CISC 322
Assignment 2
Concrete Architecture of ScummVM
Nov 18, 2024

Group-26

Antony Li

Ghaith Khalil

Owen Gimbel

Yuchen Lin

Mitchell Nagler

Jasmine Zangeneh

Table of Contents

- Abstract
- Derivation Process
- Top-Level Conceptual vs Concrete Architecture & Reflexion Analysis
- SCI Engine Conceptual vs Concrete Architecture & Reflexion Analysis
- Use Cases & Sequence Diagrams
- Lessons Learned
- Conclusion

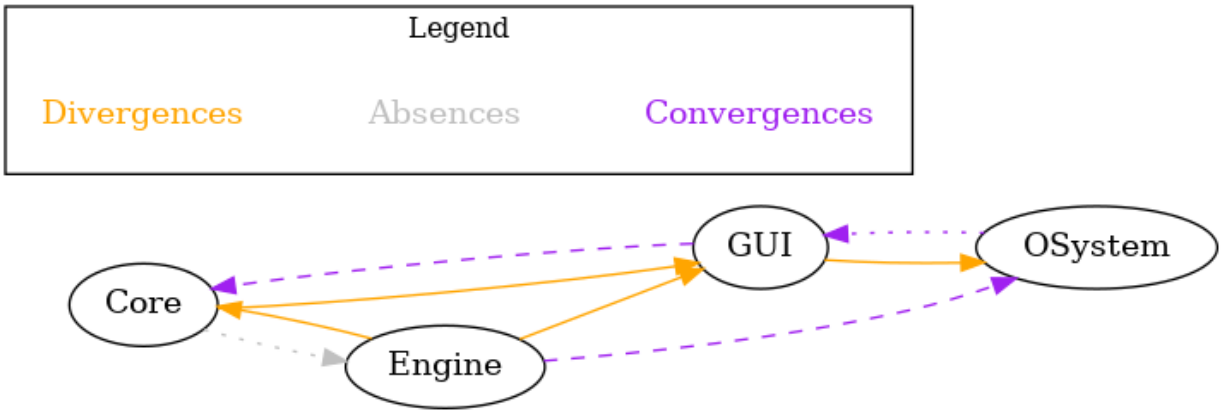
Abstract

In this report, our group analyzes the concrete architecture of ScummVM, the open-source software for playing classic graphical adventure and role-playing games on modern systems [1]. We do this by using Understand by SciTools, software that helps analyze code projects by displaying important information such as call trees, cross-references, and most importantly for us, dependencies [2].

Derivation Process

To better understand the concrete architecture of ScummVM, we utilized SciTools Understand, a code evaluation software that analyzes and visualizes the relationships between various modules in a codebase. By inputting ScummVM's code, the tool generated a dependency graph that mapped how different modules interact with one another. This analysis helped uncover both expected and unexpected dependencies, ensuring no critical relationships were overlooked. The tool also allowed us to identify any missing dependencies or areas for improvement in the system's design.

Conceptual A1 Architecture vs Concrete Comparison



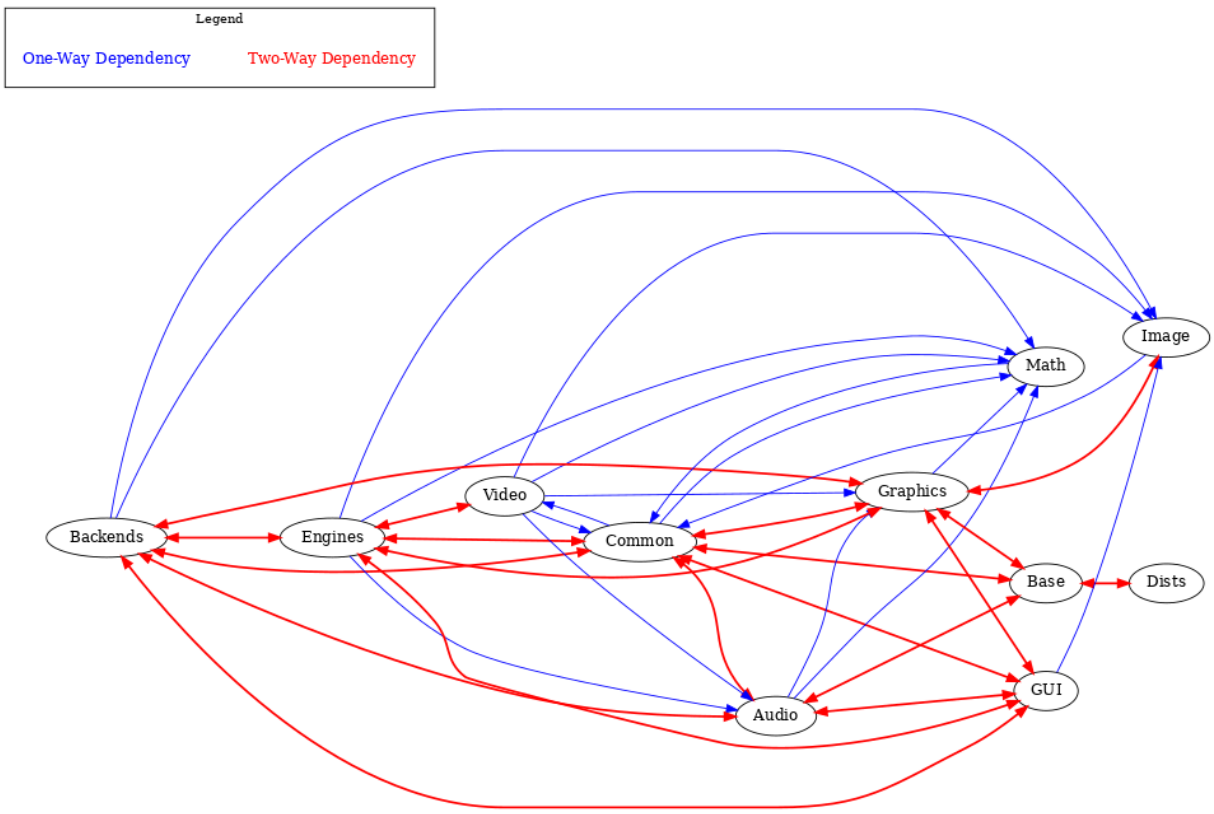
Transitioning from conceptual architecture to concrete architecture, we can see several changes shown by the diagram above. We can categorize these into divergences, absences, and convergences. This shows how the practical implementation differs from our conceptual architecture shown in the first assignment.

In terms of divergences, new dependencies that are missing in our conceptual architecture, we have: an outgoing connection from core (common) to the engine. This connection improves the coordination of shared resources and operational processes. There is also another outgoing connection from the engine to the GUI due to real-time interactions between the user interface used in accordance with the engine during gameplay.

There was one dependency that was present in our conceptual architecture that does not exist in the concrete architecture. This was a connection from the core (common) to OSystem, which deals with system-level interactions to the engine. This also adds modularity to the architecture for better understanding.

There are a couple notable convergences that should also be mentioned. These are the GUI connected to the OSystem for rendering windows or processing input events. We can also look at the OSystem connected to the engine, keeping system-resources consistent.

Concrete Architecture and Box-And-Arrow]



Now, we can look over the entire concrete architecture for ScummVM. Our analysis revealed that there are 11 integral modules in the architecture. These modules include the Backends, which handle platform-specific operations. The Engines, which interpret and execute the game logic. There are also additional modules that handle parts of the game which are the GUI, Video, Graphics, and Image. And finally the Base, Dists (which interact closely with the Base), Math, and Common modules support a wide range of functionalities and interconnect with many other components.

The architectural style that is present is a layered architectural style. This style allows for clear separation which can help with scalability, maintainability, and cross-platform compatibility. This expands on our initial analysis, so most of what is listed below is taken from our first report.

Layers

1. Application Layer
    - GUI and core application logic; depends on the layers below for resource management and rendering
    - It handles user interactions, such as navigating menus, selecting games, and configuring settings.
2. Game Engine Layer
    - Contains various game engines that act as interpreters to execute game scripts
    - Note that each engine operates independently
    - Depends on the Common Services Layer for input, graphics, audio
3. Common Services Layer
    - Includes Common, Graphics, Audio, Video, Math, Image
    - Provides shared services and utilities, such as file I/O, memory management, graphics rendering, and audio playback.
    - Depends on Platform Abstraction Layer for system-level details
4. Platform Abstraction Layer (OSystem)
    - Abstracts system-specific details such as input handling, threading, and audio/video rendering
    - Used to define a generic interface (OSystem API) used by the upper layers
    - Relies on Backend layer for platform-specific interactions (i.e. Windows vs MacOS)
5. Backends Layer
    - Implements the OSystem API for specific platforms, ensuring the application runs on various operating systems and devices

    - Directly interacts with hardware and the Platform Abstraction Layer
6. Hardware Layer
    - Underlying physical hardware and the operating system that ScummVM operates on

ScummVM also uses an interpreter style which allows it to play multiple games across multiple engines. Each game engine itself acts as an interpreter for that game's scripting language and original game logic. As an example, the SCUMM engine interprets bytecode to play Monkey Island and uses the Kyra Engine for scripts like The Legend of Kyrandia. This follows a plug-in architecture format which can be loaded and unloaded depending on what game is being run.

Reflexion Analysis for the High-Level Architecture

In our conceptual architecture, we outlined the core components and their interactions in a high-level view. The **high-level architecture** of ScummVM is built around a **layered architectural style**, with modules such as:

- **Application Layer** (handles user interface and core logic)
- **Game Engine Layer** (where game engines interpret and execute the game logic)
- **Common Services Layer** (provides shared services like memory management, graphics rendering, etc.)
- **Platform Abstraction Layer** (abstracts system-specific details for various platforms)
- **Backends Layer** (handles platform-specific operations, interacts directly with hardware)
- **Hardware Layer** (represents physical hardware)

The **reflexion analysis** of this high-level architecture involves comparing the **conceptual architecture** with the **concrete architecture** and identifying any discrepancies, absences, or convergences. Here's what we observed:

Divergences:

- **New Dependency between Core and Engine**: In the **concrete architecture**, we observed a new outgoing connection from the **Core** module to the **Engine** module. This was absent in the conceptual model. This connection is essential in the concrete architecture to coordinate shared resources and improve operational processes.
- **Real-Time Interaction between Engine and GUI**: The **conceptual architecture** did not account for the interaction between the **Engine** and **GUI** during gameplay. In the **concrete architecture**, the engine interacts with the GUI in real time, particularly for game state updates and user input. This is a key change from the initial design.

Absences:

- **Core to OSystem Dependency**: In the **conceptual architecture**, we suggested that the **Core** (common) would interact with the OSystem (platform abstraction layer). However, this dependency is absent in the **concrete architecture**. This absence reflects a shift toward a more modular approach where system-level interactions are abstracted into other layers.

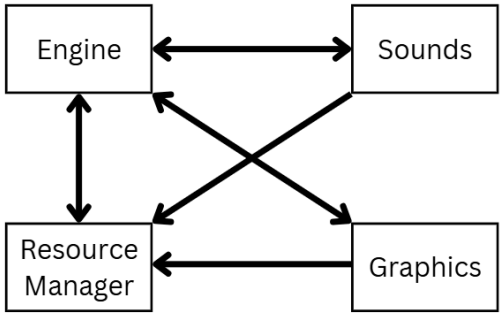
Convergences:

- **GUI to OSystem**: In the **concrete architecture**, the **GUI** and **OSystem** are connected. This wasn't clear in the **conceptual architecture**, but in practice, the **GUI** relies on the **OSystem** for rendering windows and processing user input.

- **OSystem to Engine**: The **OSystem** module in the **concrete architecture** is also connected to the **Engine** module, ensuring system-level resource consistency and facilitating platform-specific interactions, which is crucial for cross-platform compatibility.

Sci Engine Conceptual Architecture

The Sci engine's conceptual architecture is an interpreter-style architecture with the following components:



Engine

The Engine runs the game and can be divided into 2 parts:

1. The Virtual Machine (the interpreter)

The stack-based VM executes the game by being in a continuous loop. The engine shares game states with the resource manager and uses it for handling the graphics and sound subsystems, which are both triggered depending on the current game state. The game state is affected by the game events, which handle user inputs, such as the mouse movement or a key being pressed. It provides a set of extended functions for displaying graphics, playing sound, receiving input, writing and reading data to and from the hard disk, and handling complex arithmetical and logical functions [8].

2. The Kernel

"Kernel functions are the bridge between the abstract virtual machine, and the world of real programs" [9]. It is important to the engine as the kernel is necessary for the VM to read input and produce output. It is especially useful for loading resources.

The Resource Manager

The resource manager loads and manages game resources. The resources can include different types such as graphics (pic, view, font, cursor), sound (patch, sound), logic (script, vocab), and text [3]. Individual resources can be stored in one of two ways: Either in resource files or in external patch files [8]. Every other subsystem in the Sci engine's architecture depends on the resource manager. The resource manager depends on the engine's kernel for its functions. For example, the manager requires calling the kernel's load functions.

Graphics Subsystem

The graphics subsystem manages visual representation using 6 buffers (3 visual buffers, 2 priority buffers, and 1 control buffer) [5]. An important data structure used to handle graphics is ports. Ports record the states of the graphics, including storing things like pen color, current font, and cursor position [11]. One important buffer that is used by the interpreter is the control map, which is used to "check whether moving objects hit obstacles on the screen or touch zones with special meanings" [5]. The relationship between this subsystem and the interpreter is a two-way dependency. The graphics subsystem requires information about the state of the game to update the visuals for each frame, which will then be sent back to the VM. Furthermore, the graphics also rely on various kernel functions (ex. loading palettes, and setting flags) [10]. Like the other subsystems, the graphics also depend on the resource manager for its various resources such as cursor resources, view resources, (views are collections of images or sprites), font resources, pic resources (background images) [11].

Sound Subsystem

The sound subsystem handles sound resources. The ScummVM documentation describes the sound resource by stating music "is stored as a series of MIDI events, and the sound resource is basically just a MIDI file" [6]. An important part of this is the instrument mapping to the music, which is not always possible. That's why there are two types of resources for sound: patch resources, which do have instrument mapping, and sound resources, which do not adhere to General Midi mapping [8]. Like every other component, the sound subsystem depends on the engine for its kernel, and the resource manager.

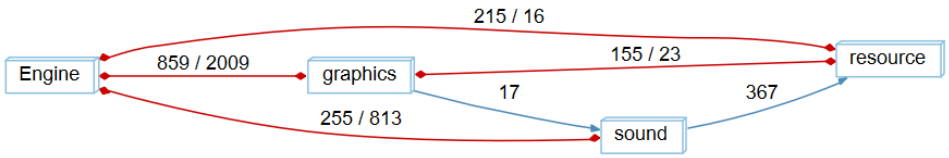


Figure 1. is the generated dependency graph created in Understand. It takes the directories engine/, graphics/, sound/, and resource/ and displays how many dependencies there are between each component, with red arrows being bidirectional.

Sci Engine Concrete Architecture

The concrete architecture of the Sci Engine is derived by using the Understand application, creating dependency graphs from resources in the Sci directory. Observing the dependency graph in Figure 1, we see that the conceptual architecture is fairly accurate. However, there are two dependencies in the concrete architecture that are not in the conceptual architecture:

1. The resource manager's dependency on the graphics subsystem
2. The graphics subsystem's dependency on the sound subsystem

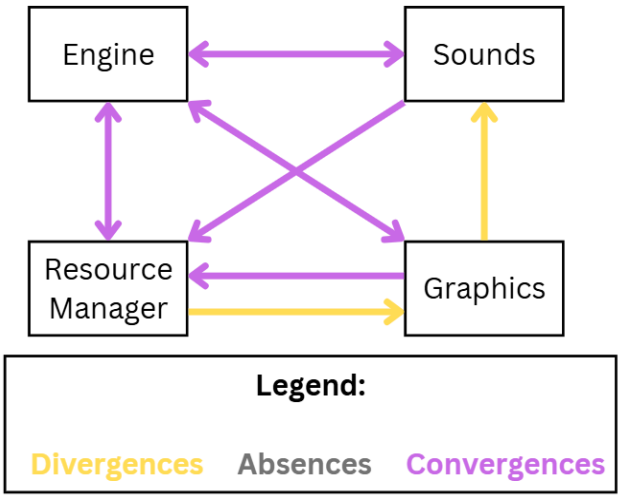


Figure 2. box-and-arrow dependency graph of the Sci engine concrete architecture with divergences, absences, and convergences highlighted.

Sci Engine Reflexion Analysis

Figure 2 displays the different dependencies between components, comparing the conceptual and concrete architectures. Divergences are the dependencies listed in both the conceptual and

concrete architecture, such as the bidirectional dependencies between the sound subsystem and the engine component. There are no absences, which are dependencies listed in the conceptual architecture but not the concrete. Most importantly here, there are two divergences, which are dependencies in the concrete architecture but not the conceptual architecture:

1. The resource manager depends on the graphics subsystem.

Referring to Figure 1, it is shown that there are 23 dependencies. All of these are related to an enumerator defined in \sci\graphics\helpers_detection_enums.h and imported from helpers.h. This enumerator is called ViewType, which simply lists keywords for different game types views, which is an important graphic resource, sorted by the number of colours. An example of this enumerator being referenced in resources is in \sci\resource\resource.h. This change was submitted in a commit by Max Horn on Feb 17, 2010 at 23:37:32, with the commit message simply being “Cleanup resource.h.” Despite this message not being helpful, it is clear that this dependency is so resources can access the ViewType enumerator. The reason for doing this, instead of redefining the enumerator or making it more global, may have been for reusability or to speed up the coding process. The latter is likely true due to the very large commit being submitted near midnight. It is possible this was a quick solution before a deadline.

2. The graphics subsystem depends on the sound subsystem.

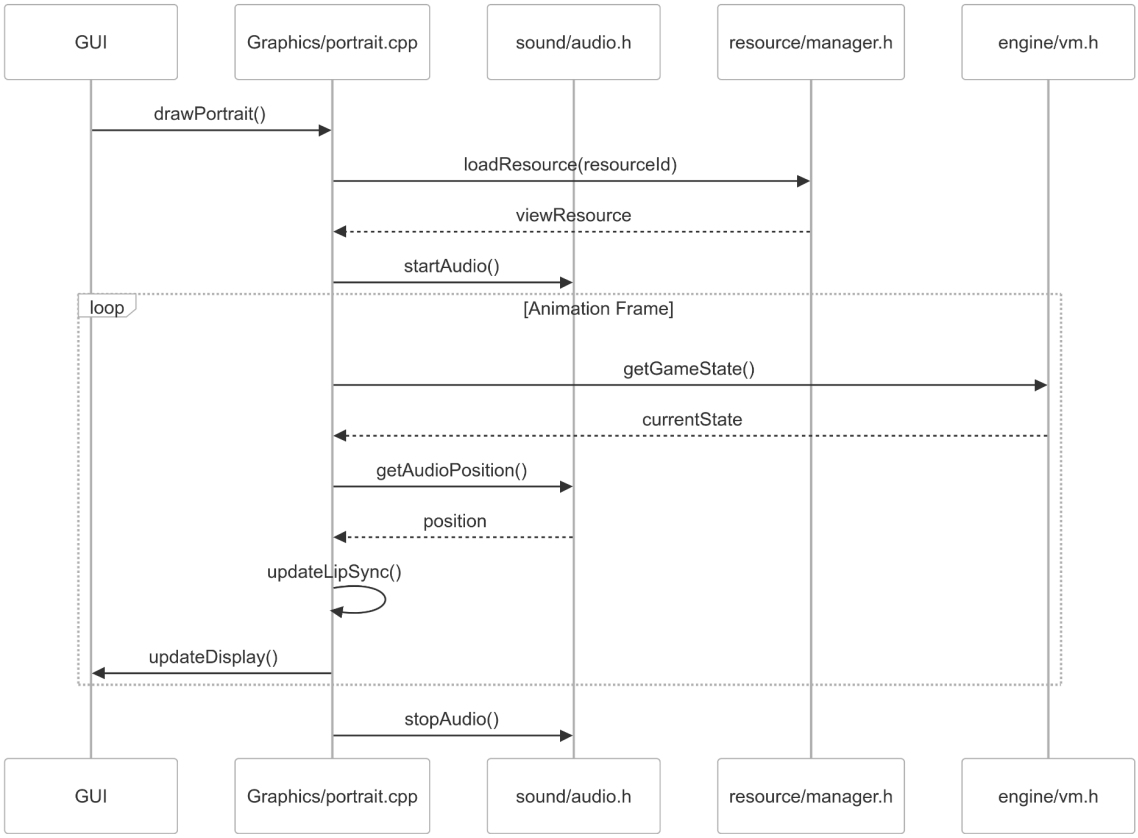
The 17 dependencies highlighted in Figure 1 are more diverse in reasons. One reason for the dependency is for lip syncing. The code in graphics/portrait.cpp imports from sound/audio.h, with the graphics code using sound functions such as startAudio(), stopAudio(), getAudioPosition(). The commit that added the import to portrait.cpp was done by Martin Kiewitz on January 6, 2010, at 18:18:52. The commit message states “adding audio support to

kPortrait, audio is now playing when portraits are shown." A file name that starts with 'k' is part of the kernel, meaning the dependency on the audio functions supports the kernel functions for the portrait file.

Use Cases & Sequence Diagrams

1. Portrait Animation with Lip Sync Use Case:

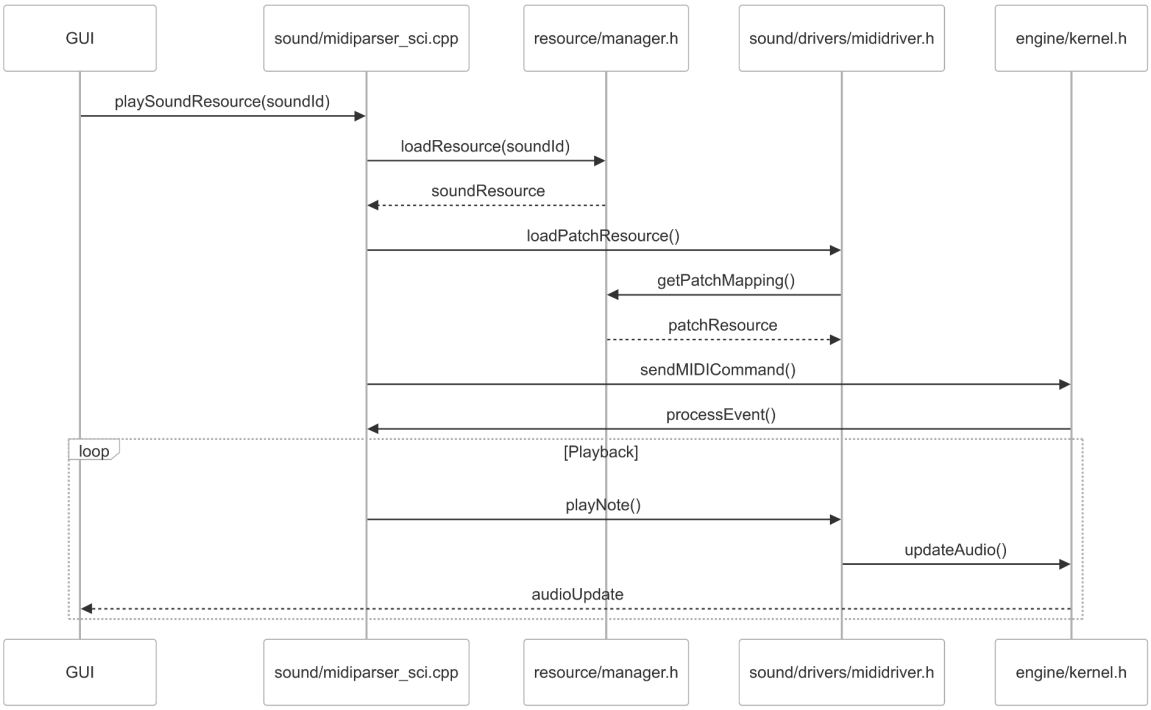
- Shows how graphics and sound components interact for synchronized character animations
- Demonstrates the concrete divergent dependency where Graphics calls Sound functions
- Uses actual method calls from portrait.cpp: startAudio(), getAudioPosition(), stopAudio()
- Shows the resource loading and VM state management process



2. Sound Resource Loading and Playback Use Case:

- Illustrates the SCI sound system's complex interaction between components
- Shows how MIDI resources and patch mappings are handled
- Demonstrates the kernel's role in sound processing
- Based on the document's description of SCI0 sound resources and MIDI events

- Includes both patch and sound resource handling as specified



Lessons Learned

From this assignment, many of our group members have learned how to use the SciTools Understand software. We've also learned more about how large C++ programs are structured from exploring the source code of ScummVM. Through this exercise, we also learned how critical it is to consider both the **conceptual** and **concrete** aspects of software architecture. The reflection analysis helped us identify areas where assumptions were made during the initial design process and provided insights into the complexities involved in implementing a modular and scalable system like ScummVM.

Conclusions

In conclusion, we've found that the concrete architecture of ScummVM differs from our proposed conceptual architecture presented in A1. Notably, several subsystems had unexpected

dependencies between them, while one of our proposed dependencies was absent in the concrete architecture. Beyond the absent dependency, ScummVM ended up much more tightly coupled than anticipated, which is quite common in large software projects. We also found the SCI engine to be similarly coupled, and with both analyses we tracked down the reasoning behind the additional or absent dependencies using reflexion analysis techniques.

References

[1] "ScummVM :: Home." *ScummVM*, www.scummvm.org.

[2] Understand: The Software Developer's Multi-Tool. https://scitools.com/

[3] "SCI/Specifications/Introduction." *ScummVM :: Wiki*, wiki.scummvm.org/index.php?title=SCI/Specifications/Introduction#Preface.

[4] "SCI/Specifications/SCI Virtual Machine/Introduction." *ScummVM :: Wiki*, wiki.scummvm.org/index.php?title=SCI/Specifications/SCI_virtual_machine/Introduction#Script_resources.

[5] "SCI/FreeSCI/Graphics/Architecture." *ScummVM :: Wiki*, wiki.scummvm.org/index.php/SCI/FreeSCI/Graphics/Architecture.

[6] "SCI/Specifications/Sound/SCI0 Resource Format." *ScummVM :: Wiki*, wiki.scummvm.org/index.php?title=SCI/Specifications/Sound/SCI0_Resource_Format.

[7] "SCI/FreeSCI/Basic Differences." *ScummVM :: Wiki*, wiki.scummvm.org/index.php?title=SCI/FreeSCI/Basic_differences.

[8] "SCI/Specifications/Introduction." *ScummVM :: Wiki*, wiki.scummvm.org/index.php/SCI/Specifications/Introduction.

[9] "SCI/FreeSCI/Kernel Hacking." ScummVM :: Wiki, wiki.scummvm.org/index.php?title=SCI/FreeSCI/Kernel_hacking.

[10] "SCI/Specifications/Graphics/Kernel Calls." *ScummVM :: Wiki*, wiki.scummvm.org/index.php?title=SCI/Specifications/Graphics/Kernel_calls.

[11] "SCI/Specifications/Graphics." *ScummVM :: Wiki*, wiki.scummvm.org/index.php?title=SCI/Specifications/Graphics.