CISC 322

Assignment 2 Report
ScummVM: Concrete Architecture

Group: BileBytes

Brandon Kim	21bmkk@queensu.ca
Ewan Byrne	21emb10@queensu.ca
Owen Sawler	21ors3@queensu.ca
John Li	21jl263@queensu.ca
Gavin Yan	21gly@queensu.ca
Norah Jurdjevic	20nrtj@queensu.ca

Abstract

ScummVM is an open source program enabling users to run and play classic adventure and role-playing games with only the original data files. The program rewrites the game's executables, thereby allowing users to play these games on systems for which they were not originally designed. In an effort to gain a stronger understanding of the concrete architecture we investigated the project files in the Github repository and with the Understand tool. We examined these files and updated our conceptual architecture to more accurately reflect the project's structure, and confirmed a concrete architecture according to our investigation. This report discusses these two architectures, their key interactions and examines any discrepancies between them, it outlines the internal architecture of the game engines subsystem with reflexion analysis and describes relevant use cases.

Introduction

ScummVM (Script Creating Utility for Maniac Mansion Virtual Machine) stands as a pioneering open-source project that enables modern platforms to run classic adventure games. Since its creation in 2001, ScummVM has evolved from a specialized interpreter for games written in the LucasArts' SCUMM engine into a comprehensive platform supporting numerous titles across an extensive library of platforms.

Following our initial investigation of ScummVM's conceptual architecture, this report presents a detailed analysis of the systems concrete architecture, derived through careful examination of the source code and directory structure using the Understand software tool. Our investigation revealed many significant differences between the conceptual and concrete architecture, particularly in the layering and dependency relationships between components.

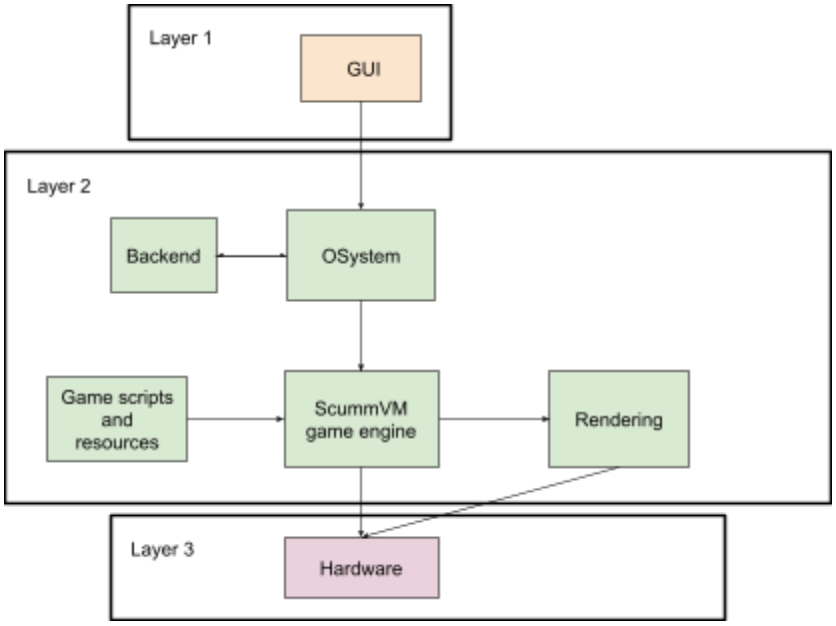
The concrete architecture demonstrates a transpicuous layered and interpreter style with a clear focus on the division of four primary layers: the Application Layer (base/), the User Interface Layer (gui/), the Game Engine Layer (engines/), and the Platform Abstraction Layer (common/system.h and backends/). These layers are supported by shared utility modules (common/, audio/, image/, video/, math/) that provide common functionality throughout the system. While our conceptual architecture captured some of these relationships, our analysis of the concrete implementation revealed several unexpected dependencies and interactions, both in the layer organization and the component interactions. Some key differences include the presence of a previously undefined Base component at the top layer, a reversed dependency relationship between some components, such as the Game Engines calling the OSystem rather than vice versa, and numerous direct cross-layered communications that were not present in our conceptual model.

To provide deeper insight into ScummVM's architecture, our team conducted a detailed investigation of the Game Engine subsystem. This analysis revealed the intricate relationships between the game engine's implementation and ScummVM's core infrastructure, particularly in how each engine functions as an interpreter for its respective game format while maintaining consistent interfaces with the platform abstraction layer and shared utilities.

Using the software reflexion framework, we conducted two distinct analyses comparing our conceptual and concrete architectures at the system-wide and subsystem levels. Our analysis revealed significant differences in both areas, particularly in component interaction and structural organizations. These findings are supported by detailed sequence diagrams of two fundamental use cases: playing a game through ScummVM and adding new games to the platform which demonstrate the key interactions between components in practice. Finally, we conclude by summarizing our key findings regarding ScummVM's architectural design choices, as well as detailing noteworthy limitations and lessons learned throughout our analysis.

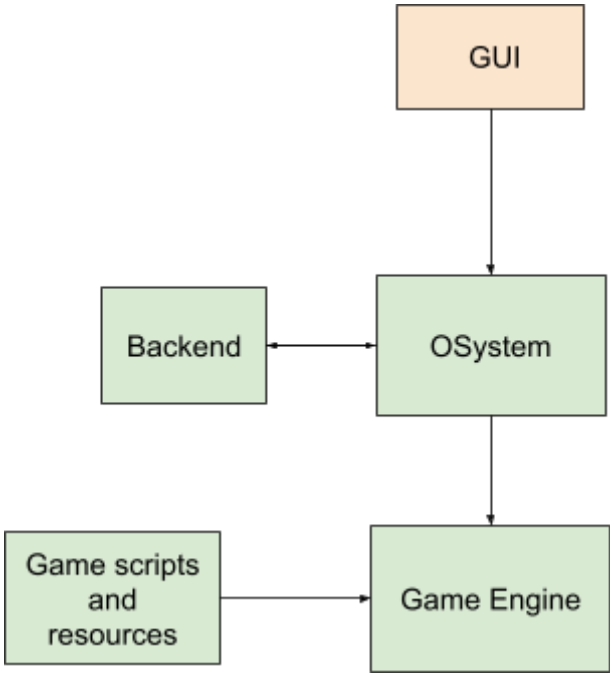
Conceptual Architecture Revision

Our original conceptual architecture from assignment 1 can be seen below:



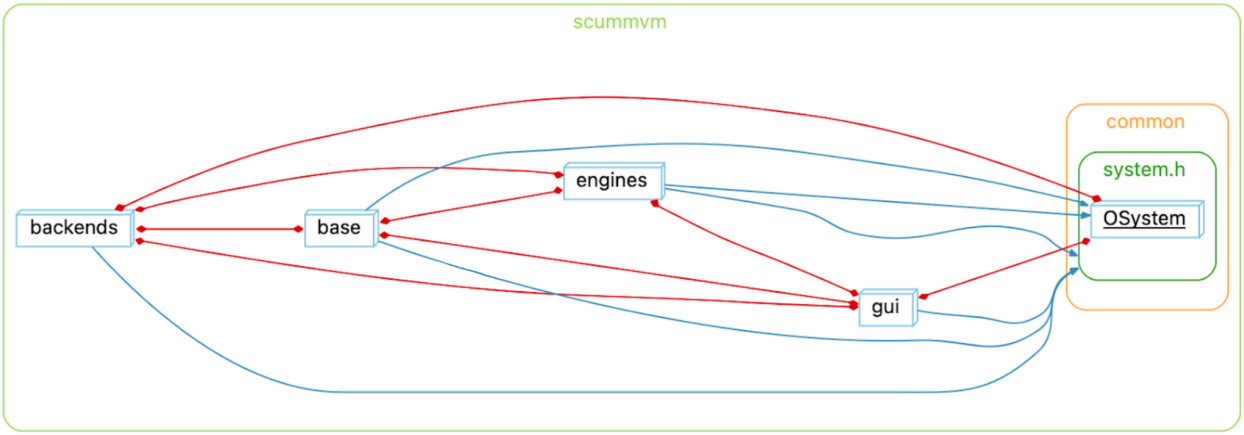
We chose to make a few changes to this architecture. Firstly, we removed the layers from the diagram, since this makes it easier to compare the diagram to the dependency graph in Understand. We are still keeping the layered style as part of our architecture, but the layers aren't a requirement in the box and line diagram, so this is more of a visual change than an architectural change. We also removed the hardware component, since the hardware is not part of the software architecture. These changes were made to make the conceptual architecture structure more

similar to the file system structure of the ScummVM repository, making it easier to identify missing dependencies and unexpected dependencies. The revised architecture is as follows:



Concrete Architecture and Derivation

ScummVM's concrete architecture can be seen below:



There are four layers:

1. Application Layer (base/)
2. User Interface Layer (gui/)
3. Game Engine Layer (engines/)

4. Platform Abstraction Layer (common/system.h, backends/)

All of which use shared utilities (common/, audio/, image/, video/, graphics/, math/).

The ScummVM project follows a layered and interpreter-style architecture, with clearly defined subsystems that interact to create a portable, extensible environment for running games. At a high level, the architecture can be divided into four primary layers: Application Layer (base/), User Interface Layer (gui/), Game Engine Layer (engines/), and Platform Abstraction Layer (common/system.h and backends/). These layers are supported by shared utility modules (common/, audio/, image/, video/, math/), which provide common functionality used throughout the system. Each layer has distinct responsibilities, and the interactions between them primarily follow a top-down approach, consistent with a layered architecture. However, the nature of the game engine layer introduces additional interactions that are reflective of an interpreter-style.

Derivation of Concrete Architecture

The layers are determined by analyzing dependency calls between files inside components' folders using the Understand software. Note that while there exist calls from lower layers to upper layers, and from utilities to other components, they are either significantly fewer than the reverse, or they are from files that are not representative of the component. For example, of the calls made to the base/ folder, they are all bidirectional, and primarily targeted towards the plugin.cpp and version.cpp, rather than the main.cpp, which acts as the entry point for the application, and is primarily what we are referring to as the Application Layer. Aside from observing the codebase through dependency calls, ScummVM also provides a wiki for developers wishing to contribute to the codebase, which gives us a high-level understanding of the concrete architecture. Additionally, the code is publicly available on GitHub, allowing us to observe various implementations of game engines.

Interactions Between Subsystems

The architecture of ScummVM is organized using a layered approach, allowing for clear separation of responsibilities and modular design. At the top level, the Application Layer (base/) acts as the entry point for the program. It handles command-line argument parsing, initializes the system, and manages the main application loop. This layer essentially oversees the entire flow of the application, orchestrating interactions among other subsystems.

Below the Application Layer lies the User Interface Layer (gui/), which is responsible for presenting the graphical user interface to users. It includes menus, dialogs, and various interactive elements that allow users to select games, configure settings, and manage saved games. This layer interfaces directly with the Application Layer to respond to user inputs and update the display accordingly.

The Game Engine Layer (engines/) comprises the core implementations of the various game engines supported by ScummVM. Each subdirectory within engines/ corresponds to a specific game engine, such as SCUMM, AGI, or SCI, responsible for interpreting the game data files and driving gameplay. These engines interact with the Platform Abstraction Layer for platform-specific tasks and utilize shared utilities for common functionalities like graphics and audio processing.

The Platform Abstraction Layer (common/system.h and backends/) provides a unified interface (OSystem) for handling platform-specific details, making ScummVM compatible with multiple operating systems such as Windows, macOS, and Linux. The common/system.h file defines the platform API, while the backends/ directory contains specific implementations for each supported platform. This layer abstracts away the underlying system differences, allowing higher-level code to remain platform-agnostic.

ScummVM also employs shared utilities, which include various common modules such as common/, audio/, image/, video/, math/ and graphics/. These provide reusable components for tasks like file handling, audio processing, image decoding, video playback, and graphics rendering, this ensures consistency and reduces code duplication across different game engines and platform implementations.

High-Level Architecture Reflexion Analysis

After completing our concrete architecture implementation of ScummVM, our team conducted a reflection analysis by investigating the divergences between our conceptual and concrete architectures. Our conceptual architecture was designed with three distinct layers: User Interface layer at the top, Game engine layer in the middle, and the Hardware Abstraction layer at the bottom. However, the analysis revealed several significant discrepancies in both the layering structure and between component interactions.

One of the most notable differences is the presence of a base/ component at the top of the Application layer, which was not anticipated in our original conceptual architecture. The concrete architecture demonstrates that the Base component maintains substantial connections with multiple core components including backends/, engines/, and the gui/ system. This allows for the Base component to take responsibility for a central hub, allowing for shared functionality across the codebase and an entry point for the program. This inconsistency between the conceptual and concrete architectures likely arose from an initial underestimate on the amount of common functionality needed across components when designing the conceptual architecture. Recognizing this gap, the inclusion of the Base component becomes essential. It prevents code duplication, provides a clean way to share essential utilities across the system, and oversees the

application flow of the entire system, arranging interactions amongst subsystems and handling critical responsibilities such as system initialization and command line argument parsing. However, the monolithic structure suggests that the Base component may take on too many responsibilities and could probably benefit by being broken down into more focused subsystems.

Another significant structure divergence is the repositioning of the User Interface layer (gui/). While our conceptual architecture placed it as the topmost layer, the concrete implementation shows it as the second layer, underneath the Base component. This inconsistency emerged from the empirical need for the UI to access common utilities and services provided by the Base component. While this reorganization is justified from a functional perspective, it represents a departure from our original clean layered design and suggests that our initial understanding of the UI's dependencies was incomplete.

Perhaps the most concerning divergence relates to the positioning of the Backend and OSystem components. In our conceptual architecture, these components were placed in layer two alongside the Game Engine component, with clear and limited interaction paths – the GUI would communicate with the OSystem, and the OSystem would communicate with the backend, acting as a middle man. However, we noticed that the backends/ and OSystem, implemented through common/system.h were moved to the Platform Abstraction Layer, providing a unified interface for handling platform-specific details, such as compatibility with multiple operating systems. This revealed more complex patterns within the concrete architecture, specifically regarding direct communication paths between components like the GUI and Backends and the GUI and Engine, violating our initial design. Instead of routing through the OSystem, the gui/ directly connects to the backend/ and engine/ components, allowing for efficient access to key graphical UI elements corresponding to specific game engines, such as game settings, menus, dialogs, and other interactive elements. This inconsistency likely arose from our previous conception of a practical layered implementation. We thought that the OSystem API was responsible for serving backend utilities including audio/, image/, video/, math/, and graphics/ to other components which we defined in the lower layers of our conceptual architecture. However, this was not the case and these new direct communication paths enable immediate access to graphics and event handling, more responsive UI updates, more efficient game file interpretation and execution, and the direct sharing of utility modules through adjoined components. Despite this change, reducing the number of unnecessary abstraction layers also compromises the clear separation of our layered architecture and creates tighter coupling between components we believed were meant to be more isolated.

Our analysis also uncovered several unexpected bi-directional interactions between components. While we initially envisioned a strictly layered architecture with clear hierarchical dependencies, the concrete implementation revealed more complex relationships. For instance, the OSystem and GUI components developed a bidirectional communication pattern rather than

the one-way interaction we had envisioned previously. This divergence emerged from the practical necessity of system event handling – while the GUI needs to make system calls, the OSystem is also required to notify the GUI of system events and state changes. This bidirectional dependency is justified as it enables responsive UI updates and proper event handling as well reveal a more interconnected and pragmatic system than our original conceptual architecture suggested. While some aspects of our layered design were preserved, the newfound direct communicates paths allow for more shared functionality, resulting in a more efficient system.

Second Level Subsystem Reflexion Analysis

To better understand ScummVM, we investigated the conceptual and concrete architectures of the game engines subsystem. This subsystem is a crucial element of ScummVM, as it is the core component of the emulation framework that enables ScummVM to run classic games on modern systems. Each engine is responsible for interpreting the game files from their original engines and executing the game logic as defined.

The primary purpose of the engines/ subsystem is to encapsulate the game-specific logic, file handling and rendering mechanics needed to emulate the original behavior of each game. Each supported game engine is implemented as a subclass of the Engine class, which provides basic methods such as run() for the main event loop, and loadGameStream() for handling save and load functionalities, among others.

This subsystem follows the interpreter architectural style, with each ScummVM game engine implementation acting as an interpreter. The engine processes the original game files, typically written in specialized languages, which contain the game's logic as well as responses to player actions. In some cases, such as SCI, the engine uses a parser to convert the original grammar rules into a form suitable for ScummVM. This ensures the scripts can be interpreted in real time, allowing ScummVM to determine the appropriate action to take as needed. Each game engine is then able to interpret high level game scripts into specific low level instructions which the system can execute.

Concrete implementations for specific engines extend the base class to define game-specific behaviors and interactions. Each engine includes a metaengine.cpp file, containing the AdvancedMetaEngine subclass, which interfaces with the ScummVM launcher, providing features like game detection, savegame management, and additional metadata. It also implements plugin registration with the help of macros like REGISTER_PLUGIN and REGISTER_ENGINE, making the engine available to ScummVM's core plugin system. Additionally, these engines typically include helper files such as detection.cpp, which define the criteria for identifying compatible game files using MD5 hashes, filenames, and specific detection logic. Any interaction between game engines and the backend (e.g. GUI or input

handling) is managed through ScummVM's infrastructure services, such as common/events.h, which allows engines to implement their own event handling (e.g., input loops within run()) while still relying on ScummVM's common event framework to ensure consistency and portability.

These engines act as isolated modules which use ScummVM's common libraries for cross-platform support and multimedia handling. For instance, graphics rendering frequently uses helper functions from common/system.h to draw onto the screen, whereas audio is managed using pre-existing decoders in sound/. This separation of concerns allows different game engines to reuse the same backend utilities while implementing distinct game logic.

Reflexion Game Engines

The architecture for engines will be the main focus of the localization subsystem of ScummVM as it is a key component within the overall architecture of ScummVM. The key difference between the concrete and the conceptual architecture for the game engine would be how ScummVM saves a game. When a user saves their game, the /engine subsystem is not solely responsible for this action, but instead relies on the metaengine.cpp file, which contains the AdvancedMetaEngine subclass to save the current state of the game. As a result, the sub system AdvancedMetaEngine, is responsible for managing and saving files to ScummVM instead of the game engine. Through this, the engine component has an additional sub system within it in its concrete architecture.

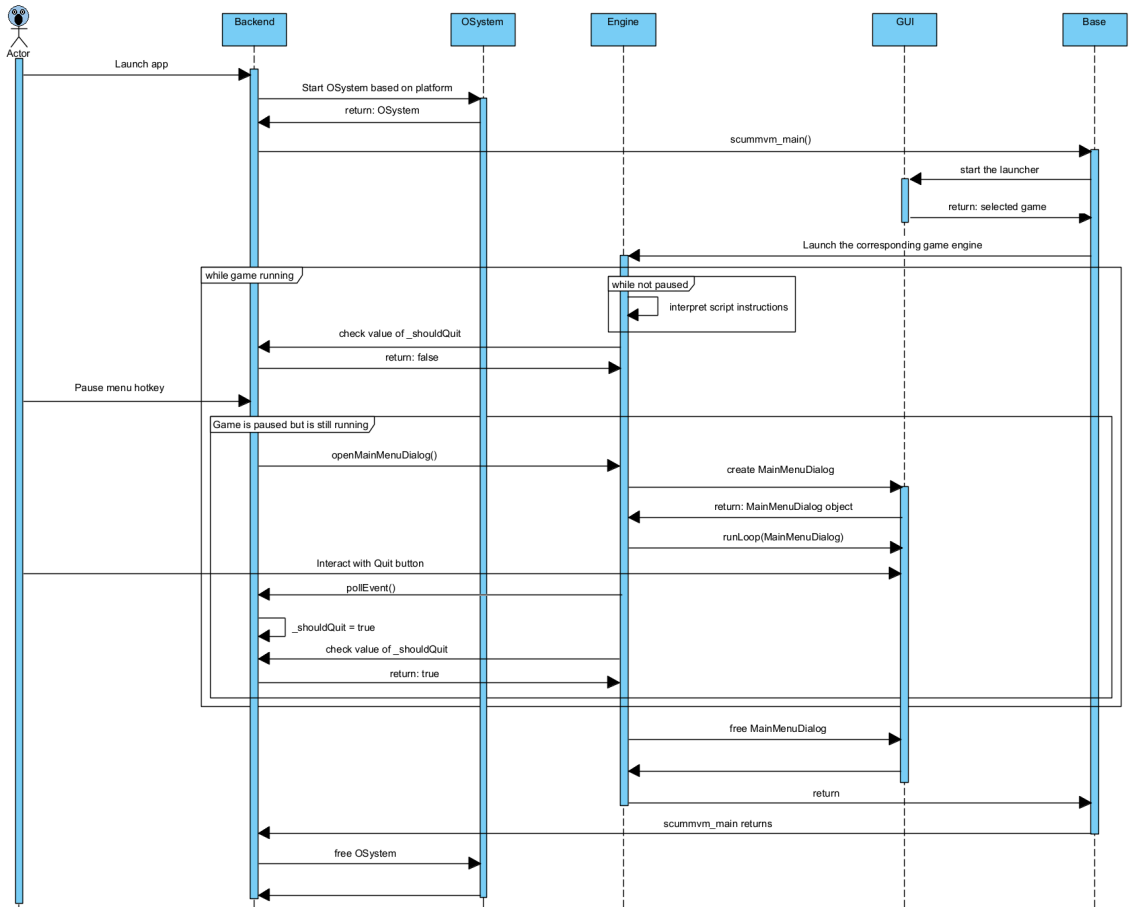
Additionally, it was discovered that the engines/ subsystem makes calls to the common/system.h, instead of the common/system.h calling to the engines/ subsystem. As a result, the engines/ subsystem relies on the common/system.h component instead of the other way around, causing a new dependency between the two. In the conceptual Architecture it was initially thought that the common/system.h needed to make a call to the engines/ subsystem to identify and use the correct engine to run the game. However, this was proven to be false as the engines/ subsystem is able to directly communicate with certain components that were initially not thought of and can call on the common/system.h. Because of this, the engines/ subsystems have a dependency with the common/system.h.

When looking at the interactions of the engines/ subsystem, it was also discovered that the engines/ subsystem directly interacts with the backend/ subsystem instead of it being reliant on the common/system.h to manage the information between the two. As a result, the engines/ subsystem is dependent on the /backend subsystem for creating graphics, backend/graphics and backend/graphics3d, and events, backends/events. As mentioned above, it was thought that the engines/ subsystem needed to be called by the common/system.h subsystem in order to be utilized, but instead it is able to directly call on other subsystems.

Finally, the /GUI and the engines/ subsystems directly interact with one another. As a result, the /GUI subsystem has a dependency on the engines/ subsystem to provide information on the game in order to display this information for the user. As a result, it does not need to rely on common/system.h to send information from the two. Instead, the GUI/ subsystem is able to directly connect with the engines/ subsystem.

Use Cases

Use case 1: Playing a game through ScummVM



This first sequence diagram showcases the typical game loop in ScummVM. It starts off with launching the application, then covers game selection, the gameplay loop and quitting the game.

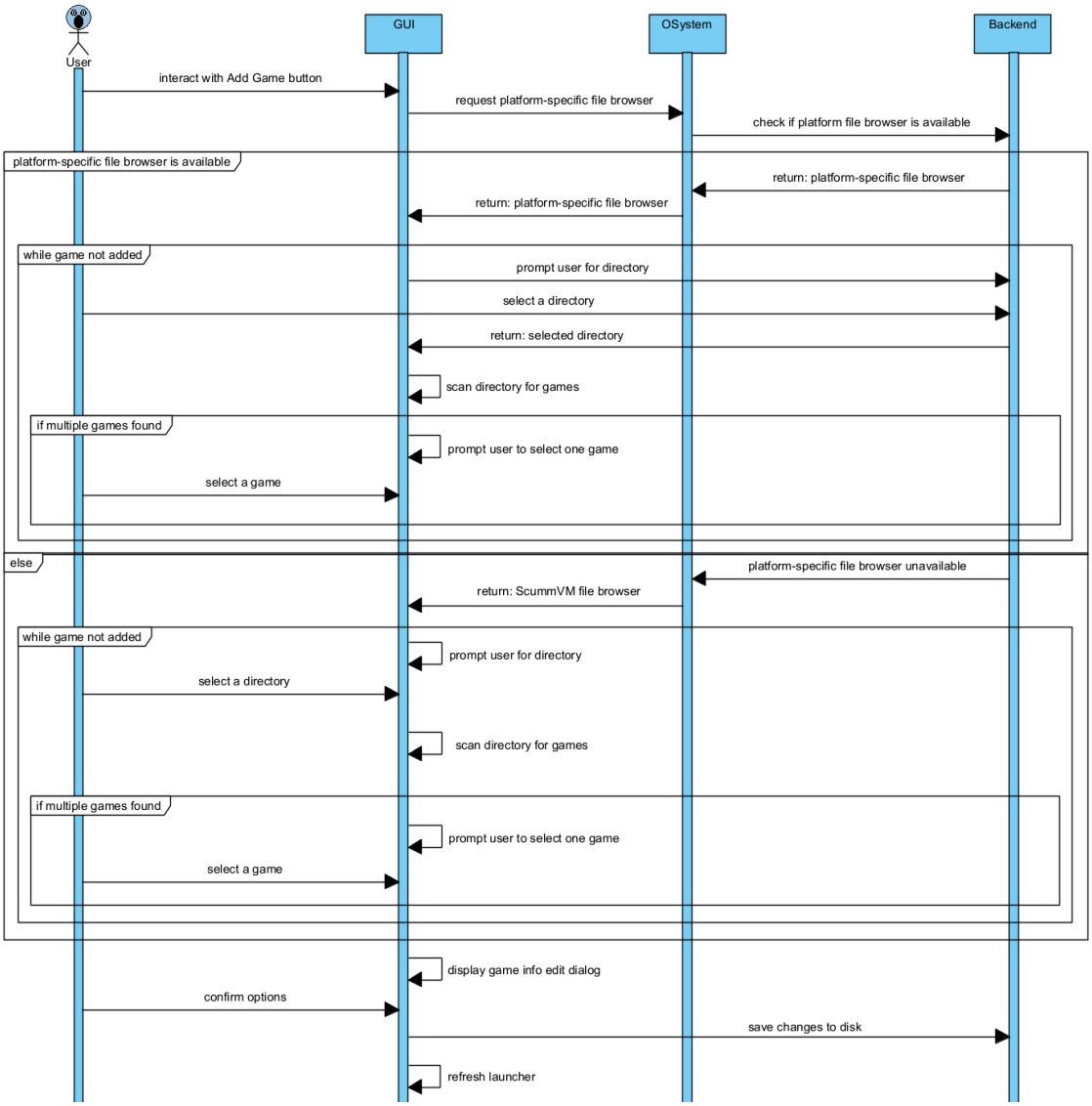
The main() method for the program is located in the Backend component, since the ScummVM executable works differently on different platforms. The platform-specific main() method starts up the OSystem based on the current backend, before calling the general scummvm_main() method from the Base component. If no game is specified, then this general main method makes a call to the GUI to start the game launcher. Here, the user can select a game, which will be returned to the Base component for execution. The Base component will

call the Engine component to launch the game engine that corresponds to the selected game. The game engine will then interpret the instructions from the original game's scripts repeatedly, looping through these instructions until the user invokes some sort of interruption.

When the user wishes to quit the game, they first need to invoke the ScummVM pause menu. This requires a platform-dependent input. As such, the input is registered in the Backend component. This calls the openMainMenuDialog() method in the Engine component, which creates a MainMenuDialog object (if it doesn't already exist), which is a subclass of Dialog. The Dialog class is part of the GUI component. Then, the Engine component invokes a loop in the GUI component, which processes the top Dialog from a stack of dialogs, and will do so until the stack is cleared, looping every frame. In other words, the pause menu interface will be displayed so long as this loop is running. The user can break the loop by interacting with certain GUI elements that are part of the MainMenuDialog object, such as the Quit button.

Once the user confirms that they want to quit, the GUI component will call a pollEvent(EVENT_QUIT) method in the Backend component, which notably sets the _shouldQuit variable to true. The Engine component, which constantly checks the value of _shouldQuit , will break out of the game loop once the value has changed. All of the active method calls in the call stack will complete their execution without any significant actions, and will return. This eventually unwinds back to the platform-specific main() method, which will free the OSystem and then terminate, returning an exit code of 0.

Use Case 2: Adding a Game to ScummVM



The second sequence diagram is for adding a game to ScummVM. This is an action that is taken from the launcher, so we will start by assuming that the launcher is currently open. Additionally, we can assume that the scummvm_main() method from the Base component is running throughout this sequence diagram, however this component was not included in the diagram since it isn't involved in adding games. We can also see from the previous sequence diagram that the OSystem and Backend components are running while the launcher is active, which is why they are active throughout this sequence diagram.

The user will interact with the add game button, then select a game to add. ScummVM will add this game to its internal library of games that the user can then quickly select to play. For this example, we will assume that the user doesn't use the "Mass add…" feature, meaning they are only adding one game to their library, since this is likely the more common way to add a game among users of this software.

First, the user selects the Add Game button on the launcher. This interaction is handled by the GUI component. ScummVM will try to bring up the platform-specific file browser, which is done by making a call to OSystem, which makes a call to the Backend component to get the platform-specific file browser. This will be returned to the GUI component so it can continue its operations. However, this might not be possible depending on the user's platform or settings. In this case, ScummVM will use its own file browser, which is handled in the GUI component. Regardless of which file browser is used, a loop is started.

The first action in this loop is to prompt the user to select a directory. The GUI component will scan the selected directory for games that it can recognize, with the goal of finding a unique result. If multiple results are found, the GUI will prompt the user to select one result from a list of options. Once a unique result is chosen, the loop will break. If the GUI component cannot detect any games in the directory selected by the user, the GUI will inform the user that no game was found and the loop will repeat.

Once the loop is broken, the GUI will display a dialog which lets the user edit some default options relating to the game, notable examples being controls and a modifiable description for the game. Upon confirming the options, the changes will be saved to the disk, so the game will be added to the internal game library. The GUI component refreshes the part of the launcher that displays the games available, and the user is now able to quickly select their game to play whenever they launch ScummVM.

Conclusion

Through careful examination of the project's Github repository and analysis of its files with the Understand tool, we developed a stronger conceptual architecture as well as a concrete architecture for ScummVM. We reorganized our original conceptual architecture to facilitate comparison with dependency graphs, altering it to follow a similar structure to the ScummVM file system. The concrete architecture we derived from these files is composed of four layers, the application layer, the user interface layer, the game engines layer and the platform abstraction layer, which work together along with shared utility modules to form ScummVM. Our research outlines discrepancies between these conceptual and concrete architectures and performs a reflexion analysis to elucidate the nature of these differences. The concrete architecture is further illuminated by investigation of the game engines subsystem and a reflexion analysis of the discrepancies in the subsystem's conceptual and concrete architectures, as well as the illustration of two key use cases. The report provides a systematic and comprehensive analysis of ScummVM's concrete architecture, with detailed discussion of any inconsistencies discovered and careful analysis of the key components required for its functionality.

Lessons Learned

The dependency graph generated by Understand from the ScummVM codebase provided a very clear and easy way to develop a high-level understanding of the codebase and concrete architecture of the software. The source code, on the other hand, provides a lower level, more complete understanding of the software. These can be used together to derive the concrete architecture in a fast and effective manner.

We found while working on the project that doing research individually and then discussing our findings relating to the concrete architecture, in-person, was an effective way of maximizing the generation of ideas while minimizing confusion later on while diving deeper into the architecture and writing the report. Working independently on different sections of the report while maintaining communication with other people working on related sections was a good way of ensuring that the architecture remained cohesive, despite working remotely and independently.

Data Dictionary

Open Source: Software with freely available original source code which may be redistributed and modified.

Naming Conventions

SCUMM – Script Creation Utility for Maniac Mansion
SCI – Sierra Creative Interpreter
API – Application Programming Interface

References

- ScummVM Github Repository. Retrieved from https://github.com/scummvm/scummvm
- ScummVM Wiki. 2023. HOWTO-Engines. Retrieved November 15, 2024 from https://wiki.scummvm.org/index.php?title=HOWTO-Engines
- ScummVM Wiki. 2023. Developer Central. Retrieve November 15, 2024 from https://wiki.scummvm.org/index.php?title=Developer_Central