Concrete Architectures of ScummVM and the SCI Game Engine

Authors:
Frankie Cao (frankie.c@queensu.ca)
Gavin Lacy (gavin.lacy@queensu.ca)
Aidan Tulk (21art8@queensu.ca)
Michal Wrobel (24lt12@queensu.ca)
Ayman Zaher (20az27@queensu.ca)
Selena Zou (20sz88@queensu.ca)

18 November 2024

Table of Contents

Appendix

1. Abstract

Briefly, ScummVM is an open-source, cross-platform software system that provides users with the ability to play select classic point-and-click adventure or role-playing video games. By creating a complete rewrite of each game's executable, ScummVM is able to run supported games on a wide variety of platforms, including (and especially) platforms for which the games were never designed. Flagship supported games include the *Monkey Island* series and *Blade Runner*.

In our A1 report, "Conceptual Architectures of ScummVM and the SCI Game Engine," we examined the official documentation of ScummVM as a top-level system and the SCI game engine in particular. In that investigation, we proposed that ScummVM uses a layered architectural style to enforce a separation of concerns between frontend, common code, and backend components. We proposed that the SCI engine employs an interpreter architecture to achieve easy portability across operating systems.

In this report, we seek to establish a concrete architecture for the overall ScummVM system, the top-level Engines subsystem, and the SCI game engine by examining the actual project source code. As a result of our investigation, we propose modifications to our original layered architecture model for ScummVM. We propose a layered architecture for the Engines subsystem. Although we do not propose changes to our chosen interpreter-style architecture for the SCI engine, we examine it in more depth.

2. Introduction and Overview

Architectural drift refers to a phenomenon in software engineering wherein as software systems grow in size and complexity, the documented conceptual architecture and concrete implementation drift further apart [1]. The result is that the initially-proposed conceptual architecture of a system may not accurately describe how the components in this system are laid out and interact with each other (for example, proposed subsystems may be merged, or code for one feature may be implemented instead in another software area). These discrepancies may make enhancements, bug fixes, refactoring, or other maintenance more difficult.

As a long-lived open-source software system, ScummVM is naturally subject to some architectural drift between its original conception and actual implementation. This report therefore seeks to identify, investigate, and reconcile these differences. We do so by mapping source code files to the architectural components elaborated in our conceptual architecture, then examining the dependencies between each. Our primary sources of such information are the Understand 6.5 tool, which is used to investigate dependencies, as well as the ScummVM GitHub repository, which hosts the raw source code [2].

Although our overall understanding of ScummVM as a layered system does not change, unexpected dependencies in the system indicate rearrangements of some components between layers. With respect to the Engines subsystem, we observe a clear division between core functionality required by all engines, and the unique engines themselves; thus, we propose a two-layered architecture, similar to a client-server model. Based on convergent dependencies in the SCI game engine, we maintain our conclusion that it employs an interpreter architecture.

Finally, with our strengthened understanding of ScummVM and its subsystems, as well as its convergent and divergent dependencies, we discuss the system's external interfaces and two representative use cases in greater depth.

3. Architecture

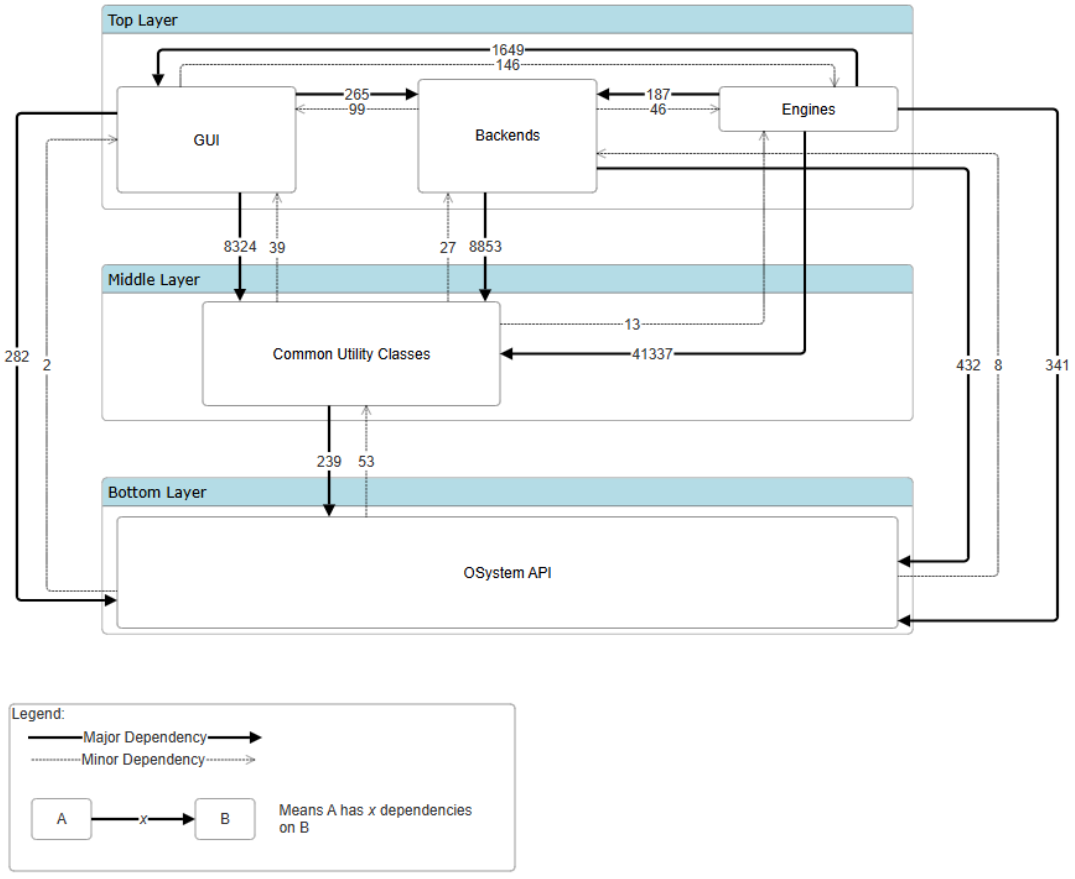
3.1 Overview

In our reflexion analysis [3] of the ScummVM system, we consider first the ScummVM platform itself as a system (section 3.2.1), then the Engines subsystem (section 3.2.2), which contains the code for each individual game engine supported. We also examine in particular the SCI game engine as an example, and compare its concrete architecture with the proposed conceptual architecture in our A1 report (section 3.2.3). At the end of this section, we briefly discuss alternate candidate architectures and our reasons for rejecting them.

3.2 Concrete Architecture Styles

3.2.1 ScummVM

In our A1 report, we proposed a layered conceptual architecture, largely based on documentation and knowledge of the reference architectures of modern game engines. This conceptual architecture had the GUI and Engines in the top layer, the common classes in the middle, and the backends and OSystem API on the bottom. Using the Understand tool, we were able to map each file in ScummVM's source code repository to its appropriate component. The dependencies between each component are shown below.



Generally, large and long-lived systems like ScummVM are prone to architectural drift as an artifact of changing requirements, misunderstandings, hack fixes, etc. Thus, we are not surprised to see the large number of unexpected dependencies evident above. However, two numerical observations help guide us in updating our conceptual architecture:

1. Each two-way dependency shown above heavily favours one direction. That is, for any two components A and B with mutual dependencies, A has many more dependencies (on the order of hundreds or thousands) than B does on A. This is most evident in the relationship between the common utility classes and engines components: the common classes depend on the engines a mere 13 times; by contrast, the engines call the common classes over 41 000 times. These types of lopsided mutual dependencies, which can be approximated to one-way dependencies with a few outliers, indicate that ScummVM still fundamentally uses a layered architectural style, with higher-level components acting as clients (i.e., *callers*) to the lower-level components (i.e., *callees*) which serve them. In the example mentioned above, the engines component is a client to the common classes which serve it.
2. Out of 62 342 total dependencies between all components, 58 514 of them (nearly 95%) are dependencies from the GUI, engines, and backends on the common classes. The common classes in turn predominantly depend on the OSystem API, which is largely self-sufficient. This pattern suggests a reshuffling of our original layered model to the one shown above, where the top layer is made up of the GUI, engines, and backends; the middle layer contains the common utility classes; and the lowest layer contains the OSystem API.

Bearing these observations in mind, we therefore propose a modified layered architecture from our A1 model:

1. Top layer: the GUI, engines, and backends components. Since these components are in the same high-level layer, they can easily communicate between each other. This permits efficient two-way communication between the user-facing components (GUI and engines) to minimize latency. The presence of the engines component in this layer also allows for easy conversion of any necessary structures to be portable between platforms.
2. Middle layer: the common utility classes. This component acts as a middleman, providing a common language for all top-level components to communicate with the OSystem API.
3. Bottom layer: the OSystem API. The view of this layer is unchanged; the OSystem API defines the supported features available to any game engine.

The largest modification to our model is the moving of the backends component from the lowest layer to the highest. We had initially thought that because it was one of the lower-level components and documented to directly implement the OSystem API, it should be grouped together with the OSystem API in the lowest layer. However, further examination of the actual code dependencies reveals that the backends component has significant dependencies on the data structures and services provided by the common utility classes.

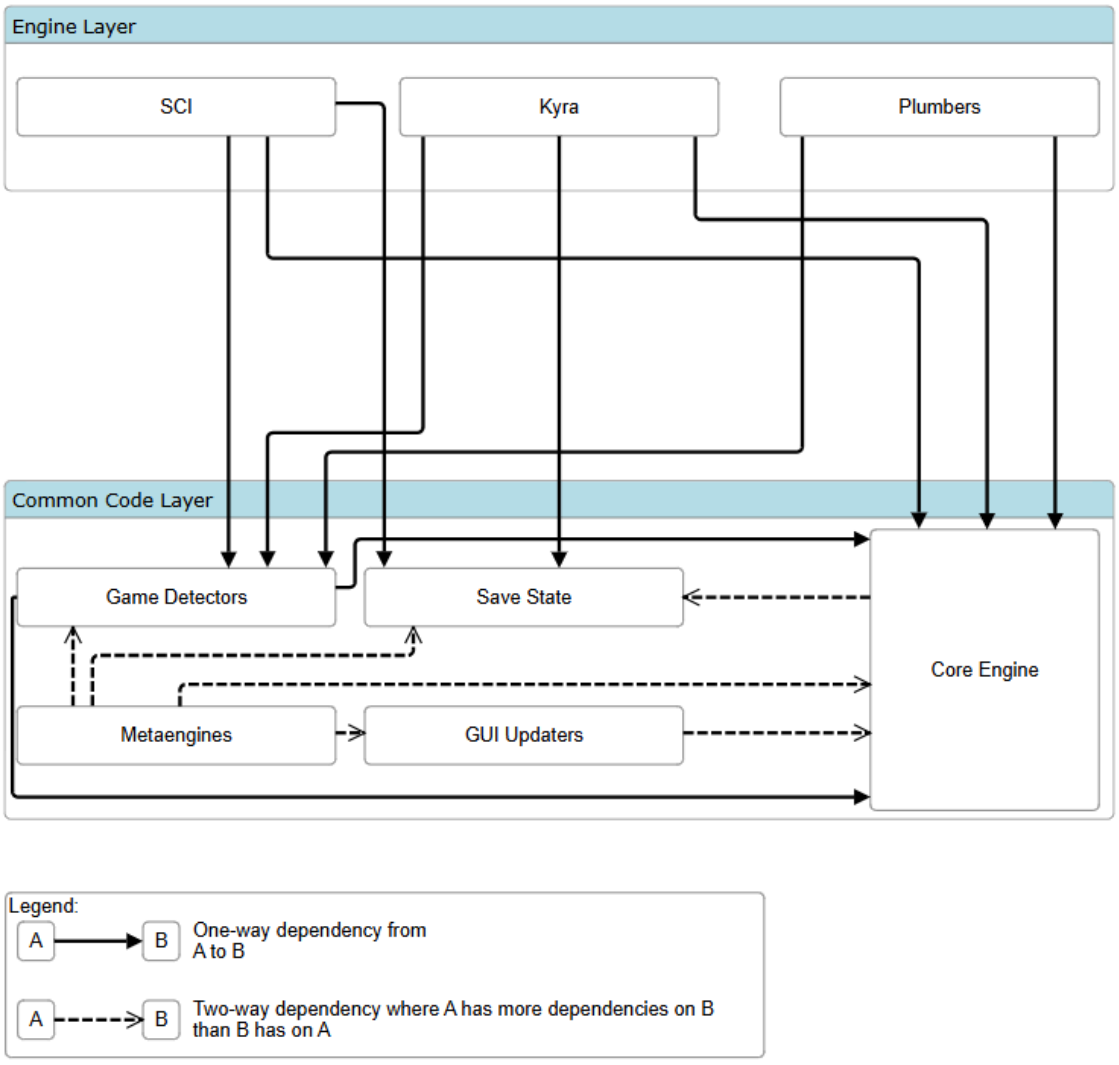
To understand this seemingly backwards dependency, consider an example. One piece of functionality the backends may implement on behalf of the OSystem API is accepting keyboard input (e.g., the user searches a game in the GUI). To hold this keyboard input, together with necessary metadata about the user's input, it needs a common structure — a programmer-defined buffer — that can be shared between the GUI (the source of the input data) and the OSystem API, which eventually manages the storage of this data. Common structures such as buffers are implemented by the common utility classes, hence why this component is depended on by the backends.

We observe that there are still significant unexpected dependencies: in particular, there exist many "skip-level" dependencies; i.e., dependencies from the top layer directly to the bottom layer, skipping the common utility classes in the middle layer altogether. The reason for this is relatively simple. Although a layered architecture attempts to promote a separation of concerns, where ideally each layer should be adequately served by the APIs provided by the layer beneath it, it is often necessary, for time and memory efficiency reasons, to avoid nested function calls and go directly to the source. For example, the GUI directly accesses the KeyMapArray structure defined by the OSystem API in order to minimize latency when providing the keymapping feature to the user. In general, as ScummVM is an interactive gameplay system, it operates under real-time system constraints, especially with respect to user-perceived latency. Thus, certain shortcuts, such as allowing components to call other components from non-adjoining layers, are occasionally taken in order to guarantee speed.

3.2.2 The Engines Subsystem

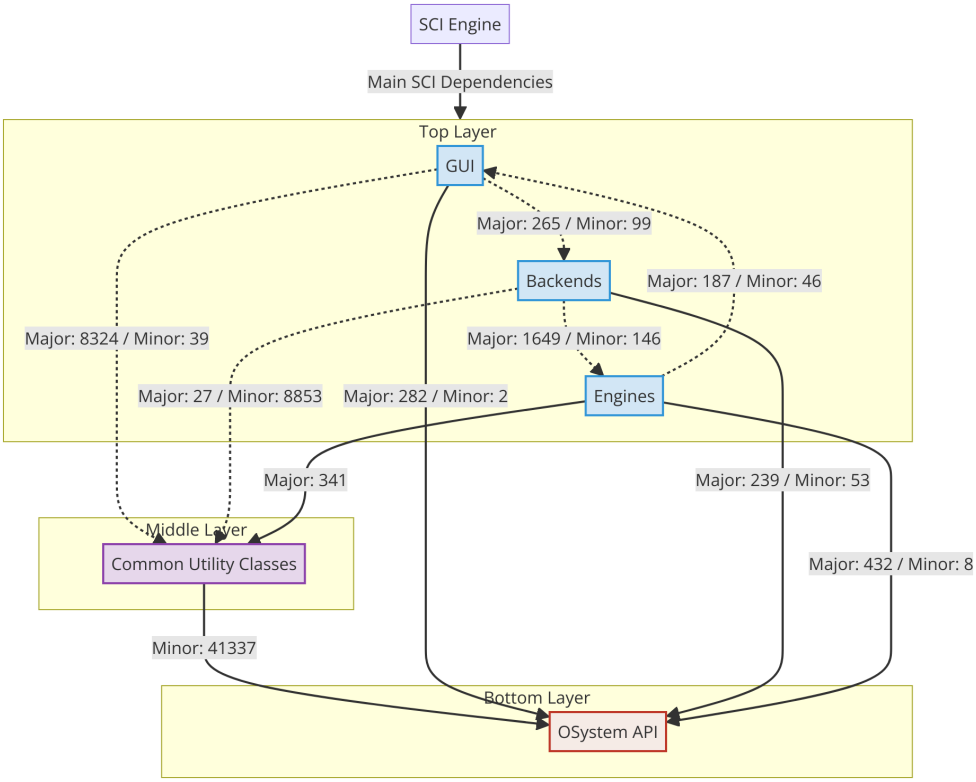
The Engines component of ScummVM comprises two main subcomponents. The first is the common code shared by all game engines (i.e., the core engine, game and engine metadata, common functionality such as save-state and GUI dialogs, etc.). This common code is contained in a set of source code and header files inside the "engines/" directory. The second subcomponent comprises the code for all game engines (e.g., SCI, Kyra, Plumbers, etc.) supported by the ScummVM platform, organized into a separate directory for each individual engine [4].

Using the Understand tool, we are able to observe that each of the game engines calls the common engine code for core functionality like loading/saving the current game state (loadGameState()/saveGameState() respectively), running the core engine (unsurprisingly via the function call to Engine()), etc. This implies a client-server or layered relationship between the individual engines and the common code subcomponents: the engines (client) depend on core services provided by the common code (server). We therefore propose a two-layered sub-architecture for the top-level Engines subsystem, with the top layer comprising the individual game engines, and the bottom layer which serves it being comprised of the core common code (diagram below).



3.2.3 The SCI Game Engine

As mentioned in our A1 report, the Sierra Creative Interpreter (SCI) game engine is one of the game engines currently implemented and supported by ScummVM and allows ScummVM to play many of the games originally created for the SCI engine. Using the Understand tool, we were able to determine how the ScummVM team re-implemented the SCI engine for use in ScummVM, and how it integrates with the rest of the ScummVM codebase. As seen in the figure below, the ScummVM SCI engine implementation takes advantage of ScummVM's layered architecture and the abstractions to provide identical functionality to the original engine while maintaining the same platform portability as ScummVM.



After determining the main components in ScummVM utilized by the SCI implementation, we analyzed the dependencies to determine the role each component played in the implementation of ScummVM's SCI engine. As expected from the knowledge we gained from section 3.2.2, we found the SCI engine utilizes the Common Engine Code layer to easily integrate into ScummVM and take advantage of all the abstractions it offers. However, we found the SCI engine also uses a few components outside of the Engines component as well to implement its functionality, the GUI, Backends, OSystem API and most notably the Common Utility Classes layers. The SCI engine uses the Common Utility Classes extensively to perform platform dependant operations through ScummVM's abstractions and OSystem API to ensure support on all platforms ScummVM supports. Notably the SCI engine implementation uses the common utility classes to work with the mouse and cursor, modify graphics and text rendered and perform graphical transformations. The common utility classes are also used for a variety of other tasks such as working with audio, performing complex math operations and decoding of images and videos. The SCI engine does also interact with the OSystem API directly to perform some operations such as loading resources from the file system, and interacting with any SCI engine plugins the user may have added to ScummVM. The backends component is not used as extensively compared to the OSystem API but the SCI engine takes advantage of it to implement keyboard binding and handling game saves. The GUI component is the last component we found the SCI implementation depended on, and provides the engine with a robust mechanism of creating and displaying graphical interfaces such as menus and dialogs.

Furthermore, utilizing the Understand tool we were able to create a rough concrete architecture of the SCI engine implementation inside of ScummVM itself. We found that

it is similar to the architecture we proposed in our A1 report but has an additional dependency from the Sierra PMachine to the SCI Interpreter. In the ScummVM implementation of the SCI engine the entire Sierra PMachine has been reimplemented using the abstractions provided to allow it to run on any platform ScummVM does. On top of that the ScummVM team reimplemented the SCI interpreter to use the reimplemented PMachine and abstractions in ScummVM. Just like in our A1 report, the Sierra PMachine is still responsible for providing a virtual-machine for the SCI interpreter to run on top of and the interpreter is responsible for loading and executing games. The unexpected dependency comes from some function calls from the PMachine implementation to the Interpreter implementation to allow the PMachine to modify some of the state related to the Interpreter.

3.3 Alternatives Considered

3.3.1 ScummVM

As a "video game engine recreator" [5], ScummVM naturally suggests an event-driven structure: all the essential use cases of adding, playing, or saving a game are accomplished by listening for user input, such as keystrokes or mouse clicks, to trigger a cascade of actions (for example, the user clicking "Start" in the GUI should cause the game-launching routine to run). Based on this assumption, we briefly considered a publish-subscribe ("pub-sub") architectural style. However, while pub-sub elements are certainly present — one only needs to realize the point-and-click nature of the games forces the system to "subscribe" to user-triggered events and act accordingly — ScummVM does not generally rely on an event-broadcast system. Communication between different components is typically direct (not decoupled, as it would be in a pub-sub system), and events such as user input are handled in a procedural manner via direct method call rather than asynchronously via a central event broadcasting queue.

With the large number of unexpected dependencies between almost every pair of components, we also considered proposing a modular architecture style. The main difference between a layered and modular architecture is that while layering components institutes a hierarchy and promotes separation of concerns (e.g., the GUI and engines components in the top layer may not concern themselves with the hardware-specific implementations carried out by the OSystem API in the bottom layer), a modular architecture breaks down a system into more or less independent modules, each of which implements a specific piece of functionality. In a modular architecture, components (i.e., modules) are free to communicate between each other, while a layered architecture should *in theory* only allow communication between components in adjoining layers [6]. However, further investigation of ScummVM's dependency graph showed a strong directional flow from the top-level components (GUI, engines, backends) through the common utility classes towards the low-level OSystem API. Comparatively, dependencies in the reverse direction (bottom-to-top) or skip-level dependencies (e.g., from the GUI directly to the OSystem API, skipping the common utility classes) are much fewer, and are generally permitted for efficiency reasons.

3.3.2 The Engines Subsystem

As mentioned previously, the Engines subsystem exhibits some characteristics of a client-server architecture. Most significantly, it has a common code (server) component providing core services to the individual engines (clients). However, a few traits of client-server architectures are missing. First, the clients in client-server systems are typically involved in presenting an interface to the user. In ScummVM, all user presentation is accomplished by the GUI and audio/graphics components in the common utility classes. The engines, as a general rule, do not play a role in interfacing with the user. Second, client-server architectures are usually used in distributed systems, with a central server residing in one location and providing services to distributed clients (potentially over a large range). The individual game engines in ScummVM and their common code are all local to the user's machine, not distributed over a network. Based on these factors, we reject a client-server architectural model for the Engines subsystem, and choose a layered approach instead.

4. External Interfaces

4.1 Game Files

The concrete architecture shows that the files of the games are not included within the architecture of ScummVM nor any of the engines included within. This is in line with the conceptual architecture which showed that the files for the game had to be legally acquired in order to be allowed to use it. The game files can be acquired in multiple ways which are then acted on by ScummVM and the engine that is being used. The concrete architecture reveals that the games are detected within the "base" subsystem which uses a command line file to detect the presence of the games. These game files are then sent to the engines subsystem to be acted on by the selected engine.

4.2 Cloud Services

The cloud services are found in the backend subsystem. These cloud services can be used to transport files across systems and allow users to store save data among multiple devices. Inside the cloud directory, there are other directories to specify the type of cloud server that is being used like onedrive, google drive, and dropbox. There are also storage files located within the cloud directory which are able to talk with the cloud to save the storage within a configuration file. Most of the dependencies are linked with the "common" subsystem which handles many of the shared utilities across ScummVM.

4.3 Local Web Server

Also located within the backend subsystem are the local web server files which use a client-server style architecture to create a web server that can hold game files. This is very similar to the cloud services and is used when the cloud is not the preferred choice for any number of reasons. Most of the dependencies are once again linked to the common subsystem, with a few linked to the base system.

5. Use Cases

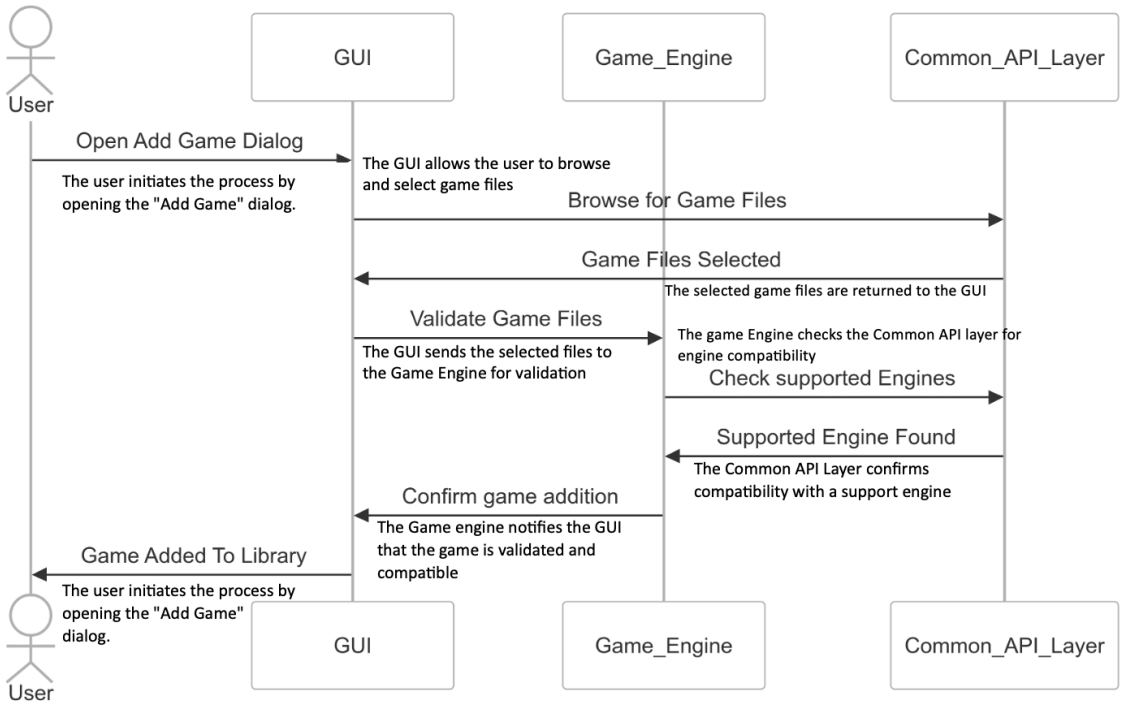
5.1 Adding a Game

As would be expected, the initial process of instantiating a game by selecting its specific game files are relatively the same without any real unexpected dependencies. Despite the process

being essentially the same, its modality differs; one such absence is the exclusion of a FileSystem component. Instead, reading and writing of files is done by ScummVM's own dedicated File class in the common component.

Looking into the specific file located at "scummvm\common\file.h" we can see that there are sets of methods for the classes "File" and "DumpFile" to read and write the binary data from files. This data is then held in ScummVM's own "File" class where it can be read from using said methods. Afterwards, the compatibility of the game with the current engines is verified. In the concrete architecture there are a few unexpected dependencies with some of the engines inside the "Game_Engine" component of the diagram but essentially, the process is not changed.

The usage of the File class in the common layer to handle file operations shows that contrary to what was said in the first assignment, the methodology of retrieving and reading files is not machine dependent. Instead, it is kept the same by having the proprietary file-handling files in the common layer do all the work.

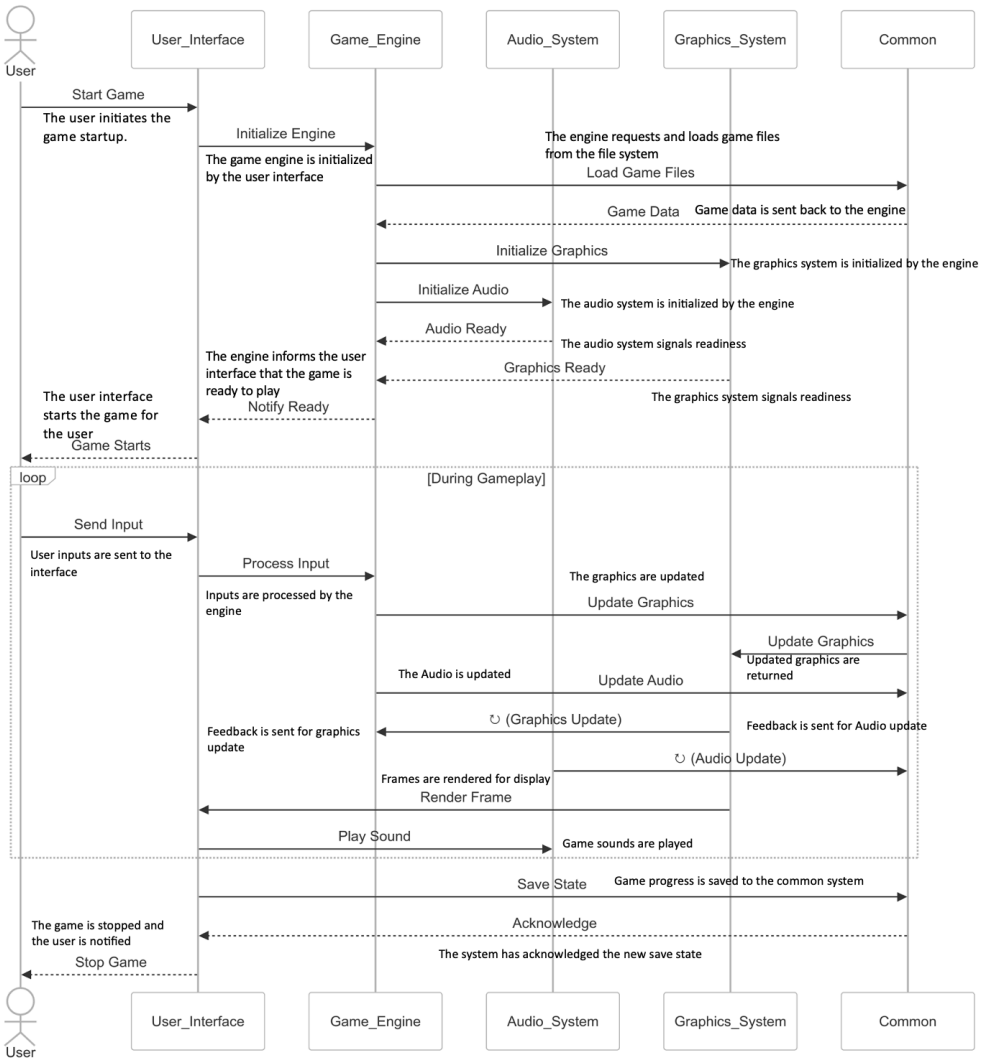


5.2 Playing a game

On the contrary, the dependency structure of playing a game changes significantly. Within the SCI component there are concurrent calls to the audio, graphics, and GUI components to get the user's input and update the game state. All of this follows a sequential order similar to that of what was discussed in the conceptual architecture derived explained in A1 where the data flows from the GUI to engines in cycles. However, inside the concrete architecture of ScummVM, there are deviations from the conceptual architecture such as the unexpected dependency where audio and graphics depend on the common component.

By doing further analysis of some of the divergences in the concrete architecture we can see that the inclusion of the “scummvm\common\forbidden.h” file causes the unexpected dependencies between audio and graphics. Looking into this file’s documentation we can see it says “This header file is meant to help ensure that engines and infrastructure code do not make use of certain "forbidden" APIs, such as fopen(), setjmp(), etc”.

We can see through the code and documentation that this file intercepts common functions such as fopen() and replaces them with a garbage string causing a compilation error. This is done to prevent the platform-specific behavior of these functions because ScummVM has its own way to handle file I/O and system interactions. In short, this is done to ensure consistency between different systems. For example, this ensures that instead of using fopen() to open files, developers use ScummVM’s File class at “scummvm\common\file.h” to guarantee file handling is consistent across platforms. As mentioned in the 3.2, there are many more minor divergences and dependencies that are not mentioned. But this divergence, highlights how developers create work-arounds in order to meet functional requirements such as consistency which is why it was emphasized in this use-case.



6. Conclusions

From the dependency graph of ScummVM's top-level components, we conclude that the system overall adopts a layered concrete architecture, with the layers divided into top (highest-level), middle, and bottom (low-level) components. This choice of architecture coincides with our initial conceptual architecture; however, some modifications are made. Most importantly, the backends component moves from the bottom layer to the top, where it and the GUI and engines all utilize the common utility classes in the middle layer to communicate with the OSystem API at the bottom. We additionally conclude that the Engines subsystem uses a two-layered architecture, with the lower layer containing the common code for all core engine services, and the individual game engines residing in the upper layer. The SCI game engine does not demonstrate significant architectural drift between its conceptual and concrete architectures. As a consequence of its emphasis on portability, it establishes an interpreter concrete architecture, complete with p-machine and interpreter components.

Further directions for research include other subsystems of ScummVM, as well as other game engines supported by ScummVM (e.g., Kyra, Plumbers, etc.). As our research is limited by time constraints and the large size of the ScummVM project, we have not investigated in depth all unexpected dependencies; in particular, bidirectional dependencies between the GUI, engines, and backends components in the top layer of our proposed architecture remain inadequately discussed. In general, avenues for future research lie largely in closer examination of the codebase.

Appendix

7. Data Dictionary

Term	**Definition**
Architectural drift	A phenomenon in software engineering where as systems grow in size, complexity, and age, their conceptual and concrete architectures drift apart.
Reflexion analysis	An approach whereby software engineers investigate project source code and dependencies to determine where the concrete architecture differs from the conceptual model and investigate why.
Game engine	A system that provides libraries and APIs to be used by developers to ease the creation of computer-based games.
P-Machine	A virtual machine designed to execute portable code regardless of the host platform.
Interpreter	A software system that directly executes instructions that have not been compiled for the platform.

8. Naming Conventions

Abbreviated Term	**Full Form**
SCI	Sierra Creative Interpreter
GUI	Graphical User Interface
API	Application Programming Interface

9. Lessons Learned

As with any project in software engineering, the depth of investigation is limited by the time constraints. With respect to this report, we suggest that narrowing our scope (with regard to the subsystem selected for deeper analysis) earlier in our process would have allowed us to better direct our investigation. We would then have been able to look deeper into the origins of the many unexpected dependencies and made more efficient use of the “Sticky Notes” method. Generally speaking, more experience with tools such as Understand would also have helped us to better leverage their capabilities and get our investigation off the ground.

References

[1] Bedir Tekinerdogan. “Architectural drift analysis using architecture reflexion viewpoint and design structure reflexion matrices.” *Software Quality Assurance,* p. 221, 2016. Accessed 18 Nov. 2024. Available: https://doi.org/10.1016/B978-0-12-802301-3.00010-7.

[2] Sandulenko, E. et al. “scummvm,” GitHub Repository, 2024. Accessed 18 Nov. 2024. Available: https://github.com/scummvm/scummvm.

[3] A. E. Hassan and B. Adams. (2024). Module 06: Reflexion Models (PowerPoint slides]. Available: https://onq.queensu.ca/d2l/le/content/959322/viewContent/5711428/View.

[4] Sandulenko, E. et al. “scummvm/engines,” GitHub Repository, 2024. Accessed 18 Nov 2024. Available: https://github.com/scummvm/scummvm/tree/master/engines.

[5] “About ScummVM.” ScummVM Wiki. Accessed 18 Nov. 2024. Available: https://wiki.scummvm.org/index.php?title=About.

[6] A. E. Hassan and B. Adams. (2024). Module 03: Architecture Styles [PowerPoint slides]. Available: https://onq.queensu.ca/d2l/le/content/959322/viewContent/5711378/View.