GoF (Gang of Four) Design Patterns

for

Soccer Live

Version: 1.0

Prepared by: Usman Awan | 22F-3378

Organization: FAST NUCES CFD Date: 2 nd October, 2

Contents

[1. Creational Pattern: Singleton Pattern 3](#_Toc184657757)

[Purpose 3](#_Toc184657758)

[Steps For Making this Design Pattern: 3](#_Toc184657759)

[Java Code 3](#_Toc184657760)

[VS Code Interface: 5](#_Toc184657761)

[Output: 5](#_Toc184657762)

[2. Structural Pattern: Adapter Pattern 6](#_Toc184657763)

[Purpose 6](#_Toc184657764)

[Steps For Making this Design Pattern: 6](#_Toc184657765)

[Java Code 6](#_Toc184657766)

[VS CODE Interface 8](#_Toc184657767)

[Output: 8](#_Toc184657768)

[3. Behavioral Pattern: Observer Pattern 9](#_Toc184657769)

[Purpose 9](#_Toc184657770)

[Steps For Making this Design Pattern: 9](#_Toc184657771)

[Java Code 9](#_Toc184657772)

[VS Code Interface: 12](#_Toc184657773)

[Output: 12](#_Toc184657774)

[Table of Three GoF Design Patterns 12](#_Toc184657775)

A diagram of different types of design

Description automatically generated

# 1. Creational Pattern: Singleton Pattern

## Purpose

The Singleton pattern is used to ensure that the **Streaming Engine** is a single instance throughout the application. This guarantees consistent behavior and avoids issues like resource conflicts when multiple streams are initiated. The StreamingEngine class will implement this pattern to provide a global point of access to the instance.

## Steps For Making this Design Pattern:

1. Create a StreamingEngine class.
2. Implement the Singleton pattern to ensure only one instance exists.
3. Provide a global point of access to the instance.

## Java Code

// SingletonPattern.java

public class SingletonPattern {

public static void main(String[] args) {

// Fetching the single instance of StreamingEngine

StreamingEngine engine = StreamingEngine.getInstance();

engine.startStream("Team A vs Team B");

}

}

class StreamingEngine {

// Static variable to hold the single instance

private static StreamingEngine instance;

// Private constructor to prevent external instantiation

private StreamingEngine() {

System.out.println("Streaming Engine Initialized");

}

// Public static method to provide access to the single instance

public static synchronized StreamingEngine getInstance() {

if (instance == null) {

instance = new StreamingEngine();

}

return instance;

}

// Example method to simulate streaming

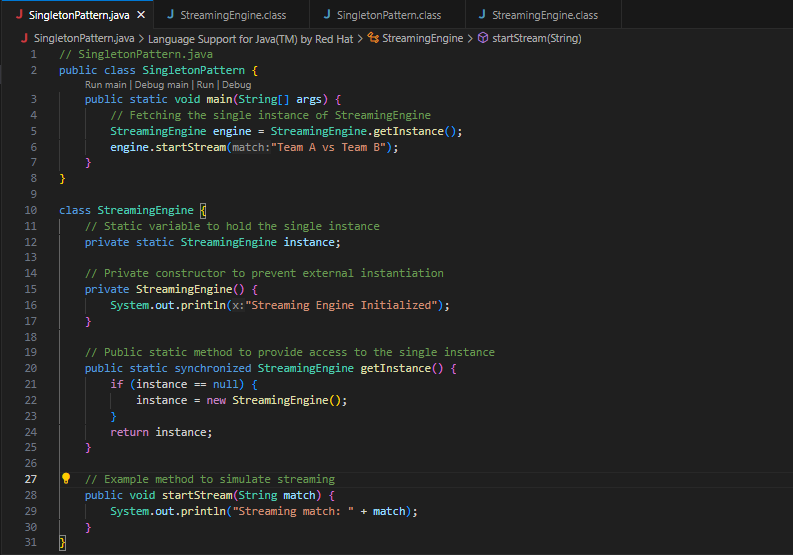
public void startStream(String match) {

System.out.println("Streaming match: " + match);

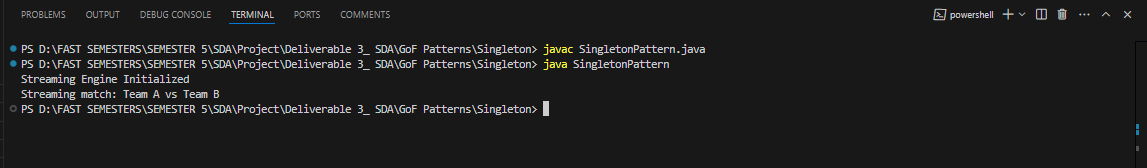
}

}

## VS Code Interface:



## Output:



# 2. Structural Pattern: Adapter Pattern

## Purpose

The Adapter pattern is used to enable compatibility with multiple video codecs and streaming formats. The MediaAdapter class acts as a bridge between the Streaming Engine and specific video players (e.g., for MP4 and MKV formats), ensuring smooth playback across different formats.

## Steps For Making this Design Pattern:

1. Define a MediaPlayer interface for playback functionality.
2. Implement a VideoPlayer class for a specific codec.
3. Create an Adapter class to translate requests from the StreamingEngine to VideoPlayer.

## Java Code

// AdapterPattern.java

public class AdapterPattern {

    public static void main(String[] args) {

        MediaPlayer adapter = new MediaAdapter();

        adapter.play("MP4", "match.mp4");

        adapter.play("MKV", "highlight.mkv");

        adapter.play("AVI", "unsupported.avi");

    }

}

interface MediaPlayer {

    void play(String videoFormat, String videoFile);

}

class VideoPlayer {

    public void playMP4(String file) {

        System.out.println("Playing MP4 file: " + file);

    }

    public void playMKV(String file) {

        System.out.println("Playing MKV file: " + file);

    }

}

class MediaAdapter implements MediaPlayer {

    private VideoPlayer videoPlayer;

    public MediaAdapter() {

        this.videoPlayer = new VideoPlayer();

    }

    @Override

    public void play(String videoFormat, String videoFile) {

        if (videoFormat.equalsIgnoreCase("MP4")) {

            videoPlayer.playMP4(videoFile);

        } else if (videoFormat.equalsIgnoreCase("MKV")) {

            videoPlayer.playMKV(videoFile);

        } else {

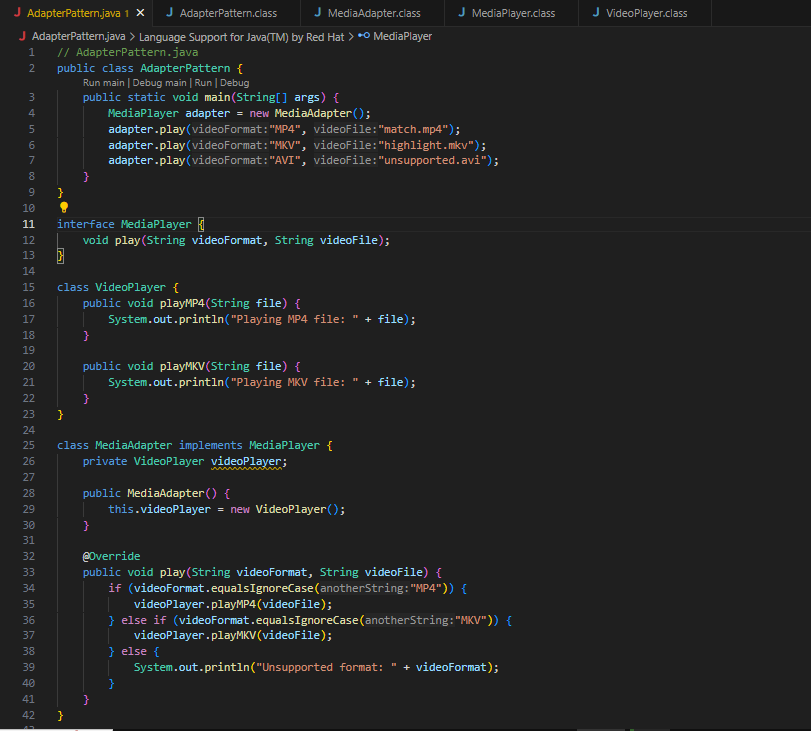
            System.out.println("Unsupported format: " + videoFormat);

        }

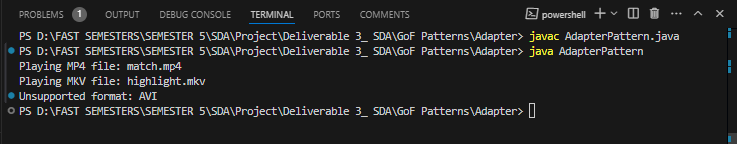
    }

}

## VS CODE Interface



## Output:



# 3. Behavioral Pattern: Observer Pattern

## Purpose

The Observer pattern is used to notify users in real-time about **match updates** and **breaking news**. The MatchUpdateNotifier class acts as the **Subject** that maintains a list of observers, while each UserDevice represents a concrete **Observer**.

## Steps For Making this Design Pattern:

1. Create a Subject class that maintains a list of observers.
2. Define an Observer interface.
3. Implement concrete observer classes (e.g., UserDevice).
4. Notify observers when updates occur.

## Java Code

// ObserverPattern.java

import java.util.ArrayList;

import java.util.List;

public class ObserverPattern {

public static void main(String[] args) {

MatchUpdateNotifier notifier = new MatchUpdateNotifier();

// Create observers

UserDevice device1 = new UserDevice("Phone");

UserDevice device2 = new UserDevice("Tablet");

// Register observers

notifier.addObserver(device1);

notifier.addObserver(device2);

// Notify observers

notifier.notifyObservers("Goal scored by Team A!");

}

}

interface Observer {

void update(String message);

}

class UserDevice implements Observer {

private String deviceName;

public UserDevice(String deviceName) {

this.deviceName = deviceName;

}

@Override

public void update(String message) {

System.out.println(deviceName + " received update: " + message);

}

}

class MatchUpdateNotifier {

private List<Observer> observers = new ArrayList<>();

// Add observer

public void addObserver(Observer observer) {

observers.add(observer);

}

// Remove observer

public void removeObserver(Observer observer) {

observers.remove(observer);

}

// Notify all observers

public void notifyObservers(String message) {

for (Observer observer : observers) {

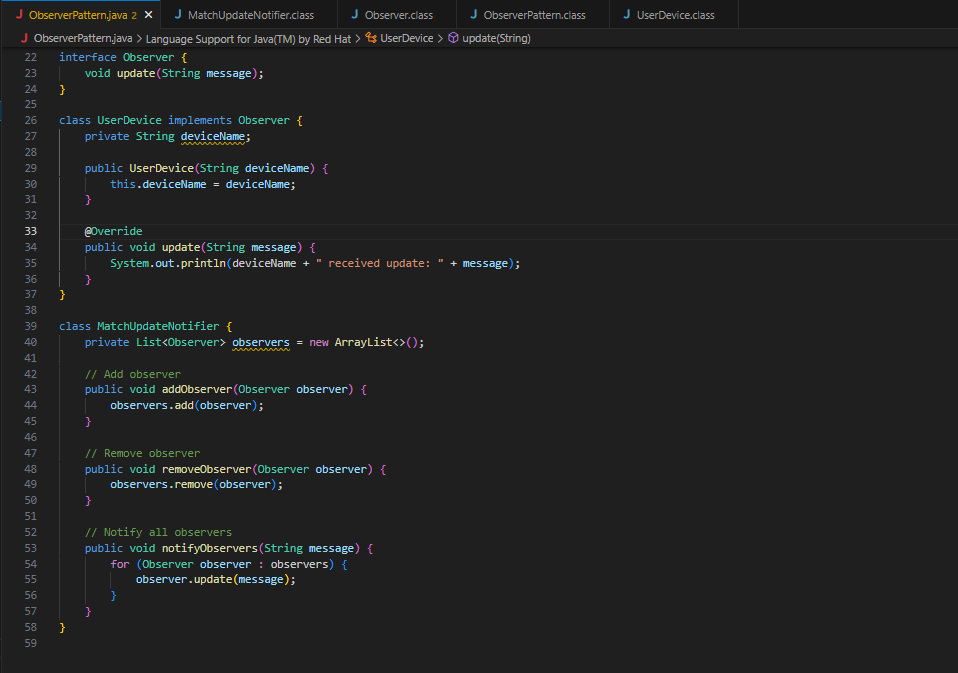
observer.update(message);

}

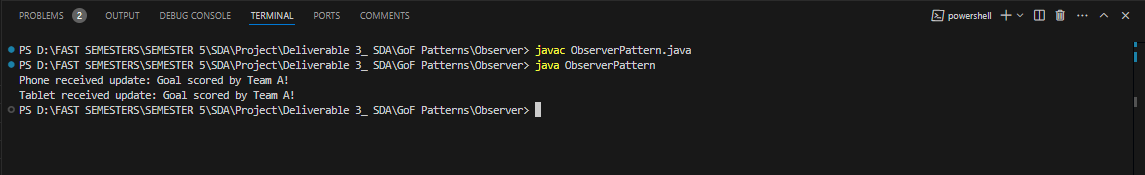
}

}

## VS Code Interface:



## Output:



# Table of Three GoF Design Patterns

|  |  |  |
| --- | --- | --- |
| Pattern | Purpose | Implementation |
| Singleton | Ensure a single instance of the **Streaming Engine**. | Implemented with private constructor and a static method to return the single instance. |
| Adapter | Support multiple video codecs and formats for streaming. | An adapter bridges the gap between the **Streaming Engine** and video playback using the MediaAdapter class. |
| Observer | Notify users of match updates and breaking news in real-time. | The MatchUpdateNotifier class manages observers, and updates are pushed to registered UserDevice objects. |