

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。

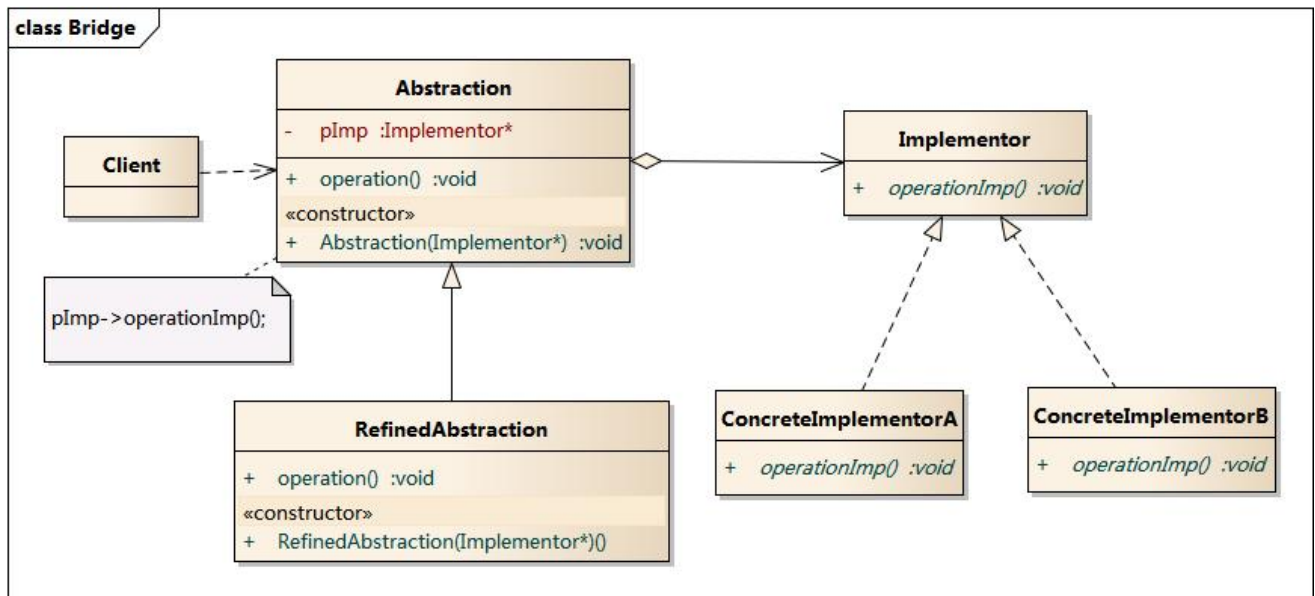
2.2. 模式定义

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。

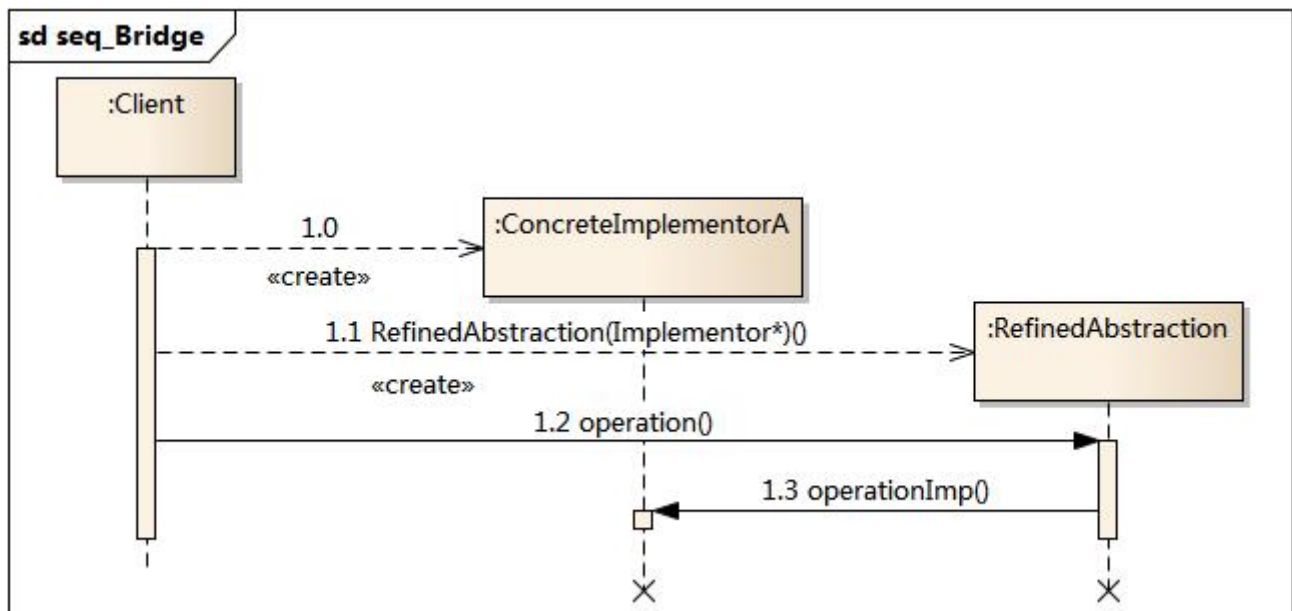
2.3. 模式结构

桥接模式包含如下角色：

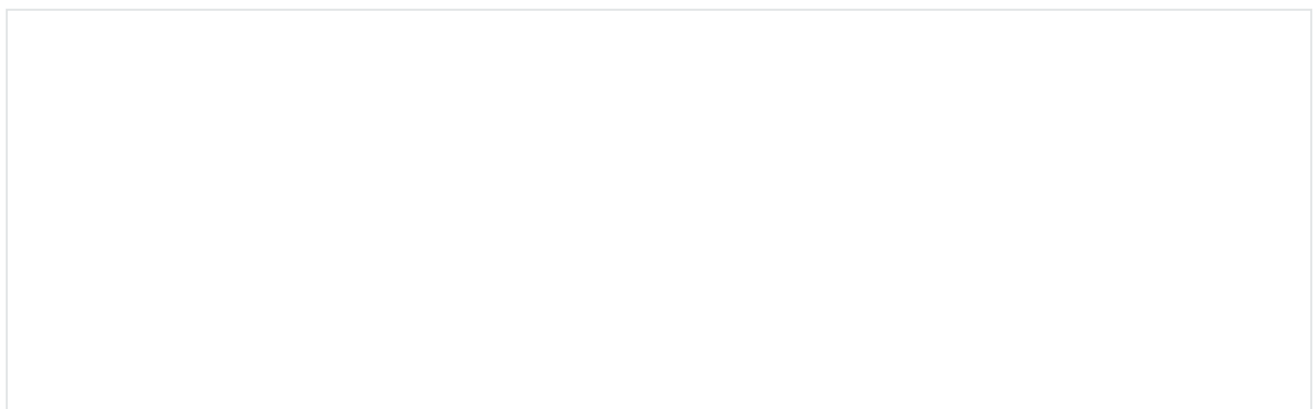
- Abstraction：抽象类
- RefinedAbstraction：扩充抽象类
- Implementor：实现类接口
- ConcreteImplementor：具体实现类



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

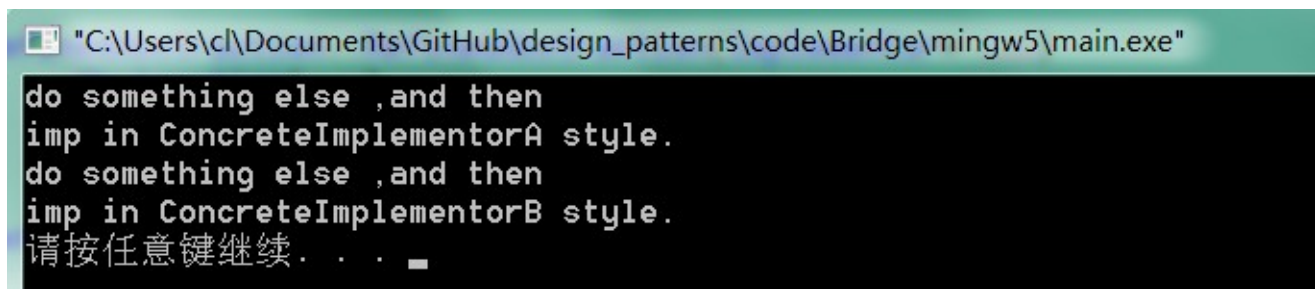
```

```

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。