Adapter Design Pattern

The Adapter Design Pattern is a structural pattern that allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface that a client expects.

Customer Requirement:

Imagine you are working on a project where you need to integrate a new third-party library that provides advanced audio functionality. However, the existing codebase uses a different audio interface. You need a way to make the new library's interface compatible with the existing one without modifying either.

Solution: Adapter Pattern

The Adapter Pattern helps in this scenario by creating an adapter that sits between the existing codebase and the new library. The adapter translates calls from the existing interface to the new library's interface, allowing them to work together seamlessly.

Components of the Adapter Pattern:

1. Target Interface: This is the interface that the client expects. In our case, it's the existing audio interface in the codebase.
2. Adaptee: This is the new interface that is incompatible with the existing system. In our case, it's the third-party library's audio interface.
3. Adapter: The adapter implements the target interface and translates calls from the target to the adaptee. It adapts the interface of the adaptee to the target interface.

Example: Audio Player

Let's say the existing codebase uses the `AudioPlayer` interface to play audio, but the new third-party library uses `AdvancedMediaPlayer`.

1. Target Interface (`AudioPlayer`):

```java
public interface AudioPlayer {
    void play(String audioType, String fileName);
}
```

2. Adaptee (`AdvancedMediaPlayer`):

```java
public interface AdvancedMediaPlayer {
    void playMp3(String fileName);
    void playMp4(String fileName);
}
```

3. Adapter (`AudioPlayerAdapter`):

```java
public class AudioPlayerAdapter implements AudioPlayer {
    AdvancedMediaPlayer advancedMusicPlayer;

    public AudioPlayerAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("mp3")) {
            advancedMusicPlayer = new Mp3Player();
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer = new Mp4Player();
        }
    }
```

```
    @Override
    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp3")) {
            advancedMusicPlayer.playMp3(fileName);
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}
```

4. Concrete Classes Implementing `AdvancedMediaPlayer`:

```java
public class Mp3Player implements AdvancedMediaPlayer {
    @Override
    public void playMp3(String fileName) {
        System.out.println("Playing mp3 file. Name: " + fileName);
    }

    @Override
    public void playMp4(String fileName) {
        // Do nothing
    }
}

public class Mp4Player implements AdvancedMediaPlayer {
```

```
    @Override
    public void playMp3(String fileName) {
        // Do nothing
    }

    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: " + fileName);
    }
}
```

5. Client Code:

```java
public class AdapterPatternDemo {
    public static void main(String[] args) {
        AudioPlayerAdapter audioPlayer = new AudioPlayerAdapter("mp3");
        audioPlayer.play("mp3", "song.mp3");

        audioPlayer = new AudioPlayerAdapter("mp4");
        audioPlayer.play("mp4", "video.mp4");
    }
}
```

How It Works:

- Client: The client uses the `AudioPlayer` interface to play audio files.

- Adapter: The `AudioPlayerAdapter` converts the `AudioPlayer` interface to the `AdvancedMediaPlayer` interface.

- Adaptee: The `Mp3Player` and `Mp4Player` classes implement the `AdvancedMediaPlayer` interface and play the respective audio formats.

Benefits of the Adapter Pattern:

1. Compatibility: The Adapter Pattern allows otherwise incompatible interfaces to work together.

2. Reusability: It enables the reuse of existing classes by adapting their interfaces to work with others.

3. Flexibility: You can introduce new adapters without changing the existing codebase.

When to Use the Adapter Pattern:

- When you want to use an existing class, but its interface is not compatible with the code you need to work with.

- When you want to create a reusable class that can work with unrelated classes or interfaces.