KODI Concrete Architecture Report

CISC 322/326 - Group 4

Kabeer Adil - 20ma101@queensu.ca
Kate Edgar - 20kie@queensu.ca
Nicholas Tillo - 20njt4@queensu.ca
Raksha Rehal - 20rsr3@queensu.ca
Yu Xuan Liu - 20yxl3@queensu.ca
Xiyun (Victoria) Cao - 18xc17@queensu.ca

TABLE OF CONTENTS

ABSTRACT

The following report has been crafted to discuss the concrete architecture for the KODI multimedia entertainment application through analysis of its XBMC source code. This analysis exists to draw upon the differences between KODI's concrete architecture and our extrapolated notions of KODI's architecture from our previous conceptual report. After examining the dependencies of the top-level entities within KODI's system, it became clear that the conceptual architecture we drew up and the objective architectural implementation of KODI do differ in a few ways. However, something that remains clear is that it uses a layered-style architectural system with aspects of a repository style to bring together its overall structure. This report will also discuss the concrete subsystems within KODI, how they interact with one another, and how they differ from our previous notions regarding the software's design. We will do this via reflexion analysis for the high level architecture and through various sequence diagrams that utilize relevant use cases in order to illustrate our points.

INTRODUCTION & OVERVIEW

This report's purpose is to recover and document the concrete architecture for KODI. In the previous report, we covered the conceptual architecture, which is essentially a "big-picture" look into how KODI runs. Concrete architecture is more rooted in ideas of the actual relations present within the program's source code and configuration. Through the use of a code analysis tool called Understand, we were able to look at visualized mappings of the source code and its various subsystems and their dependencies, which our group then analyzed and reviewed. The first section of the report will go into detail regarding architectural styles, components and their many interactions; all of which are necessary for updating our understanding of the conceptual architecture which in turn allows us to recover the concrete architecture more definitively. Additionally, we will be recovering the concrete and conceptual architecture of a subsystem, and perform a reflexion analysis on top-level concrete architectures as well as their subsystems. The following sections describe the modifications made to our conceptual architecture and why these changes were appropriate given our added knowledge. Following that is a breakdown of some of KODI's use cases based on our updated understandings. Each of our use cases will feature a sequence diagram and a box and arrows diagram regarding the selected use case.

ARCHITECTURAL STYLE(S)

After analyzing the source code, we can confirm that the main architectural styles used in KODI are the same as we explained in our report of KODI's conceptual architecture. KODI employs an architectural structure that utilizes **layered** and **modular** styles. KODI has multiple separate modules that all deal with different categories of operations, and communicate outside

of their own module only on the topmost level. This ensures that the components are able to be replaced easily, added onto at a high-level, and can be utilized on their own. These top level modules are part of KODI's layered design and are used to distinguish interactions among subsystems and reduce dependencies that may impact the final modular design. With this structure, KODI is able to handle issues regarding scalability and portability more efficiently.

ARCHITECTURAL OVERVIEW

We are able to notice, then, that the overall structure of KODI is largely the same as we gathered in finding the conceptual architecture. The modular style holds, as the code directories are all sorted into corresponding modules, demonstrated below.

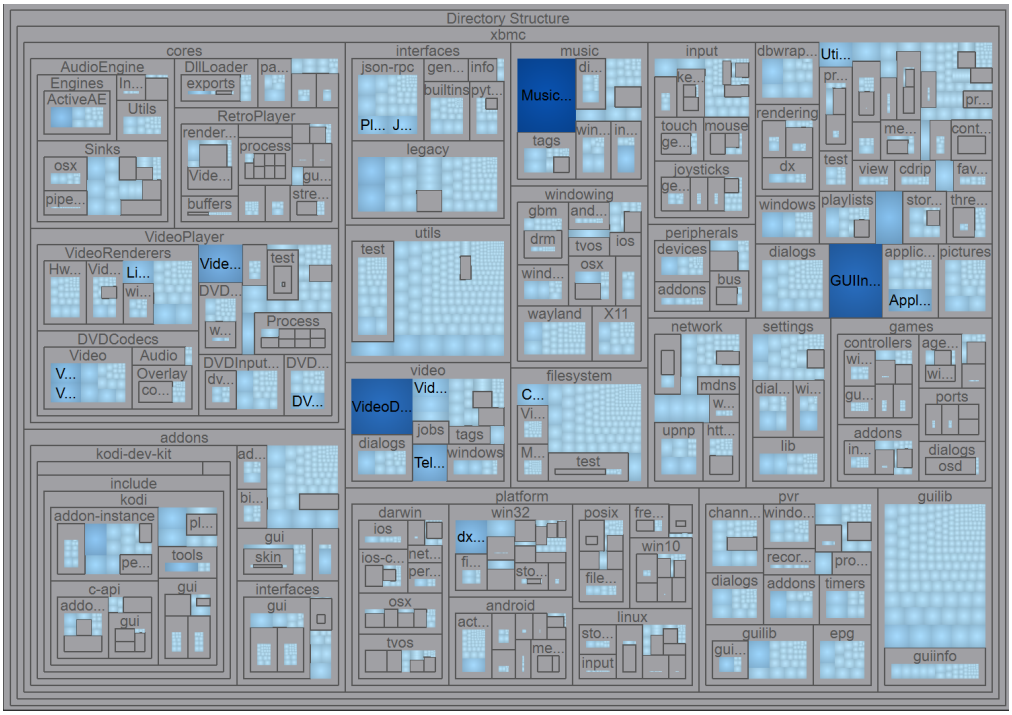


Figure 1: Understand treemap of the files in the KODI code base, sorted by directory. The colour gradient gets darker depending on the amount of code lines.

In our conceptual architecture, we defined KODI as being separated by layers of abstraction. While it is still true that the architecture is layered, the layers are not grouped by levels of abstraction, but rather just grouped by the different directories. We elaborate on this prospect in our reflexion analysis. For now, what this means for the concrete architecture is that there will be quite a difference in dependencies.

In order to derive the concrete architecture from the source code, we needed to group together the top-level components into categories of subsystems. Luckily, our conceptual architecture already covered several important entities of the system. Hence, we were able to tweak them by grouping them into more functional categories, the same way the directories of KODI are grouped. Our top level components generally stay the same. The biggest difference at this level is that we are able to be more specific when naming these groups. Our higher-level components are mostly the same, save from the fact that we neglected to add a few as we overlooked their significance in the conceptual architecture. To derive the top-level subsystems that were not present in our conceptual architecture, we followed a similar process of looking at the largest directories and their functionalities within the system.

Through this process, we grouped together similar lower-level components under our main subsystems. We were sure to explore each directory to ensure that we covered the most major features coded into the system. In order to derive the dependencies, we were able to select a "View Dependencies" option available for each of the directories visible in the treemap displayed above. This allowed us to observe both the internal and external dependencies of the system in a list-like format; from which we can scan it to determine whether any of the other subsystems are listed. This is elaborated on in our interaction breakdown section further below.

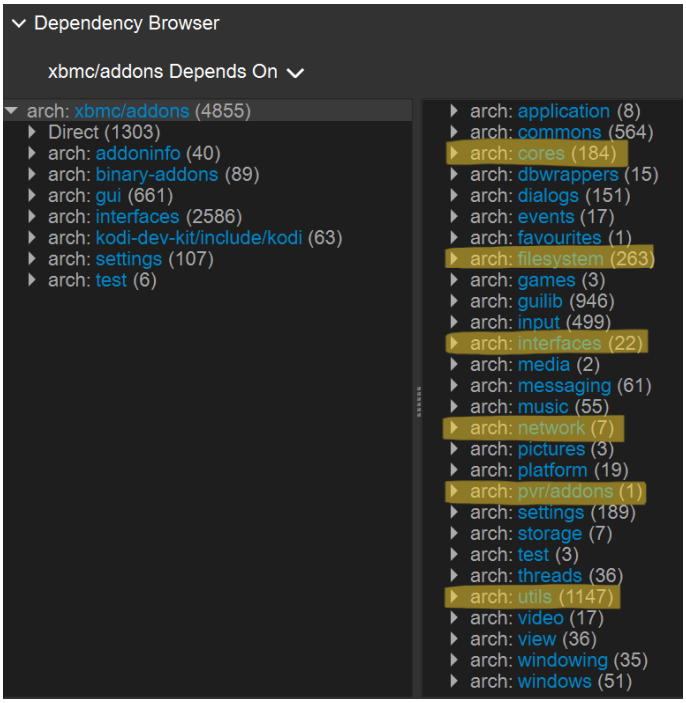


Figure 2: Example dependency browser for the "add-ons" subsystem, with its extracted top-level dependencies highlighted.

From this derivation, we are able to adjust our conceptual architecture to build the concrete architecture for KODI, displayed below.

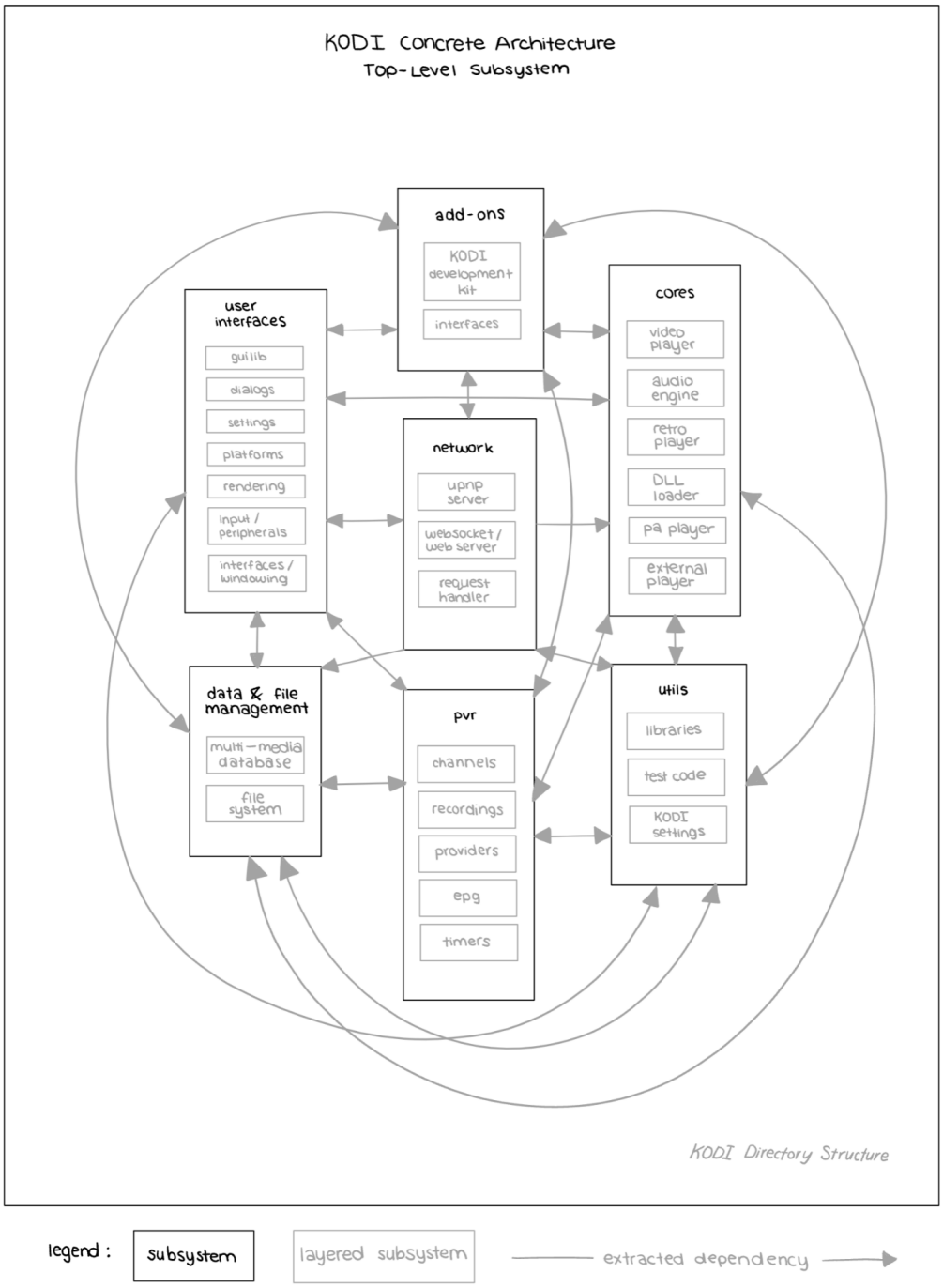


Figure 3: Concrete architecture of KODI at the top-most subsystem level.

COMPONENT BREAKDOWN

This section will elaborate into the functionality of each of the subsystems established in the concrete architecture that we found for KODI above.

1. User Interfaces

Handles the appearance of the system displayed to the user. Also responsible for i/o procedures.

- **Interfaces / Windowing** - skins in KODI are represented by different windows that are represented by .xml files [1]. The windows are all structured similarly, but are different depending on the operating system that they run on.
- **Input / Peripherals** - the external devices and equipment that can be attached to and interact with the system are called peripherals. Corresponding drivers need to be installed in the KODI system in order for it to work with input received through these peripherals. Examples of devices that are dealt with in code blocks of this subsystem are the mouse, keyboard, joystick controller, bluetooth devices. There is also a bus that is responsible for data transfer.
- **UI Settings** - there are a variety of adjustments and controls that may be changed via this subsystem… from GUI dialog, to window and display settings, to media and library adjustments, various aspects can be tweaked to the user's liking through the components in this subsystem. There is also an advanced settings file that can be manually edited to tweak technical issues [8].
- **guilib** - guilib is KODI's framework library for its GUI. It deals with all aspects that are appearance-related such as working with fonts, windows, text, buttons, etc.
- **Platforms -** a sector of code is dedicated for ensuring portability across multiple platforms including Android, osX, Linux, Windows, Darwin, etc. Each platform directory includes different thread and event handlers, depending on the OS. This section also includes Posix (Portable Operating System Interface) which contains helpful APIs and other standards to help enhance portability within KODI.
- **Dialogs** - these refer to a variety of prompts that pop up for user input, and have mostly to do with the way they are displayed in the GUI rather than the actual functions handled through these dialog boxes.
- **Rendering -** this subsystem specifically refers to ways to render the actual program of KODI itself. This includes multiple sections for OpenGL, OpenGLES, WindowsDX.

2. Add-Ons (Third-Party Integrations)

Third-party programs and features that are made to enhance user experience within KODI.

- **Kodi Development Kit** - includes many tools that are intended to help developers have an easier time writing add-on code for the KODI system [9]. This includes stubs (called Kodistubs) of code that help get the developers going, and include instances that deal with various KODI features such as its GUI, peripherals, APIs, settings, etc.
- **Interfaces** - premade interfaces are also given as tools for developers to work with.

3. Cores (Media Controller)

Various systems that constitute as media controllers depending on the media format.

- **Video Player** - responsible for reading and displaying DVD-video and Blu-ray discs, as well as videos from other popular video formats [6]. Also accounts for accessibility in video playback, including features such as subtitles and other quality characteristics.
- **Audio Engine** - handles the processing and output of all audio in the multi-media player, including mixing, encoding, upmix, processing, etc. [7].
- **Retro Player -** allows for the downloading and playing of old retro games through an emulated system [3]. This subsystem essentially combines together to act as a game player.
- **DLL Loader** - loads and manages DLLs (Dynamic-Link Libraries) to allow for other sections to use them. These libraries are a collection of smaller programs that are able to perform helpful tasks to be used in the bigger programs and sections of code that make up KODI [10].
- **PA Player** - a multipurpose audio player, developed by KODI themselves, in order to cover a wider range of files [5].
- **External Player / PlayercoreFactory -** users may need different playback software than is offered in KODI, especially for large quantities of content… hence the external player allows them to download and use KODI files and functionalities to assist with the player [11]. For example, an external player can allow for the use of a movie grade renderer, while still using KODI's software to scrape the internet for the file.

4. Data & File Management

A repository-style storage for both the core code of KODI, as well as its multi-media files.

- **Multi-Media Database** - the databases for videos, music, pictures, games, and playlists are all sorted separately in the directory, but for the sake of coherency in KODI's architecture, we have decided to group it all together under one database containing all the multi-media data.
- **File System** - Maintains files and source codes that are used throughout the system. The code is kept separate from the multi-media storage. Files for everything, from add-ons to events and resources are kept here.

5. Network (Connection Controller)

Handles the way that KODI connects to external entities such as the Internet or other users.

- **UPnP** - KODI can be used as a UPnP server or client, which means that users can either stream and share their media libraries with others on the network, or they can receive music and video from one of these servers [12].
- **Web Socket / Web Server** - KODI employs these client-server communication methods. The web socket allows the transfer of data to and from clients and the server in realtime, and the web server provides several HTTP services [13].

- **HTTP Request Handler** - contains code that is used in handling the implementation of HTTP servers. This includes handling features related to the web interface, JSON-RPC, web plugins, etc.

6. PVR (Personalized Video Recorder)

Records live TV and save it to the file system… used for recording and replaying TV programs.

- **Channels** - while KODI does not have an inherent TV-tuning function to offer a range of channels to the user, it has code that acts as the front-end of a live TV. These can be achieved via installing PVR add-ons (the client), PVR backend software (the server), and using a tuner card [14].
- **Recordings** - code pertaining to the acquisition and retention of the recordings attained by the PVR is provided within the PVR subsystem.
- **Providers** - this subsystem maintains a list of all the channel providers within itself,
- **EPG** - an Electronic Program Guide (EPG) is a menu-based system displaying current and upcoming TV programs [15]. This service again is not directly offered through KODI, but the system is equipped with the code to read and work with an EPG if added to the system.
- **Timers** - components offered in this directory work to self-automate recordings based on timers initiated by users requesting recordings… in other words, the code offered here is how the system knows when to start and stop the PVR.

7. Utils

General code utilities that serve in assisting developers with ensuring the functionality and quality of their code, as well as smoothing over coherence for the system itself.

- **Test Code** - test code is present throughout various directories of KODI. For coherency, we have grouped it together under the 'Utils' category. All major directories have their own unique test code contained within themselves, but there is also a specialized directory containing various test codes for the system itself.
- **Settings** - with specific emphasis on library, dialog, and windows settings, there are various control, display, and i/o aspects that can be adjusted to the developer's liking within the code contained here, for quality metrics.

INTERACTION BREAKDOWN

Looking back to the conceptual architecture and the proposed architectural style reveals a lot of the interaction between top-level components. Conceptually, the architecture dictated that all components interact solely on the upper layers of their implementation, with the lower layers interacting with the layers directly beside it. However, upon investigating the main source code

of KODI through Understand, we notice there are a much larger amount of dependencies than we proposed via the conceptual architecture.

Interactions Between Top-Level Subsystems

Extracted dependencies were gathered through looking at the "View Dependencies" option in Understand, These dependencies are not only direct function calls, but also object declarations, inherited methods, and constant variable referencing …

1. **User Interfaces** - Depends on all other top level subsystems.
2. **Add-Ons** - Depends on all other top level subsystems.
3. **Cores** - User Interfaces, Add-ons, Data & File Management, PVR, Utils.
4. **Data & File Management** - User interfaces, Add-ons, Cores, PVR, Utils.
5. **Network** - User Interfaces, Add-ons, Cores, Data & File Management, Utils.
6. **PVR** - Add-ons, Core, Data File Management, Utils, User Interfaces,
7. **Utils** - Depends on all other top level subsystems.

INNER SUBSYSTEM ARCHITECTURE (OF MEDIA CONTROLLER)

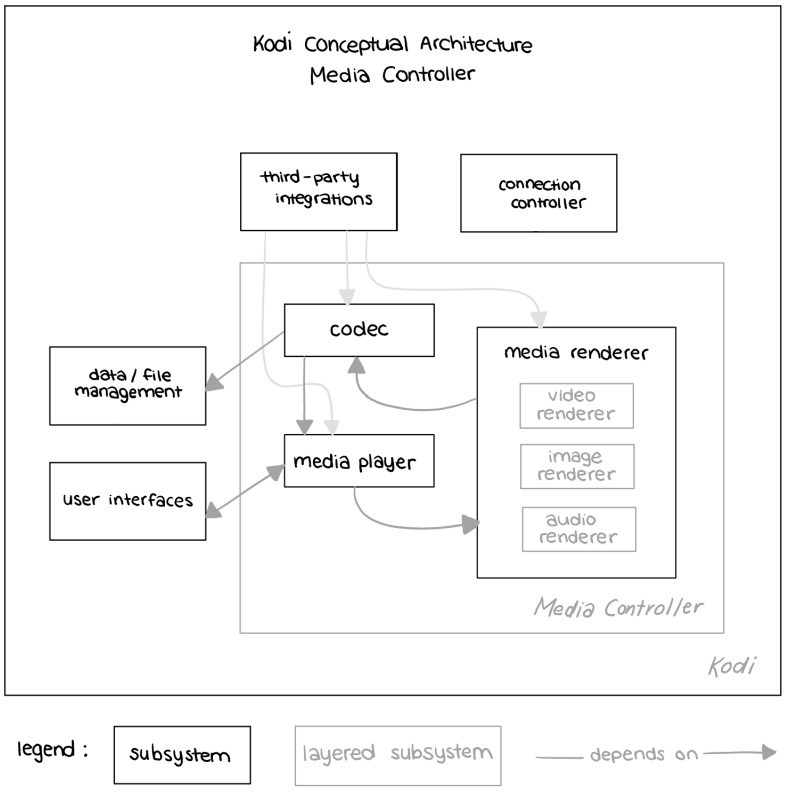


Figure 4: Conceptual architecture of the media controller component.

The main components within the conceptual architecture of KODI were the **Media Renderer**, to manage the rendering of every form of media, the **Media Player** to displays multi-media, and the **Codec** to decode the data signal into a usable form. These components have interactions that span to other components and components in themselves.

For the concrete architecture, these sections are able to be broken down deeper into subcomponents, gathered by inspection of treemaps, file layout and dependency graphs, Including Butterfly, Calls & Called By, Depends On, and Depended On By Graphs in Understand. This is supplemented by external research and forums created by KODI developers. Depicted below is the Call & Called By Graph, that was used most often to gather the dependencies.

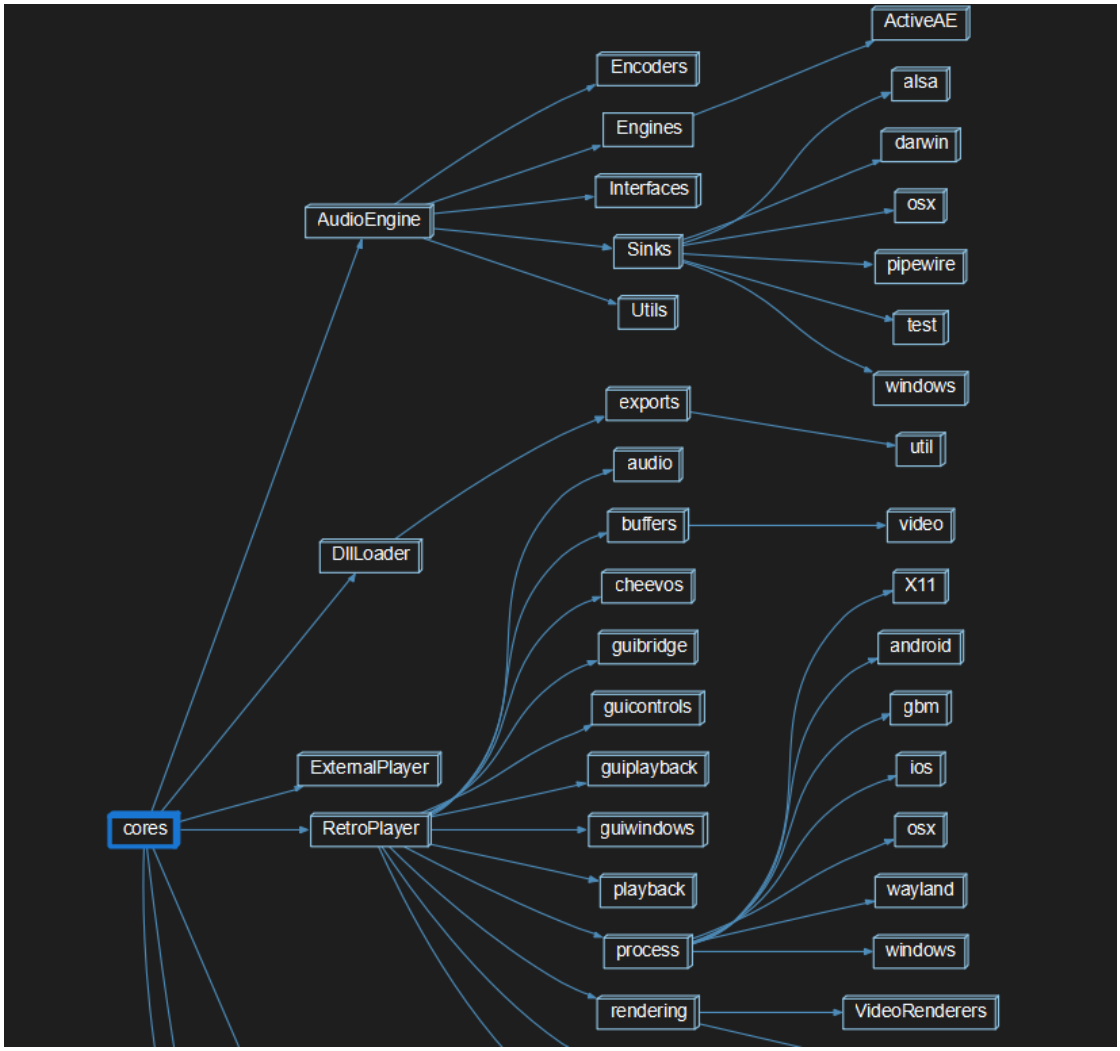


Figure 5: Understand "Graph Architecture" showing the makeup of the 'cores' subsystem.

Using this information, we can extract the concrete architecture. Dependencies of lower-level subsystems can be observed in detail in the descriptions below.

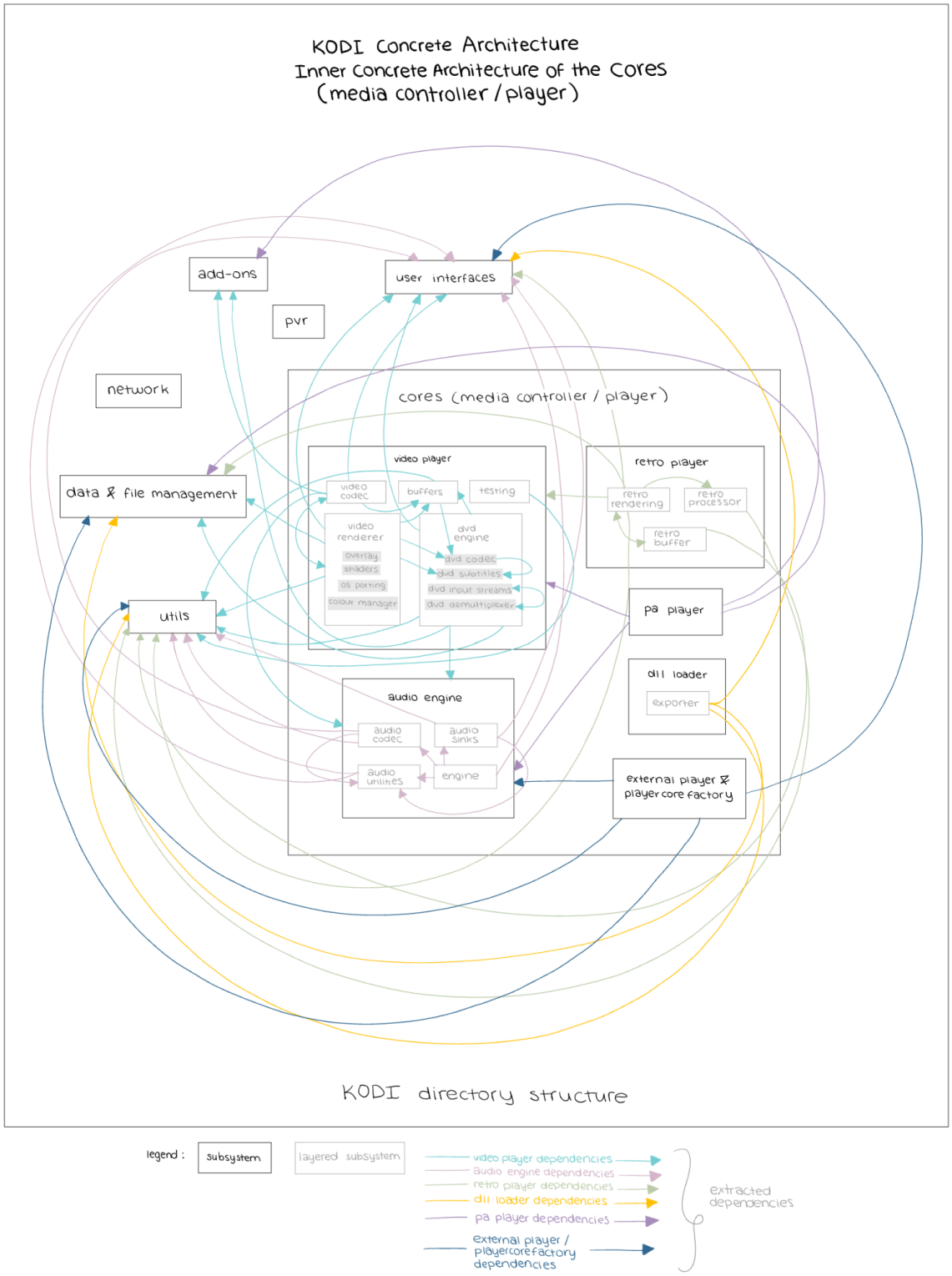


Figure 6: Concrete architecture of the cores 'media controller' component.

Cores (Media Controller) Sub-Components and Subsystems

1. **Video Player**
   Overall Dependencies: Audio Engine, Platform, Settings, Windowing, guilib, Rendering, Addons, File System, User Interface, Buffers,
   Subcomponents:
   i. **Video Codec -** Gathers and decodes the data stored from storage and turns it into a raw stream of data.
      Dependencies: Buffers, Addons, Video Renderer, Audio Engine, User Interface (windowing, guilib, rendering).
   ii. **Video Renderer** - Gathers the raw data stream and renders it into a video stream ready for display.
      Overall Dependencies: Video Codec, DVD Codec, DVD Subtitles, Buffers, UI (Rendering, Windowing, guilib), File System, Utils.
      Subcomponents:
      - **Shaders -** Uses many shaders techniques to help render the video. Conversion Matrices, Filters and Shaders are all employed.
        Dependencies: UI (Windows, Platform, Rendering, guilib), Utils, Filesystem.
      - **OS Porting -** Multiple files are used in order to maintain usability on many platforms.
        Dependencies: Buffer, UI (guilib, Rendering) Utils.
      - **Overlay**
        Dependencies: Utils, UI (Windowing).
      - **Colour Manager**
        Dependencies: Utils, File System, UI (Settings).
   iii. **DVD Engine -** Allows for the reading and displaying of DVDs on the KODI system.
      Overall Dependencies: UI (Windowing, Interface, rendering, guilib), Buffers, Addons, DVD Subtitles, Audio Engine.
      Subcomponents:
      - **DVD Codec -** Decodes the data signal that comes from the DVD into a usable form for the rest of the components.
        Dependencies: Buffers, Addons, DVD Subtitles, Audio Engine, User Interface (Windowing, rendering, guilib).
      - **DVD Subtitles -** Uses multiple subtitle formatting methods .
        Dependencies: User Interfaces (guilib), DVD Codec, Utils, Filesystem, DVD Input Streams.
      - **DVD Demultiplexer -** Inputs multimedia files into different streams (audio, visual, etc) and sends them to their correct inputs,

Dependencies: DVD Input Streams. User Interfaces (Platforms), Utils.

- **DVD Input Streams** - Controls input so that the user is able to interact with the DVD (For example, playing, Pausing, Title select screens).
  Dependencies: DVD Demultiplexer, Addons, Utils, Windows, File System, User Interface.

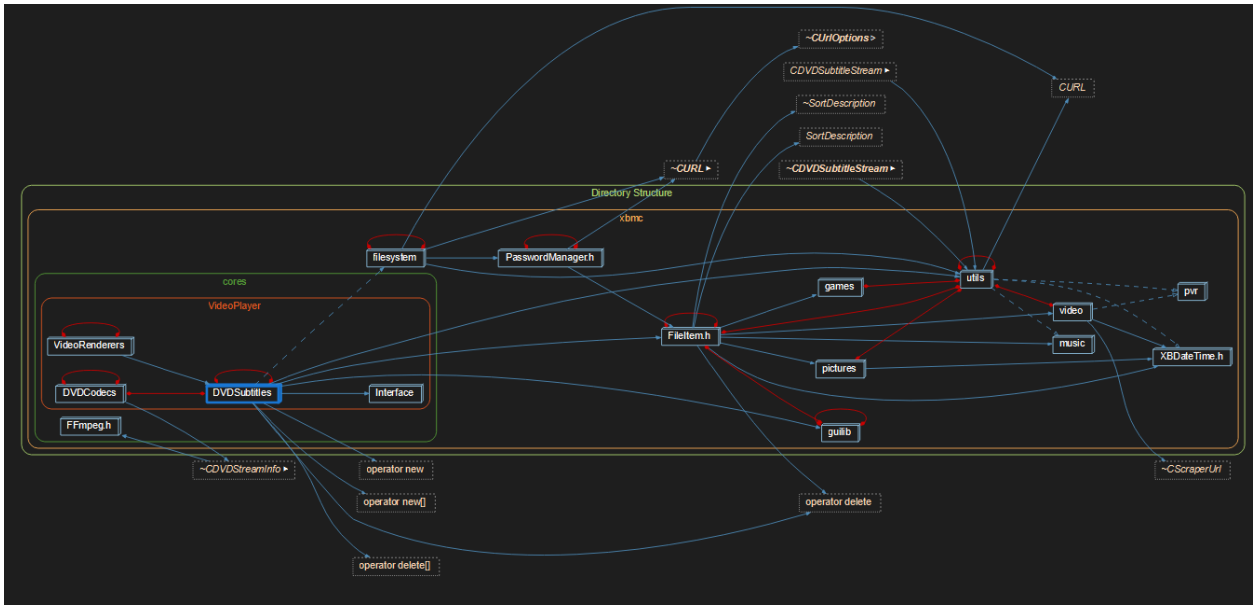


Figure 7: Understand "Calls and Calls By" Graph showing dependencies and component breakdowns.

iv. **Buffers -** Allows for the video player to be loaded ahead of time in order to maintain a smooth user experience.
Dependencies: DVD Codecs, Utils.

v. **Testing -** Holds and maintains a test suite that allows for swift testing of all functionalities.
Dependencies: Utils.

2. **Audio Engine**
Overall Dependencies: Windowing, Settings, Platform, Guilib, Video Player, File System, User Interfaces, Utils.
Subcomponents:

i. **Audio Codec** - Decodes the data signal into a usable form for the rest of the components.
Dependencies: Audio Utilities, Utils, User Interfaces.

ii. **Audio Sinks** - Manages all outputs of the audio to differing operating systems and output devices.
Dependencies: Platform, User Interfaces, Utils, Audio Utilities.

iii. **Audio Utilities** - Overall utilities for use in the Audio Engine.

Dependencies: Platform, Utils.

iv. **Engine** - Does the bulk of the functionalities of this section, handles all mixing, processing and playing.
Dependencies: Audio Codec, Audio Sinks, Audio Utilities. User Interfaces (Settings, Platforms, guilib).

3. **Retro Player**
Overall Dependencies: Audio Player, Utils, Games, Video Player, Settings, Games, Windowing.
Subcomponents:
   i. **Retro Rendering** - Acts as a video renderer for the processed games.
   Dependencies: UI (Rendering, Settings, Windowing) Retro Buffers, Data & Files (Games) Retro Processor, Video Player, Utils.
   ii. **Retro Processor** - Acts as the emulated game system so retro games are able to run.
   Dependencies: Utils.
   iii. **Retro Buffer** - Similar to the video buffer, it allows the retro player to process and load ahead of time in order to maintain a smooth user experience.
   Dependencies: Utils, Retro Rendering.

4. **PaPlayer**
Dependencies: Audio Engine, File System, Addons, Video Player.

5. **DllLoader**
Overall Dependencies: File System, Platform, Utils.
Subcomponents:
   i. **Exporter**
   Dependencies: File System, Platform, Utils.

6. **External Player & PlayerCoreFactory**
Dependencies: Audio Engine, User Interface, (Windowing, guilib) Utils, File System.

USE CASES

1. Kodi will seamlessly resume playback of the last watched live radio channel upon startup

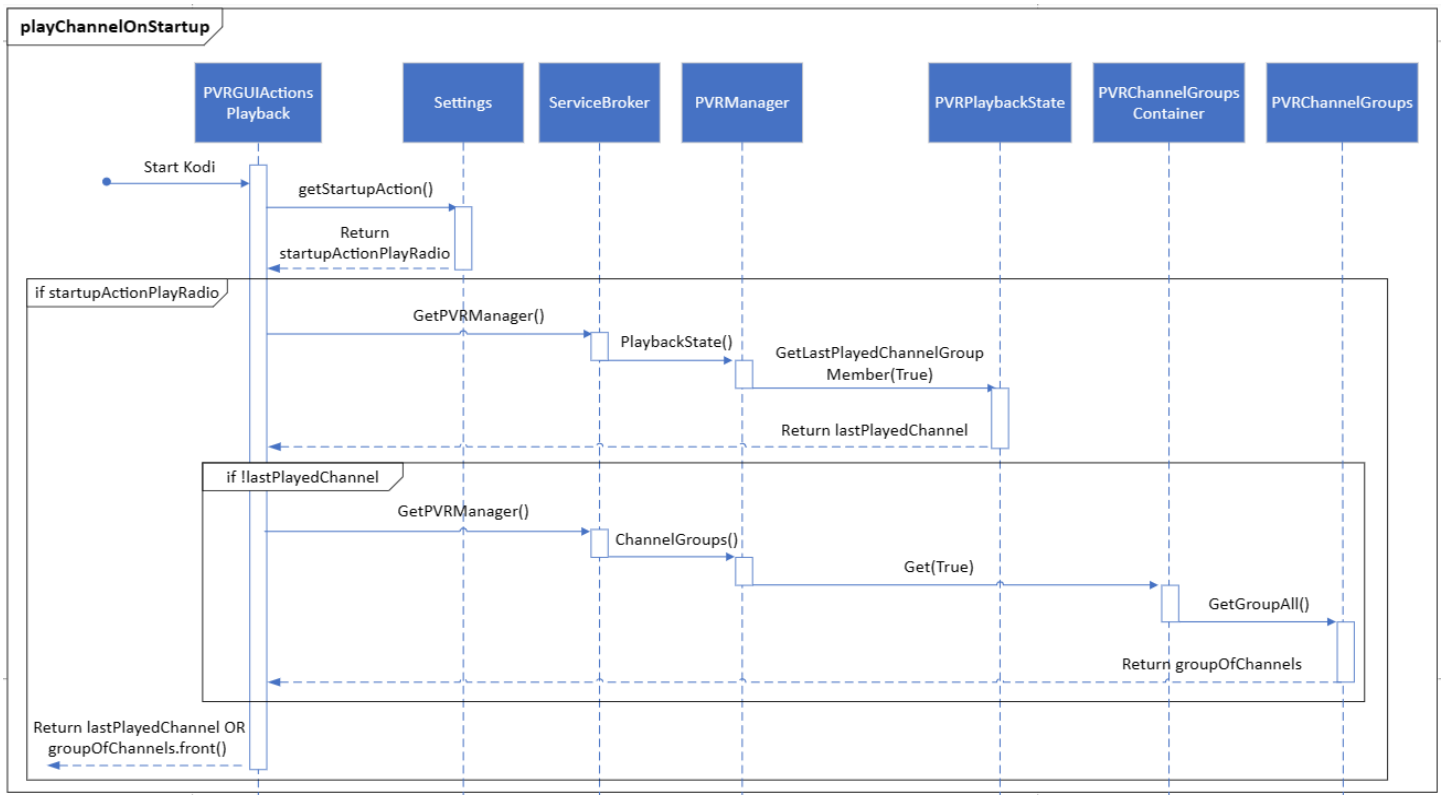


Figure 8: Sequence diagram illustrating how automatic playback works

- GetLastPlayedChannelGroup and Get accept a boolean value, True to get the radio group member and False to get the TV group member (Understand).

2. Kodi enables users to conveniently explore upcoming weather forecasts for their location

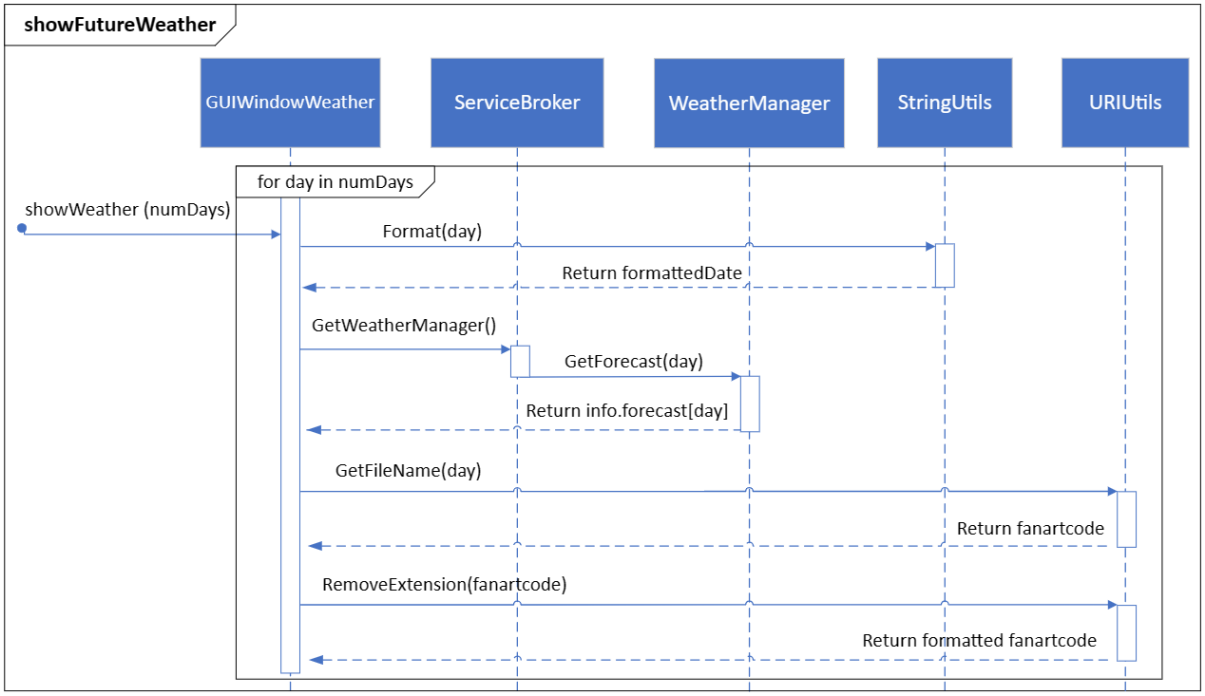


Figure 9: Sequence diagram illustrating how users can check the future weather for their location

- fanartcode contains the weather icon for a given day's forecast (Understand)

REFLEXION ANALYSIS

Top-Level Reflexion Analysis

The architectural style is quite similar, except for added dependencies and a few added top-level subsystems likely due to evolutions over the program's lifespan. Another key difference is that the layers are not abstract. We can see this sentiment demonstrated in an example; when observing the calls made by one of the low-level subsystems, the video codec, we can see that it makes calls to components *outside and above* its module. Ideally in an abstract model, this component would only make calls to lower- and equal-level entities within its module, and pass this information to higher-level entities in its module as well. Instead, we can notice that it calls into other top-level subsystems itself, outside of its directory.

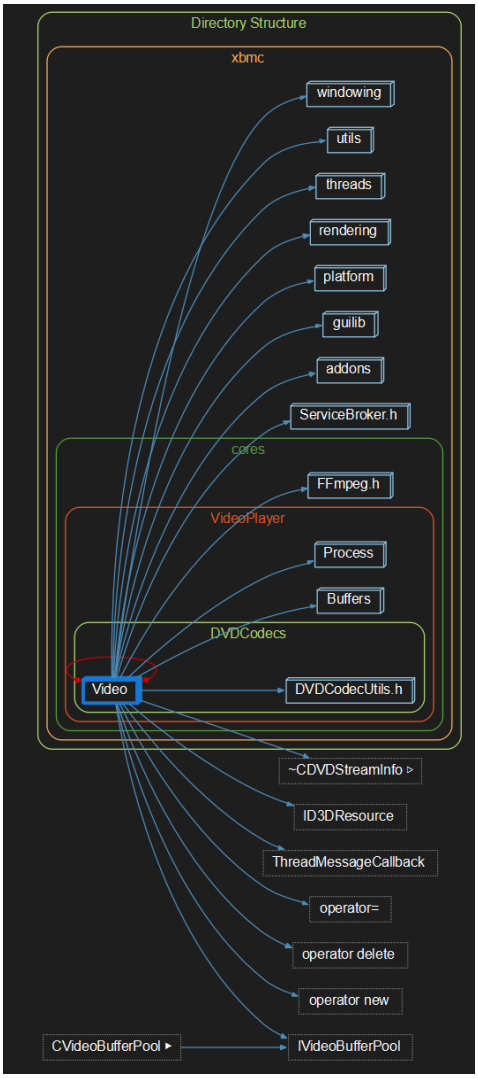


Figure 10: Understand map of the calls made by the Video Codec component.

This is likely due to the nature of the program as an open source volunteer project, as there are no rigorous incentives to maintain key programming practices. In addition, the program has many more dependencies between all the major components. As the program is attempting to emulate a modular style, the excess dependencies that each module has is detrimental to the overall scalability and modifiability of the system.

Cores/Media Controller Subsystem Reflexion Analysis

In terms of similarities, we can note the maintenance of the Video Player, and Video Renderer sections, as they remain largely unchanged keeping their purposes and connections. Another key similarity is the range of media types to be depicted. Different media types such as DVD, Audio, Video, External Players help to maintain its identity as a multimedia player,

On the other hand, some key changes for this section are the fact that the codec is not its own separate component, but rather is integrated into every section that deals with data encryption. This is likely due to the increased number of data types all with their own unique decoding and coding procedures, therefore, there need not be a centralized codec, but a codec for each type of media (Video, Audio, Game etc). There are also many more functionalities, and an increased number of dependencies among all sections. Likely due to new evolutions to the system that have introduced these new dependencies and divergences. For example, the PaPlayer, Retro Player, and External Player all were not represented in conceptual architecture because they are new introductions to the system, coming as recently as 2019, and therefore explain some of the missed dependencies.

Key New Dependencies

PaPlayer → Audio Player

The PaPlayer is responsible for managing audio playback. It relies on the audio engine to handle various aspects such as initializing the codec, reading audio samples, and synchronization (Understand).

PaPlayer → User Interface

The PaPlayer depends on the User Interface to ensure a seamless user experience, managing audio stream details, playback speed, volume, seamless seeking, and updating the GUI with relevant information (Understand).

Retro Player → Video Player

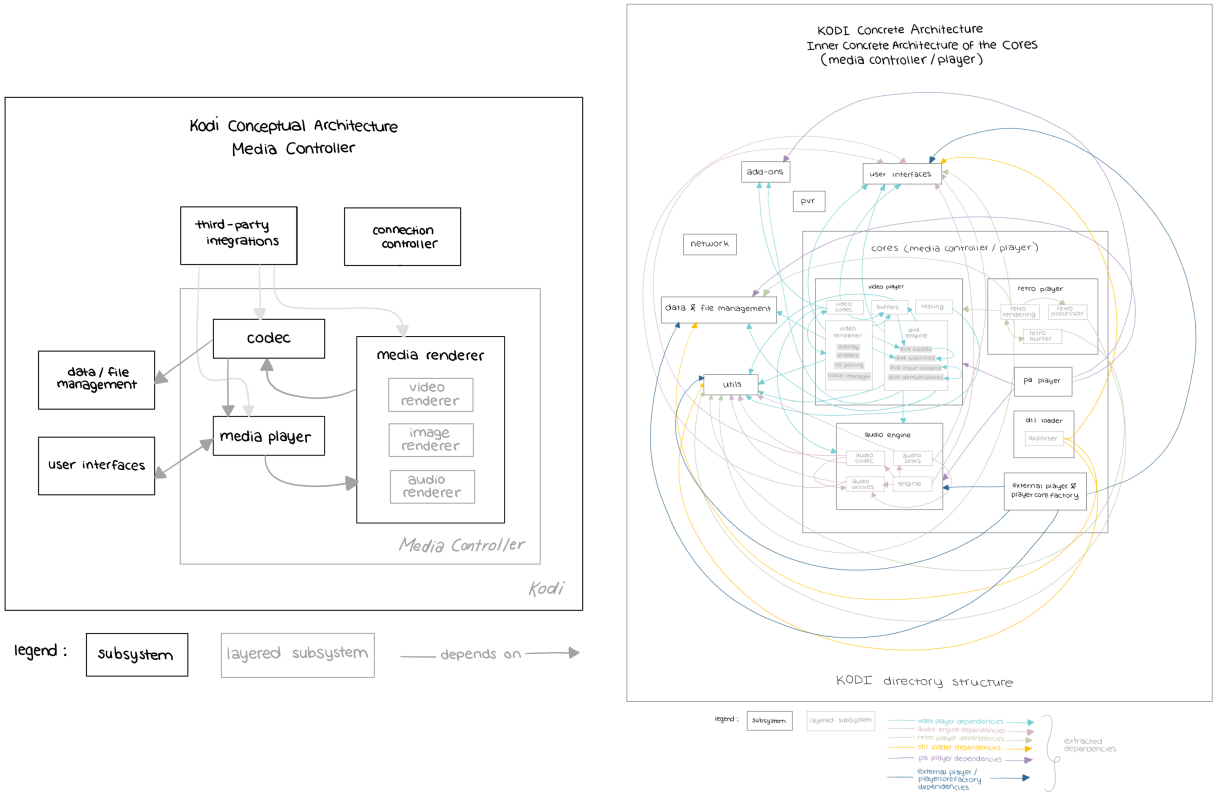
The Retro Player depends on the Video Player's capabilities for rendering, handling input actions, managing focus, and coordinating with the GUI system (Understand).

External Player → Data & File Management

The External Player depends on Data and File management because it interacts with file related entities and data structures. This includes opening and closing files, handling file paths, parsing URLs, and manipulating strings representing file-related information (Understand).

External Player → User Interfaces

The External Player depends on User Interface elements such as dialogues and window creation and management (Understand).



Media Controller Subsystem: Conceptual Architecture

Media Controller Subsystem: Concrete Architecture

Figure 11: The vast difference in complexities can be observed via all the concrete dependencies required to make the media controller run in code.

DATA DICTIONARY

Dynamic-Link Libraries(DLL) - A DLL is a library that contains code and data that can be used by more than one program at the same time. It is a collection of small programs that larger programs can load when needed to complete specific tasks.

NAMING CONVENTIONS

XBMC - XBOX Media Centre
UPnP - Universal Plug and Play
HTTP - Hypertext Transfer Protocol
GUI - Graphical User Interface
DLLs - Dynamic-Link Libraries
JSON-RPC - JavaScript Object Notation-Remote Procedure Call
PVR - Personalized Video Recorder
EPG - Electronic Program Guide
PAPlayer - Psycho-acoustic Audio Player
OS - Operating System
UI - User Interface

LESSONS LEARNED

We neglected details, and lost out on complexities due to overlooking dependencies and other exceptions that only arose once establishing the concrete code. We only examined the structure from a top-level view without taking into consideration the various subcomponents responsible for the core processing and communication between different layers. Furthermore, we did not take into account the possibility of certain components that possess unique properties among the various subsystems. One such example is the codec, though not an independent component, is an integral part of data encryption, automatically integrated with other components responsible for encrypting information. This means that although not viable on its own, it's a component integral to KODI's architecture regardless and provides intrinsic support for the structure. Though we were aware of KODI's existence as a large-scale open-source project, we failed to take into consideration certain aspects of that which could affect the architecture of KODI. An example of this would be the time and release date of certain aspects. We failed to consider that Retroplayer was a recent addition, so there were certain things/elements that we didn't account for.

CONCLUSIONS

KODI is an architectural structure that utilizes layered and modular styles. It is layered by the various directories, which can be seen/proven when it calls to components outside and above its module. At this point, we can see how it directly calls into other top-level subsystems outside its directory, as opposed to our initially proposed abstract style in which ideally components could only make calls to entities equal to or lower than themselves within the module. The directories are sorted into corresponding modules, which when looked upon from the top-level subsystems involve seven primary components. Each of these can then be further broken down deeper into subcomponents through our findings of the concrete architectural structure. An example of this is the Media Renderer, one of the primary components within the architectural structure of KODI. Upon breaking it down for our findings within the concrete architecture, we can see how it has components that in themselves have subcomponents that possess dependencies of their own and communicate with subcomponents both outside and of themselves. All of these combined help support our conceptual architecture findings regarding the modular-styled construction of KODI's architectural model.

REFERENCES

[1] “Window structure,” Window Structure - Official Kodi Wiki, https://kodi.wiki/view/Window_Structure (accessed Nov. 19, 2023).
[2] “Subtitles,” Subtitles - Official Kodi Wiki, https://kodi.wiki/view/Subtitles (accessed Nov. 19, 2023).
[3] “Getting Started With Kodi Retroplayer,” Kodi Community Forum, https://forum.kodi.tv/showthread.php?tid=340684 (accessed Nov. 19, 2023).
[4] “Webserver,” Webserver - Official Kodi Wiki, https://kodi.wiki/view/Webserver accessed Nov. 19, 2023).
[5] “Archive:paplayer,” Archive:PAPlayer - Official Kodi Wiki, https://kodi.wiki/view/Archive:PAPlayer (accessed Nov. 19, 2023).
[6] “Videoplayer,” VideoPlayer - Official Kodi Wiki, https://kodi.wiki/view/VideoPlayer (accessed Nov. 19, 2023).
[7] “AudioEngine,” AudioEngine - Official Kodi Wiki, https://kodi.wiki/view/AudioEngine (accessed Nov. 19, 2023).
[8] “Settings,” Settings - Official Kodi Wiki, https://kodi.wiki/view/Settings (accessed Nov. 19, 2023).
[9] “Development tools,” Development Tools - Official Kodi Wiki, https://kodi.wiki/view/Development_Tools#Frameworks (accessed Nov. 19, 2023).
[10] “Archive:debug dynamic link libraries,” Archive:Debug Dynamic Link Libraries - Official Kodi Wiki, https://kodi.wiki/view/Archive:Debug_Dynamic_Link_Libraries (accessed Nov. 19, 2023).
[11] “External players,” External players - Official Kodi Wiki, https://kodi.wiki/view/External_players (accessed Nov. 19, 2023).
[12] “UPnP,” UPnP - Official Kodi Wiki, https://kodi.wiki/view/UPnP (accessed Nov. 19, 2023).
[13] “Webserver,” Webserver - Official Kodi Wiki, https://kodi.wiki/view/Webserver (accessed Nov. 19, 2023).
[14] [1] “PVR,” PVR - Official Kodi Wiki, https://kodi.wiki/view/PVR (accessed Nov. 19, 2023).
[15] JagPuro, “3 easy ways to set up an EPG on kodi ,” Kodi Tips, https://koditips.com/set-up-an-epg-on-kodi/ (accessed Nov. 19, 2023).
[16] What is Dynamic Link Library (DLL)? - techtarget, https://www.techtarget.com/searchwindowsserver/definition/dynamic-link-library-DLL (accessed Nov. 19, 2023).
[17] Deland-Han, “Dynamic Link Library (DLL) - windows client,” Dynamic link library (DLL) - Windows Client | Microsoft Learn, https://learn.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library (accessed Nov. 19, 2023).