Kodi Conceptual Architecture Report

Authors:

Carter Brisbois	20cbb2@queensu.ca
Dan Munteanu	20dmm14@queensu.ca
Maverick Emanuel	20mje3@queensu.ca
Nathan Gerryts	20njfg@queensu.ca
Oliver Dantzer	20ond1@queensu.ca
Trayden Boucher	trayden.boucher@queensu.ca

Abstract:

This report is a breakdown of the conceptual architecture of Kodi, a media player software. The report consists of an architectural overview, complete with an analysis of architectural styles and a breakdown of software modules. To analyze Kodi's architecture, it is presented from multiple styles, showing how the styles complement each other and encompassing the ways that Kodi's parts interact and share data among themselves. These styles are layered style and object-oriented style, with a discussion on how Kodi appears to have a client-server architectural style, though implementation and use cases would lead one to believe the other styles are better suited. Layered style and object-oriented style work well together within Kodi, due to its design principle for modular design and optional add-ons, allowing objects to work well together within a layered hierarchy. As well, these modules are best described as "building blocks" of Kodi, breaking them down into five building blocks, Skins, Interfaces, Content Management, Player Core, and File Sharing. Three use cases are described with accompanying sequence diagrams, showing the flow of data throughout Kodi for some of its many possible use cases. These use cases include downloading and using an add-on, downloading and selecting a new skin, and downloading and listening to audio. After analyzing Kodi from the viewpoint of various architectural styles and methodologies, it can be deduced that it has a strong foundational architecture, which is future-proof and is easily expandable without worrying about breaking the rest of its features.

Introduction and Overview:

In the constantly evolving world of software development, the foundation upon which an application is built plays a key role in its longevity and success. A thoroughly planned and well-designed software architecture ensures that an application functions seamlessly and is adaptable for the future. Technology is constantly evolving, applications must keep up with this change to remain competitive. Kodi is a perfect testament to this, it has remained competitive in the industry for over twenty years since its initial release. This report will dissect Kodi's brilliant software architecture, shedding light on the key elements that have contributed to its longevity.

Software architecture, similar to the blueprint of a building, serves as the structural foundation for software applications. In the same way that a blueprint ensures stability and functionality, a carefully planned out software architecture guarantees an application's robustness, performance, and maintainability. Good software architecture is essential for overcoming unforeseen challenges, something that all developers have had to deal with before. Planning the architecture of an application prior to developing it is the most important step in the creation process.

Kodi is a free, open-source media player application. It allows users to play and view videos, music, podcasts, and other digital media from the internet as well as content from local storage. It has continuously been in development since its initial release in 2002. It was originally designed to be a home theater application, it was released under the name *Xbox Media Player*. As more new devices were being released in the early 2000's, the developers realized the importance of support for these new devices and operating systems. Some of the first operating systems that were provided support for include; Android, Linux, MacOS, Windows. In July of 2014, the application was given the name that we all know today, *Kodi*. A huge part in Kodi's success stemmed from their support for custom add-ons for the software. They provide several open APIs to aid developers in creating add-ons. This makes the application extremely customizable to the user, allowing them to tailor it to their own needs.

This Report Consists of 9 key sections. Firstly there is the abstract, giving a brief overview of the report. Next there is the introduction and overview providing information about what Kodi is and what this report is composed of. The Architecture section describes the layout of the conceptual architecture of Kodi and how each of these parts interact with each other. The next section features a dependency diagram, it visualizes the relationships between the different components. The diagram section gives an overview and contains any necessary information about the diagrams. Next is the external interfaces diagram which lists transmitted information that is inputted or outputted from the system. Following this, the use-case section provides 3 use-case examples of the objects and modules interacting with each other. After this is the Data dictionary and naming conventions section. Finally, it wraps up with the conclusion and lessons learned sections. It details the results of our findings and discusses what we learned from the investigation.

Architecture:

Kodi could be interpreted as having a few different architectural styles. At a surface level, it would seem that it uses a client-server style. Kodi provides features for streaming online content, the perfect scenario to use a client-server architecture. The content is stored in a server which then transfers the media data to the user’s device. While this does seem to fit, there are other aspects of Kodi that would possibly make it better suited for another architectural style.

We then moved onto the idea of a layered architecture. It would make sense to have different layers for a program like Kodi as the separate layers may not have to rely on one another. This would provide advantages to the developers, such as making testing and implementation easier. However, the downsides to this model would restrict the developers in some other ways. It would make scaling and updating more difficult, and it may even impact performance in the long run. We were confident that this was close to what Kodi uses, but we knew it wasn’t completely correct.

Upon further investigation, we determined that Kodi uses an object-oriented approach. Not only this, each object has its own layered architecture to further separate different aspects of the code. For a program as large and long-lasting as Kodi, this is the smartest solution to ensure longevity. The modularity of having both independent objects and layers within each object makes scaling and improving Kodi much more manageable. Having separate objects is also beneficial to the stability of the program. One benefit is that if one of the modules fails, the others can keep the software running. This stability also means that one object can be modified without affecting the rest of the program. Since they are separate, the content and data management is not affected by changes made to another module such as the GUI.

The conceptual architecture of Kodi can be separated into five distinct objects of modules that work in parallel. The first of these focuses on skins and customizability. It contains files pertaining to skins, translations, fonts, and the GUI of Kodi. The second object focuses on add-ons, streaming clients, event servers, and external libraries. These first two are extremely useful as they provide everything needed for those looking to customize Kodi to their liking. Continuing on, the third object contains files required for handling the content regarding multimedia. The fourth object is dedicated to handling the core features of the video and audio player. It contains codecs responsible for converting the raw data files into the media itself. Finally, the fifth object specializes in file sharing. This pertains to add-ons that allow streaming, downloading, scraping, and sharing media content. These five building blocks have provided the modularity to allow the program to thrive for so long. To further expand upon each of the objects.

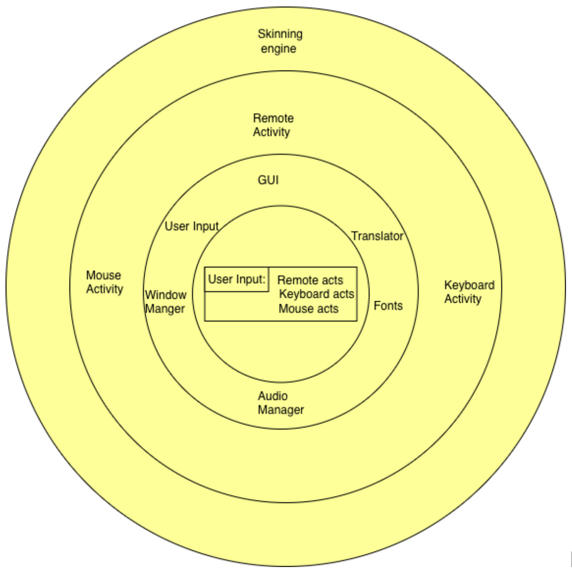

Figure 1

The first object is dedicated to handling skins and customizability. Skins can be used to change the look and feel of the Kodi GUI. Kodi has a robust framework for the GUI, making skinning and customization of its appearance very simple and accessible. Users can create their own, modify existing, and share skins with other users. This can be further broken down into its separate layers. The bottom layer handles user input such as mouse, keyboard and remote inputs. Moving up, the next layer includes modules for handling windows, GUI, fonts, and the translator. These modules depend on the third layer, it has modules that manage the skinning engine. Finally, the fourth and top layer for this object is the skinning engine itself, which contains and manages everything that it encloses.

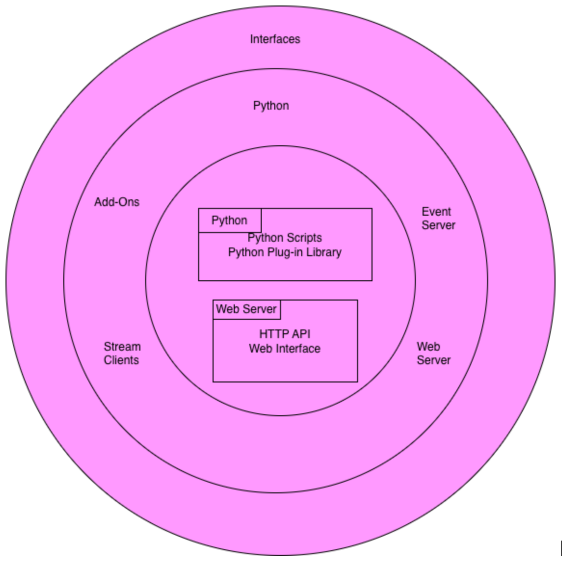

Figure 2.

Continuing on, the second object handles the interfaces. It has three layers instead of the four that the skinning object has, this is because this object does not have to handle any user input. The bottom layer contains modules that handle running scripts and plugins, it handles the API, and the web interface. The next layer contains modules that handle the running of the modules in the layer below. It also contains modules for the event server, add-ons, and stream clients. Finally, the top layer contains interface modules that manage the rest of the contained modules. These modules are extremely useful as the web server's API and interface are used to distribute information from the server. The event server and stream clients contained in this top layer handle events and streaming clients via the server.

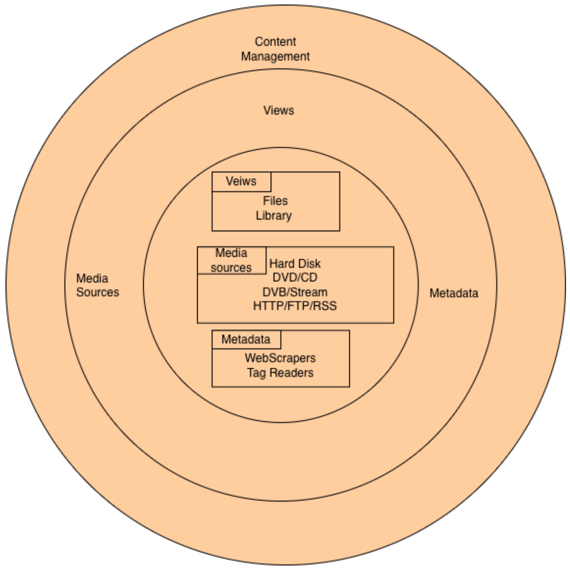

Figure 3

The third object is used for content management. It contains databases that reference files, functions, and content management packages for media use. Like the interface object, this also has three layers. The bottom layer contains modules that handle the reading of data from drives, DVDs, files, libraries, and more. The second layer has modules that separate the different sources of media being read, organizing it for the top layer to manage. The top layer, like previously stated, handles the rest of these modules. It also has its own modules for reading metadata and handling media sources.

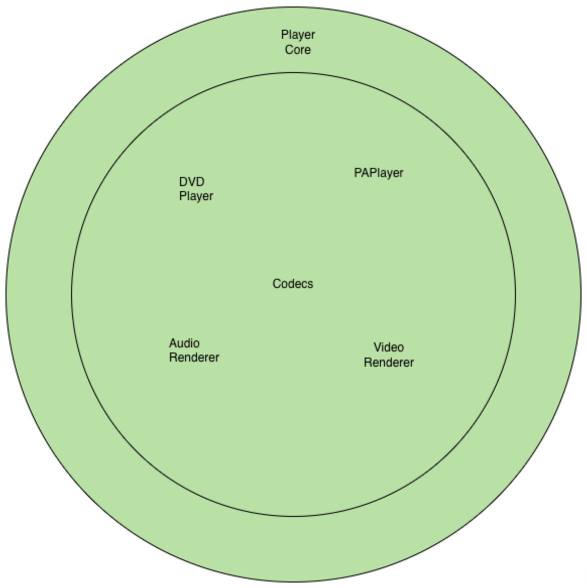

Figure 4

The fourth object is the core component used to display and play audio and video in the application. It contains renderers for audio and video and codecs responsible for the playing of media. The first of the two layers contains modules to handle rendering audio and video, codecs for decompression, and audio and video players. These modules are controlled by the top layer which forms the core of Kodi's application, the media player.

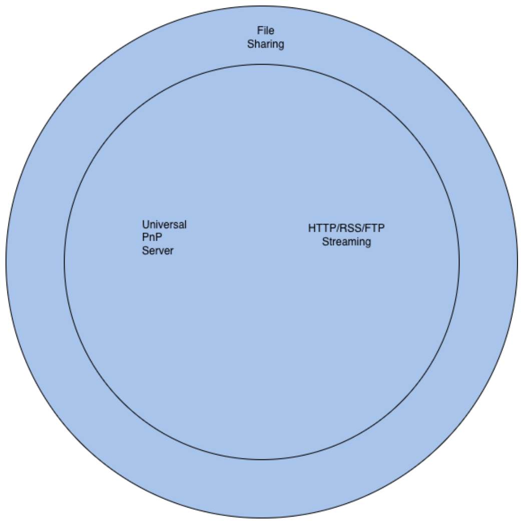

Figure 5

Finally, the last building block to Kodi's software is for file sharing and streaming. This block is specialized for sharing files, relating to add-ons that allow streaming, downloading, scraping and sharing of external movie/audio files. As well, it contains modules for HTTP/FTTP/RSS/Streaming and PnP servers. Again, this only requires two layers. The bottom of the two contains modules for the universal PnP server and streaming modules. The top layer acts as a manager for these modules, ensuring a smooth, uninterrupted user experience.

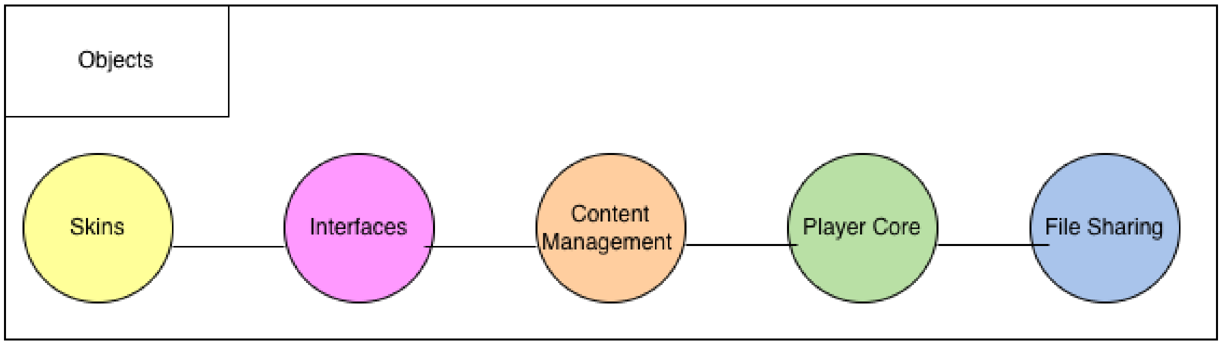

Figure 6

These five building blocks run concurrently to handle the varying aspects required for Kodi. Any communication between the objects happens at the highest layers of each. The top layers for each object essentially act as managers for the rest of the enclosed modules. Since they act as managers, it makes sense that they also handle any communications as well.

Dependency Diagram:

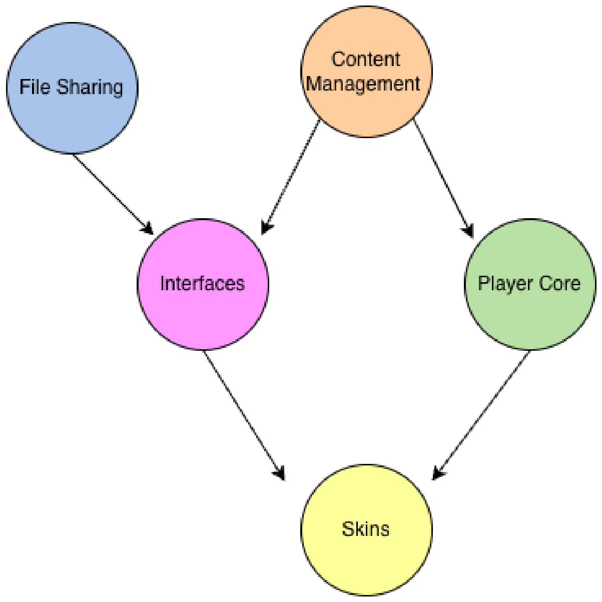

Figure 7

Diagrams:

Figure 1-5: Layered Diagram

- Circle: Each circle represents a layer in the object/module starting from the inner layer(middle)

Figure 6:

- Circle: Object/Module

- Line: Connector representing connection with all other objects/modules

Figure 7:

- Circles: Modules
- Arrows: dependencies

Figure 8-10: Sequence diagrams for the use cases, with the following diagram legend:

- Lifeline: Vertical Lines representing the objects timeline
- Actor: represents User
- Message(Solid): Represents a synchronous communication
- Message(Dashed): Represents asynchronous communication
- Activation Bar: Box on object lifeline representing activation duration
- Loop: Represents a continuous loop in program made up of box with loop in top right corner
- If statement: Represents a choice the Actor must make

External Interfaces:

- Translations, Fonts, Skins via GUI
- Media (Images, Video, Audio) via downloaded files on disk and from server
- Scrapers to read & extract metadata from external databases to fetch information about movies, shows, music, and images
- Streaming servers
- User information & login
- Video codecs

Use Cases:

1. *Downloading and Using an Add-on*

One of KODI's main features is the ability to download and use Add-on features. These Add-ons could extend one of the five modules presented in the conceptual architecture. One such example is TMDb TV Shows. This addon scrapes a community driven online database for TV show metadata (Title, Genre, Year of Release, Actors, etc.). Users would navigate using KODI's builtin interface to download this add-on. Once downloaded, the add-on automatically updates the metadata for the file on hand. This use case uses four out of the five modules present in the conceptual architecture.

Skins:

The user uses and navigates the Kodi media player to download a new add-on. The inputs are keyboard and mouse via the GUI.

Interfaces:

This is where the interfacing for the add-ons lie. This includes the Python scripts as well as the HTTPS API.

File Sharing:

The addon accesses the web, in this case TMDb.org to scalp its metadata, downloading the metadata and sending it to the data module.

Content/Data Management:

The metadata that is scraped is received here, and from there it is stored on the device and presented to the user at time of viewing.

Sequence Diagram:

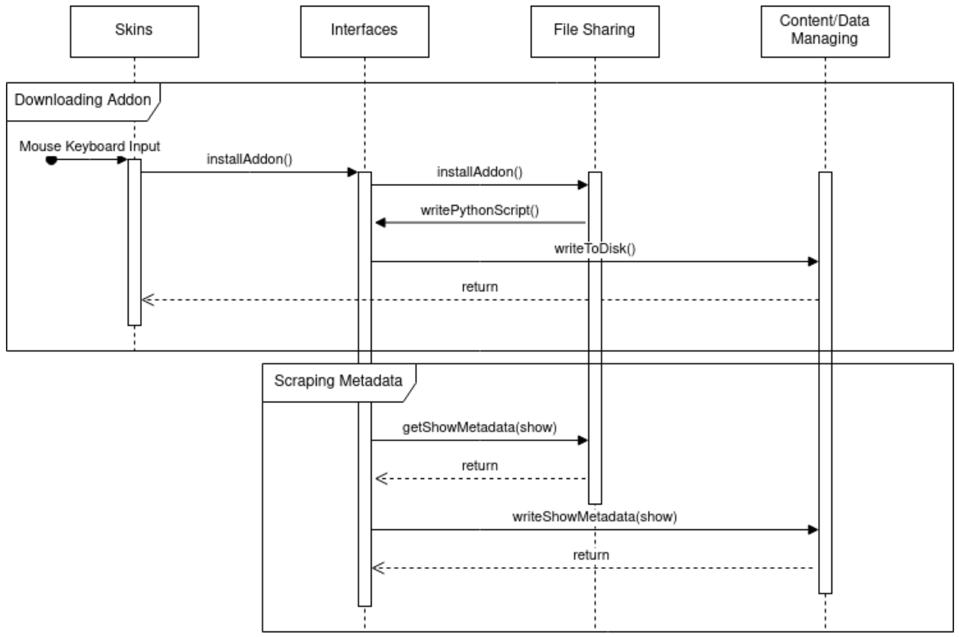

Figure 8

2. *Downloading and selecting a new skin*

Kodi allows users to change the “skin” of their GUI. This allows users to personalize and change the navigation of their KODI interface. To change a skin a user would first navigate to system settings then under Interface go to skin and select new skin. Then, a new window will appear with the option to “get more”, and from this a user will be able to select a new skin and change the interface of their KODI. This use case involves data sharing and interactions between 4 modules:

Skins:

The user uses and navigates the Kodi media player to change and personalize their skin. The inputs are via the GUI, using keyboard and mouse input to allow the user to modify their skin.

Interfaces:

This is where the web interface lies, allowing a user to view skins of other Kodi users, as well as select these skins and download them to their device.

File Sharing:

To select a new skin a user must access the web and search for a new skin to be applied to the interface. Afterwards, this skin is applied and saved to the user's device.

Content/Data Management:

After receiving the skin file from File Sharing, the new skin info will be stored here on the user's disk drive or storage device.

Sequence Diagram:

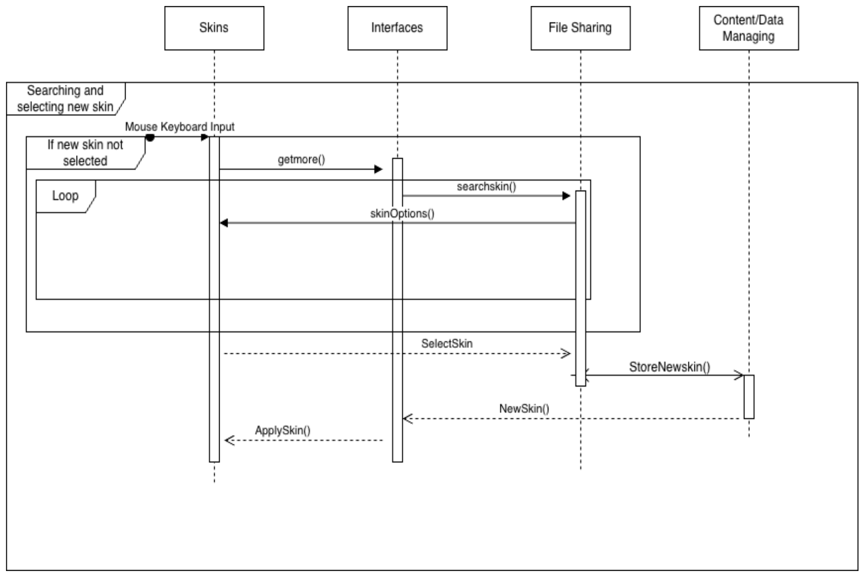

Figure 9

3. *Download and listening to audio*

Kodi allows users to download and listen to music. From the home menu select music, then you are able to view the options to add music via the GUI. Kodi will then provide an interface to input the music source. Next you can use File Sharing to find a FTP based provider to download audio. Next Kodi will scan the source to the library and finally from the music library you may play music. This Use case involves 4 modules of the conceptual architecture.

Skins:

The user inputs their selection to either download or select already downloaded music via the GUI, using keyboard and mouse input. This input is sent to file sharing to download the audio, or Player Core to play the audio.

File Sharing:

File sharing downloads the audio from an FTP provider, then sends it to the data module for storing.

Content/Data Management:

The audio file is stored on the device, allowing it to be later used via input into Kodi from the user.

Player Core:

When the audio is selected to play, the Player Core will decompress the audio and send it to the GUI to play the audio.

Sequence Diagram:

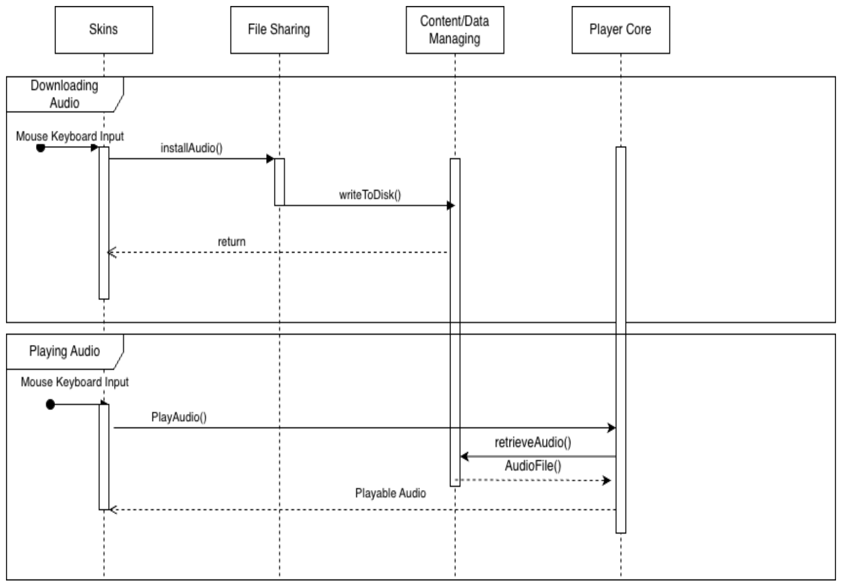

Figure 10

Data Dictionary:

Add-on: An extension that can be added to Kodi, providing additional features. Developed by a third-party.
Third-party: A developer or development team that is independent of Kodi, developing add-ons on their own.
Software: Applications that are run by a computer.
Software architecture: The design and structure of applications.

Layered architecture: A style of software architecture that involves dividing a program into layers that each perform a specific role.
Object-oriented style: A style of software architecture that involves dividing a program into self-sufficient objects that may interact with each other.

Naming Conventions:
GUI: Graphical User Interface
FTP: File Transfer Protocol
API: Application Programming Interface
PnP: Plug and Play
HTTP: Hypertext Transfer Protocol
FTTP: Fibre to the premises
RSS: Really Simple Syndication
DIVX: Digital Video Express
AC3: Audio Codec 3

Conclusions:

Through our investigation, we determined that Kodi makes use of two architectural styles. Kodi uses an object-oriented style that has five separate objects for handling different tasks. Along with this, each object has its own layered style. This further breaks down the code, making it extremely modular and easy to understand. By combining two styles, Kodi reaps the benefits of both. Through this modular design integrated with optional Python add-ons, Kodi is able to evolve and expand without having to worry about system specific dependencies or compile errors. This is because bugs in the add-on only break the add-on itself, and not Kodi as a whole due to its modularity. The potential for evolution combined with the genius architecture of Kodi is a large factor in its success it has seen for so long.

Lessons Learned:

We learned many valuable lessons from this project. It was a great way to further our understanding of the topics we've been learning about. It was also extremely valuable to investigate a real-world application of these topics. It made us realize that there's a large difference between the theory behind software architectures and actual applications of it. The theory behind it is extremely categorical, judging whether one example would be a better fit for one architecture or another. Through this investigation, we realized that those lines can blur when discussing such a large-scale application. Due to the scale of a program like Kodi, it may be useful to developers to combine multiple architectures rather than stick to just one. Blending different styles together can help developers by further breaking down such a large application. We also came to the realization that you can design programs in many different ways. As long as it’s organized and clear, you can make the same application using more than just one architectural style. This was made clear when we realized that Kodi may use more than just one style of architecture. We had many different stages where we thought Kodi was one architecture, only to realize that it doesn’t entirely match. At first we thought Kodi would be a client-server architecture. We then realized that there’s much more to the program than we thought, so we switched to focusing on a layered architecture. After further thought, we didn’t completely agree that it just uses layers. That’s when we came to our final conclusion that it’s a mix of both object-oriented and layered styles. This process taught us a lot, it gave us a much better understanding of the topic than any lecture can teach us about.

References:

Corrales, L., Fonseca, M., González, G., Loría, J., & Sedó, J. (2023, July 10). *Architecture*. Official Kodi Wiki. https://kodi.wiki/view/Architecture

Dreef, K., van der Reek, M., Schaper, K., & Steinfort, M. (2015, April 23). *Architecting software to keep the lazy ones on the couch*. Delft Students On Software Architecture. http://delftswa.github.io/chapters/kodi/