CISC 322/326 Assignment 2
Concrete Architecture of Kodi

Group 11
Schuyler Good
Dylan Walker
Tim Mah
Akshay Desale
Qays Ebrahim

November 14th

Abstract

Our team conducted an in-depth analysis of Kodis source code using the Understand software, reviewing the file structure dependencies, and summarizing its main structure. We found many divergences between our conceptual and concrete architecture, mostly a difference in top-level concrete sub-system components. For example, we conceptualized that the settings component and database component wouldn't interact with each other, but after further discovery, we revealed that they do have dependencies on one another to save user-specific data such as skins, fonts, etc. We gathered a total of 11 top-level concrete subsystems outlined in our Concrete Architecture section including a diagram of how they all interact. The architectural styles we identified include Repository, Layered, Object-Oriented Programming, and Client Server. We chose to explore the Player Core subsystem which was a crucial component for media playback, it housed all the different media players including video, pictures, music, and games. In addition, our report also elaborates on two use cases which are Downloading Addons and Playing a Video. Overall our report dives into the concrete architecture of Kodi, and the differences we identified with our conceptual architecture.

Architecture Derivation

Our derivation of the conceptual architecture of Kodi can be broken down into three stages. Firstly we needed to refer back to our conceptual architecture to get an idea of the components and subsystems we presumed made up the application. This gave us a starting point to derive the concrete architecture because we knew what type of components to look for when breaking down the components.

Once we considered the components from the conceptual architecture, we tried to match them with classes and folders from the source code. We looked for names that resembled the ones from our conceptual architecture (the locations of components in the file system). During this process, we managed to uncover new components that we had not imagined in our conceptual architecture. We took note of them for the next part of the report, so that we could generate a more complete view of the concrete architecture.

Feeding Kodi’s source code into Understand gave us a breakdown of the largest classes and components. We generated a dependency diagram, greatly helping us develop an understanding of the subsystems. The Understand app was incredibly useful, allowing us to see a simplified version of all the components within the architecture and all of the subsystem interactions. The app also allowed us to refine our view of the architecture style, since we generated a detailed connection map between components. Lastly, we generated function graphs, which show which functions require what components. It was also used to figure out how functions worked together and supported one another.

Concrete Architecture

Top-level Concrete Subsystems

User Input

The user input component is an essential component that handles Kodi's different user input types. When looking into the source code for Kodi, using the Understand software, we noticed that the tools directory resembles the user input component because it depends on platform (kodi's connection to each user's operating system) and guilib (user interface).

User Interface

The main function of the user interface component is to display content to the user. It displays media from the Player core component as well as GUI and system elements like settings. We found that the directory guilib handled the User Interface functionality. The name of the directory is short for "graphical user interface library," leading us to confirm that it does handle the user interface.

Settings

The settings component handles the user's system preferences, such as skins, fonts, etc. It interacts with the Metadata component and the Database component to fetch and mutate the data whenever the user changes it. More obviously, this component is housed in the settings directory. We deduced that this directory is dependent on the lib folder and file system as well.

Views Manager

The Views Manager component handles both how the user can view media and the file manager for the music, videos, games, and pictures libraries. It interacts with the Database component to get the files from the database to send to the player core and eventually be displayed or outputted into the User Interface using the Video player, PA player (pulse audio player), or Retro player (used for games).

Addons

The Addons component is an integral component of Kodi because it allows 3rd party developers to add and extend Kodi's pre-existing functionality. It interacts with a database to fetch all the publicly available add-ons and also interacts locally to retrieve user downloaded or created addons. The addons component also interacts with the Plugins & Script component, which allows for 3rd party users and developers to customize parts of Kodi.

Player Core

The Player Core component is responsible for media playback. It accesses the database to gather media files. Then it decodes the files and works with the hardware to smoothly render the content and display it on the user interface. The Player Core is located on the business layer and interacts with aspects of the data layer to access content, as well as the presentation layer to output media to the user.

Metadata

The Metadata component is essential to the functionality of Kodi because it gives the system a means to differentiate between media types. The component stores information gathered from media files in the form of tags, which it then uses to sort and display the content for the user. This component interacts with the database to gather information about each of the titles and stores the file's metadata. It then interacts with the Views Manager to display the file's information to the user. The Metadata component is located within the data layer of the codebase.

Database

The Database component is a central component to the functionality of Kodi. It houses all the metadata and file contents for each video, game, picture, or music item used by the application. It includes any media content accessed from the Sources component and through addons. It is located in the data layer of the application and has connections to many other components in both the data layer and business layer. It accesses files' metadata to ensure that the Database and Metadata component have the same information. The Player Core also accesses the database to retrieve media files that need to be played. Users can share and manage content in the database remotely using the Web Server.

Plug-ins/Scripts

The Plug-ins/Scripts are the components that allow you to access content from various sources like streaming services, online repositories, websites etc. They add functionality to Kodi, enabling users to stream videos, music or access other contents directly. These components let users build their own scripts to perform various tasks within the Kodi environment.
These components also interact with the Web Server component to access the publicly available plugins of various streaming services.These components interact alongside the Addons component to implement 3rd party developers' scripts.

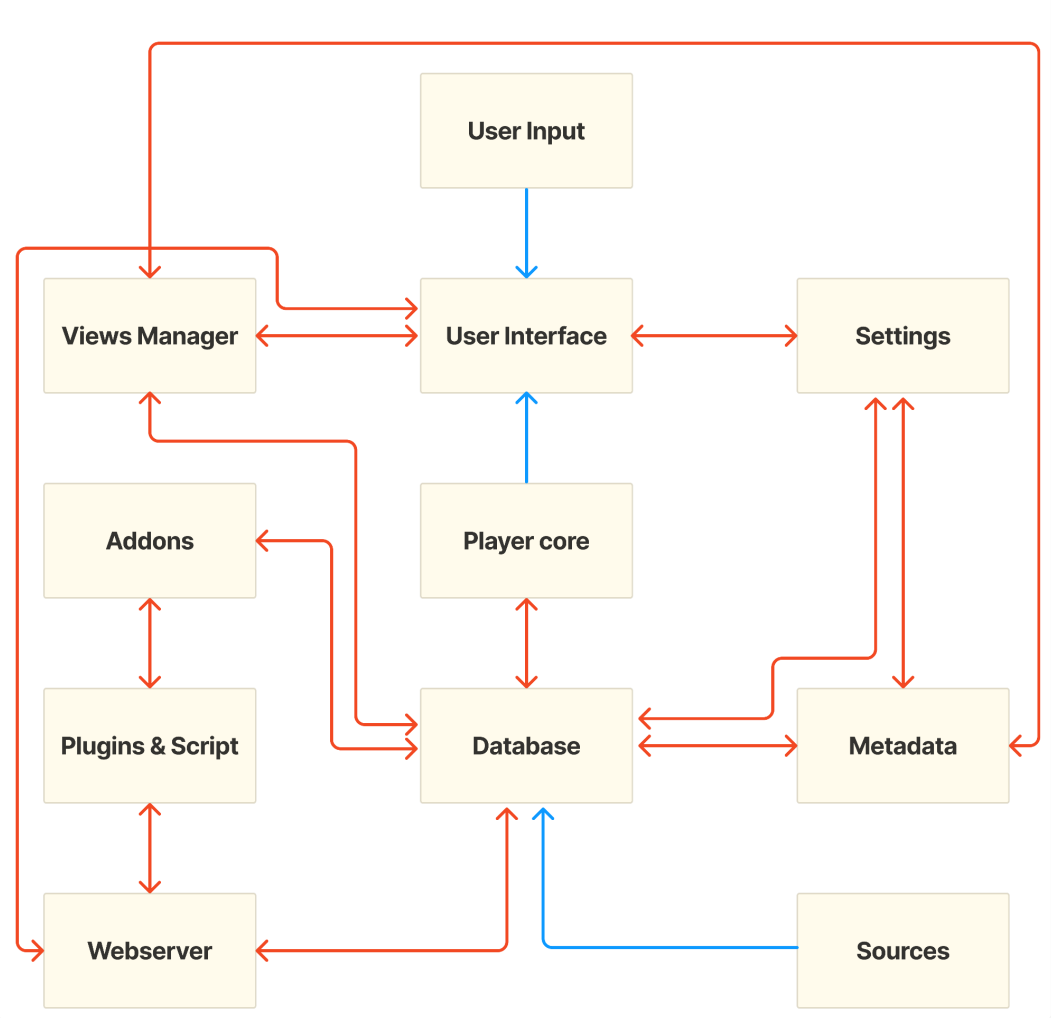
Web Server

The web server component provides users with a means to manage their content library remotely through a web browser or third party application. Located in the business layer, the web server also helps users access previously created addons and send them through to the Plugins and scripts module.

Sources

The Sources component allows users to manage and access media content. It is made up of all of the local media content that the user can add into Kodi. It gathers media from various local sources, such as the hard drive, DVD players, and CD players. The Sources component also allows users to access network protocols such as HTTP, FTP, etc. to gather and play multimedia content. It is the backbone for the application and it feeds all of the content it accesses into the database component. It is also located at the lowest level of the architecture in the data layer.

Component Interaction Diagram



Main architecture style

Architectural Styles:

Repository Style

First, we have the central SQLite database in the architecture where all the application's data is stored. Playercore, metadata, addons, settings, and view manager all interact with the database to access the data. We can also see the clear separation between the data access layer (which includes metadata) and the database. The database is used by the business layer; the main repository acts as the central database. Secondly, upon inspection using the Understand app, we found that the repository provides an interface that abstracts the underlying data storage operations used to access the data by other components. Finally, the rest of the application is shielded from the specifics of the data storage implementations. For example, it doesn't matter what computer Kodi is installed, it will automatically detect the data sources within the computer and hence it is flexible in switching or upgrading the underlying data storage.

Layered Style

Our argument that the main architecture of Kodi is a layered style comes from the hypothesis that all the components can be arranged in the following layers: Client, Presentation, Business, and Data. Each layer specifically encapsulates tasks and combines to provide the full application. The Client layer interacts with Kodi using the services provided by the Presentation layer. The Presentation layer includes all the components that involve a user interface, as well as an interface to write plugins/scripts. All the other GUI components fall under this layer. The Business layer involves all the important logic and operations of Kodi while supporting the Presentation layer. It includes add-ons, web server,and player core components. This layer uses the services provided by the Data layer, containing the metadata and files, to carry out different operations and execute logic. The Data layer includes the main storage database that encapsulates the data and data operations while providing services to the Business layer.

Object-Oriented Programming Style

Upon inspecting the Understand files, we saw how classes played a big role within Kodi. All the components are represented by a class and have separator methods such as "GUI renderers" and "CRUD operations". We further found many classes using instances of objects or class methods defined in other classes. For example, methods and objects from the Music and Video classes are used in the VideoPlayer class. This demonstrates that Kodi uses encapsulation and abstraction, core Object Oriented Programming (OOP) attributes. Additionally, inheritance can be observed among classes like Videoplayer, Retroplayer, and PA player, which provides further arguments that the OOP architecture style is one of the main styles in Kodi.

Client Server Style

We deduced that the concrete architecture also includes client-server style architecture, since there is a Web Server component. The web server would need to fetch data from a server to retrieve data essential to some functionalities of Kodi. If a user requests to download an addon, the request comes from the client side through to the web server. The server then fetches the data, which allows the user to download content such as videos, pictures, games, and audio back to the client side. This also allows the user (or multiple users) to manage their libraries and software remotely.

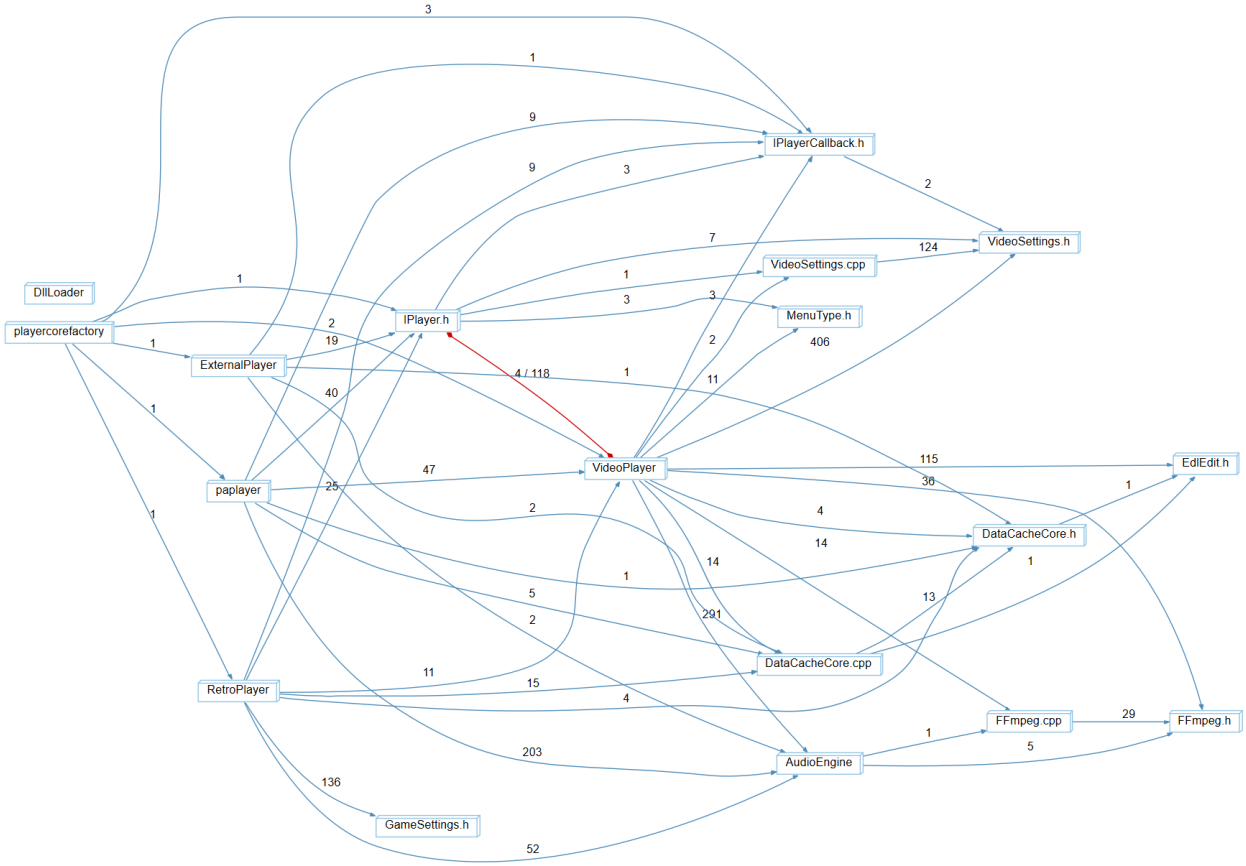
The Player Core Subsystem

The player core subsystem is one of the most important parts of the Kodi codebase. Being made up of several very important components such as VideoPlayer, AudioEngine, FFmpeg, and the PaPlayer, the player core is the project's universal hub for playing all kinds of media. It does this by being fairly low level, located in the business layer, and is connected to two of the other main components in Kodi. The cores module, as it is known in the file system, has direct references to the tools and lib directories - meaning it is used from the user input with tools, all the way to the operating system with lib. According to the Kodi Architecture Wiki, some specific tasks the player core is responsible for are parsing multimedia files into file formats recognizable to Kodi, synchronizing the media correctly such that the multimedia content does not contain delays in the audio or video, and managing the playback flow such as pausing a multimedia content and outputting audio and video to the user.

Looking into the player core component, it appears that the most important sub component - measured by number of references - is the video player, which itself contains numerous sub components. The video player, true to its name, mainly takes care of the playback and management of the video components. This purpose is evident as the majority of the references from this component go to the lib component, which is responsible for interfacing with the hardware and the operating system, implying a need to display content. Some of the sub components of video player further lend credence to this idea, with names such as DVDInputStreams and VideoRenders. Similarly, the AudioEngine component manages audio playback. There is not much evidence in the code base to support this, but it is known that the player core handles audio and there is only one audio related component here so it is a reasonable assumption. There are also a few links from the AudioEngine to the lib component, again implying a necessity to output audio to the operating system.

One of the main benefits of Kodi is its ability to handle many different types of media and play them, without slowdowns or issues. The two components responsible for this feature are FFmpeg and PaPlayer. FFmpeg, which stands for “Fast Forward Moving Picture Experts Group” is a free and open source software that is able to parse and translate many different media types into another. In Kodi, this software is used to translate given media files into formats usable by

Kodi. The other component mentioned, PaPlayer, focusses more on stability. PaPlayer, which stands for “Pulse Audio Player,” lets users have normal video & audio playback in Kodi while using a browser or other applications. It also allows Kodi playback of video or audio to be paused in order to run a game or another sound or video source.



Dependency Diagram of Player Core

Reflection Analysis of Player Core

New Subsystems:

Player Core

The player core was a subsystem that we did not really take into consideration while considering the conceptual architecture. Rather, we assumed that the parts contained within the player core would be their own components. To that end, the player core itself is a new subsystem

FFmpeg

While we did know that Kodi would have some system that would be able to convert multimedia file formats to media playable by Kodi, we had no specific subsystem for this in the conceptual architecture. After analysis of the the player core component, the FFmpeg is a significant component that is in the concrete architecture

PaPlayer

In the conceptual architecture, the stability of the media player regarding the quality and the timing of the playback was not considered at all. Having now seen it in the concrete architecture, we realise that this component is significant to Kodi.

New Dependencies

VideoPlayer → lib

One dependency that we did not consider was that the video player and decoder objects would need to have access to the operating system files stored in the concrete architecture's lib directory. This dependency would need to exist because the player components need to know what type of monitor to, and how to display it.

Tools → VideoPlayer

Another dependency we saw in the concrete architecture that we did not anticipate was that the core component, specifically the VideoPlayer, would be referenced by the tools component. This connection makes sense, as the VidePlayer would need to know when the user clicks, and on what - information which comes from the tools component since it acts as the user input of Kodi.

Removed Dependencies

Addons → Player Core

We initially assumed that the video players would need access to the addons so they could play different media types, but after examining Kodi's source code, this does not appear to be the case.

Discrepancies Between Conceptual and Concrete Architecture

Our Conceptual Architecture

Our group believed Kodi's fundamental conceptual architecture was the layered architecture. We broke Kodi down into four main layers; the data, business, presentation, and

client layers. The data layer stored all relevant media files, metadata, user preferences, etc. We defined the business layer as the layer that handles all the background functionality of the app (logic to make it work). The presentation layer was defined as anything relating to the GUI and UI, which the user would interact with. Lastly, we defined the client layer to categorize Kodi's addons, and other server based features.

Besides the layered architecture, our group also identified that Kodi used a repository architecture and a client/server architecture. This was because the database would use a centralized repository (addons, some media, plugins, etc.) and the repository was held online. It also explained the Kodi feature which allowed remote management of the system by one or multiple users.

Similarities

Kodi does have separate layers that are similar to the ones we defined in our first report. We broke Kodi down into data, business, presentation, and client layers, which lines up with the observed concrete architecture. Our group also identified Kodi's repository architecture. We observed the repository provides an abstraction layer to its data that other layers can use. The last similarity between our conceptual architecture and the observed concrete architecture is the use of a client server architecture style. The observed client/server architecture was exactly what we described in our first report; it allows for the use of addons, plugins and remote management.

Differences

There were many differences between our conceptual architecture and the actual concrete architecture. Kodi has a heavy reliance on the Object Oriented Programming architecture style, utilizing classes, methods, and instances to provide protection and abstraction for its data. Besides the layered architecture, Kodi is broken down by subsystems. While they can be categorized by the layer architecture style, it is notable because they all interact with each other and it is not a very strict hierarchy. The concrete repository architecture also had more functionality then what we had written about in our first report. It provides an abstraction layer for other components to access its data in a safe way. The repository is also implemented in a way that the user's hardware and OS do not affect its functionality.

Reflection Analysis

Metadata

In our conceptual architecture, we did not realize that metadata would need to be a component at all. After analyzing the concrete architecture in Understand we found that each of the media types: video, pictures, music, and games had their metadata inside each piece of content in the form of tags. This metadata provided users with information about the titles in their library, meaning the GUI component was dependent on this component for said information. The metadata was also used to query the database to find media content that the

user wanted to enjoy. The database was organized by the tags to provide users with relevant media options. Therefore the views manager also needed access to the metadata to know how to query the database for the types of media the user wanted. Lastly, the database itself stored tags and so it needed to connect with the metadata component and vice versa to have tags stay consistent in the metadata and the database for accurate querying.

Graphical User Interface

We previously believed that the GUI would be its own separate component which would house the different graphical elements and components needed to render onto the screen, handling the different devices, screen types, and user systems. After analyzing the concrete architecture, we realized that this was not the case. The GUI system worked more in an OOP style where each component had GUI elements housed within each sub-system. For example, the video directory had some video GUI elements, as well as the games directory. It's important to note however that there was also a "guilib" directory that housed a majority of the GUI elements, it's just that not ALL elements were in it.

Incorrect link between settings and database

Another small anomaly that was discovered in the architecture was the interactions between the settings component and the database component. In our conceptual architecture, we envisioned that the setting component would save its data locally, and wouldn't interact with the database directly, however, there were dependencies between them. We imagine that the settings link to the database to save user-specific data such as skins, colors, fonts, etc.

GUI depending on Player Core

One unexpected link we found in our concrete architecture was between the GUI and the player core. When we drafted our conceptual architecture we had our GUI component very isolated in the front end with many components between it and any of the media players. Once we looked at the dependencies of the actual architecture we realized that the player core would directly output rendered media to the user interface and not have to go through intermediary components.

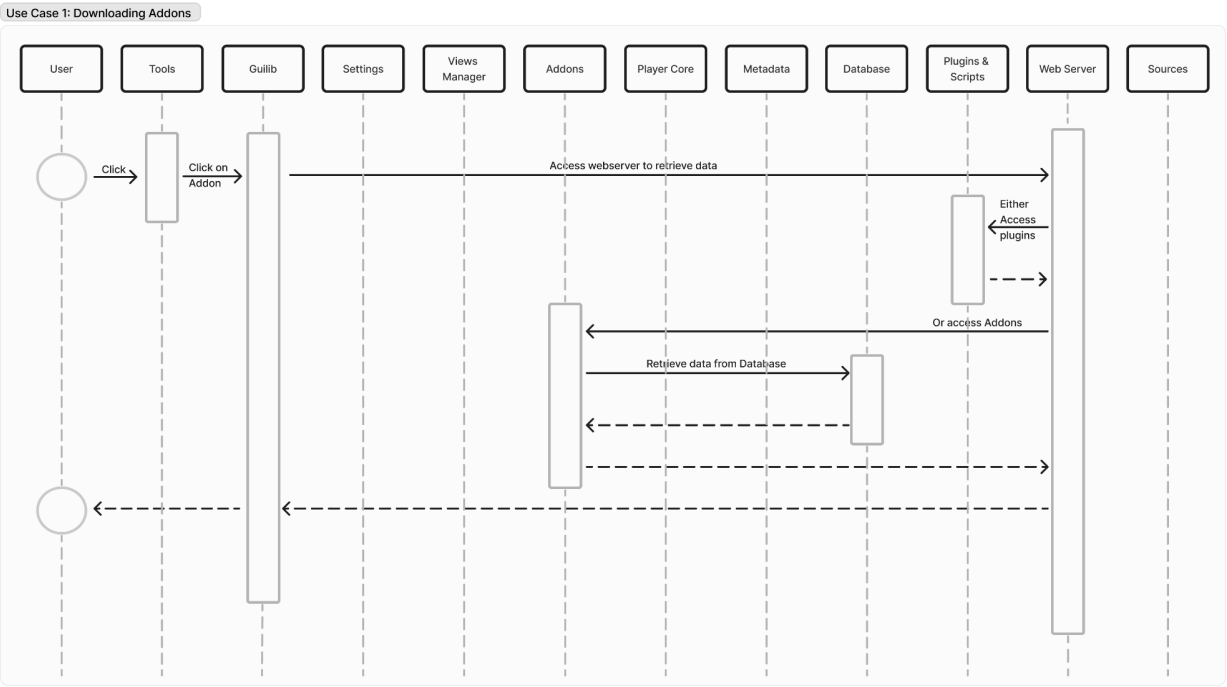
Web Server depending on Database

We found that the Web Server Component depended on the Database component to modify the library remotely. Kodi allows users to update their media libraries remotely through a web browser. Therefore there needed to be a two way dependency between the database and web server both for users to access their library remotely and make changes to it.

Use Cases

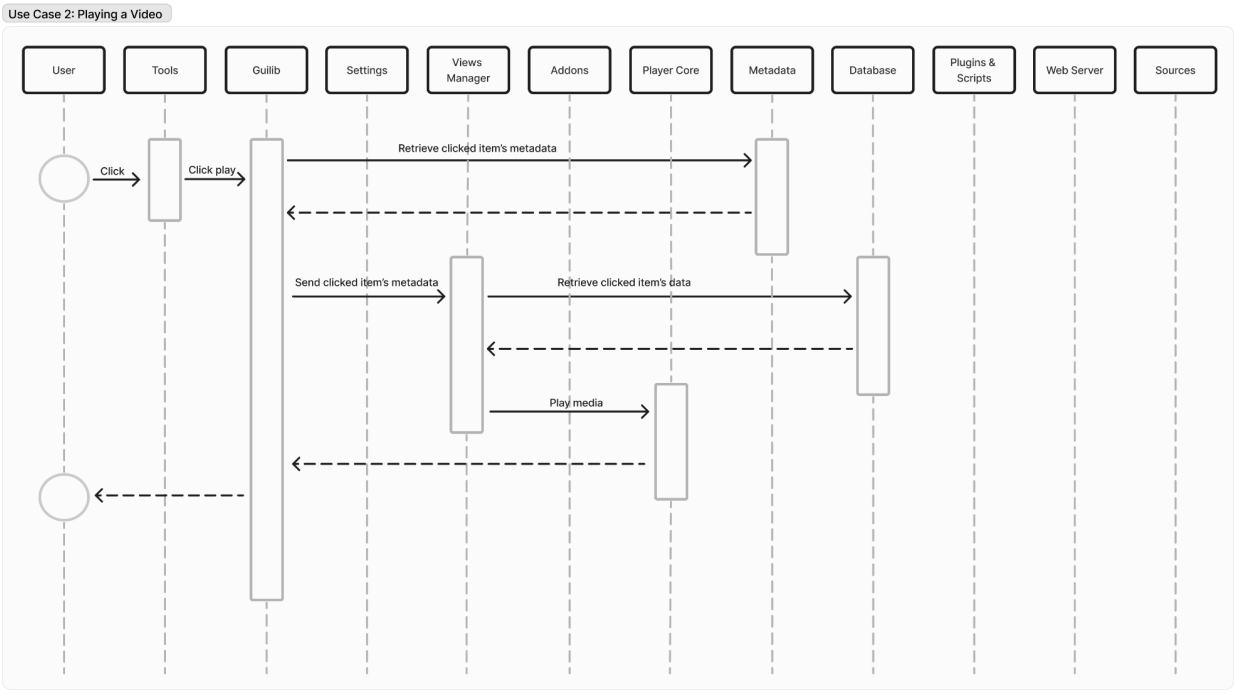
To further understand the above architecture, we explore two use cases that traverse through most of the sub-systems previously described. The first use case is for if the user wants to add additional add-ons. The other use case we are examining is playing custom media that is already installed onto the device.

Use Case 1: Downloading Addons



In this use case, the user wants to download any piece of content that is available, either directly through the web server's plugin storage or in the database, and move that content to their local storage for offline use. The user only needs to select the download button on any piece of content to initiate this process. This is accomplished specifically using a peripheral device such as a mouse, shown here under the Tools header. From there, the UI element Guilib will access the webserver with the necessary data and begin waiting for its response. The web server then processes the given data and decides if it is available immediately through the plugin storage or if it is in the database. If the data is a plugin, it will simply retrieve it and send it back. Otherwise, the web server will access the Addons component. The add-ons component will then access the database and send it back to the web server. Finally, when the web server has everything that the UI requested, it sends it back to be displayed to the user.

Use Case 2: Playing a Video



In this use case, the user is wanting to access one of their locally stored pieces of media, in this case, a video, and play that using the media player. To begin with, the user will select the play button on their video using a peripheral device. The UI element will parse this click and determine what was clicked on. Next, it retrieves the metadata about this video from the Metadata components and sends that data to the Views Manager. The Views Manager takes this data and retrieves the actual video file from the Database which it then sends to the Player core to be played. The Player Core then plays the media to the UI.

Lessons Learned

After a comprehensive analysis of the Kodi architecture, our team gained insight into the implementation, design, and operation of how an all-encompassing multimedia playing software works. We examined the differences between conceptual and concrete architectures to gain a new perspective on the reasons why Kodi behaves the way it does. OOP Style and modular design are important in a complex system like Kodi, separating the components and functionality, improves ease of system management and maintenance, as well as component reusability. We also identified the importance of doing our research, to analyze the concrete architecture we had to dive deep into the source code and architecture to see how each component interacted, and was dependent on one another. This took time, and in the future, we will be allocating more time to it.

Conclusion

We derived our understanding of Kodi's concrete architecture by building on top of our conceptual architecture knowledge. We matched the folders and classes of the code to different components and layers. The "Understand" application was a massive help, as it allowed us to see a breakdown of the system by components and function. The dependency diagrams helped us visualize how all the subsystems are interconnected. The main systems we found were the User Input, User Interface, Settings, Views Manager, Addons, Player Core, Metadata, Database, Plugins/Scripts, Web Server, and the Sources subsystems. Most of them were self-explanatory. Kodi's concrete architecture styles were very similar to the ones we mapped out in our conceptual architecture report. Kodi can be broken down into a layered architecture, a repository-style architecture, a server/client architecture, and an object-oriented architecture. In our first report, we did not realize how much Kodi used OOP principles. It is a major part of the software and is how the different components interact and pass/protect information. We believed the GUI worked as its own separate layer, but it is actually spread out between different systems. Further, we did not know about the Metadata component, which plays a big part in the concrete architecture. It is used for displaying, sorting, and queueing different media.

Naming Conventions / Glossary

GUI - Graphical User Interface
UI - User Interface
OOP - Object Oriented Programming, a style that uses objects to represent data and to hide or pass on data to other objects.
OS - Operating system, the software that runs the computer.
Skins - A separate graphical view of the software for aesthetic purposes.
Media - Stored information in the form of audio or visual videos/sounds
Multimedia - Media that may come in many different file formats
Dependency - A connection between two pieces of software that implies one component relies on another

References

Architecture. Architecture - Official Kodi Wiki. (n.d.). https://kodi.wiki/view/Architecture

Kodi. GitHub. (n.d.). https://github.com/xbmc/xbmc