**Design Rationale**

The refactoring of the old MediaPlayer into a Modular Media Streaming Suite applies several structural design patterns to achieve flexibility, maintainability, and scalability. The legacy system was tightly coupled and hard to extend. Through the Adapter, Decorator, Composite, Bridge, and Proxy patterns, the new system becomes modular, reusable, and easier to evolve.

**Adapter Pattern**

**Where:**

The Adapter pattern is used to handle multiple media sources such as local files, HLS streams, and remote APIs. In the old system, the player contained many if-else statements to load different sources, creating repetitive and messy code. The refactored version defines a MediaSource interface with specific implementations like LocalFileSource, HLSSource, and RemoteAPISource. Each class adapts its unique fetching mechanism to a unified structure that the player can easily manage.

**Why:**

The Adapter pattern unifies different interfaces under one abstraction. It eliminates duplication, reduces code complexity, and supports the Open-Closed Principle — allowing new source types (like DASH or RTSP) to be added without modifying existing code. It also promotes polymorphism by letting the player interact with any media source through a common interface.

**Decorator Pattern**

**Where:**

This pattern is applied to add optional features such as subtitles, audio equalizer, and watermarking. Previously, these were controlled by hardcoded boolean flags inside the playback method. The new system introduces a MediaProcessor interface and decorators like SubtitleDecorator, EqualizerDecorator, and WatermarkDecorator, which wrap around the main processor. Example:

processor = new WatermarkDecorator(new SubtitleDecorator(processor));

**Why:**

The Decorator pattern allows dynamic addition of features without creating new subclasses for every feature combination. It supports runtime flexibility, Single Responsibility, and Open-Closed Principles, enabling users to stack or remove effects easily without changing core playback logic.

**Composite Pattern**

**Where:**

The Composite pattern manages playlists containing both single media files and nested sub-playlists. In the legacy system, only flat lists were supported. The new version defines a MediaItem interface, with FileItem representing single files and Playlist acting as a container holding multiple media items. This allows the player to play both single files and complex nested playlists.

**Why:**

Composite treats individual and grouped items uniformly. It simplifies playlist handling and supports recursive structures, allowing tree-like organization of media content. The client code can simply call player.play(playlist) regardless of complexity.

**Bridge Pattern**

**Where:**

This pattern separates playback abstraction from rendering implementation. The legacy design relied on a boolean flag to toggle between hardware and software rendering, creating tight coupling. The refactored design introduces MediaPlayerAbstraction that holds a Renderer interface, with implementations like HardwareRenderer and SoftwareRenderer.

**Why:**

The Bridge pattern decouples abstraction from implementation, enabling independent modification of both. It allows the player to switch rendering modes dynamically and makes future extensions (like GPU rendering) possible without altering core playback logic.

**Proxy Pattern**

**Where:**

The Proxy pattern is used for caching and controlling access to remote media streams. The RemoteStreamProxy acts as a stand-in for the real remote source. It fetches data once and stores it locally, returning cached data for future requests.

**Why:**

Proxy provides a surrogate interface to manage access efficiently. It improves performance, reduces network usage, and adds flexibility for extra functions like authentication or logging, all without changing the client-side code.

**Summary**

By applying these structural design patterns, the new Modular Media Streaming Suite becomes a cleaner, modular, and extensible system. While the number of classes increases slightly, the benefits in scalability, reusability, and maintainability far outweigh the added complexity. Together, these patterns ensure that the system follows SOLID principles, making it ready for future media sources, plugins, and rendering strategies.