## Design Rationale

### Introduction

The Improved Media Player system is made using several Object-Oriented Design Patterns to create a flexible and easy-to-maintain multimedia program. It uses the Adapter, Bridge, Decorator, Composite, and Proxy patterns to make the system more organized, expandable, and efficient.

1. **Adapter Pattern**

**Classes Involved**:

MediaSource, LocalFilePlayer, HLSStreamPlayer, RemoteAPIPlayer

The Adapter Pattern is used so that the system can handle different kinds of media input such as local files, live streams, or media from online sources. The MediaSource interface gives one common way for these different types to load a file. Each class implements the load(String fileName) method differently. For example, LocalFilePlayer loads files from storage, HLSStreamPlayer connects to live streams, and RemoteAPIPlayer loads media from a website or API. This pattern makes it easy to add new media types later without changing the old code. It follows the rule of being open for adding features but closed for changing existing ones.

1. **Bridge Pattern**

**Classes Involved**:

Renderer, HardwareRenderer, SoftwareRenderer, BaseMediaPlayer

The Bridge Pattern separates the player’s features from how it displays or renders the media. The Renderer interface defines how rendering should be done, and it is implemented by HardwareRenderer and SoftwareRenderer. The BaseMediaPlayer connects both parts by playing the media while using the chosen renderer. This pattern helps developers create new renderer types without editing the player’s main code. It gives more freedom and makes the program easier to update or expand.

1. **Decorator Pattern**

**Classes Involved:**

Player, BaseMediaPlayer, PlayerDecorator, SubtitleDecorator, EqualizerDecorator, WatermarkDecorator

The Decorator Pattern adds extra features to the media player without changing its original class. The PlayerDecorator acts like a wrapper around a Player object. Each decorator class adds a new function by extending the play() method. For example, SubtitleDecorator adds subtitles, EqualizerDecorator adjusts sound, and WatermarkDecorator adds a watermark to the video. This pattern allows users to turn features on or off anytime. It also keeps each class focused on one function only, making the code simple and clean.

1. **Composite Pattern**

**Classes Involved**:

PlaylistComponent, Song, Playlist

The Composite Pattern helps organize songs and playlists in a tree-like structure. The PlaylistComponent interface gives a common way to use both songs and playlists with the showDetails() method. The Song class acts as a leaf, and the Playlist class acts as a container that can hold many songs or other playlists. For example, a playlist named “My Favorites” can include individual songs and another playlist like “Mixed Hits”. This pattern makes it easy to handle both single and grouped media items using one structure.

1. **Proxy Pattern**

**Classes Involved**:

RemoteMedia, RealRemoteMedia, RemoteMediaProxy

The Proxy Pattern is used for handling online or remote media. It improves speed by using caching. The RemoteMediaProxy acts as a middleman between the user and the RealRemoteMedia class. When a file is played for the first time, it is stored in cache. The next time the same file is played, it is loaded from the cache instead of being downloaded again. This saves time and reduces data usage, making streaming faster and more efficient.

1. **Integration of Patterns**

Each design pattern works together to make the whole system function smoothly.

* The Adapter Pattern loads media from different sources.
* The Bridge Pattern handles how the media is displayed.
* The Decorator Pattern adds extra player features.
* The Composite Pattern manages songs and playlists.
* The Proxy Pattern helps stream media faster.

By combining these patterns, the Improved Media Player becomes more flexible, organized, and easier to maintain.

1. **Design Principles Used**

* **Abstraction**

Interfaces like MediaSource, Renderer, Player, and PlaylistComponent hide the inner code and show only the main functions.

* **Encapsulation**

Each class hides its internal details and only shows needed methods.

* **Polymorphism**

Achieved through method overriding in adapters, renderers, and decorators.

* **Composition** **over** **Inheritance**

Many parts use object composition instead of extending classes.

* **Loose** **Coupling**

Each part can be changed or tested without breaking the others.

### Conclusion

The Improved Media Player project shows how different design patterns can work together in one system. By using the Adapter, Bridge, Decorator, Composite, and Proxy patterns, the program becomes more flexible, well-structured, and efficient. Each pattern has its own purpose, and when combined, they make the media player easier to improve and more powerful for future updates.