ScummVM Concrete Architecture Report

Authors - Group 11

Andrew Clausen	21awc7@queensu.ca
Arda Utku	21au10@queensu.ca
Arianne Nantel	20aan8@queensu.ca
Daniel Dousek	22dd1@queensu.ca
Kyra Salmon	21kms22@queensu.ca
Oliver Macnaughton	20owdm@queensu.ca

Abstract

In this report, we explore the concrete architecture of ScummVm (a platform that allows users to play games on a variety of platforms for which they were not created for) and compare it to its previously showcased conceptual architecture. We delve into the ScummVm codebase using the Understand software to highlight the convergences (dependencies that we expected to be in the architecture were present in the concrete architecture), the absences and the divergences between the conceptual and the concrete architecture of ScummVM. Through this process, we confirmed that ScummVM is best represented by a layered architectural style as expected by our previous work. While many dependencies were expected we also encountered many unexpected dependencies. We explored in detail the reason for these divergences between the concrete and conceptual architectures, digging through the code to determine, what are their causes and if they are justified.

Furthermore, we explored one of the ScummVm subsystems, namely the Sierra Creative Interpreter (SCI) engine. Building on what we discussed in our initial analysis, we again used Understand to confirm that the SCI engine utilizes an Interpreter architecture style. Looking through the code, we found many unanticipated bidirectional dependencies, which we visualized in a dependencies graph.

In addition, we talk about ScummVm's external interfaces. The external interface allows the user to interact with the system through mainly the GUI, the file system and a network. This allows users to load their games, set game options (like sound volume, graphics quality, etc…) and save game files to a cloud-based service.

Finally, we conclude by reviewing our use cases. Revisiting these allows us to better describe the functionality of key game mechanics such as loading and saving a game, according to the concrete architecture. Allowing us to better reflect on how ScummVM accomplishes these tasks.

Introduction and Overview

As previously denoted in our conceptual architecture report, ScummVM is a volunteer developed program which allows users to run adventure games on systems which they were not designed for. Picking up where the conceptual architecture report left off, this report goes on to detail our findings within the concrete architecture. Additionally, this deeper understanding allows us to delve deeper into the workings of the SCI engine.

The concrete architecture covered in this report was derived with the assistance of Understand software. This tool allowed us to upload the files of interest into the software and begin re-structuring it to match our conceptual layered architecture. Given this newly built structure we were then permitted to conduct a reflection analysis and draw a number of comparisons to our former understanding of ScummVM. We determined the necessity for the renaming and re-structuring of our layers; The largest change being the addition of the ‘common’ layer. The common layer contains modules and files required by all other layers within the ScummVM architecture. It additionally allowed us to help make sense of some of the many bi-directional dependencies that we hadn’t previously accounted for in our conceptual architecture before reviewing the code.

A large section of this report is dedicated to investigating divergences. (A dependency between components where one was not previously anticipated based on the conceptual architecture.) The aforementioned divergences are each regarded individually to determine their purpose. This allows us to determine why they may have been added and the proposed validity of each dependency as far as maintaining the ScummVM system requirements. These divergences are located in all different components of ScummVM, however, we were able to determine a valid justification for the existence of each. Despite the large number of divergences, we were able to conclude that the convergence between the conceptual and concrete architecture solidified our beliefs that ScummVM is a layered architecture as we had previously stated. All anticipated dependencies were easily located using the Understand tool.

Using a number of detailed figures, we further discuss the Sierra Creative Interpreter (SCI) engine, building on the information we had gathered in the previous report. It is a crucial subsystem used within ScummVM, and without it developers would have been forced to build one entirely from scratch (a demanding task on a volunteer developer project). The SCI engine functions as an interpreter within ScummVM, this is essential for translating scripts from their original OS software to something run via Scumm. (The exclusive goal of the software.)

Within the layered architecture style of ScummVM, the SCI engine runs with its own interpreter architectural style. It depends on a number of bi-directional function calls between its components. These dependencies include communications between the interpretation engine and the program state, the internal state interpreter and the program being interpreted, the program being interpreted and the program state, and the internal state interpreter and the program state.

Another essential interactive process within ScummVM, are the external interfaces. These are primarily generated for user interaction through the GUI. This is how the user/client is able to interact with Scumm, indicating their preferences (things such as settings) and uploading or receiving information. The user will have to complete tasks through the external interface such as loading the game file(s) they desire to play into the software so that Scumm can complete the necessary manipulations.

Use cases are also provided at the end of the report. These use cases remain consistent with those in the previous report, thus permitting us to notice the changes and compare between the conceptual and concrete architecture.

Architecture

Concrete Architecture Derivation Process

We derived our concrete architecture for ScummVM by loading the dependency files into the Understand tool. Due to our successful conceptual architecture (based on feedback), we decided it best to use both the architectural styles and some of the top-level subsystems such as GUI and the SCI engine in order to maintain consistency between projects. Upon loading the ScummVM project into Understand, we used these subsystems described in assignment 1 to group folders and files based on functionality (what the code did), dependencies (what calls the code made to other files), and contextual clues (folder/file name, language, etc.). In assignment 1, we proposed that ScummVM architecture was layered (see Table 1), and we also proposed a dependency diagram between the top-level components of the application. By having these conceptual materials from assignment 1, and seeing the source code dependencies in Understand, we were prepared to begin developing our concrete architecture for ScummVM.

Presentation Layer **GUI** Seen as the ScummVM launcher or running game. This layer is responsible for the user interface, and allows the user to interact with the application. The presentation layer is coordinated by the operating system, with possible help from rendering or graphics libraries such as OpenGL or SDL.
Operating System **MacOS, Windows, etc.** Manages operating system resources such as window management and input handling (resources needed by presentation layer.) Implements platform-specific tasks sent by the Abstraction layer, such as rendering updates to state in GUI. Outputs commands and resources for rendering and presenting information.
Abstraction Layer **OSystem API** Allows communication between SCI engine and operating systems, handles platform specific commands, essentially converting output of logic layer into instructions that can be interpreted by the operating system. OSystem API is dependent upon ScummVM "backends", the code that instantiates the API for a specific OS, allowing the game engine to receive input from and send commands back up to the operating system.
Logic Layer **SCI Engine** Game engine, responsible for interpretation of game files, processing of game logic, tracking of game state, handling events (button press, mouse click, etc.). Essentially controls the behavior of the game and how it runs. Sends output (new game state) to OSystem API to be represented in GUI. Receives input from user-loaded game files and OSystem API calls.
Data Layer **Game Files from Local System** Game files, scripts, animations, sprites, graphics, all specific to the game. The game files are user provided, and ScummVM must be given access to the location of the users local game files in order to load them into the engine. The data layer represent the building blocks of the game that are used in order to run it.

Table 1. Layered architectural style, discusses top-level subcomponents

Conceptual vs. Concrete: Reflexion Analysis

We decided to compare our proposed conceptual architecture and the conceptual dependency diagram with the concrete architecture revealed by placing ScummVM in Understand and grouping files into top-level subsystems.

Given the existence of discrepancies between our dependency diagrams in Fig. 1 and Fig. 2 we were able to perform a reflexion analysis. (Fig. 3)

First, there are a lot of discrepancies in terms of unexpected dependencies. This type of discrepancy is called a divergence. In this next section, we will go over each divergence and provide a rationale for why/if this unexpected dependency is necessary.

On

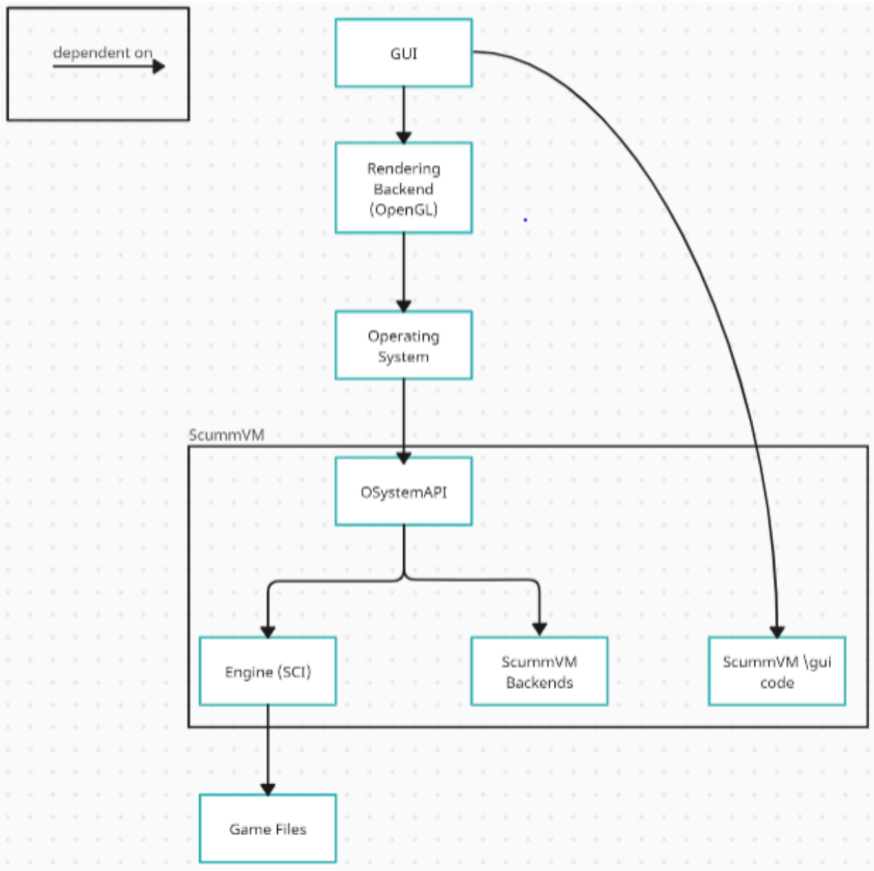


Fig 1. Dependencies in Conceptual Architecture

the positive side of things, many of the dependencies found in the concrete architecture are convergences (see Fig. 1), and we have no instances of absences (missing dependencies). Thus, we will focus on providing rationales for the divergences we have noted.
Before going over the divergences, we should clarify the addition of the submodule “common”, and the slight modification of the layout of the dependency diagram. “Common” is a collection of utility classes and codecs (compressors/decompressors) for audio, image, and video resources. Inside the module are the following folders: common/ (except for `system.h` which

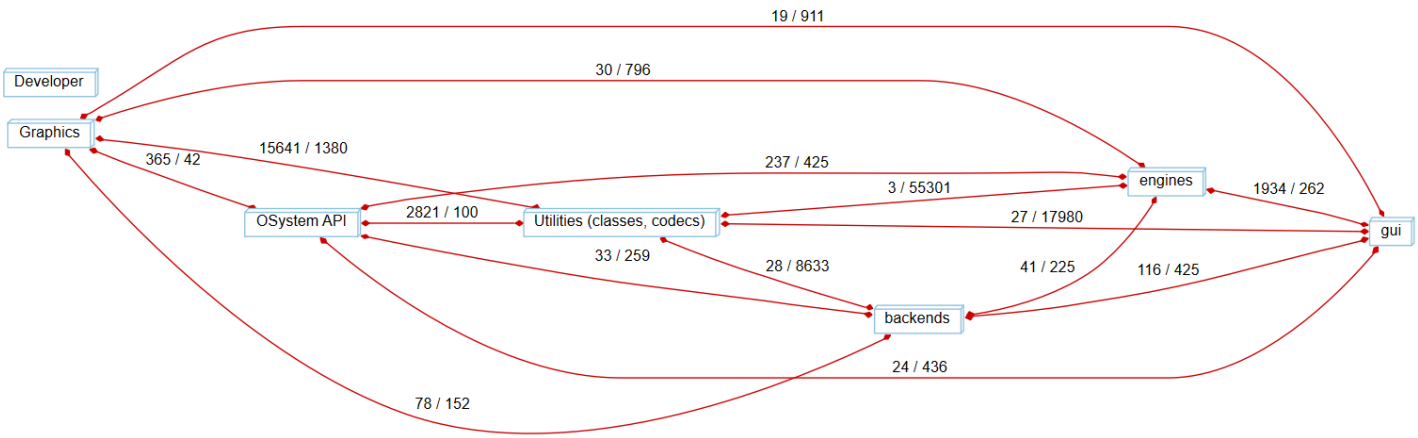
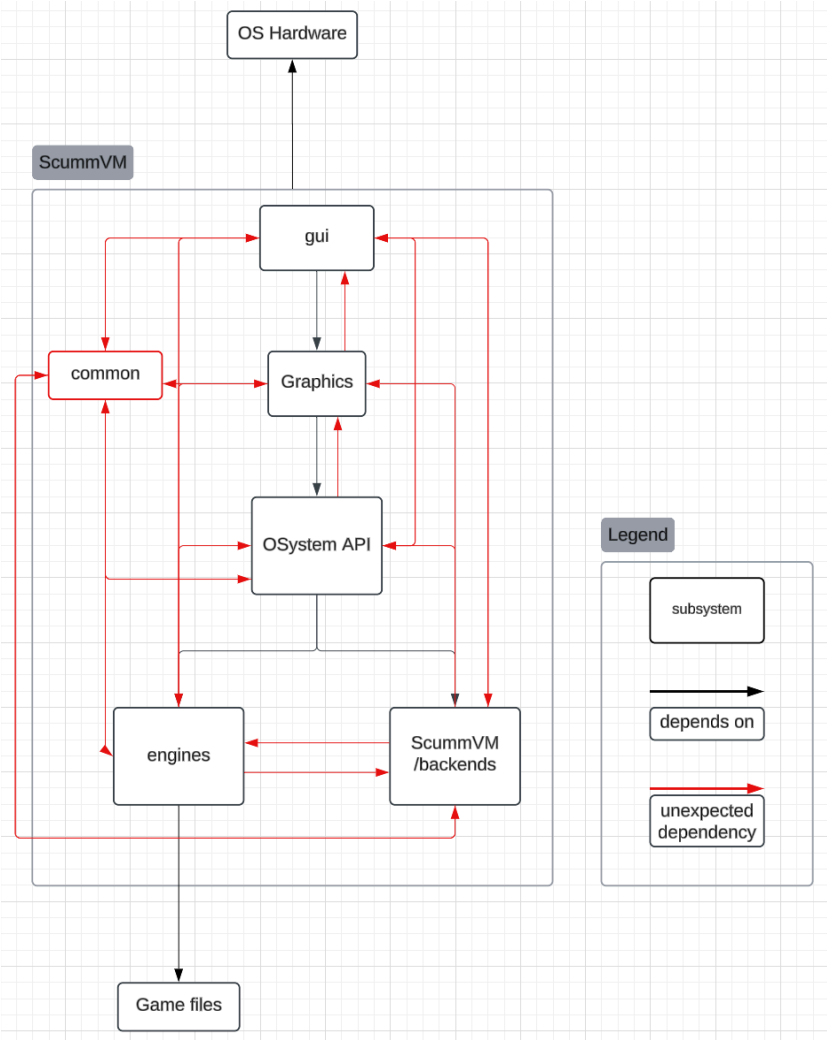


Fig 2. Shows concrete dependencies between source code in top-level subsystems of ScummVM

implements the OSystem API), image/, audio/, and video/. The module (and folder inside ScummVM) are called “common” because it includes classes, algorithms, and codecs that are used by all the subcomponents of ScummVM. The two components with the greatest



number of calls to common are the engine component (for the use of utility classes and codecs) and the GUI component (translation features, text-to-speech features, and built-in save game features in the launcher). The layout changes (ScummVM in the middle with hardware and game files on either side) was implemented to give more clarity to the boundaries of ScummVM. This includes all source code within the repository (Fig. 3), making it clear that game files (user-provided) and OS hardware are outside the scope of the concrete architecture of ScummVM itself. ScummVM is dependent upon these components, but since they are not inside of the dependency folder they should be seen as external components to the application.

Fig. 3. Reflexion Diagram based on the discrepancies between Fig. 1 and Fig. 2

Let's examine the divergences and provide some rationale:

(**Graphics**: contains cross-platform graphics rendering APIs such as OpenGL)
Graphics → GUI: Graphics makes a call to a file called `ThemeEngine.h` which defines the appearance of the GUI, its widgets, and visual elements. The graphics renderer is responsible for ensuring that the GUI is customizable and scalable based on the platform that Scumm is running on, thus this divergence is justified.
Graphics → Engines: Graphics makes a call to `engine.h` inside of the engines component, a header file which defines the engine class as well as some other related classes such as the "`PauseToken`" class. Multiple calls are going from the graphics component to the `PauseToken` class (inside of `engine.h`), indicating that the graphics component uses the classes provided in the engines folder to help perform application initialization, rendering, pauses and transitions, error handling, etc. Thus, the divergence is justified.

Graphics → Backends: Graphics has a two-way dependency with multiple files within the backends component (events, keymappers, platforms). Our graphics rendering component interacts with `sdl-events.h` (responsible for handling events on SDL platforms), `keymapper.h` (mapping input events such as key presses to game actions), and `sdl.h` (defines the OSystem_SDL class which is how Scumm is ported to SDl platforms) to help with handling user input, managing window resizing and mouse limitations (specifically for SDL platforms). Thus, the divergence is justified.

GUI → Engines: The GUI component interacts with a file called `game.h`, which provides classes such as DetectedGame, which are used by the GUI to display games that Scumm detects on the user's local system. In addition, the GUI component interacts with a file called `savestate.h` (GUI has options to save game) and `achievements.h` to display in-game achievements. Thus, the dependency between components is justified.
GUI → OSystem API: The GUI component interacts with `system.h`, which defines the OSystem API and its subsystem resources (pointers to Timer, Event, AudioCD, and saveFile managers). The rationale behind this dependency is that the GUI needs to interact with OSystem and its subcomponents to render the output of events, save files, synchronize movements, etc.
GUI → backends: the GUI component interacts with `storageFile.h` which essentially provides a way to interact with files in a cloud storage system. Since the GUI does provide the ability to store and pull game files to/from the cloud, this dependency is justified.

Engines → GUI: The engine components interact with many files inside the GUI component such as `EventRecorder.h`, `saveload.h`, `launcher.h`, `debugger.h`. The dependencies are justified by the engine's need to access the GUI for tasks such as logging errors and bugs, registering events, and managing saved game files.
Engines → Backends: Engine component dependent upon `savefile.cpp` which defines a class that allows the user to save game files on their local system (This is platform dependent and thus is in backends). It also is dependent upon the keymappings and events that are defined in `keymapper` to receive input (also platform dependent).
Engines → Graphics: The engine component relies on the graphics component to create, manage, and manipulate the game's graphical content, using classes that define surfaces, fonts, cursor, etc.
Engines → OSystem API: The engine relies on the `system.h` file to interact with the OS hardware and be ported into the platform that ScummVM is running on. Engines also interact with the base folder in the OSystem to access versioning and plugin info.

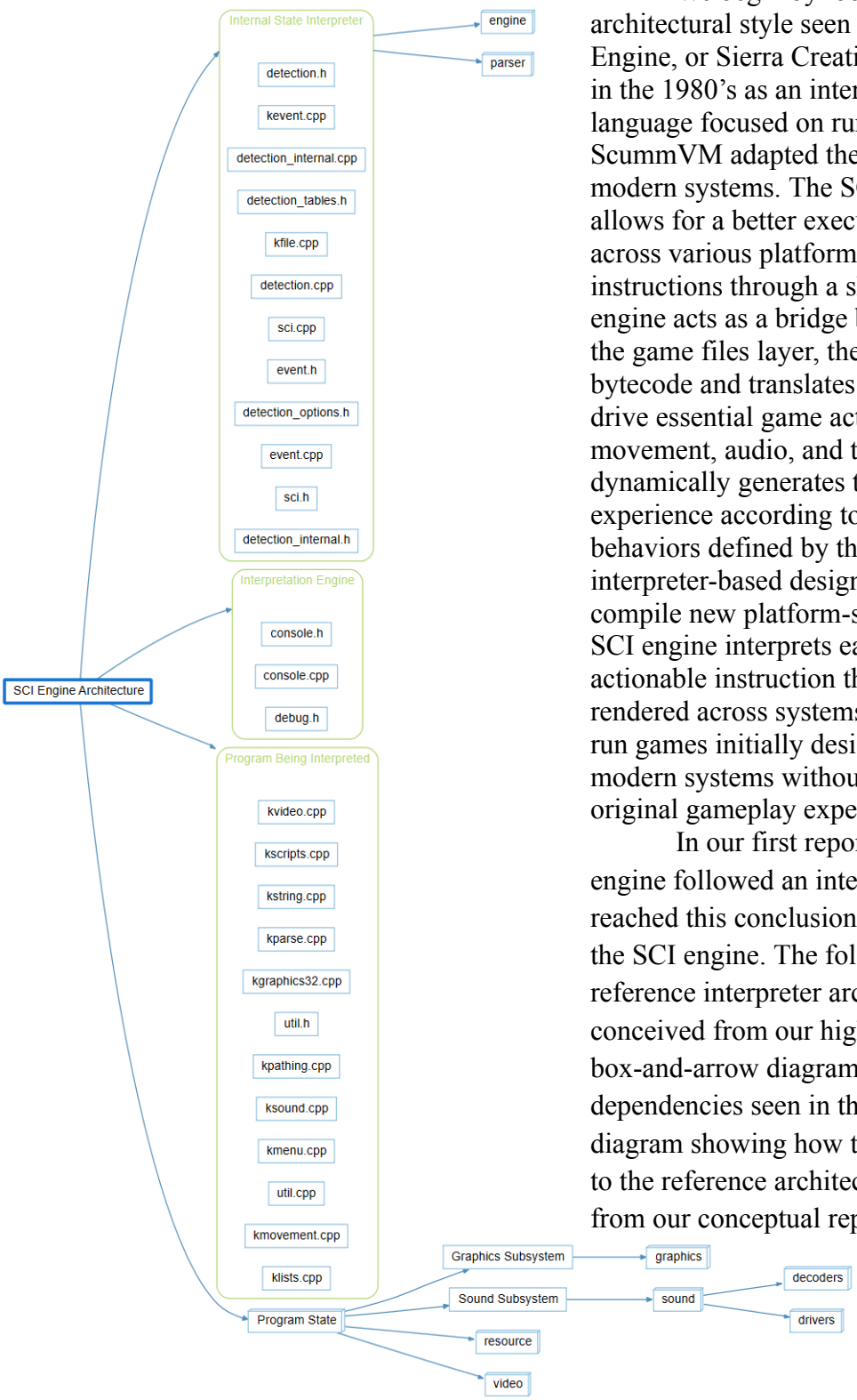
Backends → Engines: The backend component is reliant upon the engines for engine-related information (type of engine, version). It also needs access to classes defined in the engine component such as PauseToken to initialize pause functionality in the game.
Backends → OSystem API: The backend components need a connection to the command line arguments which are defined in the base/inside of the OSystem API component. Backends also interacts with `system.h`, which we believe to be logical

based on our understanding, because the OSystem API relies on the backends code to function.
Backends → Graphics: The backend components interact with the graphics components such as `surface.h` and `paletteman.h`. The `surface.h` file defines how to represent a 2D bitmap in memory. The backends need access to this definition to translate for the specific platform that ScummVM is being run on. `Paletteman.h` defines an abstract class for managing the palette of colours used for the game. The same logic follows that the backends need access to this code to configure the colour palette for the specific platform that ScummVM is running on. These are crucial steps in the configuration and initialization of the graphical presentation of ScummVM and its in-game appearance. Thus, the divergence is justified.
Backends → GUI: Backends need access to files within the GUI folder such as `widget.h`, `themeval.h`, `downloaddialog.h`, and `saveload-dialogue.h`. These files define classes for how information in the GUI should be displayed (widgets appearance, theme of layout, save dialog options). The logic here is similar to that of backends → graphics, in that the backends need access to these classes to determine how they should be rendered on the specific platform that is running ScummVM. Because these are crucial components of the UI, this dependency is also justified.

(**OSystem:** Only contains `system.h` which is the OSystem API and, defines available features a game can use)
OSystem API → Graphics: The OSystem component needs access to scalar functions, `renderer.h`, and cursormanager functions for rendering and scaling graphics on different platforms. Since this is a crucial component of the portability of ScummVM (an essential NFR), this dependency is justified.
OSystem API → GUI: The OSystem component needs access to `object.h` for class instantiations such as CommandReceiver, GuiObject, and widgets. It also needs access to gui-manager for the GUI event loop, the GUI layout and themes, and cursor management, etc. Once again the rationale behind this dependency is that OSystem needs to define how these classes will be used and shown within the specific platform that ScummVM is running on.

Notable Changes
We still wish to define the architectural style of ScummVM as layered. However, some notable changes can be seen in the sheer amount of interactions between components. The architecture is still layered, and communication is mostly seen between adjacent layers, but there are bilateral dependencies between every component, and therefore every layer. As seen in Fig. 3, we moved the OS hardware component and the game files component outside the boundaries of ScummVM. They should still be mentioned as crucial components of how ScummVM functions given a specific platform, and therefore should not be removed completely. The layered concrete architecture is better described as GUI → Graphics → OSystemAPI → Engine → Game Files, or as described in our conceptual architecture Presentation → Rendering → Abstraction → Engine → Game similar to as seen in Fig 1, with bilateral dependencies between every layer, and each layer also having a bilateral dependency with ‘common’ utility classes and codecs. The concrete architecture can be seen in Fig. 3.

Interpreter Architectural Style Seen in SCI Engine



We begin by looking at the interpreter architectural style seen in the SCI engine. The SCI Engine, or Sierra Creative Interpreter, was developed in the 1980's as an interpreter for a scripting language focused on running adventure games. ScummVM adapted the SCI engine to work with modern systems. The SCI engine within ScummVM allows for a better execution of the games logic across various platforms by interpreting bytecode instructions through a shared interpreter. The SCI engine acts as a bridge between the engine layer and the game files layer, the SCI engine processes SCI bytecode and translates it into system commands that drive essential game actions, such as character movement, audio, and text output. The engine dynamically generates the game's runtime experience according to a list of pre-set rules and behaviors defined by the original developers. This interpreter-based design eliminates the need to compile new platform-specific binaries because the SCI engine interprets each script command into an actionable instruction that can be universally rendered across systems. This makes it possible to run games initially designed for limited platforms on modern systems without impacting the games original gameplay experience.

In our first report, we stated that the SCI engine followed an interpreter architectural style. We reached this conclusion via a high-level analysis of the SCI engine. The following three diagrams show a reference interpreter architectural style, originally conceived from our high-level analysis, a box-and-arrow diagram showing the concrete dependencies seen in the SCI engine, and a reflection diagram showing how the concrete architecture maps to the reference architecture and how it deviates from our conceptual representation.

Game Files Layer

The game files layer consists of assets and resources that the game requires, including graphics, sounds, music, animations, and texts. All of these components are stored in data files which are referenced by the SCI scripts. When the engine layer calls a specific action, for example, playing a sound, the SCI engine accesses the game files layer to retrieve and render the necessary assets. The separation of data from logic allows for the assets to be reused or modified without having to alter the core game code. In ScummVM, the SCI engine handles data retrieval and integrates it with the engine layer, ensuring that every element (visual, audio, and text) matches the original game design. This structure also allows the engine to handle platform-specific functionality without compromising the quality of the assets.

Engine Layer

The engine layer is what makes each game unique and entertaining. Embedded within the SCI scripts, the engine layer defines the sequence of events, character behaviors, dialogue, and animations, pretty much everything that shapes the gameplay. By storing the game logic in these high-level scripts, the SCI engine can interpret the desired gameplay flow directly from the bytecode. This layer allows the game's fundamental mechanics to remain separate from any underlying platform, allowing the SCI engine to mimic these actions without modifying the logic itself. Due to ScummVM reading and executing these scripts within the SCI engine, games originally designed for older hardware can be played seamlessly on new systems, with the engine layer ensuring that the gameplay remains authentic.

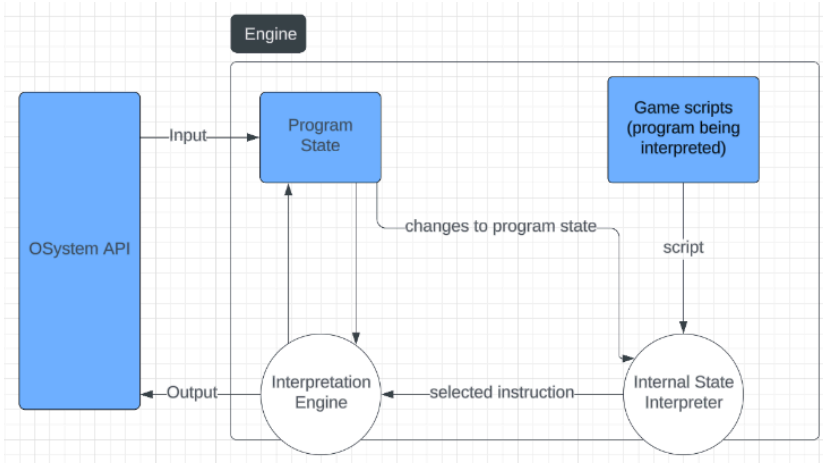


Figure 4, above is the reference diagram we had presented in our previous report for the SCI engine's interpreter architectural style.

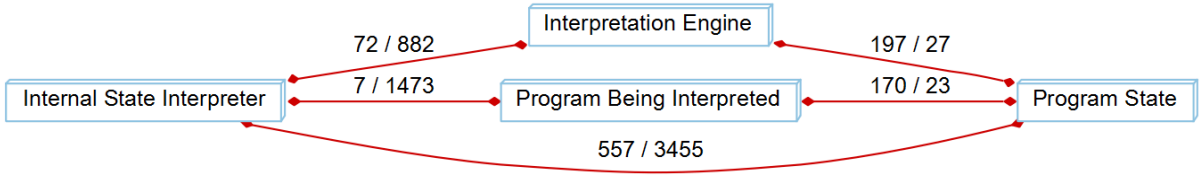


Figure 5, above is the concrete architecture for the SCI engine seen within ScummVM

In Figure 5, we see the complete set of concrete dependencies between modules for the SCI engine within the logic layer of ScummVM. As shown in the diagram, we can see that there are two-way dependencies, (known as bi-directional) between many of the different modules of the SCI engine. These dependencies are a direct result of function calls or data requests between these modules. They can be traced down to the individual lines of code seen within the files contained in each module. From the diagram, we can see that the strongest connection is between the Internal State Interpreter and the Program State, with 557 calls to files within the Program State from the Internal State Interpreter, and 3455 calls in the inverse direction. This relationship is in line with what we would expect of an interpreter architectural style. The Program State and Internal State Interpreter run in a feedback loop where the Internal State Interpreter interprets a given logic script and updates the program state accordingly.

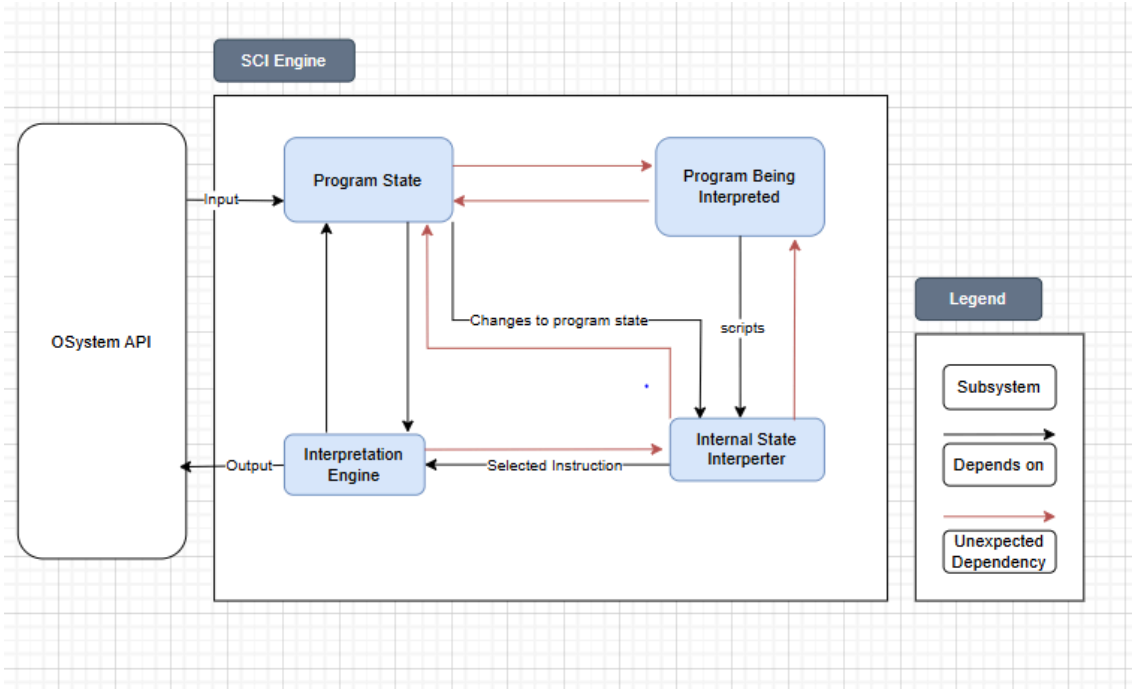


Figure 6, above is the reflexion analysis diagram for the SCI Engine

Interpretation Engine ↔ Program State: This dependency likely reflects frequent interactions as the game state is continually updated by interpreted instructions.
Internal State Interpreter ↔ Program Being Interpreted: Represents the close relationship between the script (commands being interpreted) and the interpreter itself.
Program Being Interpreted ↔ Program State: Indicates dependencies where the interpreted program directly affects or references the game state.
Internal State Interpreter ↔ Program State: This dependency reflects adjustments to the game state based on the internal state, which may not be mediated through the Interpretation Engine.

External Interfaces:

ScummVM external interfaces are mostly for the users to interact with. Its main components are the GUI, file system, and network. The GUI is where users do most of their interactions, it allows them to add/remove/modify games. When a user adds a game, they provide ScummVM with the path to the game files, they also need to provide the game's name and platform. This is done by files like `launcher.cpp` and `editgamedialog.cpp`, and they also allow ScummVM to organize the games. Users also use the GUI to set game options like sound volume, graphics quality, controls, and language. This is done in the `options.cpp` file. Another important part of the GUI is saving and loading games. Users can save their progress through the GUI, which uses files like `saveload-dialog.cpp` to save the game. The GUI supports cloud storage which allows users who want to play their saved games on multiple devices, the saved games can be saved to places like Google Drive or Dropbox. This is done in the `cloudconnectionwizard.cpp`, file.

Once users add a game through the GUI, ScummVM reads the selected game files, the game files are accessed by the certain engine needed for that game, the engine takes care of the specific game mechanics and visuals based on the original game. Also, any settings users adjust in the GUI, such as audio levels or controls/key binding, that is saved to the `scummvm.ini` file. This allows users to set up the game how they like. ScummVM also keeps the changed data from the original game.

The network features allow users to modify the game to their liking. For example, they can download extra content like DLCs, themes, or language packs, they can also check for software updates on those extra things. Cloud synchronization is also important in the network features because it allows users to sync their saved games online and continue playing across different devices. This is also connected to the saving files to Dropbox.

Because ScummVM is for video games, it accesses your computer's hardware like I/O devices. The GUI manages these inputs so that when users click on icons or press keys, ScummVM knows what was clicked and how to respond. The file `gui-manager.cpp` handles these interactions. Display settings, such as screen size or theme, can also be adjusted through the GUI.

Another important and cool thing that the ScummVM gui does is that it displays error messages and provides a debugging tool, so if a user has an issue uploading/running a game they know what's wrong. The debug tool in ScummVM allows users to input commands to help them understand the error. This is done by `console.cpp` and `debugger.cpp`. And the error messages are displayed with `error.cpp` and `message.cpp`.

Example Use Case 1 (Reworked): User loads and starts a game using ScummVM. This is a use case for the layered architecture, demonstrating how the various layers work together and interact. Thus allowing the game to run smoothly and the user to be satisfied.
Sequence of Steps:

1. [GUI → Engine] The user launches Scummvm and selects a previously purchased game from their files via the GUI. The GUI uses a file called classes in the file `game.h`, to display which game files the user has on its computer.
2. [Graphics → GUI] The Graphics layer makes a call to the `ThemeEngine.h` file in the GUI layer. This defines the appearance of the GUI on a specific platform. Allowing the game list to be displayed and allowing the user to pick their game of choice.
3. [Graphics → Backends] The Graphics layer then makes a call to the Backend Layer to register the user's key press. This is done through a call to the `keymapper.h` and the `sdl.h` file.
4. [OSystem API → Graphics] Once a game is selected, OSystemAPI is responsible for generating platform-specific commands to aid the game engine in execution. This is done by a call from the OSystem API layer to the `renderer.h`, and `cursormanager` functions found in the Graphics layer.
5. [Engine → Graphics] The Engine then utilizes functions in the Graphics layer to initialize the game's graphical components by using classes that define surfaces, fonts, cursor, etc.
6. [Backend] The backend layer then loads game files from the users machine by making use of the `savefile.cpp` class, which contains information about the save files on the user's machine.
7. [Backend → GUI] The backend layer also makes a couple of calls to the GUI layer to display loading information
8. [Graphics → GUI] Finally the Graphics and GUI layers work together to render the game specific elements through functions like `ThemeEngine.h`! The game is now ready to be played

Example Use case 2 (Reworked) : A User saves their game progress. This use case demonstrates how the simple action of saving a game will be executed through the different layers present in the ScummVm architecture.
Sequence of Steps:

1. [Engine → GUI] The Engine layer makes a call to the `EventRecorder.h` in the GUI layer recording the event of the user opening the in-game menu. It also ensures the GUI reflects this change.
2. [Backend → Engine] The backend then makes a call to the PauseToken class defined in the Engine layer to initialize the pause function.
3. [Graphics → GUI] The Graphics layer then makes a call to the `ThemeEngine.h` file, defining the GUI. It displays the menu where a save button can be found.
4. [Graphics → Backends] The Graphics layer then makes a call to the Backend Layer to register the user's key press. This is done through a call to the `keymapper.h` and the `sdl.h` file.
5. [OSystem API] The OSystem API then initiates the saving operation through one of its subsystems, the saveFile managers.
6. [Engine → Backend] The engine then makes a call to `savefile.cpp`, which defines a class which permits the user to save their game files on their local system.

7. [Backend] The Backend then uses the `storageFIle.h` file to manage the saved files in a cloud storage system.
8. [Backend → GUI] The backend then utilizes the `saveload-dialogue.h` file found in the GUI Layer to render the saving and completion messages.
9. [Graphics → GUI] Finally the Graphics and GUI layers work together to render the success message through functions like `ThemeEngine.h`! The game is now saved

Data Dictionary:

- **Conceptual Architecture**: The conceptual architecture represents how developers think of a system. Developers define meaningful relations as well as the key components the system should have here. It can be compared to the blueprints of a house.
- **Concrete Architecture**: The concrete architecture represents the relations and the key components that appear in the system. It is devised by looking through the source code of a project to see how the different components interact. It can be compared to the structure of a built house.
- **Reflexion Analysis**: A way of comparing the conceptual and the concrete architecture to identify if they match. In a reflection analysis, we examine the dependencies we expected to appear and the ones that do appear to explain any differences.
- **Layered Architecture**: The layered architecture style is best for applications that can be divided into an organized hierarchy of responsibilities or functions. Each layer has one specific job and only allows neighbouring layers to communicate, permitting abstractions when necessary.
- **Interpreter Architecture**: The interpreter architecture style is best for applications where the best language or machine to use is not available for the current system.
- **OSystem API**: Defines what a game can use (whether a user can draw on a screen, use mouse clicks, etc.). It also shields the SCI engine from the operating system itself.
- **SCI Engine**: A game engine responsible for interpreting game files, processing game logic, tracking game state, and handling events (e.g., button presses, mouse clicks).
- **Bytecode:** Computer object code that an interpreter converts into binary so it can be read by a computer's hardware processor.

Naming Conventions:

- **GUI:** Graphical User Interface
- **SCI:** "Sierra Creative Interpreter" or "Script Code Interpreter"
- **OS:** Operating System
- **API:** Application Programming Interface
- **SDL:** Simple DirectMedia Layer
- **DLC:** Downloadable content.
- **I/O:** Input/Output

Conclusions:

In conclusion, we used the Understand software to explore the dependencies within both the ScummVm software and the SCI engine, allowing us to compare what we thought the conceptual architecture would look like to the reality of the concrete architecture. Solidifying our understanding of both of their architectural styles, with ScummVm following a layered architecture and the SCI engine an interpreter architecture style. Furthermore, we explored how the external interfaces interacted with the different layers of the ScummVM architecture allowing users to interact with the system. Finally, we revisited the use cases explored previously providing a more well-rounded understanding of how they truly function. In future assignments, we will propose a new feature for ScummVm, while discussing the changes that would be required in the conceptual/concrete architecture to ensure they remain accurate.

Lessons Learned:

Through our time spent investigating the concrete architecture of ScummVM, the most notable lessons we learned were about the dependencies when compared to what we had predicted with our conceptual architecture. There were a great number more dependencies than we had initially envisioned. In addition, these dependencies were primarily bi-directional. This overall means that the individual components and layers were communicating more than the 'ideal' layered architecture style. It is our belief that this can primarily be attributed to the fact that this is an open source project with very minimal documentation. Clearer documentation and/or naming conventions would be necessary for developers newer to the project to rapidly understand the architecture and where the best places for their files is. This would additionally help to reduce some potentially redundant function calls. This is further compounded by the fact that after verifying the discrepancies we found, they were all innately logical and justified in their contexts; thus meaning they were not harming the functioning of the project in any way, nor were they implemented by error.

References:

SCI Programming Community. SciProgramming, https://sciprogramming.com/. Accessed 13 Nov. 2024.

SCI Companion Documentation. SCI Companion, https://scicompanion.com/Documentation/. Accessed 13 Nov. 2024.

Sierra Creative Interpreter - Sierra Help Wiki. Sierra Help, http://sciwiki.sierrahelp.com/index.php/Sierra_Creative_Interpreter. Accessed 13 Nov. 2024.

Sierra Creative Interpreter Engines. Sierra Chest, https://sierrachest.com/index.php?a=engines&id=2. Accessed 13 Nov. 2024.

Developer Central - ScummVM Wiki. ScummVM, https://wiki.scummvm.org/index.php/Developer_Central. Accessed 13 Nov. 2024.

What is DLC in Gaming? HP Tech Takes, https://www.hp.com/us-en/shop/tech-takes/what-is-dlc-in-gaming. Accessed 13 Nov. 2024.

Content Item 5711391. Queen's University OnQ, https://onq.queensu.ca/d2l/le/content/959322/viewContent/5711391/View. Accessed 13 Nov. 2024.

Appendix A - Component Architecture