Concrete Architecture of ScummVM

Authors
Harry George – 20327293
Deniz Aydin – 20299521
Ben Hilderman – 20374738
Riley Spavor – 20218837
Kurtis Marinos – 20336452
Jacob Skiba – 20361187

Table of Contents

1.0 Abstract

This report examines the concrete architecture of ScummVM, an open-source platform designed to run classic adventure games by interpreting their original game engines. Building upon the conceptual architecture outlined in Assignment 1, this analysis investigates ScummVM's realized implementation, focusing on its event-driven and interpreter-based design. These architectural styles enable dynamic interactions between subsystems and seamless execution of game-specific scripts, ensuring cross-platform compatibility and extensibility.

The findings reveal significant divergences from the conceptual layered architecture, as ScummVM's concrete implementation emphasizes decentralized, event-driven communication and modular subsystems, where game engines act as interpreters coordinating with supporting components like graphics, audio, and user interfaces. The report highlights these differences through reflexion analysis, supported by dependency diagrams and use cases, offering insights into the system's design trade-offs. Finally, the report concludes with recommendations for optimizing ScummVM’s architecture while preserving its core objectives of adaptability and game preservation.

2.0 Introduction

ScummVM is an open-source software platform that enables users to play classic adventure games by interpreting their original game engines. Initially developed to support games using the SCUMM (Script Creation Utility for Maniac Mansion) engine, ScummVM has evolved to accommodate a broad range of engines, making retro games accessible on modern platforms. By bridging the gap between outdated game formats and contemporary systems, ScummVM plays a vital role in preserving gaming history.

This report builds upon the conceptual architecture presented in Assignment 1, shifting focus to the system's concrete implementation. While the conceptual model proposed a layered structure with modular interactions, the concrete architecture reflects a more dynamic reality. ScummVM's design relies on event-driven communication and the interpreter pattern, where game engines serve as interpreters for game-specific scripts, orchestrating subsystem interactions such as rendering visuals, managing audio, and handling user inputs.

This analysis explores the practical implications of ScummVM's architectural decisions, investigating the interactions between key subsystems and the trade-offs required for scalability and cross-platform functionality. Discrepancies between the conceptual and concrete architectures are evaluated through reflexion analysis, offering insights into the challenges of implementing large-scale open-source software. Finally, the report concludes with recommendations to enhance ScummVM’s architecture while maintaining its core goals of adaptability and retro game preservation.

3.0. Derivation Process

We derived ScummVM’s concrete architecture by analyzing its GitHub repository, using the Understand tool for static analysis, and consulting community forums for additional context. The GitHub repository provided an overview of the code structure and key subsystems, such as Engines, GUI, and Common, with a specific focus on the SCI engine’s role.

The Understand tool allowed us to visualize subsystem dependencies, analyze code complexity, and identify performance-critical components. Dependency diagrams clarified interactions between major subsystems like Graphics, Audio, and Common, while also highlighting the SCI engine’s integration.

Community forums and documentation offered insights into ambiguous dependencies and subsystem responsibilities, helping validate our findings and understand design decisions.

This process revealed challenges, such as complex subsystem interactions and a large codebase, which required careful cross-referencing of multiple resources. Despite these hurdles, the combined approach provided a clear view of the concrete architecture, setting the foundation for further analysis and reflexion.

4.0 Top-Level Architecture

ScummVM's concrete architecture is centered on an **event driven** model and **interpreter design**, enabling dynamic interactions between its modular subsystems. This approach allows ScummVM to adapt to the unique requirements of over 300 supported games while maintaining cross-platform compatibility. Unlike the layered conceptual architecture described in Assignment 1, the concrete implementation reflects a decentralized structure where game engines serve as the primary drivers of subsystem interactions.

Description of Subsystems

The key subsystems in ScummVM include Engines, Common, GUI, Graphics, Audio, Image, Math, and Backends. These components work cohesively, orchestrated by the event-driven flow of game logic. The Engines subsystem, in particular, interprets game-specific scripts and manages requests to supporting modules. Below is an overview of the primary subsystems and their roles:

- **Engines:** Interprets game scripts, manages game states, and serves as the central control for game-specific functionality. It coordinates with Graphics, Audio, and GUI subsystems to execute game logic dynamically.
- **Common:** Provides shared utilities and abstractions to all subsystems, ensuring consistency and reducing redundancy.
- **GUI:** Handles user interface elements like menus and settings, dynamically interacting with Engines to execute user-driven events.
- **Graphics:** Renders visuals based on instructions from Engines, ensuring compatibility with game-specific graphical requirements.
- **Audio:** Manages sound playback, including music, effects, and dialogues, as requested by the Engines.
- **Image:** Processes graphical assets such as sprites and backgrounds, ensuring format compatibility and integration with the Graphics subsystem.
- **Math:** Supplies essential calculations for Engines and Graphics, including transformations, scaling, and spatial computations.
- **Backends:** Provides platform-specific functionality, abstracting hardware differences to ensure seamless operation across multiple devices.

Event-Driven Communication

The event-driven nature of ScummVM governs the interactions between subsystems:

- User actions, such as input through the GUI, generate events that are processed by the Engines.
- The Engines issue requests to subsystems like Graphics and Audio, which then execute tasks asynchronously when possible.
- Subsystems communicate bidirectionally, enabling dynamic updates to game states, rendering, and audio playback.

Interpreter Role

At the heart of ScummVM's architecture, each game engine operates as an interpreter for game-specific scripts. This design allows ScummVM to replicate the behavior of the original game engines while maintaining compatibility with modern hardware. The interpreter pattern ensures that game logic is executed in real-time, triggering subsystem responses as defined by the script.

Control and Data Flow

Control flow originates from the Engines, which dynamically manage subsystem interactions based on game events. Data flow is similarly decentralized:

- **Event Queues:** Events, such as user inputs or game logic triggers, are queued and dispatched to the relevant subsystems.
- **Shared Resources:** Common provides a centralized repository for configurations, runtime states, and shared utilities accessed by multiple subsystems.

This event-driven and interpreter-based architecture enables ScummVM to adapt dynamically to a wide range of games and platforms, prioritizing flexibility and scalability over strict adherence to hierarchical layers.

5.0 Dependency Analysis

The dependency analysis of ScummVM's architecture reveals a complex and interconnected system, driven by its event-driven and interpreter-based design. While the conceptual model suggested a hierarchical structure, the concrete architecture illustrates a more decentralized network of relationships. At the heart of this architecture lies the Engines subsystem, orchestrating real-time interactions with supporting components such as Graphics, Audio, and GUI. This decentralized configuration reflects practical optimizations aimed at achieving scalability and cross-platform compatibility.

Insights from the Dependency Diagram

The dependency diagram offers a visual representation of the relationships among ScummVM's subsystems, emphasizing dynamic, event-driven communication rather than a rigid hierarchical structure. The arrows depict both standard and bidirectional dependencies, highlighting critical control and data flow paths throughout the system.

The Engines subsystem emerges as the central orchestrator, exhibiting the highest number of outward dependencies. By interpreting game scripts and dynamically interacting with components like Graphics and Audio, it fulfills the requirements of game logic. Furthermore, its dependency on the Common subsystem underscores the reliance on shared resources for operations such as memory allocation, file handling, and configuration.

The Common subsystem serves as a shared backbone, linking nearly all other components. It provides utilities such as data structures, runtime state management, and platform-agnostic abstractions, ensuring consistent interactions across subsystems while supporting ScummVM's extensibility.

Subsystems such as Graphics and Audio specialize in specific outputs, handling tasks like rendering visuals and playing sounds as triggered by Engines. Meanwhile, the Backends subsystem offers platform-specific functionalities—including inputs, outputs, and file handling—that enable ScummVM to function seamlessly across multiple platforms.

Bidirectional dependencies further enhance real-time interactions between components. For instance, the GUI subsystem relies on Graphics for rendering visuals, while simultaneously updating the Engines based on user input.

Control and Data Flow Dynamics

Control flow within ScummVM is predominantly decentralized. User inputs captured by the GUI are converted into events, which are then dispatched to the Engines for interpretation. The Engines, in turn, trigger responses from subsystems such as Audio and Graphics. Real-time updates between components ensure a seamless exchange of information; for example, Graphics can request data from Common to render visuals, while the Engines update GUI states based on game progress.

Data flow emphasizes the efficient use of shared resources. Events are managed through an event queue, ensuring asynchronous processing wherever possible. The Common subsystem acts as a centralized repository, minimizing redundancy and maintaining consistency across the architecture.

Performance-Critical Relationships

Certain dependencies are pivotal to ScummVM's performance. The frequent communication between Engines and Graphics ensures real-time rendering of visuals based on game logic, with any delays potentially causing visual lag and affecting gameplay. Similarly, precise synchronization between Engines and Audio is crucial for aligning audio playback with visual and gameplay events, creating an immersive user experience. Additionally, Backends must optimize platform-specific operations such as file I/O and input handling to ensure smooth performance on various devices.

Concurrency Considerations

Although ScummVM primarily operates in a single-threaded mode due to its event-driven design, some subsystems leverage concurrency to boost performance. Audio playback often runs in a separate thread to avoid interruptions during gameplay, while Graphics employs buffering techniques for smooth frame transitions. Asynchronous save-state handling further minimizes disruptions, allowing for seamless user experiences.

Implications for Architectural Design

This dependency analysis underscores ScummVM's reliance on dynamic, event-driven interactions over static layering. This architectural choice enhances responsiveness, scalability, and adaptability, allowing ScummVM to efficiently support a diverse array of games and platforms.

6.0 Chosen Subsystem Analysis

This section focuses on the SCI Engine, a critical subsystem within ScummVM. Responsible for emulating Sierra's Creative Interpreter, the SCI Engine enables many classic Sierra games to run seamlessly on modern platforms. The analysis explores the subsystem's role, internal architecture, and its interactions with other components of ScummVM.

Subsystem Overview

The SCI Engine is a vital part of ScummVM's Engines subsystem, designed to emulate the gameplay mechanics, script execution, and interaction logic of Sierra games. Its extensive interactions with other subsystems underscore its central role in ScummVM's architecture. For instance, the SCI Engine integrates with Graphics to render scenes and animations according to game logic, with Audio to play sound effects and background music, and with the GUI to manage user inputs and provide a cohesive interface. Additionally, it relies on the Common subsystem for accessing shared utilities and core resources, such as configuration data and runtime management. The dependency diagram effectively illustrates these relationships, highlighting how the SCI Engine’s files and modules contribute to the broader functionality of ScummVM.

Internal Architecture

The SCI Engine's internal architecture is divided into several key files and modules, each serving a distinct purpose. The engine.cpp and engine.h files form the engine's core, handling the main execution loop and bridging game-specific logic with subsystem calls, such as Graphics and Audio. The metaengine.cpp and metaengine.h files are responsible for game detection and configuration, ensuring the engine is correctly set up based on user input and game files. In-game dialog windows and overlays are managed by dialogs.cpp and dialogs.h, which integrate user interactions with game menus. Save-state functionalities are facilitated by saves.cpp and savestate.h, enabling serialization of game states. Lastly, the advancedDetector.cpp and advancedDetector.h files assist in recognizing and verifying game files, ensuring compatibility with ScummVM.

Interactions and Dependencies

The SCI Engine showcases several interdependent relationships between its components and other subsystems. For instance, engine.cpp handles runtime execution while relying on metaengine.cpp for environment configuration, ensuring the correct game logic is executed. Dialog management through dialogs.cpp integrates closely with the GUI subsystem to present interfaces like inventory screens or game options. Save-state functionalities leverage utilities from the Common subsystem for file handling and data serialization. The advanced detection mechanisms implemented in advancedDetector.cpp validate user-provided game files, enabling seamless game initialization.

Concurrency and Performance Considerations

Concurrency plays an important role in the SCI Engine’s design to ensure optimized performance. Rendering tasks are delegated to the Graphics subsystem to prevent delays in game execution, while audio playback operates on a separate thread, maintaining uninterrupted sound output. Save-state serialization further minimizes gameplay disruptions by employing asynchronous operations where supported.

7.0 Reflexion Analysis

The reflexion analysis examines the differences between the conceptual architecture developed in Assignment 1 and the concrete architecture derived from the Understand tool and the SCI engine code. This comparison reveals significant divergences, demonstrating how ScummVM’s implementation transitions from a theoretical layered design to a more practical, event-driven, and interpreter-based model. These differences reflect the system's adaptations to support a wide variety of games and ensure scalability across platforms.

High-Level Comparison

While there are notable alignments with the conceptual architecture, certain divergences illustrate the pragmatic choices made during ScummVM's development. The modular design, which was central to the conceptual architecture, remains a strong feature of the concrete implementation. Subsystems like

Engines, Graphics, Audio, and GUI are clearly delineated, and the Engines subsystem fulfills its intended role as the central component, interpreting game scripts and orchestrating subsystem interactions. Similarly, the Common subsystem adheres to its conceptual purpose of providing shared resources and utilities, reducing redundancy and promoting consistency.

However, several key differences emerge. The conceptual architecture proposed a layered model with distinct separations between tiers such as the user interface, logic, and subsystems. In contrast, the concrete architecture adopts a decentralized, event-driven structure, allowing components to interact dynamically based on events. Control flow also diverges from the hierarchical structure envisioned in the conceptual model, exhibiting distributed control where the Engines subsystem facilitates real-time interactions across subsystems. While the conceptual design emphasized a predominantly single-threaded approach, the concrete architecture incorporates concurrency in areas like audio playback and save-state handling to ensure smooth performance. Additionally, tighter coupling between components such as dialogs.cpp (GUI) and engine.cpp (Engines) is observed, deviating from the loosely coupled components envisioned in the conceptual design.

Subsystem-Specific Comparison: SCI Engine

The SCI Engine aligns with its conceptual design as an interpreter responsible for managing game-specific logic, user inputs, and state transitions. Its interactions with subsystems such as Graphics and Audio mirror the expected behavior, translating script instructions into actionable outputs. However, the concrete architecture reveals divergences in implementation details.

The file structure and module responsibilities within the SCI Engine deviate from the cohesive unit outlined in the conceptual model. Instead, the concrete design features a distributed structure, with modules such as `dialogs.cpp`, `saves.cpp`, and `metaengine.cpp` handling distinct functionalities. Concurrency mechanisms are also more prominent than anticipated, with threading used for audio playback and save-state operations to enhance performance. Additionally, the `advancedDetector.cpp` module introduces a tighter integration with the SCI Engine for game validation, reflecting an added layer of dependency not accounted for in the conceptual design. The SCI Engine's real-time event handling further illustrates a shift from the centralized control proposed in the conceptual model to a more dynamic execution approach.

Rationales for Divergences

The divergences between the conceptual and concrete architectures stem from practical considerations. The adoption of an event-driven model addresses real-world demands for responsiveness and scalability, particularly for resource-intensive tasks like rendering and audio playback. Concurrency mechanisms, though absent from the original conceptual design, enhance performance and provide a seamless gameplay experience. Additionally, the need to support legacy games necessitates tighter subsystem integration, diverging from the purely modular structure envisioned earlier. The evolution of the architecture to accommodate a growing library of supported games and platforms also required more interconnected and flexible components.

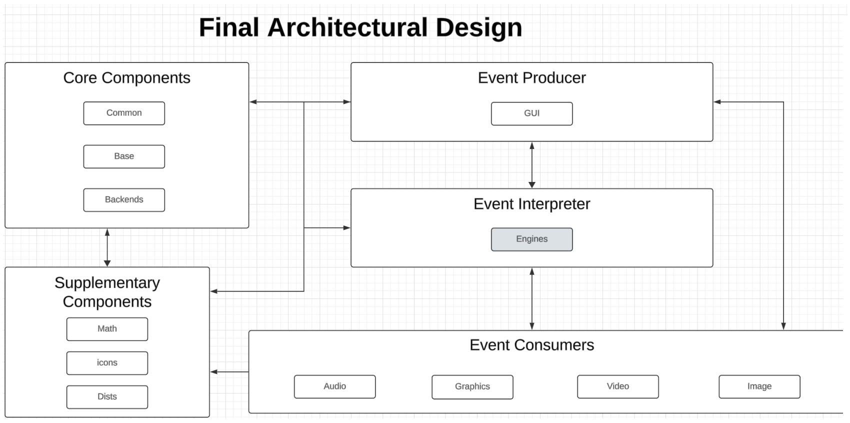
Implications for System Design

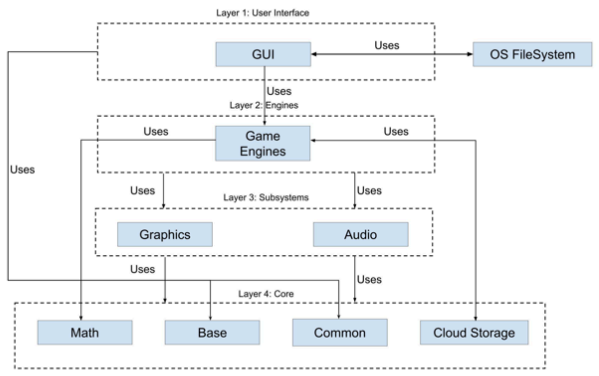
The reflexion analysis underscores the balance ScummVM achieves between modularity and maintainability and the practical challenges of supporting a diverse array of games. While the event-driven and interpreter-based model departs from the hierarchical assumptions of the conceptual architecture, it ensures scalability and adaptability. These findings highlight the importance of flexibility in real-world software engineering, particularly for systems with broad and evolving requirements.

8.0 Diagrams and Use Cases

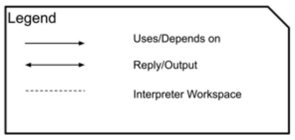
New (left) vs. Old (right) Concrete Architecture Diagram

The updated concrete architecture diagram illustrates ScummVM's event-driven flow and decentralized design. Unlike the conceptual model's layered structure, the diagram highlights the dynamic, bidirectional dependencies between subsystems. The Engines subsystem serves as the core interpreter, translating game logic into actions executed by supporting modules like Graphics, Audio, and GUI. Bidirectional interactions between GUI and Graphics reflect real-time event-driven feedback during gameplay, while the Common subsystem provides shared resources and ensures consistent functionality across all modules.



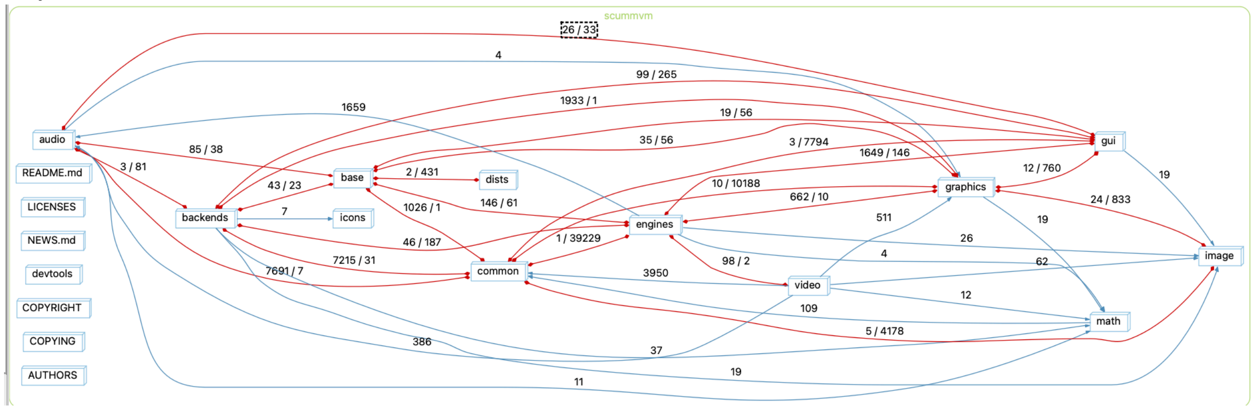


This section presents diagrams and sequence use cases that illustrate ScummVM's concrete architecture. The updated diagrams emphasize the event-driven and interpreter-based characteristics of the system, while the use cases demonstrate the practical manifestation of these interactions.



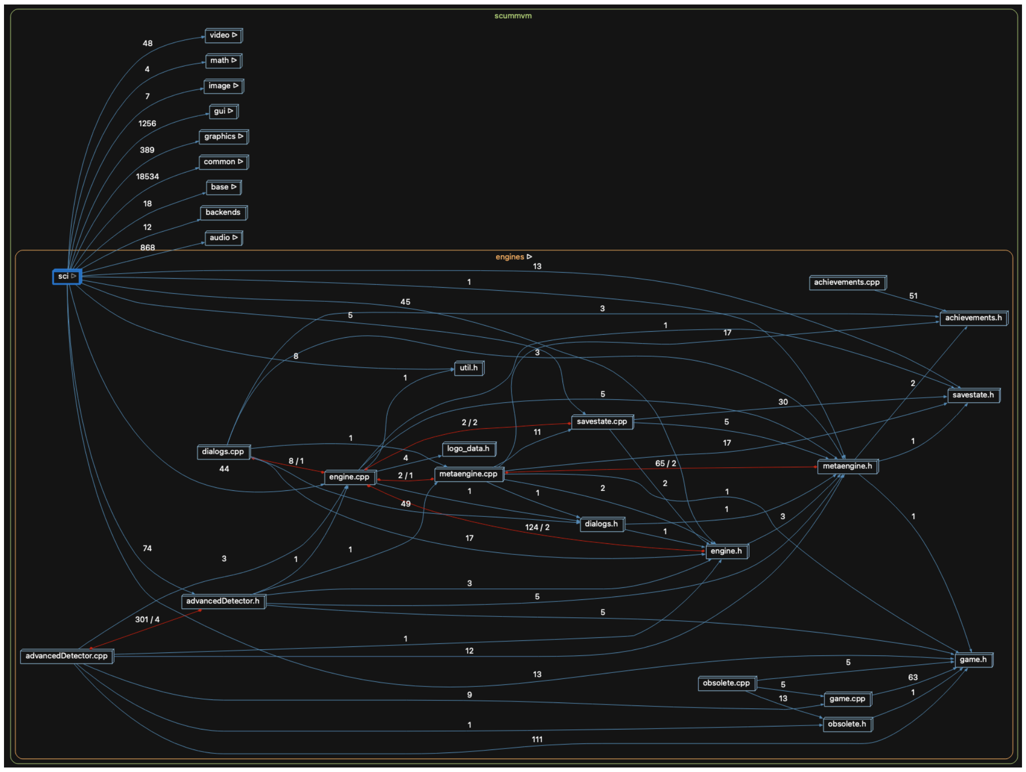
8.1 Component Dependency Diagram

The dependency diagram highlights the intricate relationships between ScummVM's components. Unlike the conceptual architecture, which lacked many critical dependencies, the concrete architecture reveals a complex web of interactions. This complexity confirms that ScummVM's architecture is not layered, as a layered design would exhibit linear dependencies between adjacent layers. Instead, the diagram reflects a dynamic, non-linear structure.



8.2 Subsystem-Specific Diagram: SCI Engine

The SCI Engine's internal structure demonstrates how its modules interact both internally and externally with other subsystems. For example, dialogs.cpp integrates with GUI to display user interfaces during gameplay, while saves.cpp manages save-state functionality by leveraging asynchronous operations. The central engine.cpp file orchestrates these interactions, interpreting game scripts and dispatching commands to external modules like Graphics and Audio. This event-driven architecture ensures real-time responsiveness and efficient resource management.

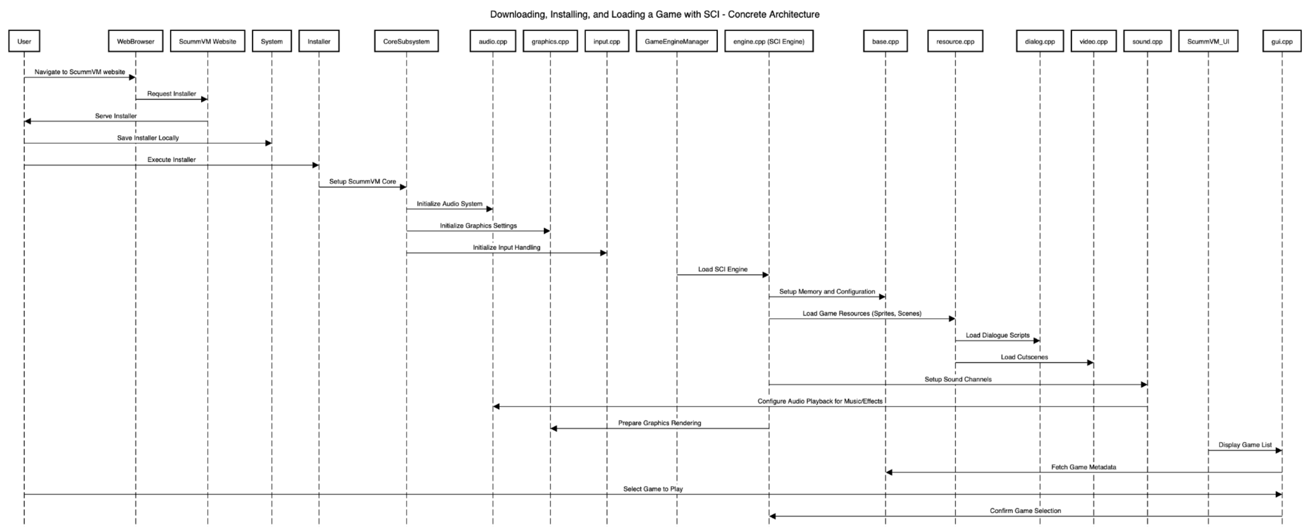


8.3 Use Cases

The following use cases provide concrete examples of ScummVM's event-driven design in action, focusing on the SCI Engine's role in orchestrating gameplay.

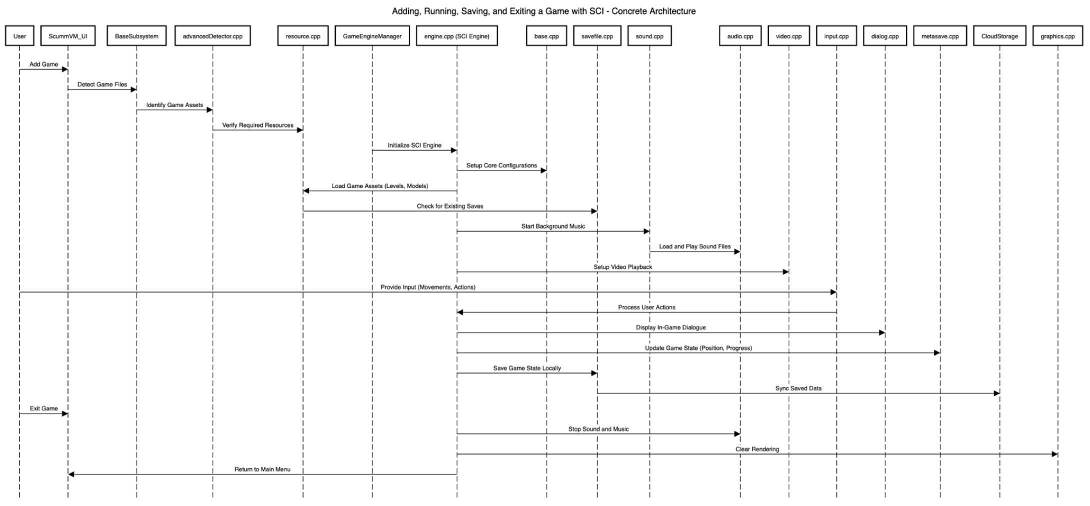
8.3.1 Use Case 1: Downloading, Installing, and Loading a Game with SCI

This use case begins with the user downloading and installing ScummVM. After installation, the launcher initializes the Core subsystem, which in turn loads the GUI, Graphics, and Audio modules. When the user selects a game, an event is sent from the GUI to the Engines subsystem, which validates the game files through advancedDetector.cpp. Once validation is complete, the SCI Engine initializes game assets, triggering the Graphics subsystem for rendering and the Audio subsystem for sound playback. Gameplay begins with the SCI Engine interpreting game logic and coordinating subsystem responses dynamically.



8.3.2 Use Case 2: Adding, Running, Saving, and Exiting a Game with SCI

This use case starts when the user adds a game through the GUI, which triggers the SCI Engine to validate game files using advancedDetector.cpp. During gameplay, the SCI Engine processes real-time user inputs received via GUI, updating the game state and triggering subsystem actions like rendering visuals through Graphics and playing sound effects via Audio. When the user chooses to save progress, the SCI Engine utilizes saves.cpp to serialize the game state, ensuring asynchronous file handling to minimize gameplay interruptions. On exiting, the SCI Engine clears resources, stops audio playback, and seamlessly returns control to the GUI.



These diagrams and use cases illustrate ScummVM’s dynamic and decentralized architecture, emphasizing its ability to handle diverse games and user interactions efficiently through event-driven and interpreter-based mechanisms.

9.0 External Interfaces

ScummVM employs an interpreter-based architectural style to support diverse platforms, adapting its core functionality through platform-specific backends located in the backends/ directory, with entry points such as posix-main.cpp for Linux and macosx-main.cpp for macOS. Its graphical user interface (GUI) is built on a customizable Theme Engine (gui/themes/), utilizing the STX format to manage resolution-dependent layouts and assets for consistent visuals across devices. Input handling (keyboard and mouse) is abstracted for uniform interactivity, while the event-driven system manages various game engines, each implementing the Engine::run() function to interpret game files and media formats. The Python-based scummtheme.py tool streamlines the creation and packaging of GUI themes, compiling them into zip formats or embedding them directly in ScummVM. While primarily designed for offline use, ScummVM is progressively introducing cloud-based save synchronization and networking features. The Launcher, combined with libraries like TinyGL and SDL, ensures smooth rendering and input handling while providing a unified interface for game selection and execution.

10.0 Data Dictionary

ScummVM	A program that allows you to run certain classic graphical point-and-click adventure games.
SDL	Simple DirectMedia Layer, a cross-platform library used by ScummVM for handling graphics, input, and audio.
API	Application Programming Interface, which provides a set of functions for interfacing with other software components.
Engine	Refers to the part of ScummVM responsible for running individual games, each game often requiring a specific "engine."
Graphics Subsystem	Responsible for rendering game graphics and UI. Uses SDL for rendering across platforms.
Audio Subsystem	Plays back audio files and sound effects. Supports a variety of formats (e.g., MP3, Ogg).
Input Subsystem	Manages keyboard, mouse, and game controller input.
Scripting Engine	Runs game-specific scripts, such as SCUMM scripts for LucasArts games.

Operating Systems	Windows, macOS, Linux, iOS, Android, and other platforms.

11.0 Naming Conventions

Consistent and intuitive naming conventions are critical for maintaining code readability, ensuring collaboration among contributors, and facilitating long-term maintainability. ScummVM adheres to specific naming standards across its modules, classes, files, and functions. This section outlines the naming conventions used in ScummVM's concrete architecture.

Core Classes

- **Common::File** - Handles file I/O operations.
- **Audio::Mixer** - Manages audio mixing and playback.
- **Graphics::Surface** - Manages the 2D graphics drawing surface.

File Structure

- **/engines/**: Contains subdirectories for each supported game engine.
- **/gui/**: Contains code related to the ScummVM interface.
- **/audio/**: Implements audio functionalities.
- **/graphics/**: Manages rendering and graphical interface elements.
- **/backends/**: Platform-specific code, e.g., for SDL integration.
- **/common/**: Contains utilities and shared resources used by various modules.

12.0 Conclusions

The analysis of ScummVM’s concrete architecture highlights its shift from a conceptual layered model to a dynamic, **event-driven** and **interpreter-based** design. This approach ensures real-time responsiveness and adaptability, enabling seamless interactions between subsystems and efficient execution of game logic.

The Engines subsystem acts as a central orchestrator, dynamically interacting with components like Graphics, Audio, and GUI, while the Common subsystem provides shared resources to maintain consistency. These architectural choices prioritize scalability and performance, supporting a diverse library of games across platforms.

While the event-driven design enhances flexibility, it introduces complexity in managing dependencies and maintaining concurrency in areas like audio playback and save-state handling. Despite these challenges, ScummVM's architecture balances modularity and extensibility, offering valuable insights into the pragmatism required for large-scale software systems.

13.0 Lessons Learned

Analyzing ScummVM’s architecture provided valuable insights into architectural design and practical challenges in implementing complex systems.

1.Subsystem Responsibilities

ScummVM's separation of roles, such as Engines managing game logic and Backends handling platform-specific tasks, demonstrated the importance of clear responsibilities. This design supports flexibility for adapting to new games and platforms.

2. Bridging Concept and Implementation

The comparison between conceptual and concrete architectures highlighted inevitable divergences. For instance, added dependencies and concurrency mechanisms in the SCI engine revealed how real-world constraints shape implementation choices.

3. Balancing Tools and Manual Analysis

The Understand tool was essential for visualizing dependencies and relationships, but manual inspection and documentation were critical for clarifying ambiguous interactions. Combining both approaches ensured a deeper understanding.

4. Value of Documentation

Developer forums and official documentation were invaluable for understanding design decisions and clarifying complex dependencies. This experience reinforced the importance of accessible and detailed resources for maintaining large systems.

5. Challenges with Legacy Systems

Working with older game logic in the SCI engine required accommodating legacy constraints while implementing modern features. This highlighted the need for careful integration strategies to preserve original functionality.

6. Team Collaboration

Effective communication and shared tools allowed us to address challenges collectively and ensure a cohesive analysis. Regular updates kept the team aligned throughout the project.

7. Recommendations

These lessons emphasize the importance of clear roles, iterative refinement of designs, and leveraging both tools and documentation for thorough analysis. They will guide our approach to future projects involving complex or legacy systems.

14.0 References

1. ScummVM Team. (2023). *ScummVM main repository*. GitHub. Retrieved from https://github.com/scummvm/scummvm
   This repository provided access to the source code, directory structures, and documentation essential for analyzing ScummVM's concrete architecture. It allowed us to investigate subsystem dependencies, examine design patterns, and verify implementation details.
2. ScummVM Forums. (2010). *How is ScummVM's Architecture Structured?* https://forums.scummvm.org/viewtopic.php?t=7886. Accessed 12 Nov. 2024.
3. SDL Wiki. *SDL3 FrontPage*. *n.d.*, https://wiki.libsdl.org/SDL3/FrontPage. Accessed 14 Nov. 2024.