CCSW 413 Lab 5: Java Adapter and Facade Design Patterns Lab

March 9, 2024

1 Introduction

1.1 Overview

This Lab session aims to provide a comprehensive understanding of Adapter and Facade Patterns through real-world scenarios and hands-on implementation. In this lab, we will explore two structural design patterns: Adapter and Facade. These patterns help in connecting incompatible interfaces and simplifying complex subsystems, respectively.

1.2 Objective

The purpose of this lab is to equip students with practical knowledge and skills in applying the Java Adapter and Facade Design Patterns. By the end of this session, you'll understand the benefits of using the Adapter and Facade Design Patterns.

2 Lab Session 1: Adapter Design Pattern

Definition

The Adapter pattern allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, making them compatible without changing their original code.

Benefits

- **Reusability:** Allows the reuse of existing code with different interfaces.
- **Flexibility:** Adapts existing classes to work with new interfaces, providing flexibility in system design.
- **Interoperability:** Facilitates the collaboration of systems with incompatible interfaces.

When to Use

Use the Adapter pattern when:

- Integrating existing systems with different interfaces.
- Reusing existing classes in a new system with a different interface.
- Creating a system that should work with multiple third-party libraries with varying interfaces.

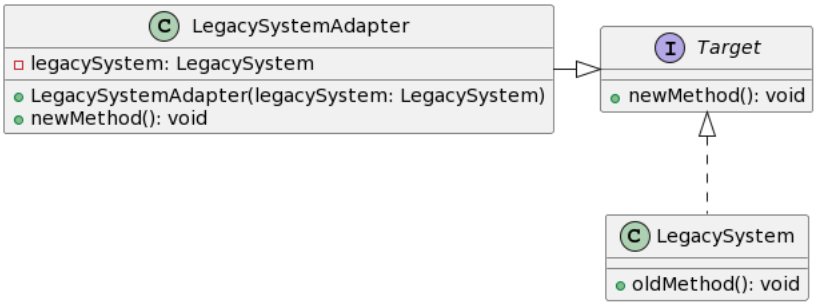

Figure 1: Class Diagrams for Integrating Existing Systems with Different Interfaces

Problem Scenario Example 1

Problem Description

Consider a legacy system that uses a class with a method `oldMethod()`. We want to integrate this with a new system that expects a method `newMethod()`. The Adapter pattern can help bridge this gap.

Java Code Example

```java
// Existing System
class OldSystem {
    void legacyMethod() {
        System.out.println("Executing legacy method");
    }
}

// Target Interface
interface NewInterface {
    void newMethod();
}

// Adapter
class Adapter implements NewInterface {
    private OldSystem oldSystem;

    public Adapter(OldSystem oldSystem) {
        this.oldSystem = oldSystem;
    }

    @Override
    public void newMethod() {
        oldSystem.legacyMethod();
    }
}

// Client Code
public class AdapterExample {
    public static void main(String[] args) {
        OldSystem oldSystem = new OldSystem();
        NewInterface adapter = new Adapter(oldSystem);
        adapter.newMethod();
    }
}
```

Listing 1: Integrating existing systems with different interfaces using Adapter design pattern

Guided Code Walkthrough

In this Adapter pattern example, we have an existing class `LegacySystem` with a method `oldMethod()`. To integrate it with a new system that expects `newMethod()`, we create an adapter class `LegacySystemAdapter`. This adapter class implements the expected interface (`NewSystem`) and delegates calls to `oldMethod()` to maintain compatibility.

In the client code (`AdaptableSystemClient`), we see how seamlessly the legacy system is integrated with the new system without modifying the existing code. The Adapter pattern allows the client code to work with different systems through a common interface.

In Lab Task 1

Extend the Adapter pattern example to adapt a different legacy system with a distinct method signature.

Lab Session 2: Facade Design Pattern

Definition

The Facade pattern provides a simplified interface to a subsystem, defining a higher-level interface that makes the subsystem easier to use.

- **Simplicity:** Simplifies the complex interactions of a subsystem, providing an easy-to-use interface.
- **Decoupling:** Reduces dependencies by providing a single entry point to the subsystem, decoupling the client from its components.
- **Maintenance:** Eases maintenance by encapsulating subsystem changes within the Facade.

When to Use

Use the Facade pattern when:

- Providing a simple interface to a complex subsystem.
- Decoupling the client from the inner workings of a subsystem.
- Creating a unified interface for a set of interfaces in a subsystem.

Problem Scenario Example 1

Problem Description

Consider a multimedia subsystem with components like `AudioPlayer`, `VideoPlayer`, and `ImageLoader`. To simplify usage, we create a `MultimediaFacade`.

Java Code Example

```
// Subsystem Components
class AudioPlayer {
    void playAudio(String audioFile) {
        System.out.println("Playing audio: " + audioFile);
    }
}

class VideoPlayer {
    void playVideo(String videoFile) {
        System.out.println("Playing video: " + videoFile);
    }
}

class ImageLoader {
    void loadImage(String imageFile) {
```

```
        System.out.println("Loading image: " + imageFile);
    }
}

// Facade
class MultimediaFacade {
    private AudioPlayer audioPlayer;
    private VideoPlayer videoPlayer;
    private ImageLoader imageLoader;

    public MultimediaFacade() {
        this.audioPlayer = new AudioPlayer();
        this.videoPlayer = new VideoPlayer();
        this.imageLoader = new ImageLoader();
    }

    void playMedia(String mediaFile) {
        if (mediaFile.endsWith(".mp3")) {
            audioPlayer.playAudio(mediaFile);
        } else if (mediaFile.endsWith(".mp4")) {
            videoPlayer.playVideo(mediaFile);
        } else if (mediaFile.endsWith(".png") || mediaFile.endsWith(".jpg")) {
            imageLoader.loadImage(mediaFile);
        } else {
            System.out.println("Unsupported media type");
        }
    }
}

// Client Code
public class FacadeExample {
    public static void main(String[] args) {
        MultimediaFacade multimediaFacade = new MultimediaFacade();
        multimediaFacade.playMedia("audio.mp3");
        multimediaFacade.playMedia("video.mp4");
        multimediaFacade.playMedia("image.jpg");
    }
}
```

Listing 2: Providing a simplified interface to a set of interfaces in a subsystem using Facade design pattern

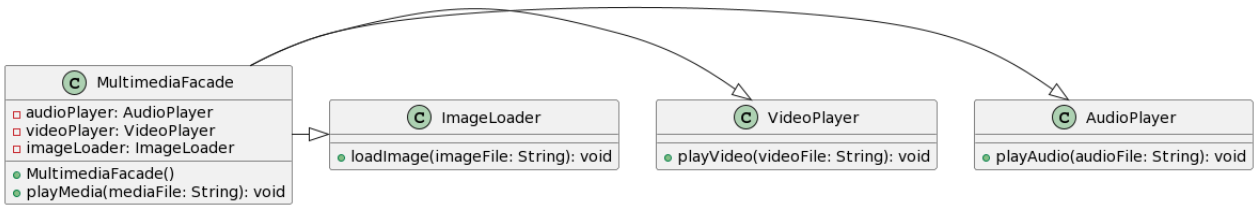

Figure 2: Class Diagrams for using Facade design pattern

Guided Code Walkthrough

In this Facade pattern example, we have a multimedia subsystem with various components. To simplify the interaction for the client, we introduce a `MultimediaFacade`.

The `MultimediaFacade` acts as a single entry point for the client, providing simplified methods for common actions like playing audio, video, and loading images. This shields the client from the complexities of the subsystem.

Observe how the client code (`MultimediaClient`) interacts with the subsystem through the facade, making the overall system more understandable and maintainable.

In Lab Task 1

Enhance the `MultimediaFacade` to handle additional multimedia formats.

Lab Assignment

Assignment 1: Real-World Scenario (Adapter Pattern)

Imagine you are working on a project that integrates with multiple payment gateways, each having a different interface for processing payments. The goal is to create a unified payment processing system in your application that can seamlessly work with any payment gateway. Apply the Adapter design pattern to adapt the existing payment gateway interfaces to a common interface within your system.

Requirements:

1. Choose at least three different payment gateways with distinct interfaces.
2. Create adapter classes for each payment gateway to adapt their interfaces to a common payment processing interface.
3. Implement a client code that demonstrates the usage of the unified payment processing system with adapters for different gateways.
4. Ensure that the client code remains agnostic to the specific payment gateways.

Use the class diagram in Figure 3 to fulfil the requirements described and demonstrate how the Adapter pattern is applied to create a unified payment processing system. Adapters are used to adapt the interfaces of different payment gateways to a common `PaymentProcessor` interface. Deliverables:

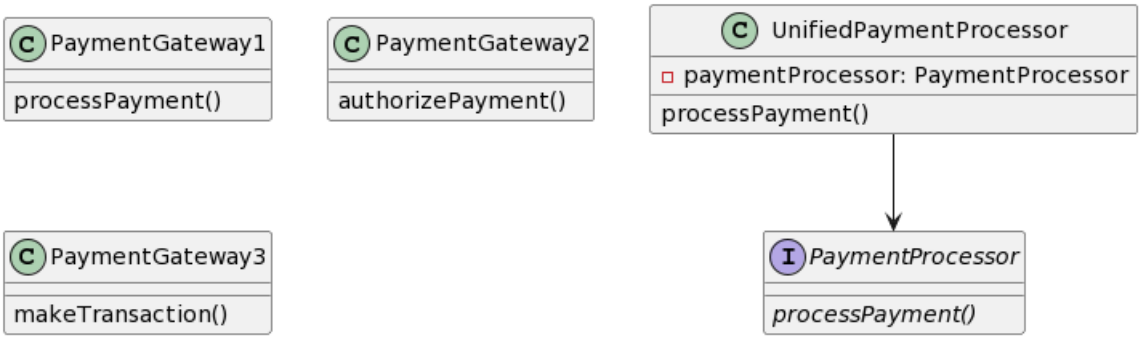

Figure 3: Use Adapters pattern to adapt the interfaces of different payment gateways to a common `PaymentProcessor` interface

1. Provide the Java code for your solution, including the adapter classes and client code.
2. Create a class diagram to illustrate the structure of your solution.

Assignment 2: Real-World Scenario (Facade Pattern)

Consider a complex subsystem in a hospital management system that involves multiple components such as patient information retrieval, appointment scheduling, and medical record management. Design a Facade to simplify the interaction with this subsystem, providing a unified and user-friendly interface for common operations in the hospital management system.

Requirements:

1. Identify at least three subsystem components related to hospital management.
2. Create a Facade class that encapsulates the complex interactions of the identified subsystem components.
3. Implement a client code that demonstrates the usage of the unified payment processing system with adapters for different gateways.

4. Design methods in the Facade for common operations, such as retrieving patient information, scheduling appointments, and managing medical records.

5. Implement a client code that utilizes the Facade to perform operations on the hospital management subsystem.

Use the class diagram in Figure 4 to fulfil the requirements described. illustrates the Facade pattern applied to simplify interactions with a hospital management subsystem. The `HospitalManagementFacade` provides a unified interface to retrieve patient information, schedule appointments, and manage medical records, encapsulating the complexities of the subsystem. Deliverables:

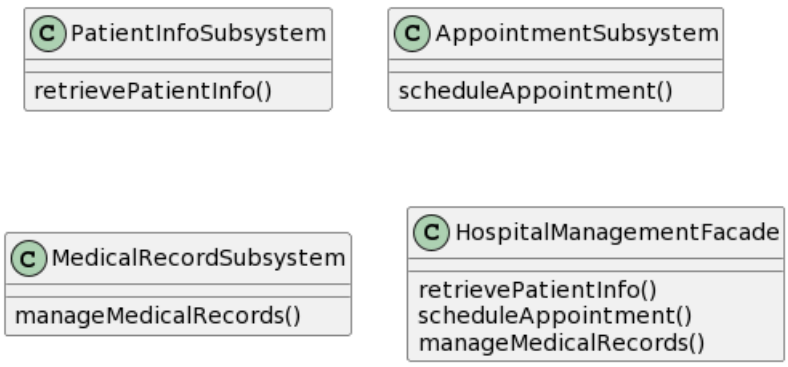

Figure 4: Use Facade pattern to simplify interactions with a hospital management subsystem

1. Provide the Java code for your solution, including the adapter classes and client code.

2. Create a class diagram to illustrate the structure of your solution.