，外观类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与外观对象打交道，而不需要与子系统内部的很多对象打交道。
- 外观模式主要优点在于对客户屏蔽子系统组件，减少了客户处理的对象数目并使得子系统使用起来更加容易，它实现了子系统与客户之间的松耦合关系，并降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程；其缺点在于不能很好地限制客户使用子系统类，而且在不引入抽象外观类的情况下，增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。
- 外观模式适用情况包括：要为一个复杂子系统提供一个简单接口；客户程序与多个子系统之间存在很大的依赖性；在层次化结构中，需要定义系统中每一层的入口，使得层与层之间不直接产生联系。

5. 享元模式

5.1. 模式动机

面向对象技术可以很好地解决一些灵活性或可扩展性问题，但在很多情况下需要在系统中增加类和对象的个数。当对象数量太多时，将导致运行代价过高，带来性能下降等问题。

- 享元模式正是为解决这一类问题而诞生的。享元模式通过共享技术实现相同或相似对象的重用。
- 在享元模式中可以共享的相同内容称为内部状态(Intrinsic State)，而那些需要外部环境来设置的不能共享的内容称为外部状态(Extrinsic State)，由于区分了内部状态和外部状态，因此可以通过设置不同的外部状态使得相同的对象可以具有一些不同的特征，而相同的内部状态是可以共享的。
- 在享元模式中通常会出现工厂模式，需要创建一个享元工厂来负责维护一个享元池(Flyweight Pool)用于存储具有相同内部状态的享元对象。
- 在享元模式中共享的是享元对象的内部状态，外部状态需要通过环境来设置。在实际使用中，能够共享的内部状态是有限的，因此享元对象一般都设计为较小的对象，它所包含的内部状态较少，这种对象也称为细粒度对象。享元模式的目的是使用共享技术来实现大量细粒度对象的复用。

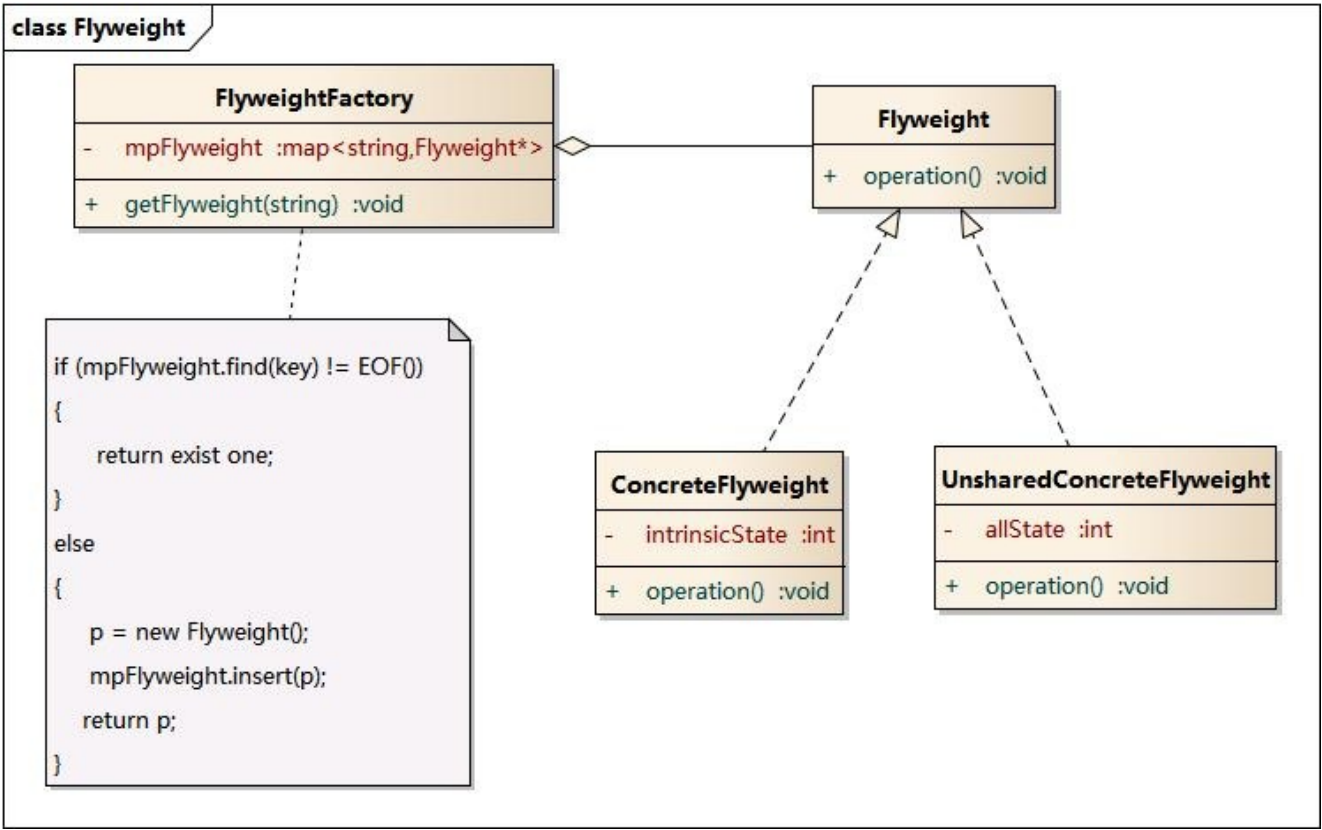
5.2. 模式定义

享元模式(Flyweight Pattern)：运用共享技术有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。由于享元模式要求能够共享的对象必须是细粒度对象，因此它又称为轻量级模式，它是一种对象结构型模式。

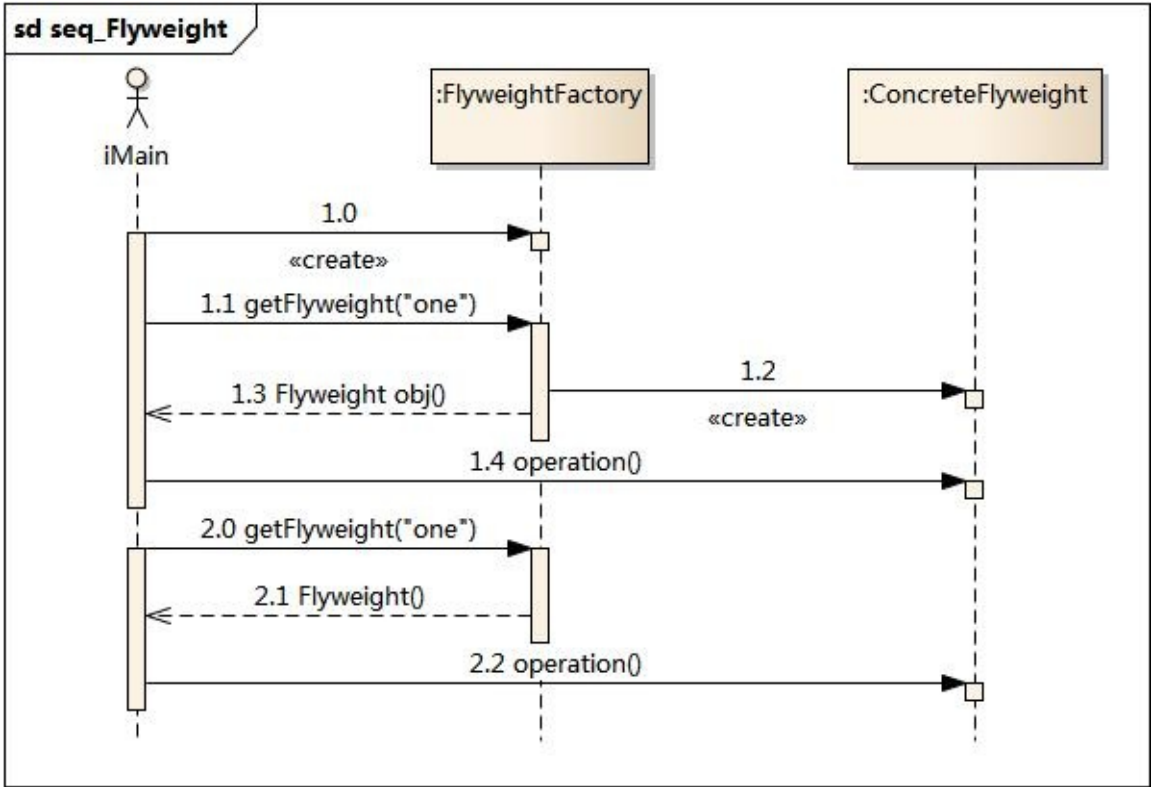
5.3. 模式结构

享元模式包含如下角色：

- Flyweight：抽象享元类
- ConcreteFlyweight：具体享元类
- UnsharedConcreteFlyweight：非共享具体享元类
- FlyweightFactory：享元工厂类



5.4. 时序图



5.5. 代码分析

```
1. #include <iostream>
```

```

2. #include "ConcreteFlyweight.h"
3. #include "FlyweightFactory.h"
4. #include "Flyweight.h"
5. using namespace std;
6.
7. int main(int argc, char *argv[])
8. {
9.     FlyweightFactory factory;
10.    Flyweight * fw = factory.getFlyweight("one");
11.    fw->operation();
12.
13.    Flyweight * fw2 = factory.getFlyweight("two");
14.    fw2->operation();
15.    //aready exist in pool
16.    Flyweight * fw3 = factory.getFlyweight("one");
17.    fw3->operation();
18.    return 0;
19. }

```

```

1. //////////////////////////////////////////////////
2. // FlyweightFactory.cpp
3. // Implementation of the Class FlyweightFactory
4. // Created on:      06-十月-2014 20:10:42
5. // Original author: colin
6. //////////////////////////////////////////////////
7.
8. #include "FlyweightFactory.h"
9. #include "ConcreteFlyweight.h"
10. #include <iostream>
11. using namespace std;
12.
13. FlyweightFactory::FlyweightFactory(){
14.
15. }
16.
17.
18.
19. FlyweightFactory::~FlyweightFactory(){
20.
21. }
22.
23. Flyweight* FlyweightFactory::getFlyweight(string str){
24.     map<string, Flyweight*>::iterator itr = m_mpFlyweight.find(str);
25.     if(itr == m_mpFlyweight.end())
26.     {
27.         Flyweight * fw = new ConcreteFlyweight(str);
28.         m_mpFlyweight.insert(make_pair(str, fw));
29.         return fw;
30.     }
31.     else
32.     {
33.         cout << "aready in the pool,use the exist one:" << endl;

```

```

34.         return itr->second;
35.     }
36. }

```

```

1.  //////////////////////////////////////
2. //  ConcreteFlyweight.h
3. //  Implementation of the Class ConcreteFlyweight
4. //  Created on:      06-十月-2014 20:10:42
5. //  Original author: colin
6.  //////////////////////////////////////
7.
8. #if !defined(EA_C0AF438E_96E4_46f1_ADEC_308EF16E11D1__INCLUDED_)
9. #define EA_C0AF438E_96E4_46f1_ADEC_308EF16E11D1__INCLUDED_
10.
11. #include "Flyweight.h"
12. #include <string>
13. using namespace std;
14.
15. class ConcreteFlyweight : public Flyweight
16. {
17.
18. public:
19.     ConcreteFlyweight(string str);
20.     virtual ~ConcreteFlyweight();
21.
22.     virtual void operation();
23.
24. private:
25.     string intrinsicState;
26.
27. };
28. #endif // !defined(EA_C0AF438E_96E4_46f1_ADEC_308EF16E11D1__INCLUDED_)

```

```

1.  //////////////////////////////////////
2. //  ConcreteFlyweight.cpp
3. //  Implementation of the Class ConcreteFlyweight
4. //  Created on:      06-十月-2014 20:10:42
5. //  Original author: colin
6.  //////////////////////////////////////
7.
8. #include "ConcreteFlyweight.h"
9. #include <iostream>
10. using namespace std;
11.
12.
13. ConcreteFlyweight::ConcreteFlyweight(string str){
14.     intrinsicState = str;
15. }
16.
17. ConcreteFlyweight::~ConcreteFlyweight(){
18.

```



```
19. }  
20.  
21. void ConcreteFlyweight::operation(){  
22.     cout << "Flyweight[" << intrinsicState << "] do operation." << endl;  
23. }
```

运行结果：



5.6. 模式分析

享元模式是一个考虑系统性能的设计模式，通过使用享元模式可以节约内存空间，提高系统的性能。

享元模式的核心在于享元工厂类，享元工厂类的作用在于提供一个用于存储享元对象的享元池，用户需要对象时，首先从享元池中获取，如果享元池中不存在，则创建一个新的享元对象返回给用户，并在享元池中保存该新增对象。

享元模式以共享的方式高效地支持大量的细粒度对象，享元对象能做到共享的关键是区分内部状态(Internal State)和外部状态(External State)。

- 内部状态是存储在享元对象内部并且不会随环境改变而改变的状态，因此内部状态可以共享。
- 外部状态是随环境改变而改变的、不可以共享的状态。享元对象的外部状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入到享元对象内部。一个外部状态与另一个外部状态之间是相互独立的。

5.7. 实例

5.8. 优点

享元模式的优点

- 享元模式的优点在于它可以极大减少内存中对象的数量，使得相同对象或相似对象在内存中只保存一份。
- 享元模式的外部状态相对独立，而且不会影响其内部状态，从而使得享元对象可以在不同的环境中被共享。

5.9. 缺点

享元模式的缺点

- 享元模式使得系统更加复杂，需要分离出内部状态和外部状态，这使得程序的逻辑复杂化。
- 为了使对象可以共享，享元模式需要将享元对象的状态外部化，而读取外部状态使得运行时间变长。

5.10. 适用环境

在以下情况下可以使用享元模式：

- 一个系统有大量相同或者相似的对象，由于这类对象的大量使用，造成内存的大量耗费。
- 对象的大部分状态都可以外部化，可以将这些外部状态传入对象中。
- 使用享元模式需要维护一个存储享元对象的享元池，而这需要耗费资源，因此，应当在多次重复使用享元对象时才值得使用享元模式。

5.11. 模式应用

享元模式在编辑器软件中大量使用，如在一个文档中多次出现相同的图片，则只需要创建一个图片对象，通过在应用程序中设置该图片出现的位置，可以实现该图片在不同地方多次重复显示。

5.12. 模式扩展

单纯享元模式和复合享元模式

- 单纯享元模式：在单纯享元模式中，所有的享元对象都是可以共享的，即所有抽象享元类的子类都可共享，不存在非共享具体享元类。
- 复合享元模式：将一些单纯享元使用组合模式加以组合，可以形成复合享元对象，这样的复合享元对象本身不能共享，但是它们可以分解成单纯享元对象，而后者则可以共享。 享元模式与其他模式的联用
- 在享元模式的享元工厂类中通常提供一个静态的工厂方法用于返回享元对象，使用简单工厂模式来生成享元对象。
- 在一个系统中，通常只有唯一一个享元工厂，因此享元工厂类可以使用单例模式进行设计。
- 享元模式可以结合组合模式形成复合享元模式，统一对享元对象设置外部状态。

5.13. 总结

- 享元模式运用共享技术有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用，它是一种对象结构型模式。
- 享元模式包含四个角色：抽象享元类声明一个接口，通过它可以接受并作用于外部状态；具体享元类实现了抽象享元接口，其实例称为享元对象；非共享具体享元是不能被共享的抽象享元类的子类；享元工厂类用于创建并管理享元对象，它针对抽象享元类编程，将各种类型的具体享元对象存储在一个享元池中。
- 享元模式以共享的方式高效地支持大量的细粒度对象，享元对象能做到共享的关键是区分内部状态和外部状态。其中内部状态是存储在享元对象内部并且不会随环境改变而改变的状态，因此内部状态可以共享；外部状态是随环境改变而改变的、不可以共享的状态。
- 享元模式主要优点在于它可以极大减少内存中对象的数量，使得相同对象或相似对象在内存中只保存一份；其缺点是使得系统更加复杂，并且需要将享元对象的状态外部化，而读取外部状态使得运行时间变长。
- 享元模式适用情况包括：一个系统有大量相同或者相似的对象，由于这类对象的大量使用，造成内存的大量耗费；对象的大部分状态都可以外部化，可以将这些外部状态传入对象中；多次重复使用享元对象。

6. 代理模式

6.1. 模式动机

在某些情况下，一个客户不想或者不能直接引用一个对象，此时可以通过一个称之为“代理”的第三者来实现间接引用。代理对象可以在客户端和目标对象之间起到中介的作用，并且可以通过代理对象去掉客户不能看到的内容和服务或者添加客户需要的额外服务。

通过引入一个新的对象（如小图片和远程代理对象）来实现对真实对象的操作或者将新的对象作为真实对象的一个替身，这种实现机制即为代理模式，通过引入代理对象来间接访问一个对象，这就是代理模式的模式动机。

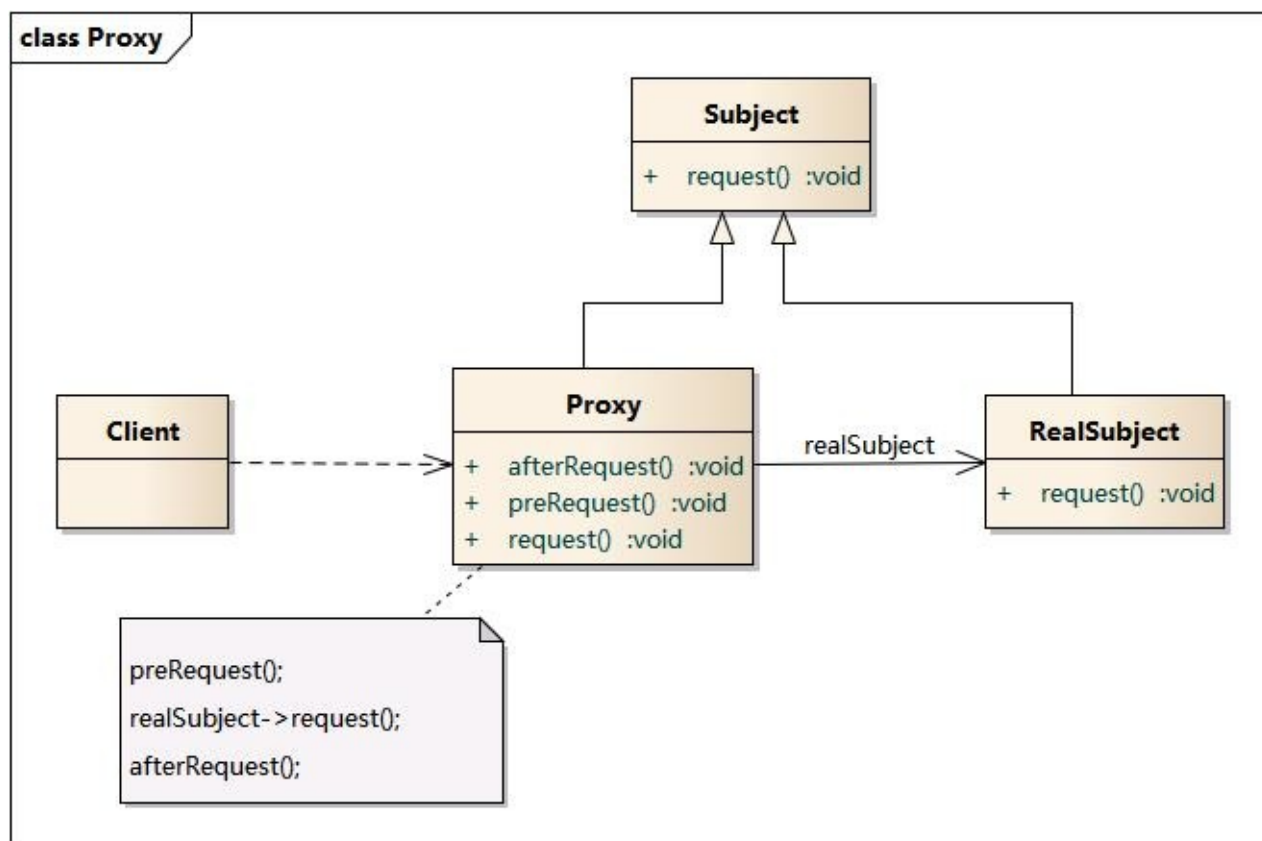
6.2. 模式定义

代理模式(Proxy Pattern)：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫做Proxy或Surrogate，它是一种对象结构型模式。

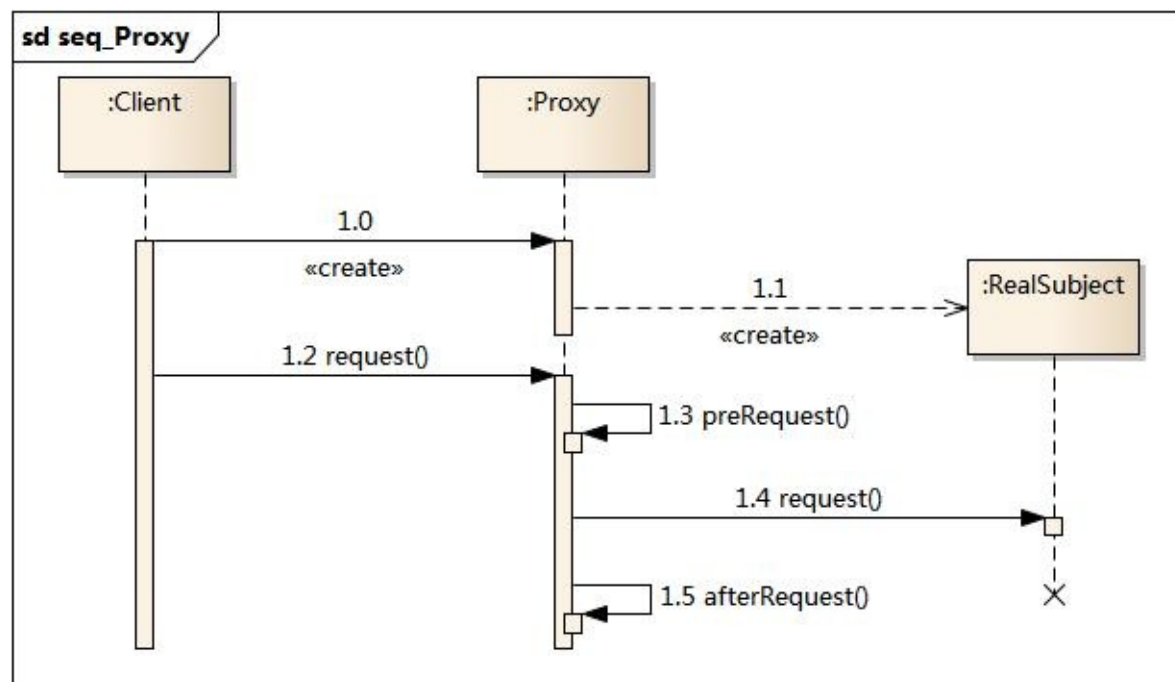
6.3. 模式结构

代理模式包含如下角色：

- Subject：抽象主题角色
- Proxy：代理主题角色
- RealSubject：真实主题角色



6.4. 时序图



6.5. 代码分析

```

1. #include <iostream>
2. #include "RealSubject.h"
3. #include "Proxy.h"
4.
5. using namespace std;
6.
7. int main(int argc, char *argv[])
8. {
9.     Proxy proxy;
10.    proxy.request();
11.
12.    return 0;
13. }

```

```

1. //////////////////////////////////////
2. // Proxy.h
3. // Implementation of the Class Proxy
4. // Created on:      07-十月-2014 16:57:54
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #if !defined(EA_56011290_0413_40c6_9132_63EE89B023FD__INCLUDED_)
9. #define EA_56011290_0413_40c6_9132_63EE89B023FD__INCLUDED_
10.
11. #include "RealSubject.h"
12. #include "Subject.h"

```

6. 代理模式

```
13.
14. class Proxy : public Subject
15. {
16.
17. public:
18.     Proxy();
19.     virtual ~Proxy();
20.
21.     void request();
22.
23. private:
24.     void afterRequest();
25.     void preRequest();
26.     RealSubject *m_pRealSubject;
27.
28. };
29. #endif // !defined(EA_56011290_0413_40c6_9132_63EE89B023FD__INCLUDED_)
```

```
1. //////////////////////////////////////
2. // Proxy.cpp
3. // Implementation of the Class Proxy
4. // Created on:      07-十月-2014 16:57:54
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #include "Proxy.h"
9. #include <iostream>
10. using namespace std;
11.
12.
13. Proxy::Proxy(){
14.     //有人觉得 RealSubject对象的创建应该是在main中实现；我认为RealSubject应该
15.     //对用户是透明的，用户所面对的接口都是通过代理的；这样才是真正的代理；
16.     m_pRealSubject = new RealSubject();
17. }
18.
19. Proxy::~~Proxy(){
20.     delete m_pRealSubject;
21. }
22.
23. void Proxy::afterRequest(){
24.     cout << "Proxy::afterRequest" << endl;
25. }
26.
27.
28. void Proxy::preRequest(){
29.     cout << "Proxy::preRequest" << endl;
30. }
31.
32.
33. void Proxy::request(){
34.     preRequest();
```

```
35.     m_pRealSubject->request();
36.     afterRequest();
37. }
```

运行结果：



6.6. 模式分析

6.7. 实例

6.8. 优点

代理模式的优点

- 代理模式能够协调调用者和被调用者，在一定程度上降低了系统的耦合度。
- 远程代理使得客户端可以访问在远程机器上的对象，远程机器可能具有更好的计算性能与处理速度，可以快速响应并处理客户端请求。
- 虚拟代理通过使用一个小对象来代表一个大对象，可以减少系统资源的消耗，对系统进行优化并提高运行速度。
- 保护代理可以控制对真实对象的使用权限。

6.9. 缺点

代理模式的缺点

- 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。
- 实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

6.10. 适用环境

根据代理模式的使用目的，常见的代理模式有以下几种类型：

- 远程(Remote)代理：为一个位于不同的地址空间的对象提供一个本地的代理对象，这个不同的地址空间可以是在同一台主机中，也可是在另一台主机中，远程代理又叫做大使(Ambassador)。
- 虚拟(Virtual)代理：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建。
- Copy-on-Write代理：它是虚拟代理的一种，把复制(克隆)操作延迟到只有在客户端真正需要时才执行。一般来说，对象的深克隆是一个开销较大的操作，Copy-on-Write代理可以让这个操作延迟，只有对象被用

到的时候才被克隆。

- 保护(Protect or Access)代理：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限。
- 缓冲(Cache)代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
- 防火墙(Firewall)代理：保护目标不让恶意用户接近。
- 同步化(Synchronization)代理：使几个用户能够同时使用一个对象而没有冲突。
- 智能引用(Smart Reference)代理：当一个对象被引用时，提供一些额外的操作，如将此对象被调用的次数记录下来等。

6.11. 模式应用

EJB、Web Service等分布式技术都是代理模式的应用。在EJB中使用了RMI机制，远程服务器中的企业级Bean在本地有一个桩代理，客户端通过桩来调用远程对象中定义的方法，而无须直接与远程对象交互。在EJB的使用中需要提供一个公共的接口，客户端针对该接口进行编程，无须知道桩以及远程EJB的实现细节。

6.12. 模式扩展

几种常用的代理模式

- 图片代理：一个很常见的代理模式的应用实例就是对大图浏览的控制。
- 用户通过浏览器访问网页时先不加载真实的大图，而是通过代理对象的方法来进行处理，在代理对象的方法中，先使用一个线程向客户端浏览器加载一个小图片，然后在后台使用另一个线程来调用大图片的加载方法将大图片加载到客户端。当需要浏览大图片时，再将大图片在新网页中显示。如果用户在浏览大图时加载工作还没有完成，可以再启动一个线程来显示相应的提示信息。通过代理技术结合多线程编程将真实图片的加载放到后台来操作，不影响前台图片的浏览。
- 远程代理：远程代理可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。客户完全可以认为被代理的远程业务对象是局域的而不是远程的，而远程代理对象承担了大部分的网络通信工作。
- 虚拟代理：当一个对象的加载十分耗费资源的时候，虚拟代理的优势就非常明显地体现出来了。虚拟代理模式是一种内存节省技术，那些占用大量内存或处理复杂的对象将推迟到使用它的时候才创建。 - 在应用程序启动的时候，可以用代理对象代替真实对象初始化，节省了内存的占用，并大大加速了系统的启动时间。

动态代理

- 动态代理是一种较为高级的代理模式，它的典型应用就是Spring AOP。
- 在传统的代理模式中，客户端通过Proxy调用RealSubject类的request()方法，同时还在代理类中封装了其他方法(如preRequest()和postRequest())，可以处理一些其他问题。
- 如果按照这种方法使用代理模式，那么真实主题角色必须是事先已经存在的，并将其作为代理对象的内部成员属性。如果一个真实主题角色必须对应一个代理主题角色，这将导致系统中的类个数急剧增加，因此需要想办法减少系统中类的个数，此外，如何在事先不知道真实主题角色的情况下使用代理主题角色，这都是动态代理需要解决的问题。

6.13. 总结

在代理模式中，要求给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫做Proxy或Surrogate，它是一种对象结构型模式。 - 代理模式包含三个角色：抽象主题角色声明了真实主题和代理主题的共同接口；代理主题角色内部包含对真实主题的引用，从而可以在任何时候操作真实主题对象；真实主题角色定义了代理

角色所代表的真实对象，在真实主题角色中实现了真实的业务操作，客户端可以通过代理主题角色间接调用真实主题角色中定义的方法。 - 代理模式的优点在于能够协调调用者和被调用者，在一定程度上降低了系统的耦合度；其缺点在于由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢，并且实现代理模式需要额外的工作，有些代理模式的实现非常复杂。远程代理为一个位于不同的地址空间的对象提供一个本地的代表对象，它使得客户端可以访问在远程机器上的对象，远程机器可能具有更好的计算性能与处理速度，可以快速响应并处理客户端请求。 - 如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建，这个小对象称为虚拟代理。虚拟代理通过使用一个小对象来代表一个大对象，可以减少系统资源的消耗，对系统进行优化并提高运行速度。 - 保护代理可以控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限。

行为型模式

行为型模式(Behavioral Pattern)是对在不同的对象之间划分责任和算法的抽象化。

行为型模式不仅仅关注类和对象的结构，而且重点关注它们之间的相互作用。

通过行为型模式，可以更加清晰地划分类与对象的职责，并研究系统在运行时实例对象之间的交互。在系统运行时，对象并不是孤立的，它们可以通过相互通信与协作完成某些复杂功能，一个对象在运行时也将影响到其他对象的运行。

行为型模式分为类行为型模式和对象行为型模式两种：

- 类行为型模式：类的行为型模式使用继承关系在几个类之间分配行为，类行为型模式主要通过多态等方式来分配父类与子类的职责。
- 对象行为型模式：对象的行为型模式则使用对象的聚合关联关系来分配行为，对象行为型模式主要是通过对象关联等方式来分配两个或多个类的职责。根据“合成复用原则”，系统中要尽量使用关联关系来取代继承关系，因此大部分行为型设计模式都属于对象行为型设计模式。包含模式
 - - 职责链模式(Chain of Responsibility)
 - 重要程度：3
 - - 命令模式(Command)
 - 重要程度：4
 - - 解释器模式(Interpreter)
 - 重要程度：1
 - - 迭代器模式(Iterator)
 - 重要程度：5
 - - 中介者模式(Mediator)
 - 重要程度：2
 - - 备忘录模式(Memento)
 - 重要程度：2
 - - 观察者模式(Observer)
 - 重要程度：5
 - - 状态模式(State)
 - 重要程度：3
 - - 策略模式(Strategy)
 - 重要程度：4

- - 模板方法模式(Template Method)
 - 重要程度：3
- - 访问者模式(Visitor)
 - 重要程度：1目录
- 1. 命令模式
 - 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. 优点
 - 1.9. 缺点
 - 1.10. 适用环境
 - 1.11. 模式应用
 - 1.12. 模式扩展
 - 1.13. 总结
- 2. 中介者模式
 - 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. 优点
 - 2.9. 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. 观察者模式
 - 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. 状态模式
 - 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. 策略模式
 - 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. 命令模式

1.1. 模式动机

在软件设计中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，我们只需在程序运行时指定具体的请求接收者即可，此时，可以使用命令模式来进行设计，使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

命令模式可以对发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。这就是命令模式的模式动机。

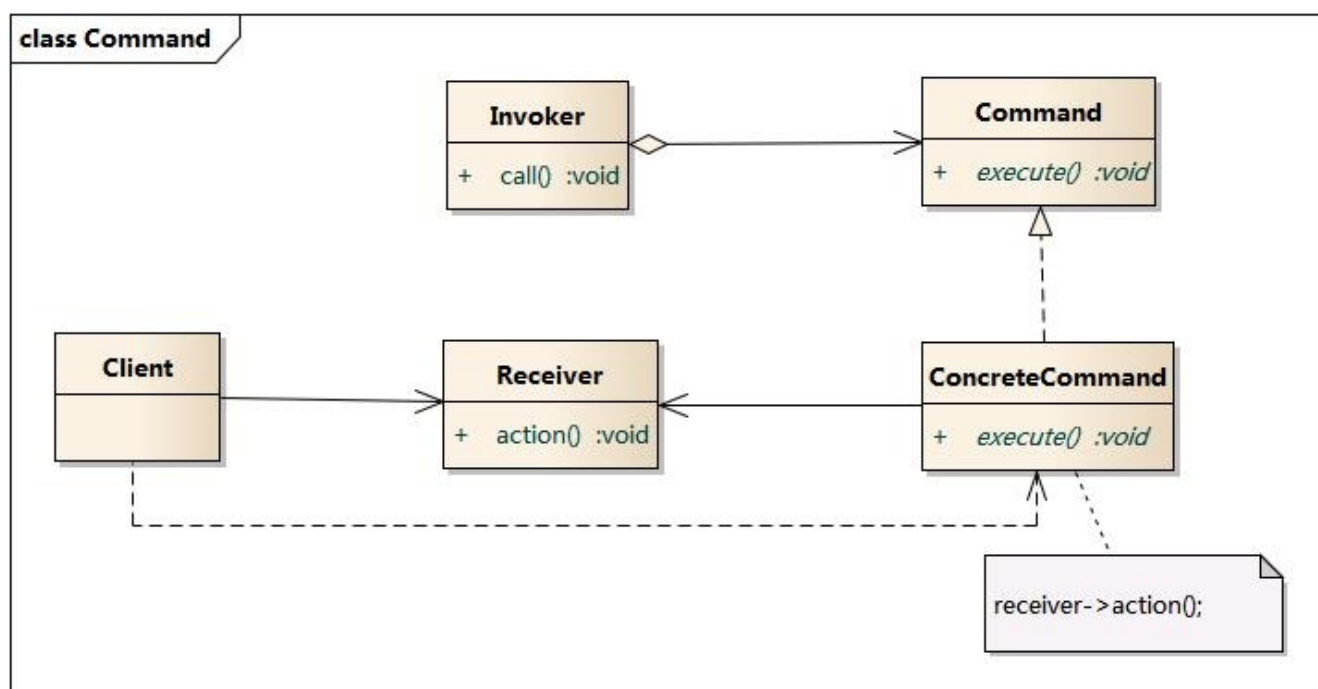
1.2. 模式定义

命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。

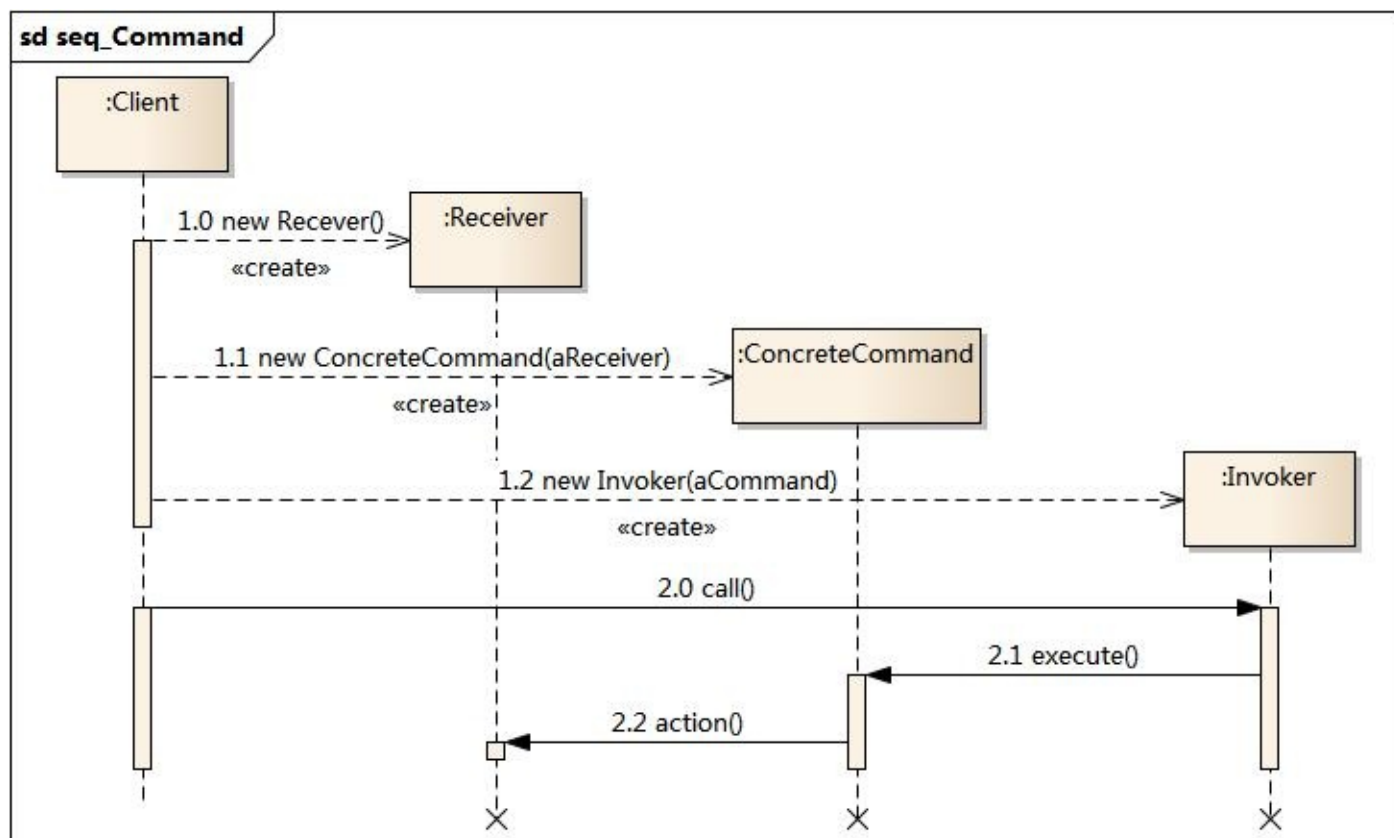
1.3. 模式结构

命令模式包含如下角色：

- Command：抽象命令类
- ConcreteCommand：具体命令类
- Invoker：调用者
- Receiver：接收者
- Client：客户类



1.4. 时序图



1.5. 代码分析

```

1. #include <iostream>
2. #include "ConcreteCommand.h"
3. #include "Invoker.h"
4. #include "Receiver.h"
5.
6. using namespace std;
7.
8. int main(int argc, char *argv[])
9. {
10.     Receiver * pReceiver = new Receiver();
11.     ConcreteCommand * pCommand = new ConcreteCommand(pReceiver);
12.     Invoker * pInvoker = new Invoker(pCommand);
13.     pInvoker->call();
14.
15.     delete pReceiver;
16.     delete pCommand;
17.     delete pInvoker;
18.     return 0;
19. }
  
```

```

1. //////////////////////////////////////
2. // Receiver.h
  
```

1. 命令模式

```
3. // Implementation of the Class Receiver
4. // Created on:      07-十月-2014 17:44:02
5. // Original author: colin
6. ///////////////////////////////////////////////////////////////////
7.
8. #if !defined(EA_8E5430BB_0904_4a7d_9A3B_7169586237C8__INCLUDED_)
9. #define EA_8E5430BB_0904_4a7d_9A3B_7169586237C8__INCLUDED_
10.
11. class Receiver
12. {
13.
14. public:
15.     Receiver();
16.     virtual ~Receiver();
17.
18.     void action();
19.
20. };
21. #endif // !defined(EA_8E5430BB_0904_4a7d_9A3B_7169586237C8__INCLUDED_)
```

```
1. ///////////////////////////////////////////////////////////////////
2. // Receiver.cpp
3. // Implementation of the Class Receiver
4. // Created on:      07-十月-2014 17:44:02
5. // Original author: colin
6. ///////////////////////////////////////////////////////////////////
7.
8. #include "Receiver.h"
9. #include <iostream>
10. using namespace std;
11.
12. Receiver::Receiver(){
13.
14. }
15.
16. Receiver::~~Receiver(){
17.
18. }
19.
20. void Receiver::action(){
21.     cout << "receiver action." << endl;
22. }
```

```
1. ///////////////////////////////////////////////////////////////////
2. // ConcreteCommand.h
3. // Implementation of the Class ConcreteCommand
4. // Created on:      07-十月-2014 17:44:01
5. // Original author: colin
6. ///////////////////////////////////////////////////////////////////
7.
8. #if !defined(EA_1AE70D53_4868_4e81_A1B8_1088DA355C23__INCLUDED_)
```

1. 命令模式

```
9. #define EA_1AE70D53_4868_4e81_A1B8_1088DA355C23__INCLUDED_
10.
11. #include "Command.h"
12. #include "Receiver.h"
13.
14. class ConcreteCommand : public Command
15. {
16.
17. public:
18.     ConcreteCommand(Receiver * pReceiver);
19.     virtual ~ConcreteCommand();
20.     virtual void execute();
21. private:
22.     Receiver *m_pReceiver;
23.
24.
25.
26. };
27. #endif // !defined(EA_1AE70D53_4868_4e81_A1B8_1088DA355C23__INCLUDED_)
```

```
1. //////////////////////////////////////
2. // ConcreteCommand.cpp
3. // Implementation of the Class ConcreteCommand
4. // Created on:      07-十月-2014 17:44:02
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #include "ConcreteCommand.h"
9. #include <iostream>
10. using namespace std;
11.
12.
13. ConcreteCommand::ConcreteCommand(Receiver *pReceiver){
14.     m_pReceiver = pReceiver;
15. }
16.
17.
18.
19. ConcreteCommand::~~ConcreteCommand(){
20.
21. }
22.
23. void ConcreteCommand::execute(){
24.     cout << "ConcreteCommand::execute" << endl;
25.     m_pReceiver->action();
26. }
```

```
1. //////////////////////////////////////
2. // Invoker.h
3. // Implementation of the Class Invoker
4. // Created on:      07-十月-2014 17:44:02
```

1. 命令模式

```
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #if !defined(EA_3DACB62A_0813_4d11_8A82_10BF1FB00D9A__INCLUDED_)
9. #define EA_3DACB62A_0813_4d11_8A82_10BF1FB00D9A__INCLUDED_
10.
11. #include "Command.h"
12.
13. class Invoker
14. {
15.
16. public:
17.     Invoker(Command * pCommand);
18.     virtual ~Invoker();
19.     void call();
20.
21. private:
22.     Command *m_pCommand;
23.
24.
25. };
26. #endif // !defined(EA_3DACB62A_0813_4d11_8A82_10BF1FB00D9A__INCLUDED_)
```

```
1. //////////////////////////////////////
2. // Invoker.cpp
3. // Implementation of the Class Invoker
4. // Created on:      07-十月-2014 17:44:02
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #include "Invoker.h"
9. #include <iostream>
10. using namespace std;
11.
12. Invoker::Invoker(Command * pCommand){
13.     m_pCommand = pCommand;
14. }
15.
16. Invoker::~Invoker(){
17.
18. }
19.
20. void Invoker::call(){
21.     cout << "invoker calling" << endl;
22.     m_pCommand->execute();
23. }
```

运行结果：

```
"C:\Users\cl\Documents\GitHub\design_patterns\code\Command\mingw5\main.exe"
invoker calling
ConcreteCommand::execute
receiver action.
请按任意键继续. . .
```

1.6. 模式分析

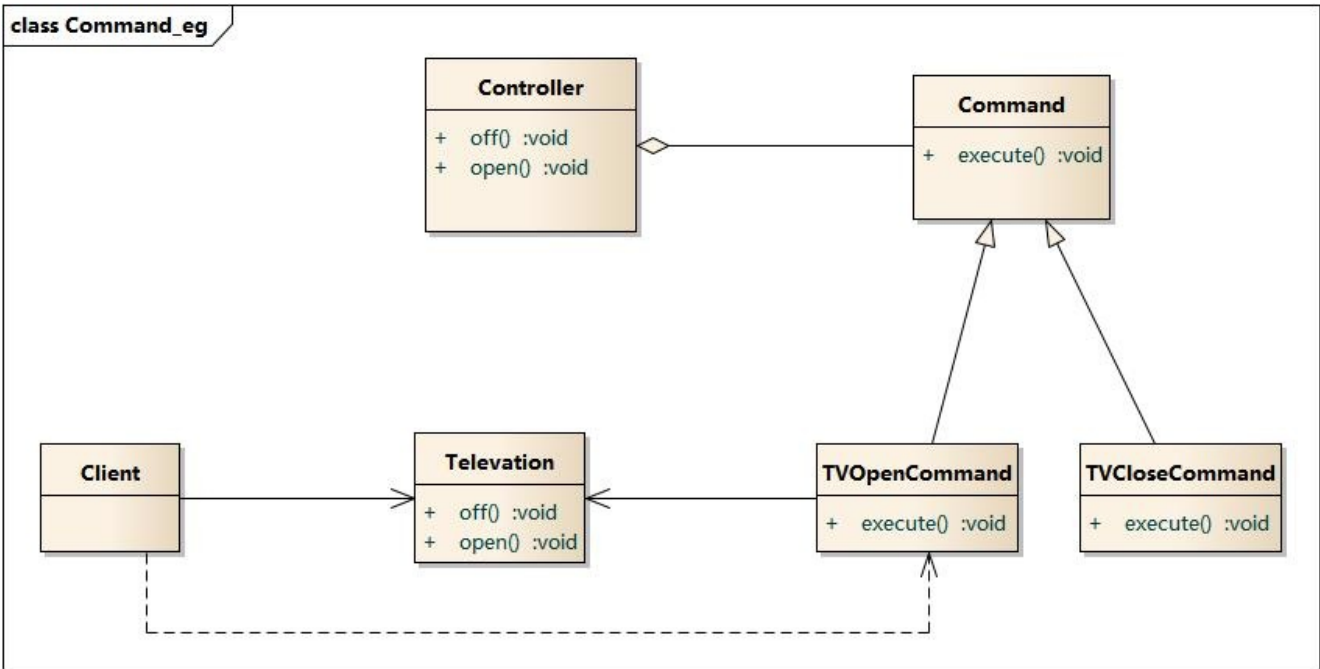
命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。

- 每一个命令都是一个操作：请求的一方发出请求，要求执行一个操作；接收的一方收到请求，并执行操作。
- 命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。
- 命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。
- 命令模式的关键在于引入了抽象命令接口，且发送者针对抽象命令接口编程，只有实现了抽象命令接口的具体命令才能与接收者相关联。

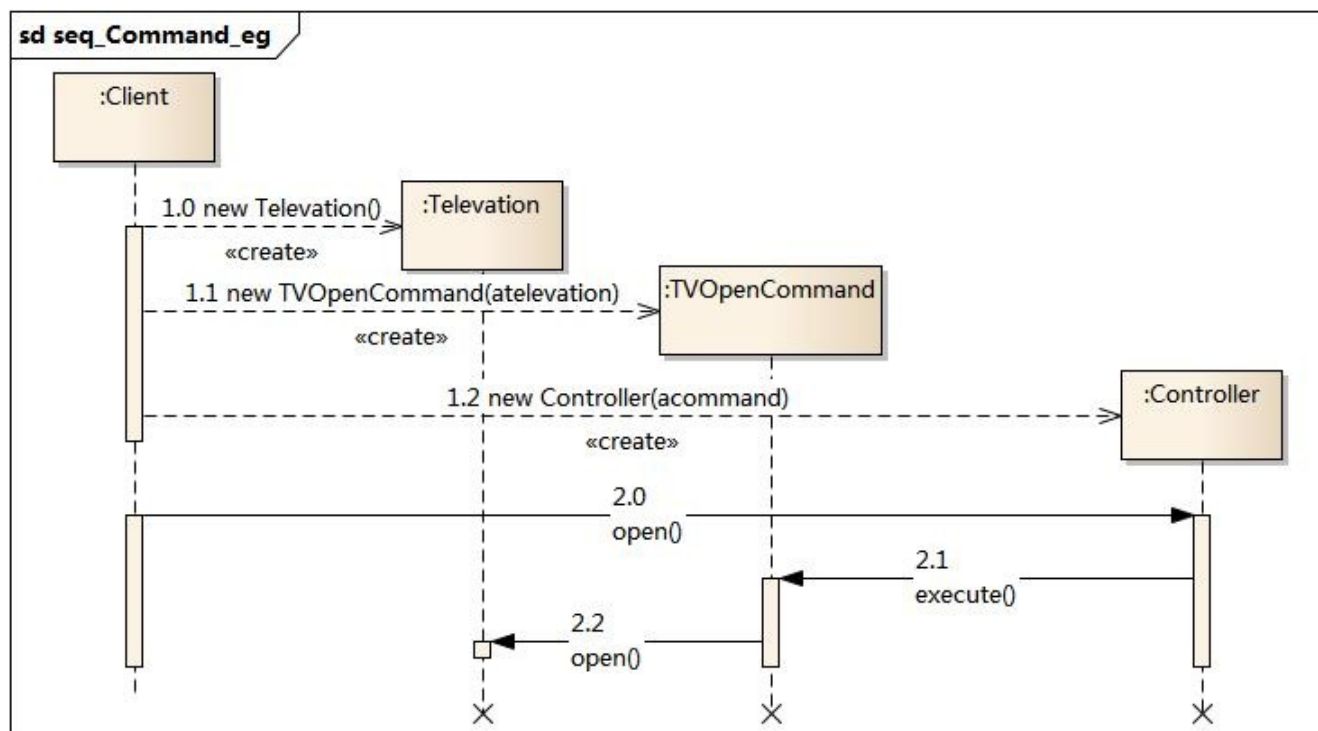
1.7. 实例

实例一：电视机遥控器

- 电视机是请求的接收者，遥控器是请求的发送者，遥控器上有一些按钮，不同的按钮对应电视机的不同操作。抽象命令角色由一个命令接口来扮演，有三个具体的命令类实现了抽象命令接口，这三个具体命令类分别代表三种操作：打开电视机、关闭电视机和切换频道。显然，电视机遥控器就是一个典型的命令模式应用实例。



时序图：



1.8. 优点

命令模式的优点

- 降低系统的耦合度。
- 新的命令可以很容易地加入到系统中。
- 可以比较容易地设计一个命令队列和宏命令（组合命令）。
- 可以方便地实现对请求的Undo和Redo。

1.9. 缺点

命令模式的缺点

- 使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。

1.10. 适用环境

在以下情况下可以使用命令模式：

- 系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。
- 系统需要在不同的时间指定请求、将请求排队和执行请求。
- 系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。
- 系统需要将一组操作组合在一起，即支持宏命令

1.11. 模式应用

很多系统都提供了宏命令功能，如UNIX平台下的Shell编程，可以将多条命令封装在一个命令对象中，只需要一条简单的命令即可执行一个命令序列，这也是命令模式的应用实例之一。

1.12. 模式扩展

宏命令又称为组合命令，它是命令模式和组合模式联用的产物。

-宏命令也是一个具体命令，不过它包含了对其他命令对象的引用，在调用宏命令的execute()方法时，将递归调用它所包含的每个成员命令的execute()方法，一个宏命令的成员对象可以是简单命令，还可以继续是宏命令。执行一个宏命令将执行多个具体命令，从而实现对命令的批处理。

1.13. 总结

- 在命令模式中，将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作模式或事务模式。
- 命令模式包含四个角色：抽象命令类中声明了用于执行请求的execute()等方法，通过这些方法可以调用请求接收者的相关操作；具体命令类是抽象命令类的子类，实现了在抽象命令类中声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中；调用者即请求的发送者，又称为请求者，它通过命令对象来执行请求；接收者执行与请求相关的操作，它具体实现对请求的业务处理。
- 命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。
- 命令模式的主要优点在于降低系统的耦合度，增加新的命令很方便，而且可以比较容易地设计一个命令队列和宏命令，并方便地实现对请求的撤销和恢复；其主要缺点在于可能会导致某些系统有过多的具体命令类。
- 命令模式适用情况包括：需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互；需要在不同的时间指定请求、将请求排队和执行请求；需要支持命令的撤销操作和恢复操作，需要将一组操作组合在一起，即支持宏命令。

2. 中介者模式

2.1. 模式动机

- 在用户与用户直接聊天的设计方案中，用户对象之间存在很强的关联性，将导致系统出现如下问题：
- 系统结构复杂：对象之间存在大量的相互关联和调用，若有一个对象发生变化，则需要跟踪和该对象关联的其他所有对象，并进行适当处理。
- 对象可重用性差：由于一个对象和其他对象具有很强的关联，若没有其他对象的支持，一个对象很难被另一个系统或模块重用，这些对象表现出来更像一个不可分割的整体，职责较为混乱。
- 系统扩展性低：增加一个新的对象需要在原有相关对象上增加引用，增加新的引用关系也需要调整原有对象，系统耦合度很高，对象操作很不灵活，扩展性差。
- 在面向对象的软件设计与开发过程中，根据“单一职责原则”，我们应该尽量将对象细化，使其只负责或呈现单一的职责。
- 对于一个模块，可能由很多对象构成，而且这些对象之间可能存在相互的引用，为了减少对象两两之间复杂的引用关系，使之成为一个松耦合的系统，我们需要使用中介者模式，这就是中介者模式的模式动机。

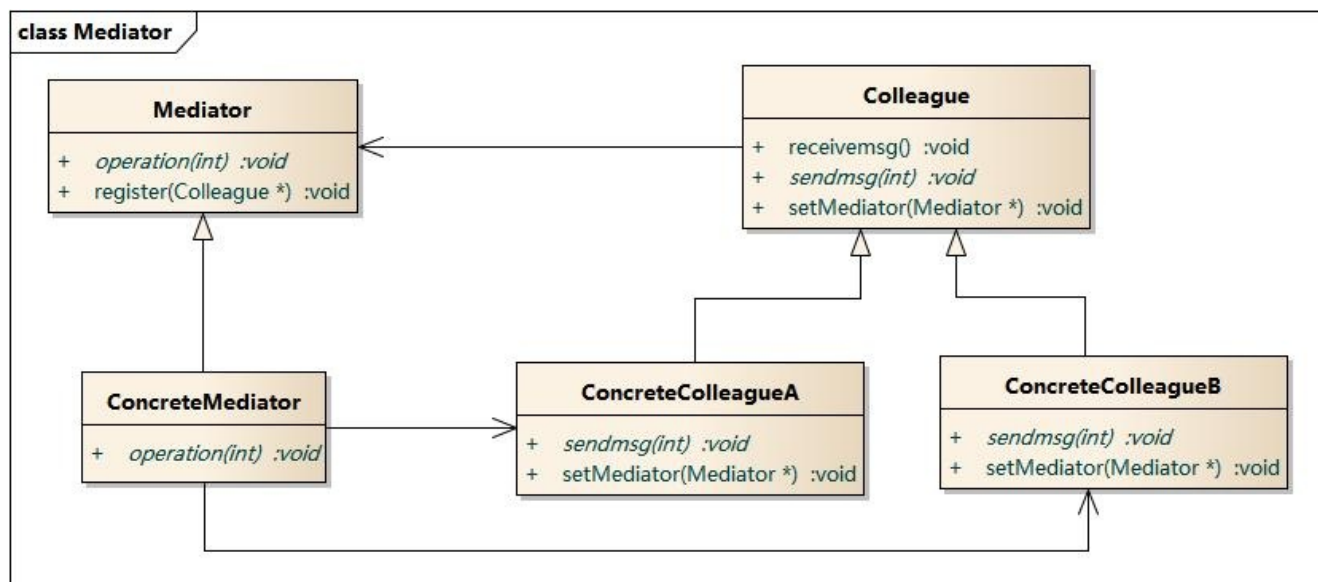
2.2. 模式定义

中介者模式(Mediator Pattern)定义：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。

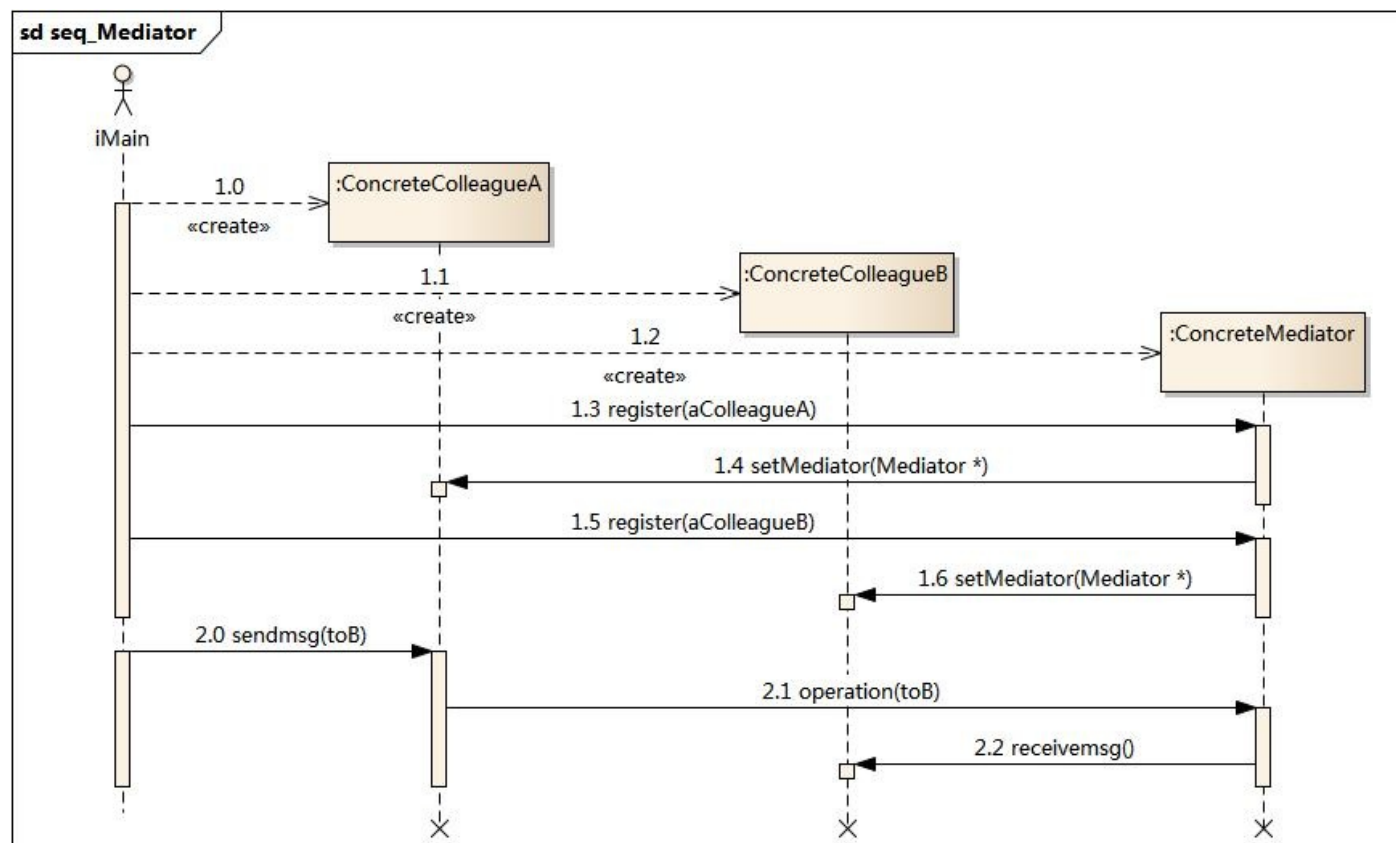
2.3. 模式结构

中介者模式包含如下角色：

- Mediator：抽象中介者
- ConcreteMediator：具体中介者
- Colleague：抽象同事类
- ConcreteColleague：具体同事类



2.4. 时序图



2.5. 代码分析

```

1. #include <iostream>
2. #include "ConcreteColleagueA.h"
3. #include "ConcreteMediator.h"
4. #include "ConcreteColleagueB.h"
5.
6. using namespace std;
  
```

2. 中介者模式

```
7.
8. int main(int argc, char *argv[])
9. {
10.     ConcreteColleagueA * pa = new ConcreteColleagueA();
11.     ConcreteColleagueB * pb = new ConcreteColleagueB();
12.     ConcreteMediator * pm = new ConcreteMediator();
13.     pm->registered(1,pa);
14.     pm->registered(2,pb);
15.
16.     // sendmsg from a to b
17.     pa->sendmsg(2,"hello,i am a");
18.     // sendmsg from b to a
19.     pb->sendmsg(1,"hello,i am b");
20.
21.     delete pa,pb,pm;
22.     return 0;
23. }
```

```
1. //////////////////////////////////////
2. // ConcreteMediator.h
3. // Implementation of the Class ConcreteMediator
4. // Created on:      07-十月-2014 21:30:47
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #if !defined(EA_8CECE546_61DD_456f_A3E7_D98BC078D8E8__INCLUDED_)
9. #define EA_8CECE546_61DD_456f_A3E7_D98BC078D8E8__INCLUDED_
10.
11. #include "ConcreteColleagueB.h"
12. #include "Mediator.h"
13. #include "ConcreteColleagueA.h"
14. #include <map>
15. using namespace std;
16. class ConcreteMediator : public Mediator
17. {
18.
19. public:
20.     ConcreteMediator();
21.     virtual ~ConcreteMediator();
22.
23.     virtual void operation(int nWho,string str);
24.     virtual void registered(int nWho, Colleague * aColleague);
25. private:
26.     map<int,Colleague*> m_mpColleague;
27. };
28. #endif // !defined(EA_8CECE546_61DD_456f_A3E7_D98BC078D8E8__INCLUDED_)
```

```
1. //////////////////////////////////////
2. // ConcreteMediator.cpp
3. // Implementation of the Class ConcreteMediator
4. // Created on:      07-十月-2014 21:30:48
```

```

5. // Original author: colin
6. //////////////////////////////////////////////////
7.
8. #include "ConcreteMediator.h"
9. #include <map>
10. #include <iostream>
11. using namespace std;
12.
13. ConcreteMediator::ConcreteMediator(){
14.
15. }
16.
17. ConcreteMediator::~~ConcreteMediator(){
18.
19. }
20.
21. void ConcreteMediator::operation(int nWho, string str){
22.     map<int, Colleague*>::const_iterator itr = m_mpColleague.find(nWho);
23.     if(itr == m_mpColleague.end())
24.     {
25.         cout << "not found this colleague!" << endl;
26.         return;
27.     }
28.
29.     Colleague* pc = itr->second;
30.     pc->receivemsg(str);
31. }
32.
33.
34. void ConcreteMediator::registered(int nWho, Colleague * aColleague){
35.     map<int, Colleague*>::const_iterator itr = m_mpColleague.find(nWho);
36.     if(itr == m_mpColleague.end())
37.     {
38.         m_mpColleague.insert(make_pair(nWho, aColleague));
39.         //同时将中介类暴露给colleague
40.         aColleague->setMediator(this);
41.     }
42. }

```

```

1. //////////////////////////////////////////////////
2. // ConcreteColleagueA.h
3. // Implementation of the Class ConcreteColleagueA
4. // Created on:      07-十月-2014 21:30:47
5. // Original author: colin
6. //////////////////////////////////////////////////
7.
8. #if !defined(EA_79979DD4_1E73_46db_A635_E3F516ACCE0A__INCLUDED_)
9. #define EA_79979DD4_1E73_46db_A635_E3F516ACCE0A__INCLUDED_
10.
11. #include "Colleague.h"
12.
13. class ConcreteColleagueA : public Colleague

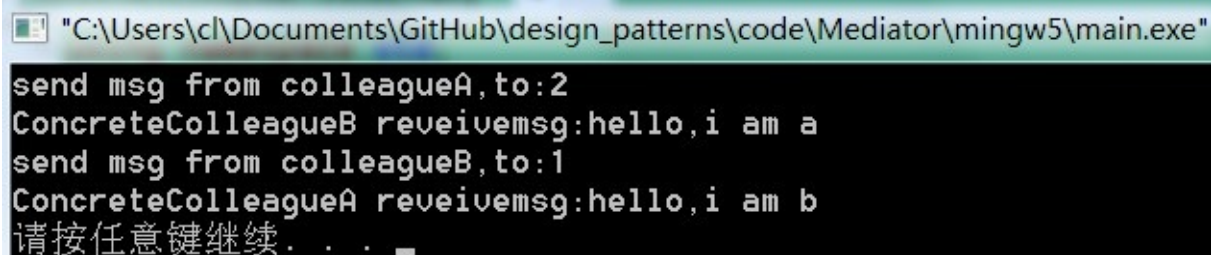
```

2. 中介者模式

```
14. {
15.
16. public:
17.     ConcreteColleagueA();
18.     virtual ~ConcreteColleagueA();
19.
20.     virtual void sendmsg(int toWho,string str);
21.     virtual void receivemsg(string str);
22.
23. };
24. #endif // !defined(EA_79979DD4_1E73_46db_A635_E3F516ACCE0A__INCLUDED_)
```

```
1. //////////////////////////////////
2. //  ConcreteColleagueA.cpp
3. //  Implementation of the Class ConcreteColleagueA
4. //  Created on:      07-十月-2014 21:30:47
5. //  Original author: colin
6. //////////////////////////////////
7.
8. #include "ConcreteColleagueA.h"
9. #include <iostream>
10. using namespace std;
11.
12. ConcreteColleagueA::ConcreteColleagueA(){
13. }
14.
15. ConcreteColleagueA::~~ConcreteColleagueA(){
16. }
17.
18. void ConcreteColleagueA::sendmsg(int toWho,string str){
19.     cout << "send msg from colleagueA,to:" << toWho << endl;
20.     m_pMediator->operation(toWho,str);
21. }
22.
23. void ConcreteColleagueA::receivemsg(string str){
24.     cout << "ConcreteColleagueA reveivemsg:" << str <<endl;
25. }
```

运行结果:



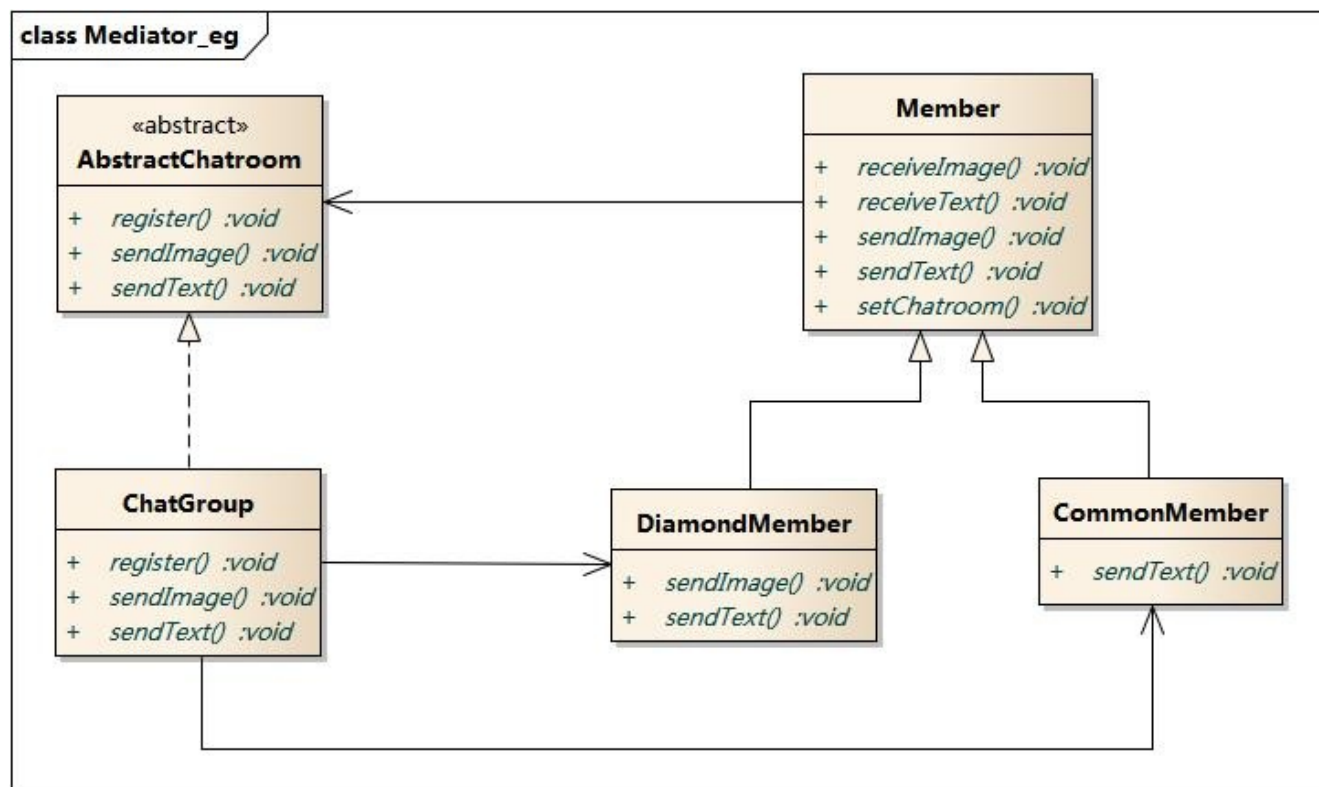
```
"C:\Users\cl\Documents\GitHub\design_patterns\code\Mediator\mingw5\main.exe"
send msg from colleagueA,to:2
ConcreteColleagueB reveivemsg:hello,i am a
send msg from colleagueB,to:1
ConcreteColleagueA reveivemsg:hello,i am b
请按任意键继续. . .
```

2.6. 模式分析

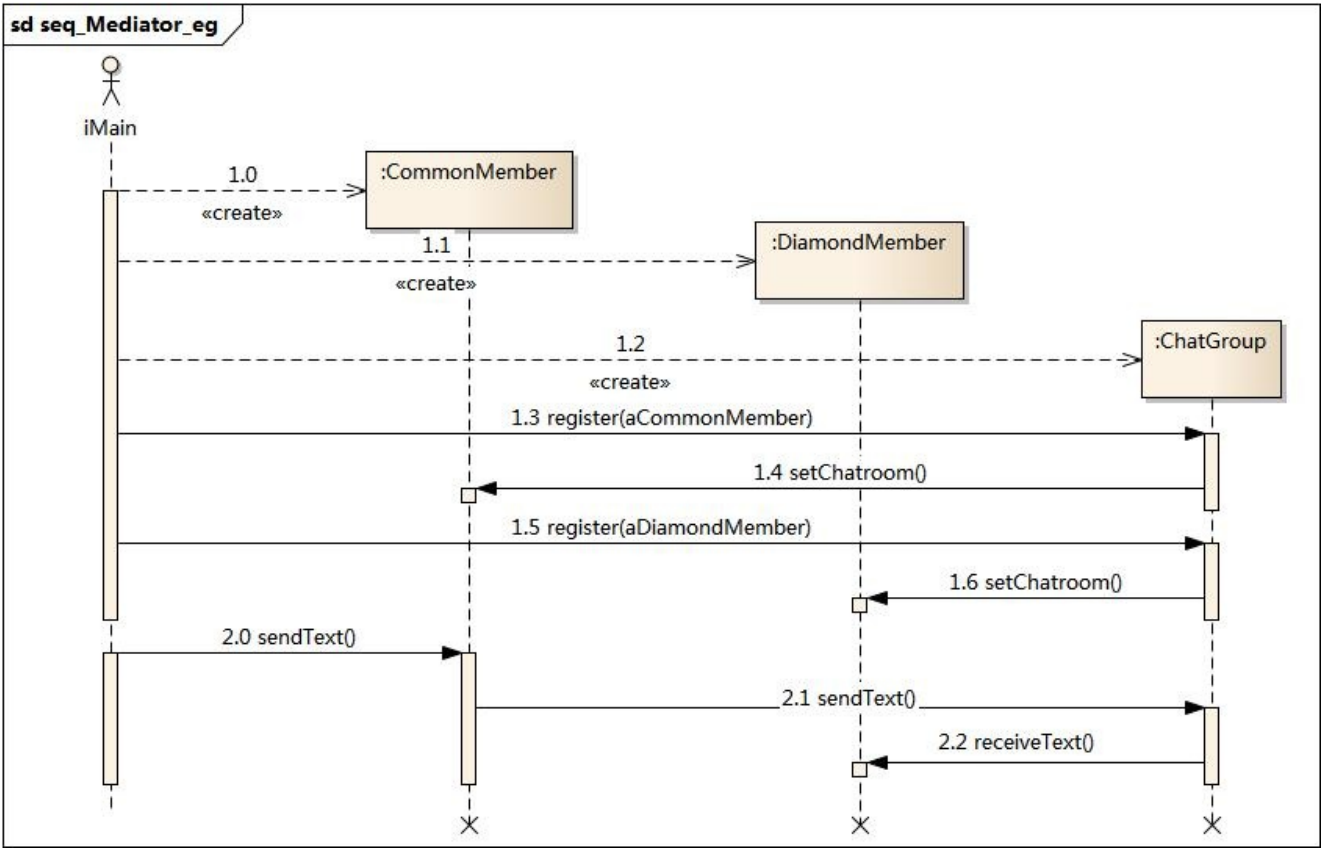
中介者模式可以使对象之间的关系数量急剧减少。

中介者承担两方面的职责：

- 中转作用（结构性）：通过中介者提供的中转作用，各个同事对象就不再需要显式引用其他同事，当需要和其他同事进行通信时，通过中介者即可。该中转作用属于中介者在结构上的支持。
- 协调作用（行为性）：中介者可以更进一步的对同事之间的关系进行封装，同事可以一致地和中介者进行交互，而不需要指明中介者需要具体怎么做，中介者根据封装在自身内部的协调逻辑，对同事的请求进行进一步处理，将同事成员之间的关系行为进行分离和封装。该协调作用属于中介者在行为上的支持。



时序图



2.7. 实例

实例：虚拟聊天室

某论坛系统欲增加一个虚拟聊天室，允许论坛会员通过该聊天室进行信息交流，普通会员 (CommonMember) 可以给其他会员发送文本信息，钻石会员 (DiamondMember) 既可以给其他会员发送文本信息，还可以发送图片信息。该聊天室可以对不雅字符进行过滤，如“日”等字符；还可以对发送的图片大小进行控制。用中介者模式设计该虚拟聊天室。

2.8. 优点

中介者模式的优点

- 简化了对象之间的交互。
- 将各同事解耦。
- 减少子类生成。
- 可以简化各同事类的设计和实现。

2.9. 缺点

中介者模式的缺点

- 在具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护。

2.10. 适用环境

在以下情况下可以使用中介者模式：

- 系统中对象之间存在复杂的引用关系，产生的相互依赖关系结构混乱且难以理解。
- 一个对象由于引用了其他很多对象并且直接和这些对象通信，导致难以复用该对象。
- 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。可以通过引入中介者类来实现，在中介者中定义对象。
- 交互的公共行为，如果需要改变行为则可以增加新的中介者类。

2.11. 模式应用

MVC架构中控制器

Controller 作为一种中介者，它负责控制视图对象View和模型对象Model之间的交互。如在Struts中，Action就可以作为JSP页面与业务对象之间的中介者。

2.12. 模式扩展

中介者模式与迪米特法则

- 在中介者模式中，通过创造出中介者对象，将系统中有关对象所引用的其他对象数目减少到最少，使得一个对象与其同事之间的相互作用被这个对象与中介者对象之间的相互作用所取代。因此，中介者模式就是迪米特法则的一个典型应用。 中介者模式与GUI开发
- 中介者模式可以方便地应用于图形界面(GUI)开发中，在比较复杂的界面中可能存在多个界面组件之间的交互关系。
- 对于这些复杂的交互关系，有时候我们可以引入一个中介者类，将这些交互的组件作为具体的同事类，将它们之间的引用和控制关系交由中介者负责，在一定程度上简化系统的交互，这也是中介者模式的常见应用之一。

2.13. 总结

- 中介者模式用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。
- 中介者模式包含四个角色：抽象中介者用于定义一个接口，该接口用于与各同事对象之间的通信；具体中介者是抽象中介者的子类，通过协调各个同事对象来实现协作行为，了解并维护它的各个同事对象的引用；抽象同事类定义各同事的公有方法；具体同事类是抽象同事类的子类，每一个同事对象都引用一个中介者对象；每一个同事对象在需要和其他同事对象通信时，先与中介者通信，通过中介者来间接完成与其他同事类的通信；在具体同事类中实现了在抽象同事类中定义的方法。
- 通过引入中介者对象，可以将系统的网状结构变成以中介者为中心的星形结构，中介者承担了中转作用和协调作用。中介者类是中介者模式的核心，它对整个系统进行控制和协调，简化了对象之间的交互，还可以对对象间的交互进行进一步的控制。
- 中介者模式的主要优点在于简化了对象之间的交互，将各同事解耦，还可以减少子类生成，对于复杂的对象之间的交互，通过引入中介者，可以简化各同事类的设计和实现；中介者模式主要缺点在于具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护。
- 中介者模式适用情况包括：系统中对象之间存在复杂的引用关系，产生的相互依赖关系结构混乱且难以理解；一

个对象由于引用了其他很多对象并且直接和这些对象通信，导致难以复用该对象；想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

3. 观察者模式

3.1. 模式动机

建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应做出反应。在此，发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展，这就是观察者模式的模式动机。

3.2. 模式定义

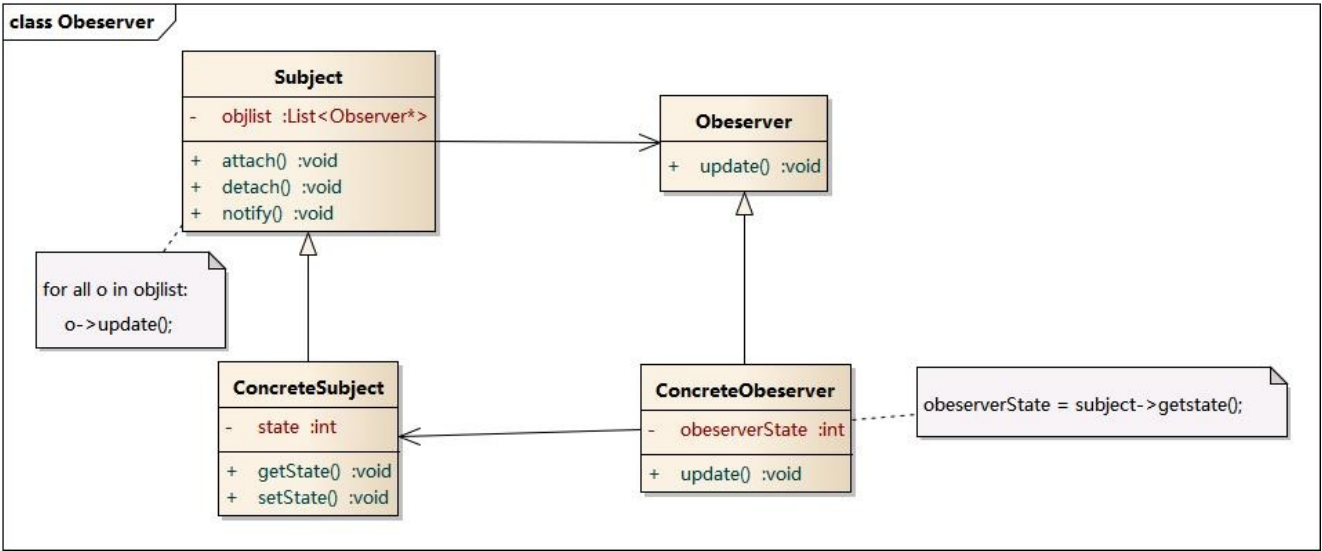
观察者模式(Observer Pattern)：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅(Publish/Subscribe)模式、模型-视图(Model/View)模式、源-监听器(Source/Listener)模式或从属者(Dependents)模式。

观察者模式是一种对象行为型模式。

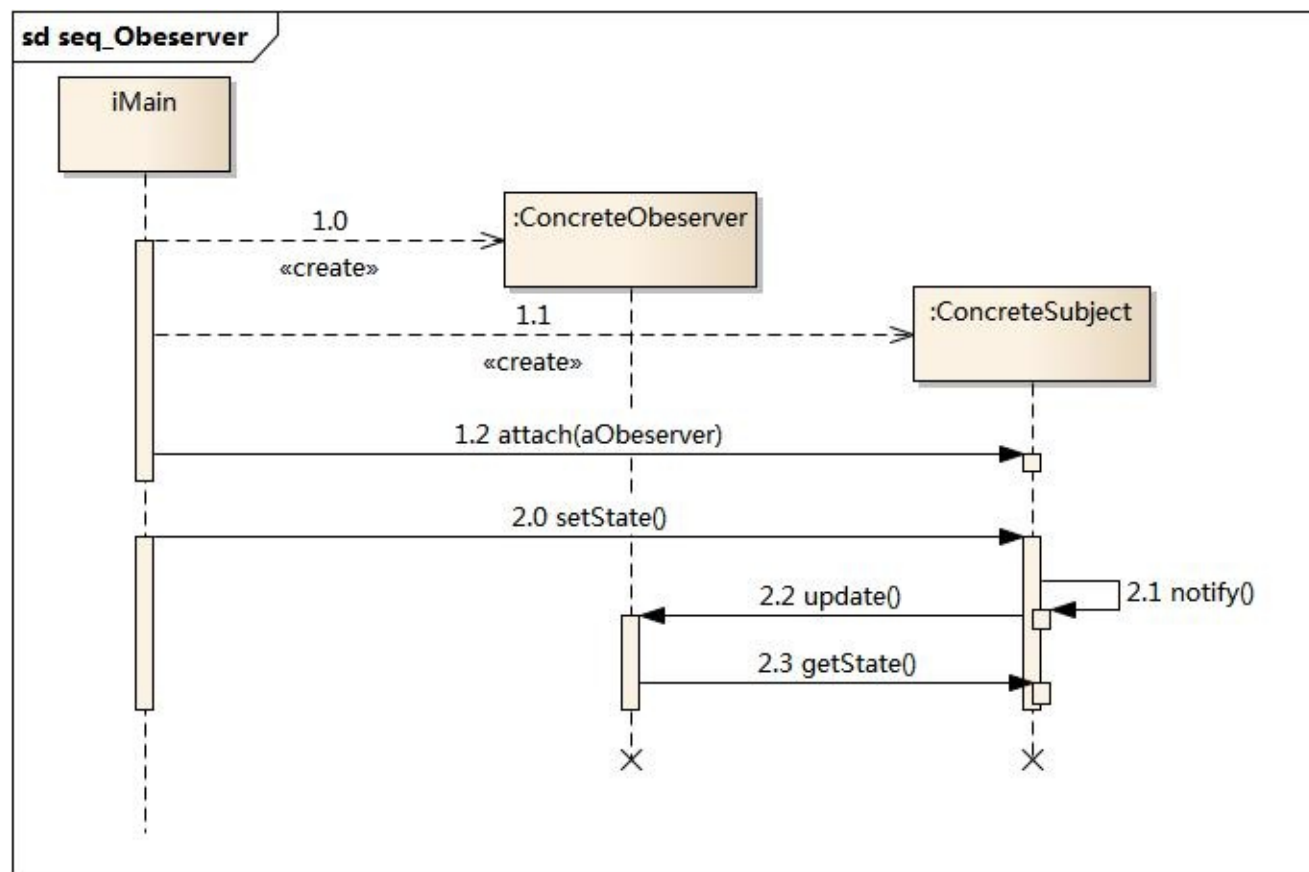
3.3. 模式结构

观察者模式包含如下角色：

- Subject：目标
- ConcreteSubject：具体目标
- Observer：观察者
- ConcreteObserver：具体观察者



3.4. 时序图



3.5. 代码分析

```

1. #include <iostream>
2. #include "Subject.h"
3. #include "Observer.h"
4. #include "ConcreteObserver.h"
5. #include "ConcreteSubject.h"
6.
7. using namespace std;
8.
9. int main(int argc, char *argv[])
10. {
11.     Subject * subject = new ConcreteSubject();
12.     Observer * objA = new ConcreteObserver("A");
13.     Observer * objB = new ConcreteObserver("B");
14.     subject->attach(objA);
15.     subject->attach(objB);
16.
17.     subject->setState(1);
18.     subject->notify();
19.
20.     cout << "-----" << endl;
21.     subject->detach(objB);
22.     subject->setState(2);
23.     subject->notify();
24.
25.     delete subject;
  
```

3. 观察者模式

```
26.     delete objA;
27.     delete objB;
28.
29.     return 0;
30. }
```

```
1.  //////////////////////////////////
2. // Subject.h
3. // Implementation of the Class Subject
4. // Created on:      07-十月-2014 23:00:10
5. // Original author: cl
6.  //////////////////////////////////
7.
8. #if !defined(EA_61998456_1B61_49f4_B3EA_9D28EEBC9649__INCLUDED_)
9. #define EA_61998456_1B61_49f4_B3EA_9D28EEBC9649__INCLUDED_
10.
11. #include "Obeserver.h"
12. #include <vector>
13. using namespace std;
14.
15. class Subject
16. {
17.
18. public:
19.     Subject();
20.     virtual ~Subject();
21.     Obeserver *m_Obeserver;
22.
23.     void attach(Obeserver * pObeserver);
24.     void detach(Obeserver * pObeserver);
25.     void notify();
26.
27.     virtual int getState() = 0;
28.     virtual void setState(int i)= 0;
29.
30. private:
31.     vector<Obeserver*> m_vtObj;
32.
33. };
34. #endif // !defined(EA_61998456_1B61_49f4_B3EA_9D28EEBC9649__INCLUDED_)
```

```
1.  //////////////////////////////////
2. // Subject.cpp
3. // Implementation of the Class Subject
4. // Created on:      07-十月-2014 23:00:10
5. // Original author: cl
6.  //////////////////////////////////
7.
8. #include "Subject.h"
9.
10. Subject::Subject(){
```

3. 观察者模式

```
11.
12. }
13.
14. Subject::~Subject(){
15.
16. }
17.
18. void Subject::attach(Obeserver * pObeserver){
19.     m_vtObj.push_back(pObeserver);
20. }
21.
22. void Subject::detach(Obeserver * pObeserver){
23.     for(vector<Obeserver*>::iterator itr = m_vtObj.begin();
24.         itr != m_vtObj.end(); itr++)
25.     {
26.         if(*itr == pObeserver)
27.         {
28.             m_vtObj.erase(itr);
29.             return;
30.         }
31.     }
32. }
33.
34. void Subject::notify(){
35.     for(vector<Obeserver*>::iterator itr = m_vtObj.begin();
36.         itr != m_vtObj.end();
37.         itr++)
38.     {
39.         (*itr)->update(this);
40.     }
41. }
```

```
1. //////////////////////////////////////
2. //  Obeserver.h
3. //  Implementation of the Class Obeserver
4. //  Created on:      07-十月-2014 23:00:10
5. //  Original author: cl
6. //////////////////////////////////////
7.
8. #if !defined(EA_2C7362B2_0B22_4168_8690_F9C7B76C343F__INCLUDED_)
9. #define EA_2C7362B2_0B22_4168_8690_F9C7B76C343F__INCLUDED_
10.
11. class Subject;
12.
13. class Obeserver
14. {
15.
16. public:
17.     Obeserver();
18.     virtual ~Obeserver();
19.     virtual void update(Subject * sub) = 0;
20. };
```

3. 观察者模式

```
21. #endif // !defined(EA_2C7362B2_0B22_4168_8690_F9C7B76C343F__INCLUDED_)
```

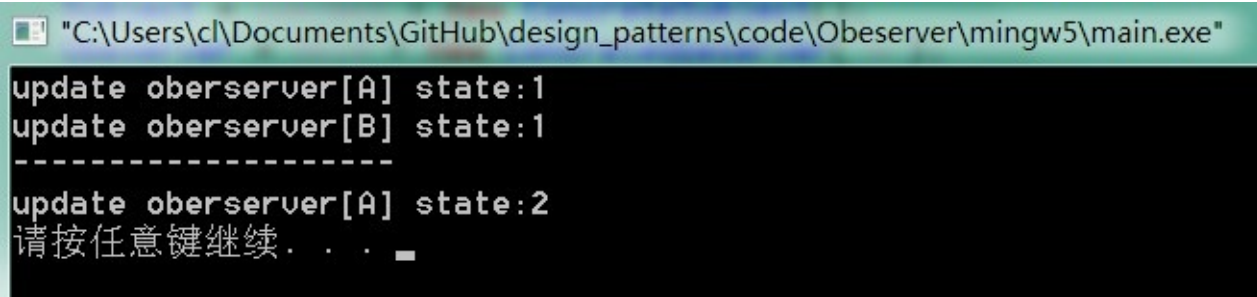
```
1. //////////////////////////////////////////////////
2. // ConcreteObeserver.h
3. // Implementation of the Class ConcreteObeserver
4. // Created on:      07-十月-2014 23:00:09
5. // Original author: cl
6. //////////////////////////////////////////////////
7.
8. #if !defined(EA_7B020534_BFEA_4c9e_8E4C_34DCE001E9B1__INCLUDED_)
9. #define EA_7B020534_BFEA_4c9e_8E4C_34DCE001E9B1__INCLUDED_
10. #include "Obeserver.h"
11. #include <string>
12. using namespace std;
13.
14. class ConcreteObeserver : public Obeserver
15. {
16.
17. public:
18.     ConcreteObeserver(string name);
19.     virtual ~ConcreteObeserver();
20.     virtual void update(Subject * sub);
21.
22. private:
23.     string m_objName;
24.     int m_obeserverState;
25. };
26. #endif // !defined(EA_7B020534_BFEA_4c9e_8E4C_34DCE001E9B1__INCLUDED_)
```

```
1. //////////////////////////////////////////////////
2. // ConcreteObeserver.cpp
3. // Implementation of the Class ConcreteObeserver
4. // Created on:      07-十月-2014 23:00:09
5. // Original author: cl
6. //////////////////////////////////////////////////
7.
8. #include "ConcreteObeserver.h"
9. #include <iostream>
10. #include <vector>
11. #include "Subject.h"
12. using namespace std;
13.
14. ConcreteObeserver::ConcreteObeserver(string name){
15.     m_objName = name;
16. }
17.
18. ConcreteObeserver::~~ConcreteObeserver(){
19.
20. }
21.
22. void ConcreteObeserver::update(Subject * sub){
```


3. 观察者模式

```
23.     m_obeserverState = sub->getState();
24.     cout << "update oberserver[" << m_objName << "]" state:" << m_obeserverState << endl;
25. }
```

运行结果：



```
"C:\Users\cl\Documents\GitHub\design_patterns\code\Obeserver\mingw5\main.exe"
update oberserver[A] state:1
update oberserver[B] state:1
-----
update oberserver[A] state:2
请按任意键继续. . .
```

3.6. 模式分析

- 观察者模式描述了如何建立对象与对象之间的依赖关系，如何构造满足这种需求的系统。
- 这一模式中的关键对象是观察目标和观察者，一个目标可以有任意数目的与之相依赖的观察者，一旦目标的状态发生改变，所有的观察者都将得到通知。
- 作为对这个通知的响应，每个观察者都将即时更新自己的状态，以与目标状态同步，这种交互也称为发布-订阅 (publishsubscribe)。目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅它并接收通知。

3.7. 实例

3.8. 优点

观察者模式的优点

- 观察者模式可以实现表示层和数据逻辑层的分离，并定义了稳定的消息更新传递机制，抽象了更新接口，使得可以有各种各样不同的表示层作为具体观察者角色。
- 观察者模式在观察目标和观察者之间建立一个抽象的耦合。
- 观察者模式支持广播通信。
- 观察者模式符合“开闭原则”的要求。

3.9. 缺点

观察者模式的缺点

- 如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
- 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。
- 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

3.10. 适用环境

在以下情况下可以使用观察者模式：

- 一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。
- 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。
- 一个对象必须通知其他对象，而并不知道这些对象是谁。
- 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象.....，可以使用观察者模式创建一种链式触发机制。

3.11. 模式应用

观察者模式在软件开发中应用非常广泛，如某电子商务网站可以在执行发送操作后给用户多个发送商品打折信息，某团队战斗游戏中某队友牺牲将给所有成员提示等等，凡是涉及到一对一或者一对多的对象交互场景都可以使用观察者模式。

3.12. 模式扩展

MVC模式

- MVC模式是一种架构模式，它包含三个角色：模型(Model)，视图(View)和控制器(Controller)。观察者模式可以用来实现MVC模式，观察者模式中的观察目标就是MVC模式中的模型(Model)，而观察者就是MVC中的视图(View)，控制器(Controller)充当两者之间的中介者(Mediator)。当模型层的数据发生改变时，视图层将自动改变其显示内容。

3.13. 总结

- 观察者模式定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅模式、模型-视图模式、源-监听器模式或从属者模式。观察者模式是一种对象行为型模式。
- 观察者模式包含四个角色：目标又称为主题，它是指被观察的对象；具体目标是目标类的子类，通常它包含有经常发生改变的数据，当它的状态发生改变时，向它的各个观察者发出通知；观察者将对观察目标的改变做出反应；在具体观察者中维护一个指向具体目标对象的引用，它存储具体观察者的有关状态，这些状态需要和具体目标的状态保持一致。
- 观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个目标对象，当这个目标对象的状态发生变化时，会通知所有观察者对象，使它们能够自动更新。
- 观察者模式的主要优点在于可以实现表示层和数据逻辑层的分离，并在观察目标和观察者之间建立一个抽象的耦合，支持广播通信；其主要缺点在于如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间，而且如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。
- 观察者模式适用情况包括：一个抽象模型有两个方面，其中一个方面依赖于另一个方面；一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变；一个对象必须通知其他对象，而并不知道这些对象是谁；需要在系统中创建一个触发链。
- 在JDK的java.util包中，提供了Observable类以及Observer接口，它们构成了Java语言对观察者模式的

3. 观察者模式

支持。

4. 状态模式

4.1. 模式动机

- 在很多情况下，一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做状态，这样的对象叫做有状态的(stateful)对象，这样的对象状态是从事先定义好的一系列值中取出的。当一个这样的对象与外部事件产生互动时，其内部状态就会改变，从而使得系统的行为也随之发生变化。
- 在UML中可以使用状态图来描述对象状态的变化。

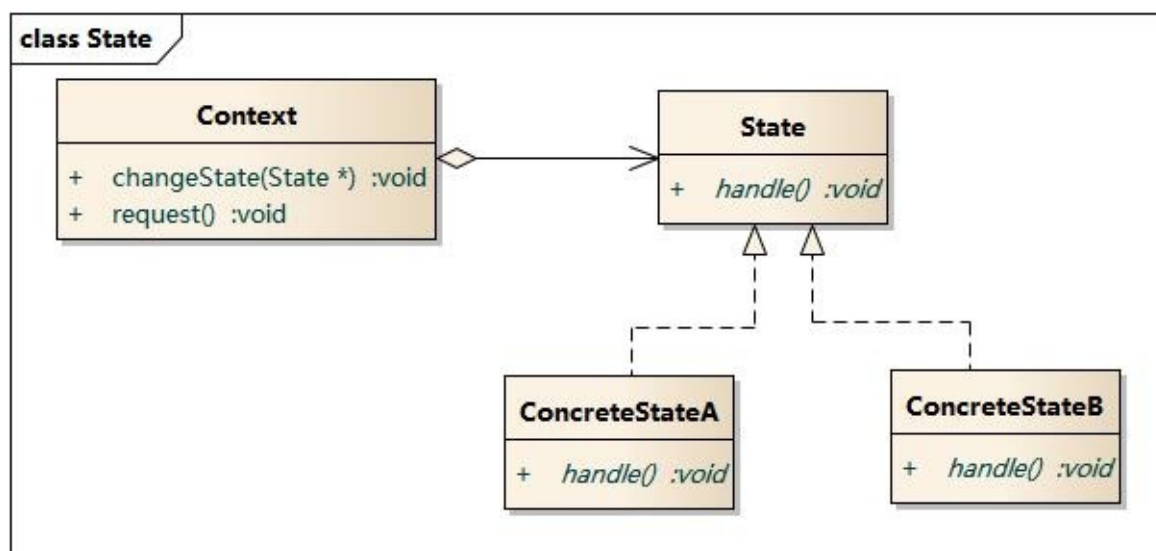
4.2. 模式定义

状态模式(State Pattern)：允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象(Objects for States)，状态模式是一种对象行为型模式。

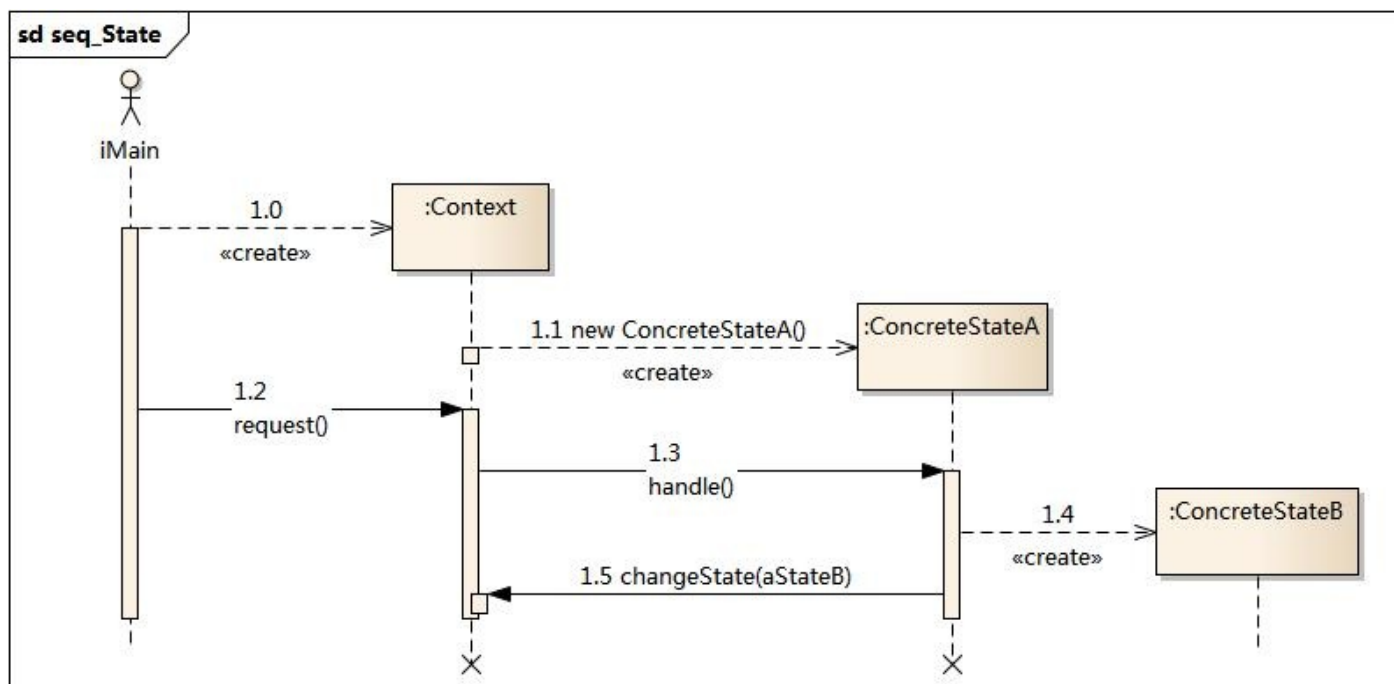
4.3. 模式结构

状态模式包含如下角色：

- Context：环境类
- State：抽象状态类
- ConcreteState：具体状态类



4.4. 时序图



既然是状态模式，加上状态图，对状态转换间的理解会清晰很多：

4.5. 代码分析

```

1. #include <iostream>
2. #include "Context.h"
3. #include "ConcreteStateA.h"
4. #include "ConcreteStateB.h"
5.
6. using namespace std;
7.
8. int main(int argc, char *argv[])
9. {
10.     char a = '0';
11.     if('0' == a)
12.         cout << "yes" << endl;
13.     else
14.         cout << "no" << endl;
15.
16.     Context * c = new Context();
17.     c->request();
18.     c->request();
19.     c->request();
20.
21.     delete c;
22.     return 0;
23. }

```

```

1. //////////////////////////////////////
2. // Context.h
3. // Implementation of the Class Context

```

4. 状态模式

```
4. // Created on:      09-十月-2014 17:20:59
5. // Original author: colin
6. ///////////////////////////////////////////////////////////////////
7.
8. #if !defined(EA_F245CF81_2A68_4461_B039_2B901BD5A126__INCLUDED_)
9. #define EA_F245CF81_2A68_4461_B039_2B901BD5A126__INCLUDED_
10.
11. #include "State.h"
12.
13. class Context
14. {
15.
16. public:
17.     Context();
18.     virtual ~Context();
19.
20.     void changeState(State * st);
21.     void request();
22.
23. private:
24.     State *m_pState;
25. };
26. #endif // !defined(EA_F245CF81_2A68_4461_B039_2B901BD5A126__INCLUDED_)
```

```
1. ///////////////////////////////////////////////////////////////////
2. // Context.cpp
3. // Implementation of the Class Context
4. // Created on:      09-十月-2014 17:20:59
5. // Original author: colin
6. ///////////////////////////////////////////////////////////////////
7.
8. #include "Context.h"
9. #include "ConcreteStateA.h"
10.
11. Context::Context(){
12.     //default is a
13.     m_pState = ConcreteStateA::Instance();
14. }
15.
16. Context::~~Context(){
17. }
18.
19. void Context::changeState(State * st){
20.     m_pState = st;
21. }
22.
23. void Context::request(){
24.     m_pState->handle(this);
25. }
```

```
1. ///////////////////////////////////////////////////////////////////
```

4. 状态模式

```
2. // ConcreteStateA.h
3. // Implementation of the Class ConcreteStateA
4. // Created on:      09-十月-2014 17:20:58
5. // Original author: colin
6. ///////////////////////////////////////////////////////////////////
7.
8. #if !defined(EA_84158F08_E96A_4bdb_89A1_4BE2E633C3EE__INCLUDED_)
9. #define EA_84158F08_E96A_4bdb_89A1_4BE2E633C3EE__INCLUDED_
10.
11. #include "State.h"
12.
13. class ConcreteStateA : public State
14. {
15.
16. public:
17.     virtual ~ConcreteStateA();
18.
19.     static State * Instance();
20.
21.     virtual void handle(Context * c);
22.
23. private:
24.     ConcreteStateA();
25.     static State * m_pState;
26. };
27. #endif // !defined(EA_84158F08_E96A_4bdb_89A1_4BE2E633C3EE__INCLUDED_)
```

```
1. ///////////////////////////////////////////////////////////////////
2. // ConcreteStateA.cpp
3. // Implementation of the Class ConcreteStateA
4. // Created on:      09-十月-2014 17:20:58
5. // Original author: colin
6. ///////////////////////////////////////////////////////////////////
7.
8. #include "ConcreteStateA.h"
9. #include "ConcreteStateB.h"
10. #include "Context.h"
11. #include <iostream>
12. using namespace std;
13.
14. State * ConcreteStateA::m_pState = NULL;
15. ConcreteStateA::ConcreteStateA(){
16. }
17.
18. ConcreteStateA::~~ConcreteStateA(){
19. }
20.
21. State * ConcreteStateA::Instance()
22. {
23.     if ( NULL == m_pState)
24.     {
25.         m_pState = new ConcreteStateA();
```

```

26.     }
27.     return m_pState;
28. }
29.
30. void ConcreteStateA::handle(Context * c){
31.     cout << "doing something in State A.\n done,change state to B" << endl;
32.     c->changeState(ConcreteStateB::Instance());
33. }

```

运行结果：

4.6. 模式分析

- 状态模式描述了对对象状态的变化以及对象如何在每一种状态下表现出不同的行为。
- 状态模式的关键是引入了一个抽象类来专门表示对象的状态，这个类我们叫做抽象状态类，而对象的每一种具体状态类都继承了该类，并在不同具体状态类中实现了不同状态的行为，包括各种状态之间的转换。在状态模式结构中需要理解环境类与抽象状态类的作用：
- 环境类实际上就是拥有状态的对象，环境类有时候可以充当状态管理器(State Manager)的角色，可以在环境类中对状态进行切换操作。
- 抽象状态类可以是抽象类，也可以是接口，不同状态类就是继承这个父类的不同子类，状态类的产生是由于环境类存在多个状态，同时还满足两个条件：这些状态经常需要切换，在不同的状态下对象的行为不同。因此可以将不同对象下的行为单独提取出来封装在具体的状态类中，使得环境类对象在其内部状态改变时可以改变它的行为，对象看起来似乎修改了它的类，而实际上是由于切换到不同的具体状态类实现的。由于环境类可以设置为任一具体状态类，因此它针对抽象状态类进行编程，在程序运行时可以将任一具体状态类的对象设置到环境类中，从而使得环境类可以改变内部状态，并且改变行为。

4.7. 实例

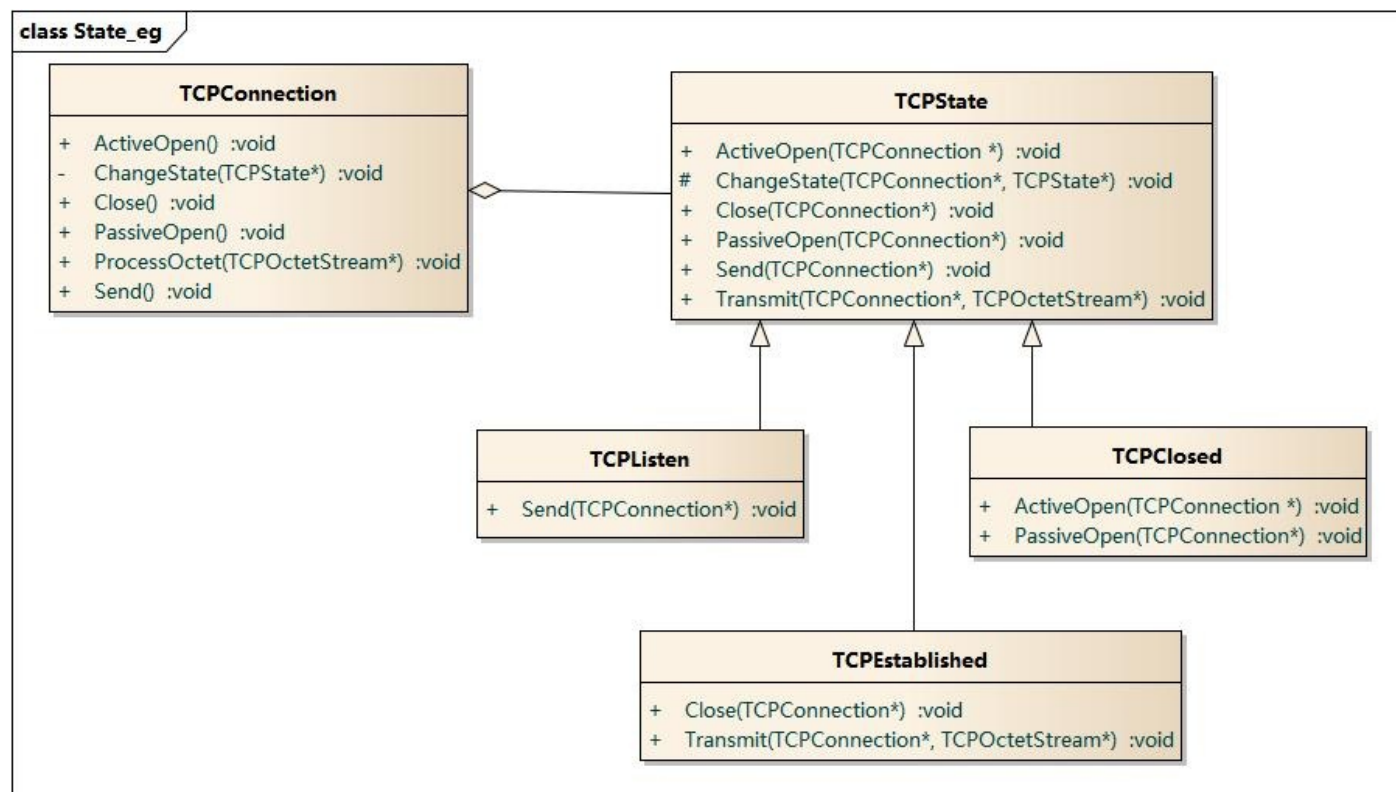
TCPConnection

这个示例来自《设计模式》，展示了一个简化版的TCP协议实现；TCP连接的状态有多种可能，状态之间的转换有相应的逻辑前提；这是使用状态模式的场合；

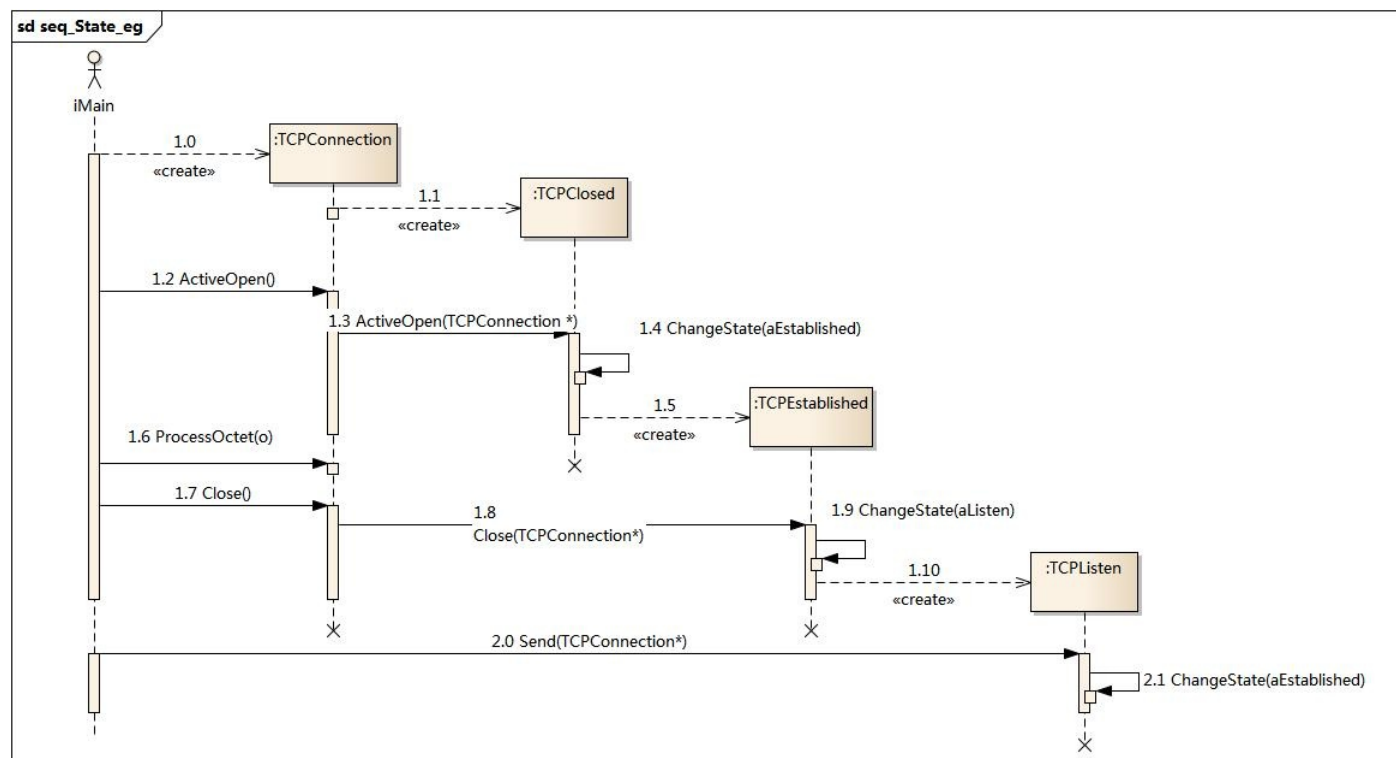
状态图：



结构图：



时序图：



4.8. 优点

状态模式的优点

- 封装了转换规则。
- 枚举可能的状态，在枚举状态之前需要确定状态种类。
- 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。

- 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。
- 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

4.9. 缺点

状态模式的缺点

- 状态模式的使用必然会增加系统类和对象的个数。
- 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
- 状态模式对“开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态；而且修改某个状态类的行为也需修改对应类的源代码。

4.10. 适用环境

在以下情况下可以使用状态模式：

- 对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为。
- 代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。在这些条件语句中包含了对象的行为，而且这些条件对应于对象的各种状态。

4.11. 模式应用

状态模式在工作流或游戏等类型的软件中得以广泛使用，甚至可以用于这些系统的核心功能设计，如在政府OA办公系统中，一个批文的状态有多种：尚未办理；正在办理；正在批示；正在审核；已经完成等各种状态，而且批文状态不同时对批文的操作也有所差异。使用状态模式可以描述工作流对象（如批文）的状态转换以及不同状态下它所具有的行为。

4.12. 模式扩展

共享状态

- 在有些情况下多个环境对象需要共享同一个状态，如果希望在系统中实现多个环境对象实例共享一个或多个状态对象，那么需要将这些状态对象定义为环境的静态成员对象。
- 简单状态模式与可切换状态的状态模式
 - 简单状态模式：简单状态模式是指状态都相互独立，状态之间无须进行转换的状态模式，这是最简单的一种状态模式。对于这种状态模式，每个状态类都封装与状态相关的操作，而无须关心状态的切换，可以在客户端直接实例化状态类，然后将状态对象设置到环境类中。如果是这种简单的状态模式，它遵循“开闭原则”，在客户端可以针对抽象状态类进行编程，而将具体状态类写到配置文件中，同时增加新的状态类对原有系统也不造成任何影响。
 - 可切换状态的状态模式：大多数的状态模式都是可以切换状态的状态模式，在实现状态切换时，在具体状态类内部需要调用环境类Context的setState()方法进行状态的转换操作，在具体状态类中可以调用到环境类的方法，因此状态类与环境类之间通常还存在关联关系或者依赖关系。通过在状态类中引用环境类的对象来回调环境类的setState()方法实现状态的切换。在这种可以切换状态的状态模式中，增加新的

状态类可能需要修改其他某些状态类甚至环境类的源代码，否则系统无法切换到新增状态。

4.13. 总结

- 状态模式允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象，状态模式是一种对象行为型模式。
- 状态模式包含三个角色：环境类又称为上下文类，它是拥有状态的对象，在环境类中维护一个抽象状态类State的实例，这个实例定义当前状态，在具体实现时，它是一个State子类的对象，可以定义初始状态；抽象状态类用于定义一个接口以封装与环境类的一个特定状态相关的行为；具体状态类是抽象状态类的子类，每一个子类实现一个与环境类的一个状态相关的行为，每一个具体状态类对应环境的一个具体状态，不同的具体状态类其行为有所不同。
- 状态模式描述了对对象状态的变化以及对象如何在每一种状态下表现出不同的行为。
- 状态模式的主要优点在于封装了转换规则，并枚举可能的状态，它将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为，还可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数；其缺点在于使用状态模式会增加系统类和对象的个数，且状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱，对于可以切换状态的状态模式不满足“开闭原则”的要求。
- 状态模式适用情况包括：对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为；代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。

5. 策略模式

5.1. 模式动机

- 完成一项任务，往往可以有多种不同的方式，每一种方式称为一个策略，我们可以根据环境或者条件的不同选择不同的策略来完成该项任务。
- 在软件开发中也常常遇到类似的情况，实现某一个功能有多个途径，此时可以使用一种设计模式来使得系统可以灵活地选择解决途径，也能够方便地增加新的解决途径。
- 在软件系统中，有许多算法可以实现某一功能，如查找、排序等，一种常用的方法是硬编码(Hard Coding)在一个类中，如需要提供多种查找算法，可以将这些算法写到一个类中，在该类中提供多个方法，每一个方法对应一个具体的查找算法；当然也可以将这些查找算法封装在一个统一的方法中，通过if...else...等条件判断语句来进行选择。这两种实现方法我们都可以称之为硬编码，如果需要增加一种新的查找算法，需要修改封装算法类的源代码；更换查找算法，也需要修改客户端调用代码。在这个算法类中封装了大量查找算法，该类代码将较复杂，维护较为困难。
- 除了提供专门的查找算法类之外，还可以在客户端程序中直接包含算法代码，这种做法更不可取，将导致客户端程序庞大而且难以维护，如果存在大量可供选择的算法时问题将变得更加严重。
- 为了解决这些问题，可以定义一些独立的类来封装不同的算法，每一个类封装一个具体的算法，在这里，每一个封装算法的类我们都可以称之为策略(Strategy)，为了保证这些策略的一致性，一般会用一个抽象的策略类来做算法的定义，而具体每种算法则对应于一个具体策略类。

5.2. 模式定义

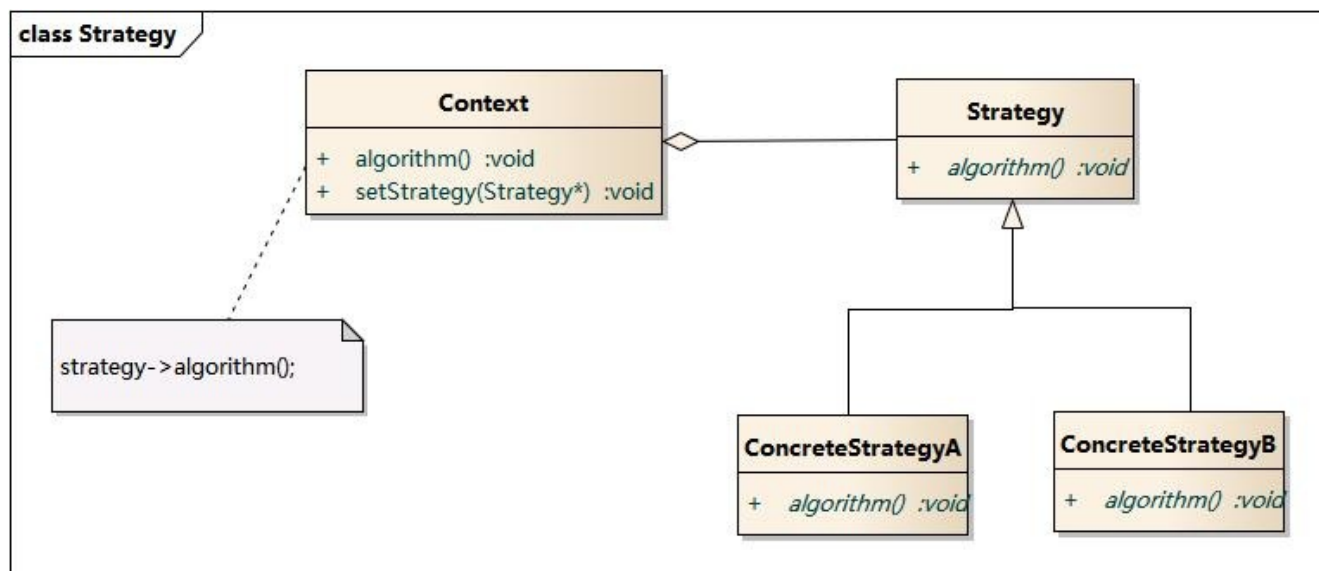
策略模式(Strategy Pattern)：定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化，也称为政策模式(Policy)。

策略模式是一种对象行为型模式。

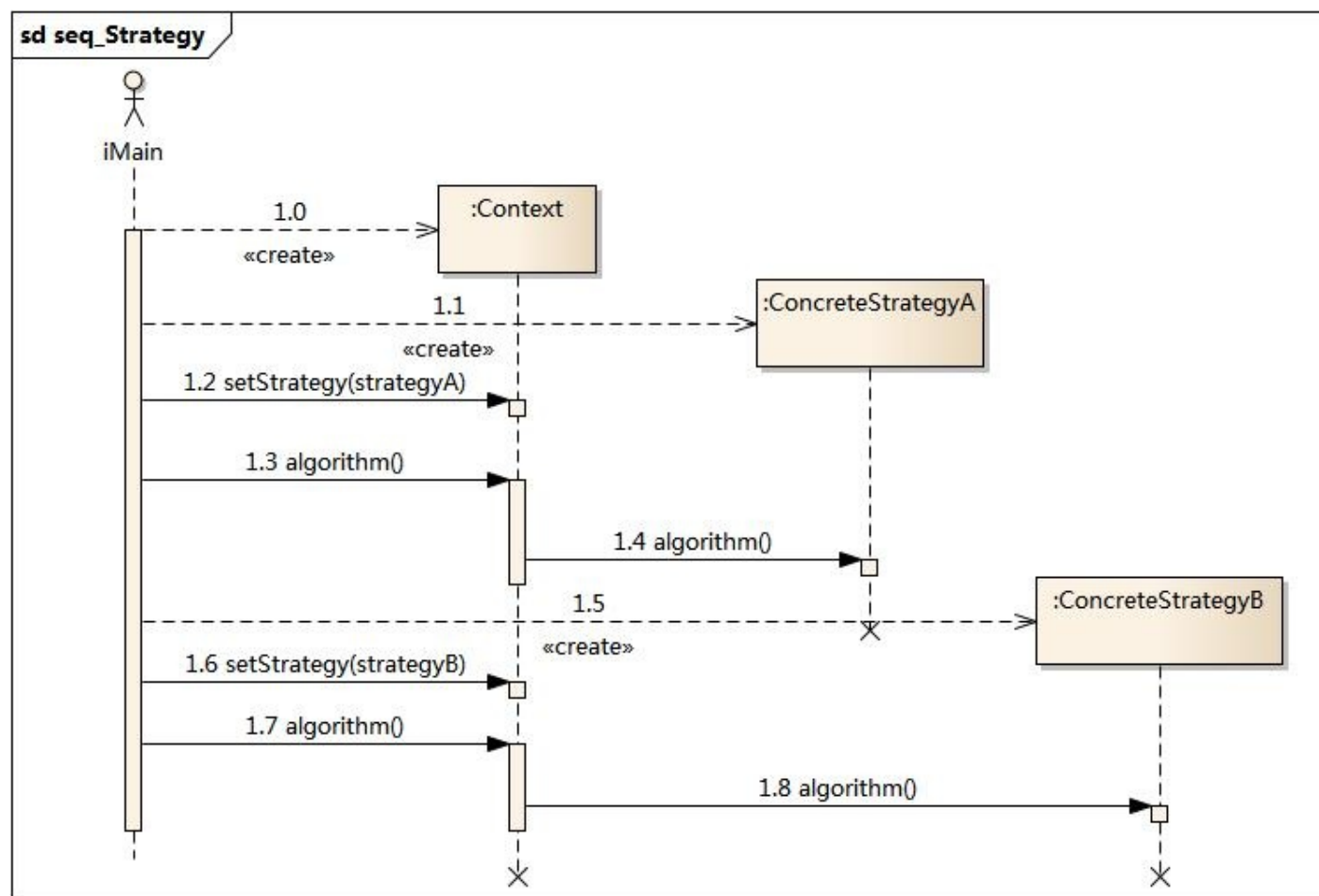
5.3. 模式结构

策略模式包含如下角色：

- Context：环境类
- Strategy：抽象策略类
- ConcreteStrategy：具体策略类



5.4. 时序图



5.5. 代码分析

```

1. #include <iostream>
2. #include "Context.h"
3. #include "ConcreteStrategyA.h"
  
```

```

4. #include "ConcreteStrategyB.h"
5. #include "Strategy.h"
6. #include <vector>
7. using namespace std;
8.
9. int main(int argc, char *argv[])
10. {
11.     Strategy * s1 = new ConcreteStrategyA();
12.     Context * cxt = new Context();
13.     cxt->setStrategy(s1);
14.     cxt->algorithm();
15.
16.     Strategy *s2 = new ConcreteStrategyB();
17.     cxt->setStrategy(s2);
18.     cxt->algorithm();
19.
20.     delete s1;
21.     delete s2;
22.
23.     int rac1 = 0x1;
24.     int rac2 = 0x2;
25.     int rac3 = 0x4;
26.     int rac4 = 0x8;
27.
28.     int i = 0xe;
29.     int j = 0x5;
30.
31.     int r1 = i & rac1;
32.     int r2 = i & rac2;
33.     int r3 = i & rac3;
34.     int r4 = i & rac4;
35.
36.     cout <<"res:" << r1 << "/" << r2 << "/" << r3 << "/" << r4 << endl;
37.
38.     return 0;
39. }

```

```

1. //////////////////////////////////////
2. // Context.h
3. // Implementation of the Class Context
4. // Created on:      09-十月-2014 22:21:07
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #if !defined(EA_0DA87730_4DEE_4392_9BAF_4AC64A8A07A4__INCLUDED_)
9. #define EA_0DA87730_4DEE_4392_9BAF_4AC64A8A07A4__INCLUDED_
10.
11. #include "Strategy.h"
12.
13. class Context
14. {
15.

```

```

16. public:
17.     Context();
18.     virtual ~Context();
19.
20.
21.     void algorithm();
22.     void setStrategy(Strategy* st);
23.
24. private:
25.     Strategy *m_pStrategy;
26.
27. };
28. #endif // !defined(EA_0DA87730_4DEE_4392_9BAF_4AC64A8A07A4__INCLUDED_)

```

```

1. //////////////////////////////////////
2. // Context.cpp
3. // Implementation of the Class Context
4. // Created on:      09-十月-2014 22:21:07
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #include "Context.h"
9.
10. Context::Context(){
11. }
12.
13. Context::~Context(){
14. }
15.
16. void Context::algorithm(){
17.     m_pStrategy->algorithm();
18. }
19.
20. void Context::setStrategy(Strategy* st){
21.     m_pStrategy = st;
22. }

```

```

1. //////////////////////////////////////
2. // ConcreteStrategyA.h
3. // Implementation of the Class ConcreteStrategyA
4. // Created on:      09-十月-2014 22:21:06
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #if !defined(EA_9B180F12_677B_4e9b_A243_1F5DAD93FE1D__INCLUDED_)
9. #define EA_9B180F12_677B_4e9b_A243_1F5DAD93FE1D__INCLUDED_
10.
11. #include "Strategy.h"
12.
13. class ConcreteStrategyA : public Strategy
14. {

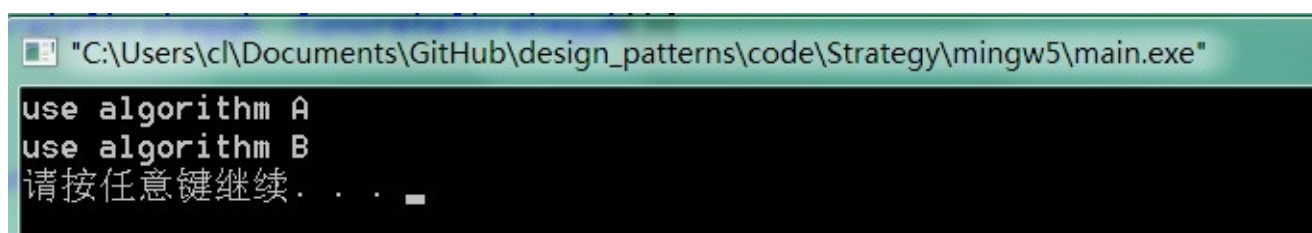
```


5. 策略模式

```
15.
16. public:
17.     ConcreteStrategyA();
18.     virtual ~ConcreteStrategyA();
19.
20.     virtual void algorithm();
21.
22. };
23. #endif // !defined(EA_9B180F12_677B_4e9b_A243_1F5DAD93FE1D__INCLUDED_)
```

```
1. //////////////////////////////////////
2. // ConcreteStrategyA.cpp
3. // Implementation of the Class ConcreteStrategyA
4. // Created on:      09-十月-2014 22:21:07
5. // Original author: colin
6. //////////////////////////////////////
7.
8. #include "ConcreteStrategyA.h"
9. #include <iostream>
10. using namespace std;
11.
12. ConcreteStrategyA::ConcreteStrategyA(){
13.
14. }
15.
16. ConcreteStrategyA::~~ConcreteStrategyA(){
17.
18. }
19.
20. void ConcreteStrategyA::algorithm(){
21.     cout << "use algorithm A" << endl;
22. }
```

运行结果：



```
"C:\Users\cl\Documents\GitHub\design_patterns\code\Strategy\mingw5\main.exe"
use algorithm A
use algorithm B
请按任意键继续. . .
```

5.6. 模式分析

- 策略模式是一个比较容易理解和使用的设计模式，策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。
- 在策略模式中，应当由客户端自己决定在什么情况下使用什么具体策略角色。
- 策略模式仅仅封装算法，提供新算法插入到已有系统中，以及老算法从系统中“退休”的方便，策略模式并不决定在何时使用何种算法，算法的选择由客户端来决定。这在一定程度上提高了系统的灵活性，但是客户端需要理

解所有具体策略类之间的区别，以便选择合适的算法，这也是策略模式的缺点之一，在一定程度上增加了客户端的使用难度。

5.7. 实例

5.8. 优点

策略模式的优点

- 策略模式提供了对“开闭原则”的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。
- 策略模式提供了管理相关的算法族的办法。
- 策略模式提供了可以替换继承关系的办法。
- 使用策略模式可以避免使用多重条件转移语句。

5.9. 缺点

策略模式的缺点

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。
- 策略模式将造成产生很多策略类，可以通过使用享元模式在一定程度上减少对象的数量。

5.10. 适用环境

在以下情况下可以使用策略模式：

- 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。
- 一个系统需要动态地在几种算法中选择一种。
- 如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。
- 不希望客户端知道复杂的、与算法相关的数据结构，在具体策略类中封装算法和相关的的数据结构，提高算法的保密性与安全性。

5.11. 模式应用

5.12. 模式扩展

策略模式与状态模式

- 可以通过环境类状态的个数来决定是使用策略模式还是状态模式。
- 策略模式的环境类自己选择一个具体策略类，具体策略类无须关心环境类；而状态模式的环境类由于外在因素需要放进一个具体状态中，以便通过其方法实现状态的切换，因此环境类和状态类之间存在一种双向的关联关系。
- 使用策略模式时，客户端需要知道所选的具体策略是哪一个，而使用状态模式时，客户端无须关心具体状态，环

境类的状态会根据用户的操作自动转换。

- 如果系统中某个类的对象存在多种状态，不同状态下行为有差异，而且这些状态之间可以发生转换时使用状态模式；如果系统中某个类的某一行为存在多种实现方式，而且这些实现方式可以互换时使用策略模式。

5.13. 总结

- 在策略模式中定义了一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化，也称为政策模式。策略模式是一种对象行为型模式。
- 策略模式包含三个角色：环境类在解决某个问题时可以采用多种策略，在环境类中维护一个对抽象策略类的引用实例；抽象策略类为所支持的算法声明了抽象方法，是所有策略类的父类；具体策略类实现了在抽象策略类中定义的算法。
- 策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列的策略类里面，作为一个抽象策略类的子类。
- 策略模式主要优点在于对“开闭原则”的完美支持，在不修改原有系统的基础上可以更换算法或者增加新的算法，它很好地管理算法族，提高了代码的复用性，是一种替换继承，避免多重条件转移语句的实现方式；其缺点在于客户端必须知道所有的策略类，并理解其区别，同时在一定程度上增加了系统中类的个数，可能会存在很多策略类。
- 策略模式适用情况包括：在一个系统里面有许多类，它们之间的区别仅在于它们的行为，使用策略模式可以动态地让一个对象在许多行为中选择一种行为；一个系统需要动态地在几种算法中选择一种；避免使用难以维护的多重条件选择语句；希望在具体策略类中封装算法和与相关的数据结构。