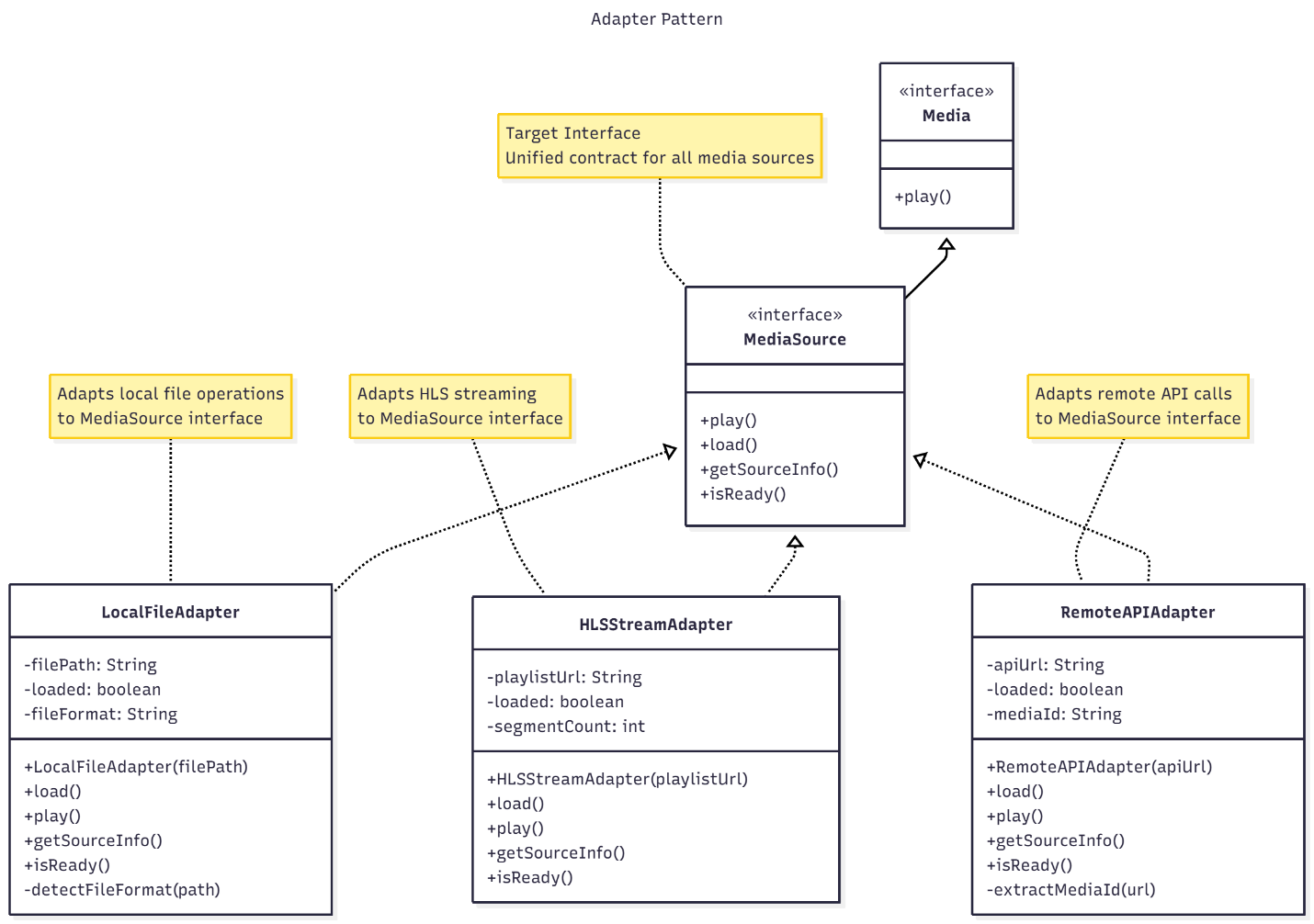
**John Lewis B. Hitio BSIT-3A**

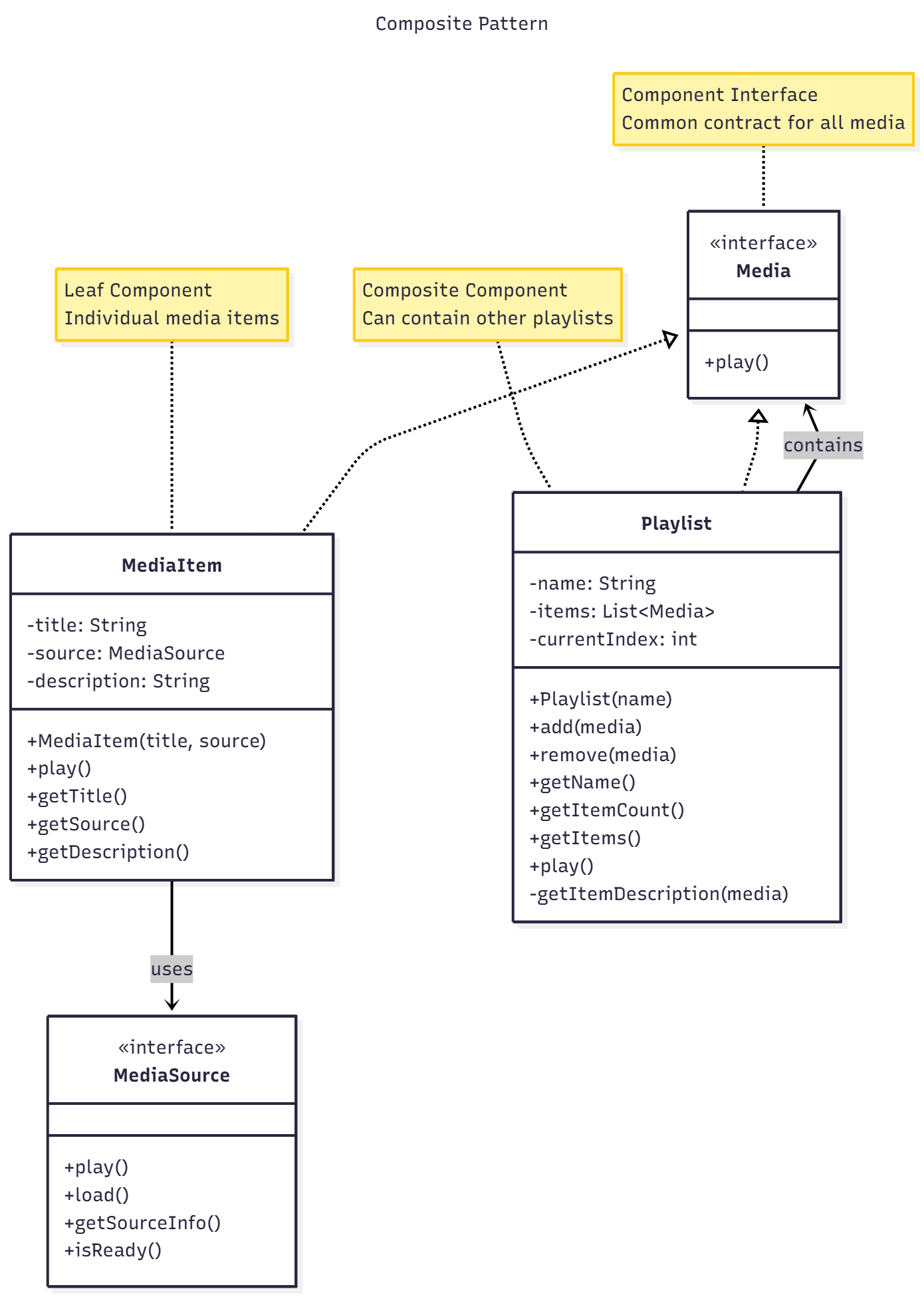
**Architecture Overview**

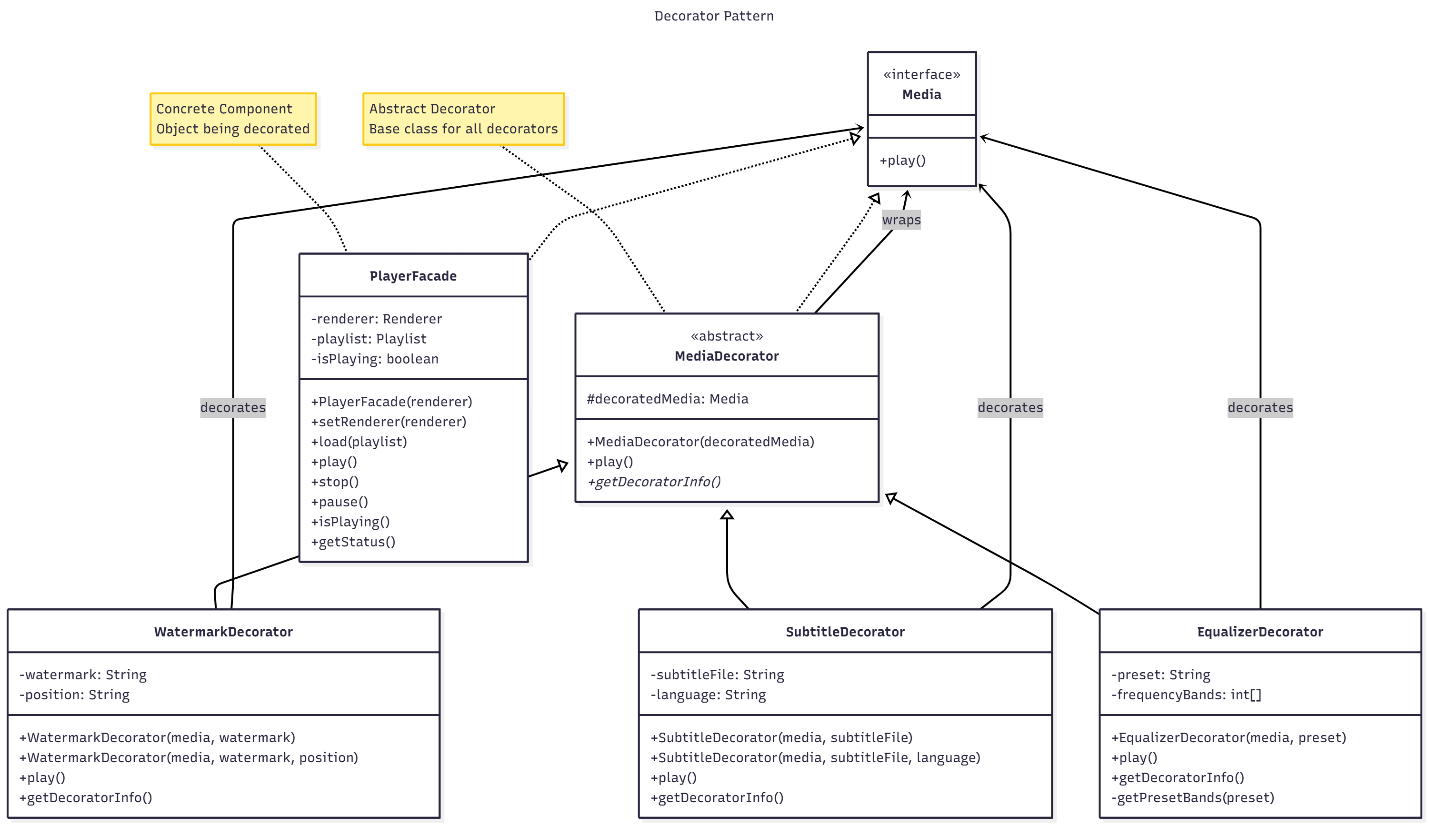
The system uses structural design patterns to build a flexible media player.  
PlayerFacade acts as the main controller, managing playlists, renderers, and media sources.  
Adapters unify different media inputs, Proxies handle caching, Composites manage playlists, Strategies switch rendering modes, and Decorators add features like subtitles or watermarks.  
This design keeps the system modular, easy to extend, and simple to use.

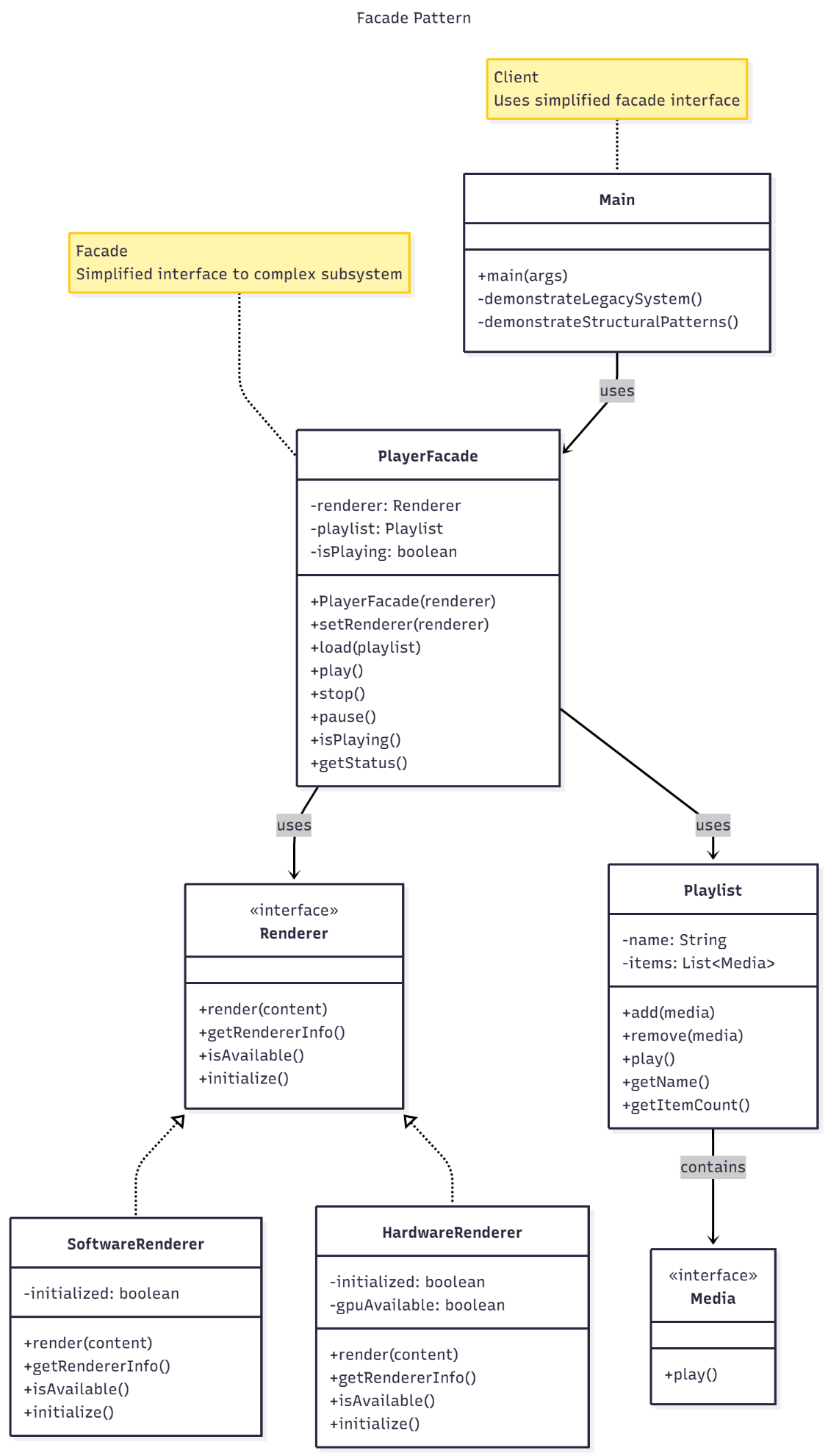
**UML Diagrams:**

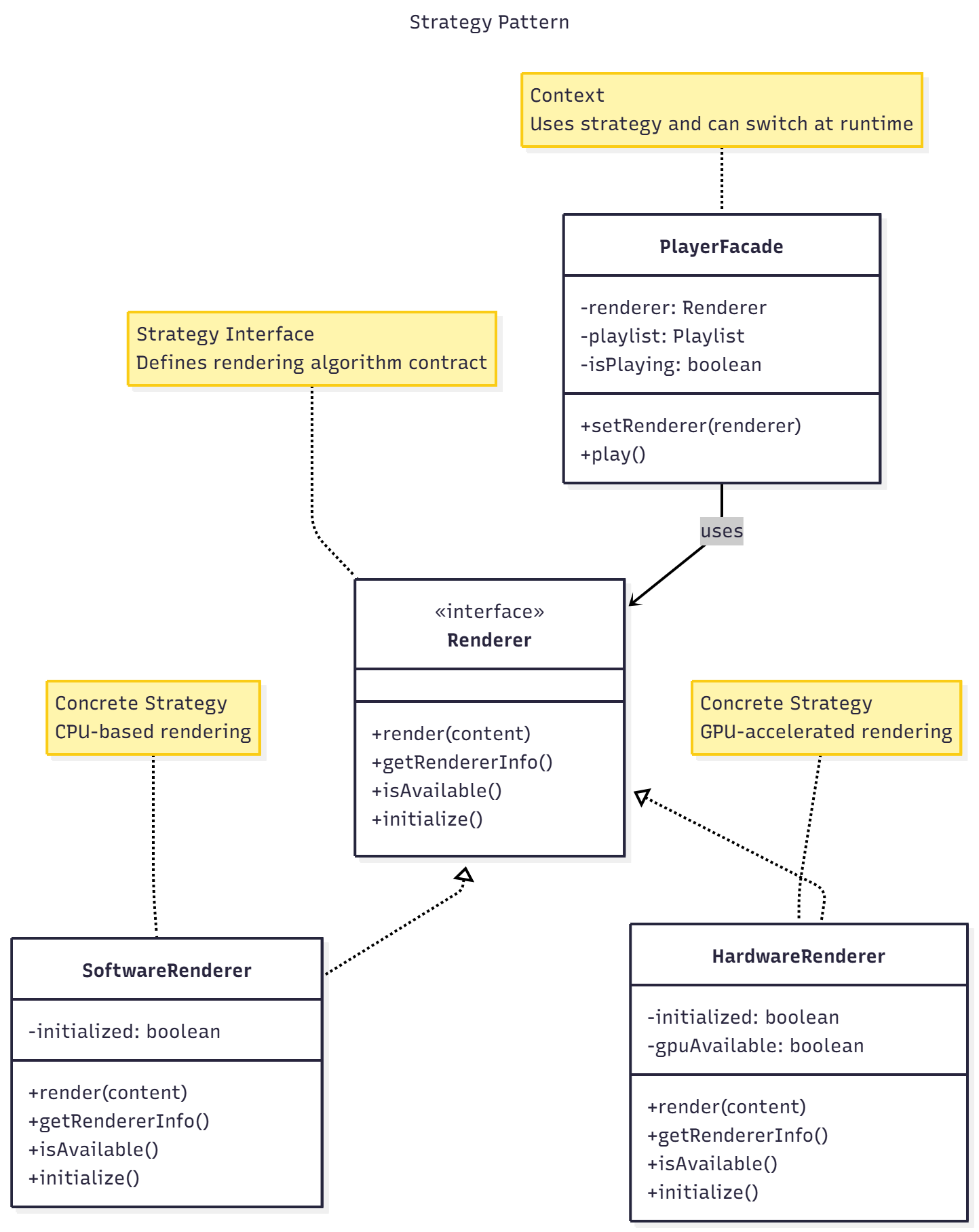


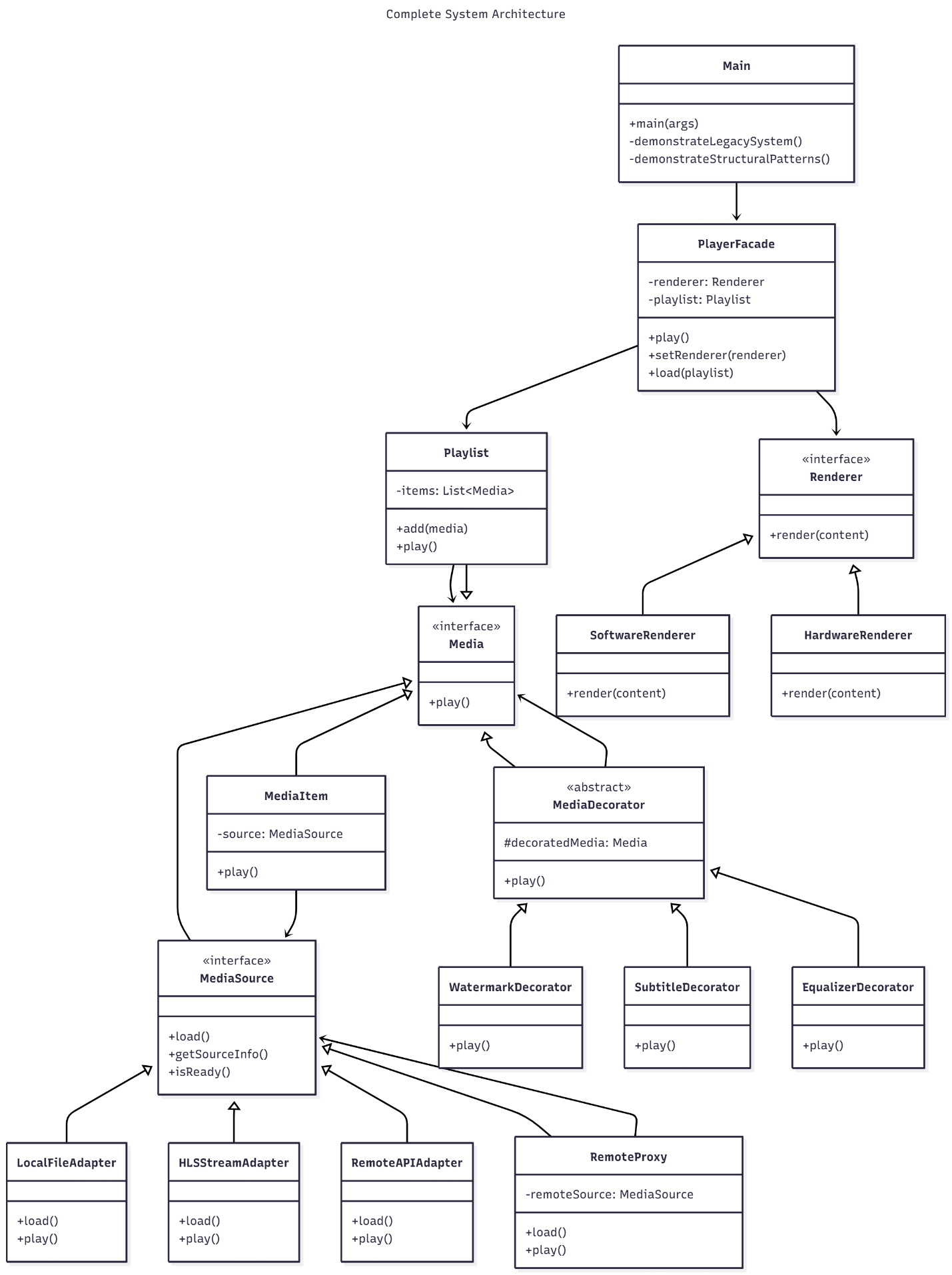




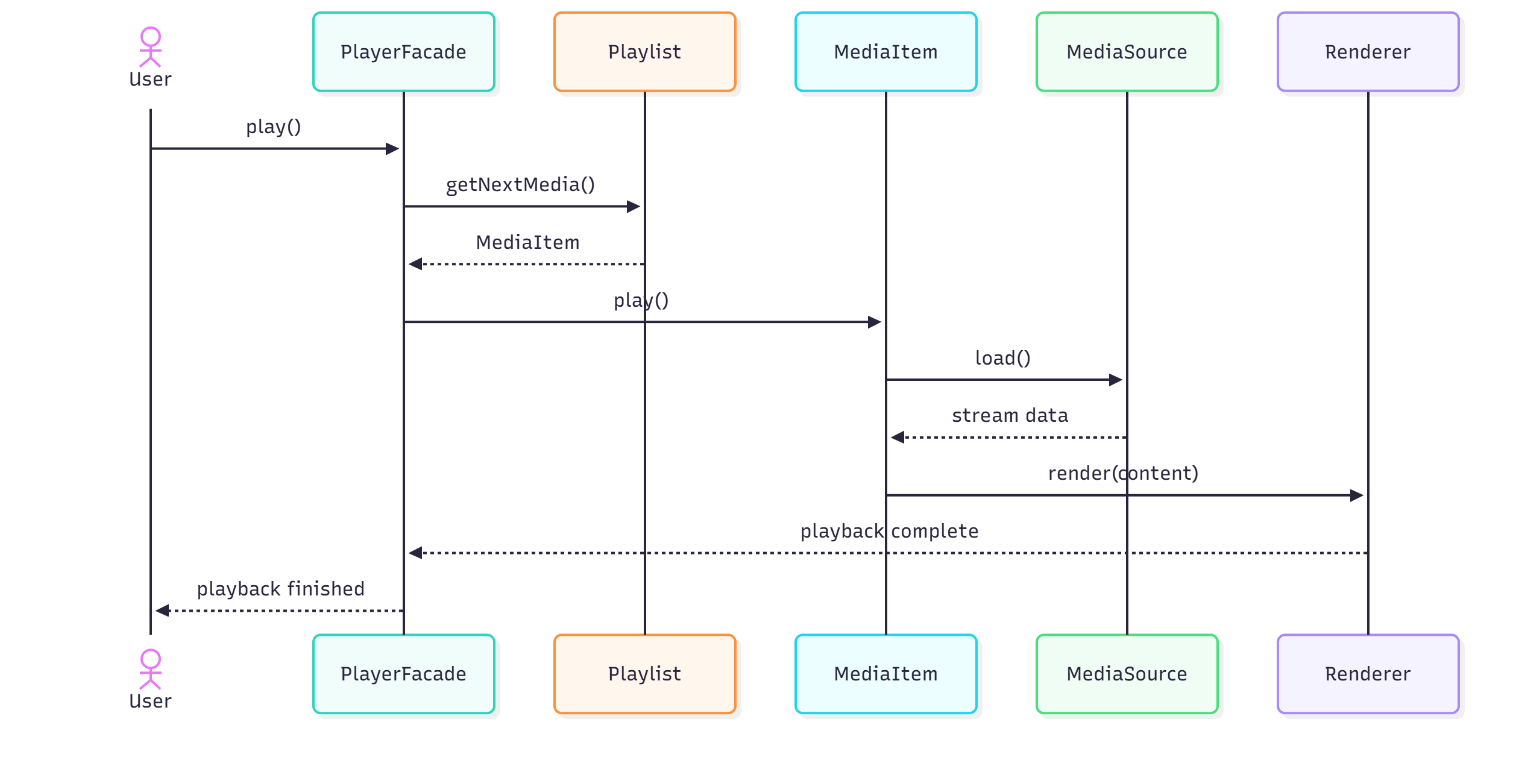




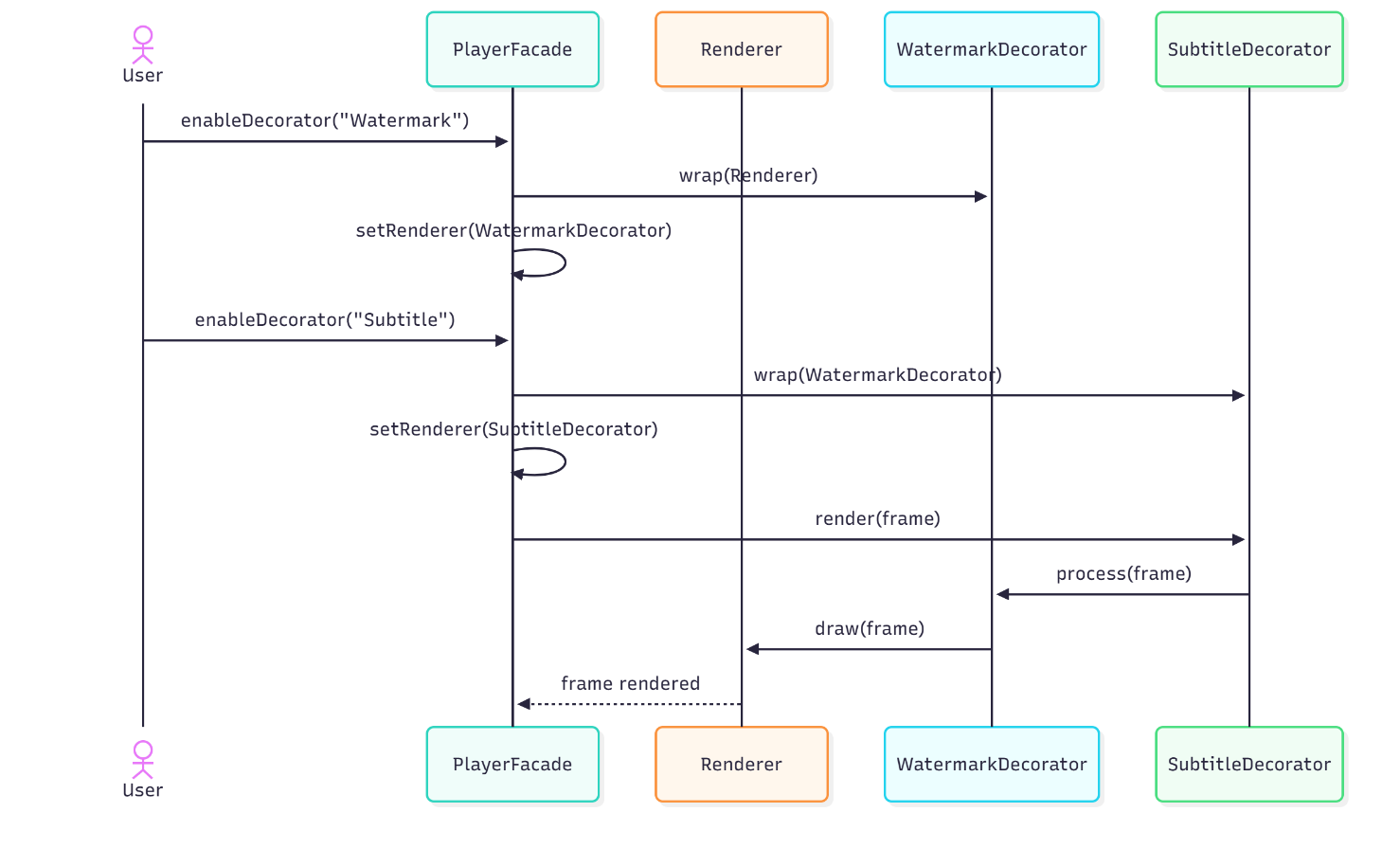




**Sequence Diagram:**

Play Media Flow  


Apply Decorator Stack



**Design Rationale: Structural Design Patterns Implementation**

**Adapter Pattern – Unified Media Source Interface**

The **Adapter Pattern** was used to make different types of media sources (like local files, HLS streams, and remote APIs) work the same way.  
Before, each type had its own code, which caused duplication and confusion.  
Now, all media sources follow one common interface called MediaSource.  
The classes LocalFileAdapter, HLSStreamAdapter, and RemoteAPIAdapter handle their own details but look the same to the rest of the system.  
This means we can add new source types later without changing existing code.

**Proxy Pattern – Transparent Caching System**

The **Proxy Pattern** was added to improve performance when loading remote media.  
The RemoteProxy class wraps a MediaSource and adds caching, so the system doesn’t have to download the same stream every time.  
It behaves exactly like the original source, so other parts of the program don’t need to change.  
This makes playback faster and saves bandwidth, while keeping the caching logic separate from the main code.

**Composite Pattern – Hierarchical Playlist Structure**

The **Composite Pattern** lets playlists contain both single media items and other playlists.  
Both MediaItem and Playlist use the same interface called Media.  
This means you can play a single file or an entire playlist using the same method.  
For example, a playlist called *“My Collection”* can include another playlist like *“Action Movies”*.  
When you call play() on a playlist, it automatically plays everything inside it.  
This makes the system simple and consistent.

**Decorator Pattern – Dynamic Feature Enhancement**

The **Decorator Pattern** was used to add extra features (like subtitles, watermarks, and audio effects) without changing existing classes.  
The MediaDecorator is the base class, and other decorators like WatermarkDecorator, SubtitleDecorator, and EqualizerDecorator add new features.  
These decorators can be added or removed at any time and can work together.  
For example, you can have both subtitles and a watermark active at once.  
This makes the system flexible and easy to update with new features in the future.

**Strategy Pattern – Runtime Renderer Switching**

The **Strategy Pattern** helps the player switch between different rendering methods at runtime.  
The Renderer interface defines how rendering works, and there are two main strategies: SoftwareRenderer (CPU) and HardwareRenderer (GPU).  
This allows the system to choose the best rendering option depending on the device or performance needs.  
It keeps rendering code separate and easier to test or replace without changing other parts of the system.

**Facade Pattern – Simplified System Interface**

The **Facade Pattern** makes the system easier to use by providing one main control point: PlayerFacade.  
Instead of dealing with playlists, renderers, and media sources separately, users can just call simple methods like play() or setRenderer().  
The facade hides all the internal complexity and handles coordination behind the scenes.  
This makes the whole system cleaner, easier to use, and less error-prone.