Kodi Concrete 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:

For this report, our group was tasked with constructing a concrete architecture for the Kodi software. We then had to compare our concrete architecture to our conceptual architecture that we previously created. When comparing the architectures, we found that they were extremely similar but there were some striking differences that will be further discussed in this report. Like the conceptual architecture, our concrete architecture featured a mix of an object-oriented style and a layered style. The main difference we found was that our concrete architecture uses six objects rather than the five we mapped in our conceptual architecture. We also thoroughly investigated the player core, one of the top-level subsystems used to develop Kodi. We broke it down into its separate parts, detailing what each part does and the use-case of the overall subsystem. This deepened our understanding of the concrete architecture greatly, providing us with a better understanding of how different components of a subsystem work together.

Introduction and Overview:

Converting from a conceptual design to a concrete architecture is easier said than done. As development continues, it often becomes clear that things will not pan out as perfectly as the design. Unexpected dependencies will start to rise up between subsystems and it is your goal as a developer to keep things decoupled and efficient. In the worst case scenario, the project could become a total flop as the scale increases.

While it is difficult to create a conceptual architecture, having a plan ahead of time makes it much easier. Being diligent with your conceptual design is rewarding as it gives you a much better idea of how the concrete architecture will be laid out. In our case, we had a very thorough conceptual architecture of Kodi that aided us in creating our concrete architecture. We didn’t follow it exactly, but it gave us an idea of the top-level subsystems we would need to organize the files into. We soon realized that the top-level subsystems that we were using would not encapsulate all the files, however. We then decided to add one new object to act as a “utility” object that contains files that can be used by all the other top-level subsystems.

To create the concrete architecture, we used a software called Understand to break down the file structure to make it more comprehensive. Seeing as Kodi is such a large project with about a million lines of code, this was essential to guide us in the right direction to create our concrete architecture. This software also allowed us to organize the files of Kodi into our concrete architecture right there in the same program that broke it down for us. We organized the files to the best of our ability using our best judgement along with our limited knowledge of the programming languages used in Kodi. The Understand software was then able to create graphs for our concrete architecture, giving us a visualization of what we just created.

This report consists of eight key sections. To begin, the abstract gives a brief overview of the report. Next, the introduction gives an overview of what Kodi is and what this report is composed of. The architecture section describes the concrete architecture of Kodi and compares it against our conceptual architecture. This features a dependency graph to visualize the relationships between various components. The diagram section gives an overview and contains any necessary information about the diagrams. Next is the external interfaces diagram which details transmitted information that is sent to or from the system. Following this, the use-case section provides 3 use-case examples of the objects and modules interacting with each other. Continuing on, the next section presents a data dictionary and clarifies any possibly confusing naming conventions. Finally, it wraps up with the conclusion and with a summary on what we learned. These sections detail the final results of our findings and discuss why this was beneficial for us as students.

Architecture:

Our concrete architecture turned out to be very similar to the conceptual architecture we previously found. We divided the architecture into six separate subsystems. We delved into the code to determine the use-cases for all the files and determined that these six subsystems were the most suitable for Kodi's architecture. The subsystems consist of the following: content management, file sharing, interfaces, player core, skins, and utils. The various subsystems have distinct functionalities, they collaborate to form the overall system of Kodi. Each of these subsystems are organized further, forming layers within each of them. This follows what we found for our conceptual architecture. It uses the same mix of architectural styles, an object-oriented style and a layered style. We will briefly explain the architecture and functionality of each top-level subsystem and we will delve further into the player core to further understand how these subsystems work.

The first subsystem is dedicated to content management. It contains files necessary for managing the media content in Kodi. It is organized to have three main subsystems to handle different tasks involving; metadata, sources, and views. Seeing as Kodi is a media player software, this subsystem is vital to its functionality. The next top-level subsystem handles file sharing. It contains files necessary for communicating with the web server to do so. It couldn't be divided much further at a higher level, however, the next top-level subsystem could be. The interface subsystem can be broken down into three subdirectories. The subdirectories are the following: application instance, configuration, and python. They handle different functions, however, the biggest of the three is the application instance subdirectory. Kodi was designed to be run on many different types of devices, this subsystem is vital to ensure that it runs smoothly on any device. It contains files to handle different platforms or operating systems. Continuing on, the skins subsystem allows for a lot of customizability by the user. It contains files related to the GUI and user input, things that the user may want to change. This enables the user to create or download skins that completely change the design of Kodi. This is one of key components that has contributed to Kodi's long-lasting success. The utils subsystem doesn't have one clear function, rather, it contains files that support different functionalities that may be used by any of the other top-level subsystems. The final top-level subsystem, and the one that we will be thoroughly examining, is the player core.

The player core is arguably the most important of all the top-level subsystems. It is necessary for viewing or listening to media, the primary focus of Kodi. It can be broken down into the different player cores, so the highest level consists of a subdirectory containing the different cores. This subdirectory contains three further subdirectories all dedicated to different aspects of the media player. They include the AudioEngine, the RetroPlayer, and the VideoPlayer subdirectories. These subdirectories each handle different tasks to comprise the main functionality of the player core. Going further, the functions of each are as follows. The audio engine, as it says in the name, contains files required to handle any audio aspect of the video player. Continuing on, the retro player contains files that enable the user to play retro games using Kodi's software. Finally, the video player contains codecs, handles video rendering, and contains files that control the overall process. A visualization of the architecture of the player core can be seen in Figure 1.

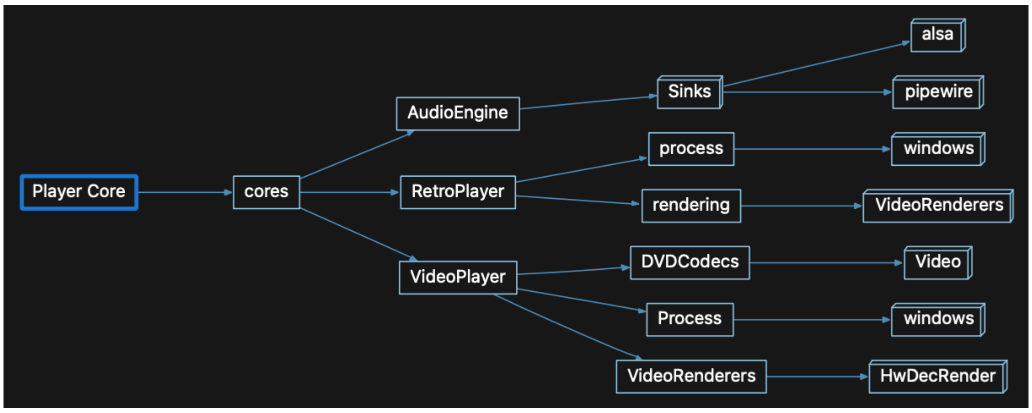


Figure 1

While our concrete architecture was similar to the conceptual architecture, we did encounter some differences. Our conceptual architecture used both an object-oriented style and a layered style. According to our conceptual architecture, there are five separate objects to handle different functions of Kodi. Each object is then further broken down into layers. Each of the objects can then operate concurrently to carry out the various tasks required to run the program. The most striking difference we found between our conceptual architecture and our concrete architecture is the number of objects. While our conceptual architecture used five objects, we found that the concrete architecture uses six objects. It uses all the same objects, but additionally has a utils object. As previously described, this extra object serves as a sort of "helper" object as it contains small files that may be used by other objects. This was the biggest difference we found, though. The rest of the concrete architecture was extremely similar to our conceptual architecture.

Reflexion Analysis: High-level Architecture

Conceptual dependency diagram:

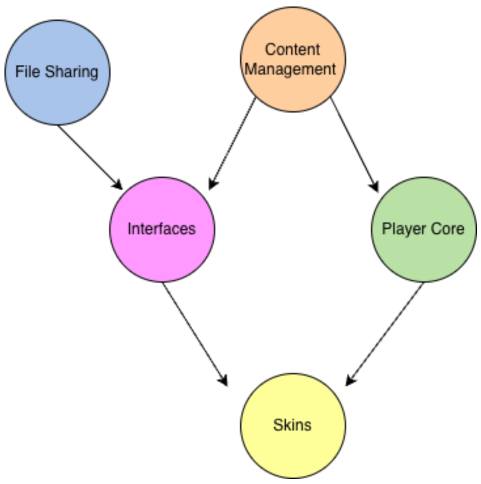


Figure 2

Concrete dependency diagram:

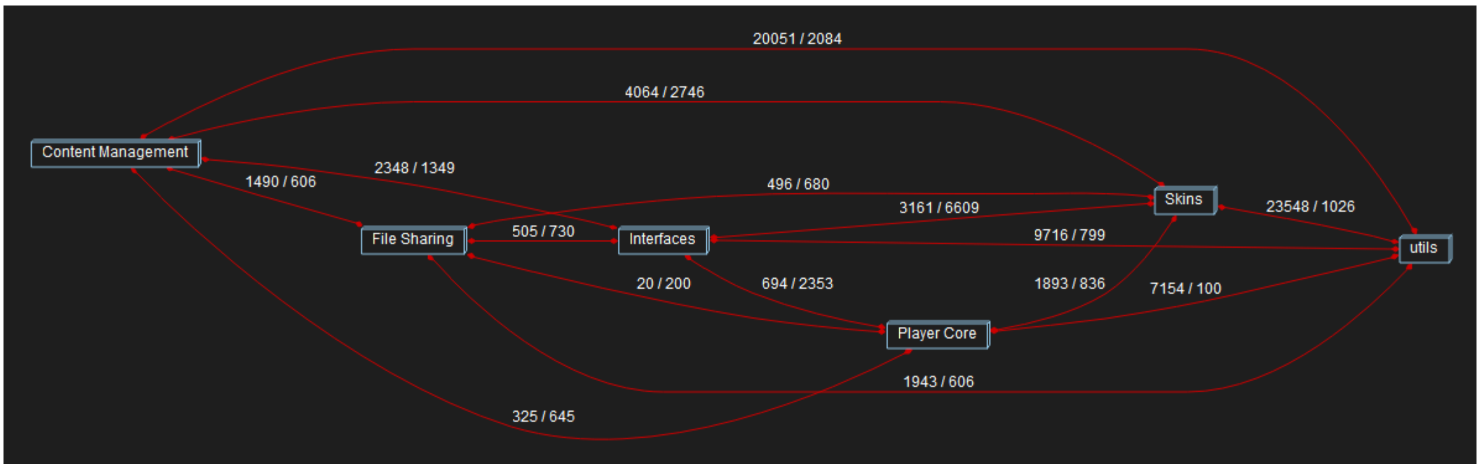


Figure 3

As shown above, there are some clear differences between the concrete and conceptual diagrams. In our conceptual dependency diagram (Figure 2), all the dependencies are 1-way and only Interfaces and Skins have more than 1 source depending on them. However, after using Understand and deriving our concrete architecture, it is visible through the concrete dependency diagram (Figure 3) that there are a greater number of dependencies than previously thought, as well as the introduction of a new “utils” subsystem. In the concrete diagram, every subsystem is dependent on and is depended on by every other subsystem. As a result, the divergences from the conceptual architecture through unexpected dependencies are seen below.

Source	Depends on	Rationale
Content Management	utils	utils contains essential functions for Content Management.
Content Management	Skins	When skins are changed, they must be loaded into user preferences and managed by Content Management.
Content Management	File Sharing	When new content is loaded from file sharing there are files in content management that will store this information.
File Sharing	utils	utils contains essential functions for FIle Sharing.
File Sharing	Content Management	Files are downloaded locally from external databases.
File Sharing	Skins	When new skins are added, they are saved to the file system.
File Sharing	Player Core	Files related to video playback such as external video players are downloaded and run.
Interfaces	utils	utils contains essential functions for Interfaces.
Interfaces	Content Management	Scripts and libraries for accessing metadata from databases.
Interfaces	File Sharing	Scripts and libraries from the file system are run.
Interfaces	Player Core	Scripts and libraries for running and utilizing video players.

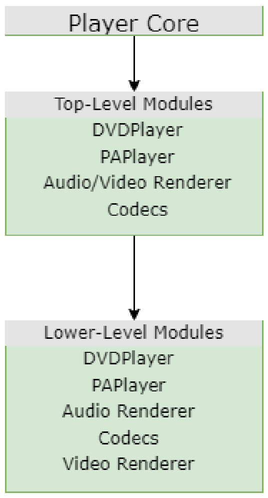
Player Core	utils	utils contains essential functions for Player Core.
Player Core	Content Management	The player core depends on Content management as it stores the data to read drives containing videos/audio and or the dvd reader.
Player Core	Interfaces	Video player uses add-ons and scripts/libraries for preferences and modifying video playback.
Player Core	File Sharing	Video player is run from the file system for video playback.
Skins	utils	utils contains essential functions for Skins.
Skins	Content Management	Content Management is where the data for different types of skins is contained.
Skins	File Sharing	GUI allows users to download/select videos and skins from the file system.
Skins	Player Core	GUI uses video player and codecs to render and display video.
Skins	Interfaces	GUI uses interface files to display the interface on Kodi main display.
Utils	Content Management	Certain essential functions in utils are only applicable when working with Content Management
Utils	Skins	Certain essential functions in utils are only applicable when working with Skins.
Utils	File Sharing	Certain essential functions in utils are only applicable when working with File Sharing.
Utils	Player Core	Certain essential functions in utils are only applicable when working with Player Core.
Utils	Interfaces	Certain essential functions in utils are only applicable when working with Interfaces.

Reflexion Analysis: Player Core subsystem

Dependency Diagrams:

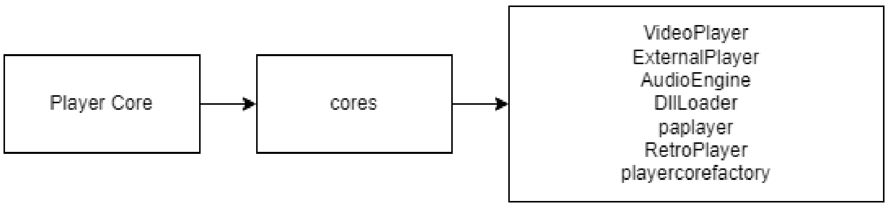
Conceptual Architecture:

Figure 4



Concrete Architecture:

Figure 5



As shown above, the way the concrete architecture (Figure 5) is organized differs slightly from our conceptual architecture (Figure 4), based on the *Development View* analysis from Delft Students (Dreef, K. et al.), although the subsystems contained within Player Core allowing for video playback are as expected. This includes further subsystems such as VideoPlayer and ExternalPlayer, as well as other files for saving and modifying user settings and preferences for video playback. As a result, the divergences from the conceptual architecture through unexpected dependencies are seen below.

Source	Depends on	Rationale
Player Core	cores	Cores is a subfolder containing files as well as other subsystems pertaining to video playback.
cores	VideoPlayer	Contains multiple subfolders that allow the video to be decompressed, rendered, and processed.
cores	ExternalPlayer	Contains files for video playback via external video players.
cores	AudioEngine	Contains multiple subfolders that are used for competing multiple input/output tasks.
cores	DllLoader	Contains files for dynamic link library for code reuse and optimization.

cores	paplayer	Cores depends on the paplayer because it contains files that allow the audio to be played.
cores	RetroPlayer	Cores depends on the RetroPlayer as it contains files that allow the core to play games that can be added by using emulator addons.
cores	playercorefactory	The core depends on the playercorefactory when needing to use an external player .

Diagrams:

Figure 1:
- Line: Represents a connection between objects
- Square: Represents a location for files that help do what is on the box

Figure 2, 4-5:
- Arrow: Represents 1-way dependency

Figure 3:
- Red arrow: 2-way dependency, numbers indicate # of dependencies each way

Figure 6-7:
- Lifeline: Vertical Lines representing an object's timeline
- Message(Solid): Represents a synchronous communication
- Message(Dashed): Represents asynchronous communication
- Activation Bar: Box on object lifeline representing activation duration

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:

Use case 1: The user watches content saved on the local device.

One of the essential use cases of KODI is viewing content stored locally on the disk. The content management subsystem handles access to the disk as well as handles the different content “views” the user can access, in this case video (.mp4, .mov, .mkv). Here is a breakdown of each subsystem:

Skins:
The skins subsystem contains the objects that control user input and input dialogs. Here the program processes the input against the GUI. In this case, the user has decided on a file to load from the current library selection. Once playback has begun, this is the layer the user will invoke pause/play, and volume GUI elements from.

Interfaces:
Upon receiving the GUI input the interface interprets the input and decides what needs to happen. On clicking the selected video, the interface calls for the Content Management subsystem to load the content from the disk and pass control over to the Player Core.

Content Management:
When Content Management receives orders to load a particular file, it will access the hard drive, interpret the “view” in this case video and send it to the player core.

Player Core:
The player core decodes the video file and begins to stream the video. This subsystem holds control until the video is either stopped or has finished playing.

It’s important to note that the sequence diagram doesn’t include the Utils subsystem, but may access utilities for any tools that it may need to convert or parse through JSON or Boolean logic.

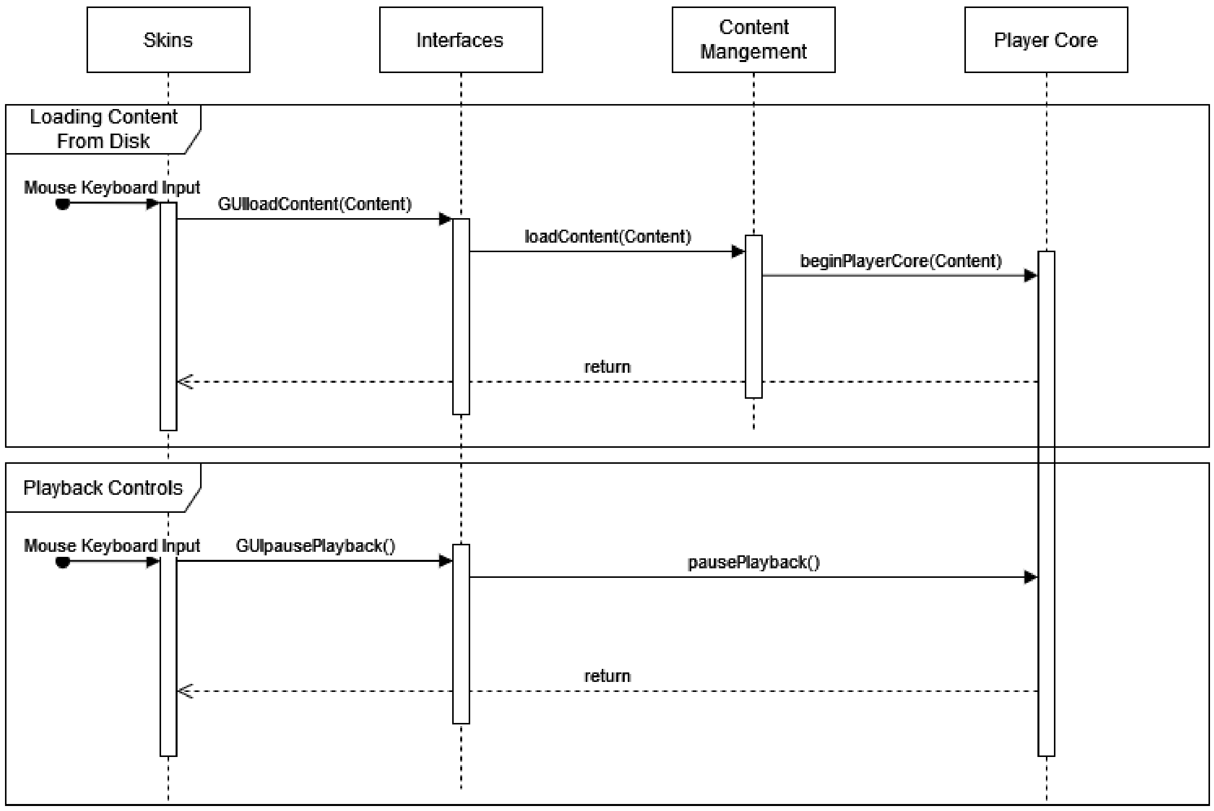


Figure 6

Use case 2: The user selects a device to stream content to & from.

Another one of the reasons for Kodi's success is the ability to stream content over your local network via a UPnP connection to multiple devices such as a mobile device. In this use case we'll look at two Kodi sequences in parallel and how to establish a streaming connection.

Skins:

Controls user interaction with the GUI. Similar to use case 1, the user can use mouse input, or touch screen input to select buttons, sliders, and other GUI elements.

Interfaces:

Interprets the GUI inputs and calls the File Sharing/Web Server subsystem.

File Sharing/Web Server:

This is the subsystem where the two devices communicate. In this use case they are establishing a connection and loading the currently available content that is able to stream. After establishing a connection, the user would then be able to select the content they would want to view.

Content Management:

On the streaming device, the Web Server accesses the available files to stream and sends that information back to the playback device. From there, the user would see the available content.

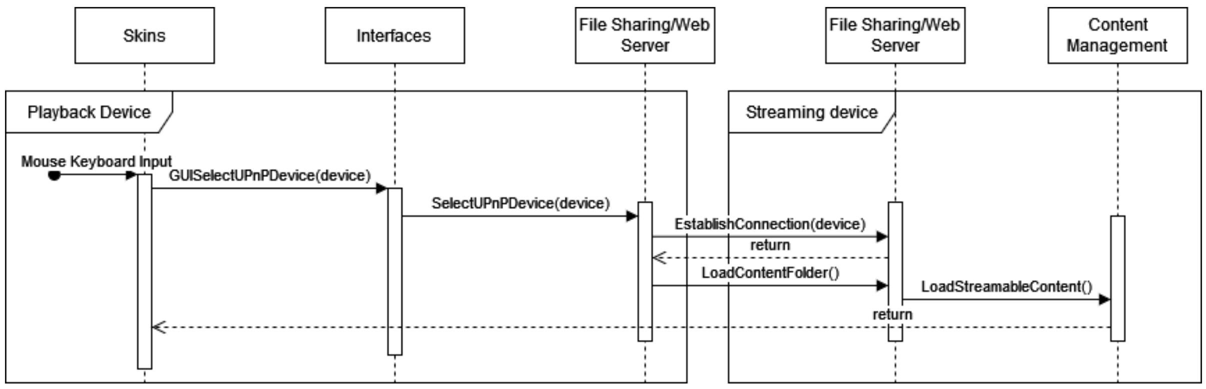


Figure 7

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.

Dynamic Link Library: A collection of small programs for use by larger programs to complete specific tasks.

Codec: A process that can compress or decompress audio or video data.

Metadata: A set of data that provides contextual information about another file.

JSON: A human readable, text-based data transfer format.

Boolean Logic: A method of calculating logical operations involving true and false statements.

Naming Conventions:

DLL: Dynamic Link Library

GUI: Graphical User Interface

UPnP: Universal Plug and Play

Conclusions:

Through our investigation, we found that Kodi’s concrete architecture is similar to its conceptual architecture. While they are similar, though, they do have some vitally important differences. They both follow the same architectural styles as they feature an object-oriented style and a layered style. Each object specializes in handling a different key function to comprise the overall system. Furthermore, each object can be broken down into different layers, each having different modules to assist in carrying out the object’s main function. By combining two architectural styles, Kodi retains the advantages of both. There are differences, however, between the conceptual and the concrete architectures. The main two being the difference in the number of objects and the difference in dependencies. To expand on these, our conceptual architecture featured five objects tasked with different functions whereas we found that the concrete has six. The concrete architecture contains the same five objects that the conceptual architecture has, but it additionally features a utilities object. This provides small functions for the other objects to use to perform specific functions. In terms of the dependencies, we found that the concrete architecture had significantly more dependencies than our conceptual architecture. Our conceptual architecture had each module relying on either one or zero other modules. On the other hand, our concrete architecture showed that every module depends on and is depended on by every other module. Overall though, the two architectures are quite similar. This is a great example of how projects end up in the real world, the planned architecture of a program may not always be the exact same as how the software turns out.

Lessons Learned:

This report gave us hands-on experience to deepen our knowledge of software architecture and allowed us to further build our teamwork skills. The group aspect of this project gave us all more experience working in a group format, something key for working in the software development industry. Furthermore, working on a large-scale project like Kodi gave us insight to what a real-world project looks like. Throughout the process of writing this report, many valuable lessons were learned. While formulating the concrete architecture, we learned that the number of unexpected dependencies were far greater than we previously expected. Essentially, this means that the modules rely on each other much more than we originally thought. While we knew they would be different, it didn’t really sink in until we saw the difference between the conceptual and the concrete architectures that we made ourselves. It’s just so different seeing the differences in a real example rather than just learning about it in a lecture. Finally, organizing and distributing the files of Kodi into various subsystems had a big learning curve that we had to overcome. To do so, we had to rely more on decision-making rather than our limited understanding of C++. Overall, this assignment gave everyone the opportunity to further develop their skills related to software architecture and gave us valuable group-work experience.

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/