# 外观模式简介

## 概述

1. 外观模式通过引入一个外观角色来简化客户端与子系统之间的交互，为复杂的子系统调用提供一个统一的入口，使子系统与客户端的耦合度降低，且客户端调用非常方便。
2. 在软件开发中有时为了完成一项较为复杂的功能，在客户类中需要和多个业务类交互，而这些需要交互的业务类经常会作为一个整体出现，由于涉及的类比较多，导致使用时代码较为复杂，此时可以提供一个外观角色，由它来负责和多个业务类进行交互，而客户类只需要与该类交互。
3. 外观模式通过引入一个新的外观类（Facade）来实现此功能，外观类为多个业务类的调用提供了一个统一的入口，简化了类与类之间的交互。
4. 在外观模式中，那些需要交互的业务类被称为子系统（SubSystem）；外观模式中所指的子系统是一个广义的概念，它可以是一个类、一个功能模块、系统的一个组成部分或者一个完整的系统。子系统通常是一些业务类，实现了一些具体的、独立的业务功能。
5. 如果没有外观类，那么客户类需要和多个子系统之间进行复杂的交互，系统的耦合度将很大；而引入外观类之后，客户类只需要和外观类交互，客户类与子系统之间有缘的复杂引用关系由外观类来实现，从而降低了系统的耦合度；外观类将客户类与子系统的内部复杂性分隔开，使得客户类只需要与外观类打交道。
6. 外观模式（Façade Pattern）的定义如下：为子系统中的一组接口提供一个统一的入口。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。
7. 外观模式又称为门面模式，它是一种对象结构型模式。



## 结构

外观模式没有一个一般化的类图描述，通常使用以下图描述：



如图所知，外观模式包含以下2个角色：

1. Façade（外观角色）：在客户端可以调用它的方法，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；在正常情况下，它将所有从客户端发来的请求委派给到相应的子系统，传递给相应的子系统对象处理。
2. SubSystem（子系统）：在软件系统中可以有一个或者多个子系统角色，每一个子系统一般不是一个单独的类，而是一个类的集合，这些类相互搭配以完成某种功能集，实现子系统的功能；每一个子系统都可以被客户端直接调用，或者被外观角色调用，它处理由外观类传过来的请求；子系统并不知道外观的存在，对于子系统而言，外观角色仅仅是另外一个客户端而已。

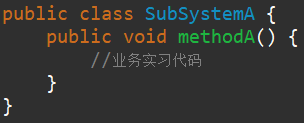
# 实现

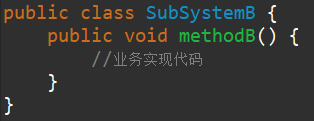
## 实现原理

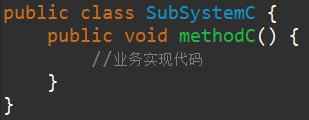
外观模式的主要目的在于降低系统的复杂程度，在面向对象软件系统中，类与类之间的关系越多，不能表示系统设计得越好，反而表示系统中类之间的耦合度太大，这样的系统在维护和修改时都缺乏灵活性，因为一个类的改动会导致多个类发生变化，而外观模式的引入在很大程度上降低了类与类之间的耦合关系。

在引入外观模式之后，增加新的子系统或者移除子系统都非常方便，客户类无需进行修改（或者极少的修改），只需要在外观类中增加或移除对子系统的引用即可。从一点来说，外观模式在一定程度上并不符合开闭原则。

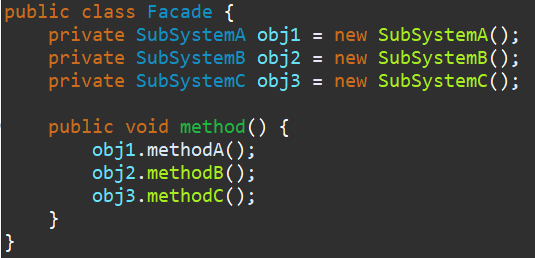
1. 外观模式中所指的子系统是一个广义的概念，它可以是一个类、一个功能模块、系统的一个组成部分或者一个完整的系统。子系统类通常是一些业务类，实现了一些具体的、独立的业务功能，其典型代码如下：





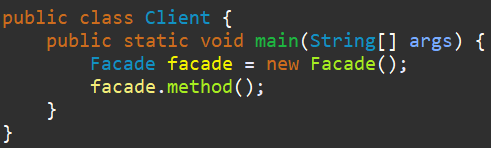


1. 在引入外观类之后，与子系统业务类之间的交互统一由外观类来完成，在外观类中通常存在如下代码：



由于在外观类中维持了对子系统对象的引用，客户端可以通过外观类来间接调用子系统对象的业务方法，而无须与子系统对象直接交互。

1. 引入外观类后，客户端代码变得非常简单，典型代码如下：



## 实例

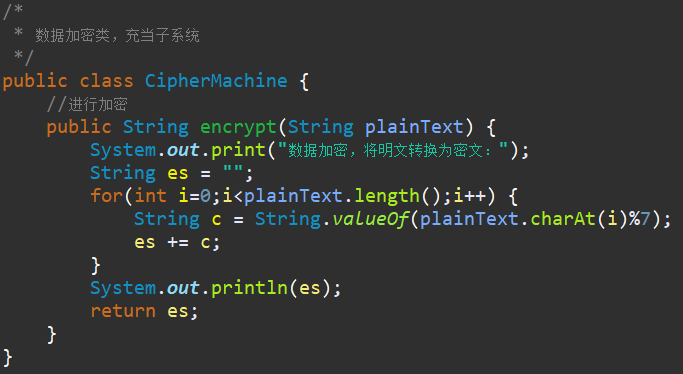
实例说明：某软件公司要开发一个可应用于多个软件的文件加密模块，该模块可以对文件中的数据进行加密并将加密之后的数据存储在一个新文件中，具体的流程包括3个部分，分别是读取源文件、加密、保存加密之后的文件，其中读取文件和保存文件使用流来实现，加密操作通过求模运算实现。这3个操作相对独立，为了实现代码的独立重用，让设计更符合单一职责原则，这3个操作的业务代码封装在3个不同的类中。EncryptFacade充当外观类，FileReader、CipherMachine和FileWriter充当子系统类，其结构图如下：



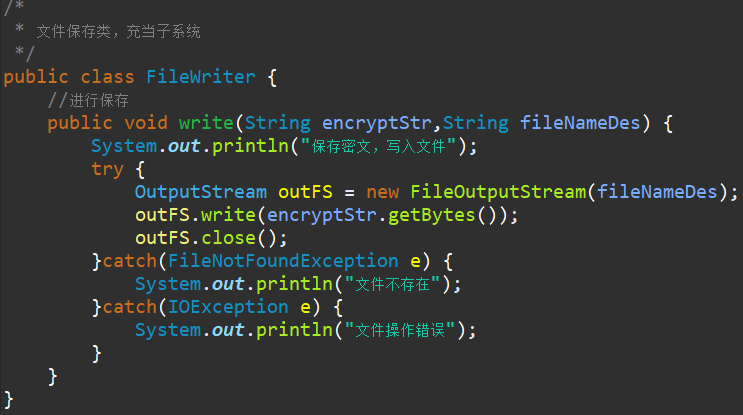
1. FileReader：文件读取类，充当子系统。



1. CipherMachine：数据加密类，充当子系统。



1. FileWriter：文件保存类，充当子系统。



1. EncryptFacade：加密外观类，充当外观类。



1. Client：客户端测试类，测试外观模式。



运行结果如下：



# 抽象外观类

## 引入

在标准的外观模式中，如果需要增加、删除或更换与外观类交互的子系统类，必须修改外观类或客户端源代码，这将违背开闭原则，因此可以通过引入抽象外观类对系统进行改进。在引入抽象外观类之后，客户端可以针对抽象外观类进行编程，对于新的业务需求，不需要修改原有外观类，而对应增加一个新的具体外观类，由新的外观类来关联新的子系统对象。

## 演示

在上一标准外观模式的实例中，其加密类使用的是基于求模运算的加密类CipherMachine，而现在需要使用基于位运算的新加密类NewCipherMachine，其代码如下：



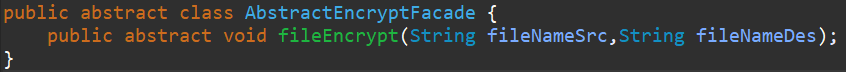
如果不增加新的外观类，只能通过修改原有外观类EncryptFacade的源代码来实现加密类的更换，将原有的对CipherMachine类型对象的引用改为对NewCipherMachine类型对象的引用，这违背了开闭原则，因此需要通过增加新的外观类来实现对子系统对象引

用的改变。

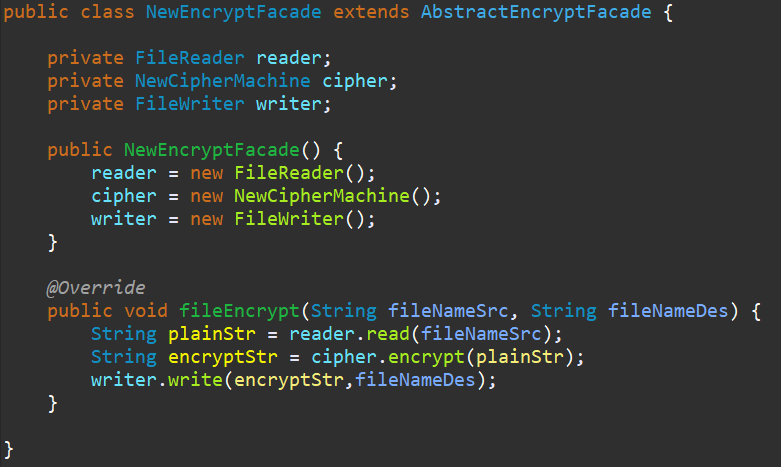
如果增加一个新的外观类NewEncryptFacade与FileReader、FileWriter以及要使用的新加密类NewCipherMachine类进行交互，虽然原有系统类库无须做任何修改，但是因为客户端代码中原来针对EncryptFacade类进行编程，现在需要改为NewEncryptFacade类，所以需要修改客户端源代码。

如何在不修改客户端代码的前提下使用新的外观类？解决方法之一是引入一个抽象外观类，客户端针对抽象外观类编程，而在运行时再确定具体外观类。部分代码如下：

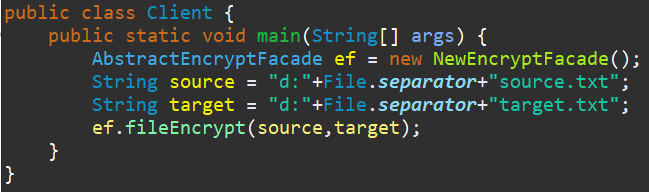
1. 抽象外观类：



1. 新增具体加密外观类，继承抽象外观类：



1. 客户端针对抽象外观类编程：



# 优缺点及适用环境

1. 外观模式通过引入一个外观角色来简化客户端与子系统之间的交互，为复杂的子系统调用提供一个统一的入口，使子系统与客户端的耦合度降低，且客户端调用非常方便。
2. 外观模式并不给系统增加任何新功能，它仅仅是简化调用接口。
3. 所有涉及与多个业务对象交互的场景都可以考虑使用外观模式进行重构，例如Java EE中的Session外观模式。

## 优点

外观模式的优点主要如下：

1. 它对客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使子系统使用起来更加容器以。通过引入外观模式，客户端代码将变得很简单，与之关联的对象也很少。
2. 它实现了子系统与客户端之间的松耦合关系，这使得子系统的变化不会影响到调用它的客户端，只需要调整外观类即可。
3. 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。

## 缺点

外观模式的缺点主要如下：

1. 不能很好地限制客户端直接使用子系统类，如果对客户端访问子系统类做太多的限制则减少了可变性和灵活性。
2. 如果设计不当，增加新的子系统可能需要修改外观类的源代码，违背了开闭原则。

## 适用环境

在以下情况下可以考虑使用外观模式：

1. 当要为访问一系列复杂的子系统提供一个简单入口时可以使用外观模式。
2. 客户端程序与多个子系统之间存在很大的依赖性。引入外观类可以将子系统与客户端解耦，从而提高子系统的独立性和可移植性。
3. 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。