# Facade Pattern

**Motivation**

Imagine that you must make your code work with *a broad set of objects* that belong to a sophisticated library or framework. Ordinarily, you’d need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.

As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

假设你必须在代码中使用某个复杂的库或框架中的众多对象。 正常情况下， 你需要负责所有对象的初始化工作、 管理其依赖关系并按正确的顺序执行方法等。

最终， 程序中类的业务逻辑将与第三方类的实现细节紧密耦合， 使得理解和维护代码的工作很难进行。

## Solution

A facade is a class that provides a *simple interface* to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

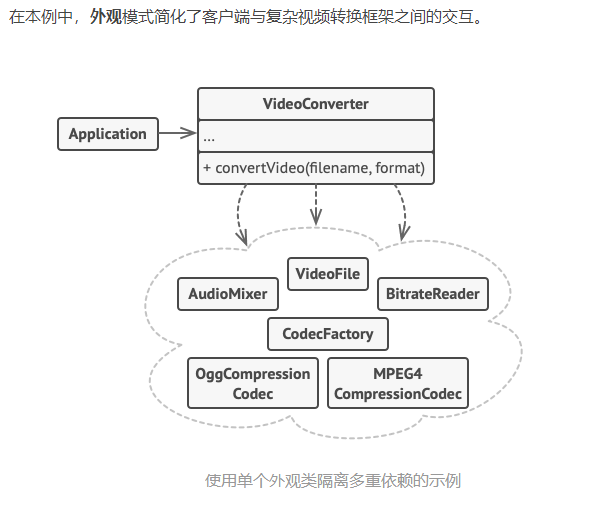
Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method encode(filename, format). After creating such a class and connecting it with the video conversion library, you’ll have your first facade.

外观类为包含许多活动部件的复杂子系统提供一个简单的接口。 与直接调用子系统相比， 外观提供的功能可能比较有限， 但它却包含了客户端真正关心的功能。

如果你的程序需要与包含几十种功能的复杂库整合， 但只需使用其中非常少的功能， 那么使用外观模式会非常方便，

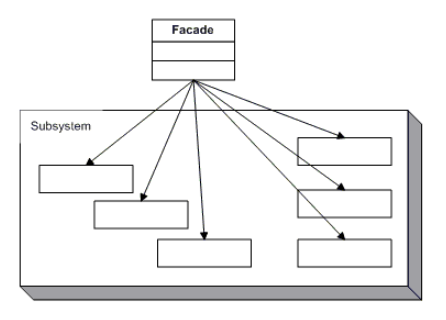
例如， 上传猫咪搞笑短视频到社交媒体网站的应用可能会用到专业的视频转换库， 但它只需使用一个包含 encode­(filename, format)方法 （以文件名与文件格式为参数进行编码的方法） 的类即可。 在创建这个类并将其连接到视频转换库后， 你就拥有了自己的第一个外观。

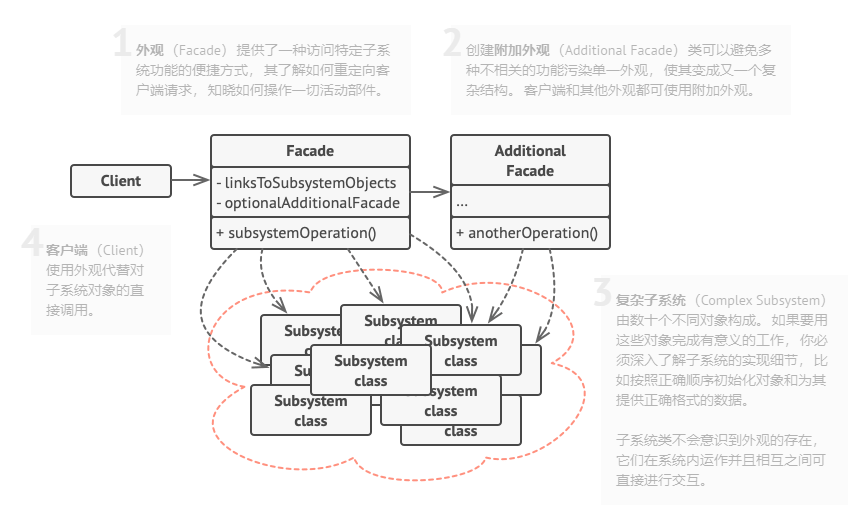


**Intent**

#Definition1 The Facade design pattern provides a unified interface to a set of interfaces in a subsystem. This pattern defines a higher-level interface that makes the subsystem easier to use.

#Definition2 **Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

****

****

**Participants**

The classes and objects participating in this pattern include:

* Facade   (MortgageApplication)
  + knows which subsystem classes are responsible for a request.
  + **delegates** client requests to appropriate subsystem objects.
* Subsystem classes   (Bank, Credit, Loan)
  + implement subsystem functionality.
  + handle work assigned by the Facade object.
  + have no knowledge of the facade and keep **no reference** to it.

**场景：**

* 当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越

来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，

也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用

上的困难。Facade 可以提供一个简单的缺省视图，这一视图对大多数用户来说已

经足够，而那些需要更多的可定制性的用户可以越过 Facade 层。

* 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入 Facade 将这个子系

统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

* 当你需要构建一个层次结构的子系统时，使用 Facade 模式定义子系统中每层的入

口点。如果子系统之间是相互依赖的，你可以让它们仅通过 Facade 进行通讯，从

* 如果外观变得[过于臃肿](https://refactoringguru.cn/smells/large-class)， 你可以考虑将其部分行为抽取为一个新的专用外观类。

**与其他模式的关系**

* [**外观模式**](https://refactoringguru.cn/design-patterns/facade)为现有对象定义了一个新接口， [**适配器模式**](https://refactoringguru.cn/design-patterns/adapter)则会试图运用已有的接口。 适配器通常只封装一个对象， 外观通常会作用于整个对象子系统上。
* 当只需对客户端代码隐藏子系统创建对象的方式时， 你可以使用[**抽象工厂模式**](https://refactoringguru.cn/design-patterns/abstract-factory)来代替[**外观**](https://refactoringguru.cn/design-patterns/facade)。
* [**享元模式**](https://refactoringguru.cn/design-patterns/flyweight)展示了如何生成大量的小型对象， [**外观**](https://refactoringguru.cn/design-patterns/facade)则展示了如何用一个对象来代表整个子系统。
* [**外观**](https://refactoringguru.cn/design-patterns/facade)和[**中介者模式**](https://refactoringguru.cn/design-patterns/mediator)的职责类似： 它们都尝试在大量紧密耦合的类中组织起合作。
  + 外观为子系统中的所有对象定义了一个简单接口， 但是它不提供任何新功能。 子系统本身不会意识到外观的存在。 子系统中的对象可以直接进行交流。
  + 中介者将系统中组件的沟通行为中心化。 各组件只知道中介者对象， 无法直接相互交流。
* [**外观**](https://refactoringguru.cn/design-patterns/facade)类通常可以转换为[**单例模式**](https://refactoringguru.cn/design-patterns/singleton)类， 因为在大部分情况下一个外观对象就足够了。
* [**外观**](https://refactoringguru.cn/design-patterns/facade)与[**代理模式**](https://refactoringguru.cn/design-patterns/proxy)的相似之处在于它们都缓存了一个复杂实体并自行对其进行初始化。 代理与其服务对象遵循同一接口， 使得自己和服务对象可以互换， 在这一点上它与外观不同。