Conceptual Architecture of Kodi

CISC 322 Assignment 2 Report

November 19, 2023

Nicholas Falconi (falconi.nicholas@queensu.ca) 20124757

Karina Verma (20kv15@queensu.ca) 20287311

Arda Ozdemir (arda.ozdemir@queensu.ca) 20370479

Jasmine van Leeuwen (20jjvl@queensu.ca) 20256908

Hashir Sami (hashir.sami@queensu.ca) 20285281

Anishka Barran (20ar14@queensu.ca) 20292268

Table of Contents

1 Abstract

This report provides an in-depth examination of the concrete software architecture of Kodi. The report begins by explaining the derivation process of the concrete architecture, using the Understand software tool to map the files in the source code to the appropriate components and subsystems, and updating the conceptual architecture as necessary. Our team extracted the concrete architecture of Kodi and compared it to the updated conceptual architecture, uncovering and adding an additional layer to our conceptual architecture. Our team performed an analysis on the vast number of dependencies present in the various files and components. We focused on the High-Level System Architecture as well as the Player Module Core architecture. Additionally, we investigate the use cases of creating and adding a song to a playlist, as well as customising the user interface through the modification of skins. The use cases were derived with direct reference from the concrete architecture and method calls used in the relevant files.

2 Introduction

This report focuses on the Concrete Architecture of Kodi. In comparison to the previous report, we now have tools at our disposal to help us analyse the architecture of Kodi. Through the use of Understand, we were able to comprehend the differences of our conceptual architecture in comparison to Kodi's actual architecture. Although we did base our architecture off of the Kodi documentation, creating the architecture ourselves through the source code gave us a more complete vision of how different pieces function together.

Understand proved to be a useful tool as it created a visual reference for the components we saw in the source code. The ability to do this helped us quickly develop a concrete architecture that reflected more accurately on the components and subsystems themselves, compared to our conceptual view of the system. Furthermore, a reflexion analysis on the discrepancies was done to discuss our changes and the motivations behind them.

During the conceptual phase, we believed Kodi's architecture could function in multiple ways- layered, repository, or even interpreter style. After carefully looking through the source code, comparing it to Kodi documentation, and understanding different subsystems of Kodi, it became more clear that Kodi has a layered architecture style.

3 Derivation Process

Kodi's concrete architecture was derived using the SciTools software Understand. Our team imported Kodi's source code into Understand, which allowed us to inspect file dependencies and create our proposed architectures.

To derive our architecture, we began by setting up our conceptual architecture model from our first report. We first set up the presentation layer, business layer and data layer, then the subsystems within each. The Understand Tool allowed our team to sift through the files and directories of Kodi and add them to the appropriate layer and subsystem. While categorising files, we noticed that there were several groups of files that did not belong to any category we had previously created or that there were files that too many subsystems were dependent on. After observing these discrepancies, we came to the conclusion of creating a new layer, the Interface Layer. Within this new layer, we decided to create three new subsystems to encapsulate each of the functionalities we had been unable to group into any other layer.

4 Modifications to Conceptual Architecture

Kodi uses a layered architecture, containing four layers: the **Business Layer**, **Data Layer**, **Presentation Layer,** and **Interface Layer**, each with their own subsystems. The largest and most notable modification to the Conceptual Architecture is the addition of the **Interface Layer**.

The Interface Layer contains information that is related to the system used to run the application. We decided to create the Interface Layer, as there were several groups of files that were related to the functionality of the actual application itself, such as tests, threads, and error handling, that we were unable to sort into any part of the architecture we had created thus far.. Because it was clear that these files were intertwined with the actual functioning of Kodi itself, we decided to create the Interface Layer.

We decided to further split the Interface Layer into three distinct subsystems. Firstly, the Application Module which contains files related to the actual environment of Kodi itself, as well as information related to the platform it is running on. Secondly, the Exception Module, which contains all files related to any exceptions that can occur globally throughout Kodi. It does not contain information related to specific errors which may be found in certain functionalities or modules. The third and final subsystem is the Utility Module, which contains the tests and threading components of Kodi. We found this division to be optimal as it keeps each distinct functionality encapsulated within its subsystem, yet allows for an overarching theme of utilisation across the layer.

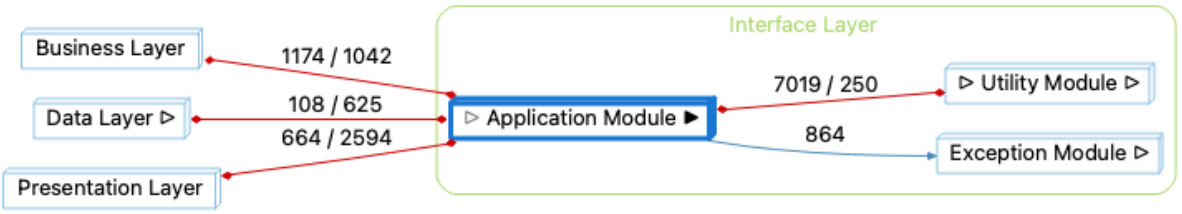
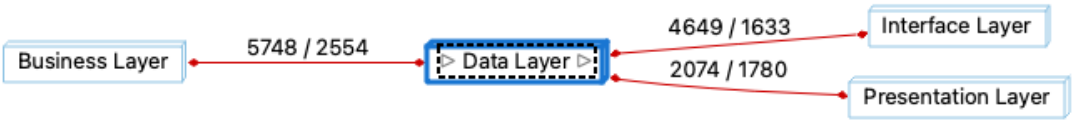


Diagram which illustrates the subsystems present in the interface layer, as well as the interactions of the interface layer with other layers

5 Concrete Architecture of Kodi

Kodi's concrete architecture is organised in a layered architectural style. This follows our teams' initial proposal of a layered architecture. This style allows for components to interconnect without creating strict dependencies on one another. It also provides layers of isolation between the different functionalities, and allows each layer to service the layer above it.



We provide an in-depth overview of the layers and subsystems, building on the insights gained from our conceptual architecture. Throughout this analysis, we highlight any significant differences or features we uncovered and reference our first report's findings if a particular module does not experience any notable differences.

5.1 Interface Layer

The Interface Layer is the newly created layer, containing all information related to the running of the application itself. The Interface Layer ensures a smooth and consistent experience for the user through its three distinct subsystems.

5.1.1 Application Module

The Application Module contains all information related to the actual Kodi application. It stores information on the different Operating Systems and environments it can be run on, as well as the interfaces through which it can be interacted with. The application module ensures smooth functioning of the actual application itself by introducing programs which monitor and handle the volume, settings, and power. This ensures a consistent experience across the application, creating a predictable and consistent experience for the user.

5.1.2 Exception Module

The Exception Module contains information related to base errors or exceptions that can occur throughout Kodi's execution. The module introduces the ability to log any events that have occurred

so that they can later be reviewed. Allowing all of the base exceptions and errors to be stored in one place ensures that messaging stays consistent throughout the application.

5.1.3 Utility Module

The Utility Module contains the files which relate to the actual execution of the Kodi program itself. It contains the programs which allow for threading and concurrency, the paths for the Dynamically Linked Libraries (DLLs), caches, the date and time, and more. The Utility Module also contains tests which are used to verify that certain core aspects of the program, such as the date and time, environment, and actual application itself, are functioning as expected. It contains the definitions and operations for nearly all static components which must be consistent across the entire application, such as basic data structures, including buffers and bitstreams. This standardisation ensures consistency and modifiability across the Kodi application.

5.2 Presentation Layer

This layer encompasses files and packages responsible for user information display, including fonts, language packs, user input handling, and view management. In comparison with our previous analysis from the first report, we found notable updates in the GUI User Input Module, and the User Preferences Module.

5.2.1 User Input Module

The user input module manages and processes user input from input devices such as keyboards, touchscreens, and remotes. This module handles the communication between these input devices and the software application, ensuring that user input is captured accurately and processed appropriately. A notable new feature that was discovered is that Kodi contains support for text to speech input.

5.2.2 User Preferences

The user preferences module contains a variety of subsystems related to selections the user can make to customise their experience. As previously mentioned in the conceptual architecture, the user preferences module does contain the ability to change the language, text, and interface of the display. The user also has the ability to customise far more of their Kodi experience, through the settings subsystem in the user preferences module.

In addition to the aforementioned functionality, the user preferences module also allows the user to set up their own profile, allowing each Kodi user on a particular installation to uniquely set up their own preferences and customisations. The user also has the ability to create playlists or select favourites out of the media they have imported into Kodi. Favourites were categorised under User Preferences because having a favourite is fundamentally a user preference, as is the creation of playlists. Each favourite and playlist would be unique to the particular user and therefore their preference.

5.3 Business Layer

The Business Layer comprises files and packages associated with server interactions, external libraries, and add-ons. It includes a client responsible for establishing connections with streaming servers. Additionally, this layer handles the reading and display of video and audio files utilising

codecs like DIVX and AC3. These codecs are essential for transforming raw data into a user-friendly viewing experience. The most notable change within the Business Layer is the addition of the PVR (Personal Video Recorder) Module. The Python Module and Web Server module also experience notable changes, as the Python Module incorporates two new subsystems, and the Web Server Layer includes heightened security protocols.

5.3.1 PVR Module

Contained within the business layer is the newly added PVR module. The PVR (Personal Video Recorder) module allows Kodi users to browse and record live TV through the application. Similar to a VCR, a PVR has the ability to pause, rewind, stop, or fast-forward a recorded program (“What Is a PVR?” *Flussonic*). The PVR has the ability to record a program and replay it almost immediately, with a slight time lag, which makes it appear to be able to manipulate live programs as if they were recorded programs.

5.3.2 Python Module

Kodi's Python module acts as a bridge between the user and Kodi's core functionalities, allowing users to develop custom add-ons (scripts and plugins) that enhance Kodi's functionality and user experience.

A notable change within the Python module, is the addition of two new subsystems: the Python subsystem, and the add-on subsystem. The Python subsystem contains files that are strictly related to the execution of Python within Kodi. The add-on subsystem contains all other files that are related to the addition of plug-ins and add-ons. Within the source code of Kodi, support is natively provided for GUI, Input, Weather, and Game add-ons.

5.3.3 Web Server Module

The Web Server module grants users the capability to remotely manage their multimedia content through a web browser or other applications. This allows users to access their multimedia library, even when they are not directly interacting with the Kodi application. Multiple users can access and manage the same multimedia content simultaneously if they are connected to a shared local network.

Notably, the Web Server module contains support for several different web protocols and their associated requests, as well as rigorous authentication to ensure the web server remains secure. This addresses the initial security concerns presented with the Web Server Module, where anyone with a link to the user’s Kodi Web Server was able to remotely control it. The authentication protocols implemented include a username and password system, as well as a Zero Trust framework which when combined, provide an appropriate level of security for the web server.

5.4 Data Layer

This layer is responsible for managing all files, functions, and content related to multimedia operations. It includes tasks such as saving files, organising them on disks, streaming content from external servers, and transforming data into formats including FTP, RSS, and HTTP files. The Data Layer includes the Metadata Element, the Source Element, and the View Element, none of which experience significant changes from the conceptual architecture.

6 Player Core Module Analysis

The Player Core Module consists of many components that help maintain and facilitate the main functionality of video playback in Kodi. The Video and Music components will be the emphasis of the module analysis. The Video component contains many features responsible for retrieving metadata for the video player, whereas the Music component works with similar metadata except for the music library and album processing as well.

6.1 Video

6.1.1 Video Info Scanner

The CVideoInfoScanner class, an extension of CInfoScanner, is designed for scanning, retrieving, and processing video information. It offers public methods like Start and Stop for controlling the scanner, AddVideo for updating the database, RetrieveVideoInfo for fetching details, and GetArtwork for obtaining associated visuals. The class introduces a virtual protected method, Process, which is likely the core processing logic for the scanner. The header file also defines a structure, SScanSettings, detailing parameters for the video scanning process, and includes type aliases like IVideoInfoTagLoader and SScanSettings. Additionally, there are static methods declared in the header file, such as ApplyThumbToFolder and DownloadFailed, which are likely utility functions for internal use. The class contains various member variables, including flags, a starting directory, a video database, and sets to track paths for counting and cleaning.

In summary, the CVideoInfoScanner class encapsulates video processing functionalities, employing a range of methods, structures, and member variables. The inclusion of static methods and a protected virtual method enhances its modularity and flexibility for video information scanning and management.

6.1.2 Video Info Downloader

Video Info Downloader is another subclass specialising in retrieving movie information, error handling, threaded task processing, and extracting details about movies and episodes. The constructor initialises class members, including a new instance of CCurlFile for handling HTTP requests, while the destructor manages resource cleanup. An essential method, InternalFindMovie, retrieves movie information by calling the FindMovie method of the scraper (m_info).

The class's main processing loop, the Process method, operates when the class runs in a separate thread. It determines the current state (m_state) and takes actions accordingly, such as finding a movie or retrieving details. The FindMovie method is responsible for locating movie information, running either in threaded or unthreaded mode. In threaded mode, it creates a new thread, waits for its completion, reports progress, and checks for thread cancellation. Additional methods, such as GetArtwork, GetDetails, GetEpisodeDetails, and GetEpisodeList, manage the retrieval of artwork, details, episode details, and episode lists, respectively, using the scraper (m_info). The CloseThread method cancels the HTTP request, stops the thread, resets the HTTP handler, and sets the state to DO_NOTHING.

6.2 Music

6.2.1 Music Info Loader

This component focuses on different methods of managing music related information. The constructor initialises the class, creating instances of CFileItemList and CMusicThumbLoader, while the destructor handles thread termination, memory deallocation, and cleanup. Notable methods include OnLoaderStart for initialization, LoadAdditionalTagInfo for loading additional tag information, and OnLoaderFinish for cleanup.

Additionally, the class interacts with the music database, fetching information about artists, albums, and songs. Member variables include `m_mapFileItems`, `m_thumbLoader`, `m_strPrevPath`, and counters for database hits and tag reads. To sum it up, this class is primarily utilised in managing music-related information within Kodi.

6.2.2 Music Library Queue

The CMusicLibraryQueue class is responsible for managing jobs related to a music library. The class includes a constructor to initialise the object with default settings and a destructor for clearing the job queue.

The class offers functionality to initiate various music library-related jobs, such as exporting and importing the library, scanning for albums or artists, cleaning the library, and handling job cancellation. It supports checking the scanning status, stopping ongoing library scanning, and checking if the queue is currently processing any job. The Refresh method deletes the music database directory cache and updates the user interface.

The OnJobComplete method is overridden to handle the completion of a job, updating the list of queued/running jobs and triggering a refresh if necessary. Overall, this class is designed to encapsulate tasks related to the music library, ensuring proper sequencing and execution of jobs within the context of the larger Kodi software.

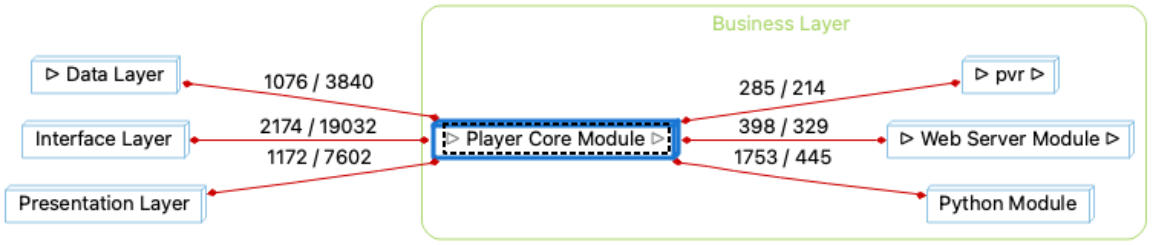
6.2.3 Album Processing

Album processing focuses on CAlbum, a class representing information about music albums in the Kodi media centre software. The code includes various header files for functionalities such as file handling, music tags, settings, utility functions, and logging. The code defines a structure named ReleaseTypeInfo to store information about album release types, and initialise an array of ReleaseTypeInfo structs named releaseTypes with information about album and single release types.

The class implementation begins with the constructor, which takes a CFileItem object and extracts information from its CMusicInfoTag to initialise various album properties. The SetArtistCredits function manages artist credits, considering parameters like artist names, hints, and MusicBrainz IDs. MergeScrapedAlbum is responsible for merging scraped album information, updating artist credits and other details based on MusicBrainz IDs. Other member functions handle tasks such as getting genre information, album artist details, MusicBrainz album artist IDs, converting release types to strings, and setting/getting various date-related information.

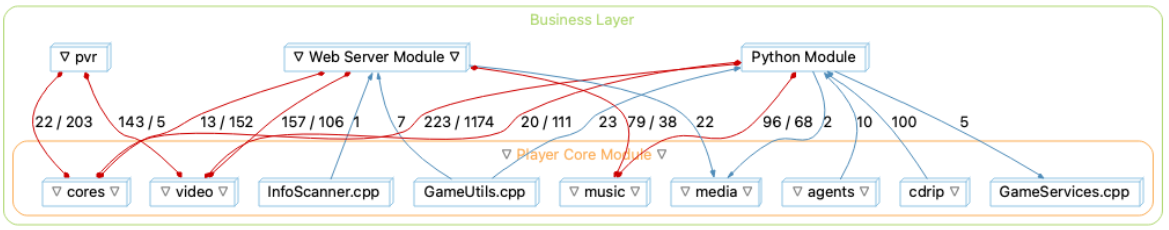
6.3 Reflexion Analysis

The Player Core Module consists of components which maintain and facilitate video playback in Kodi. The Video component contains many features responsible for retrieving metadata for the video player, whereas the Music component works with similar metadata in relation to the music library and album processing.



The above diagram illustrates the dependencies of the Player Core Module both within the Business Layer, and the architecture of Kodi itself.

Initially, we had believed the PVR Module, the Web Server Module, and the Python Module to function independently of each other, as illustrated in the above diagram. We believed that each of the modules would be contained, due to their differing core functionalities. After mapping the files to the corresponding subsystems on Understand, we found that each of the modules were much more intertwined than previously thought.



The above diagram illustrates the actual dependencies of the subsystems of the Player Core Module and the Business Layer

The concrete architecture of the Player Core Module contains no absences, but does feature a divergence, in that the modules are more interconnected than previously anticipated. We believe these discrepancies are due to the intertwined nature of each of the modules. For example, all four modules share the “video” component, because they are all able to perform some functionality related to video. The PVR is able to record and manipulate live video. The Web Server Module has the ability to remotely manipulate Kodi to interact with video components. The Python Module has the ability to enable add-ons which relate to Kodi’s ability to play videos. Lastly, the Player Core Module maintains the functionality of video playback for Kodi.

Kodi’s documentation was a major inspiration in the development of our conceptual architecture. As a result, our concrete architecture aligned pretty similarly when analysing the code. The Player Core module remains in the Business Layer of our architecture and contains all the multimedia components required for the functioning playback of the application. One difference between the conceptual architecture and the concrete is that Kodi uses a lot of integration with the Data Layer to ensure the

business layer works seamlessly. For example, HTTP requests are an important aspect of the Video Info Downloader component. HTTP Requests are normally utilised in the Sources subsystem in the Data Layer. The continuous combination of different parts from different layers reinforces the idea that Kodi’s architectural style is very heavily layered.

7 Investigation of Discrepancies

7.1 Top-Level Architecture Reflexion Analysis

7.1.1 Interface Layer

The Interface Layer is newly introduced in Kodi's concrete architecture. It manages system-specific interactions and enhances user experience through three subsystems: Application, Exception, and Utility Modules. This layer was not detailed in the conceptual architecture, highlighting an increased focus on system abstraction and user interaction at runtime.

7.1.2 PVR Module

The PVR Module is part of the business layer, allowing live TV streaming and recording. Its addition reflects a broader media functionality that extends beyond the original conceptual scope, demonstrating Kodi's adaptation to user demand for live TV features.

7.1.3 Python Module Enhancements

The Python Module in the concrete architecture has evolved with two new subsystems for script execution and add-on management. These changes indicate a more granular approach to add-on integration and script handling, which were not fully fleshed out in the conceptual design.

7.1.4 Web Server Module Security

The concrete architecture introduces advanced security protocols within the Web Server Module, addressing security concerns that were not explicitly considered in the conceptual architecture.

7.2 Unexpected Dependencies

7.2.1 Interface Layer ⇔ Business and Data Layers

Unexpected dependencies might arise as the Interface Layer interacts with other layers, particularly the Application Module which depends on the underlying Business and Data Layers.

7.2.2 PVR Module ⇔ Player Core Module

The PVR Module likely creates dependencies with the Player Core Module for media processing, which were not anticipated in the conceptual layout.

7.2.3 Python and Web Server Modules ⇔ Core Functionalities

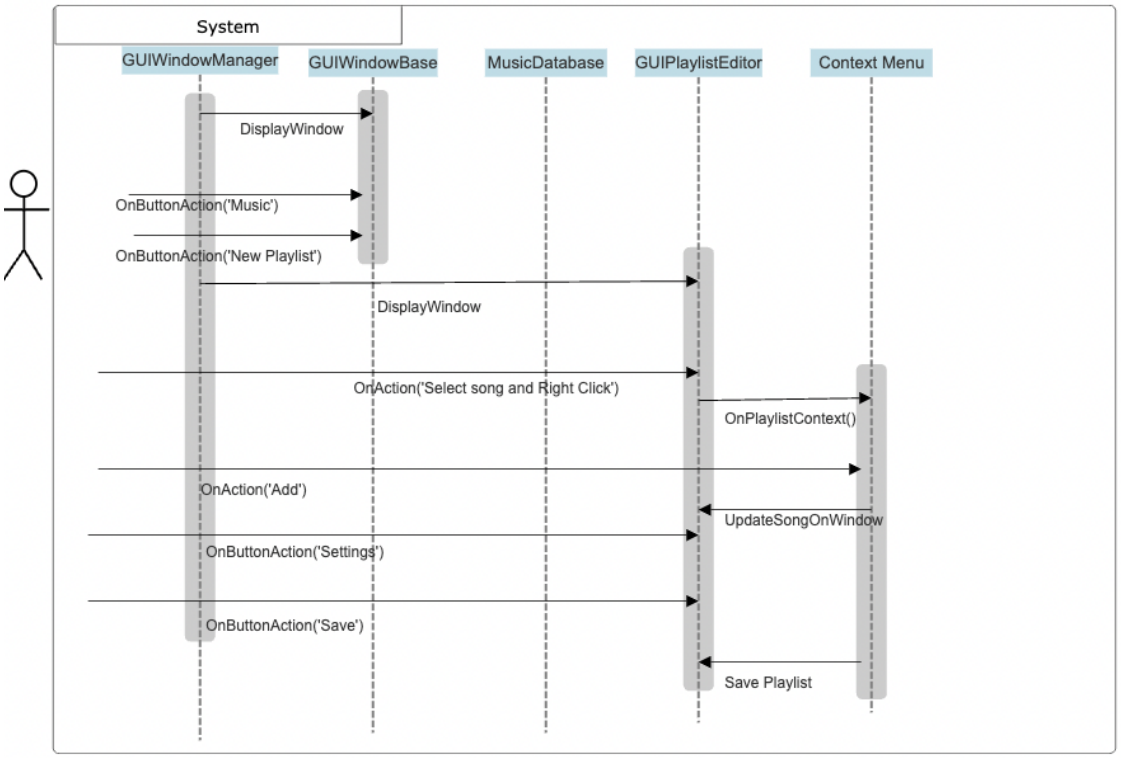
The enhancements in Python and Web Server Modules introduce new dependencies with Kodi's core functionalities, particularly for remote access and add-on management, which were not evident in the conceptual architecture.

8 Sequence Diagrams

8.1 Use Case 1: Creating a new playlist (adding music)

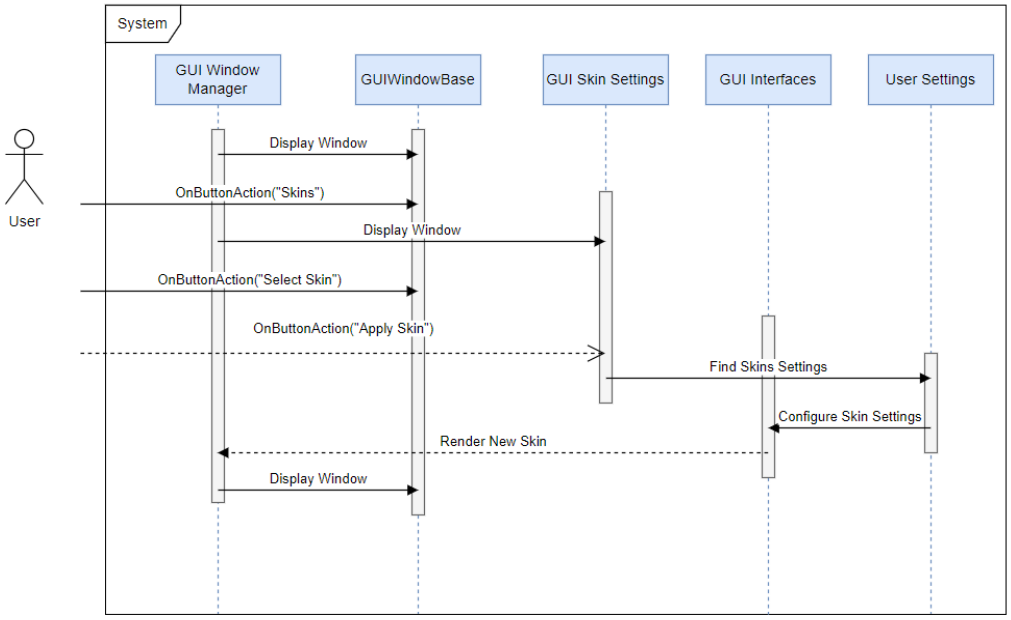
The user interacts with Kodi's interface through Kodi's GUIWindowBase and browses through music. The window manager displays the appropriate window, from which the user selects Music and selects the option of 'New Playlist'. The window manager then facilitates a window change from GUIWindowBase to GUIPlaylistEditor.

The GUIPlaylistEditor displays all of the songs currently in the playlist and songs stored in the user's library. The user highlights the desired song and selects it, triggering an OnAction() call. Through the method OnPlaylistContext() the user is prompted with a context menu, which is a general menu for user input with functionalities dependent on the currently selected item. The user selects add, triggering an OnButtonAction() call on the context window. The GUIPlaylistEditor contains two windows, one that displays songs in the library, and one that has the status of the playlist currently being edited. The contents of the current playlist window are updated with the newly added song. The user then selects the settings icon and saves the playlist, triggering another OnAction() call. The saved playlist is then stored in Kodi's music database.



8.2 Use Case 2: Customizing User Interface Settings (Changing Skin)

The user interacts with the system through OnAction() calls and OnButtonAction() calls. The user initiates changing the skin by navigating through the GUI from the System Settings page to the Skins Settings page. Kodi displays a set of interface options where the user can make the selection “Skins” to open the skins settings. The GUIWindowManager initially displays the GUIWindowBase for the settings page and when “Skins” is selected the window manager then displays the GUISkinSettings. The user can then click on the desired skin to install it, and select yes to keep the changes. If the skin is not already installed the user would have to install it through the addons module. The desired skin is then configured through the User Settings. Kodi is refreshed and the system is updated with the new skin.



9 Lessons learned are discussed

In our first report, we focused too much on the abilities of Kodi and not how Kodi actually worked as an application. This was evident upon review of our conceptual architecture, as there was a great deal of source code related to the actual inner workings of Kodi, and nowhere to place it according to our proposed conceptual architecture. Another challenge was deriving information and the concrete architecture from the source code. Even with the use of the software Understand as a tool, examining and analysing the mappings of the various files and components is overwhelming. The software itself was also a learning curve. After exploring the various diagrams, we discovered which forms of mapping were useful to us and how to hide irrelevant connections. In addition, we learned to focus on only certain subsystems and expand smaller sections.

10 Conclusion

Kodi Media Player stands as a remarkable example of collaborative digital innovation in the realm of multimedia entertainment. Our report has offered an extensive exploration of Kodi's concrete architecture, as well as a reflection on how it aligns with the concrete architecture. Kodi's architecture is primarily characterised by its layered approach, dividing the application into four distinct layers: the Interface layer, Presentation Layer, Business Layer, and Data Layer.

The report dives into each of the layers, updating the conceptual architecture where necessary, and highlighting any notable additions. The report also analyses the Player Core Module, and identifying and performing analyses on each of its subsystems. The report performed an analysis on any discrepancies found between the conceptual architecture and the concrete architecture on both the top-level, and within the Player Core Module, noting any potential reasons for those changes. Furthermore, we explored the two use cases. One was the process of adding a song to a playlist and

the other was customising the user interface by modifying skins. In conclusion, Kodi's architectural design reflects a Layered Architectural style, as suggested by the conceptual architecture, and confirmed by the concrete architecture.

11 Sources

“What Is a PVR?” *Flussonic*, Flussonic, flussonic.com/glossary/pvr/#:~:text=A%20personal%20video%20recorder%20. Accessed 19 Nov. 2023.
Architecture - Official Kodi Wiki. Kodi.wiki. Published 2022. Accessed November 20, 2023. https://kodi.wiki/view/Architecture