# JEFFREY INDONG IT3A

# Design Rationale: Structural Patterns in the Modular Media Streaming Suite

The Modular Media Streaming Suite refactors a monolithic legacy media player into a flexible, extensible system using structural design patterns to address its limitations—code duplication, rigid structure, and lack of plugin support—while meeting requirements for multiple media sources, on-the-fly feature plugins, composite playlists, runtime renderer switching, and stream caching. Below, we outline the structural patterns applied in the provided Lab2.java implementation, referencing specific classes (Source, MediaApp, Renderer, MediaPlayerBridge, Player, PlayerDecorator, MediaItem, RemoteMedia, RemoteMediaProxy) and explaining their use and rationale, ensuring modularity, extensibility, and maintainability.

## Adapter Pattern: Unifying Media Sources

The legacy codebase duplicated logic for handling local files, HLS streams, and remote APIs. The Adapter pattern was applied through the Source interface, with concrete implementations (LocalSource, StreamSource, ApiSource) adapting source-specific behaviors into a unified connect method. The MediaApp class uses a Source instance to abstract media access, as seen in the main method’s dynamic source selection. This eliminates duplication, allowing MediaApp to interact with any source uniformly, and supports adding new sources (e.g., cloud storage) without modifying core logic. The pattern was chosen for its ability to bridge incompatible interfaces, enhancing flexibility and maintainability.

## Bridge Pattern: Runtime Renderer Switching

To enable runtime switching between hardware and software rendering, the Bridge pattern was used. The Renderer interface defines rendering behavior, with HardwareRenderer and SoftwareRenderer providing concrete implementations. The MediaPlayerBridge abstract class, extended by AdvancedMediaPlayer, decouples rendering from playback logic, allowing AdvancedMediaPlayer to switch renderers dynamically (as seen in the main method’s user-driven selection). Unlike the legacy system’s hardcoded rendering, this pattern enables flexibility for device-specific optimization, chosen for its ability to separate abstraction from implementation, supporting runtime configurability.

## Decorator Pattern: Dynamic Feature Plugins

To support on-the-fly features like subtitles, equalization, and watermarking, the Decorator pattern was implemented via the Player interface and BaseMediaPlayer class. Decorators (SubtitleDecorator, EqualizerDecorator, WatermarkDecorator) extend PlayerDecorator, wrapping BaseMediaPlayer to add features dynamically, as shown in the main method’s conditional decorator stacking. This avoids modifying core playback logic, unlike the legacy system’s rigid approach, and allows runtime feature combinations (subtitles and watermarking). The pattern was selected for its adherence to the Open-Closed Principle, enabling extensible feature addition without altering existing code.

## Composite Pattern: Hierarchical Playlists

The requirement for composite playlists was addressed using the Composite pattern. The MediaItem interface unifies Track (single media) and Playlist (collections), enabling recursive composition. The Playlist class manages a list of MediaItem instances, and PlaylistManager displays the hierarchy, as demonstrated in the main method’s nested playlist setup. This overcomes the legacy system’s flat playlist structure, providing scalability for complex playlist hierarchies. The pattern was chosen for its ability to treat individual and composite elements uniformly, simplifying playback management.

## Proxy Pattern: Stream Caching

For remote stream caching, the Proxy pattern was applied via the RemoteMedia interface and RemoteMediaProxy class, which wraps RealRemoteMedia. The StreamController uses RemoteMediaProxy to cache remote streams (e.g., APIs), as shown in the main method’s conditional proxy usage. This reduces latency and bandwidth, unlike the legacy system’s direct streaming. The pattern was selected for its transparency—clients interact with RemoteMediaProxy as if it were RealRemoteMedia—and its ability to add caching without modifying source or playback logic.

## Why These Patterns?

The Adapter pattern, via Source and MediaApp, unified media sources, eliminating duplication. The Bridge pattern, with Renderer and MediaPlayerBridge, enabled flexible renderer switching, replacing hardcoded logic. The Decorator pattern, using Player and PlayerDecorator, supported dynamic features, addressing the lack of plugins. The Composite pattern, through MediaItem and Playlist, provided scalable playlist management. The Proxy pattern, with RemoteMediaProxy, added efficient caching. These patterns ensure modularity, extensibility, and maintainability, though decorator stacking may introduce performance overhead, mitigated by efficient implementation in MediaController. Alternatives like inheritance were rejected for their rigidity, ensuring the system aligns with project goals.