Concrete Architecture of ScummVM

November 18, 2024

Queen's University
CISC322
Instructor: Prof. Bram Adams

Corey McCann 20344491 21ccm12@queensu.ca
Hayden Jenkins 20344883 21hjdm@queensu.ca
Ben Leray 202347460 21bl45@queensu.ca
Simon Nair 20297356 21scn1@queensu.ca
Jeremy Strachan 20336601 21js138@queensu.ca
Ryan Van Drunen 20331633 21rdbv@queensu.ca

Table of Content

1. Abstract

ScummVM is an advanced software emulator designed to run retro games built for old hardware on modern platforms. Initially created to only support titles of the Scumm engine, it has expanded over the years to support dozens of game engines and hundreds of games. ScummVM's success stems from its sophisticated architecture built around a layered style architecture to emphasize modularity and portability. This enables developers to add support for new game engines without significantly altering the codebase, making it an essential tool for preserving and experiencing retro games.

2. Introduction and Overview

This report explores ScummVM's concrete architecture by analyzing its source code. While similar to the derived conceptual architecture, our concrete architecture has several key differences, such as the relocation of the Game UI Component, the integration of the Resource Management System into the Core Engine, and the addition of a shared utility layer. To understand these differences, we will use a software reflection framework.

We will also examin the Sci Engine Subsystem by constructing both the conceptual and concrete architectures, which also follow a layered style with an interpreter style architecture at the core of the engine. By analyzing the differences between the concrete and conceptual architectures and each component's dependencies through a reflexion analysis framework, we can uncover valuable insights into ScummVM's architecture and its underlying functionality.

3. Derivation Process

To better understand the concrete architecture of ScummVM and its subsystems, a combination of the SCI Tools Understand, and online documentation were used to derive and understand its underlying components and different processes.

3.1 Derivation Process of the Concrete Architecture

The SciTools Understand program was the primary tool used to investigate the concrete architecture of ScummVM. Understand is a professional software development tool that allows users to perform code analysis on large programs. By using Understand, our team generated a dependency graph of each component and their relationships. By analyzing the different interactions of each module and their dependencies, a mapping of the concrete architecture was created. By using our conceptual architecture of ScummVM from Assignment 1, we also performed a reflexion analysis on the concrete architecture to understand and explore divergent and absent dependencies.

3.2 Derivation Process of the Conceptual and Concrete Architecture of the Sci Engine Subsystem

The conceptual architecture of the Sci Engine Subsystem was found in a similar method to ScummVM's conceptual architecture detailed in Assignment 1. Using a combination of online

high-level documentation and different game engine and emulator reference architectures, a conceptual map was created to model ScummVM's Sci Engine.

The concrete architecture of the Game Engine Subsystem was then derived using SciTools Understand in the same process described for the concrete architecture of ScummVM but with a few differences. In addition to relying on the dependency graph generated by Understand, documentation, and file names as well as the functions and logic within the files were used to determine the relationship between the different architecture components. This was possible as the source code for the SCI engine was much smaller and more manageable then that of the entire ScummVM system. By inspecting files, folder structures, C header files, function calls, variables, and code comments, we were able to divide the source code into specific component groups. This approach allowed for a much more accurate and detailed representation of the concrete architecture of the Sci Engine.

4. Concrete Architecture

4.1 Review of ScummVM Conceptual Architecture

The conceptual architecture of ScummVM combines multiple styles to enable classic adventure games to run on modern platforms. Its layered structure includes the User Interface Layer, which manages the GUI and settings, and the Core Engine Layer, responsible for resource management and the virtual machine. The Platform Abstraction Layer standardizes platform-specific tasks, ensuring compatibility across different device platforms, while the Game Engine Layer supports individual game engines like SCI and Kyra. Lastly, the Sub-System Layer manages essential components such as graphics, audio, and input-output.

This architecture also incorporates the interpreter style, where game scripts are processed and executed through the Core Engine, Virtual Machine, and Kernel rather than through hardware emulation. The architecture ensures modularity and extensibility, with key components like the Resource Management System, Virtual Machine, and Sub-Systems working cohesively. An approach that allows ScummVM to maintain cross-platform compatibility and support a wide range of classic games without altering its core structure.

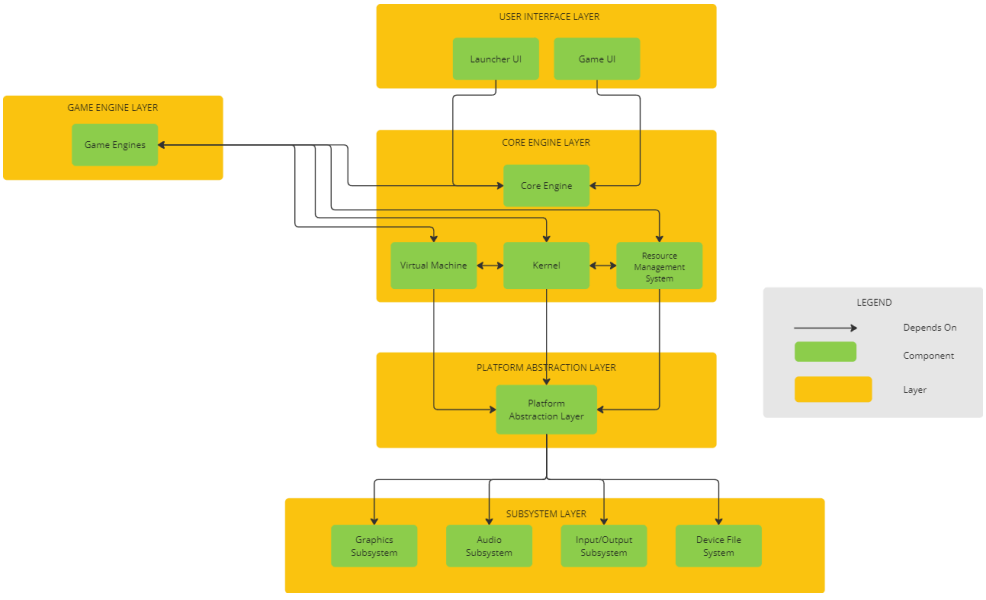


Figure 1: Box and Line Diagram of Conceptual Architecture

4.2 Overview of Top-Level Concrete Architecture

The Top-Level Concrete Architecture of ScummVM closely aligns with the conceptual architecture we derived, incorporating several different architectural styles. The primary architecture style still remains layered, with different subsystems being grouped together to increase performance and maintainability. This layering style, most importantly, supports game engines and subsystems to be added or modified independently of each other, making the architecture highly extensible and adaptable to new platforms and engines. The interpreter style is also clearly present in the concrete architecture, with the virtual machine and kernel components interpreting game scripts. There is also still the client-server style for managing inter-component communication, and keeping the user interface separated from all the core game logic. However, as we will see, the concrete architecture, particularly its dependencies, reveals a more tightly coupled system then initially anticipated, making some of the architectural styles less evident than in the conceptual architecture.

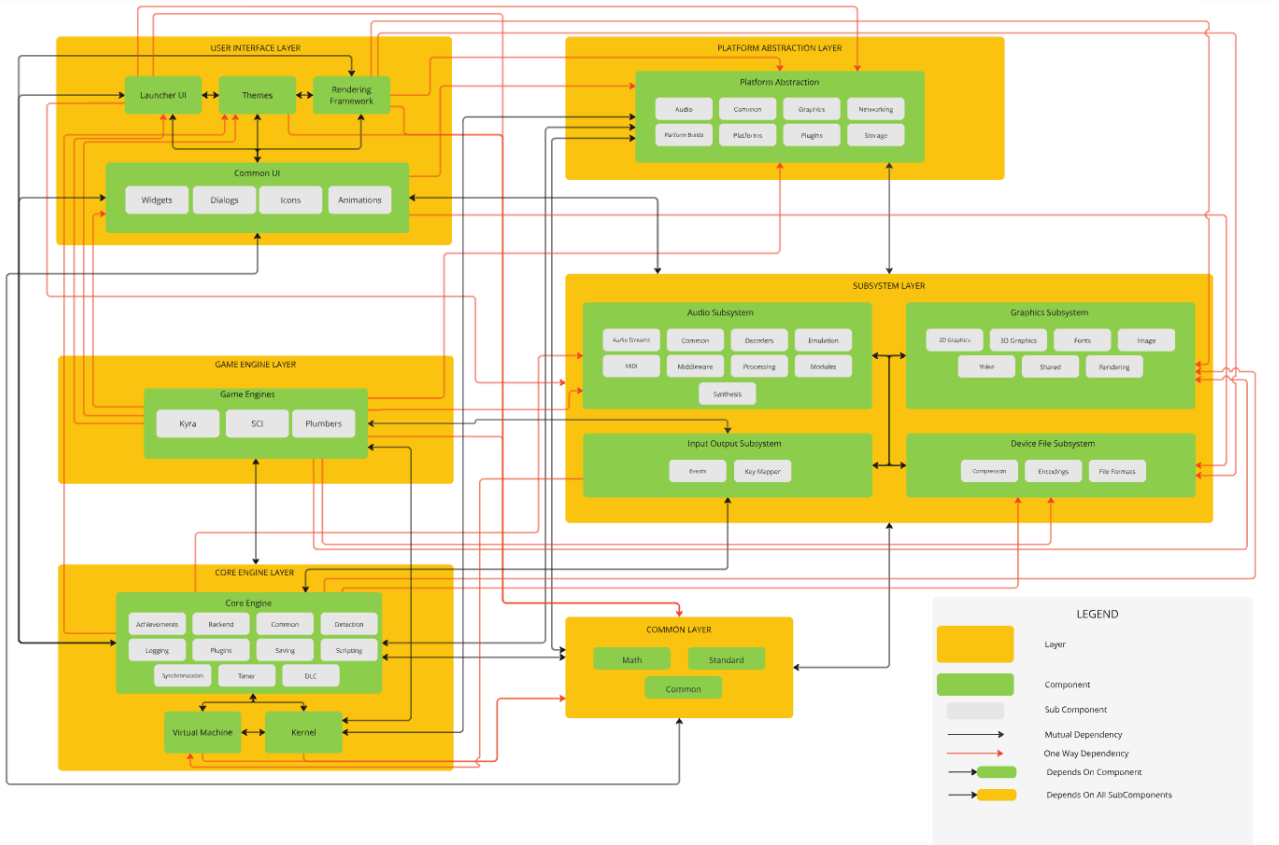


Figure 1: Box and Line Diagram of Concrete Architecture

4.3 Reflexion on Top-Level Concrete Architecture

Our concrete architecture of ScummVM reveals several differences from the conceptual architecture we initially proposed. This analysis examines these differences and explains the underlying rationale for each architectural deviation.

Game UI Component Relocation

The Game UI Component relocation was a significant deviation in dependencies from the User Interface Layer to the Game Engine Layer, with the Game Engine Component now directly calling the Common UI components and Rendering Component. This change resulted from the fact that game-specific UI logic is inherently tied to game engine functionality, with different engines often requiring unique UI implementations. By integrating UI handling directly into the Game Engine Layer, the architecture achieves better performance and reduces cross-component complexity, as demonstrated through the SCUMM engine's ability to control its inventory system UI directly through the GUI rendering framework.

Resource Management System Integration

The Resource Management System (RMS) simplification and integration into the Core Engine and Game Engine Components streamlines resource handling by directly relying on the Platform Abstraction Layer and Subsystem Layer. This architectural change eliminates the need for a separate system while still providing flexibility for game engines to implement custom resource management as needed. This change better reflects the concrete implementation, as seen in the SCI engine's direct sprite resource management functionality.

Addition of Common Layer

The introduction of the Common Layer adds a new component to manage access to shared utilities and standard C libraries across all layers and components. This addition significantly reduces code duplication and increases maintainability by providing standardized implementations of basic data structures and other commonly used functions.

Enhanced User Interface Layer

The Enhanced User Interface Layer, featuring the new Rendering Framework and Common UI components, represents a more sophisticated and accurate approach to UI management. This enhancement creates a clearer separation of logic between GUI rendering and general UI logic while providing a set of reusable components such as widgets, dialogs, icons, and animations. This improvement ensures consistency across the application and reduces duplication of common UI components, as evidenced by the button widget and dialog implementations shared between the launcher and in-game menus, as well as the window management logic shared across different engines.

Other Key Dependency Changes

↔ : mutual dependency
→ : one-way dependency

Game Engine → Common UI (Changed from indirect to direct)

In the conceptual architecture, the Game Engine communicated with the Game UI indirectly through the Core Engine. This was replaced with a direct dependency to reflect the inherent coupling between the game-specific UI logic and the Game Engine functionality.

Launcher UI ↔ Core Engine (Changed from one-way to mutual)
Changed from one-way to mutual to enable real-time state synchronization and allow the Core Engine to provide essential services while receiving game selection and initialization information, maintaining UI responsiveness.

Graphics Subsystem ↔ Device File Subsystem (Direct mutual dependency)
Originally, file access was routed through the Platform Abstraction Layer, resulting in an indirect relationship. This was revised to a direct mutual dependency to address graphics performance and resource management efficiency, allowing more streamlined access to assets.

Themes ↔ Rendering Framework (New component)
The conceptual architecture had no explicit theme system in place. The concrete implementation features direct dependencies for theme rendering, allowing for consistent themes across all the UI components and dialogs, which was originally overlooked in our conceptual architecture.

Core Engine → Themes (New component)
This resulted from the addition of the Theme Component to maintain control of the themes across all UI components and visuals, including in the Core Engine.

Input Output Subsystem ↔ Virtual Machine (New mutual dependency)
Initially, there was no direct communication between these components. The concrete implementation added this dependency to allow direct I/O handling for game engine scripts, aswell as improved I/O handling performance and increased game responsiveness.

Launcher UI → Platform Abstraction (Direct access)
New direct dependency to bypass intermediary components and improve and simplify responsiveness and logic related to managing settings, launching game engines, adding and managing games, and accessing saved game states.

Launcher UI ↔ Common UI (New centralized system)
This divergence arose from the addition of the new Common UI component, which provides reusability and reduces code duplication. It centralizes shared UI elements like icons, cursors, dialogs, and animations.

5. SCI Engine Subsystem

5.1 Conceptual Architecture ~ Sci Engine

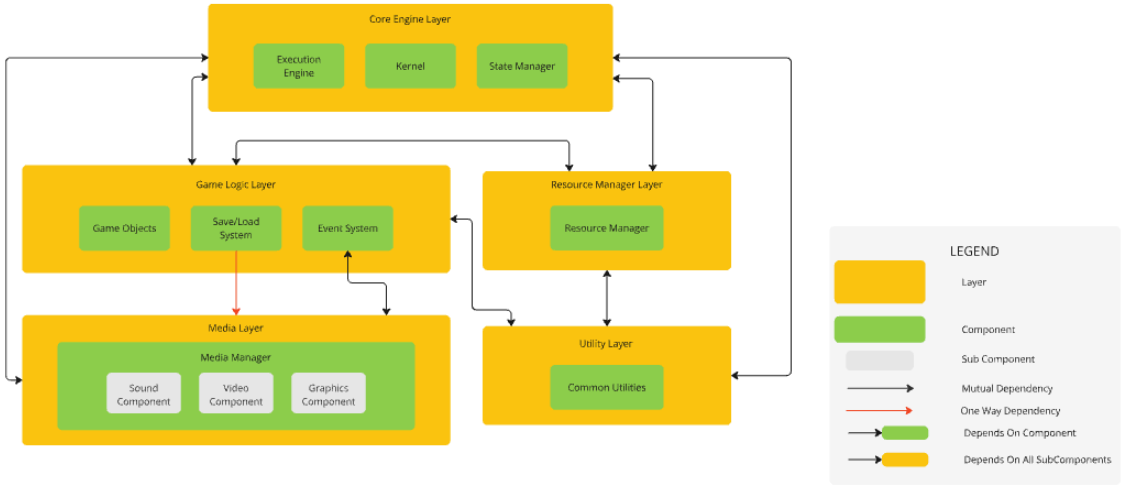


Figure 3: Box and Line Diagram of SCI Engine Conceptual Architecture

Our conceptual architecture for the Sci game engine follows a well-organized layered architecture where each layer has a clear role and responsibility. This model represents expected dependencies and highlights the importance of modular interaction between components. To give a structured analysis of our conceptual architecture, we will first give a description of each of the layers as well as their subcomponents, and finish with an explanation of the core dependencies.

The **Core Engine Layer** serves as the foundational layer responsible for managing the core runtime environment for the game. It handles the execution of game scripts, state transitions, and low-level operations, providing the backbone for other layers to function.

- **Execution Engine:** this component is responsible for running game scripts and handling the primary logical flow within the game. It includes functionalities of the script engine and virtual machine, interpreting game instructions and translating them into actionable events.
- **Kernel:** The Kernel provides essential services and system calls that the game scripts rely on. It acts as an intermediary between high-level game logic and low-level system functions.
- **State Manager:** Manages the game state, keeping track of variables, memory segments, and execution flow. This includes functionalities for saving and restoring the game's internal state, which is crucial for resuming gameplay seamlessly.

The **Game Logic Layer** handles the game's high-level logic and interactions, bridging between user actions and game outcomes. It defines the rules and events that drive gameplay.

- **Save/Load System:** Manages the saving and loading of game states, allowing players to store their progress and return to it later.
- **Game Objects:** Represents interactive entities within the game, such as characters, items, and environmental elements. Game Objects are manipulated and updated as the player progresses throughout the game.
- **Event System:** Handles the management and triggering of game events, including interactions, scripted sequences, and real-time processing of player input.

The **Media Layer** is responsible for rendering audio, video, and graphical components of the game. It translates in-game events into sensory feedback for the player.

- **Media Manager:** Oversees all media-related operations, ensuring synchronization and rendering of audio and visual elements. It includes:
    - **Sound Component:** Handles background music, sound effects, and other audio-related features.
    - **Video Component:** Manages video sequences or cutscenes that are part of the game experience.
    - **Graphics Component:** Responsible for rendering graphical elements, including 2D/3D models, animations, and on-screen text.

The **Resource Management Layer** provides access to game assets, such as textures, sounds, and scripts. It handles the loading, organization, and caching of resources required by other layers.

- **Resource Manager:** Manages all game resources, managing asset loading and memory usage. It interfaces with the files system to load, retrieve, and manage assets used by the Core Engine and Game Logic Layer

The **Utility Layer** offers auxiliary functions and tools that support development, debugging, and general-purpose operations.

- **Common Utilities:** Combines common utilities and debug tools, providing diagnostics, error logging, and general-purpose functionalities that aid in both game execution and development.

Explanation of Conceptual Dependencies

Core Engine Layer ↔ Game Logic Layer

The Core Engine Layer is fundamental and provides execution capabilities required by the Game Logic Layer. Specifically, the Execution Engine interprets scripts that define game object behavior and event logic. The State Manager also maintains the current game state, which the Game Logic Layer needs for event handling and saving/loading.

Core Engine Layer ↔ Media Layer

The Core Engine Layer requires the Media Layer to render the results of script executions and state changes. The Execution Engine and State Manager may trigger changes in the media elements, such as updating graphics or playing sounds based on the game state.

Core Engine Layer ↔ Resource Management Layer

The Core Engine Layer depends on the Resource Management Layer to retrieve assets like scripts and textures, which are essential for the Execution Engine and Kernel operations.

Core Engine Layer ↔ Utility Layer

The Core Engine Layer requires utilities for debugging and game detection capabilities, which are essential in a modular engine that may need to adjust or optimize its execution based on specific game versions or different host platforms.

Game Logic Layer (Event System) ↔ Media Layer

The Game Logic Layer triggers audio and visual changes based on game events or interactions. For instance, when a game object performs an action, it might initiate sound or animation in the Media Layer.

Game Logic Layer (Save/Load System) → Media Layer

The Save/Load System needs to restore or preserve certain media-related states during game save and load operations.

Game Logic Layer ↔ Resource Management Layer

Game logic components, such as the Save/Load System, need to access assets managed by the Resource Management Layer. The Save/Load System requires access to stored states for saving and retrieving game progress.

Game Logic Layer ↔ Utility Layer

The Game Logic Layer also requires utilities for debugging and other supported functionality, especially during event handling and game interaction logic. This could be helpful for tracking game object states and events.

Resource Management Layer ↔ Utility Layer

The Resource Management Layer leverages the Utility Layer's debugging functions to log loading errors. Common utilities can also support file handling and memory management tasks.

5.3 Concrete Architecture ~ Sci Engine

Upon mapping the source files into their respective components with SciTools Understand, the majority of files naturally aligned with the proposed conceptual architecture. However, a few areas required modifications to better fit the actual file structure and functionalities of the system. Below are the main differences from the initial conceptual architecture and the rationale behind each update:

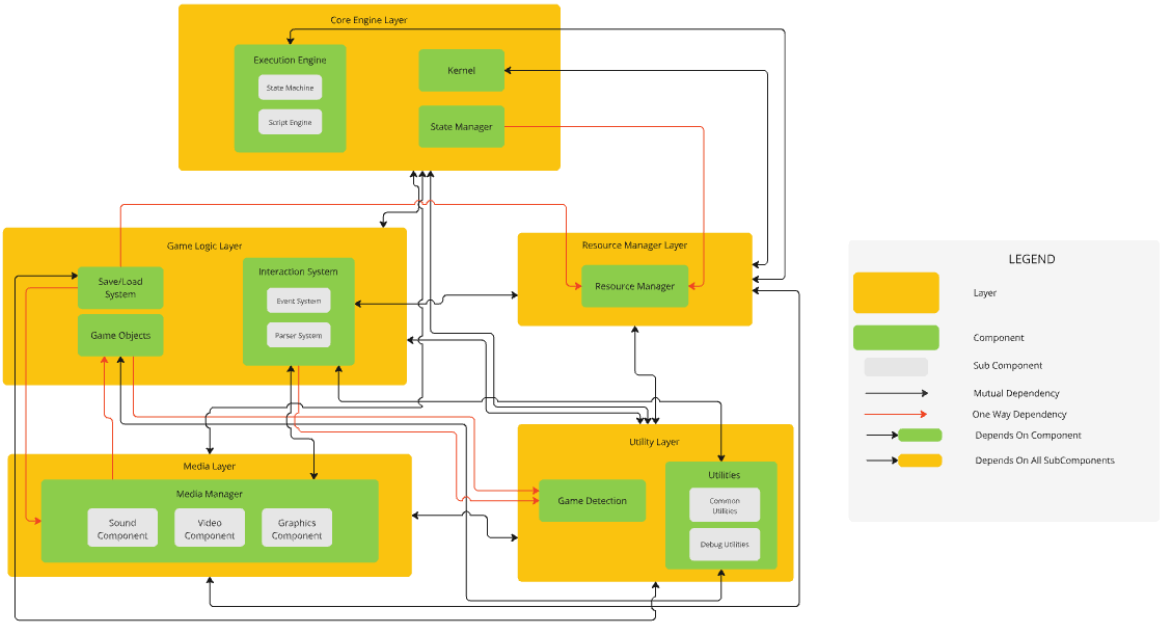


Figure 4: Box and Line Diagram of SCI Engine Concrete Architecture

Core Engine Layer - Execution Engine

The Execution Engine component was enhanced by introducing two subcomponents: The Virtual Machine and Script Engine. This addition reflects the essential roles played by these subcomponents in executing and managing game logic. The Virtual Machine is primarily responsible for interpreting and running game scripts, while the Script Engine manages the overall flow and execution of in-game events. Several files in the codebase mapped to each of these to merit their inclusion as distinct subcomponents under the Execution Engine.

Game Logic Layer - Interaction System

A new Interaction System component was created within the Game Logic Layer. Within the new Interaction System are the subcomponents: Event System and Parser System. The decision to combine these into an Interaction System stems from the need to organize and manage player interactions and game responses as a cohesive unit. The Event System handles game events and interactions, while the Parser System processes and interprets player commands. Since the Parser System did not map naturally into other components and primarily interacts with player commands, grouping it with the Event System under an Interaction System made sense.

Utility Layer - Separate Game Detection and Utilities Components

The Utility Layer was modified by separating the Game Detection and Utilities functionalities into two separate components. During the mapping process, it was evident that Game Detection files were significant to the system's overall functionality, warranting a standalone component. This separation helps clarify the role of Game Detection, which is responsible for identifying the game versions and configurations that are crucial in initializing the correct engine behavior.

The Utilities component was also restructured, gaining two subcomponents: Debug Utilities and Common Utilities. The number of files and specific functionalities related to debugging and common utilities justified this separation. Debug Utilities handles tasks related to diagnostics, performance monitoring, and system stability, while Common Utilities provides general-purpose functions used across various layers.

5.4 Reflexion Analysis ~ Sci Engine

In this reflection analysis, we examine the differences between the conceptual and concrete architectures of the Sci Engine, focusing on notable divergences in component dependencies as revealed by SciTools Understand. This analysis prioritizes missing dependencies between entire layers, as well as discrepancies in expected inter-layer relationships. Rather than detailing every minor variation, this report highlights only key divergences.

1. Media Layer ↔ Utility Layer

Unexpected Dependency: In the conceptual model, a dependency between the Media Layer and Utility Layer was not expected. Typically, media components manage rendering tasks and do not require direct access to utility functions.

Explanation: In the concrete architecture, the Media Layer uses services from the Utility Layer, mainly for debugging, game detection, and other support functions. This shows that media components depend on utilities for checking game-specific conditions and improving performance. This dependence helps with debugging media processes and suggests a tighter coupling with utility functions for handling media-related issues, which wasn't expected in the simpler conceptual model.

2. Media Layer ↔ Resource Management Layer

Unexpected Dependency: Originally, we thought that the Resource Management Layer would handle asset loading, while the Media Layer would focus on rendering assets provided by higher layers (Core Engine or Game Logic). A direct dependency between these layers suggests a deviation from the architectural pattern that we anticipated.

Explanation: The Media Layer in the concrete architecture interacts directly with the Resource Management Layer. For instance, the Media Manager accesses resources such as audio files and patches, through calls to functions like ResourceId and findResource in files such as adlib.cpp and amigamac1.cpp. This indicates that media components dynamically load resources during runtime, bypassing higher layers. This design choice implies that media components require specific resource access for loading, rendering, or playback functions, requiring a direct link to the Resource Management Layer.

3. Differences in Game Logic Layer ↔ Resource Management Layer Interactions

In the conceptual model, a mutual dependency was anticipated between the Game Logic Layer and the Resource Management Layer, with all subcomponents interacting. However, the concrete architecture presents a more nuanced relationship:

Save/Load System Dependency: The Save/Load System was expected to have a two-way dependency with the Resource Management Layer. However, in the concrete model, this relationship is one-directional; the Save/Load System relies on the Resource Management Layer to access resources for saving and loading game states but does not provide information back, nor does it use any of the functionality implemented by the RML.

Interaction System Two-Way Dependency: The Interaction System (comprising of the Event System and Parser System) shows a two-way dependency with the Resource Management Layer in the concrete architecture. This suggests that the Interaction System uses resources for functions like vocabulary lookup (as evidenced by vocabulary-related dependencies), while the Resource Management Layer relies on the Interaction System for real-time updates or commands related to game events. This mutual dependency highlights the dynamic requirements for handling in-game interactions and responding to player inputs.

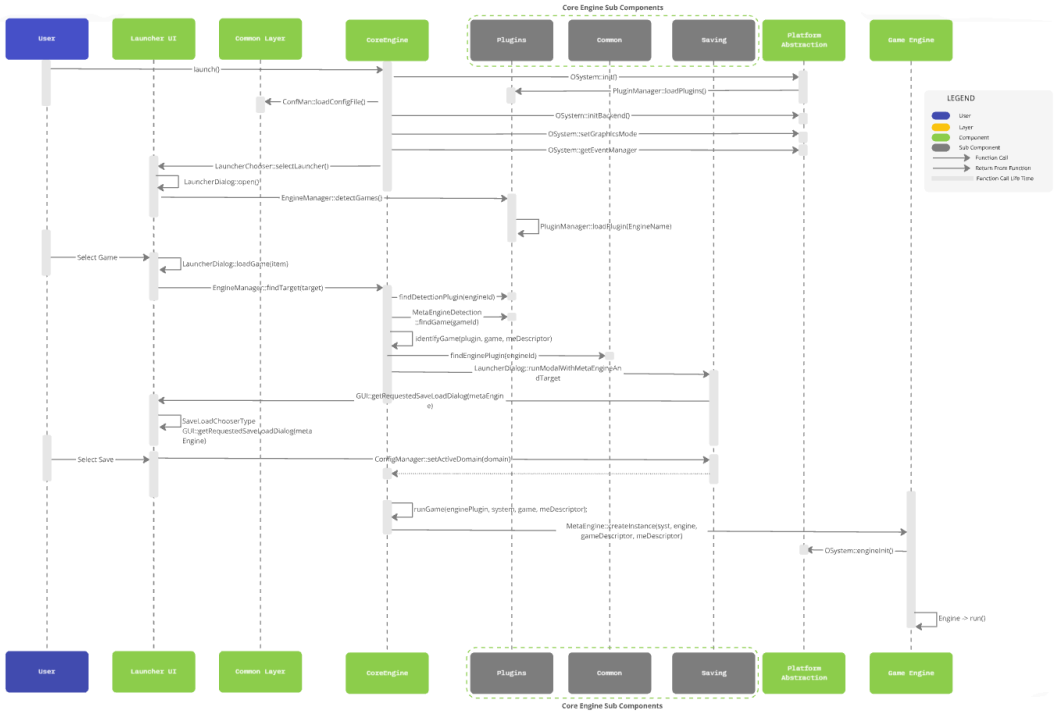
Lack of Dependency for Game Objects: The conceptual architecture presumed that Game Objects would need access to the Resource Management Layer for assets or data. However, no dependency exists in the concrete model, suggesting that Game Objects either pre-load their required resources or do not request assets dynamically, relying instead on other components like the Save/Load System or Interaction System for any required resources.

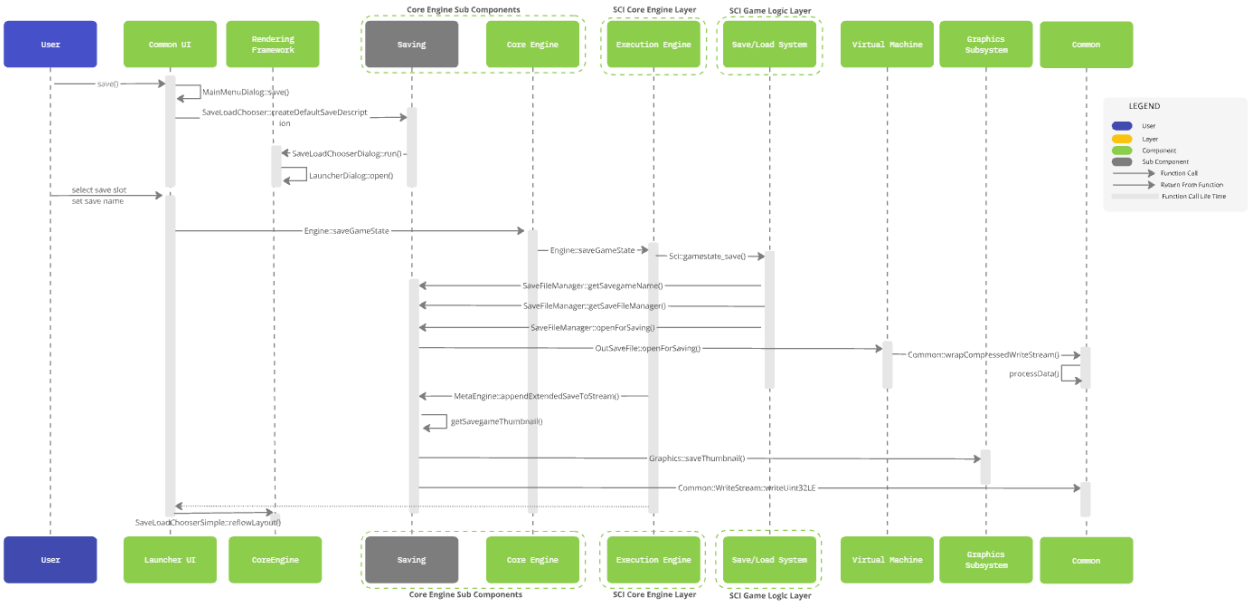
This analysis of the deviations between the conceptual and concrete architects provides insight into the practical dependencies of the Sci Engine, highlighting real-world requirements for component interaction and resource handling.

6. Sequence Diagram

6.1 Use Case 1: Game Launch Sequence

The process begins with the user launching the ScummVM application, which initializes the Core Engine and backend, loads the necessary configuration settings, and launches the Launcher UI. This UI displays available games, allowing the user to select one or load new games from their filesystem. Several other backend dependencies are initiated at this time, such as the Event Component, Graphics Component, and other platform-specific components, all through interaction with the Platform Abstraction Layer. Once a game is selected by the user, the Core Engine identifies it and determines the required Game Engine. The Game Meta Engine is then created along with the activation of any plugins and pre-game resources. This setup process then triggers the SaveLoad dialog, which prompts the user to select a previously saved game state. Upon selection, the game domain is loaded and initialized, and control is returned to the main ScummVM loop, which launches the initialized game engines and the selected title.





6.3 Use Case 2: Game Saving

The game-saving sequence is the process that preserves a player's progress in a game. This process begins when a user requests to save their game through the main menu UI. This executes the main save dialog through the Core Engines Save subcomponent which opens the dialog through the Rendering Framework. Once the user has selected a save slot and given it a

name the dialog returns execution control to the Core Engine, which invokes the saveGameState function for the currently running engine. We will focus on the immplementation within the SCI Engine for our sequence diagram. In the case of the SCI Engine, there is a gamestate_save function that invokes and initializes the main Core Engine SaveFileManger and opens a file for saving. The main saving logic involves serializing and saving the current state of the Execution Engine's State Machine. This is a very complex process that involves saving all relevant game variables and conditions aswell as graphics information about current visuals and game objects. This saved information is then passed to be compressed and written to a file. This happens within the Virtual Machine and the Common Layer. If the process is successful, execution is returned to the main Core Engine Saving subcomponent, where further metadata is saved and stored. This metadata includes crucial information such as the date and time of the save, specific game object information, version details, and various state offsets. This metadata serves an important purpose beyond simple record-keeping. It helps ensure compatibility when loading saves across different versions of ScummVM or between different platforms and is also used when cloud saves occur (out of scope). The metadata save also fetches and saves a game thumbnail using the graphics engine. This image is then tagged with the saved state name and information so that it can be referenced in the UI when saved states are reloaded later. The save state process ends by writing the saved state file to the user's file system through the Device File Subsystem and the Common Layer, showing a success message to the user back in the main menu UI.

7. Data Dictionary

SDL: A cross-platform software development library designed to provide a hardware abstraction layer for computer multimedia hardware components
MIDI: Standard protocol used to communicate music-related data between electronic musical instruments, computers, and other audio devices
API: A set of protocols and tools that allows different software applications to communicate with each other by defining methods and data formats for interaction.

8. Naming Conventions

ScummVM - Script Creation Utility for Maniac Mansion Virtual Machine
SCUMM - Script Creation Utility for Maniac Mansion
SCI - Sierra Creative Interpreter
RMS - Resource Management System
VM - Virtual Machine
PAL - Platform Abstraction Layer
UI - User Interface
GUI - Graphical User Interface
SDL - Simple DirectMedia Layer
I/O - Input / Output

OS - Operating System
MIDI - Musical Instrument Digital Interface
API - Application Programming Interface

9. Conclusions and Lessons Learned

The report outlines and examines the evolution of the ScummVM and SCI engine architectures through various models of their conceptual and concrete architecture. To achieve this, the team employed the Reflexion framework, highlighting changes between the conceptual architecture proposed and the concrete architecture. The concrete models for this project were derived from the Understand software which enables static code analysis to document and deconstruct ScummVM's component relations. Although the initial conceptual architecture was generally accurate, a few key differences were identified such as the relocation of the User Interface Layer to the Game Engine Layer, the Resource Management System integration within the core engine and game engine components, as well as the addition of a shared utilities layer. For the conceptual architecture of the SCI engine, a mock-up was conceived through research of reference models from similar games and online resources. The proposed model was developed as a layered model, with each layer assuming a specific role to enable smooth interaction between components. Upon inspection and analysis, the Sci engine concrete architecture largely resembled the proposed conceptual architecture, with minor adjustments to reflect the practical challenges of the final implementation. The Core Engine Layer was refined with the addition of Virtual Machine and Script Engine subcomponents, a new Interaction system within the Game Logic Layer combined the Event System and Parser System, and the Utility Layer was restructured, dividing the Game Detection and Utilities into separate components.

10. References

Developer central. ScummVM. (n.d.-a). https://wiki.scummvm.org/index.php/Developer_Central
Howto-backends. ScummVM. (n.d.-b). https://wiki.scummvm.org/index.php?title=HOWTO-Backends
Howto-Engines. ScummVM. (n.d.-c). https://wiki.scummvm.org/index.php?title=HOWTO-Engines

Understanding the interface. Understanding the interface - ScummVM Documentation documentation. (n.d.). https://docs.scummvm.org/en/v2.8.0/use_scummvm/the_launcher.html