CISC 322

A2

Kodi - Conceptual Architecture

Date: November 19, 2023

Authors:

Ricardo Chen (Team Leader) – 21rc20@queensu.ca

ZhouJun Pan (Presenter) – 19zp9@queensu.ca

Jiahao Ni (Presenter) – 19jn35@queensu.ca

Lucy Zhang – 19ytz@queensu.ca

Shuaiyu Chen - 21sc33@queensu.ca

1.0 Abstract:

This report explores Kodi's concrete architecture style by integrating the analysis of its conceptual architecture from previous reports with our new understanding of Kodi's system

at the concrete level. The goal is to comprehensively examine the discrepancies between Kodi's conceptual and concrete architectures. The focus of this report is on providing detailed explanations of Kodi's top-level concrete subsystems and one of its 2nd-level subsystems, along with their interactions. Initially, we used "Scitools Understand" to map source files to the conceptual components established in prior analyses, allowing us to delve into the architecture of these subsystems and fully understand their interactions and dependencies on the concrete level. Then, to align with our updated understanding, we modified our conceptual architecture by adding and removing necessary components. Reflection analysis is also performed for both the high-level architecture and the selected 2nd-level subsystem to exam the discrepancies. In addition, two user cases are presented using sequence diagrams to better illustrate how the concrete architecture operates. Finally, we draw our conclusions and discuss the lessons we learned.

2.0 Introduction and Overview:

In our previous discussions, we delved into the conceptual architecture of Kodi, unraveling its structural framework (see Figure 1). This report takes a closer look at Kodi's concrete architecture, exploring how the conceptual design is put into practice within the system in reality. Notably, distinctions between conceptual and concrete architectures are evident, with the latter focusing on the actual implementation and execution of the conceptual architecture in real-world scenarios. Discrepancies between dependencies and subsystem components in conceptual and concrete architectures are common, prompting a detailed examination to identify which components or their dependencies need adjustment, highlighting the significance of reflection analyses.

Our approach involves a thorough study of the given source files of Kodi, the mapping between the top-level source files and our conceptual components, and comprehensive reviewing of the dependencies graph generated by "Scitools Understand." Through these efforts, we've uncovered additional subsystems and unexpected dependencies, contributing to a more comprehensive understanding of the system's architecture. Our findings reveal 3 new components utilizing the pub-sub architectural style, while removing 4 unnecessary components on the concrete level. This derivation process will be explained in detail in this report.

Additionally, we revisit the two use cases and sequence diagrams presented in earlier reports, making necessary modifications based on the refined conceptual architecture. The concluding section of this report provides an overview of Kodi's concrete architecture. We also address limitations encountered during this analysis and share valuable insights gained from our exploration of concrete architecture.

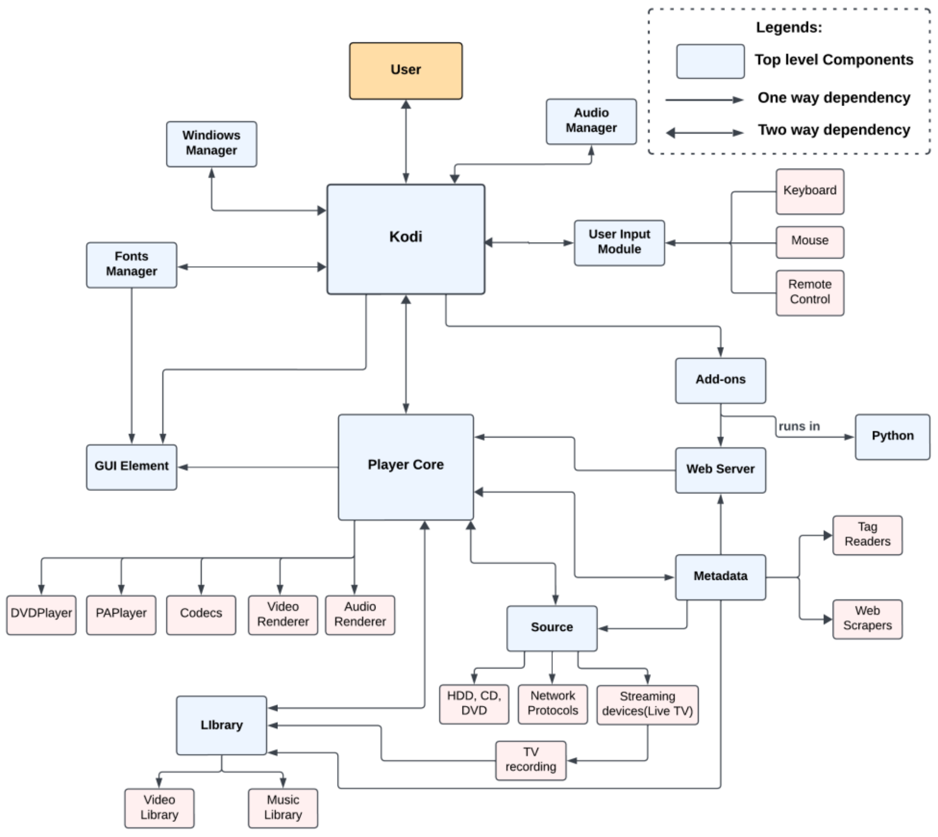


***Figure 1**: Original Conceptual Architecture*

3.0 Architecture Derivation Process:

Understanding the concrete architecture of Kodi involved a comprehensive derivation process. Our journey began with the conceptualization of Kodi's architecture in the previous report. Once all the top-level subsystems were determined, we gained a better insight into how Kodi's system operates and how each operation is achieved through the interactions of different layers.

To delve into the complexity of Kodi's concrete architecture, we initiated the process by creating and examining the raw concrete architecture—a product of mapping all source files to our proposed conceptual subsystems, as generated by "Scitools Understand" (See Figure 2). While travelling through the road of mapping source files to the proposed conceptual subsystems, we encountered challenges inherent to the abstract nature of the conceptual architecture. Instances arose where certain subsystems lacked corresponding source files, and conversely, files posed challenges in aligning with appropriate subsystems, highlighting potential gaps between our conceptual architecture and the concrete implementation. Despite these challenges, effective team communication enabled us to gather diverse insights, ensuring that we completed the mapping with the best fits possible, this raw concrete architecture served as our baseline, offering a snapshot of the initial state of the concrete architecture graph before any refinement. In the journey of mapping conceptualization to concrete implementation, we recognized the dynamic nature of translating abstract concepts into concrete instances. The mapping process not only revealed areas of alignment but also identified discrepancies, prompting us to refine and redefine the architecture for a more accurate representation of Kodi's system.

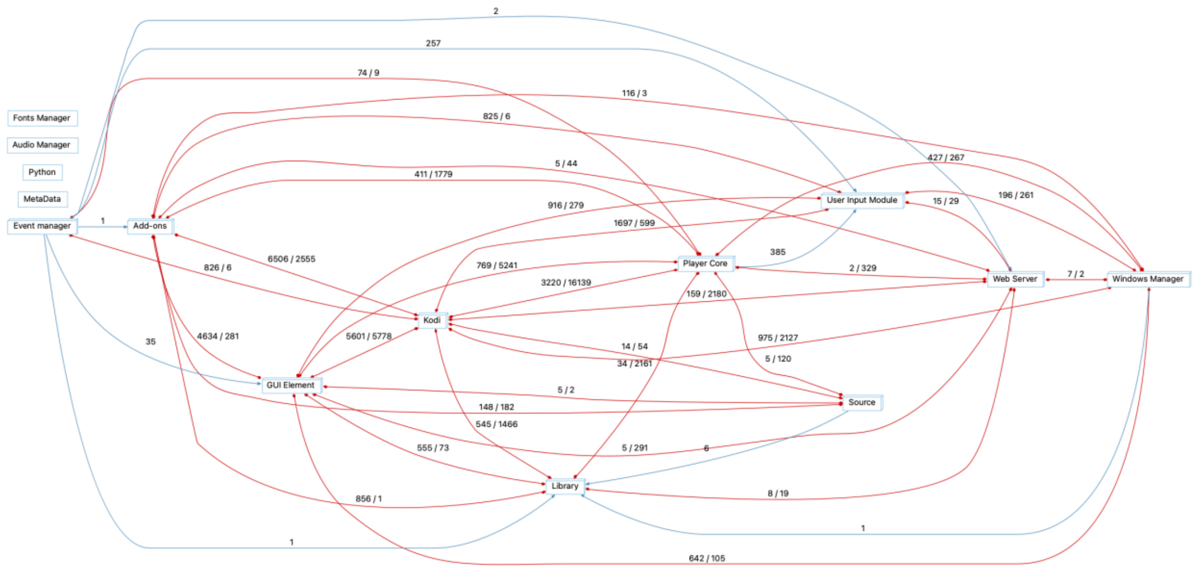


***Figure 2**: Raw Concrete Architecture Obtained by Understand with original subsystems*

As we moved through the refinement stages, our attention turned to resolving the identified discrepancies in the concrete architecture. This involved a thorough analysis of missing subsystems and files that posed challenges in categorization, requiring a deeper understanding of the relationships between the files and subsystems. The objective was to ensure that every proposed subsystem found a counterpart in the refined concrete architecture. Conversely, each top-level entity file found its corresponding subsystem. If this wasn't achievable, then we needed to either add or remove subsystems. This approach aimed at providing a comprehensive mapping that accurately reflected the connections between the conceptual and the actual implementation.

During this refinement, by practicing the W4 Approach we learned in class, we added three crucial subsystems to enhance the completeness and functionality of Kodi's concrete architecture. The "Event Manager" was introduced to facilitate efficient handling of system events, contributing to improved responsiveness. The "Storage" subsystem was incorporated to manage data storage, enhancing the overall data management capabilities of Kodi. Additionally, the "Platform" subsystem was introduced to provide a robust foundation for system operations, ensuring compatibility and efficient functioning across diverse platforms (e.g. Linux, Windows, Android).

Concurrently, to streamline and optimize the architecture, we made the decision to remove four subsystems that were deemed unnecessary for Kodi's operational efficiency. The “Fonts Manager” and “Audio Manager” were eliminated, as their roles were absorbed by other components, e.g. the functionality of “Audio Manger” is included in the GUI subsystem. The “Python” subsystem was removed to simplify the architecture, eliminating unnecessary complexity, given that all the necessary source files were mapped into the Add-ons subsystems. Lastly, the "Metadata" subsystem was excluded, as its functions were integrated into other existing components, ensuring a more streamlined and efficient architecture. After the refinement process, we obtained the updated concrete architecture generated by "Scitools Understand" (See Figure 3).

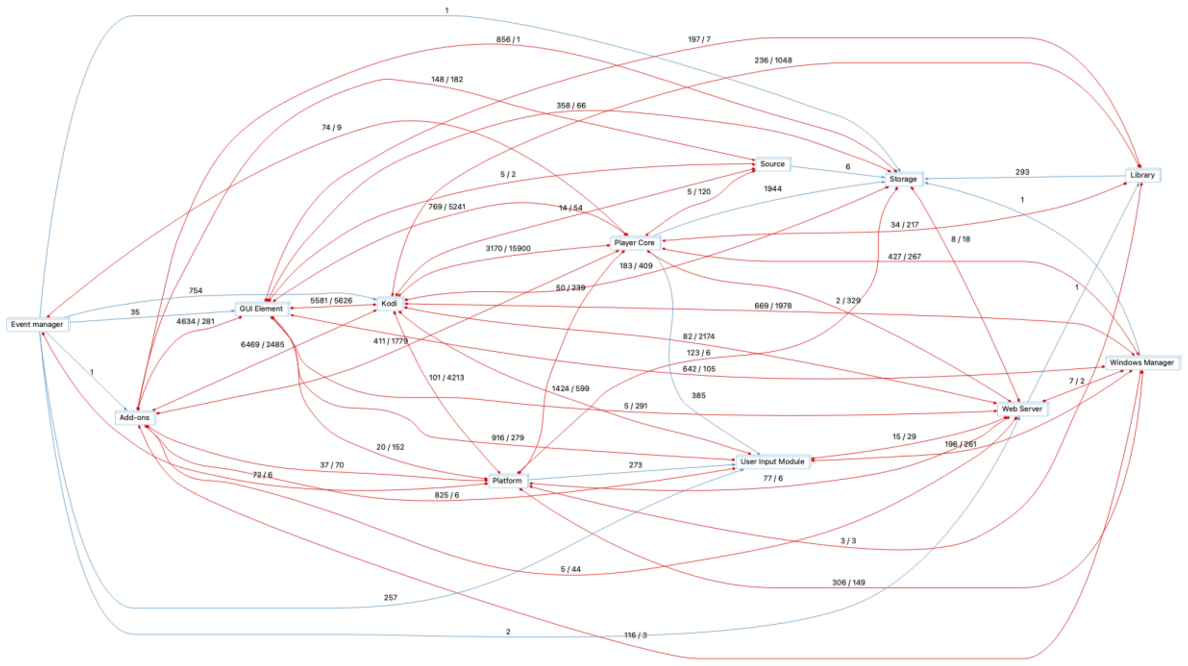


Figure 3: *Concrete Architecture Obtained by Understand with updated subsystems*

4.0 Architecture Analysis

4.1 Concrete Architecture Subsystems and Interactions

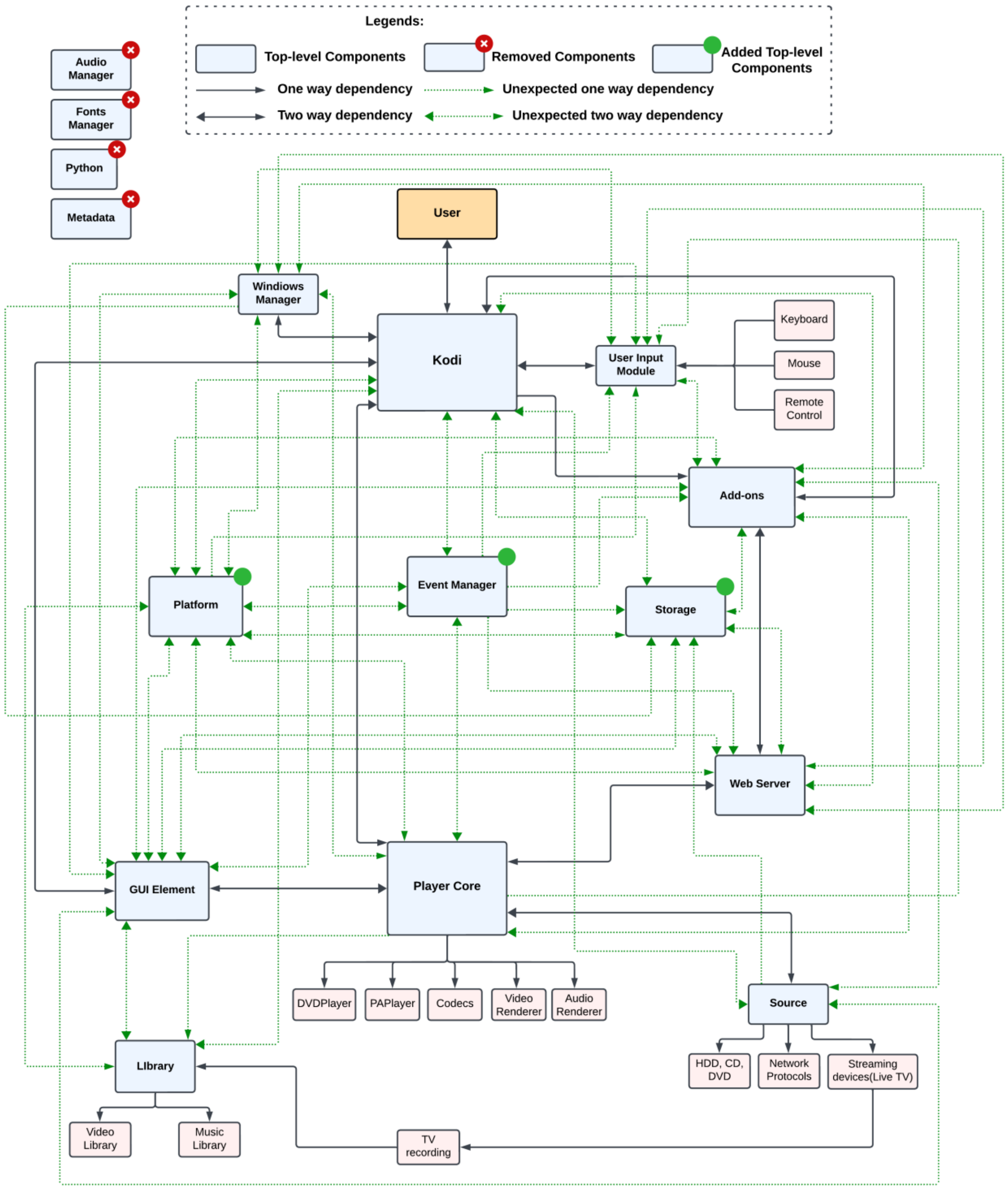


Figure 4: *Concrete Architecture of Kodi*

In Kodi's concrete architecture, the Pub-Sub is the main architectural style, providing a flexible communication mechanism for the entire system. This style, also known as implicit invocation, allows various components within the system to communicate and interact independently of each other. Kodi's core components do not directly invoke a procedure of others but are notified by announced events. For instance, rather than commanding the interface component to update directly, the player core announces an event when the media playback state changes. Also, the architecture allows for dynamic component interaction. New plug-ins can be added to Kodi at any time without changing other parts of the existing system, which gives Kodi great flexibility.

After understanding the main architectural style of Kodi, the Publish/Subscribe (Pub-Sub) style, we will further explore the top-level concrete subsystems of Kodi and how these subsystems interact and collaborate.

Kodi: As the center of the entire application, it is responsible for coordinating the features of various subsystems. It controls not just how the user interface appears and responds to operations from the user, but also core tasks like playing back media, configuring settings, and maintaining the system. All operations initiated through the user interface, add-ons, or network services require processing and forwarding by the Kodi core to ensure overall collaboration and data consistency of the system.

Add-ons: The add-ons subsystem provides the software with extremely high extensibility, allowing developers and users to customize and extend its features. Users can add new games, video channels, and even interface skins to Kodi by installing different add-ons. These add-ons can be updated independently without affecting Kodi's core functionality while providing users with a personalized experience.

GUI Element: The top-level subsystems responsible for the user interface (UI) in the Kodi concrete architecture. It provides a visual interface that enables users to interact with Kodi, such as browsing the media library, selecting media to play, etc.

Library: By automatically categorizing users' multimedia content, the media library enhances users' browsing and search experiences by making it easier for them to locate the content they need.

Player Core: This component is the core of all media playback features. Multimedia content is played, controlled, and managed by this subsystem. It can decode a variety of audio and video formats and process audio and video output, which provides a smooth and high-quality playback experience to the app users.

Source: This subsystem manages and organizes media sources within the application. In addition to reading local files, it enables network-based access to media sources and streaming devices to play multimedia content. Its main purpose is mainly to allow users to add or access different content sources, this component must ensure the availability and security of the data source while handling possible data synchronization and updates.

User Input Module: The user input module subsystem interacts directly with the user, converting user operations into commands that the system can recognize. It supports a variety of input devices, including remote controls, mouse, and keyboards, then will provide instant feedback on user input. Furthermore, it handles external signal interference and abnormal input to ensure input accuracy and system stability.

Web Server: The web server component enables users to manage their content remotely through a web browser or other applications. Web servers also need to ensure the security of network communications and can use encryption protocols and authentication mechanisms to prevent unauthorized access.

Window Manager: The Window Manager controls the visual layout and window behavior of the Kodi user interface. It is responsible for handling window opening, closing, minimizing, and other interactive actions. It needs to provide intuitive user navigation and clear visual feedback while ensuring the UI is responsive and smooth.

Platform: As a cross-platform media center, Kodi needs to ensure that it can run on different hardware and operating systems. This will allow Kodi to reach a wider user base.

Event Manager: The event manager is responsible for receiving events from various components of Kodi (such as user interface, player core, add-ons, etc.) and distributing these events to components that subscribe to the corresponding events. The event manager supports asynchronous messaging, ensuring Kodi remains efficient and responsive when multitasking, such as media playback and downloading content simultaneously.

Storage: The storage component is responsible for the permanence of all data in Kodi. This includes user configurations, playlists, and cached media files. This component is designed to balance data security with efficient access and rapid response when users access or modify data. Additionally, to address potential system failures, the storage component is equipped with backup and data recovery functionalities, guaranteeing that the system can be restored to its previous state in the event of an error, thus safeguarding user data from loss.

5.0 Chosen 2nd-level Subsystem

"DVDPlayer" is the designation for the internal media player employed for playing DVDs within Kodi. Kodi, an open-source media center, empowers users to organize and enjoy their personal media collection, encompassing videos, music, and images. Functioning as a specialized component within Kodi, DVDPlayer is tailored to manage DVD content, including menus, navigation, and playback. Its integration into the Kodi interface ensures a cohesive and smooth user experience.

DVDPlayer has traditionally used an object-oriented design (OOD) style. Object-oriented design is a programming paradigm that employs objects—instances of classes that encapsulate data and methods—to structure software. It encourages modularity, reusability, and maintainability, making it suited for complicated systems such as media players.

The object-oriented style allows for the encapsulation of functionality within objects, fostering a modular and adaptable architecture. Each class or object in the system represents a different entity or component, and interactions between objects assist the desired behavior.

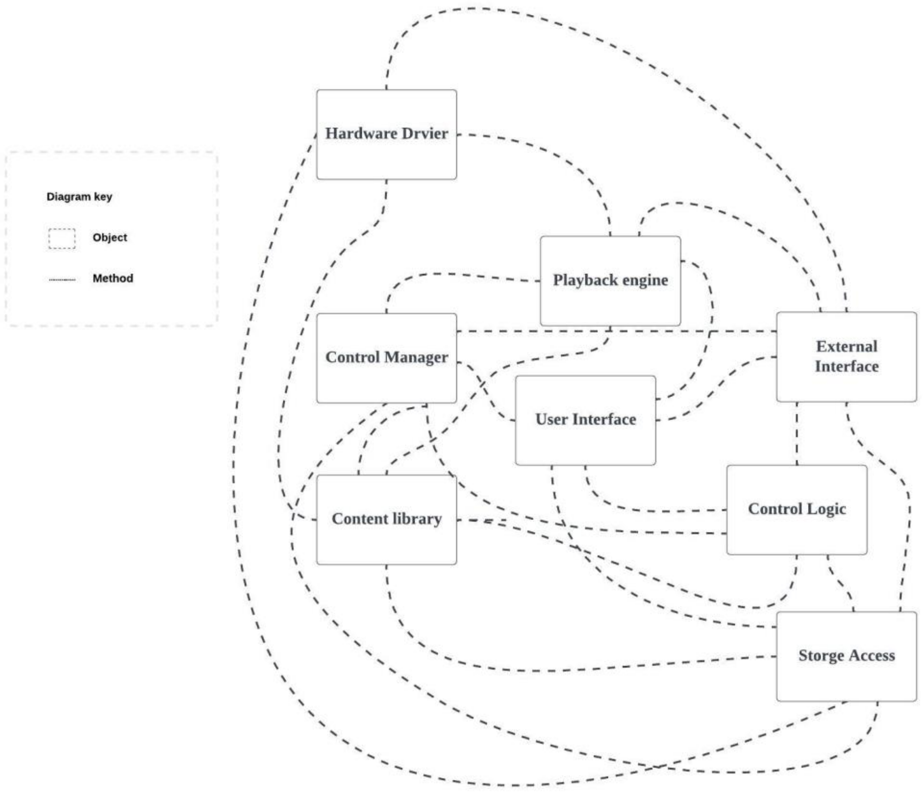


***Figure 5**: Conceptual Architecture of DVDPlayer Subsystem*

Concrete View

The concrete architecture of the DVDplayer subsystem in KODI is illustrated in Figure 4. The DVDplayer module begins its operation by collecting data from various sources, including （but not limited to） the DVD Storage Engine, Content Library, and Input Manager. Unlike a one-time data intake, the DVDplayer continuously retrieves and processes data as it performs its functions. The detailed data flow and interdependencies within the DVDplayer module are depicted in Figure 3, providing a comprehensive view of how the internal components interact and rely on each other during the playback and control processes.

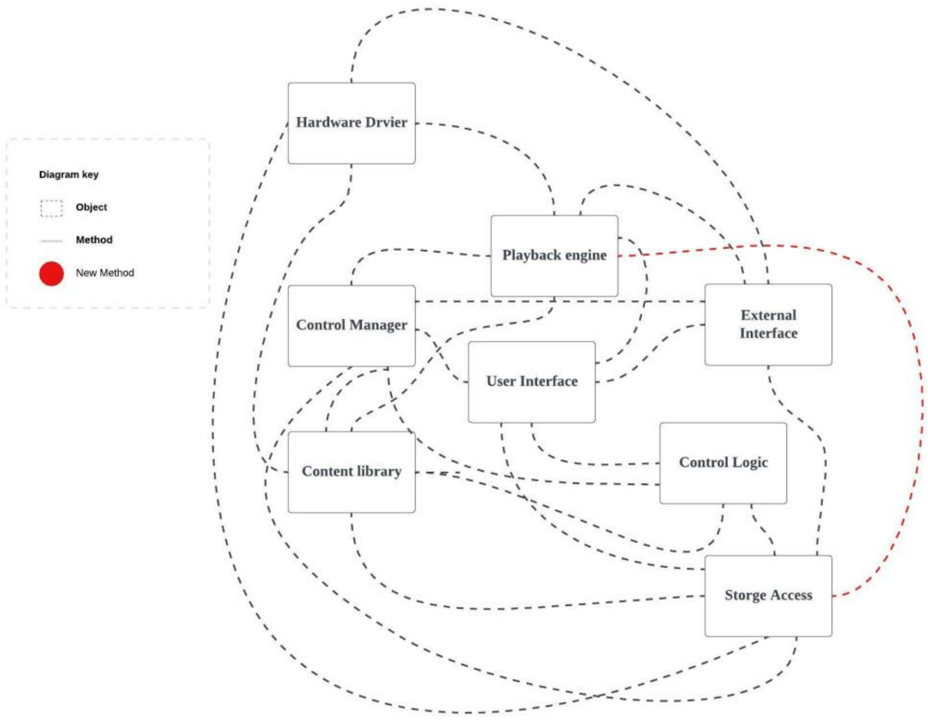


Figure 6: *Concrete Architecture of DVDPlayer Subsystem*

Given that the External Interface and Control Logic modules serve distinct functions and operate independently, eliminating the dependency may improve system flexibility and maintainability. Changes in one module do not affect the other because of loose coupling.
A direct reliance may not be required if the External Interface does not directly rely on Control Logic for its functionality or if Control Logic does not directly require inputs from the External Interface. Eliminating unneeded dependencies accords with the modularity principle, making each module easier to comprehend, alter, and maintain in isolation.

The dependence line between the playback Engine and Storage Access is critical for enabling effective DVD content retrieval during playing. It ensures dynamic management of content, adaptability to varied storage configurations, and optimal data flow, all of which contribute to a flawless and user-friendly DVD playback experience within the Kodi DVDPlayer Subsystem.

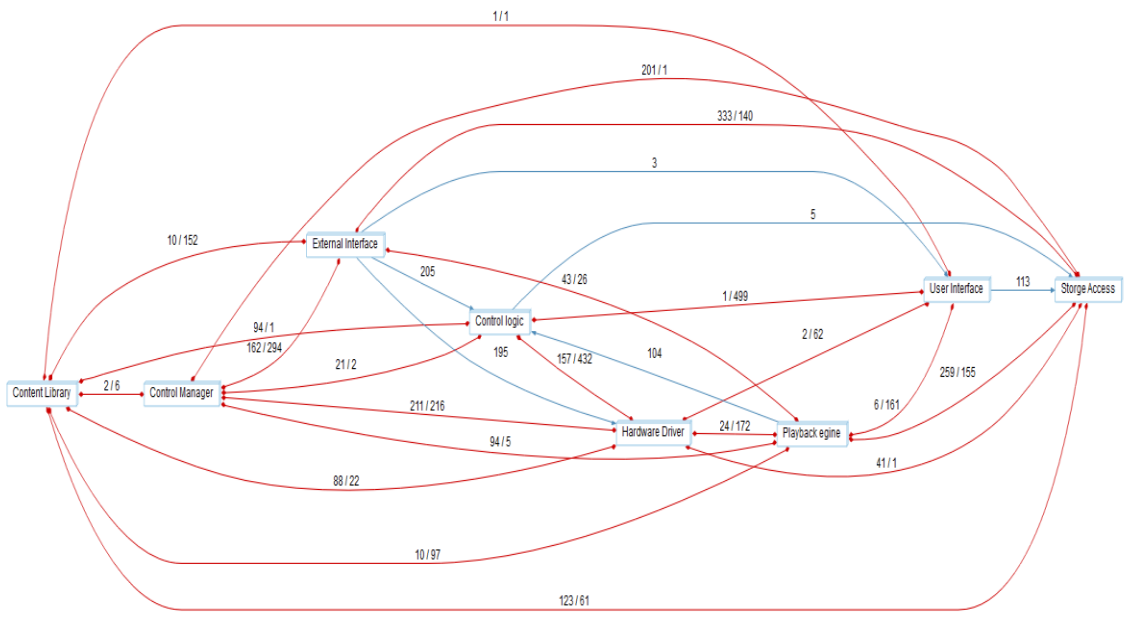


***Figure 7**: Understand Dependency Diagram of DVDPlayer Subsystem*

The "Storage Access" module is critical in the context of Kodi's DVDPlayer subsystem since it serves as the gateway for the subsystem to communicate with storage devices and retrieve crucial data for DVD playback. Its principal responsibility is to manage the interaction between the subsystem and storage devices, as well as to ensure access to DVD content, metadata, and other important details required for flawless playing. This module allows users to engage with the system by allowing them to browse their DVD collections or digital media libraries and reply to requests for specific content playback. Storage Access is also important in maintaining data integrity, adjusting to diverse storage configurations and media kinds, improving performance for efficient data retrieval, and connecting with external devices or services. Storage Access, in essence, is a key component of the DVDPlayer subsystem, delivering a smooth and adaptive DVD playback experience within the Kodi media center. A parallel organizational structure emerges in the world of Kodi's DVDPlayer subsystem, consisting of interdependent modules that collectively form a cohesive network. Content Library, Hardware Driver, Playback Engine, Control Manager, User Interface, External Interface, Control Logic, and Storage Access work together to produce a successful DVD playback environment. The User Interface module, stands out with its various dependencies, serving as a central point for user interaction. The Hardware Driver, in particular, is critical in establishing communication between the OS and the hardware components required for DVD playback. The Control Manager, orchestrates processes among several nodes. Storage Access, an important component of the DVDPlayer subsystem, manages interactions between the system and storage devices, ensuring that DVD content is retrieved seamlessly. The whole architecture, has peer-to-peer characteristics, with each module interconnected via dependencies, enabling a collaborative and efficient DVD playback experience within the Kodi media center.

6.0 Reflextion Analysis

6.1 Reflextion Analysis -The high-level architecture

We add a new node called Event manager, which mainly focuses on processing and scheduling the threads. Event Manager is a component in Kodi that is responsible for accepting remote device input on all platforms. It simplifies interfacing input devices with Kodi and accepts commands from event clients such as LIRC, joysticks, PDAs, and iPhones. The "EventServer API" is used to program event clients and provides a simple, reliable way to communicate with and control Kodi. The Event Manager folder contains the threads subfolder and some I/O processing files.

The Storage Node is added to represent the interface and mechanisms through which Kodi accesses and manages media content stored on various storage devices.
Including a dedicated Storage Node recognizes the significance of seamless media retrieval from local drives, network-attached storage, or streaming sources. It highlights the importance of handling different storage scenarios for a comprehensive media center solution.

What's more, the concrete architecture has a Platform as one of the top-level subsystems. To enhance the variety of users, this component has different programs to handle different systems which users prefer, like Android, Linux, win64, win32. Here are some main unexcepted dependency after update the concrete architecture compared with the original conceptual architecture:

Event Manager => User Input Module

The software's ability to process user input is facilitated by an event management component, which is designed to efficiently and effectively manage the flow of events and data within the software. This component is responsible for coordinating and synchronizing the various asynchronous and parallel processes that are involved in processing user input, such as event handlers, callbacks, and promises. In the code level, the dependency browser shows that the event manager has a dependency on <InputManager.h> and sub architecture like actions, mouse, even touch in User Input Module. In the conceptual level, when a user interacts with Kodi through an input device, the User Input Module captures the input and generates corresponding events based on the user's actions. These input-generated events are then passed to the Event Manager, which is responsible for distributing these events to the appropriate event listeners or handlers registered within Kodi.

GUI Elements <=> User Input Module

GUI elements in Kodi represent the visual components of the user interface, including menus, buttons, sliders, and other interactive elements. These elements are designed to provide a user-friendly way for users to navigate, select options, and control various features of the media center, while the responsibility of User Input Module has illustrated above. In the code level, the dependency browser shows that the GUI elements have dependency on <InputCodingTable.h>, <Action.h> and <ActionTranslater.h>. In the conceptual level, when a user interacts with an input device, such as pressing a remote-control button or navigating through a keyboard, the User Input Module captures the raw input data. The User Input Module processes the raw input data and translates it into meaningful input events. For instance, a button press on a remote control is translated into a "select" event. The generated input events are then propagated to the relevant GUI elements within Kodi. Each GUI element has event listeners that are designed to respond to specific types of input events. When an input event reaches a GUI element, the associated event listener for that element is triggered.

Player Core => Platform

A component that integrates sound renderer, video renderer, DVDplayer, and other important functions， Player Core has some dependency on platform. In the code level, the dependency browser shows that <DVDVideoCodecAndroidMediaCodec.cpp> Includes <JNIXBMCSurfaceTextureOnFrameAvailableListener.h> at <DVDVideoCodecAndroidMediaCodec.cpp>:35. Through the file name, we can assert that different platforms have different performance on playing video. In the conceptual level, The Player Core relies on platform-specific mechanisms for handling media-related tasks, such as accessing file systems, managing network connections, and interacting with hardware for audio and video output. What’s more, different platforms provide different Codec support, the platform plays a crucial role in providing support for various codecs and multimedia libraries. Player Core relies on platform-specific capabilities to decode and process media content using the available codecs.

6.2 Reflextion Analysis - The architecture of the chosen 2nd level subsystem

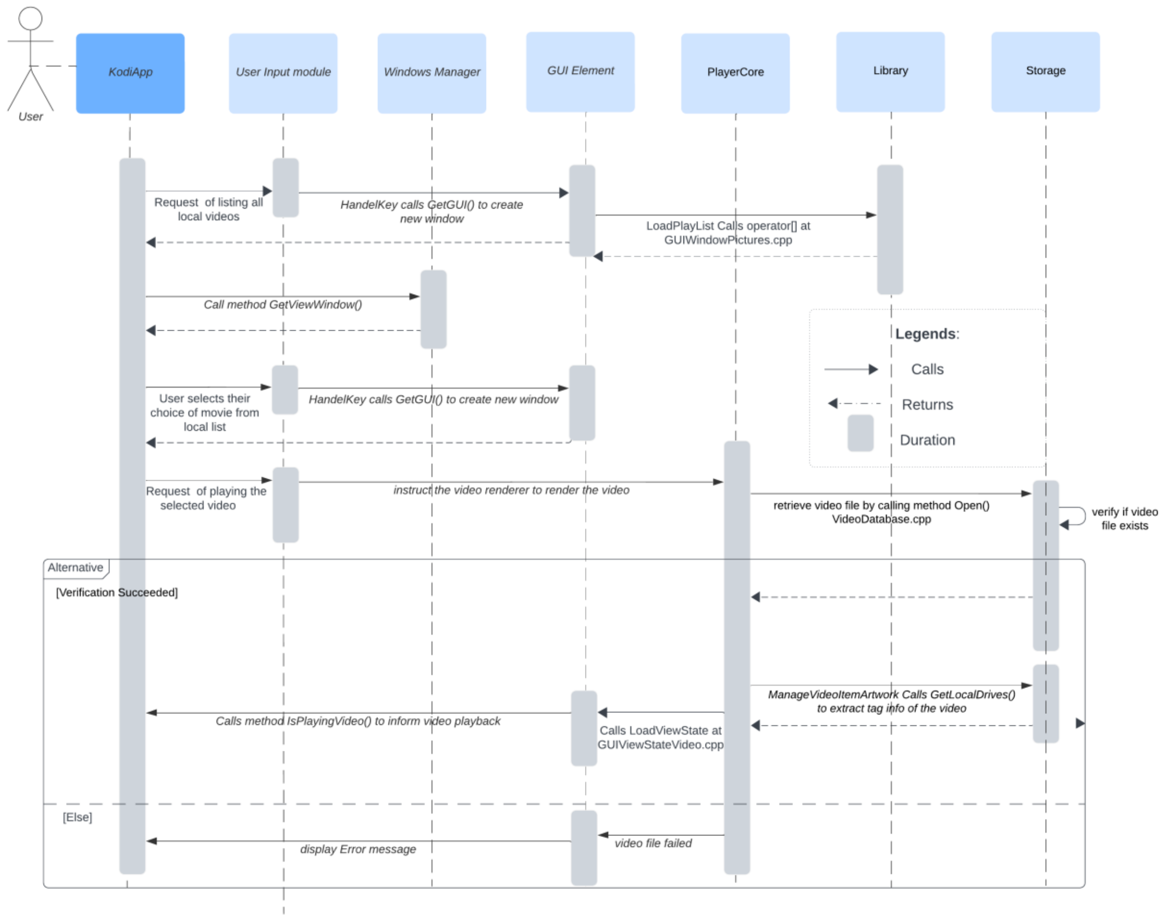
We have chosen the DVDPlayer as our second level subsystem. It has traditionally used an object-oriented design (OOD) style. The modulaty is one key feature to analysis. The divergences in the DVDPlayer subsystem architecture with separate nodes for Playback Engine and Storage Access are driven by considerations of modularity, separation of concerns, adaptability to various storage sources, and support for different media playback scenarios beyond DVDs. This design choice enhances the overall flexibility and efficiency of the Kodi media center when handling DVD playback.

Playback engine <=> Storage Access

The Playback Engine within DVDPlayer is the module that manages the entire playback process. It orchestrates the retrieval, decoding, and rendering of multimedia content, While Storage Access refers to the mechanisms and interfaces through which the DVDPlayer component accesses and retrieves media content stored on storage devices, including local drives, network-attached storage, or streaming sources. In code level, the Playback Engine has dependency on Storage Access by <MusicDatabase.cpp> Includes <MediaManager.h> at <MusicDatabase.cpp>:54. However, Storage Access has dependency on Playback Engine by <MusicDatabaseFile.cpp> Includes <MusicDatabase.h> at <MusicDatabaseFile.cpp>:12. In the conceptual level, The Playback Engine, within the DVDPlayer, is heavily dependent on Storage Access to retrieve media content. This can include reading video files, accessing DVD directories, or streaming content from network sources. In addition, Storage Access helps the Playback Engine locate and open media files, manage file pointers during playback, and handle seek operations. Storage Access also plays a role in buffering and caching mechanisms, which allows the Playback Engine to efficiently read and cache portions of the media content for smooth playback. The dependency on Storage Access influences how the Playback Engine optimizes media retrieval to ensure smooth and efficient playback, taking into consideration factors like data access speed and latency.

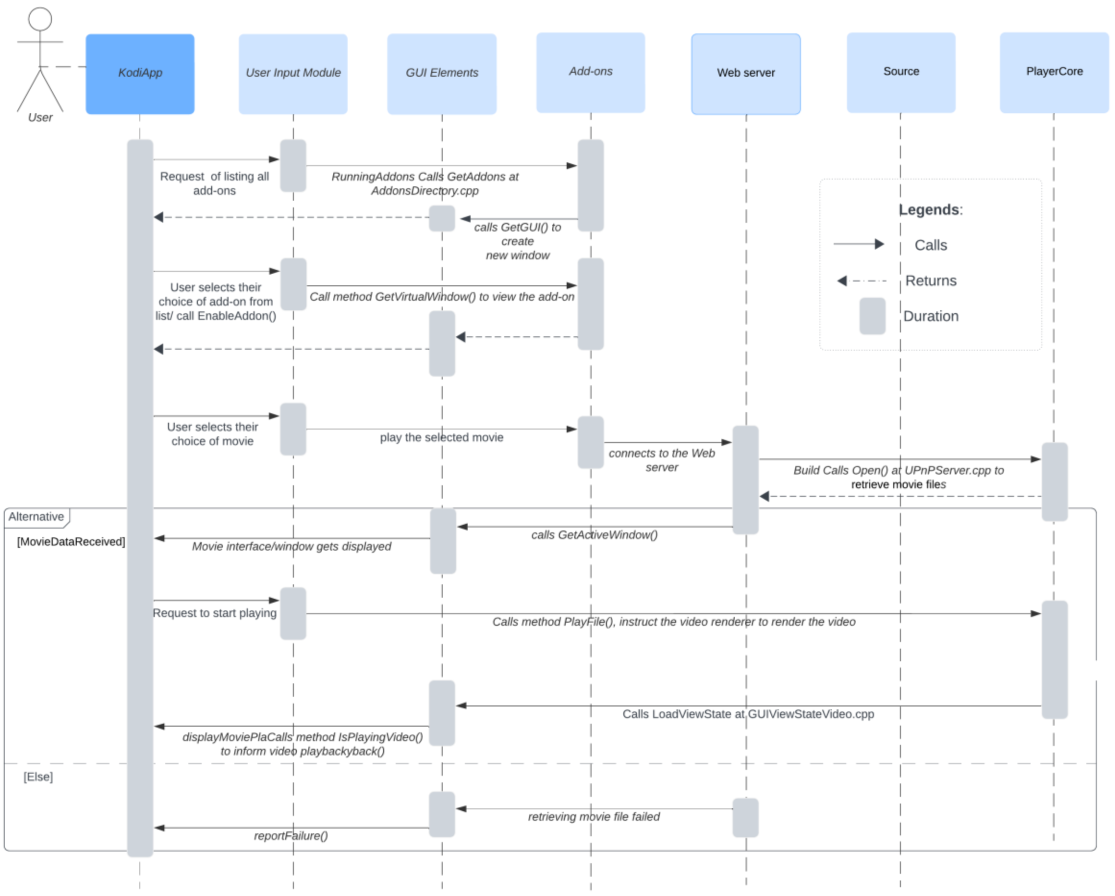
7.0 Use Cases

7.1 Use case 1


User
KodiApp
User Input module
Windows Manager
GUI Element
PlayerCore
Library
Storage
Request of listing all local videos
HandelKey calls GetGUI() to create new window
LoadPlayList Calls operator[] at GUIWindowPictures.cpp
Call method GetViewWindow()
Legends:
Calls
Returns
Duration
User selects their choice of movie from local list
HandelKey calls GetGUI() to create new window
Request of playing the selected video
instruct the video renderer to render the video
retrieve video file by calling method Open() VideoDatabase.cpp
verify if video file exists
Alternative
[Verification Succeeded]
ManageVideoItemArtwork Calls GetLocalDrives() to extract tag info of the video
Calls method IsPlayingVideo() to inform video playback
Calls LoadViewState at GUIViewStateVideo.cpp
[Else]
display Error message
video file failed

When a user chooses to play a local video, the Kodi application is initially launched. Subsequently, when the user requests to list all local videos, the User Input Module captures this action. The 'HandleKey()' method is then invoked, calling the 'GetGUI()' method to create a new window through the GUI component. A list of local videos is displayed to the user, providing them with the option to select a video of their choice. Again, the 'GetGUI()' method is employed to create the video interface, followed by retrieving the video file from the local storage through the 'Open()' method. Upon successful verification of the video file, it is fetched to the Player Core. The Player Core extracts tag information from the storage and proceeds to load the view through the GUI. Finally, the GUI invokes the 'IsPlayingVideo' method to notify the user of the playback status. Now, the user can utilize the Player Core to control the playback behavior of the video based on their preferences and needs.

7.2 Use case 2


User
KodiApp
User Input Module
GUI Elements
Add-ons
Web server
Source
PlayerCore
Legends:
Calls
Returns
Duration
Request of listing all add-ons
RunningAddons Calls GetAddons at AddonsDirectory.cpp
calls GetGUI() to create new window
User selects their choice of add-on from list/ call EnableAddon()
Call method GetVirtualWindow() to view the add-on
User selects their choice of movie
play the selected movie
connects to the Web server
Build Calls Open() at UPnPServer.cpp to retrieve movie files
Alternative
[MovieDataReceived]
calls GetActiveWindow()
Movie interface/window gets displayed
Request to start playing
Calls method PlayFile(), instruct the video renderer to render the video
Calls LoadViewState at GUIViewStateVideo.cpp
displayMoviePlaCalls method IsPlayingVideo() to inform video playbackyback()
[Else]
retrieving movie file failed
reportFailure()

When a user decides to watch an online movie, the Kodi application is initiated, followed by a request to list all installed add-ons using the 'GetAddons()' method. The GUI components create a window that enumerates all available add-ons through the invocation of the 'GetGUI' method. Users can now utilize the GUI to select their preferred add-on, such as Netflix or YouTube. The Kodi app triggers its method to enable the chosen add-on. Through the add-on's view, all its available movies are showcased, enabling users to make a selection. Subsequently, the Kodi application establishes a connection to the online streaming service via Add-ons and the Web Server. The Web server, in turn, calls the 'Build' method, which invokes the 'Open()' method. The 'Open()' method fetches all movie files to the Player Core, where the files are decoded and played by calling the 'PlayFile' method. When the user presses start, the movie view is displayed by the GUI, and video playback status is communicated to the user. These processes ensure that users can seamlessly enjoy watching online movies.

8.0 Lessons Learned

When we opened Understand to analyze the Kodi code, we uncovered many insights that had not been explored before. The Kodi codebase is remarkably extensive, with highly complex dependencies in each subsystem. To achieve a more refined concrete architecture, we conducted an in-depth analysis of the code in each top-level file. Based on the functionality of each file, we categorized and assigned them to different subsystems.

Through classification and analysis, we identified many dependencies that actually didn't exist in the abstract conceptualization. Furthermore, after conducting static analysis using Understand, we discovered numerous new mechanisms and functionalities.

Consequently, we added numerous new dependencies, removed erroneous ones, and recognized the need to introduce new subsystems for a more robust architectural design. More importantly, we found that, for better visualization in Understand, we eliminated some redundant subsystems from our original conceptual architecture, such as Python and font renderer. This helped us gain a clearer understanding of the dependencies between each subsystem and the overall structure of the architecture. This, in turn, allowed us to refine and modify our new conceptual architecture.

Additionally, we realized that effective teamwork would play a significant role in the project's development. Throughout the team project, we adopted a division of labor, with each member responsible for analyzing multiple top-level files. The analyzed files were then integrated into our architectural subsystems. This division of labor significantly increased our efficiency. Through communication within the team, we clarified many dependencies between subsystems. Importantly, this collaborative approach enabled us to delve deeper into the complex Kodi codebase and adjust the architecture to better align with the actual implementation.

During the use of Understand, teamwork not only facilitated the identification of dependencies but also provided valuable insights for rebuilding the system to enhance performance and design consistency. This collaborative process proved instrumental in gaining a comprehensive understanding of Kodi's intricate codebase and ensuring alignment with the envisioned architecture.

Simultaneously, as we delved deeper into the reflective analysis of dependencies between second-level subsystems (DVDPlayer), we gained a more profound understanding of Kodi's architecture. We discovered that severe dependencies between subsystems could lead to mutual influences, such as slowing down the operation speed of other subsystems, resulting in high latency, and more. However, by reasonably scheduling processes, efficiency can be significantly improved.

9.0 Conclusions

In this report, our focus revolves around the dependencies among Kodi and its subsystems. While analyzing the conceptual framework to derive the concrete architecture, we identified the necessity for accurate representation of the actual conceptual architecture in the initial charts from our previous reports. Through scrutinizing developer documentation and employing Scitools Understand for graphical visualization, we successfully crafted a refined conceptual architecture and utilized it to deduce the concrete architecture. Reflective analysis brought to light new dependencies among various subsystems.

Ultimately, we introduced some new components, such as "Platform" and "Storage," based

on the existing conceptual framework. To eliminate certain dependencies used for testing and maintenance, we constructed a concrete architecture to clearly delineate the dependencies among components. Furthermore, we delved into the specific subsystems of Kodi, such as DVDPlayer, and constructed a high-level conceptual architecture for them.
In summary, through meticulous observation and analysis of the Kodi architecture, coupled with the assistance of Scitools Understand, we have gained a deeper understanding of the system and its underlying architecture. This progress in our research on Kodi's software architecture is immensely valuable. Our work not only unravels the inherent complexity of the existing architecture but also lays a robust foundation for future conceptual architecture optimization and subsystem functionality research.

10.0 Data Dictionary

W4 Approach - when, which, why, who

Subsystem - A smaller, independent component within a larger system

GUI - A graphical user interface, or GUI, is a user interface design that allows people to interact with electronic devices through the use of graphical icons and auditory signals.

GUILib - GUILib is a library that allows for the simple building of GUI Menus in a software.

Peripherals - Peripherals enhance the functionality of a computer by providing input, output, storage, and communication capabilities.

PAPlayer - (or PAP for short) stands for Psycho-acoustic Audio Player

Event Manager - A software component of Kodi responsible for accepting remote device input on all platforms.

Dependency Browser - A window shows the the relationship of dependency of two components in Understand.

11.0 Naming Conventions

KISSFFT - which stands for "Keep It Simple, Stupid Fast Fourier Transform," is an open-source FFT library.

12.0 References

Kodi Community. "Archive:PAPlayer." Kodi Wiki, 20 July 2020, Archive:PAPlayer. Accessed

<https://kodi.wiki/view/Archive:PAPlayer>.
Kodi Wiki. Official Kodi Wiki. (2023). https://kodi.wiki/view/Main_Page
News. Kodi. (n.d.). https://kodi.tv/blog/
Paul, R. (2009, December 29). *XBMC 9.11 makes your open source home theater look shinier*.
Development. Official Kodi Wiki. (n.d.) https://kodi.wiki/view/Development
Dreef, K., Reek, M. van der, Schaper, K., & Steinfort, M. (2015, April 23). *Architecting software*
to keep the lazy ones on the couch. Kodi. https://delftswa.github.io/chapters/kodi/