# Assignment 3:
# Architectural Enhancement for ScummVM

CISC 322/326: Software/Game Architecture

December 2nd, 2024


Group 12

Trevor White - 21tw43@queenus.ca
Sophia Pagazani - 21snp8@queens.ca
Nasreen Mir - 21nsm5@queensu.ca
Aniss Hamouda - 21arh18@queensu.ca
Nicole Hernandez - 21nhm5@queensu.ca
Claire Whelan - 22cmw2@queensu.ca

# Table of Contents

# Abstract

Based on previously established conceptual and concrete architectures for ScummVM and the SCI engine, this report provides a detailed explanation of a proposed enhancement to the program through the addition of speedrunning tools. After explaining the merit of our proposal, we go over a SAAM analysis of the system, exploring two different potential implementations of the enhancement, one with the additional components reliant on the SCI engine's interpreter and one with more functionality in the ScummVM layers rather than the engine. We examine each implementation's impact on the architecture and how they fulfill potential non-functional requirements for our stakeholders, ultimately deciding on a ScummVM-level implementation as superior. We further explore this implementation with two use cases and examine its remaining potential risks and limitations. We conclude with a reflection on the lessons we learned during this project.

# Introduction

ScummVM is an open-source program that allows a user to play classic adventure RPGs on a variety of operating systems, in the face of the extinction or obscurity of many original platforms. While originally designed to specifically support games that were part of the SCUMM engine, the project soon expanded to support many other game engines, as well as more platforms and ports. In our examination of ScummVM's software architecture, we specifically focus on the SCI engine. Having established a conceptual software architecture in our first report and then a revised concrete architecture based on the source code in our second report, we now explore a potential enhancement to ScummVM in the form of speedrunning tools for users interested in ways to practice, record, and track their speedruns within ScummVM.

Starting with a review of the current state of the software, we go on to explain the details of our proposed enhancement. We suggest two different potential implementations, with a detailed description of each implementation's impact on the architecture of ScummVM and SCI, followed by a SAAM analysis of the two implementations' impact on our stakeholders' non-functional requirements such as accuracy, performance, and modifiability. Based on this analysis we conclude that one of the implementations is superior and provide two use case diagrams to support our explanation of the program's new software architecture.

# Current State

ScummVM is an open-source software that runs classic point-and-click games across a variety of platforms such as Windows, Linux, and Mac, along with mobile devices. It works by recreating other game engines, allowing games built with other engines to be played on ScummVM without needing the original hardware. The software is maintained by a community

of developers. While ScummVM has features such as save states, remapping controls, and fast-forward functions for speeding up gameplay, it lacks an explicit speedrun practicing tool.

## Enhancement

Our group's proposed enhancement is to add a speedrun tool to ScummVM, which would allow players to track their best time for a game and rewind to previous checkpoints. The speedrun enhancement would add a timer that can be either manually started/stopped or triggered by a player reaching a predefined in-game checkpoint, which would vary game to game. It tracks players' best times, current progress, and allows practice sessions by starting from specific checkpoints. An in-game window will be part of this feature, providing real-time data of the timer and checkpoint progress. By leveraging ScummVM's overarching framework, the system achieves cross-engine consistency, modifiability, and scalability, making it an accessible and versatile enhancement for the speedrunning community.

## Motivation, Value, and Benefit

The motivation behind the speedrun tools is to encourage a speedrunning community for older games that aren't typically speedrun by providing accessible tools to do so and to bring new users to ScummVM. Speedrunning has become a popular and competitive activity, where players strive to complete a game from start to finish as quickly as possible. Just the platform speedrun.com itself has tracked 5 million speedruns and has over 2 million registered users (Speedrun.com, 2024). According to an article by YouTube, in 2021, there were more than 1.2 billion views of speedrunning videos (Youtube.com, 2022). There is a very large, proven market for speedrunning, and we are confident the introduction of a speedrunning tool to ScummVM would help to popularize the software even more. By introducing tools tailored to the art of speedrunning, ScummVM empowers players with precise timing, checkpoint tracking, and easy-to-use interfaces, making speedrunning accessible and enjoyable for retro games. This enhancement brings value by enabling a new way to engage with classic games and creating a new community. It also improves the gameplay experience by offering features like checkpoint practice and personal record tracking, which encourage players to spend longer playing games by motivating them to master it. For casual players, the feature is non-intrusive and maintains the integrity of traditional gameplay, ensuring ScummVM remains welcoming to all users.

# Potential Architecture

## 1. Alternative 1: Engine-Independent

In this implementation of the speedrun feature, the new and affected components are mostly within the ScummVM portion of the architecture rather than in the SCI engine. This approach adds a new layer within ScummVM, so the layered architectural style for ScummVM stays the same, and the interpreter style for SCI is not impacted.

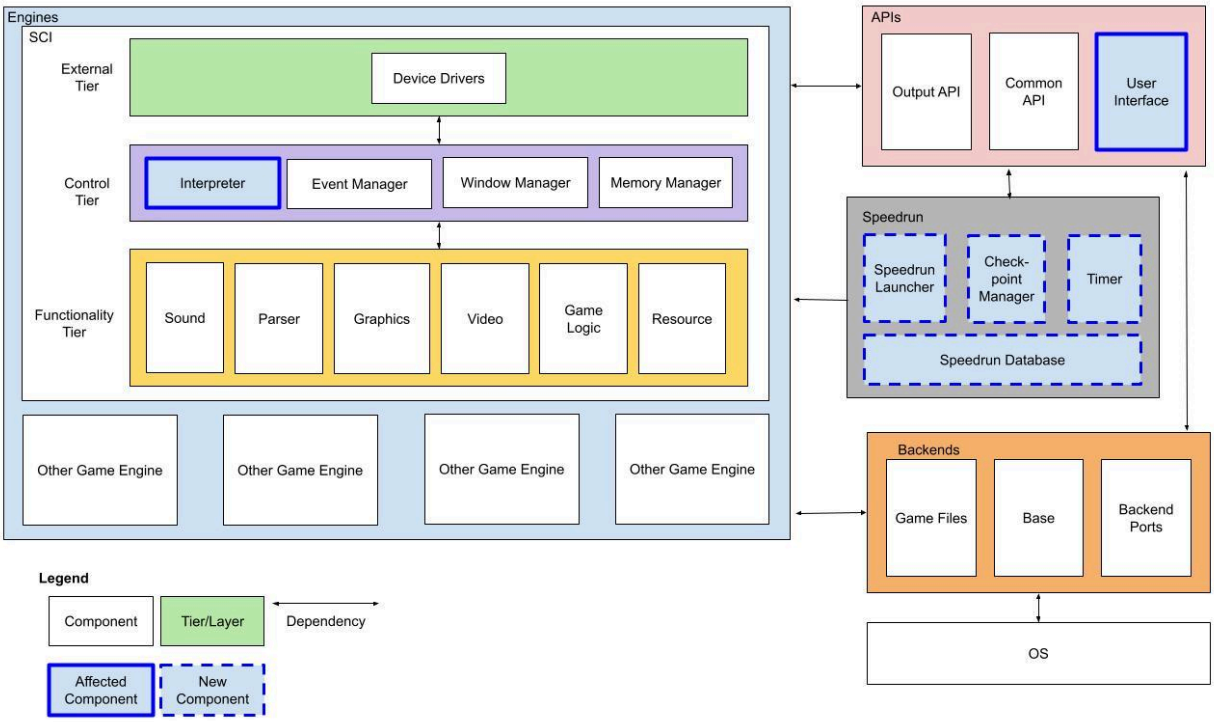


Figure 1: *Conceptual architecture of alternative 1*

As seen in Figure 1, within the new Speedrun layer, there are the Speedrun Launcher, Checkpoint Manager, Timer, and Speedrun Database components. As the name suggests, the **Speedrun Launcher** manages the launch and functionality of the speedrun feature when a flag is set from the UI. The **Timer** is the component that keeps track of how long a game is running from a start point and is activated by the Speedrun Launcher. The **Checkpoint Manager** relies on the Timer to determine the current time and the timing of checkpoints. It then sends these times to the UI so they can be displayed and saves these times in the **Speedrun Database** so that they can then be compared to other runs. The Checkpoint Manager is also the component that interacts with the Interpreter so it can save and load the checkpointed states. The Speedrun Database is also where the configuration files that store information about each game and checkpoints are located. This component is also where the permanent saves for each checkpoint are saved, instead of saving them volatilely with the Memory Manager.

These changes and additions would make it so that the Speedrun layer interacts with the Engine and API layers. There is a two-way dependency between APIs and the Speedrun layer, as

the **UI** in the API layer is what activates the speedrun mode, and then the speedrun layer sends updates to the GUI within the UI component. The speedrun layer is dependent on the engine **Interpreter** component for the saving and loading of checkpoints.

With the new feature, relevant directories for the UI component would be the `gui/` folder, and relevant files for the Interpreter component would include files that are relevant to game saving and game detection. UI also handles the GUI of ScummVM, so there would be quite a bit added to implement the speedrun timer visuals. This would mean some files being added into the `widget/` folder within the `gui/` folder so the Speedrun feature can have its own GUI when the game is running. As for the Interpreter, files such as `game.h` and `savegame.cpp` would be used to help save the game and detect the game, respectively. The game detection would be used to help to know what game is playing so that information in the configuration file and checkpoints are properly documented and loaded.

Since the existing components change very slightly, or are already pre-existing features used with few modifications, the architectural style that we established in the previous reports of **layered and interpreter styles** has not changed. The only difference with this new architecture is that there is a new layer for the Speedrun components in ScummVM's layers.

## 2. Alternative 2: Engine-Integrated

The second approach considered for the enhancement is to integrate the new functionality more closely with the SCI engine. The idea is to leverage existing components that perform similar tasks as the desired additions. A corresponding box-and-line diagram appears in Figure 2.

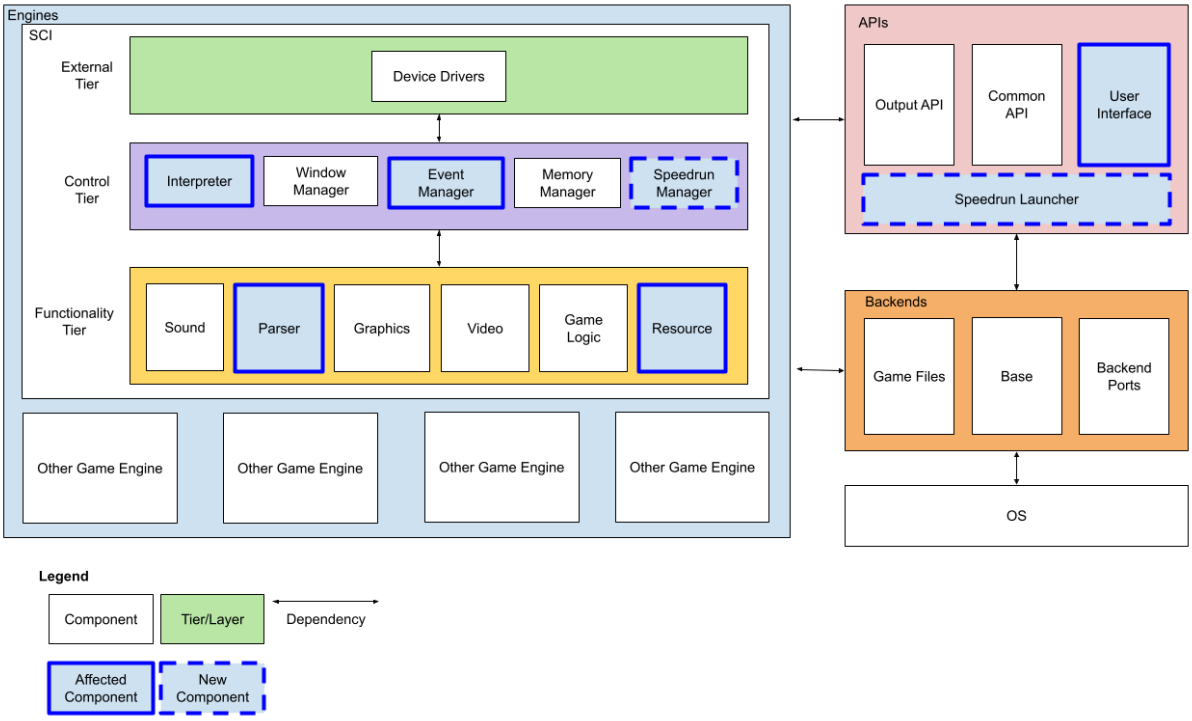


*Figure 2: Conceptual architecture of alternative 2*

The **Speedrun Launcher** component also appears in this alternative and is functionally the same, providing the logic that enables initiating a "speedrun" mode with the enhanced features. It is external to SCI, as it is only needed before the engine is loaded. In terms of interactions, the User Interface activates it via a flag. Then, it sets up the new Speedrun Manager component once the Base component has instantiated the engine.

Accordingly, the **User Interface** in the API layer includes the same new UI elements as before: a timer, a button to pause it, and on-screen checkpoint tracking. Nevertheless, it interacts a bit differently. It subscribes to the Event Manager to get the information to display, notably the timestamps when passing a checkpoint. It also communicates with the Speedrun Manager to pass control information, including requests to revert to a previous checkpoint save state.

The **Speedrun Manager** is a major controller of this enhancement. It runs concurrently to other subsystems, continuously monitoring the state of the game, automatically querying the Interpreter to get timestamps, and creating save states via the Memory Manager. This interaction is also how it reloads states on demand. By relying on the Parser, it reads checkpoint configuration data from the Resource component needed to recognize these target states.

Evidently, many existing components require expansions. The `Resource` class defined by **Resource** must include a format to list recognizable and repeatable game states, and the **Parser** must have methods to process it. The same applies for records of best times. The **Interpreter** requires a method to externally query its clock tick, and the **Event Manager** will define a new event type for transferring timestamps.

In terms of architectural style, this alternative induces modifications exclusively on the component level. In fact, this proposed method does not add any new tiers to the **semi-layered style**. Furthermore, the **publish-subscribe** methodology involved during the Event Manager's transactions is extended to signal UI checkpoint tracking for quick and decoupled responses. Most of all, it relies heavily on the **interpreter style** used in SCI, since this provides a system to track game state at any given point in time. The DSLs provided by SCI, a key trait of this style, may also simplify implementation as many of the new features reside within the engine.

To realize this strategy, the main repository for ScummVM will require a subdirectory to contain the Speedrun Launcher files. The content under `gui/` will need to be updated as well to include the new GUI features. On the SCI side, the `engines/sci/resource` subdirectory, in particular the `resource.cpp`, will be updated as it defines the `Resource` class. Similarly, the `engines/sci/parser/` directory should be updated to include new parsing functions, most likely in `kparse.cpp`. The entire `engines/sci` is where all other affected features reside, such as the event management in `event.cpp`, so it will also be affected.

# SEI SAAM Architecture Analysis

## 1. Identified Stakeholders

To begin our SEI SAAM architectural analysis, we need to identify the stakeholders that would hold interest in our enhancement, no matter the method implemented. To start, we can define *Speedrunners* as our primary stakeholders. These are players that attempt to complete games with the fastest time within a certain category with specific objectives. They could be casual players who enjoy completing speedruns as a hobby or as a way to compete/bond with their peers. However, we could also see more professional players (such as streamers or e-sports players) with their interest being centred around opening up more fields to competition in an accessible way.

Our second biggest group of stakeholders would be *Casual Players*. These players may have little to no interest in actually using the speedrunning features but wish to ensure that their gameplay experience in SCI, as well as their interaction with ScummVM, is not changed in any significant way.

Our final stakeholders would be any *Contributors* to the development of ScummVM and the SCI engine. This enhancement would open up a new avenue for them to contribute, most likely in the form of creating various speedrunning configurations for various games and game engines supported by ScummVM.

## 2. Associated Non-Functional Requirements

The most relevant non-functional requirements (NFRs) for *Speedrunners* would have to be accuracy and usability. For the former, the stakeholders are going to want to see that the timer is accurate to the millisecond. If something goes wrong there, then their record is not true to actuality and negates all effort contributed. For usability, they are looking for ease-of-use functionality, with the tools being easy to turn on and use (i.e., the timer display can be readily moved, paused, restarted, and rewound). Two other important NFRs for this category of stakeholders would be performance—the game should be able to handle the new features without any latency being experienced (and timer buttons having an optimal response time)—and scalability—software performance should remain stable when it handles, for example, five checkpoints as opposed to one hundred checkpoints.

For *Casual Players*, the most important NFR that they will value is performance. No latency should be caused by the addition of the new features, and as such, they should be able to play games with no changes to their experience. This is closely followed by usability for the same reasons. The GUI for speedrunning shouldn't make the navigation of the system any more difficult and thus should be hidden by default. Finally, the portability of the system should be maintained, with the enhancement not causing any major changes to the highly portable nature of ScummVM.

Our final group of stakeholders, *Contributors*, want to see strong modifiability in the system. In other words, they are looking for how easy it is to modify checkpoints in the system

or, on a bigger scale, how easy it is to implement a speedrunning system across multiple different engines.

# 3. Enhancement Impacts on Non-Functional Requirements

## 3.1 Alternative 1: Engine-Independent

Starting, once again, with our *Speedrunners*, this implementation supports most of its NFRs. Accuracy performs the best by far because its timer is created outside of the engine, so we see consistent cross-engine times, and there is by far more flexibility in how it works (allowing for a more time-accurate response). Usability is double-edged, where we are seeing very consistent UI across engines with the drawback of game-specific objects causing inconsistencies in measuring the same categories of speedruns due to their variability. As this method operates outside the engine, we may see a small drop in performance because the engine has increased communication with ScummVM, which increases the chance of latency. This point may be negligible since there's already a fair amount of talking between the components, and the system is performing very well. As for scalability, there could also be latency produced here, as increased checkpoints cause increased communication between the systems, as with performance.

*Casual Players* fare very well, with little to no drawbacks for their associated NFRs. When deactivated, any communication to the new Speedrun component is non-existent; the system is essentially unchanged, and thus, performance remains as it was before this enhancement was considered. The same goes for portability, as the new component (and any of the affected ones) should not affect the system foundation or anything that makes it portable. Finally, with usability, it is very easy to have any speedrunning UI disabled before even launching a game.

Modifiability for *Contributors* is a lot easier with this method of implementation due to most of the important components being in ScummVM, allowing for it to be adapted for other engines more easily. This, however, is still not able to be standardized or automated, relying on manual effort for *Contributors*, which may not be realistic in every scenario.

## 3.2 Alternative 2: Engine-Integrated

With *Speedrunners* relying on the accuracy of the system to be able to complete their runs, this approach is less than ideal since the speedrun timer relies on the internal Interpreter timer, which may not be exact since that is not its original purpose and has no real need to be (meaning it can just run on its own timing method). On the other hand, since checkpoints are internal, the system would be faster in recognizing when a checkpoint has been hit and will be able to more accurately log the time to the second. Usability is also affected in a negative way—it is significantly more inconvenient to use, as the SCI engine needs to be launched before a user can see their saved results or generally access any of the features. UI updates have to go through SCI first, then out to ScummVM, so response times for stopping the timer may be

increased, affecting performance. Alternatively, reloading to previous states might be quicker since everything in that area is handled within the already running engine. In regards to scalability, increasing the amount of checkpoints takes away some of the interpreter/engine's focus, and thus, the system as a whole can be affected by the increased load.

Performance for *Casual Players* may be impacted due to resources for the speedrun mode taking up memory space, potentially increasing the load on some caches, which then impacts the overall responsiveness of the system, even when the tools aren't being used. These changes don't really affect portability, as for much the same reasons as Alternative 1. In terms of usability, since changes are localized to the engine, impacts to *Casual Players* will vary. However, if users are running engines that have not implemented speedrunning tools, there will be no potential impacts. As for SCI, the UI should still be toggled, but the user may have to go through more steps to change it.

Contributors see no benefit to this method, with modifiability taking a heavy hit. This implementation is much harder to modify and replicate since it is specific to the engine, in this case SCI. The *Contributors* would have to create a completely different implementation from scratch for any other engine and then connect that to ScummVM. That said, while it is theoretically less ideal, this risks being a bit more realistic, as it may be impossible to design a format that is universal across all games and engines.

### 4. Final Enhancement Method

Taking into consideration both alternatives, we are seeing an overall net positive impact for Alternative 1: Engine-Independent. Not only is it theoretically much easier to adapt for more engines, but each of the stakeholders is seeing a more positive outcome in regards to their interests. Performance is mostly maintained, with usability and accuracy being held to higher standards. *Casual Players* will get to experience their games as they were before, while people interested in the new enhancement, our *Speedrunners*, can get what they need out of the feature in similar ways to most mainstream software of the same use. *Contributors* are able to easily develop new changes and features to both the enhancement and the system without either of them getting in the way. We will proceed to consider only our engine-independent implementation as it aligns most wholly with the stakeholders interests.

## Use Cases

### 1. Reloading a Checkpoint State

With the selected method, the use case in which a player returns to a previous checkpoint state to practice a section of a game is represented in Figure 3. Note that the functions shown are conceptual at this stage. The use case begins when a player presses on a GUI element signalling the desire to rewind. This informs the Checkpoint Manager of the request, which begins the process. First, it fetches the data relating to the target game state from the Speedrun Database,

having to process the stored format into one that is usable. It also reverts the timer to its value when the target state was first reached. Similarly, it must then revert all the other game elements accordingly, such as the graphical positioning of game objects and timing of the music. Since the Interpreter is involved in managing the current game state, this occurs in a loop via the engine until everything is in place. Finally, the Checkpoint Manager can update the GUI to reflect the rolled-back time and return control back to the user to continue gameplay.

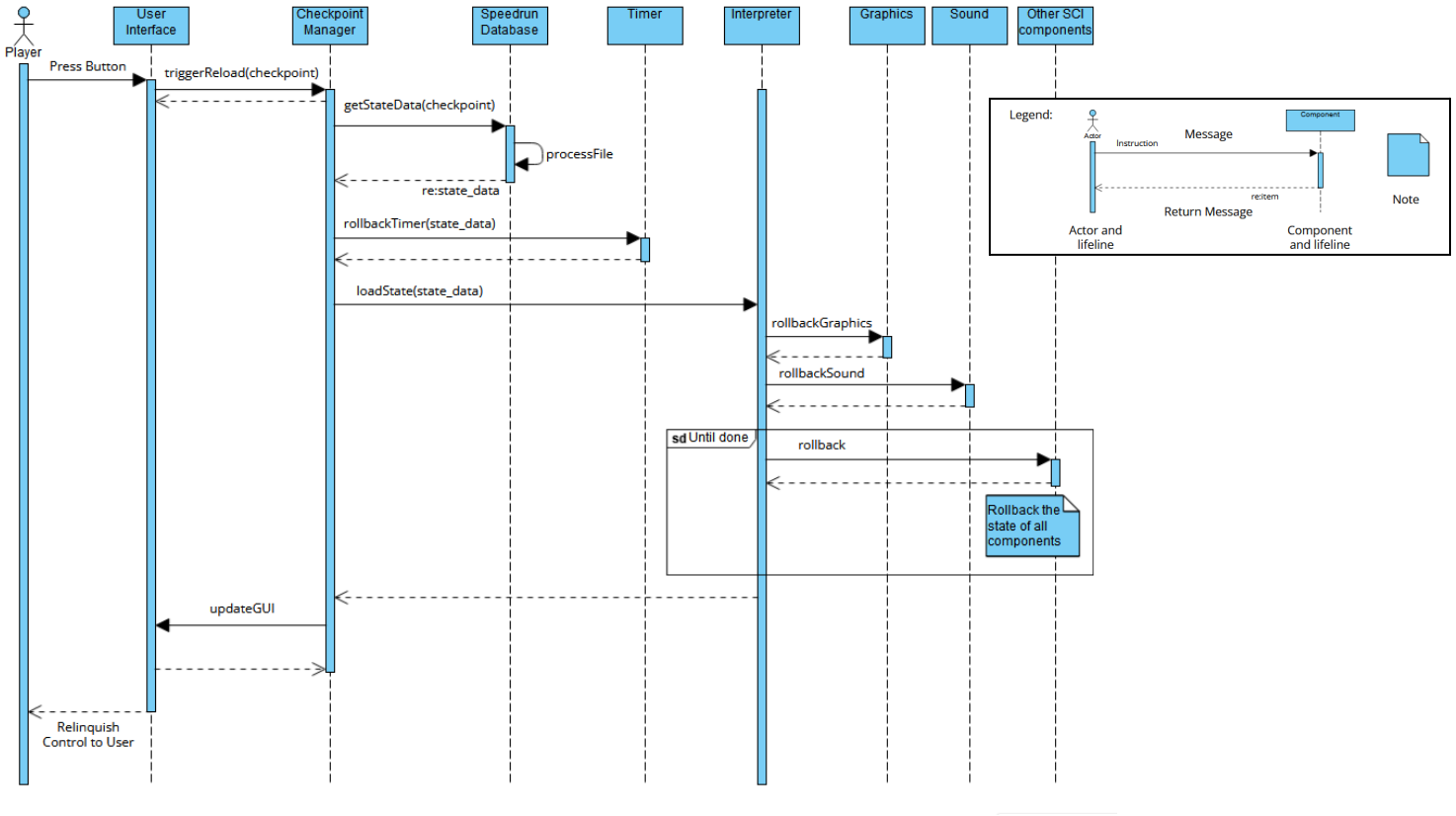


*Figure 3: Sequence diagram for reloading a checkpoint state*

## 2. Starting a Run

Next, Figure 4 represents a sequence diagram of a speedrun starting, assuming the game is already selected. First, the player must enable speedrun mode in the User Interface, which will enable the Speedrun Launcher. The player must then select which checkpoint configuration file they want in the User Interface. The User Interface will pass this to the Speedrun Launcher, and if speedrun mode is enabled, the Speedrun Launcher will call the Checkpoint Manager to load that file. The Checkpoint Manager will then open the Speedrun Database to the correct file to record times. When the user finally starts the game in the User Interface, it will call the Interpreter to begin its boot sequence, which is not entirely shown for brevity. Once the Interpreter completes the sequence, it will inform the Checkpoint Manager. The Checkpoint manager will pass this information to the Speedrun Launcher, which will initiate the Timer. For

the rest of the gameplay, the Checkpoint Manager will be tracking the time with the Timer and displaying it in the User Interface. It will also be monitoring the game state through the Interpreter, and once the game state reaches one of the checkpoints specified in the config file, it will record that time in the Speedrun Database.

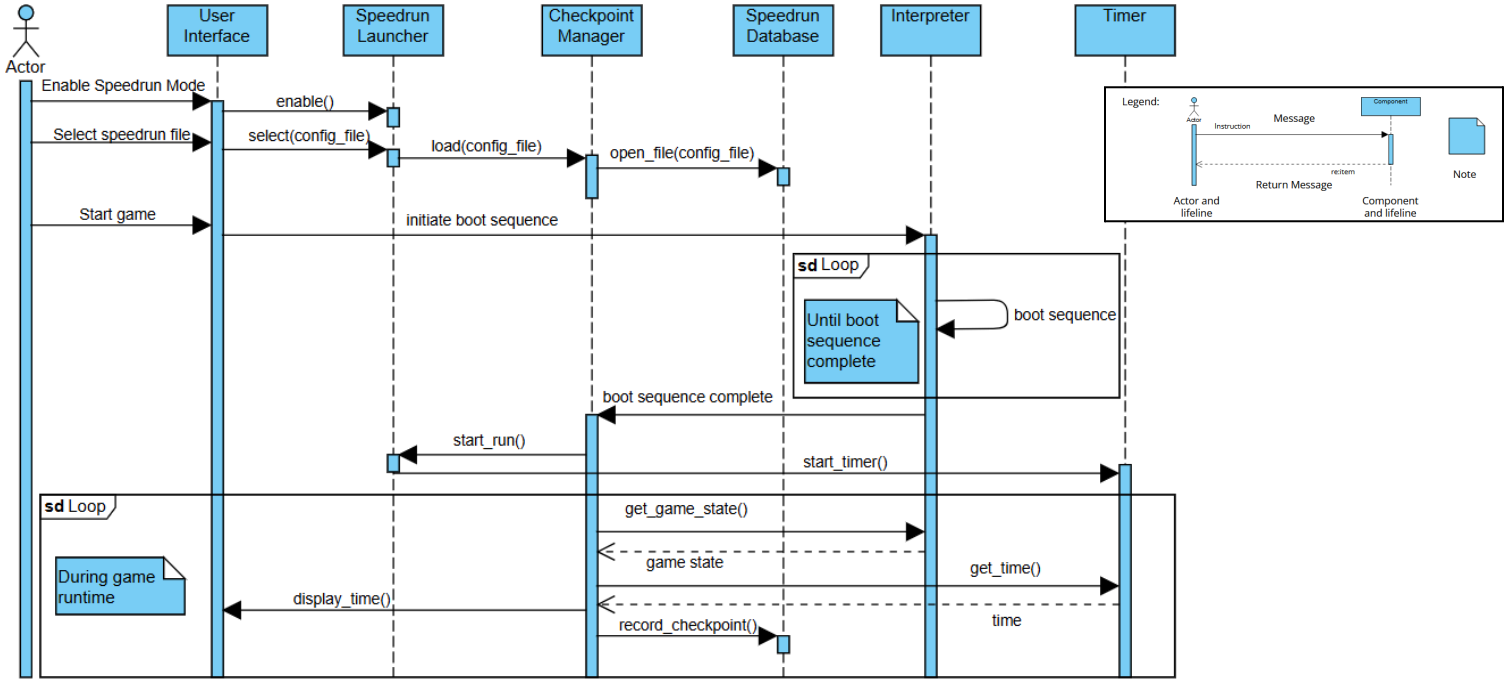


*Figure 4: Sequence diagram for launching a speedrun*

## Potential Risks and Limitations

A significant limitation of our proposed enhancement is that, although we have decided on an implementation whose core functionality will be in ScummVM, it will still rely on access to a game engine's interpreter and the existence of specifically formatted configuration files with checkpoints, all of which needs to be implemented on a case-by-case basis for each game engine. On top of limiting the immediate usability of our enhancement, this creates a new area with potential risks to the effectiveness of our enhancement resulting from inconsistent or incompatible implementations. For example, our model relies on an interpreter in the style of the SCI engine to return information about the game state in order to check whether a checkpoint has been hit. Different implementations of interpreters could have inconsistencies that could result in differing return times for the users. Additionally, some engines could have implementations that do not match the interpreter style assumed on the ScummVM side and could require extra work to implement the speedrunning tools, affecting modifiability. Other risks affecting all users include the potential that additional components, such as large configuration files or a new timer in the ScummVM layer, degrade latency.

## Plan for Testing

As our enhancement involves a base in ScummVM on top of which different game engines will have to design their own interaction, there will need to be tests for each game engine implementation to determine whether it works, and if it does not, whether the issues are coming from the ScummVM side or the engine side. Preferably, each feature will be tested as it is added to the program. In order to test the entire program once completed, we would create a thorough list of use cases related to each new component to ensure all features work as intended. One area for testing engine interaction would be the practice mode. At the time of creating the configuration files to load checkpoints, it will be necessary to test that all checkpoints can be loaded correctly. We will also do tests related to the checkpoint saves to confirm that the system can reasonably handle the required number of checkpoints.

## Maintainability, Evolvability, Testability, and Performance

As we decided to use method one, we changed very little within the system. We added a new architectural layer that is only used when speedrun mode is enabled. Outside of this, there was only a slight change to the User Interface component. This means that the evolvability of the system will be the same because speedrun mode can just be disabled, and then the system will be the same as without the change. If the system wants to make changes that also work with the speedrun mode, it may require more work in order to make those changes function with the new layer, but it should not be too complicated, assuming the User Interface and Interpreter are not drastically changed. The only exception is if a new engine is being added. Our enhancement works assuming the engine has a component with the ability to record the game state, so any new engine would need a component with those capabilities to work with the speedrun functions.

Maintainability and testability are both affected by our enhancement. The enhancement will add a new layer to the architecture, making it more work to test and maintain the system, but it will not be too complex, so the impact is not very pronounced. ScummVM is already a very large program, so the speedrun enhancements will not contribute much to the testability and maintainability of the program.

The performance of the system may be negatively affected by our enhancement. The program will be bigger due to the new code and thus risks running slower. In making our architecture, we did not try to improve the system's performance, so the only change will be that there are extra components trying to function. Our changes will not require a large amount of computing power, so most modern machines should have a negligible performance decrease, but an older machine may run the system slower as a result of our changes.

# Conclusions

During our discussion of a possible enhancement to ScummVM and SCI, we considered many things, such as accessibility and AI integration, but eventually settled on speedrunning tools after considering how it would affect our architecture. With two paths, one focused on engine integration and the other on engine independence, we chose the latter after completing our SEI SAAM analysis. We identified our stakeholders, *Speedrunners, Casual Players*, and *Contributors*, and what interests they would have in the enhancement. Our chosen method appeared to appeal to most of the associated NFRs for each stakeholder while also being a bit cleaner in implementation, which both lead to our final decision. That being said, there are still drawbacks to our method, like inconsistencies between different engine implementations, a chance of increased latency, and a potential infeasibility. All in all, we see our enhancement as overall positive and have thoroughly thought about alternatives, both in ideas and in methods. This experience has continued to shape our understanding of the software architecture of ScummVM and SCI and how to think of other systems going forward.

# Lessons Learned

Throughout the process, our group learned how much effort it takes to design features for software. However, teamwork, visualization, and analyzing what NFRs are important to stakeholders made the design process smoother. Initially, we thought it would be nearly impossible to come up with a new feature to enhance ScummVM, as it already supports a very wide range of functionalities. However, taking some time to think and hold a meeting to discuss possible enhancements, we were able to brainstorm more ideas and develop them to the point where we could see which was the most promising. Analyzing the required NFRs and how stakeholders would interact with the feature helped us decide how we should approach implementing the feature and allowed us to decide what components would be most important in the architecture. Finally, making the box and line diagram before writing the report helped us think more about how the feature would be implemented better and how dependencies would arise. These approaches allowed us to effectively design a new feature for ScummVM.

## Data Dictionary

| Word | Definition/Description |
|---|---|
| Point-and-click game | A video game controlled uniquely by clicking on various objects using a mouse. |

## Naming Conventions

| Acronym | Definition/Description |
|---|---|
| API | Application Programming Interface |
| DSL | Domain-Specific Language |
| GUI | Graphical User Interface |
| NFR | Non-Functional Requirement |
| SEI SAAM | Software Engineering Institute Software Architecture Analysis Method |
| SCI | Script Code Interpreter, or Sierra's Creative Interpreter (ScummVM Wiki, 2023) |
| ScummVM | Script Creation Utility for Maniac Mansion Virtual Machine |

## References


ScummVM. (n.d). *scummvm/scummvm*. GitHub. https://github.com/scummvm/scummvm

ScummVM API documentation: Scummvm API reference. (n.d.). *ScummVM API documentation: ScummVM API reference*. https://doxygen.scummvm.org/

ScummVM Wiki. (2023, April 17). *SCI*. https://wiki.scummvm.org/index.php?title=SCI

Speedrun. (n. d.). *Speedrun.* https://www.speedrun.com/about

Youtube. (n. d.). *Youtube.* https://www.youtube.com/trends/articles/community-spotlight-speedrunning/