Final Report for

CSF304(Design Patterns)

Bachelor of Science in Computer Science

AI Development and Data Science

Year III, Semester II

Online Banking System

Submitted by

Pema Yangzom(12210073)

Kinley Namgay(12210059)

Kinley Wangchuk(12210060)

Gyalpozhing College of Information Technology

Guided By: Mrs Tawmo

Table of Contents

1. Project Title

Music Player System

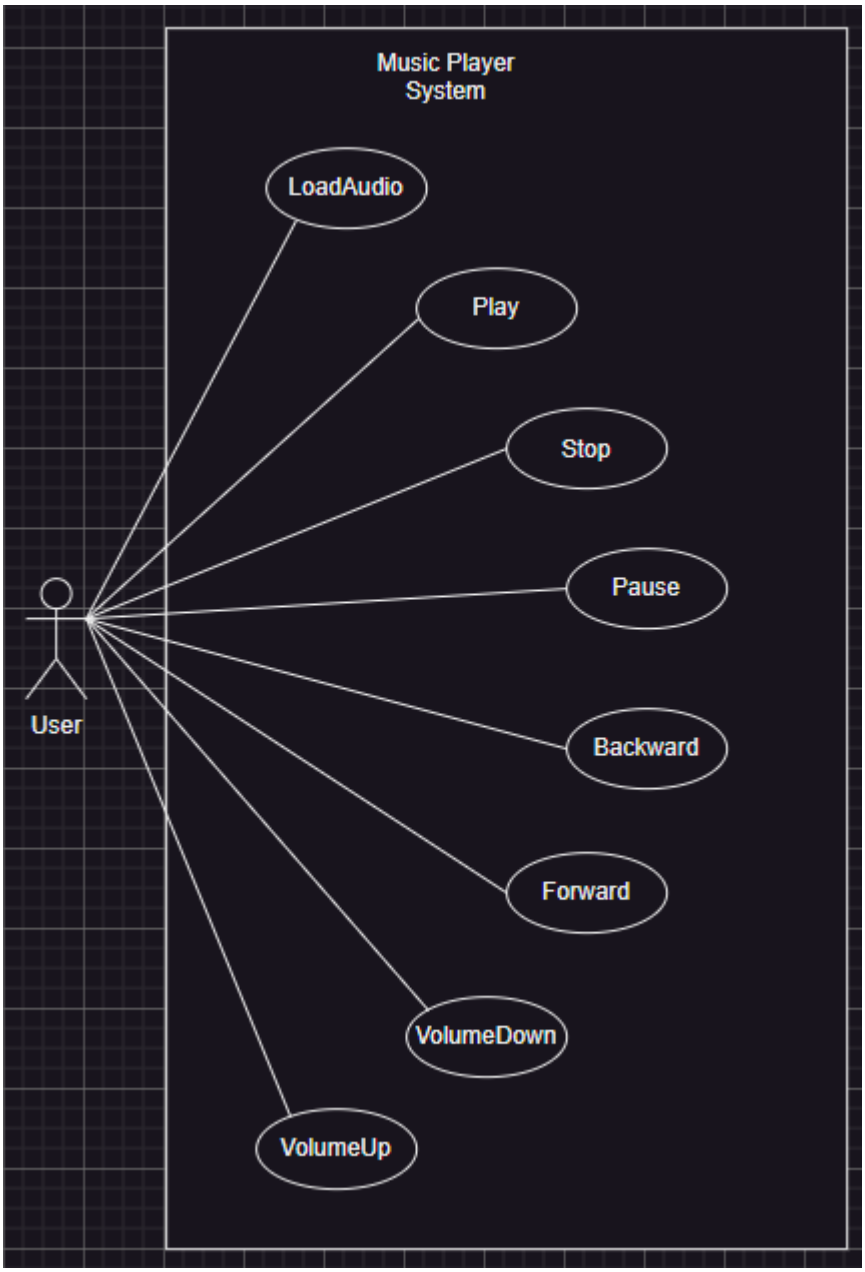
2. Brief Project Description

This project aims to develop a comprehensive and user-friendly music player system using Java, leveraging robust frameworks and design patterns to ensure scalability and reliability. The system will provide users with essential features such as play, pause, resume, stop, seek forward, seek backward, volume control, and progress tracking. Additionally, the system will support loading and playing audio files in various formats and provide real-time updates on the current state of the music player. The implementation will emphasize a modular design to facilitate future enhancements and maintenance. By integrating a developed framework and using JavaFX to demonstrate the implementations, the project employs various design patterns:

- **Mediator** for coordinating interactions between UI components and the music player,
- **Chain of Responsibility** for handling user commands like play, pause, and stop,
- **State** for managing the different states of the music player (playing, paused, stopped),
- **Command** for executing user actions such as volume control and seeking,
- **Observer** for updating the UI with the current progress and state of the music player,
- **Factory Method** for creating instances of the music player with specific configurations.

This approach not only enhances the user experience through a flexible and responsive music player but also sets a high standard for future multimedia software solutions.

3. Use Case (Functionality)



3.1 LoadAudio

- You can load the music you want to play

3.2 Play

- You can play the audio you loaded

3.3 Stop

- If you want to terminate the playback then can stop the music

3.4 Pause

- When you want to temporarily suspend playback then use Pause

3.5 Resume

- When you want to undo the pause then you can resume

3.6 SeekForward

- When you want to skip sudden part of the music use SeekFoward

3.7 SeekBackward

- When you want to rewind the music use SeekBackward

3.8 VolumeUp

- When you want to turn the music louder use Volume Up

3.9 VolumeDown

- When you want to turn the volume low use Volume Down

4. Source Code

GitLab link: https://github.com/davaibylat128/JavaMusicPlayer.git

5. User Interface: (How to use it)

The system's user interface is built using Java Swing, providing an interactive and responsive experience. Upon launching the application, users will encounter the following functionalities:

Add Audio: Users can add the audio/ music file they want to play.

Play: After adding the file the user can play the music.

Pause: If the user wants to temporarily stop the music than they can use Pause.

Stop: Users can even stop the music for permanently.

Resume: If the users wants to listen to the music again then user can resume the song they paused.

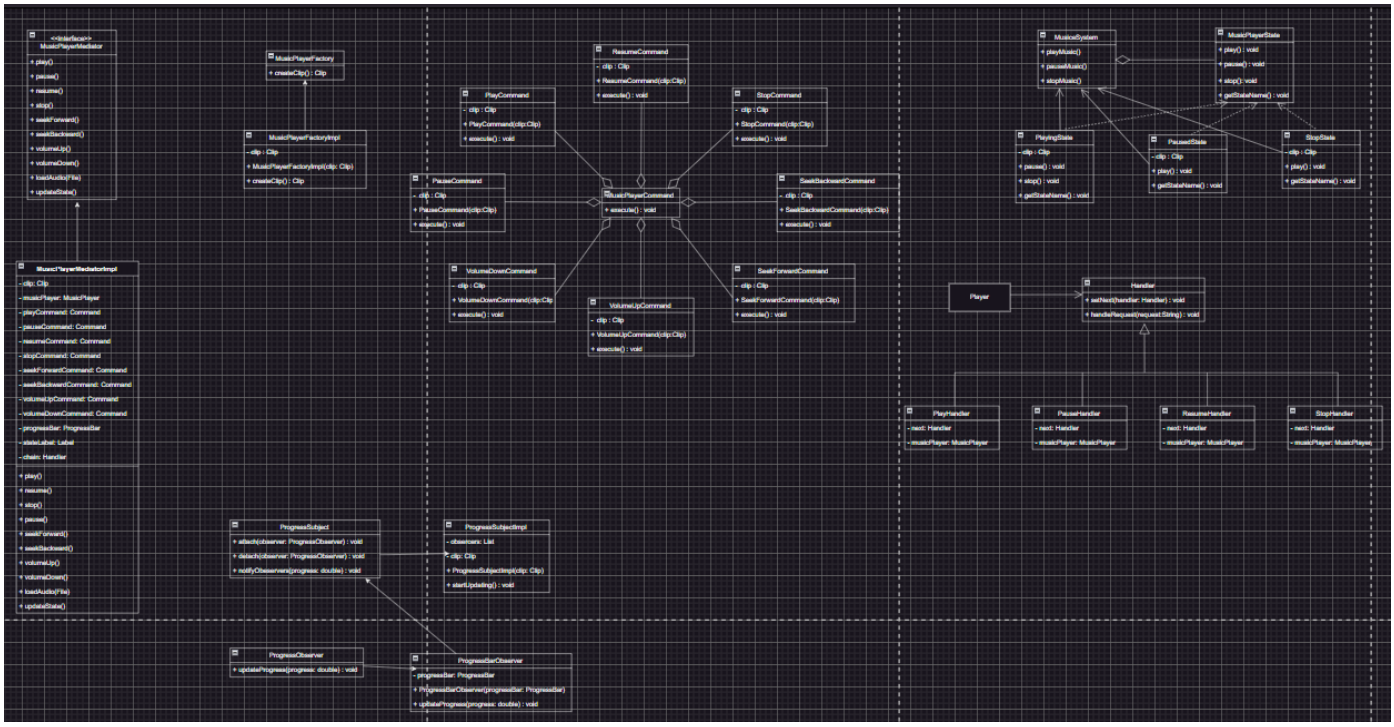
Seek Forward: If the users want to skip sudden part of the song move to next part then they can use Seek Forward.

Seek Backward: If the users want to rewind or want to listen to a sudden part again than they can use Seek Backward

Volume Up: Users can turn the music loud if they want to by using Volume Up.

Volume Down: Users can also turn down the volume by using Volume Down.

6. Class Diagrams



Note: The above diagram is not clear, so click on the link below and refer to "uml.drawio" file for a clear class diagram.

https://github.com/davaibylat128/JavaMusicPlayer.git

7. Justifications for all the design patterns used

7.1 Mediator

The Mediator Pattern is ideal for scenarios where you need to manage complex interactions between multiple objects or components. In the music player project, the mediator coordinates interactions between UI components (such as play, pause, stop buttons, and progress bar) and the music player logic. By using this pattern, the direct dependencies between UI components are minimized, promoting loose coupling and making the code more maintainable. The mediator handles all the communication, ensuring that components can work independently and changes in one component do not directly affect others.

7.2 Chain of Responsibility

The Chain of Responsibility Pattern is perfect for situations where multiple handlers can process a request in sequence. In the music player project, this pattern is used for handling user commands like play, pause, stop, and other actions. Each command (like PlayHandler, PauseHandler, etc.) is encapsulated in its own handler, promoting separation of concerns and making the command handling process more flexible and maintainable. If one handler cannot process the request, it passes it to the next handler in the chain, ensuring that the appropriate action is taken by the correct handler.

7.3 Command

The Command Pattern is effective for encapsulating a request as an object, thereby allowing parameterization of clients with queues, requests, and operations. In the music player project, this pattern is used for executing user actions such as volume control, seeking, and other commands. Each command (like PlayCommand, PauseCommand, VolumeUpCommand) is implemented as a separate class, which encapsulates the action to be performed. This promotes decoupling between the sender and receiver of the command and makes it easier to manage and extend user actions.

7.4 State

The State Pattern is useful for managing an object's behavior when its state changes. In the music player project, this pattern is used to manage the different states of the music player (playing, paused, stopped). Each state is represented by a state class (like PlayingState, PausedState, StoppedState), and the music player's behavior changes depending on its current state. This pattern allows for cleaner and more organized code, as state-specific behavior is encapsulated within state classes, making it easier to add new states or modify existing ones.

7.5 Factory Method

The Factory Method Pattern is useful for creating objects without specifying the exact class of object that will be created. In the music player project, this pattern is used for creating instances of the music player with specific configurations. The MusicPlayerFactoryImpl class implements the MusicPlayerFactory interface, providing a method to create Clip objects. This promotes flexibility and scalability, as new types of clips can be added or existing ones modified without changing the client code that uses the factory.

The Clip class in Java, part of the javax.sound.sampled package, is used for playing, stopping, and controlling audio clips. It represents a sound clip that can be loaded with audio data, and it provides methods to control playback, such as play, stop, pause, resume, and loop.

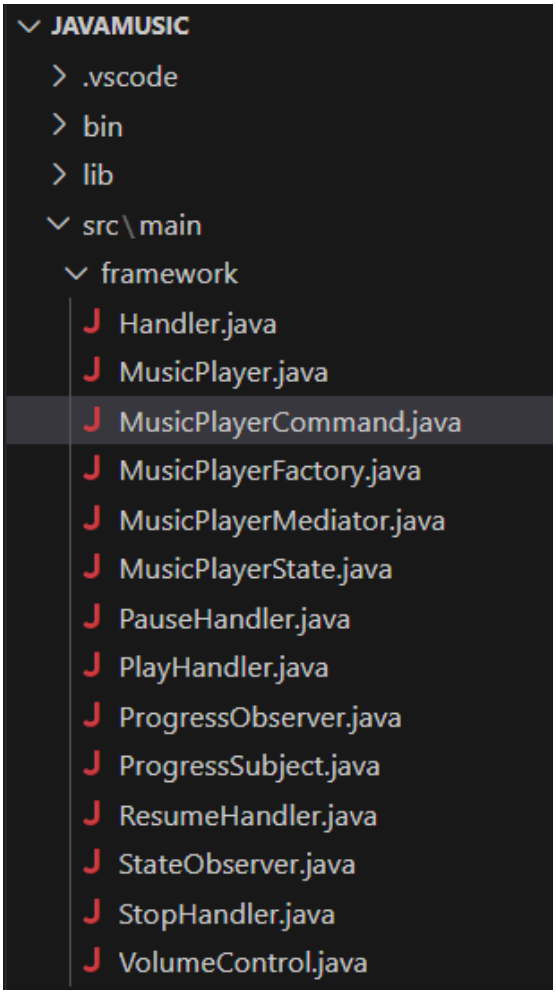
7.6 Observer

The Observer Pattern is suitable for scenarios where one object needs to notify multiple dependent objects about changes in its state. In the music player project, this pattern is used to update the UI with the current progress and state of the music player. The progress of the music player is observed by components like the

progress bar and state label. When the state or progress of the music player changes, the observers are notified and can update accordingly. This pattern helps in keeping the UI synchronized with the music player's state without tight coupling.

8. Non-changeable and Changeable state

8.1 Non-changeable State

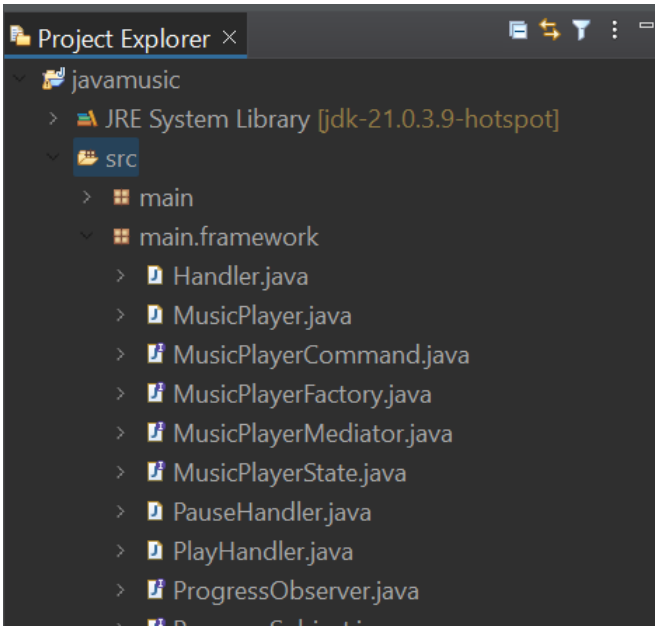



```java
package main.framework;


public interface MusicPlayerCommand {
    void execute();
}
```

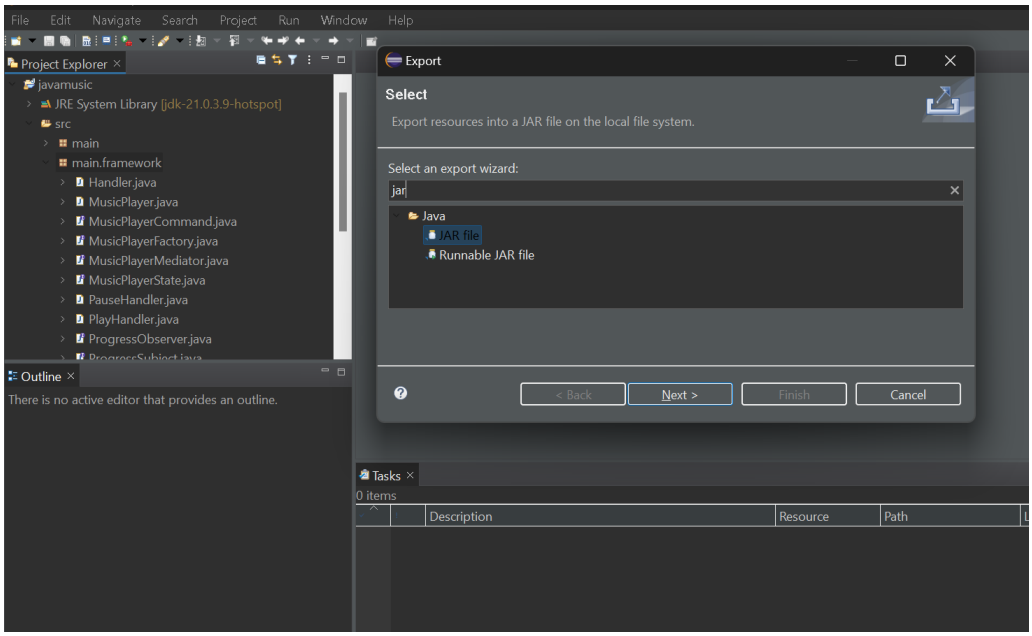
The non-changeable state is implemented inside the framework. Having a non-changeable (immutable) state within a framework is crucial for ensuring that the framework is extendable, stable, and reliable. It protects against unintended modifications, prevents errors, and maintains consistent behavior across different applications and developers. Inside the online banking framework, we especially have a pattern interface defined inside the framework which is unchangeable. This framework is then exported to the jar extension and imported inside in our online banking application to implement it according to our expectation.

8.1.1 Framework conversion to Jar extension

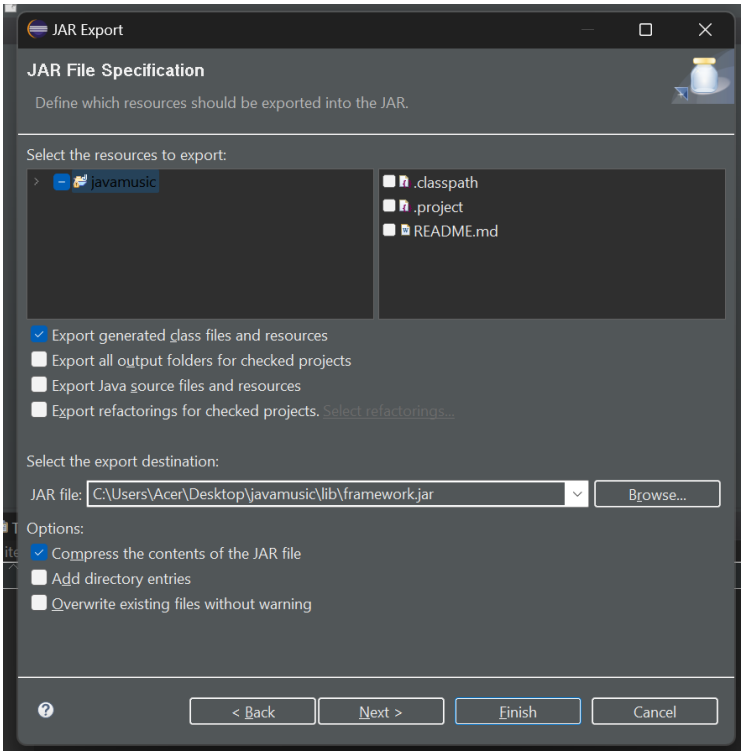
The defined framework is converted to jar extension using eclipse IDE. Converting Java frameworks to JAR extensions provides numerous benefits in terms of convenience, performance, security, and portability, making it the preferred method for packaging and distributing Java applications and libraries. The image is the framework directory. Each directory contains a non-changeable state class.



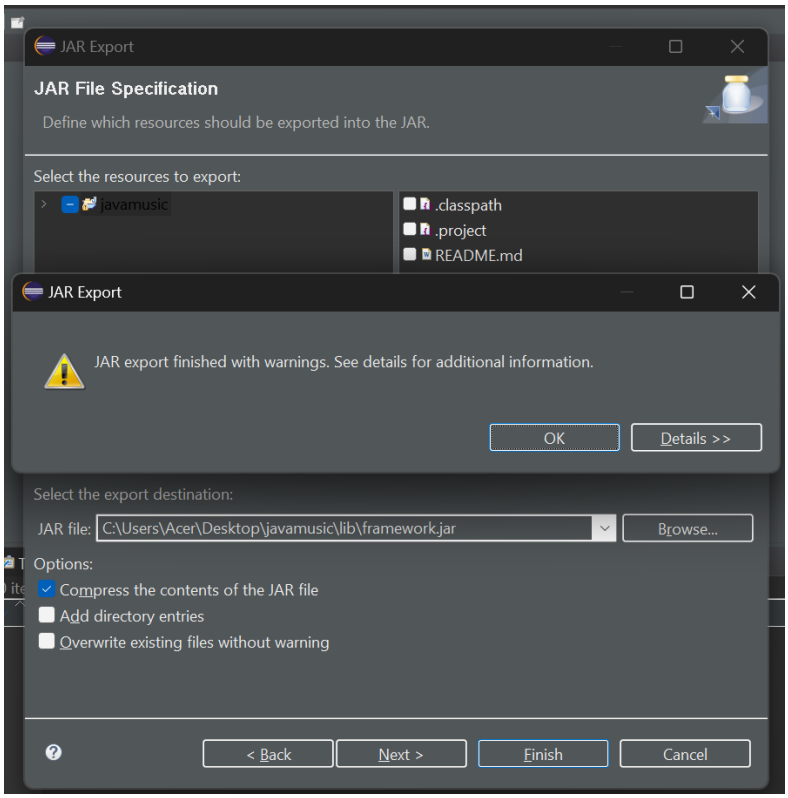
Click on export option and choose “JAR file” as extension to be exported.



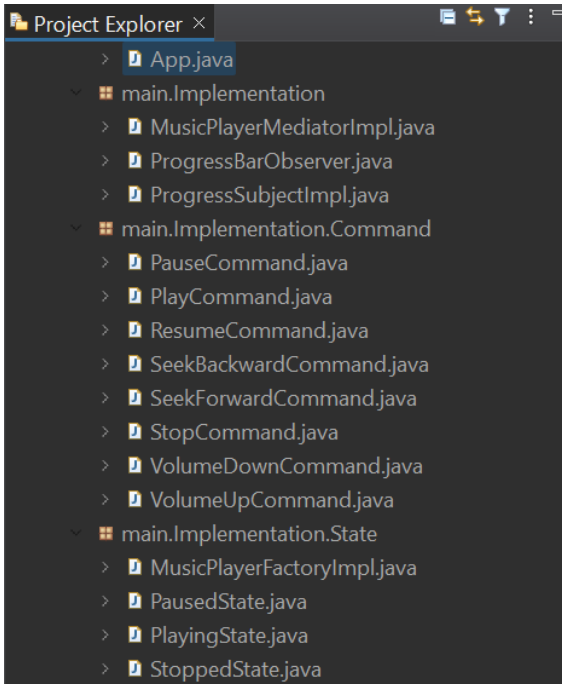
Select “.classpath” and “.project” and select the directory where jar extension file will be saved.

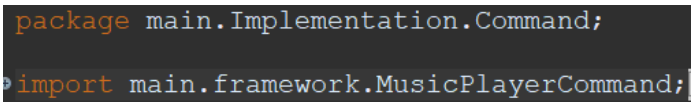


The framework is successfully converted to jar extension.



8.2 Changeable state





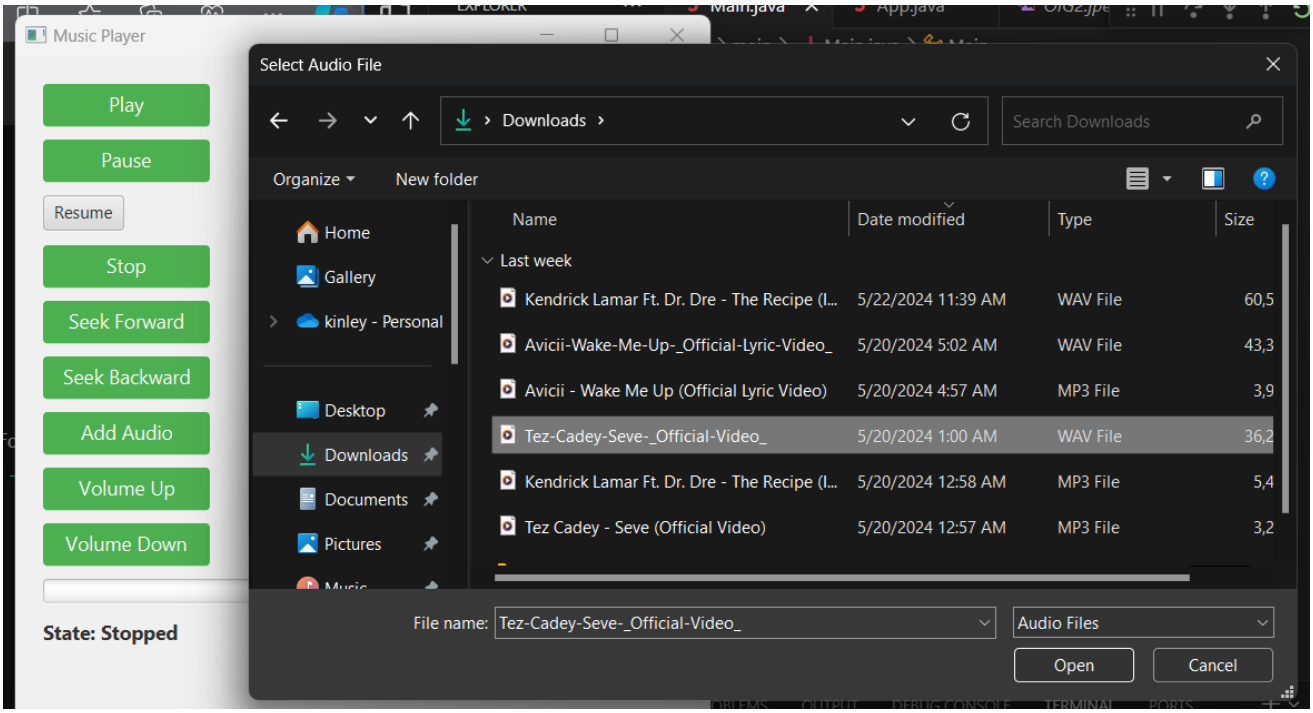
```
package main.Implementation.Command;

import main.framework.MusicPlayerCommand;
```

```
package main.Implementation.Command;

import main.framework.MusicPlayerCommand;

public class SeekBackwardCommand implements MusicPlayerCommand {
    private Clip clip;

    public SeekBackwardCommand(Clip clip) {
        this.clip = clip;
    }

    @Override
    public void execute() {
        if (clip != null) {
            long newPosition = clip.getMicrosecondPosition() - 5000000; // Seek 5 seconds backward
            if (newPosition >= 0) {
                clip.setMicrosecondPosition(newPosition);
            } else {
                clip.setMicrosecondPosition(0); // Seek to the beginning if attempting to seek before the start
            }
        }
    }
}
```

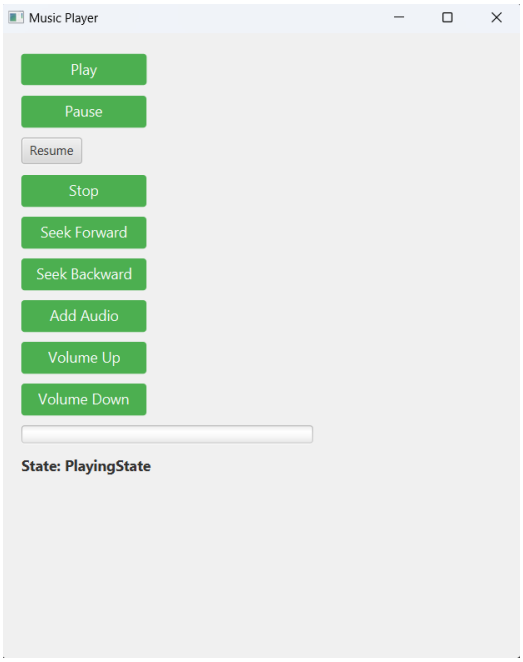
The framework is imported into our application and customized to define methods in the framework in our own way. In the above example, the MusicPlayerCommand interface is defined with the execute() method inside the framework. The SeekBackwardCommand subclass instantiated from MusicPlayerCommand implements the execute() method in its own way. This is a changeable state. The execute() method implementation changes if the developer wishes to. The changeable state is implemented in the real implementation of the application.

9. Demonstration

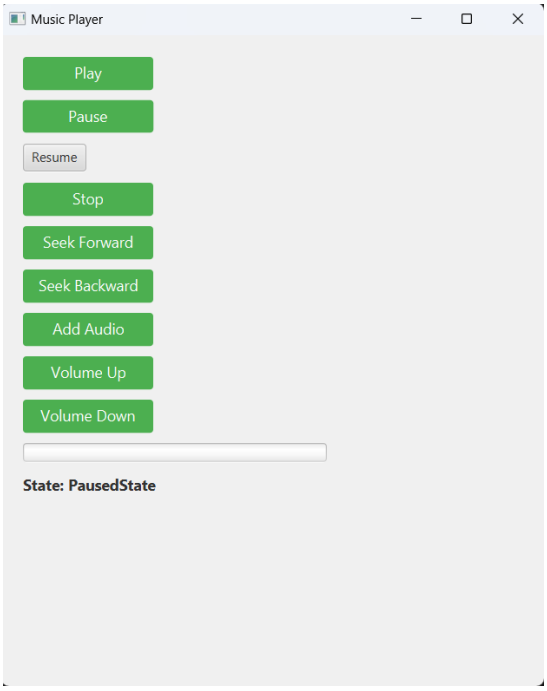
a) This is our UI and you can add the music you want to listen.



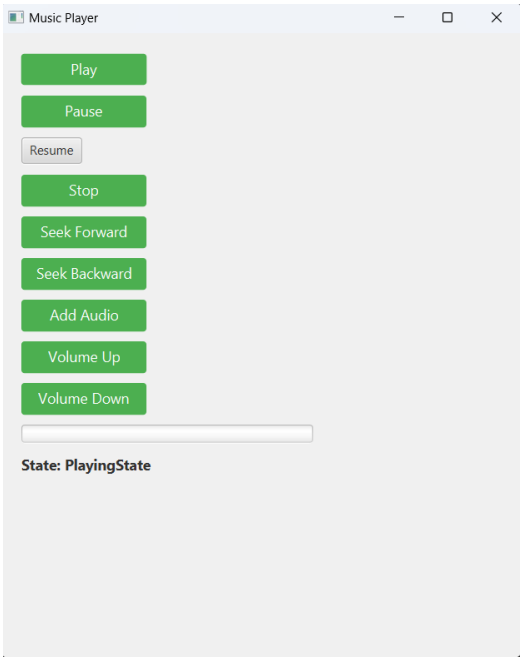
b) After you add the music you can play the music and PlayingState will be shown.



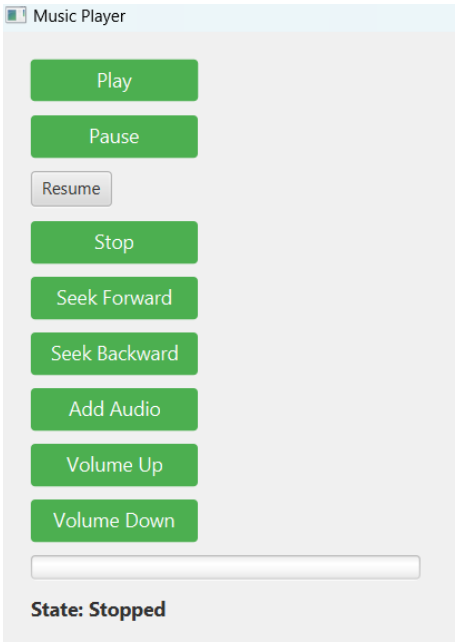
c) You can then pause your music if you feel like not listening but you don't want to completely stop the music.



d) If you want to resume what you have paused then you can use Resume and listen to music again.



e) If you want to stop the music and want to listen to another one then use Stop



10. Challenges

Building frameworks and developing applications for a music player system poses distinct challenges, such as striking a balance between crafting reusable, scalable components and addressing specific application requirements. Tensions can arise between abstract design principles and the pragmatic considerations of implementation. The endeavor necessitates a substantial upfront investment in time and resources, which can impede the rapid progress of the application. Ensuring smooth integration between framework and application components, while upholding stringent code quality standards to forestall issues impacting functionality and user satisfaction, demands meticulous planning and ongoing communication across teams. Additionally, iterative testing is imperative to synchronize framework and application objectives effectively.

Conclusion

The development of a music player system using Java and various design patterns showcases the capability to build a robust, scalable, and user-friendly application for multimedia consumption. By establishing a solid framework first, the project ensures a stable foundation for implementing specific music player features. Through the application of these design patterns, the system efficiently manages resources, offers flexibility for adding new functionalities, and promotes maintainability. This project sets a benchmark for future multimedia software solutions, emphasizing the importance of integrating strong architectural principles with practical implementation strategies. Despite the inherent challenges, the successful execution of this project underscores the potential for creating sophisticated and reliable music player systems.