

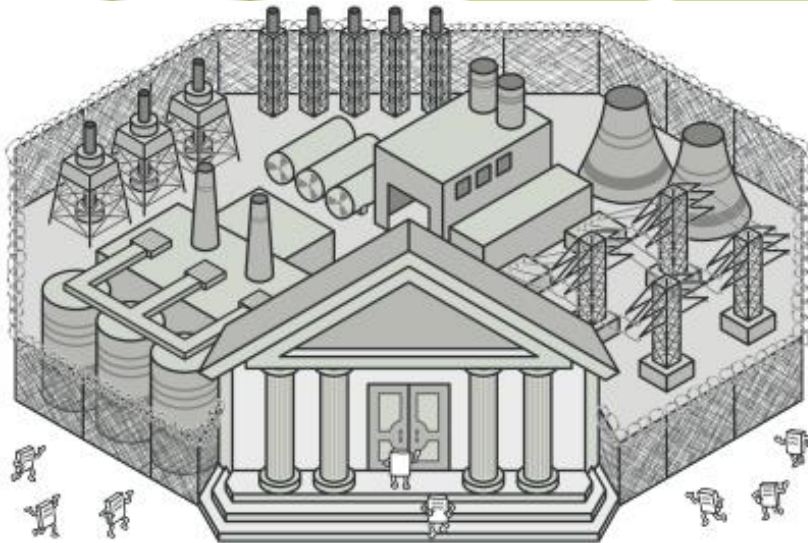
Design Patterns(SE-408)

Course Teacher

Assistant Professor

Engr. Mustafa Latif

1



FACADE PATTERN

2



Facade Pattern

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

3



Problem

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.

4

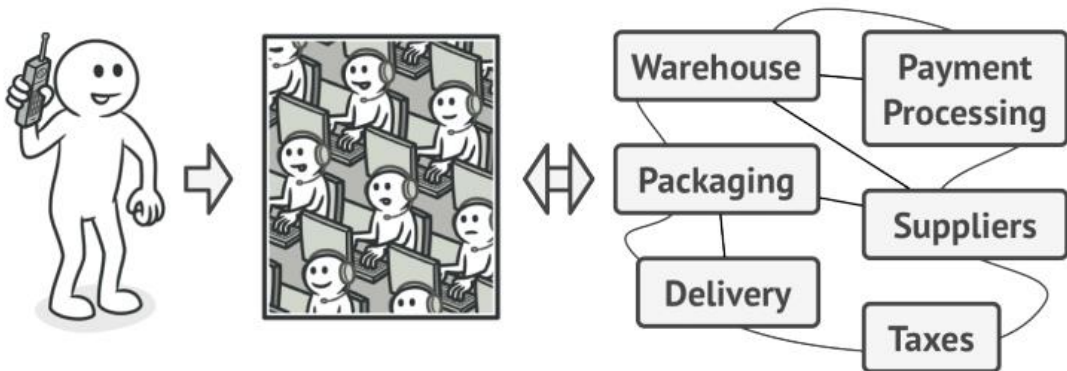
Solution

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A façade 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

5

Real-World Analogy



Placing orders by phone.

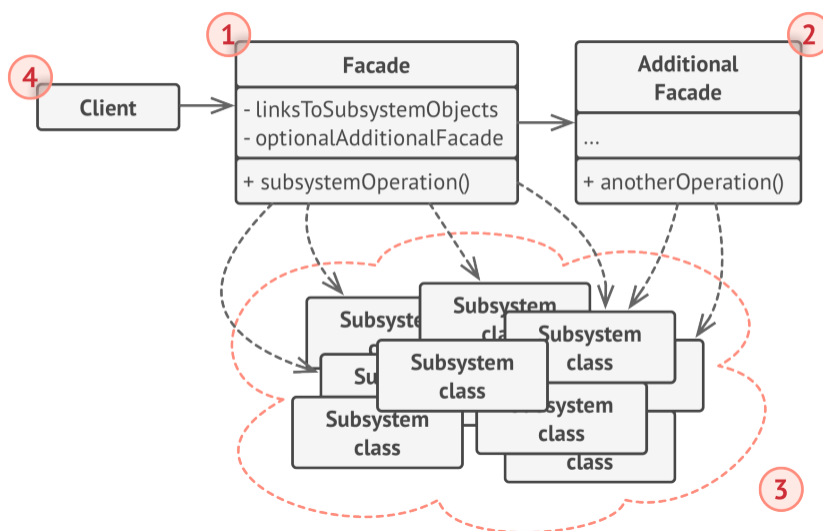
6

Real-World Analogy


- When you call a shop to place a phone order, an operator is your facade to all services and departments of the shop. The operator provides you with a simple voice interface to the ordering system, payment gateways, and various delivery services.


7

Structure



8

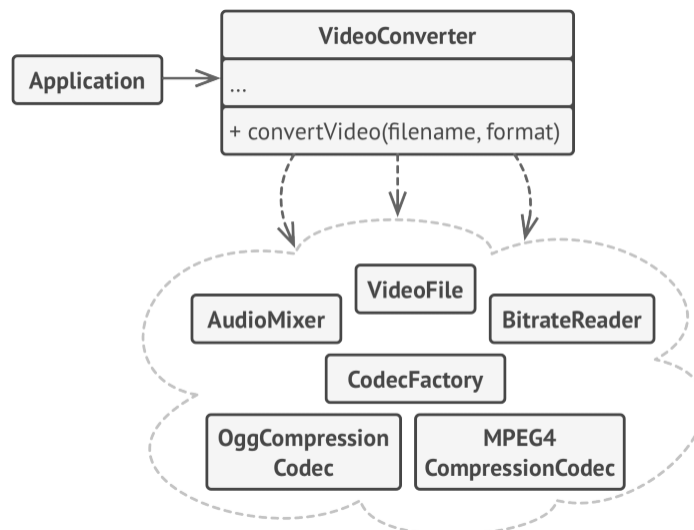
- 
- The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.
 - An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.
 - The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format. **Subsystem classes** aren't aware of the facade's existence. They operate within the system and work with each other directly
- 9

- 
- The **Client** uses the facade instead of calling the subsystem objects directly

Pseudocode

- In this example, the **Facade** pattern simplifies interaction with a complex video conversion framework.

11



An example of isolating multiple dependencies within a single facade class.

12


Pseudocode

- Instead of making your code work with dozens of the framework classes directly, you create a facade class which encapsulates that functionality and hides it from the rest of the code. This structure also helps you to minimize the effort of upgrading to future versions of the framework or replacing it with another one. The only thing you'd need to change in your app would be the implementation of the facade's methods.

13


```
1 // These are some of the classes of a complex 3rd-party video
2 // conversion framework. We don't control that code, therefore
3 // can't simplify it.
4
5 class VideoFile
6 // ...
7 class OggCompressionCodec
8 // ...
9
10 class MPEG4CompressionCodec
11 // ...
12
13 class CodecFactory
14 // ...
15
16 class BitrateReader
17 // ...
18
19 class AudioMixer
20 // ...
21
```

14



```
22
23 // We create a facade class to hide the framework's complexity
24 // behind a simple interface. It's a trade-off between
25 // functionality and simplicity.
26 class VideoConverter is
27 method convert(filename, format):File is
28 file = new VideoFile(filename)
29 sourceCodec = new CodecFactory.extract(file)
30 if (format == "mp4")
31 destinationCodec = new MPEG4CompressionCodec()
32 else
33 destinationCodec = new OggCompressionCodec()
34 buffer = BitrateReader.read(filename, sourceCodec)
35 result = BitrateReader.convert(buffer, destinationCodec)
36 result = (new AudioMixer()).fix(result)
37 return new File(result)
38
```

15

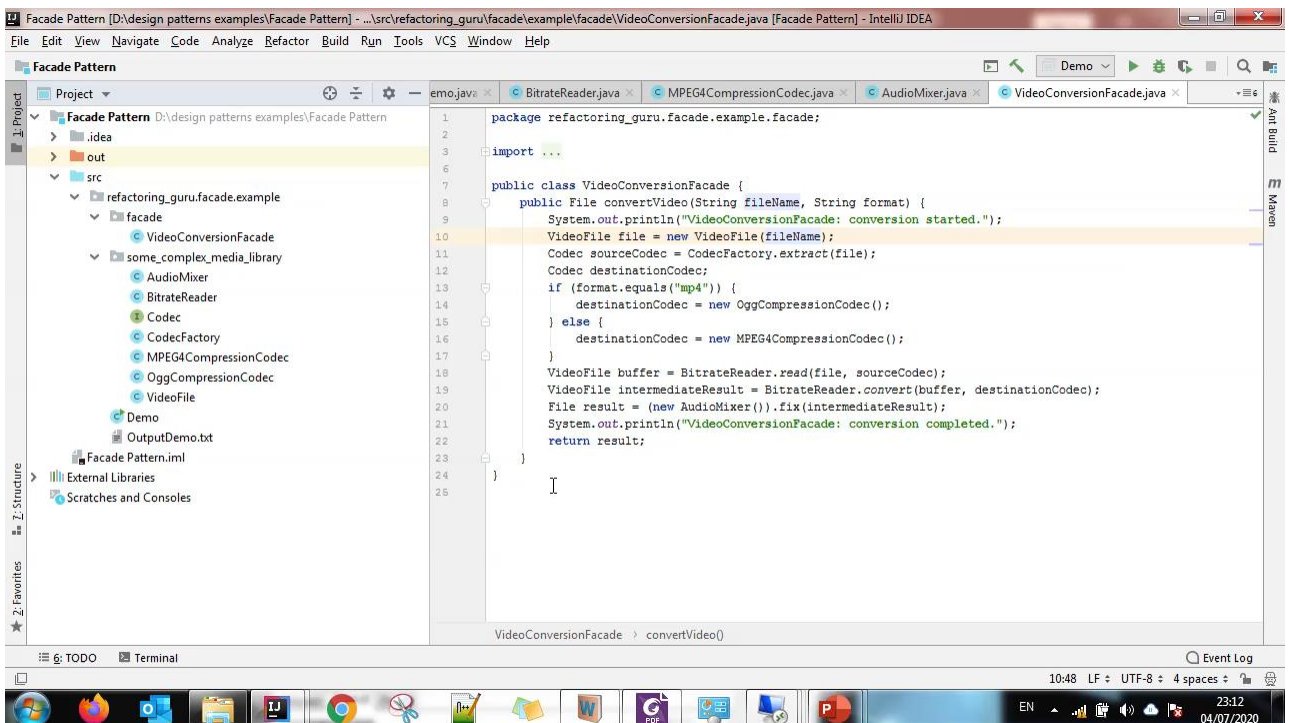


```
39 // Application classes don't depend on a billion classes
40 // provided by the complex framework. Also, if you decide to
41 // switch frameworks, you only need to rewrite the facade class.
42 class Application is
43 method main() is
44 convertor = new VideoConverter()
45 mp4 = convertor.convert("youtubevideo.ogg", "mp4")
46 mp4.save()
```

16

Java Code Example

17



The screenshot displays the IntelliJ IDEA IDE interface. The top menu bar includes File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, and Help. The title bar shows the project name 'Facade Pattern' and the file path 'D:\design patterns examples\Facade Pattern'. The left sidebar contains a Project view showing the directory structure: 'Facade Pattern' (root) with subdirectories '.idea', 'out', and 'src'. Under 'src', there is a package 'refactoring_guru.facade.example' containing a 'facade' package and a 'some_complex_media_library' package. The 'facade' package contains 'VideoConversionFacade' and 'Demo'. The 'some_complex_media_library' package contains 'AudioMixer', 'BitrateReader', 'Codec', 'CodecFactory', 'MPEG4CompressionCodec', 'OggCompressionCodec', and 'VideoFile'. The right pane shows the source code of 'VideoConversionFacade.java'. The code is as follows:

```
1 package refactoring_guru.facade.example.facade;
2
3 import ...
4
5
6
7 public class VideoConversionFacade {
8     public File convertVideo(String fileName, String format) {
9         System.out.println("VideoConversionFacade: conversion started.");
10        VideoFile file = new VideoFile(fileName);
11        Codec sourceCodec = CodecFactory.extract(file);
12        Codec destinationCodec;
13        if (format.equals("mp4")) {
14            destinationCodec = new OggCompressionCodec();
15        } else {
16            destinationCodec = new MPEG4CompressionCodec();
17        }
18        VideoFile buffer = BitrateReader.read(file, sourceCodec);
19        VideoFile intermediateResult = BitrateReader.convert(buffer, destinationCodec);
20        File result = (new AudioMixer()).fix(intermediateResult);
21        System.out.println("VideoConversionFacade: conversion completed.");
22        return result;
23    }
24 }
25
```

The bottom status bar shows 'VideoConversionFacade > convertVideo()' and 'Event Log'. The Windows taskbar at the bottom shows the time as 10:48, date as 04/07/2020, and various application icons.



Applicability

- Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.
- Use the Facade when you want to structure a subsystem into layers.

19



How to Implement

1. Check whether it's possible to provide a simpler interface than what an existing subsystem already provides. You're on the right track if this interface makes the client code independent from many of the subsystem's classes.
2. Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.
3. To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade. Now the client code is protected from any changes in the subsystem code. For example, when a subsystem gets upgraded to a new version, you will only need to modify the code in the facade.
4. If the facade becomes **too big**, consider extracting part of its behavior to a new, refined facade class.

20



Pros and Cons

- You can isolate your code from the complexity of a subsystem.
- A facade can become a **god object** coupled to all classes of an app.

21



Review

22