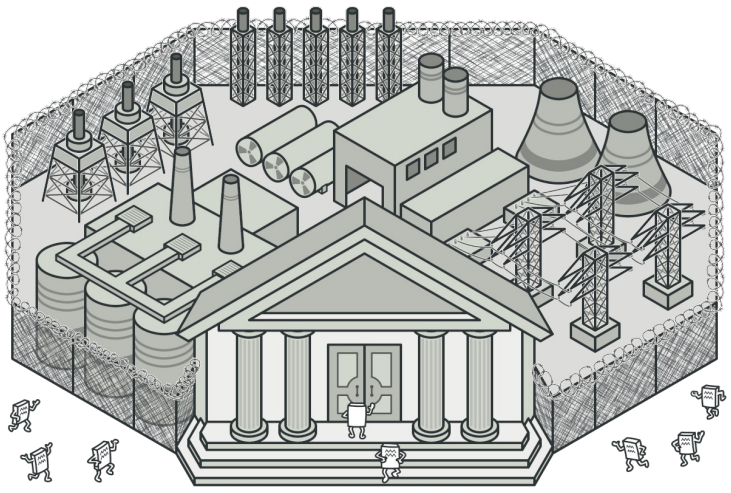
Advanced Software Engineering Part 07 – Facade Pattern

Dr. Amjad AbuHassan

Intent

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



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.

Solution

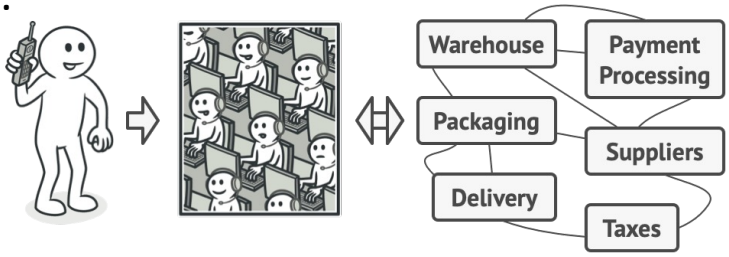
- A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts.
- A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Solution cont.

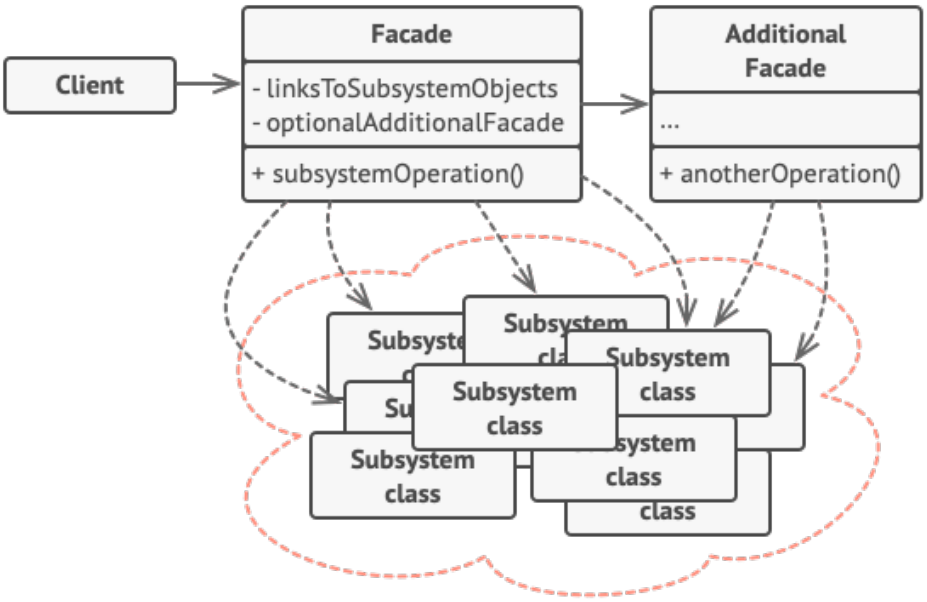
- If you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.
- For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all 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.

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.

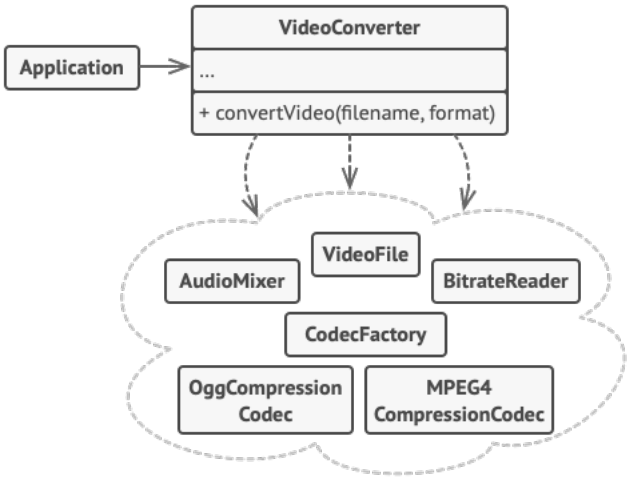


Structure



Pseudocode

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



Pseudocode cont.

```
public class VideoFile {
   private String name;
   private String codecType;

   public VideoFile(String name) {
      this.name = name;
      this.codecType = name.substring(name.indexOf(".") + 1);
   }
   public String getCodecType() {
      return codecType;
   }
   public String getName() {
      return name;
   }
}
```


Pseudocode cont.

```
public interface Codec {
}

public class MPEG4CompressionCodec implements Codec {
   public String type = "mp4";
}

public class OggCompressionCodec implements Codec {
   public String type = "ogg";
}
```

Pseudocode cont.

```
public class CodecFactory {
   public static Codec extract(VideoFile file) {
      String type = file.getCodecType();
      if (type.equals("mp4")) {
         System.out.println("CodecFactory: extracting mpeg audio...");
         return new MPEG4CompressionCodec();
      }
      else {
         System.out.println("CodecFactory: extracting ogg audio...");
         return new OggCompressionCodec();
      }
   }
}
```


Pseudocode cont.

```
public class BitrateReader {
   public static VideoFile read(VideoFile file, Codec codec) {
      System.out.println("BitrateReader: reading file...");
      return file;
   }

   public static VideoFile convert(VideoFile buffer, Codec codec) {
      System.out.println("BitrateReader: writing file...");
      return buffer;
   }
}
```

Pseudocode cont.

```
public class AudioMixer {
    public File fix(VideoFile result){
        System.out.println("AudioMixer: fixing audio...");
        return new File("tmp");
    }
}
```


Pseudocode cont.

```
public class VideoConversionFacade {
    public File convertVideo(String fileName, String format) {
        System.out.println("VideoConversionFacade: conversion started.");
        VideoFile file = new VideoFile(fileName);
        Codec sourceCodec = CodecFactory.extract(file);
        Codec destinationCodec;
        if (format.equals("mp4")) {
            destinationCodec = new MPEG4CompressionCodec();
        } else {
            destinationCodec = new OggCompressionCodec();
        }
        VideoFile buffer = BitrateReader.read(file, sourceCodec);
        VideoFile intermediateResult = BitrateReader.convert(buffer, destinationCodec);
        File result = (new AudioMixer()).fix(intermediateResult);
        System.out.println("VideoConversionFacade: conversion completed.");
        return result;
    }
}
```

Pseudocode cont.

```
public class Demo {
    public static void main(String[] args) {
        VideoConversionFacade converter = new VideoConversionFacade();
        File mp4Video = converter.convertVideo("youtubevideo.ogg", "mp4");
        // ...
    }
}
```

```
VideoConversionFacade: conversion started.
CodecFactory: extracting ogg audio...
BitrateReader: reading file...
BitrateReader: writing file...
AudioMixer: fixing audio...
VideoConversionFacade: conversion completed.
```


Applicability

Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.

- Often, subsystems get more complex over time. Even applying design patterns typically leads to creating more classes. A subsystem may become more flexible and easier to reuse in various contexts, but the amount of configuration and boilerplate code it demands from a client grows ever larger. The Facade attempts to fix this problem by providing a shortcut to the most-used features of the subsystem which fit most client requirements.

Applicability cont.

Use the Facade when you want to structure a subsystem into layers.

- Create facades to define entry points to each level of a subsystem. You can reduce coupling between multiple subsystems by requiring them to communicate only through facades.
- In video conversion example. It can be broken down into two layers: video- and audio-related. For each layer, you can create a facade and then make the classes of each layer communicate with each another via those facades.