CISC 322 ASSIGNMENT 2

In Medias Res: Kodi's Concrete Architecture

Group 2: ArchAngels

Adam Clarke	21asc6@queensu.ca	20303748
Owen Rocchi	19omr6@queensu.ca	20223890
Bianca Bucchino	20blb@queensu.ca	20232319
Aidan Sibley	20ajs18@queensu.ca	20271383
Omar Ibrahim	20omha@queensu.ca	20263583
Christian Higham	20csdh@queensu.ca	20287289

TABLE OF CONTENTS

ABSTRACT

This report sets out to compare and contrast the conceptual architecture laid out in the previous report with the concrete architecture set in place during development. The main takeaways for the high level comparison is that, due to a need for information to be transferred around continuously throughout the high level components, a layered architecture was not ideal. Although enforcing abstract design is ideal, the need for information flow across the components was more important for the development of this app, and took priority. From this information, the conclusion derived at a high level is that a repository style may have been more beneficial for the development of the app.
In particular, the DVD encoder was analyzed in depth. While from the aforementioned conceptual architecture a layered style was imposed on the application, for this particular subsystem it was revealed that a pub-sub style may have been more efficient. This was due to the elements of the application being used like a library. While the elements of this subsystem were heavily layered in design, each function was needed in a different location so passing them up and over was inefficient.
The conclusion is that iteration and reevaluation of this and most software projects is a necessary process, and after implementing some of the software it is good to evaluate and reconsider some of the design decisions.

INTRODUCTION

This report delves into the intricacies of Kodi's architectural design, contrasting its conceptual framework with the concrete architecture from its development. Kodi was initially designed to utilize a layered architecture comprising three key layers: the client, business, and data layers. This structure provided a simplified and modular approach to understanding Kodi's functionalities. However, the actual software development journey has revealed a more complex and interconnected implementation.

This report will also address the contrast between Kodi's conceptual and concrete architectures. Notably, the deviation from a strict modular approach and the more intricate interdependencies and integrations within the system. This includes the unconventional placement of helper libraries and the direct interaction of Kodi's larger systems with nested files in subsystems, challenging the initial layered architectural concept. The report will explore the reasons behind these deviations.

At the centre of Kodi's evolved architecture is a repository style, which deviates from the initial layered form. Upon examining Kodi’s concrete architecture at a high level, it can be broken down into three subsystems: User Interface, Media Handling, and System Integration. These work alongside the Core Functionality component. Although originally intended as a layered architecture, this in practice behaves more like a repository style.
The User Interface correlates with the Client layer, the Core Functionality and Media Handling with the Business layer, and System Integration aligns with the Data layer. The Core Functionality, in particular, is

pivotal, serving as Kodi's backbone by integrating various components such as the DLL Loader, External Player, Player Core Factory, Audio Engine, Retro Player, Video Player, and the now-deprecated PA Player.

The User Interface component, as the direct point of user interaction, is made up of different components handling Dialogs, Windowing, and the GUI Lib API. The Media Handling component, acting as a frontend to Core Functionality, manages custom addons and the Rendering and Games subcomponents.

System integration allows Kodi to interact with the operating system, and handles the very low level aspect of doing this. It abstracts some of the hardware/operating system specific elements away from the rest of the app.

Further, an integral part of this architecture is the DVDCodecs system. The DVDCodecs is a specialized subsystem for DVD playback. It uniquely maintains a layered architecture within itself, handling different data types on DVDs and facilitating interactions between various Kodi components.

HIGH LEVEL ARCHITECTURES

The following figures, **Figure 1** and **Figure 2** portray the conceptual architecture, and concrete architecture respectively. These diagrams will be referenced throughout the report as per their designated figure number.

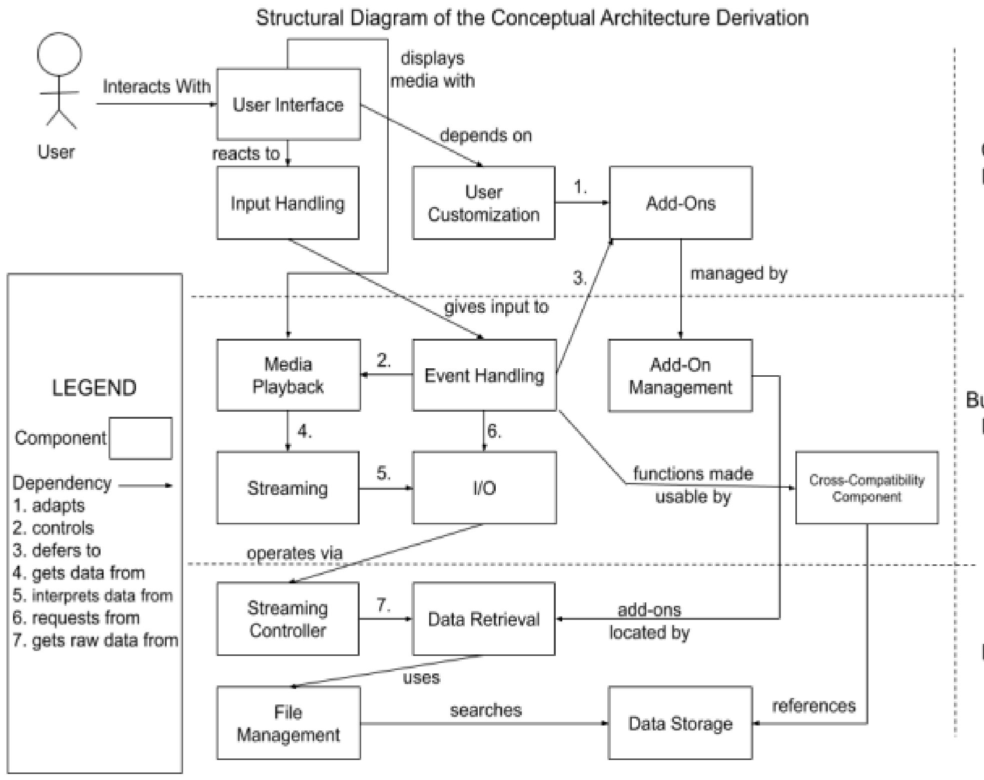


FIG. 1 - KODI CONCEPTUAL ARCHITECTURE

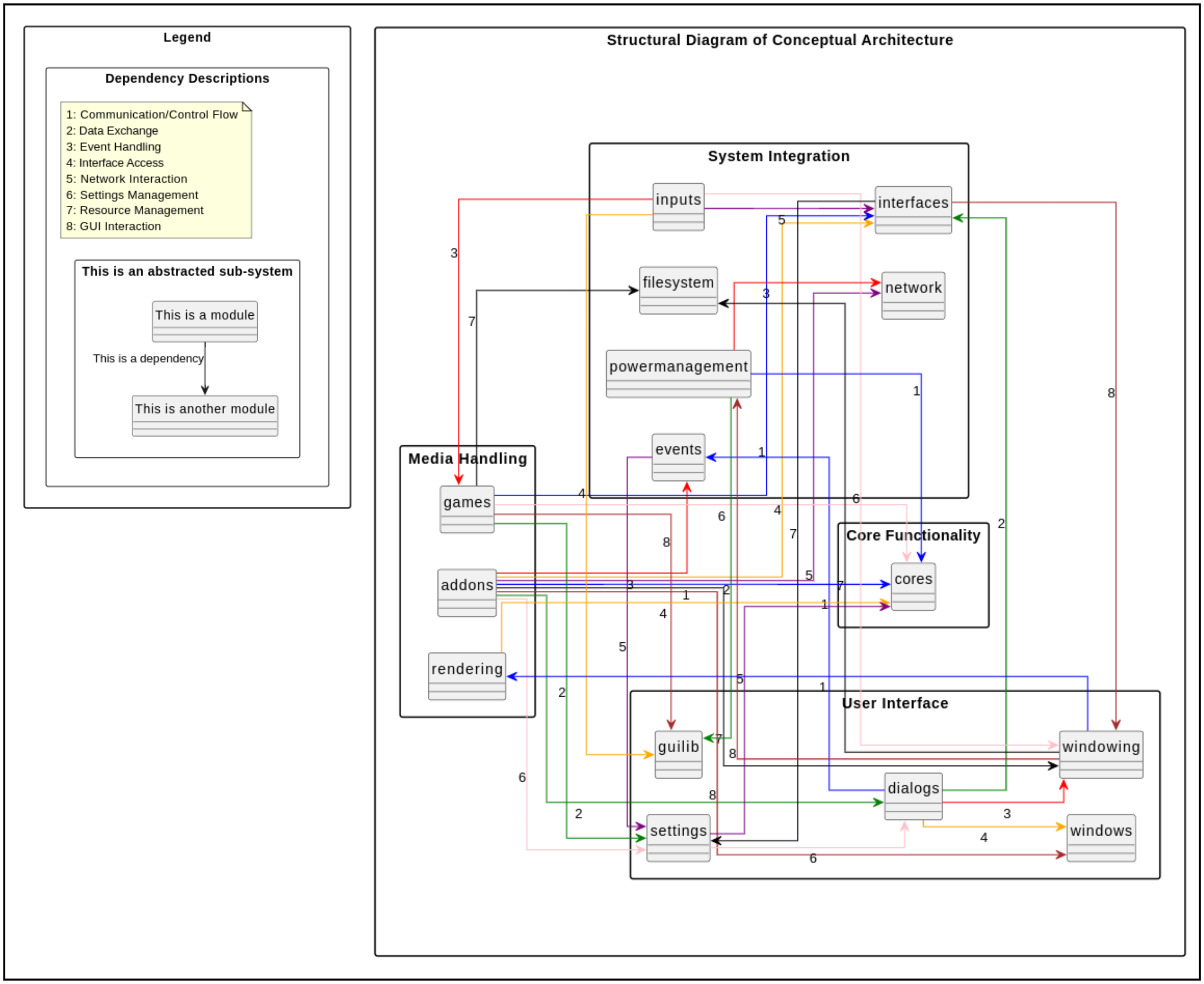


FIG. 2 - KODI CONCRETE ARCHITECTURE

MAIN ARCHITECTURE STYLE

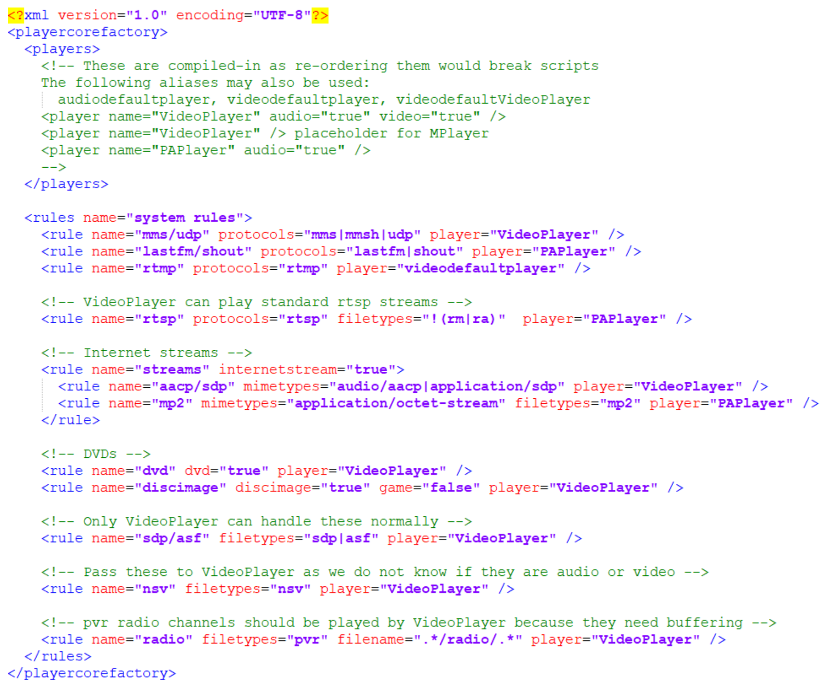
The devised conceptual architecture consisted of three abstracted layers; the client, business and data layers. This highly abstracted view enables looking at modules within the context of a convenient framework. However, as in most software developments, Kodi's concrete architecture is much more nuanced; even at a high level, the architecture is very complex and interdependent. While the conceptual architecture may be organized into layers, the concrete architecture is much more interconnected and does not strictly adhere to the layered style's rules.

Instead, the repository style was found to be much more appropriate to describe Kodi's architecture. After careful examination of the codebase, Kodi's various subsystems were neatly organized into three main categories; *User Interface*, *Media Handling*, and *System Integration* (**Figure 2**). These all communicate via a central control and information body, the *Core Functionality* subsystem.

This architectural shift from a layered to a repository style has significant implications. It enhances Kodi's flexibility in managing its diverse functionalities and improves the integration of new features and updates. However, it also introduces complexities in terms of maintaining a clear separation of context, which could impact long-term maintainability.

To illustrate, the *Core Functionality* subsystem, as a central control and information hub, exemplifies this interconnectedness. It links the user-facing elements with backend processes, ensuring a cohesive operation of the entire application. This approach, while complex, provides Kodi with the adaptability required to handle a wide range of media formats and user interactions effectively.

CORE FUNCTIONALITIES



```xml
<?xml version="1.0" encoding="UTF-8"?>
<playercorefactory>
  <players>
    <!-- These are compiled-in as re-ordering them would break scripts
    The following aliases may also be used:
      audiodefaultplayer, videodefaultplayer, videodefaultVideoPlayer
    <player name="VideoPlayer" audio="true" video="true" />
    <player name="VideoPlayer" /> placeholder for MPlayer
    <player name="PAPlayer" audio="true" />
    -->
  </players>

  <rules name="system rules">
    <rule name="mms/udp" protocols="mms|mmsh|udp" player="VideoPlayer" />
    <rule name="lastfm/shout" protocols="lastfm|shout" player="PAPlayer" />
    <rule name="rtmp" protocols="rtmp" player="videodefaultplayer" />

    <!-- VideoPlayer can play standard rtsp streams -->
    <rule name="rtsp" protocols="rtsp" filetypes="!(rm|ra)"  player="PAPlayer" />

    <!-- Internet streams -->
    <rule name="streams" internetstream="true">
      <rule name="aacp/sdp" mimetypes="audio/aacp|application/sdp" player="VideoPlayer" />
      <rule name="mp2" mimetypes="application/octet-stream" filetypes="mp2" player="PAPlayer" />
    </rule>

    <!-- DVDs -->
    <rule name="dvd" dvd="true" player="VideoPlayer" />
    <rule name="discimage" discimage="true" game="false" player="VideoPlayer" />

    <!-- Only VideoPlayer can handle these normally -->
    <rule name="sdp/asf" filetypes="sdp|asf" player="VideoPlayer" />

    <!-- Pass these to VideoPlayer as we do not know if they are audio or video -->
    <rule name="nsv" filetypes="nsv" player="VideoPlayer" />

    <!-- pvr radio channels should be played by VideoPlayer because they need buffering -->
    <rule name="radio" filetypes="pvr" filename=".*/radio/.*" player="VideoPlayer" />
  </rules>
</playercorefactory>
```

FIG. 3 - playercorefactory.xml

Central to Kodi's system architecture is the Core Functionality component, which orchestrates mission-critical data and processes. Figure 2 illustrates this component's pivotal role in Kodi's system, with other components relying on its modules for various operations. This component is a composite of several sub-modules, each specialized in handling different facets of media playback, ensuring Kodi's robust performance.

Within the architectural framework, `playercorefactory.xml` **(Figure 3)** stands out as a strategic element. It is the architectural nexus that configures the interplay between different media players. This XML file embodies the principles of modifiability and reusability, which are hallmarks of a sound software architecture. By enabling the specification of player behaviors and rules, it facilitates a flexible and extensible environment for media playback. It reveals the system's underlying design principles that prioritize modularity, configurability, and user-centricity, ensuring Kodi remains a versatile and forward-compatible media center.

Architectural Overview of Key Modules:

1. **DLL Loader**: This module simplifies the extension of Kodi's capabilities by loading additional features through dynamic link libraries. Architecturally, it exemplifies the system's modularity, allowing for the seamless integration of new functionalities.
2. **External Player**: Reflecting the architecture's adaptability, the External Player module provides a means for users to enhance Kodi's native playback capabilities, demonstrating the system's ability to accommodate external integrations.
3. **Player Core Factory**: The core of the `playercorefactory.xml` configuration, this module delineates the interaction rules for various players, supporting the architectural goal of a customizable user experience. It acts as a configuration control point within Kodi's architecture, streamlining player behavior management.
4. **Audio Engine**: As a central hub for audio processing, the Audio Engine encapsulates essential audio operations, reflecting the system's architectural commitment to robust media handling.
5. **Retro Player**: Supporting a wide range of emulators, this module's integration with the Libretro API indicates an architectural design focused on interoperability and extensive media support.
6. **Video Player**: Parallel to the Audio Engine, the Video Player module's design addresses the architectural need for a comprehensive video processing solution, ensuring Kodi's compatibility with a vast array of video formats.
7. **PA Player**: The deprecated PA Player module's historical role in audio-visual processing points to the evolving nature of Kodi's architecture, which adapts and progresses in response to technological advancements and user demands.

The `playercorefactory.xml` is a testament to the flexibility of Kodi's architecture. It showcases how the system's design caters to the adaptability required by users who wish to extend or customize their media playback experience. The XML file functions as an architectural configuration tool, mapping the capabilities of the system's components to the diverse requirements of different media types and user scenarios.

USER INTERFACE

The User Interface component is Kodi's interactive front, serving as the primary point of contact between the user and the application. It consists of several sub-modules: `Dialogs`, `Windowing`, `Windows`, `Settings`, and `GUI Lib`, which collectively define the behavior of the display interface.

The sub-modules function as follows:

1. `Dialogs`: Manages the various dialog windows for user interactions and system notifications, closely interacting with `Events` and `Windows` to ensure seamless user experiences.
2. `Windowing`: Oversees the management of different application windows, providing a framework for rendering the user interface on various devices and platforms.
3. `Windows`: Handles specific window instances within the Kodi application, such as the main menu, settings, and media playback screens.
4. `Settings`: Acts as the configurational hub where user preferences are maintained and utilized across the application.
5. `GUI Lib`: Functions as an Application Programming Interface (API) that other user-interface-related modules use to access the GUI Info database, which stores information critical to the operation of all Kodi windows. This includes, but is not limited to, the hero page, video player, music player, game emulator, and weather app.

In the context of the concrete architecture diagram, the User Interface component demonstrates a high degree of interactivity with other modules, particularly `Settings`, which is central to user customization, and `Windowing`, which facilitates the rendering of different user interfaces.

MEDIA HANDLING

The Media Handling component is responsible for orchestrating the logic behind all media-related functionalities, predominantly encompassing `Addons`, `Rendering`, and `Games`. It functions primarily as a frontend interface to the Core Functionality modules, ensuring a unified media experience.

1. `Addons`: Enhances Kodi's capabilities by allowing users to add features and manage addon binaries (executables), the addon manager GUI, and the dev-kit (used to help users develop their addons with KodiSwift).
2. `Rendering`: Kodi's principal video rendering engine leverages the Core Functionality's Video Player and interfaces with `Cores` to deliver media content. It employs platform-dependent rendering interfaces such as Direct X, OpenGL, and OpenGL ES, catering to a broad range of devices from Windows PCs to embedded systems.
3. `Games`: Acts as an interface for the Retro Player module, incorporating dialogs and ports management for game input and windowing functions, enhancing the gaming experience on Kodi.

In alignment with the concrete architecture diagram, `Addons` emerge as a nexus of interaction with various other components, signifying its role in extending Kodi's functionalities through additional user-installed features.

SYSTEM INTEGRATION

System Integration forms the underlying framework that enables Kodi to seamlessly interact with the operating system. It includes the `Input`, `Events`, `Power Management`, `Filesystem`, `Interfaces`, and `Network` sub-modules, each handling a distinct aspect of system operations.

1. `Input` and `Events`: Capture and process user input through peripherals, supporting a wide range of devices from joysticks to touch interfaces.
2. `Power Management`: Manages the device's power and battery operations, ensuring efficient energy utilization.
3. `Filesystem`: Acts as the central file management system, organizing music and video database directories.
4. `Interfaces`: Utilizes the implicit invocation architectural style to broadcast and listen for external events, enhancing Kodi's modularity and adaptability through support for JSON-RPC and Python scripts.
5. `Network`: Orchestrates all network protocol controls, supporting technologies like DACP, HTTP, MDNS, UPNP, and WebSocket for a diverse range of networking functions.

ARCHITECTURE DERIVATION PROCESS

In deriving the high-level dependency diagram (**Figure 2**), a tailored approach was necessary to navigate the complexity of the Kodi project. Utilizing the software 'Understand,' key folders within the `xbmc` directory of the Kodi source repository were examined. Due to the project's extensive scale, generating a UML class diagram for the entire source directory would yield an unreadable outcome. To circumvent this, 'Understand' offers a dependency browser that delineates 'Depends On' or 'Depended On By' relationships for all files within a given directory—streamlining the visualization of dependencies within sub-architectures.

To refine this data into a coherent high-level view, a Python script was employed to filter out redundant connections. The script's logic dictated that – if sub-system 'A' depended on 'B' (A <-- B) and 'B' on 'C' (B <-- C), any direct dependency from 'A' to 'C' (A <-- C) was deemed to be transitive in nature and thus omitted for the sake of brevity. This step was critical in garnering a readable diagram to showcase the core associations between subsystems.

Figure 4 , introduced later on in the report, represents the concrete architectural diagram for the chosen subsystem; the methodology was simplified due to fewer interdependencies. The Overlay, Audio, and Video modules within the DVDCodecs subsystem were manually parsed and diagrammed, providing a clear, module-wise representation of the sub-architecture's layout.

VIDEO PLAYER SUBSYSTEM: DVDCODECS

SUBSYSTEM ANALYSIS

The DVDCodecs subsystem is a critical component of Kodi's media player, designed to address the complex actions needed to support DVD playback including processing various data types—ranging from audio and video streams to subtitle overlays.

To achieve this, the DVDCodecs subsystem employs a layered architectural strategy. This structure not only facilitates the efficient processing of distinct media types but also promotes a modular design that simplifies maintenance and future enhancements. By dividing responsibilities into well-defined layers, the DVDCodecs subsystem enhances Kodi's overall resilience and adaptability in dealing with the multifaceted nature of DVD playback.

In the following figure, we delve into the inner workings of this subsystem, illustrating how each component contributes to the seamless playback experience that Kodi users expect.

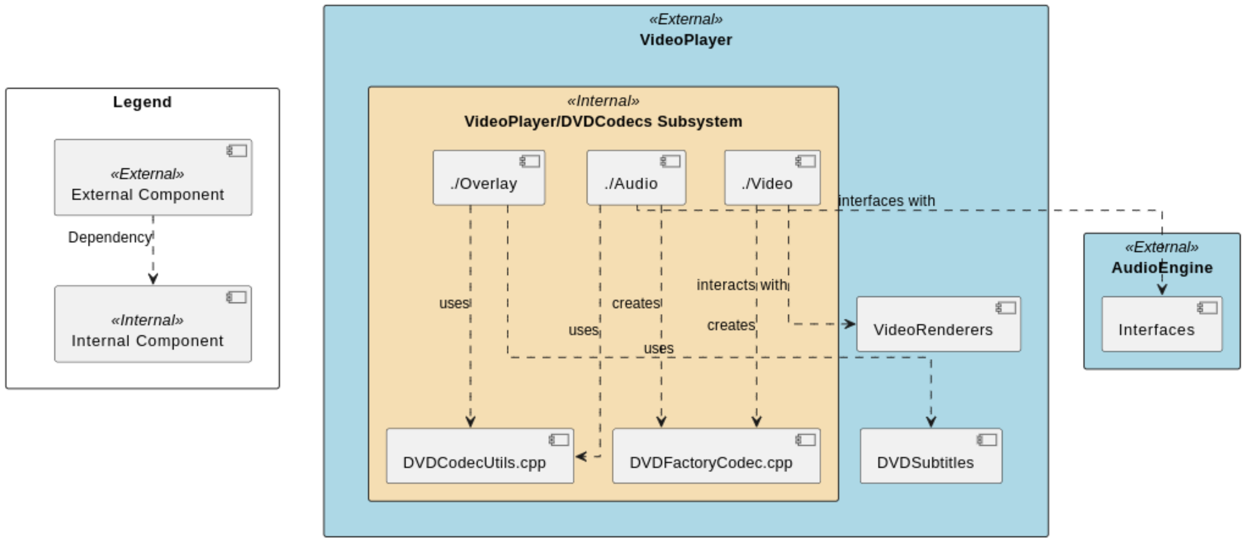


FIG. 4 - DVDCODECS SUBSYSTEM CONCRETE ARCHITECTURE

The diagram above represents the architectural abstraction of the DVDCodecs subsystem within Kodi's media player. This subsystem is organized into three key components, each dedicated to a specific aspect of media playback:

1. **Audio**: This component is responsible for the decoding and processing of audio streams from DVDs, ensuring that sound is rendered accurately during playback.
2. **Overlay**: Manages the rendering of subtitles and on-screen display elements, which are crucial for user accessibility and interaction.
3. **Video**: Handles the decoding processes for video streams, facilitating the playback of visual content from DVDs.

Central to the DVDCodecs directory is `DVDFactoryCodec.cpp`, which serves as an integration point for the subsystem. It allows the components to function autonomously yet ensures cohesive communication. `DVDFactoryCodec.cpp` is divided into three segments corresponding to the subsystem's primary components—Audio, Overlay, and Video—streamlining the interaction within the subsystem.

While each component operates as a self-contained unit, with no direct interaction between Audio, Overlay, and Video, the internal structure of these components reveals numerous interdependencies. Many files within each component, such as Audio, have dependencies on numerous other files in the same component, indicating a high degree of coupling and a lack of overall conceptual abstraction.

CONTRASTING CONCEPTUAL AND CONCRETE VIEWS

When contrasting the conceptual and concrete views of the DVDCODECS subsystem, we observe a departure from the clean, modular layered approach anticipated in the conceptual design. The concrete architecture uncovers a web of dependencies that speaks to a more complex and interconnected system.

In the conceptual architecture depicted in Figure 1, we see an orderly arrangement, where each layer of the system—Client, Business, and Data—performs discrete functions with minimal overlap. The DVDCODECS subsystem, envisioned as part of the Business Layer, is expected to act as a modular entity, facilitating a clear separation of concerns between handling audio, video, and overlay functionalities.

In contrast, the concrete architecture, as illustrated in **Figure 4**, reveals a different story. The helper library for the Video component, rather than being a localized utility within its layer, is elevated, granting broader scope and suggesting a more systemic role than initially assumed. This configuration implies a significant intertwining of functionalities across the subsystems, indicative of the complex nature of media handling in Kodi.

The direct interaction with nested files inside the Audio, Overlay, and Video components, bypassing a unifying top layer, suggests a design that evolved organically, where modularity was sacrificed for accessibility and perhaps ease of development. This pattern suggests two potential narratives:

1. **Historical Context**: The architecture of the DVDCODECS subsystem might have been shaped by Kodi's legacy as a media player primarily for DVD content. As the software expanded to accommodate newer media formats, the foundational DVD-related code was repurposed, with new functionalities built upon it rather than restructuring the existing architecture. This approach reflects a pragmatic adaptation to changing requirements, preserving historical efficiencies while embracing new capabilities.
2. **Pragmatic Development**: The lack of a concerted effort to abstract and compartmentalize the codebase could also reflect a development philosophy that values direct implementation over architectural purity. The pragmatic reuse of code, even at the expense of increasing inter-module dependencies, may have been a conscious choice driven by the need for rapid feature development and deployment.

In essence, the DVDCODECS subsystem embodies a common dilemma that software developers face in the pursuit of balancing theoretical architectural principles with the pragmatic realities of evolving software needs and legacy code constraints. It stands as a testament to the adaptability required in software design, where the original conceptual structure often yields to the practical demands of functionality, user requirements, and technological advancements.

SYSTEM REFLEXION ANALYSIS

HIGH LEVEL ARCHITECTURE

Thorough investigations revealed that the concrete architecture has some significant differences when compared to the conceptual design. Although most if not all individual portions of the system stayed the same, many deviations exist in how different pieces are connected.

The initial conceptual architecture held a strong emphasis on a layered design. The purpose of this design was to ease development by enforcing layers of abstractions between the layers of client, business, and data.

1. **Client Layer:** The functionally built version of the client layer, its concrete architecture, hardly deviated from its conceptual counterpart. Aside from the addons component, everything else ended up existing in the client layer of the application. This was done to create a more significant section of code for media to sit in.
2. **Business and Data:** The business and data layer had many parts combined to create a more general system integration. As many unexpected dependencies were created between the business and data layer, abstracting them and enforcing modularity ended up bringing more trouble than it was worth. Specifically access to data was better abstracted in the conceptual architecture, keeping access mainly within its own layer. Unfortunately this design could not be achieved within the concrete implementation. This was because the general use case of this component was not within its respective layer (the system integration) but rather by separate components, media handling and user interface.

The concrete architecture reflects a more robust media handling design compared to what was present in the conceptual plan. While there was only a single component of media playback, in practice there now exists an entire media handling section of the architecture containing functionality for addons, rendering, and games. This is a significant difference and can be attributed to a poor prediction of how the media playback would work. This can also be attributed to a change in how the components to the user interface were set up. User interface was only a component in the conceptual architecture, but in the concrete architecture is represented by multiple components. The change in abstraction introduced considerably more connections for the single media playback component to handle, justifying its transition to an entire section. Once again a better understanding of the planned system could have avoided this initial incorrect estimation.

Another change quickly noticeable is the introduction of the core functionality section. As the implementation often referenced a handful of common utilities, abstracting it to its own section became the most logical option. The cores system represents the core functionality of the application (media presentation, system integration, etc) and funnels these key aspects into one place. Within the cores, there is the video player (which is further abstracted with a layered architecture), retro video game player, external player, dll loader, an external player, an audio engine, and a general player core (handles general applications of things related to playing, is composed of only .cpp and header files, no further abstraction). Due to the critical nature of this system with respect to the rest of the application, keeping core functionality in any other layer would have created too large of a hindrance to the abstraction of the system.

Upon implementation it quickly appears that the better abstraction for the system resides in the concrete architecture. These sections more accurately split up the tasks described in the initial conceptual architecture. User interface is the most intuitive and contains all functionality relevant to the user and user input. Media handling is also logical when considering the amount of common functionality shared within any sort of component dealing with media. Also due to the sheer amount of dependencies it makes sense to have its own section. System integration keeping hold of anything low level helps to keep those details abstracted from other sections. The conceptual plan had too many potential overlaps contained within the business layer that should've been put in the data layer. Lastly, keeping core functionality its own component allows effective sharing of common functions to keep the system as simple as possible.

These moments of reflection are important in identifying the major differences between the planning and implementation of code. Recognizing these gaps can help to create better predictions for the project moving forward.

SECOND LEVEL SUBSYSTEM

The second-level subsystem analysis of DVDCodecs provides a window into the development process and the challenges encountered by Kodi's development team. The conceptual architecture's modular and recursive layering was not entirely feasible in practice, leading to a necessary departure for the concrete implementation.

The layered approach, while ideal in theory, proved less suitable for the DVDCodecs subsystem, which required a flexible architecture to accommodate code reuse across different system components. The concrete architecture adopted a pub-sub style to facilitate this requirement, albeit without the desired level of abstraction. This resulted in direct interactions between various subsystems and the DVDCodecs component, significantly increasing the dependency count.

The challenge of maintaining such an intertwined system is substantial. Modifications to a single function could necessitate updates across a vast number of files, exacerbating the difficulties in C++ development where minor changes can have far-reaching impacts.

Although the concrete architecture captures the essence of a pub-sub architecture, the absence of a dedicated abstraction layer has led to a tangle of dependencies that resemble spaghetti code. This complexity presents a formidable challenge for developers, akin to untangling years of interwoven dependencies – a task that can only be described as a development team's worst nightmare.

In conclusion, the reflexion analysis has highlighted the divergence between the conceptual and concrete architectures of Kodi, particularly within the DVDCodecs subsystem. The insights gained point to the importance of incorporating flexible design practices that can accommodate the system's evolutionary needs. A hybrid approach, blending layered and pub-sub architectures, would have offered a more practical and achievable roadmap for implementation. Future architecture planning should draw on these reflections to develop a robust, maintainable, and scalable system that aligns more closely with the conceptual ideals while remaining responsive to the practical challenges of software evolution.

SEQUENCE DIAGRAMS

SEQUENCE DIAGRAM OF HIGH LEVEL NON TRIVIAL USE CASE

This sequence diagram details the user interaction flow with Kodi's system when selecting and streaming a live sports event. It delineates the necessary steps and system modules involved, from the user's initial event selection through to the potential outcomes of a successful or failed connection, encompassing the entire experience within the user interface and streaming subsystems.

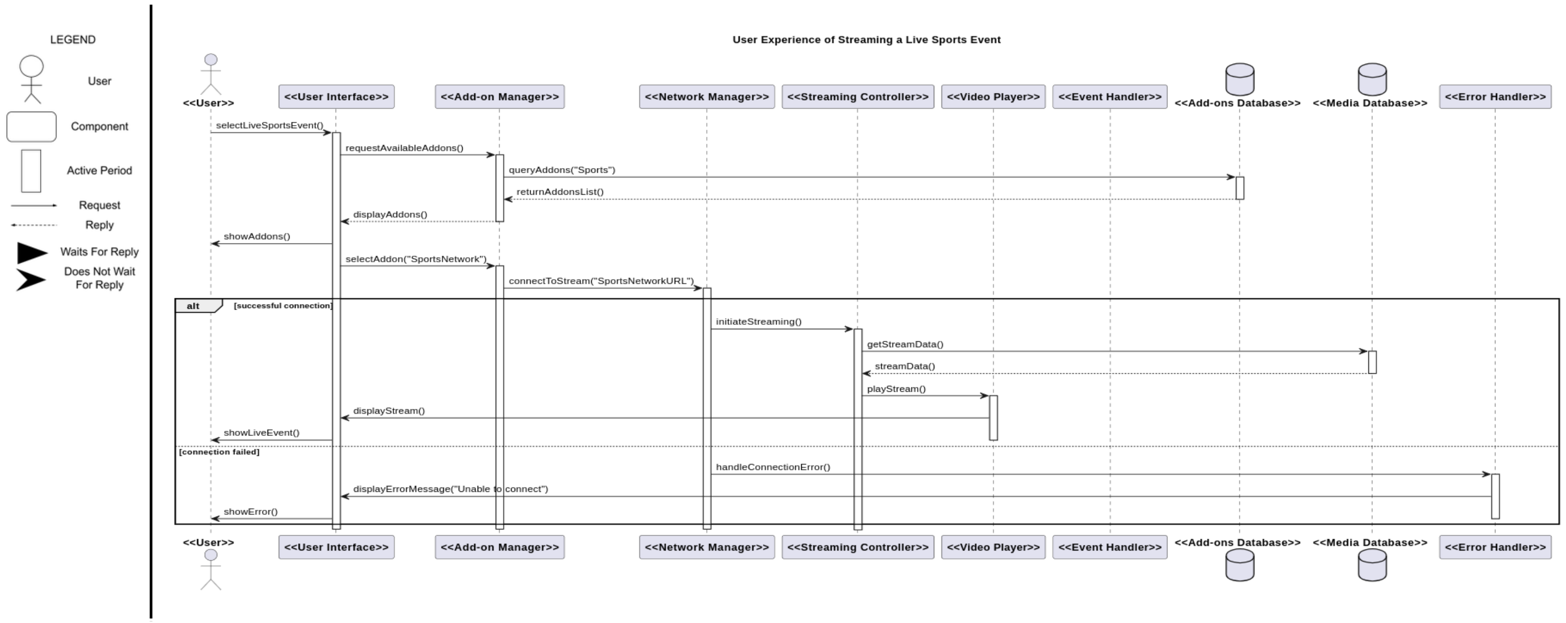


FIGURE 5 - NON TRIVIAL USE CASE - HIGH ORDER OVERVIEW

SEQUENCE DIAGRAM OF DVDCODECS SUBSYSTEM NON TRIVIAL USE CASE

The process of switching audio streams during DVD playback is a complex interaction within the DVDCodecs subsystem, requiring precise coordination between audio and video processing components. This sequence diagram illustrates the steps taken by the system when a user selects an alternative audio language track, showcasing the dynamic response of the player to user input.

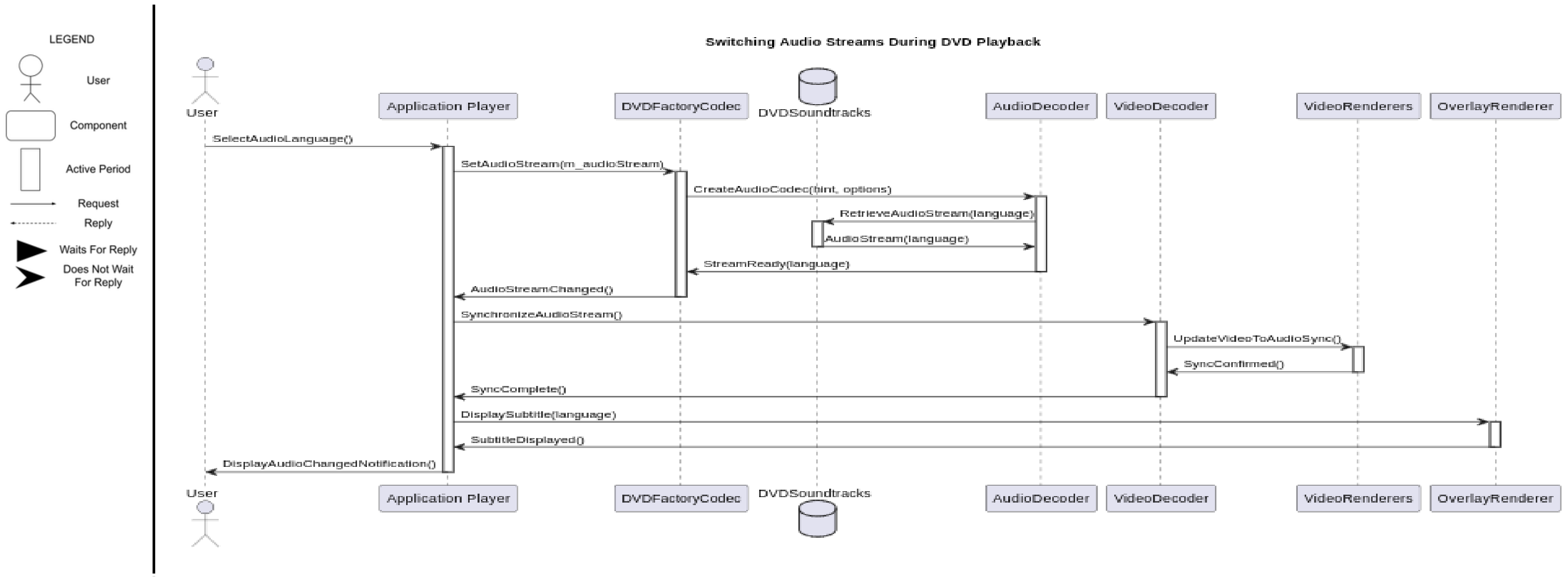


FIGURE 6 - NON TRIVIAL USE CASE - DVDCODEC SUBSYSTEM

CONCLUSION

In conclusion, exploring Kodi's concrete architecture, explicitly focusing on the subsystems and their derivations, sheds light on the intricate challenges faced during development.

Initially designed with a layered approach, the conceptual architecture encounters deviations in the concrete implementation. Notably, the subsystem analysis of DVD Codecs exposes the struggle with achieving modularity and abstraction, leading to the reconsideration of design choices. The transition from a layered to a more interconnected architecture poses maintenance challenges and highlights the importance of anticipating system complexities.

Despite its deviations, the concrete architecture demonstrates a more effective organization in certain aspects, emphasizing the need for a careful balance between abstraction and practical implementation.

Reflection on these findings underscores the importance of adapting architectural designs based on the evolving needs of the system, with future improvements requiring a thoughtful blend of layered and pub-sub elements for a more maintainable and functional implementation.

LESSONS LEARNED

Completing this project taught us many valuable lessons as a team. We learned many technical and non-technical skills that are crucial in professional environments. We learned how to differentiate between conceptual and concrete architecture, how to use specific tools to help us analyze a system from its source code and enhanced our team communication skills.

We gained crucial knowledge on how to utilize the Understand tool to break down a large repository into subsystems. By taking advantage of Understand's visual features, we got a deeper understanding of the Kodi System, and it provided us with tools to analyze further systems.

Lastly, we gained essential team-building and communication skills. By splitting up tasks and scheduling numerous group meetings for organizational and planning purposes, we were able to take on this assignment efficiently and effectively. Further, by using ample communication, we were able to reasonably split up work and communicate as a team to work through problems we needed to be more confident in.