Kodi Concrete Architecture

CISC 322

Assignment 2 Report

Sunday, November 19, 2023

Group 8 : KidKodi

Aidan Gardner - (20agg2@queensu.ca)

Gavin Chin - (21gc9@queensu.ca)

Daniel Garami - (21deg1@queensu.ca)

Barkev Keyork Sarkis - (19bks5@queensu.ca)

Mahir Khandokar - (21mhk5@queensu.ca)

Adrian Putz-Preyra - (16app2@queensu.ca)

Table of Contents:

Abstract

This report delves into a comprehensive analysis of Kodi's concrete architecture, highlighting the relationships and differences between the conceptual and concrete architectures. Utilizing the SciTools Understand software for architecture derivation, the investigation we perform reveals the top-level components and subsystem architectures within the actual Kodi source code, emphasizing the significance of reflection analysis to compare and contrast conceptual blueprints with the actual implementation. The lessons learned underline the importance of flexibility, dependency diagrams, and an understanding of subsystem interactions. Limitations, including potential inaccuracies and solely relying on static system analysis, are additionally acknowledged. Overall, the report provides a comprehensive understanding of the architectural intricacies of how Kodi is actually formed.

1. Introduction and Overview

The conceptual architecture of a software system is developed on a simplistic overview of its connecting subsystems, while the concrete architecture of that system represents the actual implementation of technology. Think about the blueprints of a house that provide the conceptual layout to be implemented. This is conceptual architecture. Now picture the fully built house. How is it similar or different from the blueprints? This is the concrete architecture and the focus of this paper. Deriving a concrete architecture is a more complex process in comparison to the derivation of a conceptual architecture, as it involves accessing the source code and extensive dependency analysis. By using a tool like Understand, it becomes possible to graph dependencies and code relationships to construct a diagram that reflects the concrete architecture of a system.

This report aims to compare the derived concrete architecture of Kodi with the previously proposed conceptual architecture. While the conceptual architecture offers a top-level view of components working within the system, the concrete architecture provides insight into the actual implementation. When comparing the two architectures, discrepancies often arise and can be seen as additional modules and functions are introduced, resulting in variations from the blueprint of the conceptual architecture to the actual structure of the concrete architecture. As a result, reflection analysis will be employed to scrutinize these differences and delve into the rationale behind the changes.

Additionally, the internal architecture of the Player Core Module will be explored in detail, comparing both its conceptual and concrete view, as well as analyzation through the use of diagrams. Finally, we will conclude our analysis with a summary of our findings.

2. Architectures Considered

Different architectural options were thoroughly examined to address challenges in the architecture transition and dependencies analysis. Among these, the layered architecture, described by its modular structure emerged as the most suitable choice for Kodi. In contrast,

pipe-and-filter and object-oriented styles were not found to be as viable of options due to the vast amount of two-way dependencies and code structure. With a focus of shared resources appearing through our analysis of the code, a repository style seemed like an option to consider, however when we think about the expenses associated with development and maintenance, it did not seem as practical for an open-source project like Kodi. Implicit-invocation was also considered as it aligns with Kodi's requirements particularly for add-ons and presentation purposes, however we ultimately decided to stick with our proposed design.

3. Derivation Process

The concrete architecture of Kodi was derived using the SciTools Understand software, which allows for the importation of source code, visual inspection of file dependencies, and creation of architectures to examine the overall relationships of subsystems. Using the modular and layered conceptual architecture that our group derived from previous analysis, we modeled this structure within Understand. This architecture consisted of four main layers, featuring Client, Presentation, Business, and Data layers, with each layer having its own set of roles, functionalities, and sub-components within the system. Having set out our conceptual architecture in Understand, we then sifted through the xbmc source code to map the actual files and directories to their conceptual locations. This process was done based on file names or the code and comments itself. As we went about this process, we noticed that many files and directories did not completely line up with our proposed conceptual architecture, and as a result, we needed to undergo slight modification and create or remove subsystems to best match the actual source code. After placing all the files within their respective subsystems, a dependency diagram was generated to allow us to recover the concrete architecture of Kodi. We then performed a reflection analysis and examined any new or unexpected dependencies between the conceptual and concrete architecture. Upon analysis completion, we then reflected on whether each unexpected dependency was due to a file being placed in the wrong component or a divergence from the conceptual architecture and moved files around until we felt that each file was indeed in the most appropriate subsystem.

4. Top-Level Architecture

4.1 Conceptual Architecture

Our original proposed architecture for Kodi embodied a layered architectural style. In this architecture, the system is built up of multiple competent layers that can interact with each other. Despite being a layered architecture, layers are not limited to only interacting with adjacent layers; rather each layer was considered independent from each other, enabling many interactions between different components. This was derived using the Kodi Wiki and development information [2]. According to the developers, the Kodi design process is to be highly modular and should still compile if a non-essential module is removed. As a result, our conceptual architecture was split into four main layers, each with different functionalities. These top-level

subsystems include the Client Layer, Presentation Layer, Business Layer, and Data Layer. The Client Layer is related to the specific platform or system that is being used to run the application. The Presentation Layer is related to the user interface and overall display of the application. The Business Layer contains files and packages related to the actual functionality of Kodi, and is where the main event handling takes place. Lastly, the Data Layer contains anything in reference to files, including saving, storing, streaming, and transforming data.

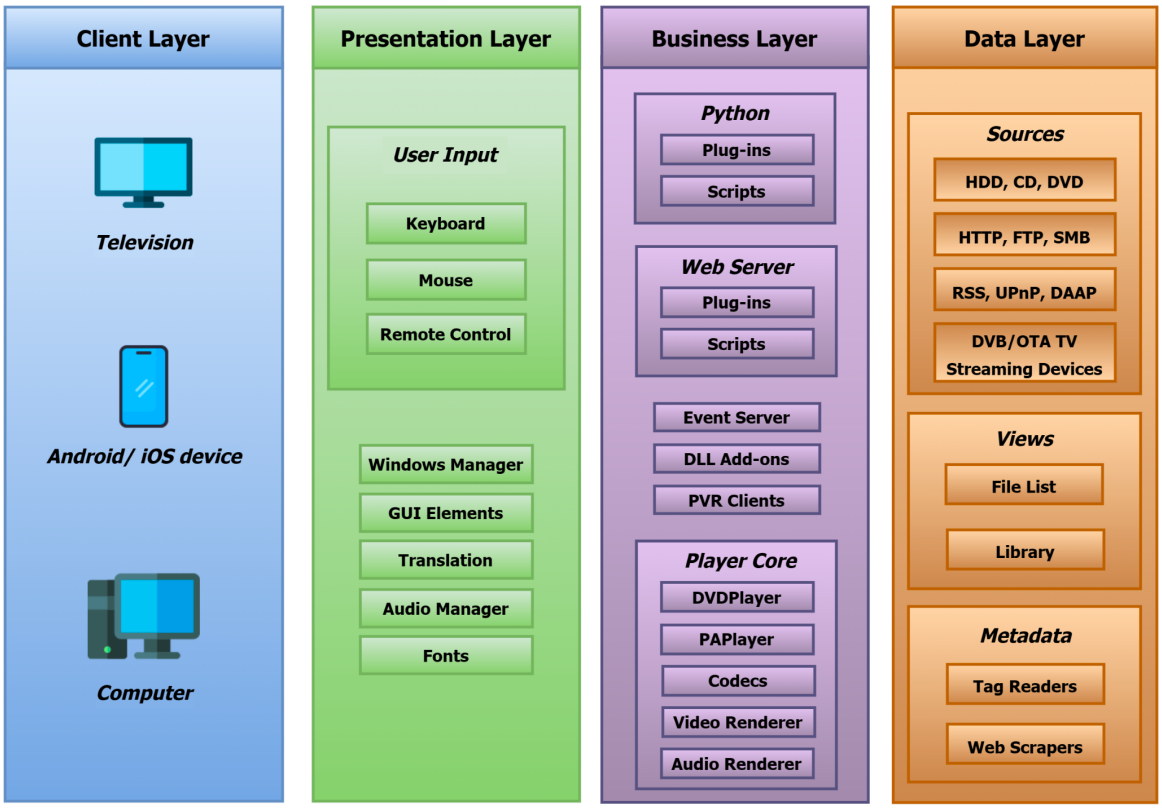

Figure 1: Conceptual Architectural Overview of Kodi

4.2 Concrete Architecture

The concrete architecture of Kodi follows a layered architecture style similar to the proposed conceptual architecture. As hypothesized before, the system is made up of layers of components that interact with other components from other subsystems. Although the base layers of the conceptual and concrete architecture are generally the same, there are a few differences when looking deeper into 2nd-level subsystems within the layers. Starting with the Client Layer, it remains consistent with the proposed conceptual architecture, holding information regarding the platforms and operating systems, as well as a new subsystem relating to power management. A majority of its interactions are dependencies on common utility files for the platform, or in mutual relation to the Presentation and Business Layer with regards to the operating system. The Presentation Layer was again decently similar to its blueprint, consisting of a User Input Module, Windows Manager, GUI Management, and Playback Settings that revolved around visual

elements and customizable settings. This is slightly different from the conceptual architecture in the sense that there did not seem to be individual subsystems relating to fonts, translation, or audio, but rather a distribution of files among different directories. In this layer, interactions between all other layers are about even, except for the Business Layer in which the Presentation Layer is highly dependent on. This makes sense since the Business Layer is directly influencing and updating Kodi. In terms of the Business Layer, the Player Core was easily identifiable within the xbmc source code and had a very similar makeup when compared to its representation in Figure 1. Additionally, add-on packages, interfaces, PVR Clients, and DLL related files can all be found in this layer. In terms of the interactions of this layer, it was again seen interacting with all layers, with a heavy focus on utility and logging, and communication with the Presentation Layer. Since this layer is primarily focused on functionality, the use of logging can help to identify errors and bugs. Finally, we look at the Data Layer and can see that it does not really fit with our more modular design. Instead, we can see that it can be generally categorized into Views and Sources. Views involve the storage of data and consist of the *filesystem*, *playlists*, and *storage* directories. Sources involve anything related to the network and include the *cdrip* and *network* directories. In what is a common theme being seen, the Data Layer interacts with all other layers, focused mainly on the storage of media and ensuring proper file formats based on settings. A new subsystem not covered in the conceptual architecture was the use of common libraries that do not relate to a specific task within a layer. Each of the libraries in this subsystem helps keep the code modular and promotes code reuse, contributing to the overall functionality and maintainability of the Kodi software. This includes directories like *commons*, *events*, *utils*, and *test* that could be used generally. Regarding some patterns within the design, most tasks consist of a .h and .cpp file that work in tandem. Within these files, instead of hard-coded relations, the use of include statements help to support modularity and reuse, as well as future support for change.

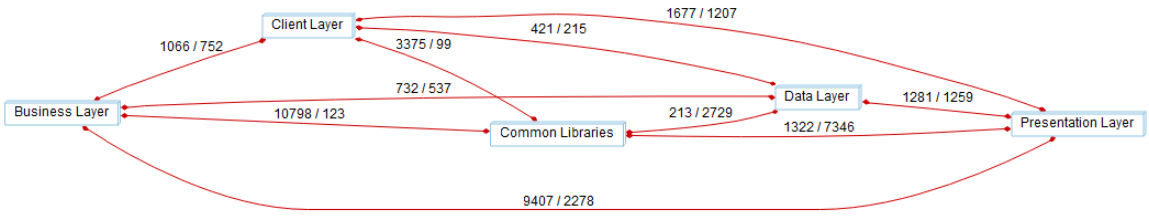

Figure 2: Dependency Graph of the Concrete Architecture of Kodi

4.3 New Top-Level Subsystems

The Common Libraries top-level subsystem is an addition to our previously proposed conceptual architecture. This subsystem consists of important components that many different parts of the system use to perform common and frequent tasks. The several libraries within this

subsystem include *commons*, *contrib*, *dbwrappers*, *events*, *test*, and *utils*. The *commons* subdirectory contains common functions and classes that are used across various parts of the codebase. *Contrib* serves as a collection of external libraries and modules that are not part of the core Kodi development, but provide extended functionality to the system. *Dbwrappers* contain database wrapper classes that are responsible for abstracting and simplifying interactions with databases, making it easier for other components of the system to store and retrieve information on media files or other relevant data. *Events* is a library that handles event management within the platform. This includes notifications or signals that other components can use to communicate with each other. The *test* library is used for testing purposes during software development. This directory includes tests that help ensure that various parts of the code are functioning as intended. Lastly, *utils* contain a variety of utility functions and tools that are used throughout the codebase that perform common tasks.

4.4 New/Unexpected Top-Level Dependencies

As shown in the concrete architecture there were no real unexpected dependencies of the existing subsystems from the original proposed conceptual architecture, however, with the addition of the new top-level subsystem (Commons Libraries), several new or unexpected dependencies occurred. From our conceptual architecture, we expected all of the layers to have interactions with each other so there were no outliers that jumped out at us. The Client Layer interacts mainly with the GUI settings, Player Core and network which are all to be expected. The Presentation Layer has a main dependency focus with the Player Core Module and addons which configure visual changes. The Business Layer has multiple expected dependencies across all layers, and again nothing really major stood out as new and unexpected. Lastly, the Data Layer holds two-way dependencies mainly with playback config and directory navigation. It has the least amount of dependencies with the Client Layer as we had again assumed. Below, we will take a look at some unexpected and new dependencies that can be found in our new architecture.

Player Core (Business) <-> windowing (Client)

Rationale: Dependencies from the windowing directory to the Player Core Module can be seen by graphic and resolution configuration between both subsystems to ensure media can be properly displayed according to specific platforms. Additionally, the Player Core contains subfolders with *ProcessInfo* files for each operating system, which demonstrates Kodi's versatility among platforms.

platform (Client) -> dbwrappers (Commons)

Rationale: The *platforms* submodule has a one-way dependency on the *dbwrappers* submodule. In the Kodi code, the *platforms* directory holds critical information in regards to the user platform. As a result, ensuring proper storage and retrieval of critical information is essential in minimizing issues. This dependency is essential as it signifies that the platform

component requires access to SQL and Database related protocols from *dbwrappers* for efficient database interaction and management within the architecture.

powermanagement (Client) <-> utils (Commons)

Rationale: The *powermanagement* submodule has a two-way dependency with the *utils* submodule. This relationship indicates a close association between power management functionalities and general-purpose utilities. The *powermanagement* submodule leverages utility functions from *utils* for efficient and standardized operations while also relaying standard logging information back. This allows Kodi to keep track of events depending on the different *PowerState* types from *PowerTypes.h*.

threads (Client) -> commons (Commons)

Rationale: The *threads* submodule has a dependency on the *commons* submodule and indicates that it relies on files like *Buffer.h*, *Exception.h*, and *ilog.h*. This makes sense since the *threads* directory relies on the overarching operating system functionality and must leverage these files to ensure correct processes are being executed and proper order is maintained.

events (Commons) <-> Playback Settings (Business)

Rationale: *events* and Playback Settings have a balanced two-way dependency, indicating a mutual influence on each other's functionalities within the Kodi architecture. In the concrete architecture, Playback Settings refers to any files related to the settings of multimedia, whether that be games, videos, music, or pictures. The interaction between the two subsystems suggests a mutual exchange of information, where *events* influence Playback Settings and vice versa.

Addons (Business) -> dbwrappers (Commons)

Rationale: *addons* has a one-way dependency on *dbwrappers* to indicate the need for efficient and standardized database interactions. The dependency ensures seamless integration of add-ons with the underlying database infrastructure, as well as easy addon lookup access.

PVR Clients (Business) -> dbwrappers (Commons)

Rationale: The PVR Clients submodule has a significant dependency on *dbwrappers* with a count of 813 connections. With PVR relating to video recording capabilities, it is crucial that these recorded videos can be stored properly and effectively. The dependency seen here ensures that this is possible.

Player Core Module (Business) <-> utils (Commons)

Rationale: The two-way dependency between the Player Core Module and *utils* is substantial. What this signifies is a strong reliance from the Player Core Module on utility functions and shared features provided by *utils*. This dependency supports Kodi's robustness, showing that operationality is not completely reliant from within the Player Core Module. This

allows other subsystems to use common utilities without major dependency on the Player Core, increasing concurrency across the system.

Sources (Data Layer) -> commons (Commons)

Rationale: The Sources submodule relies on the *commons* submodule mainly in regard to the logging properties within *ilog.h*. This dependency is mostly in relation to the *network* directory of the source code, where logging is crucial for maintaining a proper connection. This dependency ensures efficient and reliable operations related to data handling and processing across the network.

Views (Data Layer) -> commons (Commons)

Rationale: The Views submodule is quite similar to the Sources submodule in terms of the fact that it has a significant dependency on the *commons* submodule. Again we can see that logging is key, however, there are also increased dependencies from the *filesystem* directory on the *Exception.h* file in addition. This dependency is key to catching unchecked exceptions to ensure files are properly stored to avoid accidents or corruption.

5. Second Level Subsystem Analysis (Player Core Module)

5.1 Conceptual Architecture

The Player Core Module manages the overall playback functionality. It serves as the central component coordinating interactions between various submodules to ensure seamless media playback. The Player Core Module consists of several subsystems including a DVD player, PAPlayer, Codecs, Video Renderer, and Audio Renderer.

Figure 3: Conceptual Architecture of the Player Core Module

5.2 Concrete Architecture

After deriving the concrete architecture, it remains clear that the Player Core Module is a critical component responsible for managing the media playback process. The concrete architecture of the Player Core Module also consists of several subsystems, each of which play an important role in the media playback pipeline.

5.3 New Subsystems

External Player: Integrates with external players to handle specific media formats or codecs that are not natively supported by the internal player components.

Audio Player: Manages audio playback functionality, including decoding audio streams, volume control, and synchronization with the video player. It interfaces with the rendering component to produce audio output.

Video Player: Handles video playback, including decoding video streams, rendering video frames, and managing playback settings. It interfaces with the audio player to ensure synchronized multimedia playback.

Retro Player: Specialized player for retro gaming, providing support for classic gaming formats and consoles. It interfaces with the rendering components to display the retro game graphics and manages input for gaming controls.

Rendering: Manages the rendering of audio and video output, as well as graphics on associated games. This component ensures that audio is played at the correct timing and that video frames are being displayed smoothly and seamlessly synchronized with audio. It interfaces with the audio and video players to coordinate rendering.

Player Core General: Houses general functionalities and interfaces shared across different player components. This includes common utilities, configuration settings, and event handling that contribute to the overall consistency and modularity of the Player Core Module.

5.4 New/Unexepected Dependencies

Several new or unexpected dependencies occurred while analyzing the concrete architecture of the Player Core Module.

RetroPlayer (Player Core Module) -> Views (Data Layer)

Rationale: *RetroPlayer* has a dependency on Views within the Data Layer. Within the Player Core Module, *RetroPlayer* is a media player similar to the audio and video players however it relates to gaming instead of music and movies. Within the *filesytem* directory of the Views, we can see files such as *CurlFile.cpp* and *CurlFile.h*, which allows for the fetching of files from the internet. This enables the *RetroPlayer* to fetch files from online game repositories.

Audio (Player Core Module) -> utils (Common Libraries)

Rationale: Audio has a dependency on *utils*, specifically heavy dependencies on files such as *log.h* and *actorprotocol.h*, which provide logging functionality for tracking and debugging audio-related events, and communication protocols between different components of the Player Core Module respectively.

VideoPlayer (Player Core Module) -> utils (Common Libraries)

Rationale: *VideoPlayer* has a significant dependency on *utils* in the Common Libraries subsystem*,* particularly on *StreamDetails.cpp* and *StreamDetails.h,* for handling and extracting detailed information about video streams. These files contain utility functions and classes that assist in parsing and managing specific parts of video streams, such as resolution, codec details, and other relevant metadata.

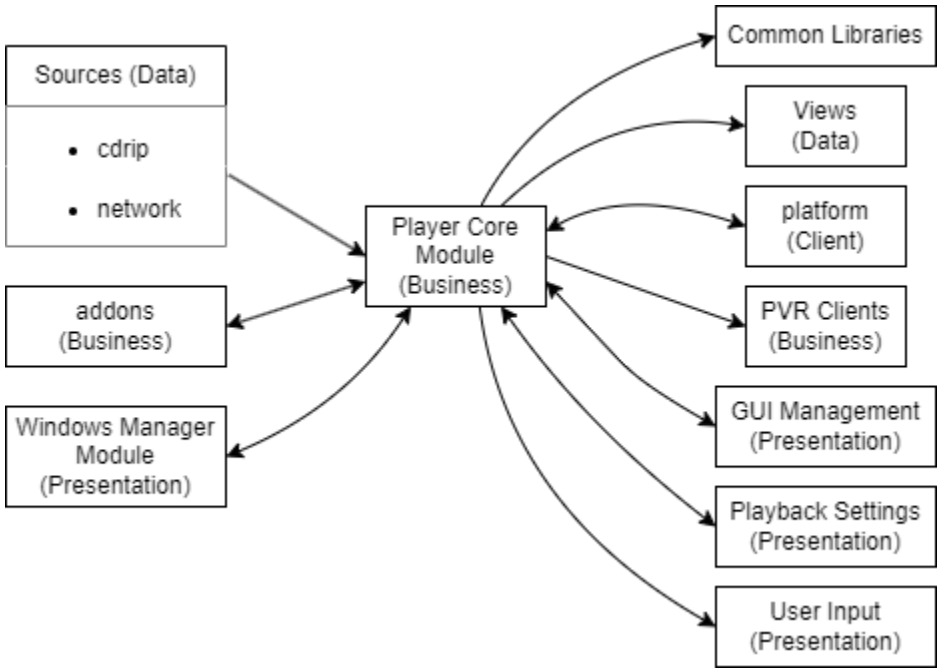

Figure 4: Dependency Graph of the Player Core Module

6. Concurrency

Kodi leverages concurrency to perform multiple tasks simultaneously. The system utilizes concurrency at various levels to enhance performance and responsiveness. One aspect that highlights the concurrency features within Kodi is multithreading in the Player Core Module. The Player Core Module involves extensive multithreading to manage concurrent tasks related to media playback. For example, the Audio Player and Video Player subsystems operate concurrently to handle audio and video streams to ensure synchronized playback. Kodi also employs concurrent handling of background tasks like media library updates and metadata fetching. These tasks are often carried out in the background while the user is interacting with Kodi's interface or during media playback. The use of common libraries supports this concurrency approach, promoting reuse and facilitating the efficient distribution of resources across the platform. Concurrency is extremely important in ensuring that these background tasks do not disrupt user experience.

7. Diagrams and Use Cases

Studying the concrete architecture derived from the Kodi source code, we can gain a better understanding on how data flow is actually implemented. This will be shown by comparing the conceptual sequence diagrams of use cases with the actual implementation in the concrete architecture, as well as exploring more functionalities with our newfound knowledge. Refer to Figure 5 to gain an understanding of the symbology being used in these diagrams.

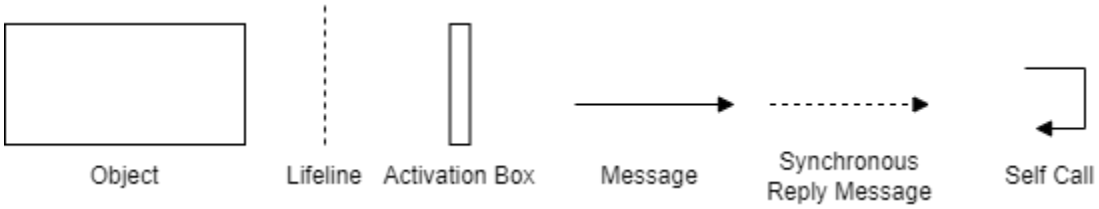

Figure 5: Sequence Diagram Legend

Use Case 1: Media Playback

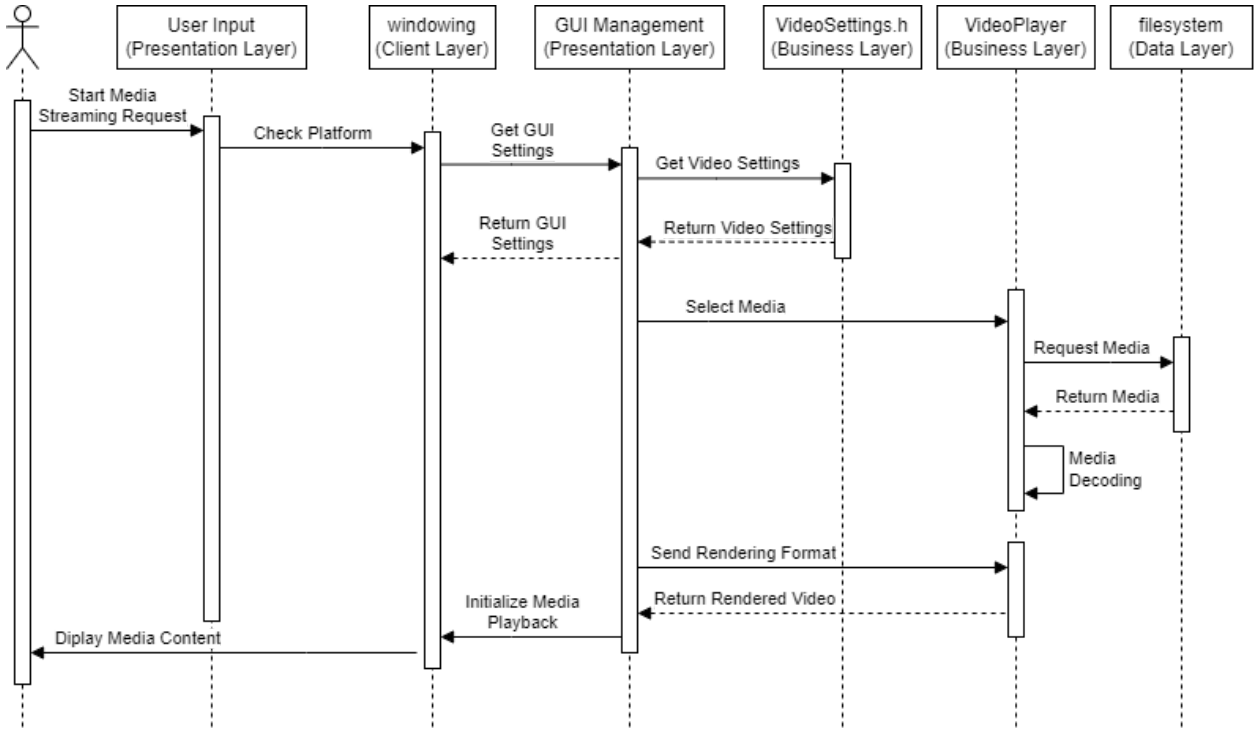

Figure 6: Sequence Diagram for Media Playback

The first use case that we will delve into is Media Playback within Kodi. Using our new understanding of the Kodi system structure, the sequence diagram of this use case has been changed to reflect newly identified divergences, as it now reflects several intermediate steps between the User Input Module to reaching the Player Core Module. The user starts by selecting

their desired media through some form of user input. This triggers a check to the Client Layer, where the windowing directory identifies the platform being used, whether it be Android, iOS, tvOS, Windows, or another. This is an important backbone to ensuring that the Kodi visuals and events can be properly formatted. The Client Layer then must update to reflect changes shown through the Kodi GUI, so it requests the GUI settings, referring to basic dialogs and displays that configure what the user can see. Next, the video settings of the media to be displayed are requested and returned. Having set up the platform for playback, the actual media files are retrieved from the *filesystem* directory within the Data Layer. The codecs within the *VideoPlayer* subsystem of the Player Core Module allow for decoding of media, and rendering occurs through a two-way dependency that signals to the *VideoPlayer* what type of rendering it should use specific to the platform. Finally, the media is ready to be displayed and is ready for the user to enjoy.

Use Case 2: Add-On Installation

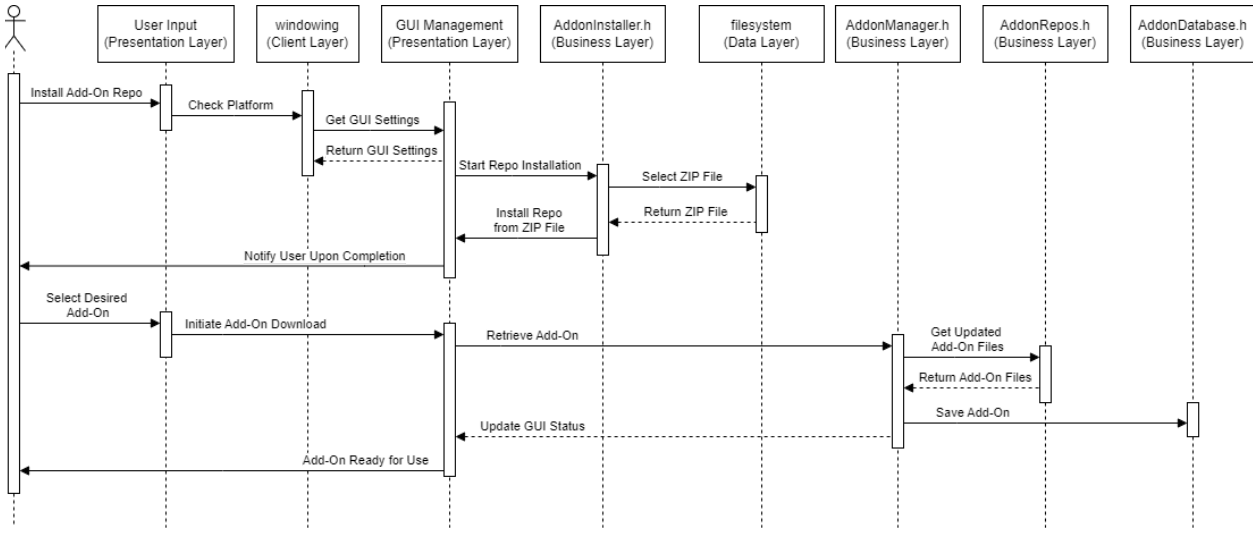

Figure 7: Sequence Diagram for Add-On Installation

The second use case that we will explore is the process of downloading add-ons in Kodi [1]. For this use case, we will assume that the source the add-on will be retrieved from has been downloaded. Similarly to the process of media playback, the first step of any process in Kodi involves checking the *windowing* directory to retrieve information on the platform. The Kodi user interface is then configured accordingly. To begin the actual process of add-on installation, the repository that the add-on is located in must be downloaded from the previously installed source. This occurs by retrieving the corresponding zip file from the *filesystem* of the Data Layer, and is finished by updating the GUI to display a success message back to the user. Upon the completion of the process above, the add-on can now be installed from the repository. The user selects the add-on they would like to install which triggers a call to the *AddonManager.h* file to retrieve the specific add-on. This is done by navigating to the *AddonRepos.h* file to identify the

associated repository and retrieve any updates if an older version had been installed previously. With a now up-to-date add-on, it is saved to the *AddonDatabse.h* file where it can be referenced easily. Now that the add-on is installed, the GUI reflects this status and notifies the user.

8. External Interfaces

As an entertainment hub, Kodi has a variety of external interfaces that allow it to interact with external services, applications, and devices. When delving into the source code, we can see that users can control Kodi remotely or from an external application using a JSON-RPC API. It is also clear that external add-on repositories are key components to the functionality of Kodi as users can discover, install, and update add-ons that expand Kodi's features and access additional content. Some of these include specific streaming services, video downloading functionalities, live streaming services, music services, etc. This can be seen and configured from the *python* interface that contains scripting functionality. Additionally, the use of network interfaces can be seen, such as Universal Plug and Play (UPnP) and Digital Audio Control Protocol (DACP). These network interfaces allow for the sharing of media content across compliant devices on the local network. To summarize, the use of external interfaces allow the Kodi source code to only contain what is necessary, contributing to its quick and efficient structure.

9. Data Dictionary

- **Graphical User Interface (GUI):** A visual interface which allows users to interact with software using menus, controls and graphics
- **Add-ons (Feature):** Small extensions or programs that add functionality
- **JSON-RPC API:** A remote procedure call specification used for data interchange primarily between a client and a network
- **Dependency**: A relationship between software components or modules where one relies on the functionality provided by another

10. Naming Conventions

API: Application Programming Interfaces

- **UPnP (Universal Plug and Play) (Protocol)**: A network protocol that enables communication with other UPnP-compliant devices
- **DACP (Digital Audio Control Protocol) (Protocol)**: A protocol used audio players that enables remote control

11. Lessons Learned / Limitations

The analysis of Kodi's concrete architecture highlighted the important role of reflection analysis in aligning conceptual and concrete architectures. Flexibility in the derivation process became apparent as some adjustments were required to make the proposed architecture compatible with the intricacies of the actual source code. Dependency diagrams that allowed us to visualize subsystem relationships and dependencies across various components played a significant role in the derivation of our concrete architecture. However, limitations include potential inaccuracies stemming from an incomplete or inaccurate understanding of the conceptual architecture, and the challenge of understanding dynamic runtime behaviors purely through static analysis of the source code. Despite these limitations, the insights gained and lessons learned provided valuable knowledge for future architectural analyses in software systems.

12. Conclusion

In this report, we analyzed Kodi's concrete architecture to provide a comprehensive understanding of the system's actual implementation compared to its conceptual representation. The identification of unexpected dependencies highlighted the need for us to be flexible and adaptive in our approach to deriving the concrete architecture. The lessons learned throughout this process emphasized the importance of reflection analysis, flexibility requirements in the derivation process, and the significance of using dependency diagrams for analysis. While limitations such as incomplete conceptual understanding and relying solely on static analysis were recognized, the insights we gained contributed to a greater understanding of Kodi's architecture and design.

13. References

[1] "Open Source Home Theater Software," Kodi, https://kodi.tv/ (accessed Nov. 19, 2023).

[2] "Architecture," Architecture - Official Kodi Wiki, https://kodi.wiki/view/Architecture (accessed Nov. 19, 2023).