## **一、引言**

在软件开发过程中，客户端程序经常会与复杂系统的内部子系统进行耦合，从而导致客户端程序随着子系统的变化而变化，然而为了将复杂系统的内部子系统与客户端之间的依赖解耦，从而就有了外观模式，也称作 ”门面“模式。下面就具体介绍下外观模式。

## **二、外观模式的详细介绍**

### **2.1 定义**

外观模式提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。使用外观模式时，我们创建了一个统一的类，用来包装子系统中一个或多个复杂的类，客户端可以直接通过外观类来调用内部子系统中方法，从而外观模式让客户和子系统之间避免了紧耦合。

### **2.2 外观模式实现**

介绍了外观模式的定义之后，让我们具体看看外观模式的由来以及实现，下面与学校中一个选课系统为例来解释外观模式，例如在选课系统中，有注册课程子系统和通知子系统，在不使用外观模式的情况下，客户端必须同时保存注册课程子系统和通知子系统两个引用，如果后期这两个子系统发生改变时，此时客户端的调用代码也要随之改变，这样就没有很好的可扩展性，下面看看不使用外观模式下选课系统的实现方式和客户端调用代码：

/// <summary>

/// 不使用外观模式的情况

/// 此时客户端与三个子系统都发送了耦合，使得客户端程序依赖与子系统

/// 为了解决这样的问题，我们可以使用外观模式来为所有子系统设计一个统一的接口

/// 客户端只需要调用外观类中的方法就可以了，简化了客户端的操作

/// 从而让客户和子系统之间避免了紧耦合

/// </summary>

class Client

{

static void Main(string[] args)

{

SubSystemA a = new SubSystemA();

SubSystemB b = new SubSystemB();

SubSystemC c = new SubSystemC();

a.MethodA();

b.MethodB();

c.MethodC();

Console.Read();

}

}

// 子系统A

public class SubSystemA

{

public void MethodA()

{

Console.WriteLine("执行子系统A中的方法A");

}

}

// 子系统B

public class SubSystemB

{

public void MethodB()

{

Console.WriteLine("执行子系统B中的方法B");

}

}

// 子系统C

public class SubSystemC

{

public void MethodC()

{

Console.WriteLine("执行子系统C中的方法C");

}

}

然而外观模式可以解决我们上面所说的问题，下面具体看看使用外观模式的实现：

/// <summary>

/// 以学生选课系统为例子演示外观模式的使用

/// 学生选课模块包括功能有：

/// 验证选课的人数是否已满

/// 通知用户课程选择成功与否

/// 客户端代码

/// </summary>

class Student

{

private static RegistrationFacade facade = new RegistrationFacade();

static void Main(string[] args)

{

if (facade.RegisterCourse("设计模式", "Learning Hard"))

{

Console.WriteLine("选课成功");

}

else

{

Console.WriteLine("选课失败");

}

Console.Read();

}

}

// 外观类

public class RegistrationFacade

{

private RegisterCourse registerCourse;

private NotifyStudent notifyStu;

public RegistrationFacade()

{

registerCourse = new RegisterCourse();

notifyStu = new NotifyStudent();

}

public bool RegisterCourse(string courseName, string studentName)

{

if (!registerCourse.CheckAvailable(courseName))

{

return false;

}

return notifyStu.Notify(studentName);

}

}

#region 子系统

// 相当于子系统A

public class RegisterCourse

{

public bool CheckAvailable(string courseName)

{

Console.WriteLine("正在验证课程 {0}是否人数已满", courseName);

return true;

}

}

// 相当于子系统B

public class NotifyStudent

{

public bool Notify(string studentName)

{

Console.WriteLine("正在向{0}发生通知", studentName);

return true;

}

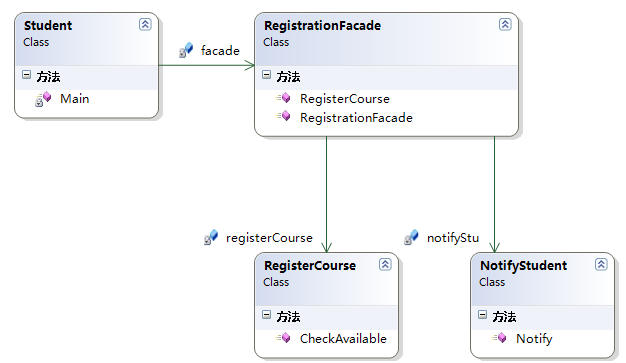
}

#endregion

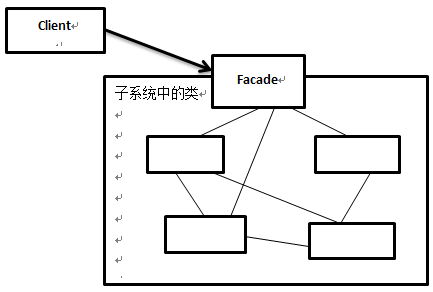
使用了外观模式之后，客户端只依赖与外观类，从而将客户端与子系统的依赖解耦了，如果子系统发生改变，此时客户端的代码并不需要去改变。**外观模式的实现核心主要是——由外观类去保存各个子系统的引用，实现由一个统一的外观类去包装多个子系统类，然而客户端只需要引用这个外观类，然后由外观类来调用各个子系统中的方法**。然而这样的实现方式非常类似适配器模式，然而外观模式与适配器模式不同的是：**适配器模式是将一个对象包装起来以改变其接口，而外观是将一群对象 ”包装“起来以简化其接口。它们的意图是不一样的，适配器是将接口转换为不同接口，而外观模式是提供一个统一的接口来简化接口**。

### **2.3 外观模式的结构**

看完外观模式的实现之后，为了帮助理清外观模式中类之间的关系，下面给出上面实现代码中类图：



然而对于外观模式而言，是没有一个一般化的类图描述，下面演示一个外观模式的示意性对象图来加深大家对外观模式的理解：



在上面的对象图中有两个角色：

**门面（Facade）角色**：客户端调用这个角色的方法。该角色知道相关的一个或多个子系统的功能和责任，该角色会将从客户端发来的请求委派带相应的子系统中去。

**子系统（subsystem）角色**：可以同时包含一个或多个子系统。每个子系统都不是一个单独的类，而是一个类的集合。每个子系统都可以被客户端直接调用或被门面角色调用。对于子系统而言，门面仅仅是另外一个客户端，子系统并不知道门面的存在。

## **三、外观的优缺点**

**优点：**

1. 外观模式对客户屏蔽了子系统组件，从而简化了接口，减少了客户处理的对象数目并使子系统的使用更加简单。
2. 外观模式实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件是紧耦合的。松耦合使得子系统的组件变化不会影响到它的客户。

**缺点：**

1. 如果增加新的子系统可能需要修改外观类或客户端的源代码，这样就违背了”开——闭原则“（不过这点也是不可避免）。

## **四、使用场景**

 在以下情况下可以考虑使用外观模式：

* 外一个复杂的子系统提供一个简单的接口
* 提供子系统的独立性
* 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。**其中三层架构就是这样的一个例子**。

## **五、总结**

到这里外观模式的介绍就结束了，外观模式，为子系统的一组接口提供一个统一的接口，该模式定义了一个高层接口，这一个高层接口使的子系统更加容易使用。并且外观模式可以解决层结构分离、降低系统耦合度和为新旧系统交互提供接口功能。