**Design Rationale: Patterns Used and Why**

This refactor applies structural design patterns to transform a monolithic media player into a modular, extensible suite. Each pattern addresses specific legacy issues like duplicated code, tight coupling, and lack of plugins.

\*\*Adapter Pattern\*\*: Used for integrating HLS streams. The legacy code likely had duplicated adaptation logic for different sources. `HLSAdapter` wraps the third-party `ThirdPartyHLSPlayer` to conform to the `MediaSource` interface. This allows uniform handling of sources (local, HLS, remote) without modifying the external library, promoting compatibility and reducing duplication.

\*\*Proxy Pattern\*\*: Applied to remote sources via `RemoteProxy`. Legacy systems might fetch remote data directly, leading to inefficiency. The proxy lazy-loads and caches the `RemoteSource`, simulating caching for streams. This adds indirection for control (e.g., caching) without changing the client code, improving performance and maintainability.

\*\*Strategy Pattern\*\*: Implements runtime switching between hardware and software rendering. In the legacy codebase, rendering might be hardcoded. The `Renderer` interface with `HardwareRenderer` and `SoftwareRenderer` implementations allows dynamic selection in `BasicMediaPlayer`. This enables easy extension (e.g., adding a new renderer) and decouples the player from specific rendering logic.

\*\*Decorator Pattern\*\*: Enables on-the-fly feature plugins like subtitles, equalizer, and watermarking. Legacy features were probably embedded monolithically, making additions cumbersome. `PlayerDecorator` and subclasses (`SubtitleDecorator`, etc.) wrap the `MediaPlayer`, adding behaviors dynamically without altering the core player. This supports flexible combinations and adheres to the open-closed principle—open for extension, closed for modification.

\*\*Composite Pattern\*\*: Manages playlists with single files and sub-playlists. The legacy might treat songs and lists differently, causing complexity. `Playable` interface unifies `Song` (leaf) and `Playlist` (composite), allowing recursive structures. Methods like `play()` and `display()` work transparently on hierarchies, simplifying playlist assembly and playback.

These patterns were chosen for their focus on object composition over inheritance, fitting the lab's emphasis on structural designs. Adapter and Proxy handle integration and access control; Strategy provides behavioral flexibility; Decorator adds responsibilities runtime; Composite builds part-whole hierarchies. Together, they eliminate monolithic issues, enhance modularity, and make the system scalable (e.g., adding new sources or features is straightforward). Trade-offs include slight performance overhead from indirection, but benefits in extensibility outweigh this for a media suite.