Kodi - Concrete Architecture

Date: November 18, 2023

CISC 322 - Software Architectures

Professor Karim Jahed

Prepared by Group 24:
Kobi
Edward Ng (Team Lead) - 20en3@queensu.ca
Arjun Devnani (Presenter) - arjun.devnani@queensu.ca
Raif Karkal (Presenter) - 20rrk2@queensu.ca
Abdul Moez Akbar - 20ama12@queensu.ca
Danyaal Sahi - 20dhs4@queensu.ca
Drake Li - 19dl53@queensu.ca

Abstract:

Kodi, originally known as Xbox Media Center, is an open-source software media player that has been available since 2002. This media player allows users to view every type of media such as movies, TV shows, music, etc. The source of media can come locally, through network media storage or from the internet and was designed to be used with televisions and remote control. However, Kodi has evolved to be available on other devices like mobile phones and laptops. XBMC Foundation developed this app using Python, C, C++, Assembly and Objective-C and can function on Android, Windows, Linux, and iOS. This report dives into the concrete architecture of Kodi by looking at the top-level subsystems and the internal architecture of one of these systems along with their interactions and the derivation process of coming up with the architecture. We will also discuss the difference between conceptual and concrete architectures using the reflexion analysis. We will then show use cases of the architecture using two sequence diagrams and discuss our lessons learned.

Top-level Concrete Subsystems And Their Interactions:

Kodi, a well-known open-source media player application, exemplifies complex software architecture by combining client-server dynamics, repository management, object-oriented principles, and a modular architecture. This combination improves Kodi's user interface and adds to its strength and adaptability, making it a popular option for a wide range of users. The modularity of the architecture also makes it so if components are removed, the system can still be functional.

The top-level subsystems presented are:

Media Playback System: The Media Playback System, an advanced subsystem built to handle a wide range of media formats, is essential to Kodi's functionality. The performance and functionality of this system, which links to external streaming providers and codecs, greatly affect the user experience.

Database Management: Kodi's Database Management subsystem, which indexes and saves media material to provide a dynamic and user-friendly browsing experience, is equally significant. The architecture of this subsystem ensures that user preferences, settings, and metadata are retrieved and used efficiently while collaborating closely with other subsystems.

Add-Ons: The Add-Ons subsystem of Kodi is the perfect example of its extensibility. This network of extensions improves functionality without compromising the integrity of the Kodi core system. The variety of services and customization choices provided by these add-ons are indicative of Kodi's versatility and user-centric design.

Setting: The Settings subsystem acts as the user's control panel, converting user selections into system configurations. Its complex interactions with other subsystems are essential for enabling customization without sacrificing the performance or dependability of the system.

The concrete architecture styles identified are:

Client-Server: Using a client-server architecture, Kodi's components can function as either clients or servers depending on the situation. For example, the media playback engine acts as a server and responds to requests from the user interface. The way Kodi interacts with internet services for streaming or metadata collecting further demonstrates this concept.

Repository: Kodi may function as a client to repositories housing a multitude of add-ons thanks to the Repository Model, which is essential for managing add-ons. This makes it easier for end users to expand the functionality of the program.

Object-Oriented: Object-oriented design principles have a strong influence on Kodi's design. This is seen by the way it is structured—it is modular and encapsulated—with parts like the media library managing its data and exposing particular functions. Inheritance and polymorphism may improve the user interface and media handling components, encouraging shared features and simplified administration.

Modular Architecture: A great example of a modular design is seen in Kodi's architecture, which uses more dynamic and adaptable patterns like event-driven, publish-subscribe (pub-sub), and pipe-and-filter architectures.

Kodi uses the pipe-and-filter approach to process media through a series of separate filters, each of which handles a particular component of rendering and decoding media. The media handling pipeline can be easily modified or extended, and processing can be done efficiently thanks to this modular architecture. The operation of Kodi's add-ons is a prime example of the publish-subscribe approach. Subscriptions to particular events or services allow components to respond independently to system actions and changes. With the use of this approach, Kodi may be made more extensible and interactive, enabling real-time updates and responsive features. The modular nature of Kodi is further emphasized by its event-driven architecture. In response to user inputs or system events, various modules react independently, demonstrating the system's ability to manage multiple tasks at once without requiring intermodular dependencies.

Concrete Architectural Interactions:

The interplay between subsystems is clearly shown in the detailed architectural view. The Database Management subsystem supports the Media Playback System, which is its core and is configurable via the Settings. By integrating with every other subsystem, the Add-Ons subsystem improves user engagement and adds new features. The core of Kodi's practical elegance lies in this interaction.

Derivation Process

To determine Kodi’s concrete architecture, we used a variety of methods including code analysis tools, documentation review, dependency analysis with architecture visualization, pattern recognition, and domain knowledge. The details of each of these processes are given below in the chronological order that they were performed.

Code Analysis Tools: To derive Kodi’s concrete architecture, the two main tools used were *Understand* and *GitHub.* Both Understand and GitHub were used to access Kodi’s source code to analyze its structure. Access to the Understand software proved to be a powerful tool when fed Kodi’s source code as it helped us gain more detailed information on the software’s code that may be overlooked through just regular code analysis. Through Understand, we could map all directories and files to determine the concrete architecture.

Domain Knowledge: Since Kodi’s main purpose is known by the team, the knowledge of the software’s domain and operation allowed us to more effectively analyze and rationalize the concrete architecture. The knowledge of the software’s workings and what it is designed for allows for a better understanding of the rationale behind specific architectural design decisions.

Documentation Review: Documentation review was utilized through information in Understand as well as internal documentation in the source code. The documentation analyzed included architecture diagrams obtained through Understand as well as comments obtained from the source code on both Understand and GitHub.

Dependency Analysis with Architecture Visualization: We also performed a dependency analysis on Kodi's features. Understand allowed us to view architecture visualizations of the dependencies between classes, modules and components through UML diagrams. This allowed us to better understand the structure of the codebase and determine Kodi's concrete architecture as compared to our conceptual architecture in Assignment 1. Third-party dependency analysis was also utilized to understand the role of third-party libraries, how they interact with various components of the software, and how they fit into the overall architecture.

Pattern Recognition: Regular code analysis was also used to supplement the other findings from Understand. Common patterns were identified in the code which provided clues about the architecture and design principles. For example, pattern matching was helpful when identifying portions of code that allowed for third-party plug-ins/APIs.

Overall, all these methods were utilized to analyze, visualize and determine Kodi's concrete architecture using the tools GitHub and Understand. The findings were then used throughout the report as our proposed concrete architecture and to compare it to our conceptual architecture derived in Assignment 1. Through our derivation process, we determined the concrete architecture better suited to a *pipe & filter, client-server* and *pub-sub* architectural style as compared to the layered conceptual architecture proposed earlier.

Derivation Process Motivations

These styles were derived from the reference architecture on GitHub (Dreef et al., 2015) using the developer view. The information gained from the reference architecture and the analysis performed using the methods described above influenced our mapping process of the Kodi modules in Understand. This way we were able to determine the top-level subsystems and how they interact with Kodi's components.

Internal Architecture and Interactions of Video Player Subsystem:

The internal architecture of the video player subsystem contains nine components as shown in Figure 1, each with its purpose and interactions. All these components contain events that can be triggered based on user interaction or system changes containing subscribers of these events. Each component is not dependent on each other, however, they invoke events of each other. Therefore, Kodi uses a publish and subscribe concrete architecture. This diverges from our previous layered conceptual architecture, as components do not need to be in adjacent layers to communicate. This will be discussed later using a reflexion analysis. The components for the internal architecture of the video player subsystem include:

Input Stream: The DVDInputStreams handles user input and commands related to video playback. The input commands can be sent to the Process component, which handles playback. These inputs can come from peripherals like keyboards and remotes.

Demuxer: The DVDDemuxers component involves taking multimedia content and breaking it into separate streams, audio and video streams. These streams can then be fed into the codecs which is a decoder to transform these streams into more useful data for playback.

Decoder: Another component is the DVDCodecs component, where the system decodes audio and video data into a format that can be used for playback. The decoded audio and video can then be sent to the audio and video renderer.

Video and Audio Renderer: This component is responsible for rendering video and audio frames. This module provides a proper output for the GUI to play which is handled in Interface.

Subtitles Handler: The DVDSubtitles component manages the subtitles by rendering and synchronizing them to the video.

Buffer: The Buffer component ensures synchronization between audio and video streams for a seamless viewing experience.

Process: The Process node manages the overall control of the playback process, including play, pause, stop, and seek operations. This component works in conjunction with the input stream to take in user commands for playback.

Interface: The Interface component may interact with external interfaces and will output the rendered video and audio transformed from the original streams for the user to watch.

Test: Lastly, the Test component contains test cases for the Video Player subsystem to ensure the subsystem meets the code quality.

All these components, although independent of one another, interact with each other to perform video playback through a pipe and filter concrete architecture style by transforming data through the pipeline. For example, a video can be passed into a demuxer to separate it into two streams, then decoded by the codecs to receive video and audio data that can be used in playback. Finally, it gets rendered by the video renderer and outputted to the GUI. While playing on the GUI, the Input Stream can trigger events in Process to adjust playback in a publish and subscribe style. As seen in the dependencies graph (Figure 2) for the Video Player subsystem, it also interacts with other subsystems outside of the Media Playback System. This includes the Add-ons, Database Management, and Settings subsystems. As discussed earlier, the Video Player module depends on the Database Management subsystem and is configurable using the Add-ons and Settings subsystem.

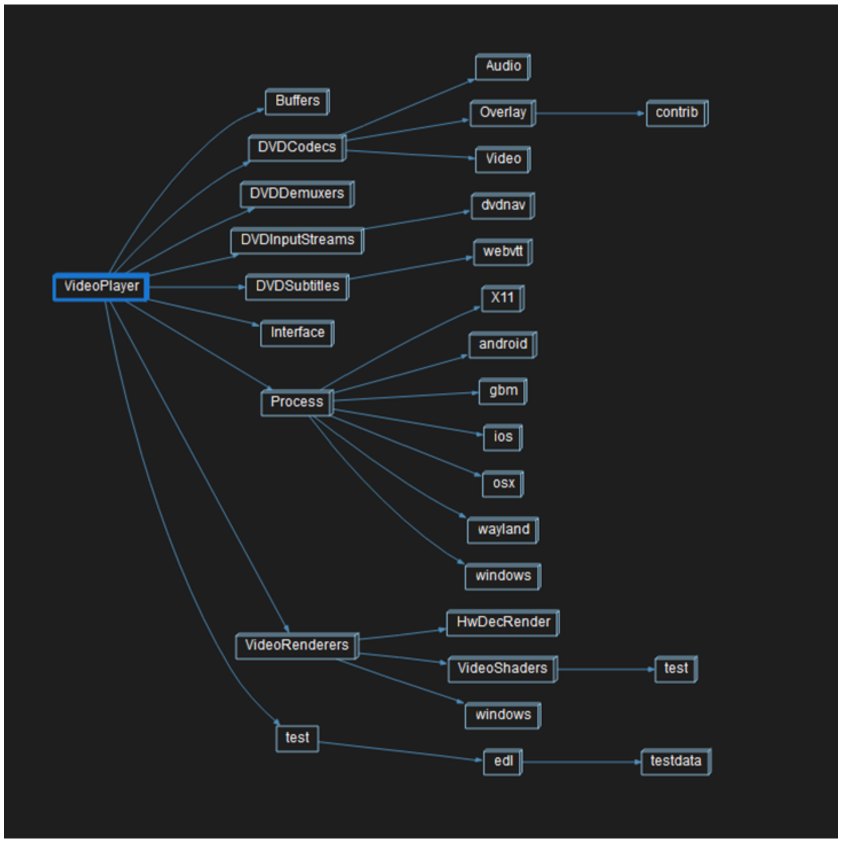

Figure 1. Concrete architecture of the video player subsystem from Understand.

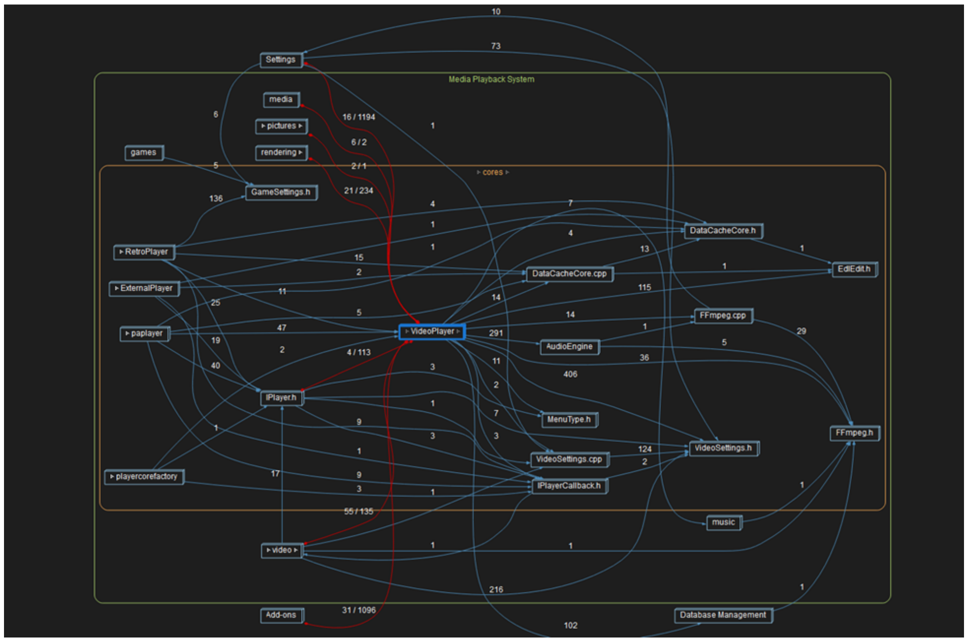

Figure 2. Dependencies graph of the video player subsystem from Understand.

Conceptual vs Concrete Architecture:

Through reflexion analysis, we can compare the system's conceptual architecture versus concrete architecture to further strengthen our understanding of its components, subsystems and interactions. Through heavy research in our first report, we have concluded that Kodi's conceptual architecture revolves around layered architecture (Ng et al., 2023). This layered architecture encompasses 4 main layers. A client layer for running Kodi, a presentation layer for input and output, a business layer for connection to external servers, and a data layer to store all media. Furthermore, we determined that Kodi makes heavy use of modularity, allowing components to interact while remaining independent (Ng et al., 2023). This conclusion was drawn due to Kodi's extensive use of components allowing developers to seamlessly create add-ons while creating minimal interface clutter. However, Through reflexion analysis, we have determined that there are many differences between Kodi's concrete architecture versus our conceptual analysis of the program. The first divergence is the lack of layered architecture in Kodi's structure. A major reason for this is due to Kodi's adherence to modularity (Dreef et al., 2015). By placing such an emphasis on modularity and ease of development Kodi's architecture would need to be highly flexible and extensible. Utilizing a layered architecture is not optimal for this case, especially when compared to other architectural styles. Another divergence is Kodi's use of pipe and filter architecture. Due to Kodi's use cases, there are many interacting components, as well as unique steps required to perform tasks depending on the user, and environment. This is showcased in Kodi's integration in multiple environments (Mac OS, IOS). A final divergence between the conceptual and concrete architecture is Kodi's use of client-server architecture (Dreef et al., 2015). Kodi allows users to construct a centralized media center to hold all desirable files. Kodi then acts as a client to process these files. Kodi also heavily utilizes the web server for its functionalities. For example, online video streaming, as well as add-ons sourced from the web. However, there are also convergences between the architectures. Our view on Kodi's reliance on modularity and object-oriented programming was further strengthened when looking at Kodi's actual architecture. This allows for Interactions between independent components as well as ease of use in developing and installing add-ons.

Second-Level Subsystem Reflexion:

A second-level subsystem revolving around video playback would have many linking components with varying dependencies (Sakhare et al., 2014). The first notable component would be the renderer module. The renderer module would be utilized to render media content both visually and audibly. It would also handle aspects such as resolution scaling and frame rate control. Depending on the renderer would be a subtitle component. This component would be utilized for generating captions and subtitles based on the given media. Again, depending on the renderer component is the player core. The player core allows for playing capabilities for media. For example, playing a video, or listening to a song. Most notable is an input handler. The input handler would allow users to input into any aspect of the software. For example, playing/pausing a video, rewinding or fast-forwarding a song, and allowing input for searching/navigating the home screens. This would be dependent on the player core to allow for input specific to the current media playback. Another notable component would be an error-handling module. This module would depend on the renderer, as well as the player core components. It would detect when there are issues regarding media rendering or playback. A final component would be an interface component. This component would be utilized to connect the components as well as display a graphical user interface for the program. Looking at the concrete architecture there are divergences from our conceptual architecture. First, there are many more components within Video Player than outlined. The concrete architecture outlines 9 components, only 4 of which were mentioned. Some missing components are buffers, DVD demuxer, process, test, and DVD codecs. Buffers are utilized for smooth uninterrupted playback and contain no dependencies. DVD demuxer is utilized for demultiplexing media used for separating streams, again with no dependencies. Furthermore, due to Kodi's cross-platform integration, they require a component relating to integration and OS detection. The process component is used to detect, and allow for integration within various environments, such as IOS, OSX, Windows, and Wayland. The test has a single dependency labelled EDL. It is utilized to determine how media should be edited and played back including information such as cuts and trims to the content. Finally, DVD codecs are utilized for decoding the media to then be displayed. It contains 3 dependencies which are the formats in which the media is decoded. The dependencies include Audio, Video and Overlay. Moreover, the dependencies are structured more linearly and straightforwardly in the conceptual architecture. It was expected that there would be many dependencies following the renderer, and video playback components, but in actuality, the component with the most dependencies was process referencing the many operating systems Kodi is compatible with. This is further showcased by the Video Renderer component which is only linked to components directly related to video rendering, such as HwDecRender, and Video Shaders, diverging from the expected connections with an input handler and exception handler.

Use Cases and Diagrams:

Use Case #1 - Muting Audio:

In Figure 3, the actor initiates the action of muting the audio in this sequence diagram by sending the "SetMute(true)" command to the "CApplicationPlayerInstance" participant, which represents the player instance. After the command is received, the "CApplicationPlayerInstance" turns on the "AudioControl" participant, which is an audio-related component. After processing the command, "AudioControl" mutes the audio. The "User" actor is then informed by the "CApplicationPlayerInstance," which relays the result—that is, that the audio is muted—back to the actor.

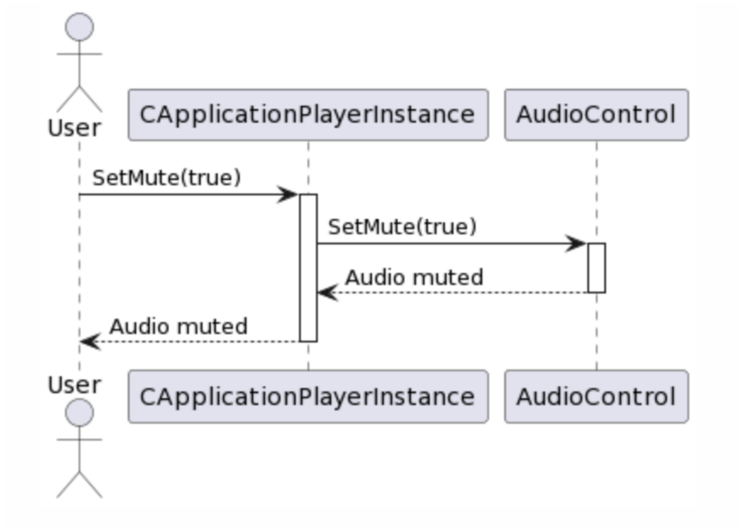

Figure 3. Sequence diagram for user muting the audio

Use Case #2 - Set Playback Speed:

In Figure 4, the "User" actor starts the action of changing the playback speed to 1.5 times by instructing the "CApplicationPlayerInstance" participant, which stands in for the player instance, to "SetPlaySpeed(1.5)". The "CApplicationPlayerInstance" initiates the "PlaybackControl" participant, which is in charge of overseeing playback-related operations, upon receiving the instruction. After processing the instruction, the "PlaybackControl" increases the playback speed by 1.5 times. The "User" actor is then informed by the "CApplicationPlayerInstance," which in turn notifies the "CApplicationPlayerInstance" of the altered playback speed.

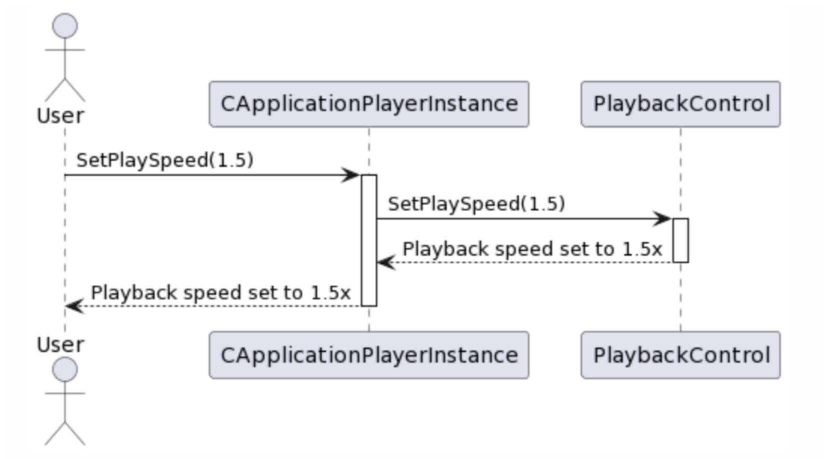
User
CApplicationPlayerInstance
PlaybackControl
SetPlaySpeed(1.5)
SetPlaySpeed(1.5)
Playback speed set to 1.5x
Playback speed set to 1.5x
User
CApplicationPlayerInstance
PlaybackControl

Figure 4. Sequence diagram to set playback speed

Figure 5 showcases the dependencies and interactions between these components. Specifically, the Database Management subsystem supports the Media Playback System, acting as its core, and is also configurable through the Settings. Additionally, Database Management collaborates with the Add-Ons subsystem. The Add-Ons subsystem enhances user engagement and introduces new features by interacting with both the Media Playback System and Database Management. Furthermore, the Settings subsystem configures all three components. The dotted lines represent divergence from the conceptual architecture diagram that we included in the first assignment. In this case, it is the interactions between components that are added after analyzing the concrete architecture of Kodi.

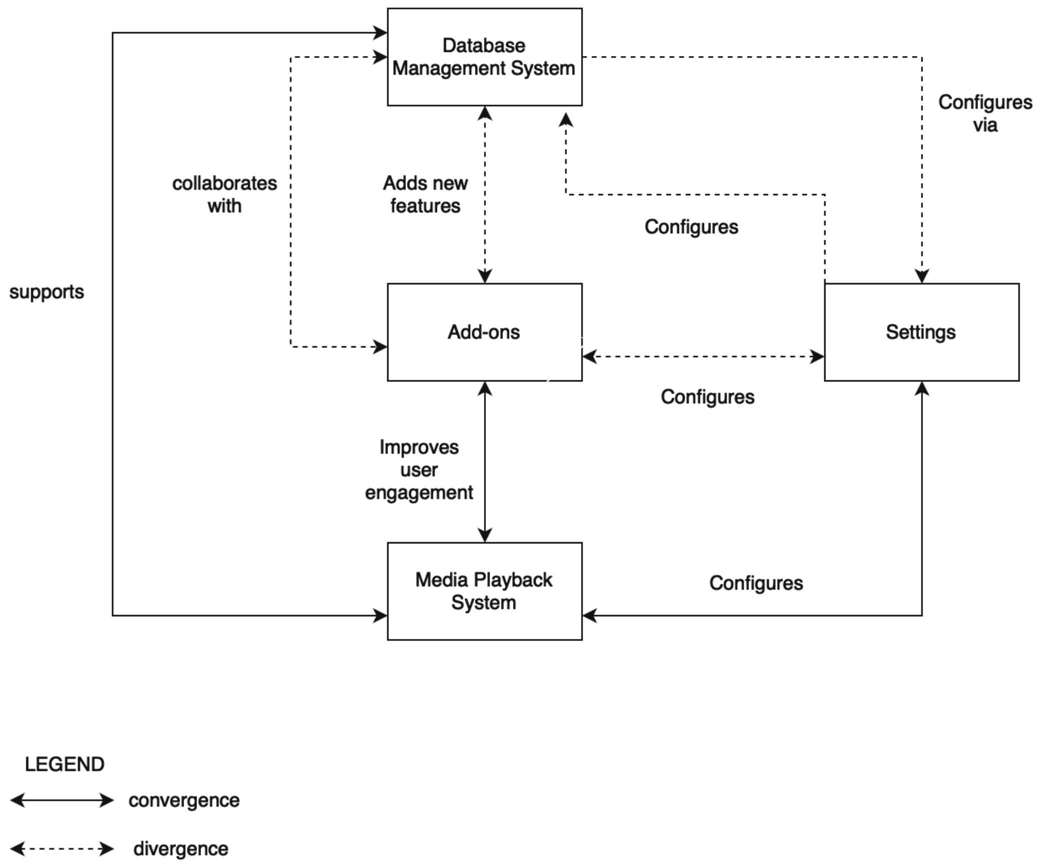

Figure 5. Box and Arrow Diagram for top-level sub-systems.

<u>**Lessons Learned:**</u>

Using the Understand software to analyze Kodi's source code, we have learned that our initial proposed layered architecture was wrong. Instead, a better architecture would be a combination of pipe and filter along with pub-sub. When analyzing the Video Player subsystem, it reveals a pipe and filter architecture. The components such as input stream, demuxer, decoder, video and audio render, form a pipeline where data flows through various processing stages, transforming it from raw input to a rendered output. This architecture allows for modularity and flexibility as each component can operate independently while contributing to the overall playback experience. Additionally, the pub-sub architecture becomes evident in the communication between the client side and server side of JSON-RPC. Pub-sub enables ease of scalability, availability, and latency, creating a flexible and loosely coupled system.

<u>**Conclusion:**</u>

In conclusion, this report examines the concrete architecture of Kodi, revealing its top-level subsystems, interactions, and internal architecture of the Video Playback subsystem. The derivation process, employing diverse methods such as code analysis tools, domain knowledge, and dependency analysis, reveals Kodi's adaptable modular architecture. The examination of the video player subsystem reveals its components and interactions, demonstrating a combination of pipe and filter combined with pub-style architecture. Using tools like Understand and GitHub helps with detailed analysis of the source code. Overall, this report enriches our understanding of Kodi's architecture while emphasizing the dynamic nature of software systems and the significance of adaptability in architectural assessments.

References:

About Kodi. Kodi. (2023). https://kodi.tv/about/

Architecture. Architecture - Official Kodi Wiki. (2023). https://kodi.wiki/view/Architecture

Video playback. Video playback - Official Kodi Wiki. (2023). https://kodi.wiki/view/Video_playback

Dreef, K., Reek, M. van der, Schaper, K., & Steinfort, M. (2015). *Architecting software to keep the lazy ones on the couch*. Kodi. https://delftswa.github.io/chapters/kodi/#modular-design

Ng, E., Devnani, A., Karkal, R., Akbar, A., Sahi, D., Li, D.. (2023). Kodi - Conceptual Architecture [Unpublished paper]. Computing Department, Queen's University.

Sakhare, Nikhil & Wasnik, Mithil & Tembhurkar, Manish. (2014). Design of Multimedia Player based on Android. https://www.researchgate.net/publication/273145809_Design_of_Multimedia_Player_based_on_Android