CISC 322/326 Assignment 2: Report

Kodi: A Concrete Architecture Analysis

November 19, 2023

Team Torrent (Group 25)

Aselstyne, Alex (alex.aselstyne@queensu.ca)

Dinari, Daniel (20dd29@queensu.ca)

Nagel, Jake (20jn29@queensu.ca)

Peterson, Jack (21jrp10@queensu.ca)

Pleava, Ryan (20rcp5@queensu.ca)

Table Of Contents

Abstract

This paper analyzes the concrete architecture of the software system *Kodi,* utilizing the main repository of Kodi, and the Kodi documentation, Kodi can be broken down into a multitude of interacting components that make up its concrete architecture. It has slightly over a million lines of code and contains [hundreds?] of files. The primary language used is C++, but there are some files and folders that utilize other languages and formats. This paper will also analyze the divergences between the concrete architecture and the conceptual architecture.

Our team found that our conceptual architecture from the previous assignment had some similarities with the concrete architecture and several divergences that we did not anticipate, primarily in the number of components, and the scope of dependencies between them. The team derived the concrete architecture by utilizing Understand, a software designed for analyzing repositories of large-scale software. Understand provided our group with all of the source code of Kodi, and our group needed to map the files and folders to components based on their name and function.

In our analysis, we broke down Kodi into 13 primary components, with some components occasionally containing subcomponents inside them. These components all have dependencies between them, which form the overall software architecture. These components all control the vital aspects of the Kodi app, including the UI, a library file interface, client-specific information, media playback and recording from a local machine or server, among another components. Additionally, the inner architecture of the Player Core component was analyzed. Our team found that the Player Core utilized 3 architectural styles, pipe and filter, for processing and modifying data, a repository for distributing and managing data for other components and a client-server interaction for playing media from the internet and loading addons.

Our team needed to alter most of the conceptual architecture. There were 3 new components not previously accounted for in the conceptual architecture, some components were renamed, and there is many dependencies between components that were previously unanticipated. Kodi's concrete architecture is heavily interconnected, with all components having several dependencies.

Over the course of the project, the team faced several issues, primarily in the mapping of the files. There are limited outside resources that contain complete documentation of Kodi's architecture, the team was highly reliant on intuition fortified with the additional resources. Upon completion of the paper, our group was satisfied with our mapping of Kodi's code to the components, and despite difficulties understanding and translating the Kodi code into English, our team is confident that we understand the inner workings of Kodi.

A reflection and analysis of our conclusions are presented at the end of the report, discussing our limitations of the conceptual architecture and reflecting on what the team learned about large-scale software projects.

Introduction and Overview

Kodi is a free and open-source multimedia center designed to be user friendly and all encompassing. The software is managed by the Kodi Foundation, though the bulk of the development is done by Team Kodi, made up of members of its contributor community. It is available for a multitude of platforms, including iOS, Android, Windows, Mac OS and Linux, and supports a variety of input devices, including mouse and keyboard, game controllers, and touch screens. Kodi has support for a variety of video, audio, and image formats.

The Kodi project was originally developed for the Xbox (2001) as a piece of "homebrew software," or software that is not officially endorsed by Microsoft. Users required a modified console to install the software, which was called XBMC (**XB**ox **M**edia **C**entre) at the time of release in 2003. Over the initial 10 years following its release, it was ported to numerous other systems, including Linux, Windows, Mac OS, Android, and iOS. In 2014, the software was renamed Kodi, to dissociate the software from its Xbox roots.

As a media centre, Kodi does not come bundled with any media, rather the user must provide it themselves. This media can either come from a local disk, another data source on the local network, or a remote server. Add-ons can be used to add additional media sources. These include popular media services such as YouTube, Twitch, Plex, and Spotify. Kodi then handles the entire media playback pipeline, delivering the content to the user for viewing.

This paper outlines the concrete architecture of the Kodi software system. It is comprised of 11 high level components connected through a layered architectural style. Each of these components is made up of discrete subcomponents as well. The subcomponent makeup for the Player Core is explored in detail below, to give context to this subcomponent make up.

Within the paper, we begin by explaining our derivation process. This should give the reader some context for how we arrived at all salient conclusions within the report. We then give a component-by-component breakdown of the system – what the components are, and how they interact. These are supported by the box-and-arrows dependency diagram below. This diagram gives a visual aid to help understand the interactions between the various components. Reflexion analysis was performed on all components and the overall system, comparing it to the conceptual architecture put forward in our previous research.

The report then focusses on one component – the Player Core. This component is broken down into its subcomponents, whose interactions are explained in detail. Reflexion analysis is performed within this section too, explaining how the Player Core and subcomponents differed from our conceptual model.

Two use cases are then explored. These give context to the components, providing examples and explanations of their interactions. The two use cases identified are playing back a local video file and installing a new addon. In contrast with the previous conceptual report, concrete method calls are included in the sequence diagram. Each diagram is also explained in depth to ensure that the use case is well understood by the reader.

We concluded that the conceptual architecture we presented originally was quite far from the real concrete model of the Kodi system. We had several divergences, such as the PVR and Common Library components. We also greatly underestimated the number of dependencies present in the system. By creating and analyzing a concrete model of the system in "Understand", we were able to identify and explain many of these divergences.

Derivation Process

Initially, we first met together and fired up Understand and loaded the Kodi application into it. After creating an Understand project we copied most of the components from our conceptual architecture as a starting point and the Kodi Wiki. Our goal was to sort all the files and folders into a component for our concrete architecture. If we could not decide on a component for the file or folder, we would make a new component.

Our primary strategy used to derive the concrete architecture was utilizing group discussion, background knowledge, intuition, and referring to the Kodi documentation when needed. Our group would select a file in the understand project, and then discuss. If the placement for the file/folder seemed obvious, our group would double-check with each other on its placement, if there was an objection, we would discuss and work it out. Due to the overbearing complexity of the files, we would occasionally resort to the Kodi documentation, the Kodi documentation provided some idea of the purpose of the file but was very vague and poorly documented. If we could still not decide on a component, we would analyze the file's code and look for familiar function calls or documentation. Due to the whole group's involvement, we were able to produce a concrete architecture that was unanimously agreed upon by the group.

Understand's projects form a side bar containing the entire directory of the application, this gives an overview of all the files and their location within the file, when needed a box and arrows diagram can be produced with the existing components, once all files have been sorted, the complete architecture can be generated, with all dependencies between components documented with the exact file in the other component the overall component depends on.

To sort all of the files and folder, we used a large screen connected to a laptop for our group to sort the files, this way we kept the Understand project consistent throughout the group and could download the project on individual devices when it was personally needed. Once we had finished the concrete architecture, we needed to alter the sequence diagrams and architecture from A1 to match our new concrete architecture. We found that our second use case, downloading the Soundcloud addon and playing a song from it was too complicated and didn't use the actual architecture of the software in a clear way, since it was largely dependent on the Soundcloud servers, so we simplified it to just the downloading addon process. We derived the new components for our use cases and function names from the concrete architecture.

High Level Reflection Analysis

After completing the conceptual architecture, there were several component dependencies we thought would be used. After analyzing the concrete architecture, we discovered both new components and new dependencies, which we had not anticipated. First, we will go over the new components and their dependencies. Then, we will discuss the new dependencies between existing components.

New Components

Common Libs

While going through the source code of Kodi, we found numerous generic "helper" functions that were used by almost every component. Some examples of these subcomponents are generic event-publish functions, string/file utility functions, base64 encoding functions, etc. Due to the usefulness and high frequency of utilization of these functions, all other components depend on this component. This component also depends on every other component, due to its util functions. Some of the util functions use types from other components, which allows it to provide more specific utilities. We did not anticipate a common libs component in our conceptual architecture, thinking every component had its own utilities built in.

PVR

Another component that we did not anticipate would exist in the concrete architecture is the PVR component. This component allows Kodi to record videos. This component shares a dependency with a few other components from our conceptual architecture. The PVR component depends on and is a dependency for every component except the Rendering and Request Manager components. This is expected since the PVR is a local device that does not interact with the internet. It also does not need rendering as it depends on the player core, which depends on the Rendering component. As for all the other components, we found that the PVR component has very few dependent files relative to the depended-on files. For example, there are 5 files in which the PVR depends on the library manager for, but 984 files in which the library manager depends on the PVR for. This makes sense because the PVR component is mostly standalone. It acts as a video recorder that other components can use if they need. When constructing our conceptual architecture, we did not know Kodi had this functionality, preventing us from including it in our design.

Library Manager

As mentioned above, one of the main use cases of Kodi is to play media files. From a user's perspective, Kodi can play media from local files, from a network file system, or from another Kodi application running on the same network. Regardless of where the files come from, the process of playing the file, using the player core, will be the same. To abstract the origin of the files, Kodi has a library manager. This component creates a virtual library of media, made up of files from different locations. This component acts as a client to the local file manager, which allows it to read local video and audio files. While the library manager would also be a client to the network manager and web app manager, we are only focusing on local media in our use cases. The library manager provides services to any component that needs to read media files. This includes the GUI manager and the player core.

Client Interfaces

Since Kodi can be run on many different hardware and OS's, it needs a way to handle the OS level functionalities. These include threading, power management, windowing, and platform. It is no surprise that this component depends on, and is a dependency for, all other components. For components to work, they usually need some operating system level tasks to be completed. These kinds of tasks can include I/O, creating/managing threads, etc. All the components in Kodi take advantage of this component to abstract the functionality. The Client Interfaces depend on all other components for smaller tasks it needs to complete. For example, it needs to read and write files when utilizing the file system on android-based devices. Some specific platforms also require rendering in the platform specific tasks. When designing our conceptual architecture, we focused on the main functionality of Kodi, disregarding it as a multi-platform technology altogether. We therefore did not think of adding this as a component.

Application Settings

Kodi can be configured with many different dials and toggles. These all exist in the application settings. Kodi makes use of an Application Settings component to configure and manage settings and app profiles. This component exports its user-defined settings to all other components, so they know in which different ways to operate. This creates a dependency for the Application Settings from every other component in the application. The Application Settings also depends on all other components, as it contains many component-specific settings within. We did not anticipate that the Application Settings would be its own component, but instead each component would have its own settings built in.

Existing Components

While that completes the new components found in the Kodi source code, there were many changes to the dependencies of existing components. We will go over those changes here.

Addon Manager

In our original conceptual architecture, we anticipated there would be an addon installer and an addon component (for each addon). After analyzing the source code, we found out that Kodi combined these components into one Addon Manager. This component has two subcomponents: Addons and Addon Installer. Since these components act for the same feature (addons), it makes sense to put them in one component.

We also noticed this component's dependency list grew. In our conceptual architecture, we assumed these components would depend on the file manager, requests manager, and player core. While these dependencies exist in the concrete architecture, there were a few new dependencies that were not anticipated. From the analysis, we found the addon manager to depend on and be a dependency for all other components of the application. The reason the Addon Manager depends on so many different components is due to the vast number of different addons built into the program. Kodi comes with addons, such as Album scrapers, weather icons, etc. These addons make use of existing Kodi components to seamlessly integrate with the software.

All Kodi's components also depend on the Addon Manager. This is because the addon system was built to be extremely versatile. Different addons can be used to configure almost any part of the Kodi app. Therefore, other components use the addon manager for configuration options.

When designing our conceptual architecture, we did not anticipate the versatility of the addons. We assumed the addons would mostly be standalone, almost like an app within the app. This is why we assumed there to be minimal dependencies between it and other components.

GUI Manager

Both our conceptual and concrete architecture included a component to manage the UI interface: The GUI Manager. While the existing dependencies persisted, we found many new dependencies within the concrete architecture. The GUI Manager depends on and is a dependency for every other component in the application. Every component depends on the GUI manager because they all require some sort of display to the user. The GUI Manager depends on all other components due to its broad functionality. The GUI Manager uses player core and renderer to show on screen menus. It uses the file manager and library manager to store cache and info. In our conceptual architecture, we thought the GUI Manager would house its own on-screen rendering component. It makes a lot more sense to use the existing rendering components to increase reusability.

Library Manager

In our conceptual architecture, we anticipated the Library Manager would only depend on the Local File Manager and the Player Core. It turns out that the Library Manager depends on every other component in the software. All other components depend on the Library Manager, except for the Rendering component. As we discussed the Application Settings, Addon Manager, GUI Manager and Common Libs above, we are going to focus on the new dependency with the PVR component. Since the PVR component records and saves videos to the existing library, it needs the library manager for managing library content. The Library Manager depends on the PVR component when displaying the PVR recordings. It uses PVR component functions to gather data about the PVR recordings in the library. Overall, our prediction of the Library Manager component was relatively accurate. Most of the new dependencies were built with the new components, which we did not include in the conceptual architecture.

Local File Manager

There were many unforeseen mutual dependencies between the Local File Manager and other components in the concrete architecture. We found that the Local File Manager depends on and is a dependency for all other components, except the Rendering component. In our conceptual architecture, we only had three components depend on the Local File Manager (Player Core, Library Manager, Addon Installer). It is evident that almost all other components read or write to files. This makes sense as files are the only way to store persistent data on a computer. Many components will keep their cache and persistent configurations in files.

We also found that the Local File Manager depends on almost all the other components. While this was a surprise to us, it made sense after analyzing the code. The Local File Manager depends on the Player Core for important encoding information. It uses this information when reading media files. The Local File Manager depends on the GUI Manager for file choosing pop-up windows. It relies on the Application Settings component for profile information. It relies on the Request Manager for network file systems. And it relies on the Client Interface for creating file choose windows. All other dependencies were described above.

When designing the conceptual architecture, we did not anticipate the Local File Manager depending on so many different components. After viewing the concrete architecture along with the source code, we understand the new dependencies.

Request Manager

During its runtime, many features of Kodi require some kind of network connection to work properly. The Request Manager component is the core of this functionality.
In our conceptual architecture, we anticipated the Request Manager to have no internal dependencies, and only be depended on by the GUI Manager, Addons, and Addon Installer. After finding the concrete architecture, we found the Request Manager depending on the Addon Manager, Application Settings, Client Interfaces, GUI Manager, Library Manager, Local File Manager, and Player Core. It is depended on by all the same except the Player Core. Since we discussed many of these in our sections above, I will focus on the more notable ones. The Request Manager has many dependencies in the Client Interfaces, since it uses OS level functions for networking. The Client Interfaces also depend on the Request Manager for local networking. Some platform specific implementations use local networks for data transfer. The Request Manager also depends on the GUI Manager for its network setup dialog box. The GUI Manager uses the Requests Manager for network event tracking. For example, it uses the Requests Manager to track when something finishes downloading, to show the user.

Overall, the discrepancy between our dependencies and the concrete architecture dependencies of the Request Manager is due to unforeseen networking use cases.

Rendering

The rendering component(s) had a large change between our conceptual and concrete architectures. In our conceptual architecture, we kept the audio, video, subtitles, and transport renders as separate components. This induces extra complexity as most media need more than one renderer. In the concrete architecture, Kodi combined all these renderers into one Rendering component. This reduces complexity across numerous different components.

In our conceptual architecture, we only anticipated a mutual dependency between the Player Core and the Renderers. In the concrete architecture, the rendering component depends on a few new components. The extra dependencies of the Rendering component are the Common Libs, GUI Manager, Client Interfaces, and Application Settings. Most of these dependencies are self-explanatory, such as the Rendering component leveraging hardware resources from the Client Interfaces. The one component that is not immediately obvious is the GUI Manager. The Rendering component depends on the GUI Manager for a couple of its platform specific rendering libraries. For example, the DirectX library, which is a subcomponent of the Rendering component, uses the GUI Man ager for its screen shot prompt.

Player Core – Reflection Analysis and Architecture Outline

We noticed several discrepancies between our conceptual architecture and our newly created concrete architecture after looking over the source code of Kodi. New components had to be added to our concrete as discussed above. One of the major components that had to be revised was Player core. The player core is responsible for all things playback. Due to the player core having responsibilities that range from different parts of Kodi, it was wise to divide the component into multiple subcomponents. These subcomponents consist of VideoPlayer, AudioPlayer, RetroPlayer, Visualizer, and ApplicationPlayer. This section will give a summary on the responsibilities of the first three components (the most important ones) and then dive deep into the dependencies between each Player Core subcomponent exploring the deviations from our conceptual architecture.

VideoPlayer

The Video Player's main responsibility is the hardware or software encode/decode and handling of the video playback including frame synchronization. This of course excludes the rending and I/O management of the file. There exist various buffers that store frames and metadata after the processing is done by the decoders. After which they output to helping utils in CommonLibs such PlayerUtils, EventStream and JobManager to complete a video playback instance. A lot of the DVD processing is done in this component as well. Such as processing different DVD codecs and demuxing (decompilation of data streams on a DVD).

The VideoPlayer depends on every main component of our concrete architecture which initially was unexpected. The main dependencies of the Video Player subcomponent consist of dependencies to Rendering and Liberary Manager for rendering and loading of the file respectively. These main dependencies we initially accounted for but the large references to platform and windowing inside Client Interfaces was the most unexpected. It is not clear why there are only dependencies between Kodi's Android app that exists inside platform but the dependency to windowing can be explained. Since Kodi runs on different platforms, every unique windowing characteristic of each platform needs to be accommodated by the VideoPlayer. How synchronization of frames happens on Windows may differ to Linux and so on.

AudioPlayer

The Audio Player's main responsibility is much of the same as the Video Player with one key difference, the entirety of the processing of Audio happens in this module.

The AudioPlayer doesn't depend on the Rendering module at all since the rendering module deals more with processing of video using various libraries like DirectX and OpenGL. However, this module does depend on most of the modules that the VideoPlayer depends on. The main dependencies that were unaccounted for in our conceptual architecture are from Application Settings and Common Libs. In specific the subcomponent SettingConditions inside Application Settings is directly feeding its conditions to the ActiveAudioEngine inside AudioPlayer that's responsible for the real-time processing of Audio. Most of this processed data gets outputted from AudioPlayer straight back to CommonLibs for distribution amongst the various libraries that exist in the system.

RetroPlayer

RetroPlayer, is a subcomponent of the PlayerCore, employing emulation libraries such as Libretro to provide a comprehensive retro gaming experience. This allows seamless integration with popular emulators like RetroArch to emulate consoles such as NES, SNES, Sega Genesis, and more. By utilizing APIs like the Libretro API, RetroPlayer harmonizes the interaction between Kodi's media management and gaming emulators.

Unfortunately, RetroPlayer and the game emulation aspect of Kodi were overlooked in our original report so the dependencies of the component were unaccounted for. The leading dependencies of RetroPlayer are from Library Manager, Client Interfaces and GUI Manager. Library Manager provides the services such as GameServices and GameUtils that are static and shared across all games to RetroPlayer's guibridge, rendering and savestates which are dynamic and different from game to game. RetroPlayer's dependency on Client Interface's windowing component is self-explanatory as the games need to run with different windowing characteristics on different OSs.

Concrete Architecture Box-and-Arrows Diagrams

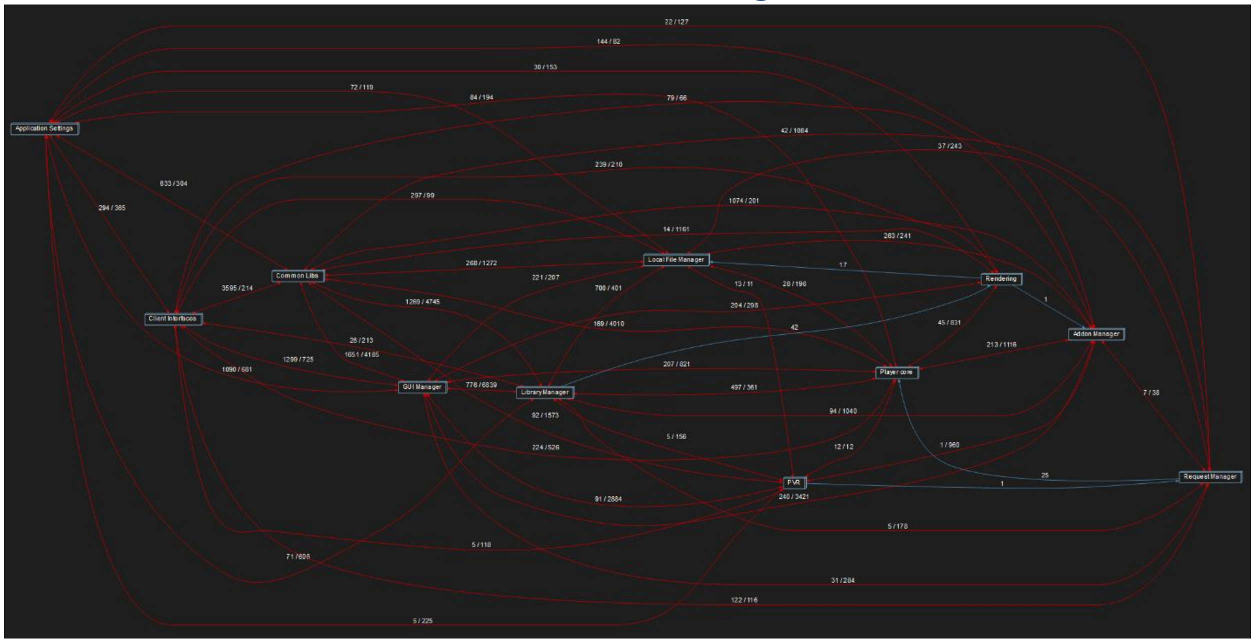

Figure 1: Overall Box-and-Arrows diagram with all outlined components and dependencies. Zoom in for increased legibility.

Use Cases

Below we provide two use cases for Kodi, which span a large portion of its functionality. The first is simple local file playback, while the second outlines downloading and installing an addon. Across these two cases, a large portion of the components in the concrete architecture are utilized and their interactions can be examined.

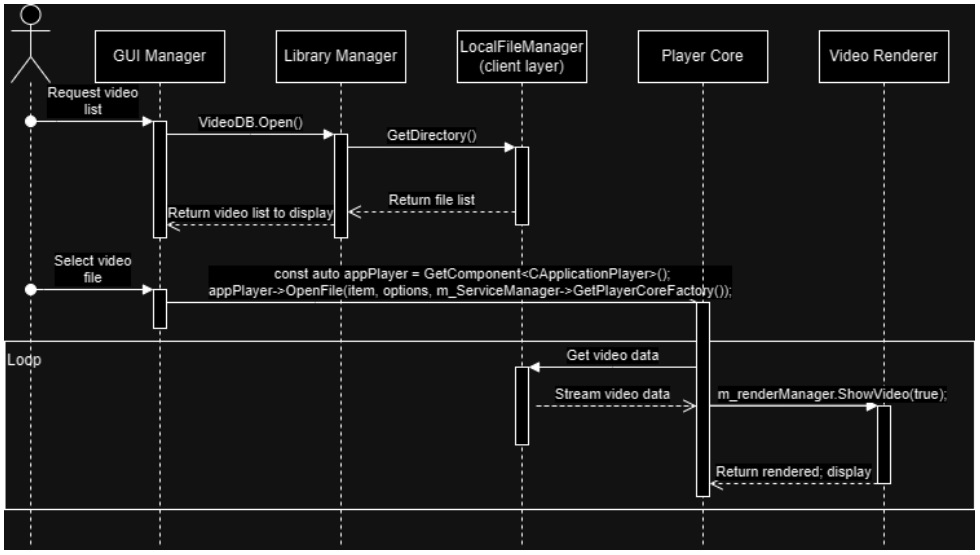
GUI Manager
Library Manager
LocalFileManager (client layer)
Player Core
Video Renderer
Request video list
VideoDB.Open()
GetDirectory()
Return file list
Return video list to display
Select video file
const auto appPlayer = GetComponent<CApplicationPlayer>();
appPlayer->OpenFile(item, options, m_ServiceManager->GetPlayerCoreFactory());
Loop
Get video data
Stream video data
m_renderManager.ShowVideo(true);
Return rendered; display

Figure 3: Sequence diagram detailing use case 1 - a user selecting and playing a video

In use case 1, the user browses a list of video files, selects one to watch, and then Kodi begins playing the file on screen. The first step in this process is displaying a list of video files to the user so they can select one to play.

To do this, the GUI manager first must interact with the Library Manager. The Library Manager keeps a database of directories where video files are stored, as well as metadata about these video files. To get an updated list of videos, the Library Manager must make a call to the Local File Manager, which returns a list of all files currently in the library folder. This information is then passed back to the GUI Manager, so that the full list of video files can be displayed to the user.

Once the user sees the list of video files, they must select a file to play. At this point, the GUI Manager makes a direct request to the player core, instructing it to play the file at the selected location. In a loop, then, the Player Core must request video data from the Local File Manager, get the selected data, and then pass the data stream to the Rendering component. The Video Renderer subcomponent then does all processing on the video, before passing the output stream back to the player core to display the result to the user. Audio rendering also occurs in a separate subcomponent.

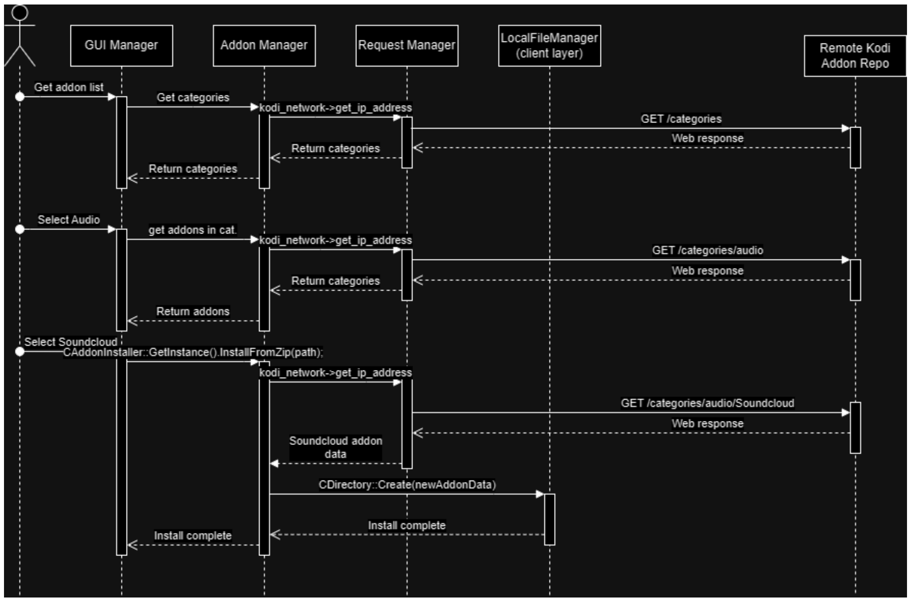

Figure 4: Sequence diagram of use case 2 – a user selects an addon from the available list, installs it, and then uses it to play a song from a remote server. Zoom-in for increased legibility.

When the user first attempts to install an addon, they must select from a list of available addons. To do this, the GUI Manager makes a call to the Addon Manager component. Although the Addon Manager **does** have a local copy of the addon repository, it also frequently connects to the Remote Kodi Addon Repository to download an updated version of the addon list. This is done through the Request Manager component. The results are then returned and displayed to the user through the GUI Manager.

The user must then select a category of addons. The process here is very similar to getting the addon list. The user selects a category, and then the list of addons in that category must be fetched from the remote repository.

Finally, the user selects the addon they would like to install. In the provided diagram, that addon is Soundcloud. The GUI Manager then makes a call to the Addon Manager, getting it to install from the provided zip file path (on the remote server). The addon manager then once again must connect to the Remote Repository, this time to download the actual addon data. Once this data is received, the Addon Manager begins the installation process. To do this, it must make calls to the Local File Manager, so that the new addon data can be installed on the local filesystem. Finally, the addon is installed and ready to be used.

Data Dictionary

Rendering: The process of decoding and displaying a piece of media.
Addon: A small piece of software that can add additional functionality to Kodi.
GUI: Graphical User Interface.
API: Application Programming Interface, used to allow applications to communicate with backend software effectively.

OS: Operating system
I/O: input / output

Lessons Learned

The process of analyzing Kodi’s concrete architecture taught us important lessons in keeping an open mind and adapting to new changes while giving us a hands-on exploration of large-scale software.

Throughout the analysis, it was evident that the conceptual architecture we observed differed greatly from the concrete architecture. Oftentimes, components ended up having far more dependencies than expected due to optimization purposes, with the original expectation being that each component would have complete modularity and functionality baked in. Besides this, there were also times when we discovered entirely new components that we had not thought of before. In times like these it was necessary to keep an open mind on the content in front of us and let go of some of the previous intuitions we had formed about the architecture as they had been proven wrong. It can sometimes be hard to let go of a particular way of thinking about something, and easier to convince yourself that your previous way of understanding it can still be applied to the current situation, despite its shortcomings. We learned that it was important to be patient with ourselves and our understanding of this material, because on average it took poring through more details to gain enough context to progress further.

Another interesting takeaway was learning about the more pragmatic architectural structure that exists at these large scales. From our group’s perspective, we all agreed that there were many key differences between the conceptual architecture from A1, and the concrete architecture from A2. Many of these differences comprised dependencies and components unaccounted for, however, when we thought about why the concrete architecture might have been executed this way there was always a relatively clear answer. For example, the common libs component basically has a codependency with every other area of the software, as it not only provides every component with some universal functionality, but also itself depends on types unique to basically every other module.

In the end, this observation also reinforced the lesson of keeping an open mind by showing us a possible trend in the transformation from a conceptual to a concrete architecture. As we became more familiar with how this large-scale software transformed throughout its integration, it became easier to understand each following piece.

Conclusion

In summary, our analysis of Kodi's concrete architecture revealed an intricate system, designed for high modularity and configurability. Compared to the conceptual architecture, the concrete had many more non-trivial components, that interacted with other components in ways not at all obvious at first.

This theme was common to both the high-level reflexion analysis and our component-level one. They make for a clear demonstration of how large-scale software will almost never be implemented in the exact way it was conceptually designed.

Furthermore, the dependency graph provides context on the interfacing patterns of the Player Core module, a key component in the software. Where there were only about 15 dependencies in our conceptual model, we found hundreds of dependencies in the concrete one. With some analysis and thinking, we were able to understand the primary components.

The sequence diagrams created in A2 contained more detail than A1, and since we had access to exact module and class names, we were able to provide more accurate information. Our use case diagrams highlight the essential control flow and interaction structure between the different components and layers of abstraction.

Looking ahead, there are several avenues for future research and improvement:

- Continue Learning: Kodi is a very large software, and as such it is likely there are still mistakes in our understanding of the architecture that have not yet been caught. We could spend more time exploring Kodi source code to try and figure out the next set of holes in our understanding.
- Performance Benchmarking: Comparative studies can be conducted against other media center software to ascertain Kodi's efficiency and identify areas for improvement.
- Security Measures: With the system's extensible nature through add-ons, enhancing security protocols within its architecture can make it more robust against vulnerabilities.

In conclusion, Kodi's architecture stands as a testament to efficient software design, blending modularity, adaptability, and systematic organization to provide a user-friendly and extensible media center solution. Future efforts should aim to build upon this strong architectural foundation to elevate Kodi's capabilities further.

References

[1] "About Kodi," Kodi.tv. [Online]. Available: https://kodi.tv/about/. [Accessed: 22-Oct-2023].

[2] Kodi.wiki. [Online]. Available: https://kodi.wiki/view/Architecture#Business_Layer. [Accessed: 22-Oct-2023].

[3] "Kodi," Github.io. [Online]. Available: http://delftswa.github.io/chapters/kodi/. [Accessed: 22-Oct-2023].

[4] "Kodi Foundation," Kodi.tv. [Online]. Available: https://kodi.tv/about/foundation/. [Accessed: 22-Oct-2023].

[5] Kodi.wiki. [Online]. Available: https://kodi.wiki/view/History_of_Kodi. [Accessed: 22-Oct-2023].

[6] "Pipe and filter," Berkeley.edu. [Online]. Available: https://patterns.eecs.berkeley.edu/?page_id=19. [Accessed: 22-Oct-2023].