

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

目录

- 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

目录

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

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，这可能是因为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。
- 在适配器模式中可以定义一个包装类，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。

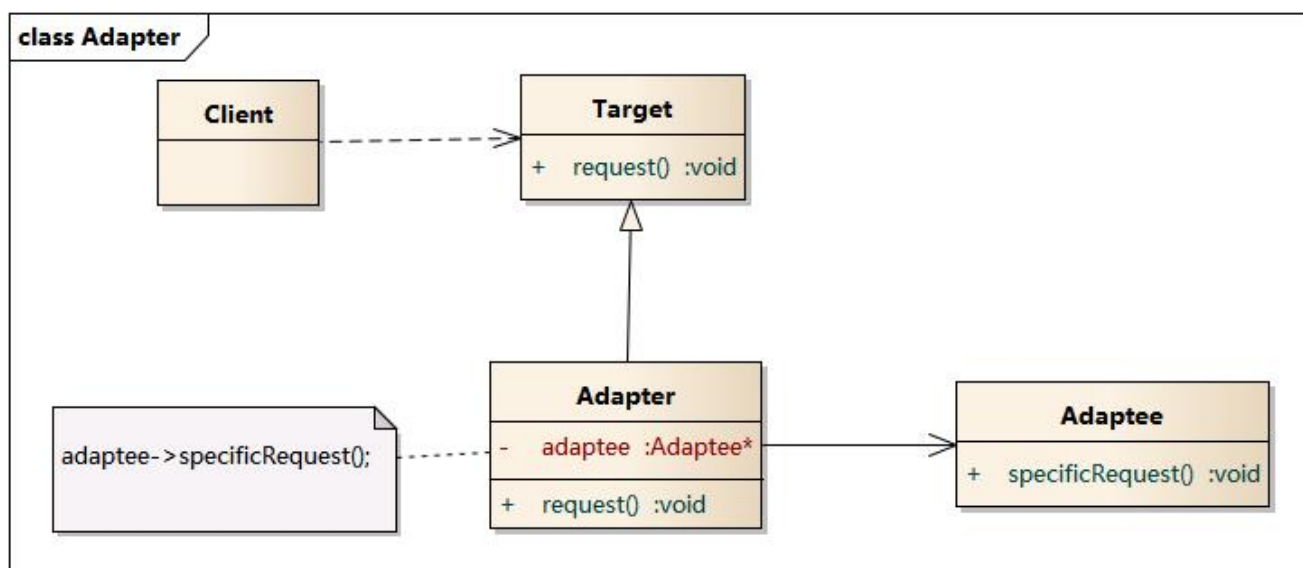
1.3. 模式结构

适配器模式包含如下角色：

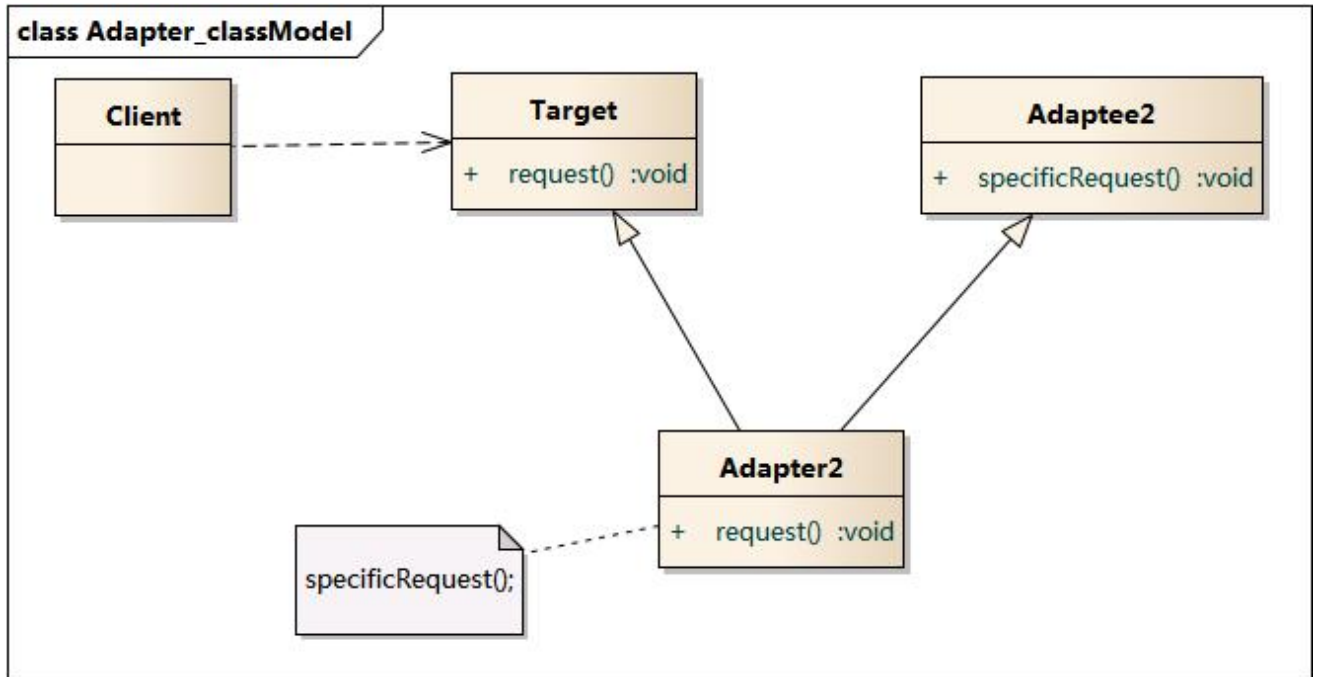
- Target：目标抽象类
- Adapter：适配器类
- Adaptee：适配者类
- Client：客户类

适配器模式有对象适配器和类适配器两种实现：

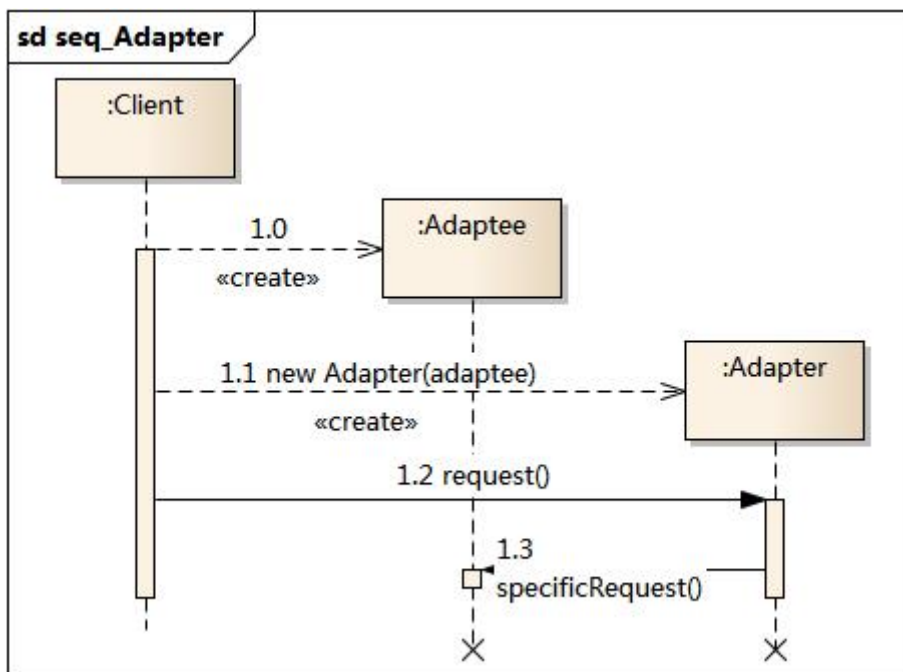
对象适配器：



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  #if !defined(EA_BD766D47_0C69_4131_B7B9_21DF78B1E80D__INCLUDED_)
8  #define EA_BD766D47_0C69_4131_B7B9_21DF78B1E80D__INCLUDED_
9
10 #include "Target.h"
11 #include "Adaptee.h"
12
13 class Adapter : public Target
14 {
15 public:
16     Adapter(Adaptee *adaptee);
17     virtual ~Adapter();
18
19     virtual void request();
20
21 private:
22     Adaptee* m_pAdaptee;
23
24 };
25 #endif // !defined(EA_BD766D47_0C69_4131_B7B9_21DF78B1E80D__INCLUDED_)
26
27

```

```

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 #include "Adapter.h"
8 Adapter::Adapter(Adaptee * adaptee){
9     m_pAdaptee = adaptee;
10 }
11 Adapter::~Adapter(){
12 }
13 }
14 void Adapter::request(){
15     m_pAdaptee->specificRequest();
16 }
17
18
19
20

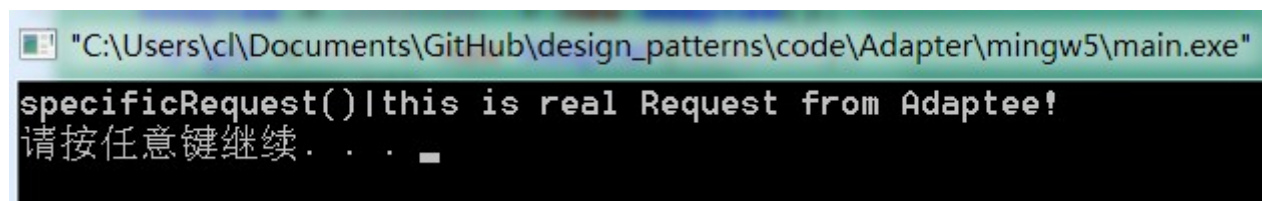
```

```

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 #if !defined(EA_826E6B4F_12BE_4609_A0A3_95BD5E657D36__INCLUDED_)
8 #define EA_826E6B4F_12BE_4609_A0A3_95BD5E657D36__INCLUDED_
9
10 class Adaptee
11 {
12 public:
13     Adaptee();
14     virtual ~Adaptee();
15
16     void specificRequest();
17 };
18 #endif // !defined(EA_826E6B4F_12BE_4609_A0A3_95BD5E657D36__INCLUDED_)
19
20
21

```

运行结果：



```

"C:\Users\cl\Documents\GitHub\design_patterns\code\Adapter\mingw5\main.exe"
specificRequest()|this is real Request from Adaptee!
请按任意键继续. . . _

```

1.6. 模式分析

1.7. 实例

1.8. 优点

- 将目标类和适配器类解耦，通过引入一个适配器类来重用现有的适配器类，而无须修改原有代码。
- 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，而這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

默认适配器模式(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

目录

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

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对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。

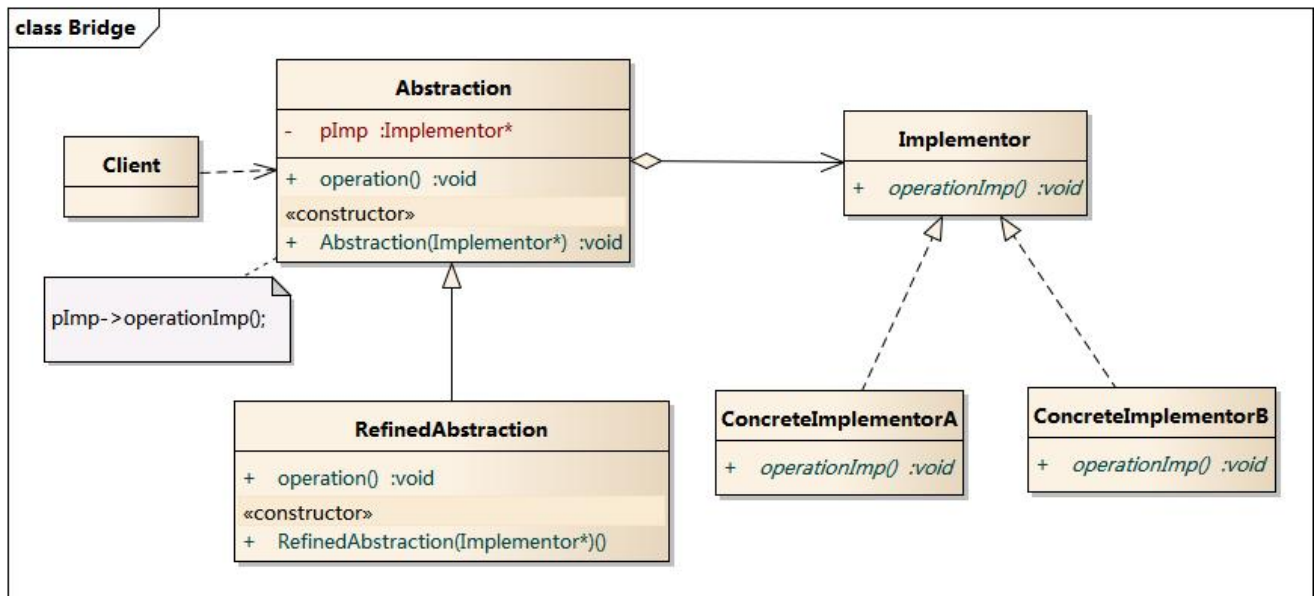
2.2. 模式定义

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。

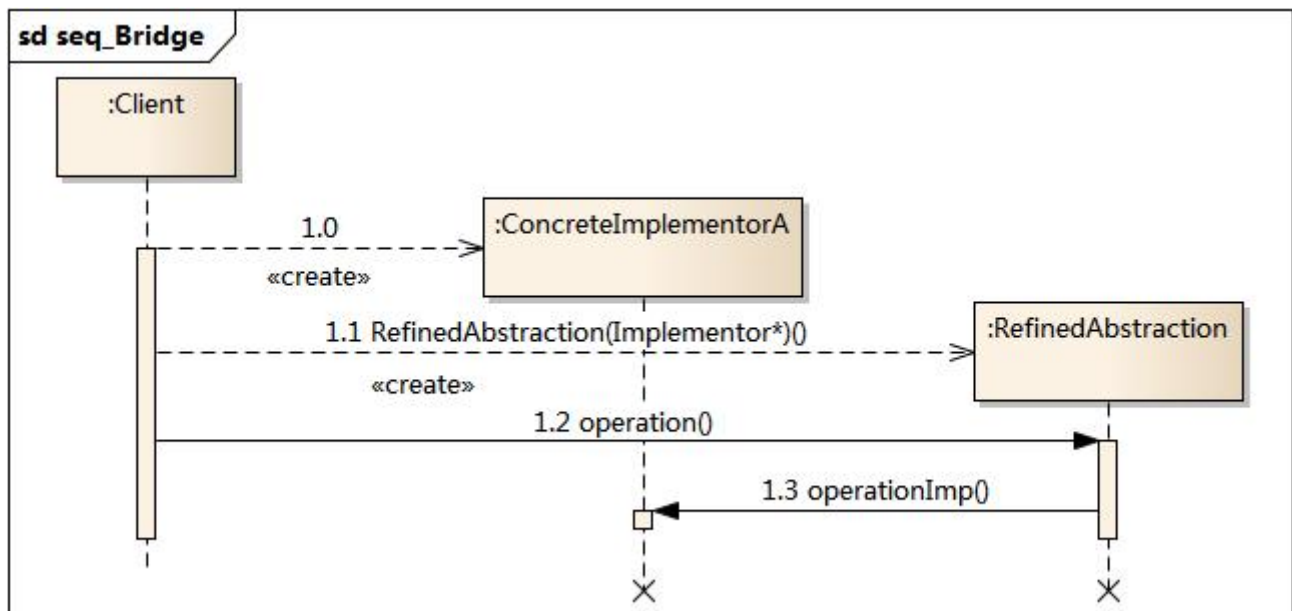
2.3. 模式结构

桥接模式包含如下角色：

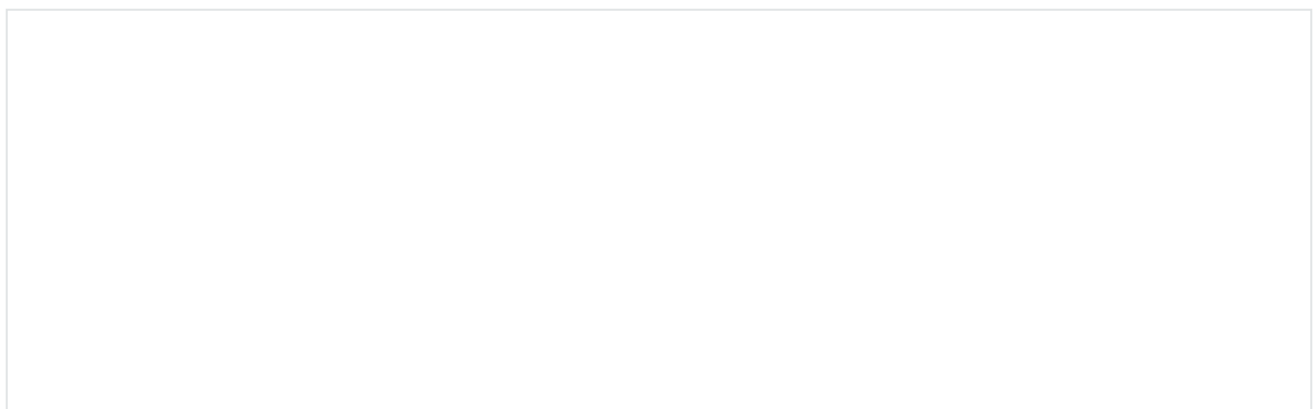
- Abstraction：抽象类
- RefinedAbstraction：扩充抽象类
- Implementor：实现类接口
- ConcreteImplementor：具体实现类



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     Implementor * pImp = new ConcreteImplementorA();
12     Abstraction * pa = new RefinedAbstraction(pImp);
13     pa->operation();
14
15     Abstraction * pb = new RefinedAbstraction(new ConcreteImplementorB());
16     pb->operation();
17
18     delete pa;
19     delete pb;
20
21     return 0;
22 }
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
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 public:
16     RefinedAbstraction();
17     RefinedAbstraction(Implementor* imp);
18     virtual ~RefinedAbstraction();
19
20     virtual void operation();
21
22 };
23 #endif // !defined(EA_4BA5BE7C_DED5_4236_8362_F2988921CFA7__INCLUDED_)
24

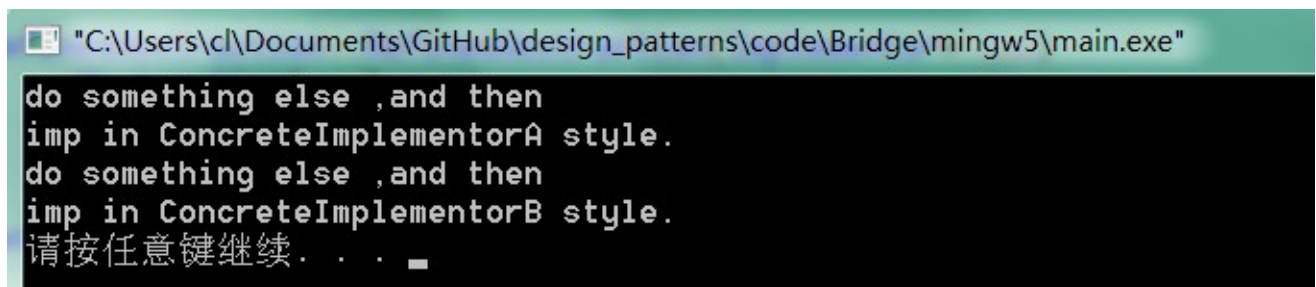
```

```

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  #include "RefinedAbstraction.h"
8  #include <iostream>
9  using namespace std;
10
11  RefinedAbstraction::RefinedAbstraction(){
12  }
13
14  RefinedAbstraction::RefinedAbstraction(Implementor* imp)
15      :Abstraction(imp)
16  {
17  }
18
19  RefinedAbstraction::~~RefinedAbstraction(){
20  }
21
22  void RefinedAbstraction::operation(){
23      cout << "do something else ,and then " << endl;
24      m_pImp->operationImp();
25  }
26
27
28
29
30
31

```

运行结果：



```

"C:\Users\cl\Documents\GitHub\design_patterns\code\Bridge\mingw5\main.exe"
do something else ,and then
imp in ConcreteImplementorA style.
do something else ,and then
imp in ConcreteImplementorB style.
请按任意键继续. . .

```

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

目录

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

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。

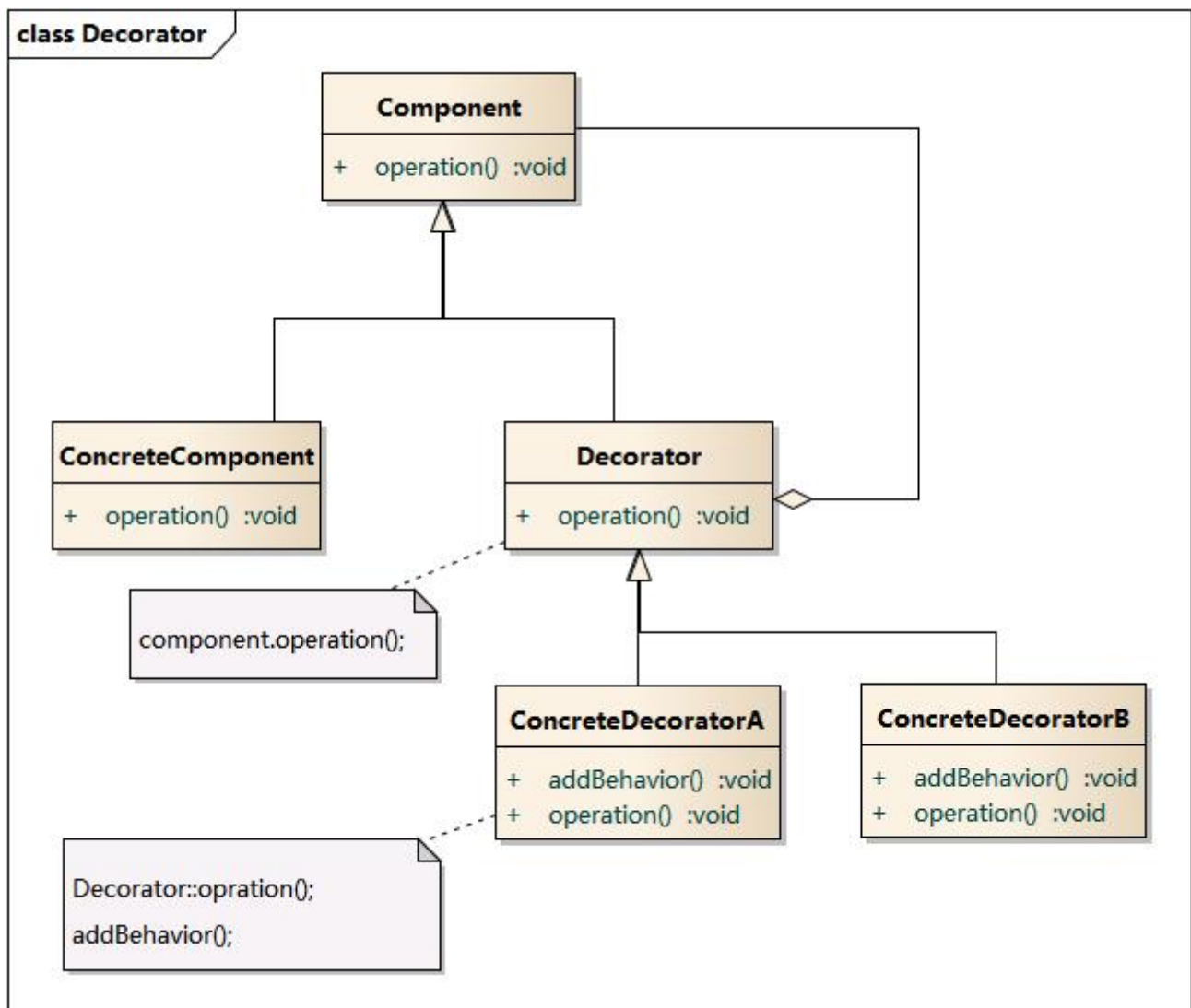
3.2. 模式定义

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。

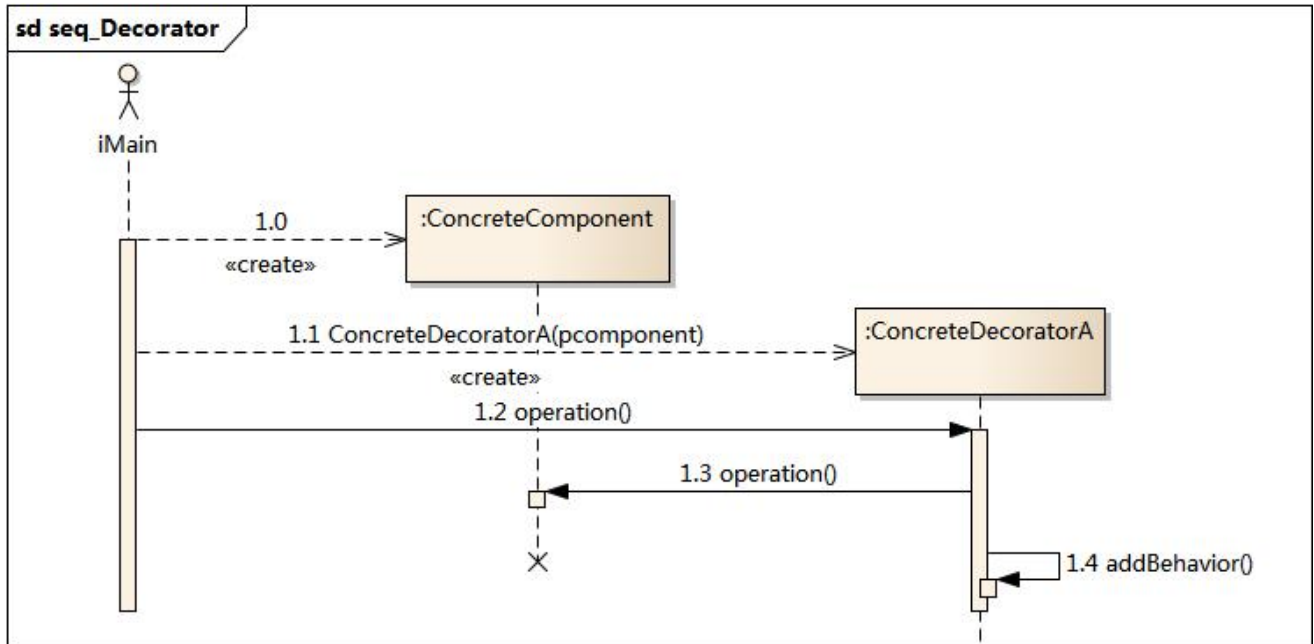
3.3. 模式结构

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 ConcreteComponent::ConcreteComponent(){
13 }
14 ConcreteComponent::~ConcreteComponent(){
15 }
16
17 void ConcreteComponent::operation(){
18     cout << "ConcreteComponent's normal operation!" << endl;
19 }
20
21
22
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  #if !defined(EA_6786B68E_DCE4_44c4_B26D_812F0B3C0382__INCLUDED_)
8  #define EA_6786B68E_DCE4_44c4_B26D_812F0B3C0382__INCLUDED_
9
10 #include "Decorator.h"
11 #include "Component.h"
12
13 class ConcreteDecoratorA : public Decorator
14 {
15 public:
16     ConcreteDecoratorA(Component* pcmp);
17     virtual ~ConcreteDecoratorA();
18
19     void addBehavior();
20     virtual void operation();
21 };
22 #endif // !defined(EA_6786B68E_DCE4_44c4_B26D_812F0B3C0382__INCLUDED_)
23
24
25

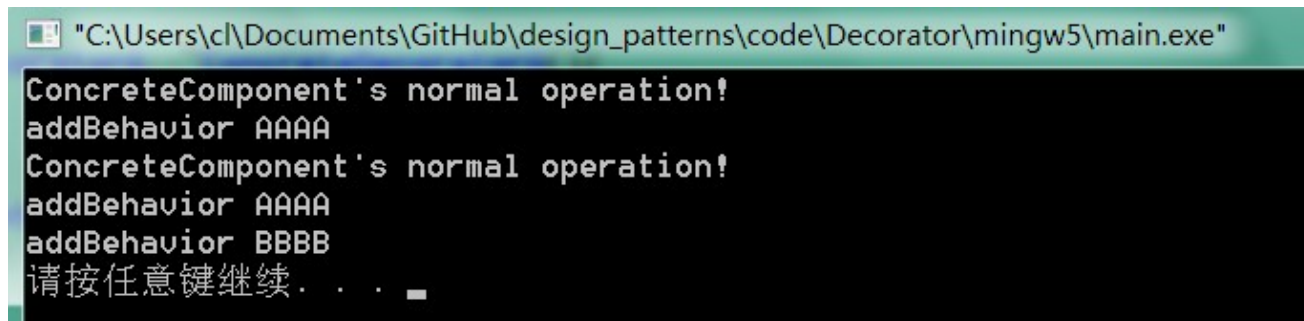
```

```

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  #include "ConcreteDecoratorA.h"
8  #include <iostream>
9  using namespace std;
10
11 ConcreteDecoratorA::ConcreteDecoratorA(Component* pcmp)
12 :Decorator(pcmp)
13 {
14 }
15
16 ConcreteDecoratorA::~~ConcreteDecoratorA(){
17 }
18
19 void ConcreteDecoratorA::addBehavior(){
20     cout << "addBehavior AAAA" << endl;
21 }
22
23 void ConcreteDecoratorA::operation(){
24     Decorator::operation();
25     addBehavior();
26 }
27
28
29
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

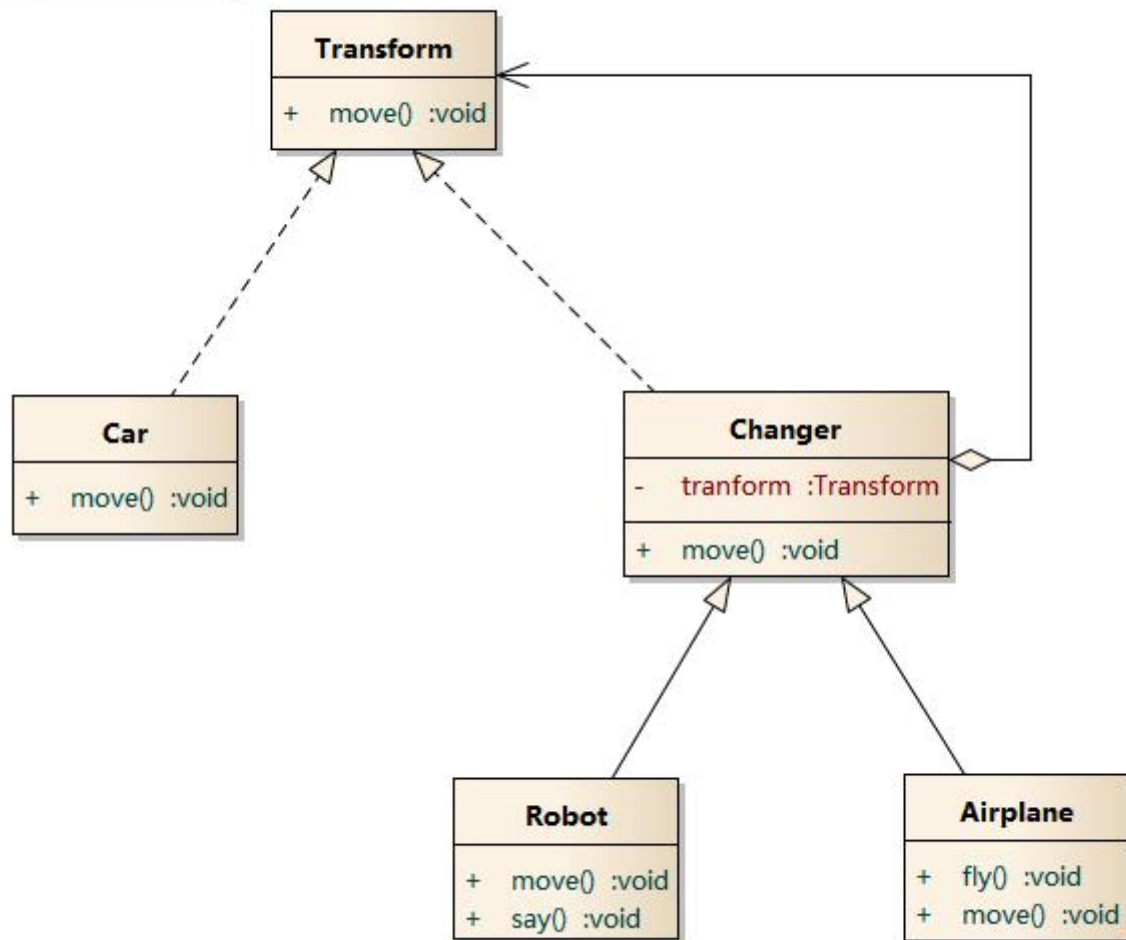
- 与继承关系相比，关联关系的主要优势在于不会破坏类的封装性，而且继承是一种耦合度较大的静态关系，无法在程序运行时动态扩展。在软件开发阶段，关联关系虽然不会比继承关系减少编码量，但是到了软件维护阶段，由于关联关系使系统具有较好的松耦合性，因此使得系统更加容易维护。当然，关联关系的缺点是比继承关系要创建更多的对象。
- 使用装饰模式来实现扩展比继承更加灵活，它以对客户透明的方式动态地给一个对象附加更多的责任。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。

3.7. 实例

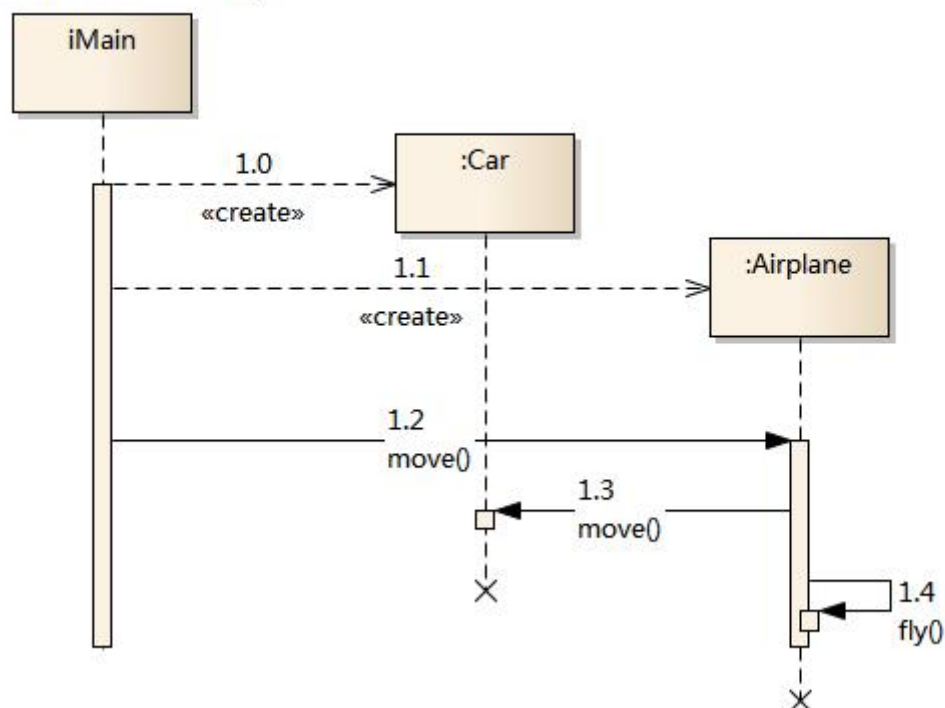
实例：变形金刚

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，除了在陆地上移动还可以在天空中飞翔。

class Decorator_eg



sd seq_Decorator_eg



3.8. 优点

装饰模式的优点:

- 装饰模式与继承关系的目都是为了扩展对象的功能，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

目录

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

4.1. 模式动机

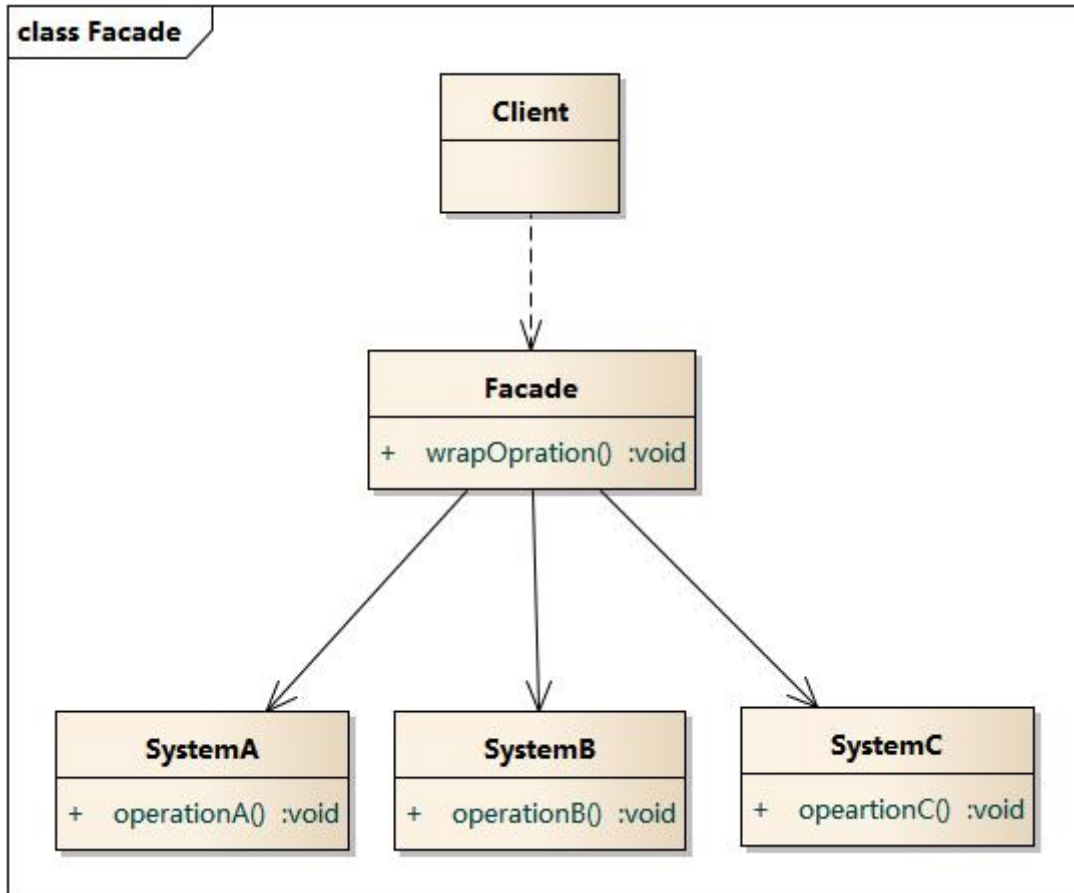
4.2. 模式定义

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。

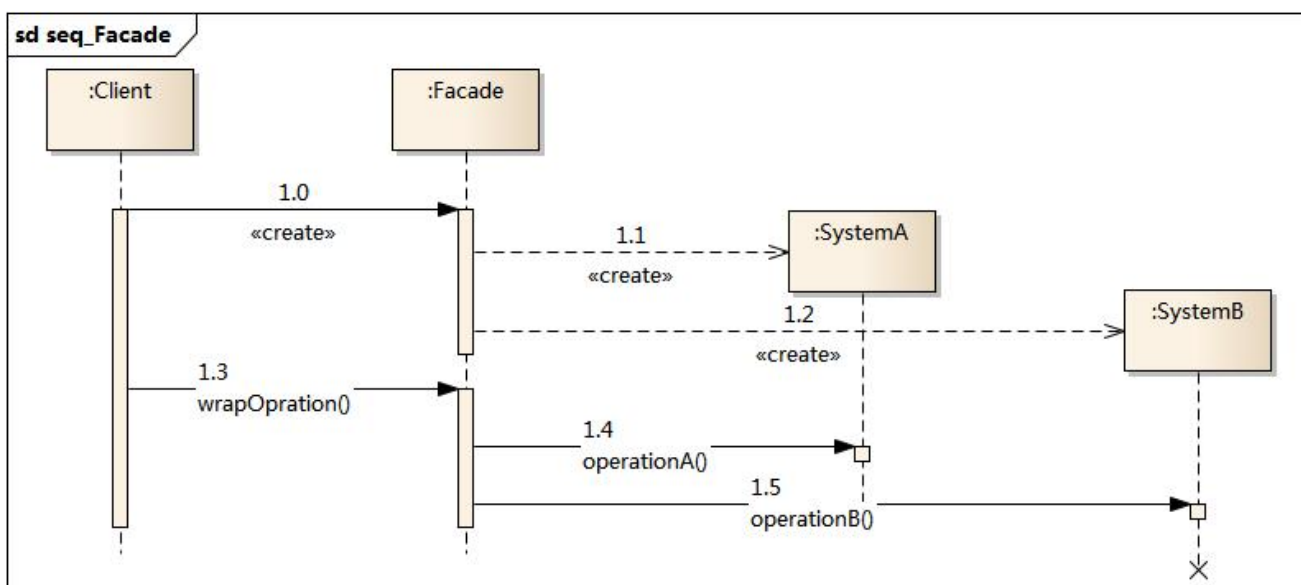
4.3. 模式结构

外观模式包含如下角色:

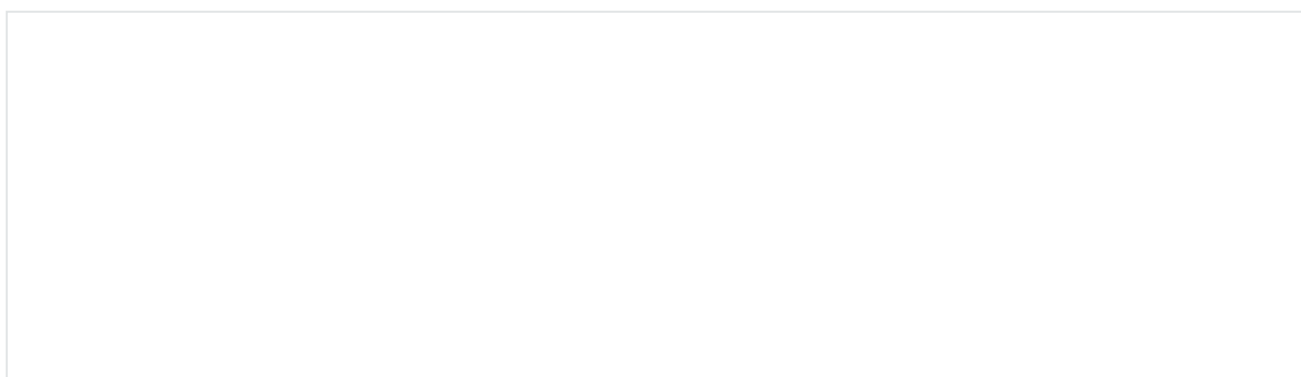
- Facade: 外观角色
- SubSystem: 子系统角色



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.wrap0pration();
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 public:
18     Facade();
19     virtual ~Facade();
20
21     void wrap0pration();
22
23 private:
24     SystemC *m_SystemC;
25     SystemA *m_SystemA;
26     SystemB *m_SystemB;
27 };
28 #endif // !defined(EA_FD130A87_92A9_4168_9B33_7A925C47AFD5__INCLUDED_)
29

```

```

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 Facade::Facade(){
11     m_SystemA = new SystemA();
12     m_SystemB = new SystemB();
13     m_SystemC = new SystemC();
14 }
15
16 Facade::~Facade(){
17     delete m_SystemA;
18     delete m_SystemB;
19     delete m_SystemC;
20 }
21
22 void Facade::wrap0pration(){
23     m_SystemA->operationA();
24     m_SystemB->operationB();
25     m_SystemC->opeartionC();
26 }
27
28
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

外观模式创造出一个外观对象，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

目录

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

5.1. 模式动机

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(IntrinsicState)，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。

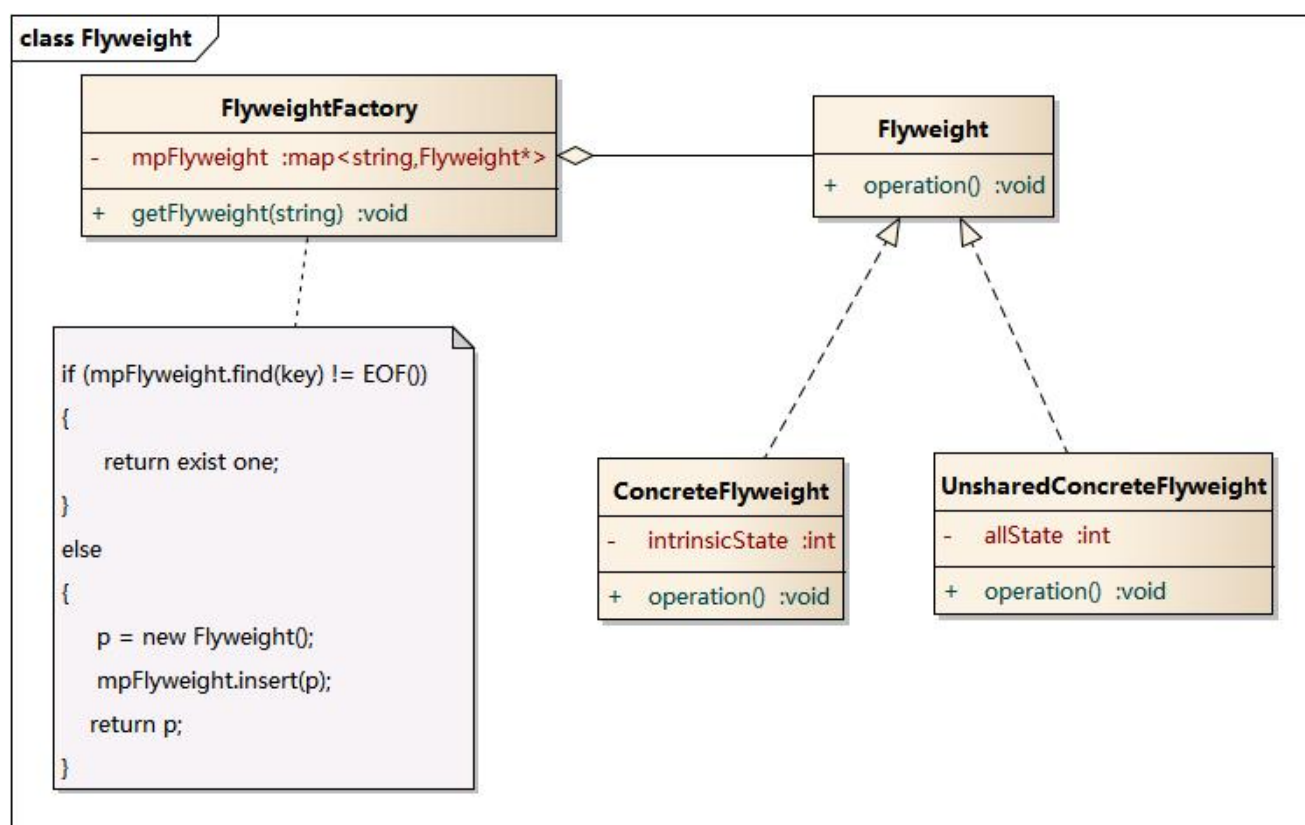
5.2. 模式定义

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。

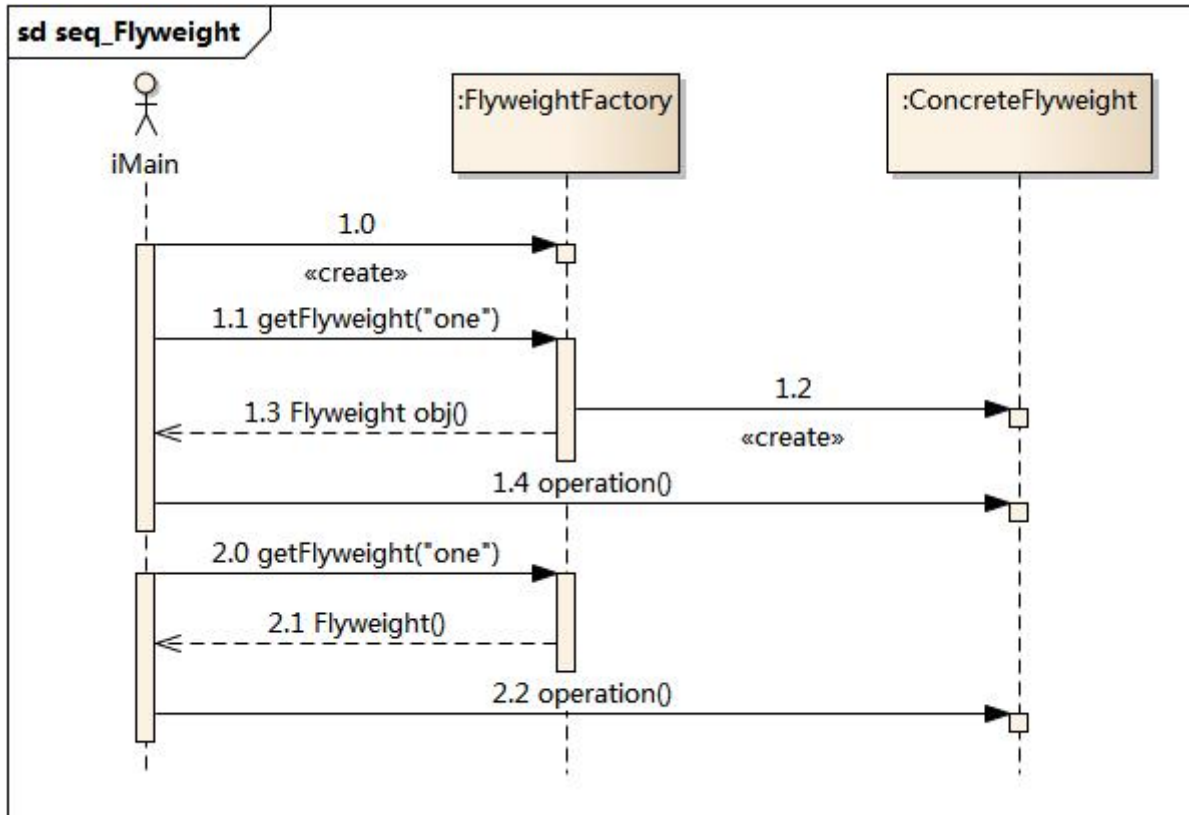
5.3. 模式结构

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     //already 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  #include "FlyweightFactory.h"
8  #include "ConcreteFlyweight.h"
9  #include <iostream>
10 using namespace std;
11
12 FlyweightFactory::FlyweightFactory(){
13
14 }
15
16 FlyweightFactory::~FlyweightFactory(){
17
18 }
19
20 Flyweight* FlyweightFactory::getFlyweight(string str){
21     map<string,Flyweight*>::iterator itr = m_mpFlyweight.find(str);
22     if(itr == m_mpFlyweight.end())
23     {
24         Flyweight * fw = new ConcreteFlyweight(str);
25         m_mpFlyweight.insert(make_pair(str,fw));
26         return fw;
27     }
28     else
29     {
30         cout << "aready in the pool,use the exist one:" << endl;
31         return itr->second;
32     }
33 }
34
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  #if !defined(EA_C0AF438E_96E4_46f1_ADEC_308EF16E11D1__INCLUDED_)
8  #define EA_C0AF438E_96E4_46f1_ADEC_308EF16E11D1__INCLUDED_
9
10 #include "Flyweight.h"
11 #include <string>
12 using namespace std;
13
14 class ConcreteFlyweight : public Flyweight
15 {
16 public:
17     ConcreteFlyweight(string str);
18     virtual ~ConcreteFlyweight();
19
20     virtual void operation();
21
22 private:
23     string intrinsicState;
24
25 };
26 #endif // !defined(EA_C0AF438E_96E4_46f1_ADEC_308EF16E11D1__INCLUDED_)
27
28

```

```

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  #include "ConcreteFlyweight.h"
8  #include <iostream>
9  using namespace std;
10
11 ConcreteFlyweight::ConcreteFlyweight(string str){
12     intrinsicState = str;
13 }
14
15 ConcreteFlyweight::~ConcreteFlyweight(){
16 }
17
18 void ConcreteFlyweight::operation(){
19     cout << "Flyweight[" << intrinsicState << "] do operation." << endl;
20 }
21
22
23

```

运行结果：

```
"C:\Users\cl\Documents\GitHub\design_patterns\code\Flyweight\mingw5\main.exe"
Flyweight[one] do operation.
Flyweight[two] do operation.
already in the pool,use the exist one:
Flyweight[one] do operation.
请按任意键继续. . .
```

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

目录

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

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

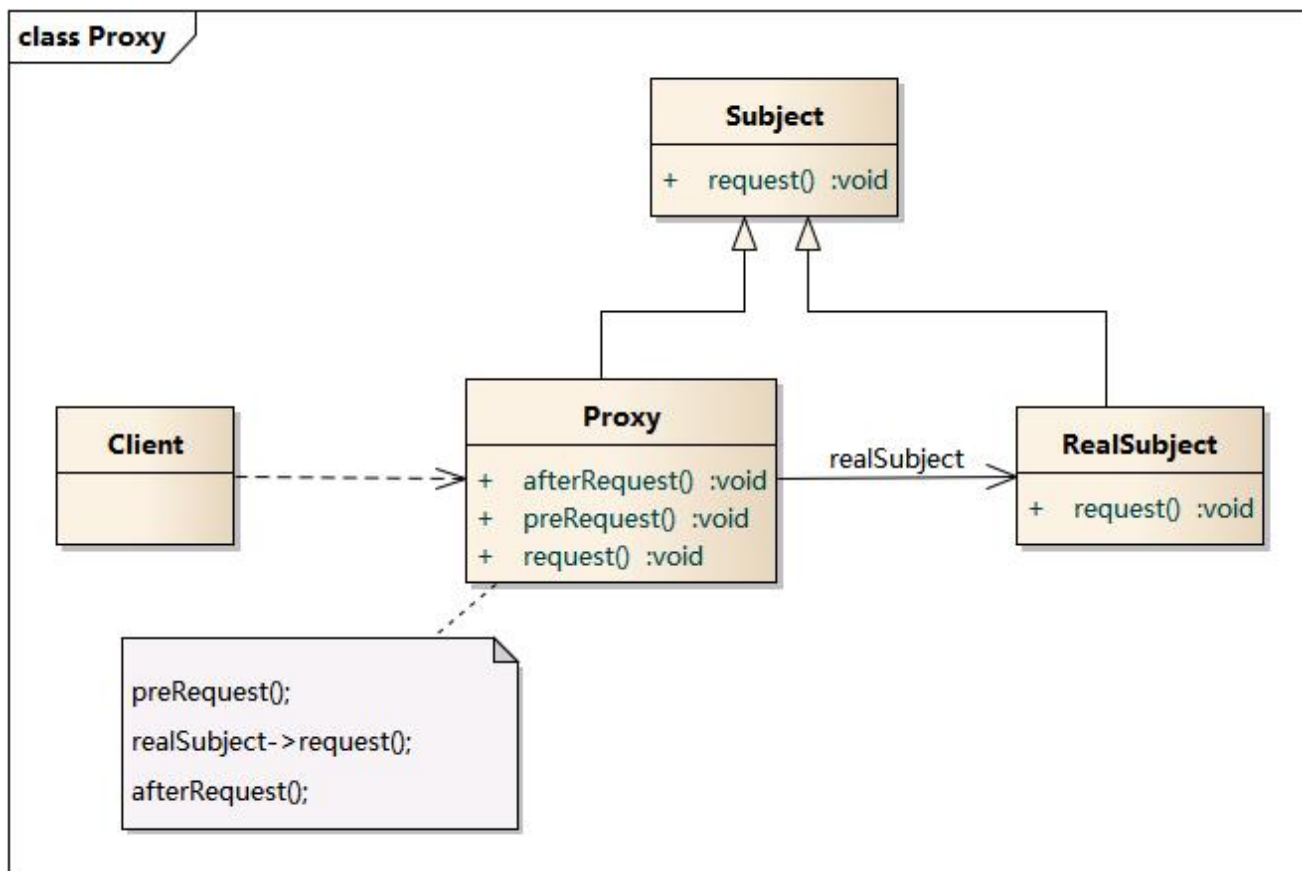
代理模式(Proxy Pattern)：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫做Proxy或Surrogate，它是一种对象结构型模式。

6.3. 模式结构

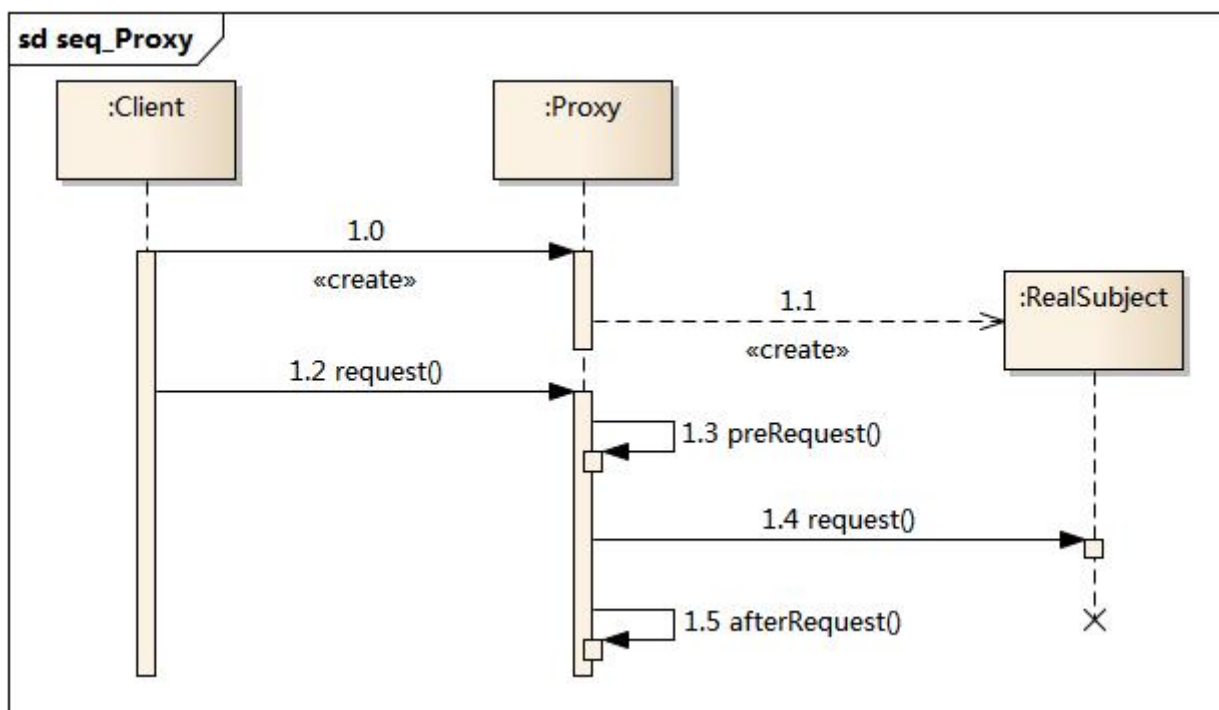
代理模式包含如下角色：

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

```

```

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  #include "Proxy.h"
8  #include <iostream>
9  using namespace std;
10
11 Proxy::Proxy(){
12     //有人觉得 RealSubject对象的创建应该是在main中实现；我认为RealSubject应该
13     //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；
14     m_pRealSubject = new RealSubject();
15 }
16
17 Proxy::~~Proxy(){
18     delete m_pRealSubject;
19 }
20
21 void Proxy::afterRequest(){
22     cout << "Proxy::afterRequest" << endl;
23 }
24
25 void Proxy::preRequest(){
26     cout << "Proxy::preRequest" << endl;
27 }
28
29 void Proxy::request(){
30     preRequest();
31     m_pRealSubject->request();
32     afterRequest();
33 }
34
35
36
37

```

运行结果：



```

"C:\Users\cl\Documents\GitHub\design_patterns\code\Proxy\mingw5\main.exe"
Proxy::preRequest
RealSubject::request
Proxy::afterRequest
请按任意键继续. . .

```

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。