Software Design Document (SDD)

Team USA
Software Engineering
Sam Houston State University

October 28, 2015

Contents

1 Software System Description & General Introduction

The software system subject to design and analysis, and referenced throughout this document as "the system", or variations thereof, is a two dimensional, point-and-click adventure game. The user takes on the role of a computer science major at Sam Houston State University as they attempt to submit their final lab for Dr. David Burris's Data Structures & Algorithms course, before the looming deadline. The user will attempt to navigate through various obstacles to complete this objective, while avoiding various traps, hostile entities, and the darkness itself. The game environment consists of 3 levels, occurring on the various floors of Academic Building 1, and in certain other areas on the SHSU campus. Each level consists of individual scenes, containing various entities, collectible items, and textures. The player will navigate through these scenes as they progress through the game.

The system has been developed using the design techniques in the following sections, and each technique has been evaluated based on the applicable metrics. **The chosen design methodology for the implementation of the system is object oriented design, using the unified modeling language, or UML.**[1] Appendix A contains all of the applicable design diagrams, while Appendix B contains all of the applicable storyboard and level progression diagrams. A full sized level progression diagram can be found attached as a supplement to the scaled diagram inside this SDD.

2 Design Features & Specifics

2.1 Three-Tier

The software system itself utilizes a three-tier design, in which there is a separate presentation element for the user to interact with, a separate data processing element, and a data parsing element that will be implemented for persistence between user sessions.

2.1.1 Presentation Element

The presentation element for the system consists of the video and audio managers. These receive and handle audio and video requests from the engine module. The application window itself is a standard window as defined by the host computer's operating system, but is abstracted through the use of the SDL2 library. Access to the audio card is also an abstraction provided by SDL2.

2.1.2 Data Processing Element

The data processing element for the system consists of the engine module and its subordinate transform modules. The engine receives data from the level file and user input. The level file specifies which resources need to be loaded and their associated resource identifiers. The engine can then execute the main loop of the system, which tracks player position versus actor position, items the player has collected, and the player's progression through the game itself. It directs the video and audio manager modules to display the correct and relevant audio and video content to the player via the application window, which comprises the presentation element.

2.1.3 Data Parsing Elements

The system makes use of a two simple file types: save and level files.

Save files consist of the player's inventory, a level ID, and a scene ID. The engine ensures the player will be able to resume this progress from the last checkpoint that was reached by making requests to the

[1]A discussion of why this choice was made occurs in section 8.

GameSaveSerializer module for loading or saving data as necessary. The save function will occur in a separate thread to ensure gameplay is not interrupted while the player progress is automatically recorded.

Level files contain a listing of the necessary resources/assets that will be utilized throughout a set of scenes. The Level module will parse a level file and load the resources specified inside of it during its initialization phase; actors and scenes will be stored inside of the Level module itself, while audio and video resources will be loaded and stored by their respective modules.

2.2 Multithreading, Deadlocks, and Race Conditions

The system implements multithreading in two modules: game saving and audio streaming. Since the system must poll each subsystem during every frame, there are very few instances where multithreading is necessary. Unnecessary multiprocessing was explicitly avoided in the design to reduce complexity and avoid possible deadlock scenarios. It is common practice in the gaming industry to have only one main thread perform most of the work in the program.

It should be noted that fully multithreaded models were considered during the design process (where video, audio, input, and game logic were each in their own thread). It was decided that the extra performance gained from multithreading did not outweigh the benefits of a less complex design. The chosen design allows for simpler maintenance and modification in the future, while avoiding unnecessarily complex relationships between modules.

2.2.1 Streaming Audio

The AudioEngine module allows the user to load a single WAV file to stream audio from disk. This audio will play in a loop continually until the next call to stream a file is made. If audio is already streaming and another call to stream audio is made, then the old thread will be stopped and the new stream will take its place. Race conditions cannot occur because only one audio stream can exist at a time and the system cannot communicate with the audio thread once spawned (other than instructing it to halt). Deadlock cannot occur because the audio stream is never polled for data.

2.2.2 Game Saving

The GameSaveSerializer module allows the user to save and load game progress. The load function is called in a non-threaded context, ensuring that the program cannot progress until the data is retrieved. When the save function is called, a new thread will be spawned and will write the player's progress to disk. This will allow the player's progress to be saved in the background while the rest of the gameplay continues without interruption. The save function will access the save file with options to lock the file for read and write, preventing multiple writes from occuring at the same time. Further, a mutex will be utilized to ensure that the save and load functions are not called while a save function is currently in progress, preventing possible race conditions. Deadlock cannot occur because no thread is polling another thread for data.

2.3 Data Organization

The system does not save large amounts of data. Therefore, integrating a database would have been a poor design decision that would create unnecessary complexity. Instead, a small save file format will be utilized for persisting player data.

The save file will store three save slots that will allow the user to store their progress and later resume gameplay. The data will be stored in a binary format to prevent the average player from tampering with it. The data for a single save will be structured as follows:

```
<int level><int scene><int number of inventory items><iteration of int inventory items...>
```

2.4 Optimization

Due to the time constraint of rendering at a consistently high frame rate, most games consider optimization of space to be secondary to optimization of time. However, the game must also store very large resources, like video and audio. This time/space trade-off is a balancing act that should require great care to not over-optimize in either direction, as a wild swing in either direction can make the gameplay a bad experience; too much optimization for time will increase the number of loading zones, which interrupt gameplay, while too much optimization for space will lower the framerate and cause the gameplay to stutter. Therefore, if the system is sufficiently performant after its design, it is best to leave it alone.

2.4.1 Optimizing for Time

The system will exhibit high cohesion and low coupling after the implementation phase due to object oriented design. This will ensure that each module can be separately optimized without affecting other modules. Inline functions may be utilized for getter and setter methods (which are called frequently) to ensure that unnecessary stack frames are not created in the main loop.

The linkage of subprograms can be performed in the following order to ensure best performance: GameEngine, Player, BaseActor (followed by the various actors), AudioEngine, VideoEngine, libSDL2. The game loop always begins with a call to SDL2 for mouse input in GameEngine and ends with a call to SDL2 for audio and video in their respective engines; this means we can ensure that the program loads SDL2 in memory at the end of the loop and will be able to utilize SDL2 again at the start of the next looping cycle.

2.4.2 Optimizing for Space

High cohesion and low coupling will allow the system to be optimized for memory usage in the future if necessary. As mentioned in Optimization for Time, the linkage between modules can be set in a particular order to reduce the number of page faults significantly.

Images that are to be displayed on screen will be transformed into textures in main memory and then stored in graphics memory during the system's loading phase. This ensures that main memory can free the images immediately after transforming them, allowing the space to be used for other tasks. To further decrease the usage of main memory, the project design forces resources that are not being utilized in a specific level to be released from memory unless explicitly marked as a core component for all levels.

2.5 Module Reuse

In order to save on development time for future projects, ideally a system should have reusable modules. Regarding this system, the audio and video modules are reusable. They are black boxes that accept .WAV and .PNG files and perform a specific action on those inputs. For the video module, it accepts .PNG files and displays them, while the audio module accepts .WAV files and plays them as sound from attached speakers. This black box design also allows the underlying video and audio functionality provided by SDL2 to be replaced with a different audio/video library without affecting other parts of the system.

The levels, art assets, and story of the system will all be defined by the level files. This allows the engine to be reused for other point-and-click games with no modification to the existing codebase; the only modified resource will be ASCII text-files.

2.6 Implementation Plan

2.7 Final Product Testing

2.8 Acceptance Criteria

The acceptance criteria for the system defined metrics which must be met for the system to be acceptable for general use. They are:

- Game textures that consume less than 512MB of video memory.
- A consistent 60 frames per second on reasonably new graphics cards.
- Persistence, implemented through the file system.
- A fully usable application interface, in the form of a window that accepts mouse input consisting of cursor movements and button actuations.

3 Design Analysis Metrics

3.1 Introduction

The following analysis metrics are the criteria that are used to evaluate the quality and effectiveness of a given design paradigm. Each design may use some or all of the presented metrics, and some metrics may apply to only one or two design methodologies. Thus, each design will clearly indicate the metrics used for analysis of that particular methodology at the beginning of the applicable section.

3.2 Metric Definitions

3.2.1 Coupling

Data Coupling Data coupling occurs when one module depends on the output of another, and the data they exchange is composed of discrete, elementary units, providing exactly what each module needs and nothing more or less. This is the lowest form of coupling with the least dependency between coupled modules.

Control Coupling Control coupling occurs when one module's output is used to influence or determine the execution logic of a subsequent module - for example, a boolean value or control flag is passed from one module into another that performs a different action depending on the value it receives. This type of coupling is detrimental to the software system because it implies that each of the coupled modules must know something about what is contained in the other, meaning one or both may not be a true black box, which indicates that one or both may not be separately implementable or maintainable.

3.2.2 Miller's Law

Miller's law tells us that, on average, the largest number of tasks that can be simultaneously held in memory is 7 ± 2. Thus, we should work for a design wherein each module complies with this law and its value. Using dynamic data flow analysis, we can see how the data flows into and out of each module, and with the hierarchical model, how control is implied to flow between modules as well. A good design will show, on average, each module having 4 to 5 tasks, and not more than 7 ± 2.[2]

[2]See softEng1.docx, page 36, by David Burris for more information on Miller's Law.

3.2.3 Graciunas's Law

Graciunas's law concerns itself with the number of control relations between modules, and predicts how this impacts the complexity of the software system. For a given set of modules, R, we have that $R = M(2^{M-1} + M - 1)$, where M is the number of subordinate modules. Because dynamic data flow analysis shows the flow of data and the implied flow of control throughout the system, we can use this law and its equation to minimize the number of inter-module relations to the ideal minimum, represented by the equation

$$I = \frac{N(N-1)}{2}$$

where I is the number of interactions between modules and N is the number of modules in the system. [3]

3.2.4 Factoring

Factoring of a system occurs when the system is divided into an upper level of control modules, controlling a lower level of operations modules. Systems that are highly factored have few upper level modules and more lower level modules, and take on a hierarchical form. Further, highly factored systems should ideally be organized such that a set of input modules collect and organize input to pass to a central transform branch, which processes the data taken from the input, or afferent, modules, and sends it to a collection of output, or efferent, modules for final formatting and output back to the user.[4]

3.2.5 Scope of Control

Scope of control in an organized, factored system is a measurement of the control of superordinate modules against subordinate modules. Ideally, the scope of control should be such that decisions made by a superordinate module *only* effect its direct subordinates, and no other modules in the system. This ensures that during maintenance and modification, debugging, and testing, problems or errors caused by one module will not have a ripple effect on other, unrelated modules in the system. This simplifies debugging and maintenance and modification, thus reducing complexity, cost, and time devoted to fixing errors.[5]

3.2.6 Black Boxes

A module is defined as a true black box whenever the module can be fully utilized with no regard for its construction, and when it performs the same action or function for every invocation, without regard to its inputs or outputs. Dynamic data flow can show the existence of black boxes in a design by showing how the inputs to a module relate to its outputs.[6]

3.2.7 Fan-In/Out

Fan-in and fan-out occur when a module is a target for multiple, higher level modules, or when a module outputs to multiple, lower level modules. Fan-in is desirable because for multiple modules needing the same function, that function can be coded, debugged, and maintained once, no matter how many modules it services. However, to ensure reduced complexity and cost, the module that is the target of fan-in should be a terminal module, and should be a true black box - it should perform the exact same function for every module it services. Fan-out is less desirable, because it implies a linkage between a servicing module and the serviced modules. A change in the servicing function may require changes in the modules being serviced.

[3]See softeng1.docx, page 40, by David Burris for more information on Graciunas's Law.
[4]See softeng2.docx, page 11, by David Burris for more information on factoring.
[5]See softeng2.docx, page 13, by David Burris for more information on Scope of Control.
[6]See softeng1.docx, page 34, by David Burris for more information on black boxes.

Further, modules which are both targets of fan-in *and* fan-out are the least desirable, as there is an implied linkage between the servicing module and the serviced modules, and also because the target module becomes subject to Graciunas's Law, causing complexity to rapidly increase.[7]

4 Design #1: Top Down Iterative Refinement

4.1 General Considerations

Top down iterative refinement focuses on the importance of factoring and the decomposition of the largest system modules into simpler, smaller ideas that are separately implementable. Modules are decomposed from left to right, with an implied flow of control in the same direction. Thus, a module on the left is said to be superordinate to the modules on its right (the subordinates). Modules with the same level of control in the hierarchy are grouped on the same level. The basic steps of top down iterative refinement are as follows:

1. Define a single node to represent the system as a whole.
2. Identify the largest conceptual chunk of which the system is composed.
3. Define these chunks to be the next level of decomposition.
4. Select a new node as the parent and iterate over steps 2 and 3 until the system has been fully decomposed into its constituent chunks, or modules.

In this way, the design will show a large system factored into its smallest components that are then separately implementable.

4.2 Metrics

The analysis metrics used for dynamic data flow are:

- Coupling (data & control)
- Miller's Law
- Graciunas's Law
- Factoring
- Black Boxes
- Scope of Control/Effect

See the appropriate subsections inside Section 3, Subsection 3.2 for definitions of the above metrics.

4.3 Analysis

The software system in question is a two-dimensional, point-and-click adventure game. We will use a decomposition diagram in our analysis. This diagram can be found in Section 8, Subsection 8.1, Subsection 8.1.1.

[7]See softeng2.docx, page 14, by David Burris for more information on fan-in/out

4.3.1 Miller's Law

Top down iterative refinement shows the largest conceptual modules for each level of decomposition. Since Miller's Law is concerned with the interaction between these modules and modules on the lower levels of decomposition, top down iterative refinement can indicate large modules with many child modules, and these modules may be subject to Miller's law. For example, consider a large module, A, that subdivides into 4 smaller modules, and the largest of these modules, B, subdivides into 9 further modules. This indicates that module B is subject to the effects of Miller's law, and that perhaps module B contains more than a single idea and could be subdivided into additional modules with fewer children. Examining the diagram for our system, we can see that the Engine module has 5 sub-modules, and is the largest module in the system. While an ideal design would have the number of child modules per parent average between 3 and 4, having 5 child modules is still within the boundary limit of 7 ± 2.

4.3.2 Graciunas's Law

Top down iterative refinement shows the relationships between parent modules and child modules, for the decomposed modules. As does Miller's law, this allows us to find modules with more relationships than would be optimal, and correct them before moving onto implementation and dealing with the issues there. In this case, the module Engine has 5 submodules, and by the equation

$$I = \frac{N(N-1)}{2}$$

where I is the number of interactions and N is the number of modules, we have that

$$I = \frac{6(5)}{2} = 10$$

And so the module Engine has 15 possible interactions with its modules and parent, assuming that data and control are limited to a single path. This is a necessary design requirement, because failing to limit data and control to a single path subject Engine to the equation

$$R = M(2^{M-1} + M - 1)$$

where R is the number of relationships and M is the number of modules. In that case, Engine would have

$$R = 6(2^5 + 6 - 1)$$

$$R = 6(32 + 5) = 6(37) = 222$$

And so we can clearly see the benefit of limiting the paths for data and control.

4.3.3 Scope of Control

Top down iterative refinement shows an implied control flow from left to right. Thus, analysis of the decomposition diagram generated by top down iterative refinement of our design shows that each parent module controls only a subset of child modules. This is imperative to a good design, because it simplifies maintenance and modification by limiting the effects of changes in one module on the other modules in the system.

4.3.4 Black Boxes

Top down iterative refinement is very useful for designing a system composed of black boxes. A black box is a module that is used by the system with no knowledge of its construction, and that module provides the exact same functionality to every module that invokes it. In short, it can be fully utilized without knowledge of how it is constructed. Because top down iterative refinement necessarily breaks down larger ideas into their constituent pieces, and these pieces directly translate to system modules, we can then construct black boxes from the resulting decomposition.

4.3.5 Factoring

The design generated by top down iterative refinement is necessarily factored. Because larger chunks are decomposed into smaller chunks, those chunks can translate, generally, directly to a hierarchical structure, with superordinate modules controlling the actions of subordinate data processing modules.

4.4 Conclusion

While top down iterative refinement is a useful methodology for showing the general design structure, it fails to show instances of coupling, does not indicate data flow, inputs and outputs, and only shows the high level "chunks" of the system. These chunks can be implemented separately as individual modules, but without another design methodology such as object oriented design, it is hard to see, in our opinion, exactly how modules will interact during implementation.

5 Design #2: Dynamic Data Flow Analysis

5.1 General Considerations

Dynamic data flow analysis is a design methodology used to demonstrate the flow of data through a software system. A good software design, when analyzed with the dynamic data flow method will show a well defined input, or afferent, branch where data is collected and acted upon by related input processing functions (such as opening a file or reading the file contents to an array for use by another module). Data processed by the afferent branch should then be passed to a well defined central transform branch where the core operations of the software are performed on the data, and output data is generated and returned. Finally, this output data is sent to a well defined output, or efferent, branch, where it is processed into its final output format, such as a report.

Two data flow diagrams are used when plotting the path of data from input to output in a software system. The first is a flow diagram, which shows data coming into the system, how it moves through the system's various modules, and eventually is output in a useful way. The second diagram is a hierarchical diagram that shows the modules that make up the system, and how they are related. This diagram shows the flow of data, but also implies the control structure of each module as it relates to the others and to the system as a whole. Data couples and control couples are indicated here as well. Further, because this type of analysis focuses on the flow of data, with an implied control structure and without any concern for the contents of the actual function bubbles present in the diagram, it is not able to determine or show the cohesiveness of any particular module. This is detrimental because it potentially allows modules that are merely coincidentally cohesive to exist until the implementation phase, when the problems associated with that form of cohesion will likely begin to appear.

5.2 Analysis Metrics

The analysis metrics used for dynamic data flow are:

- Coupling (data & control)
- Miller's Law
- Graciunas's Law
- Factoring
- Scope of Control/Effect
- Black Boxes
- Fan In/Out

5.3 Analysis

The software system in question is that of a point-and-click, two dimensional game. We will draw conclusions from our analysis using both the data flow diagram and the hierarchical diagram from the previous dynamic data flow analysis of the software system.

5.3.1 Coupling

The design exhibits high levels of data coupling between modules, and a low level of control coupling between modules. Data coupling, while potentially undesirable, is the least detrimental form of coupling, and is likely present in every software design to some degree. In general, our design is data coupled at interfaces between higher level modules and lower level modules, and the data is being passed down the hierarchy for efferent modules and up the hierarchy for afferent modules, as expected. Control coupling occurs in the design at interfaces between controller-type modules and the modules they control. Because the system is a game, it relies on certain modules to control the overall execution of the other, subordinate data processing modules. These control modules then are necessarily coupled to their subordinate processing modules. While this is undesirable from a design standpoint, it is unavoidable.

5.3.2 Miller's Law

The design does not exhibit any modules that exceed 7 ± 2 interactions. Of concern is the `Engine` module,[8] which has 8 links to other modules in the system. The links are weak, however, as some modules will spend most of their time inactive and will only be used periodically. Thus, though the engine has 8 links to other modules, it may have between 4 and 8 actually active in a single given frame. Still, it is important to keep in mind that any further requirements on this module will therefore likely result in rapid increases in implementation, debugging, and maintenance & modification difficulties.

Regarding the system as a whole, we know that it is desirable to maintain an average of 3 to 4 interactions between modules throughout the system, and we can perform basic math to determine where our system stands, based on the data flow diagrams we have obtained through our analysis. Thus, we can derive a table showing each module and the number of interactions it has with other modules:

[8]See Appendix A, Section 9.2, Subsection 9.2.2 for the appropriate diagram.

Get Mouse State	3
Create Save File	2
Parse Save File	3
Load Level	3
Engine	8
Update Player	1
Compare Player & Actor Positions	2
Video Manager	2
Render Video	2
Audio Manager	2
Play Audio	2
Save Game	2
Total:	32

So we can calulate A, the average number of interactions between modules with regard to Miller's Law by dividing 32 by the number of modules N, where $N = 12$:

$$A = 32 \div N = 32 \div 12 = 2.667$$

Therefore, while one of the design's modules approach the limits of Miller's Law, the overall average number of interactions between modules with regard to Miller's law is within the expected range of values, at 2.667.

5.3.3 Graciunas's Law

Because the design makes use of a highly factored, modular design, the majority of data is passed from one module to another, using a one-way connection where each module only communicates with its direct superior, controlling module. This ensures that each pair of modules only has one path of communication between them and, thus, only 1 relationship. The `Engine` module,[9] however, interacts with 6 modules, with two relationships between two of its subordinate modules. Thus, its number of interactions are defined by the equation

$$I = \frac{N(N-1)}{2}$$

where I is the number of interactions between modules and N is the number of modules in the system:

$$I = \frac{7(6)}{2} = 21$$

And so `Engine` has 21 possible interactions in the system, and the system as a whole has

$$I = \frac{12(11)}{2} = 66$$

Limiting data and control this way substantially reduces complexity. Not implementing this type of design results in the number of relations being defined by $R = M(2^{M-1} + M - 1)$, where M is the number of subordinate modules. If we calculate that value, we have that

$$R = 12(2^{12-1} + 12 - 1)$$

$$R = 12(2048 + 12 - 1)$$

$$R = 12(2059) = 24708$$

And so we can clearly see that by limiting the number of data and control relations between modules to the absolute minimum, we have reduced the number of relations by an approximate factor of 375.

[9] See Appendix A, Section 9.2, Subsection 9.2.2 for the appropriate diagram.

5.3.4 Factoring

Our design is highly factored, in that we have a central module, `Engine`, that delegates to another, lower level of modules, and those modules delegate to yet another level of modules. Further, we can organize the modules into afferent modules and efferent modules, all sending data to or receiving data from a central transform branch, headed by `Engine`. By having a highly factored, branched design, we can ensure that each module complies with Miller's law, Graciunas's law, and that the scope of control (discussed in the next section) and scope of effect are all within the ideal parameters. Additionally, by not sending control flags back and forth between superordinate modules and subordinate modules, we have reduced control coupling to acceptable levels.[10]

5.3.5 Scope of Control

If we examine the factored design diagram,[11] we can see how the data is being moved between modules. Each module in the design is only able to request information from the module(s) directly subordinate to it, with regard to the afferent branch. The central transform branch, consisting of the `Engine` module and its two subordinate modules, only allows control decisions to be made by `Engine`, with regard to its subordinate modules. However, one issue with this design is that one of `Engine`'s subordinate modules passes back an actor event, and the type of event `Engine` receives influences how it proceeds in its execution with regard to that event. Thus, control in this case is moving from a subordinate module to a superordinate module, which is typically considered detrimental to the system. However, these actor events are requests that do not force a behavior in the engine; the engine may or may not fulfill the request depending on what it determines is the appropriate behavior, given the current state of the system. This is the only place in the design where control is passed to superordinate modules, and because the design is that of a game, this behavior is expected, and has been taken into account.

5.3.6 Black Boxes

If we examine the data flow diagram[12], we can see that our analysis shows extensive use of black boxes. Each module sends, receives, and processes data completely independently of other modules. Therefore, any change made in one module will not affect the data processing of any other modules, which reduces complexity. This ensures that a minimum amount of time will be spent debugging errors and maintaining the software throughout its life. One exception here is, as discussed in the subsection covering scope of control, is `Engine`. Because it is control coupled with a subordinate module, a change in `Engine` or a change in the types of actor events it receives from its subordinate modules is subject to additional errors in whichever module was not modified. Although this behavior is expected and accounted for, it must be paid special attention during debugging and modification, to ensure that the system continues to function properly. While this behavior is considered negative, it is necessary for the full functionality of the game itself, and cannot be avoided.

5.3.7 Fan In/Out

Fan in and fan out are two features of software design that can be indicative of a design's quality. Our design makes extensive use of both. Looking at the factored diagram, we can see that the engine module is a target of fan out, indicating a factored design that delegates processing to lower level modules and control to superordinate modules. This is an ideal state for our system to be in. If we examine other modules,

[10]See Appendix A, Section 9.2, Subsection 9.2.2 for the appropriate diagram.
[11]See Appendix A, Section 9.2, Subsection 9.2.2 for the appropriate diagram.
[12]See Appendix A, Section 9.2, Subsection 9.2.1, for the appropriate diagram.

terminating modules, or those at the lowest levels of the system, are consistently targets for fan in, which is, again, an ideal state. For the system in question, we have an average fan out value of 5. Ideally, this value would be between 3 and 4, but it still complies with Miller's Law, which recommends not exceeding a value of 7 ± 2.

5.4 Conclusion

While dynamic data flow analysis is a very beneficial design methodology, which clearly shows the flow of data through a system, data coupling, control coupling, and allows a designer to develop a highly factored system, it cannot measure or demonstrate the amount, level, or type of cohesion within each module. Thus, for our application of game development, it provides useful metrics, but is not ideally suited for what our design seeks to achieve. It does indicate, however, that we have developed a quality design, with high factoring, black boxes, low control coupling, moderate data coupling, and highly differentiated and well defined afferent, efferent, and central control branches.

6 Design #3: Static Data Flow Analysis

6.1 General Considerations

Static data flow analysis is a design methodology using Jackson's technique. this focuses on processing the data, thus generating a program structure that reflects the data that it processes. There are four steps in Jackson's technique, used for static data flow analysis:

1. Evaluate the problem environment and define the structures for data to be processed.
2. Develop a program structure based on the data structures.
3. Define the tasks to be performed in a non-procedural manner.
4. Allocate each task to a suitable component of the basic program structure.

Jackson's technique is valuable because in a traditional design environment, N individuals results in exactly N designs. Jackson's technique allows the production of a single design for a given problem, no matter the value of N.

6.2 Metrics

Metrics used for analysis of the design produced by Jackson's technique are as follows:

- Coupling
- Miller's Law
- Graciunas's Law
- Factoring
- Black boxes
- Scope of Control/Effect

6.3 Analysis

6.3.1 Coupling

When examining the design produced by Jackson's technique, we can see that there is little coupling between modules. Each module can be implemented separately and maintained on its own, without regard to how the other modules are implemented.

6.3.2 Miller's Law

The design does not exhibit any modules that exceed the limit for Miller's Law, which is 7 ± 2 interactions. The largest number of subordinate modules handled by a superordinate module is 3. (`Game Engine` and its subordinates)

On average, the system as a whole has 11 modules, with one module handling 3 subordinates, one module handling 2 subordinates, and the other 9 only handling one subordinate. So, we have that for the average A,

$$A = (3 + 2 + 9)/11 = 1.27$$

Which is to say that on average, each module has 1.27 subordinate modules, well within the expected values of Miller's Law.

6.3.3 Graciunas's Law

Since the design in question limits the paths for data and control to a single path, we can express Graciunas's law mathematically with the following equation:

$$I = \frac{N(N-1)}{2}$$

Where, as in previous sections, I is the number of interactions between modules. Considering the `Game Engine` module, with 3 subordinates, we have that

$$I = \frac{3(2)}{2} = \frac{6}{2} = 3$$

Considering the system as a whole, we have 11 modules, and so

$$I = \frac{11(10)}{2} = \frac{110}{2} = 55$$

Which is to say that for the entire system, there are 55 possible interactions.

6.3.4 Factoring

Jackson's technique allows us to develop a highly factored design. The design shows an upper level of controlling modules with a lower level of data processing modules. This creates a design that is easier to implement and easier to maintain and modify.

6.3.5 Black Boxes

The design we have developed with Jackson's technique lends itself to implementation with black boxes. Because each module performs a single function and contains a single idea, no other module cares about how any given module is implemented or what instructions it contains. Further, each module performs the same function on every piece of data it is given, without changing its execution based on the parameters it receives (a gray box).

6.3.6 Scope of Control/Effect

Since the design already exhibits high levels of factoring, with black boxes (which imply functional cohesion), we can clearly see that each module will only control the modules that are directly subordinate, and the effects of a decision in a superordinate module only affect its subordinates. Thus, scope of control and effect are a set and subset, respectively, which is ideal for our design.

6.4 Conclusion

With Jackson's technique, we have developed a design which exhibits many of the characteristics of good software design - highly factored, black boxes, and conformity with Miller's and Graciunas's laws.

7 Design #4: Object Oriented Design (UML)

7.1 General Considerations

7.2 Metrics

7.3 Analysis

7.4 Conclusion

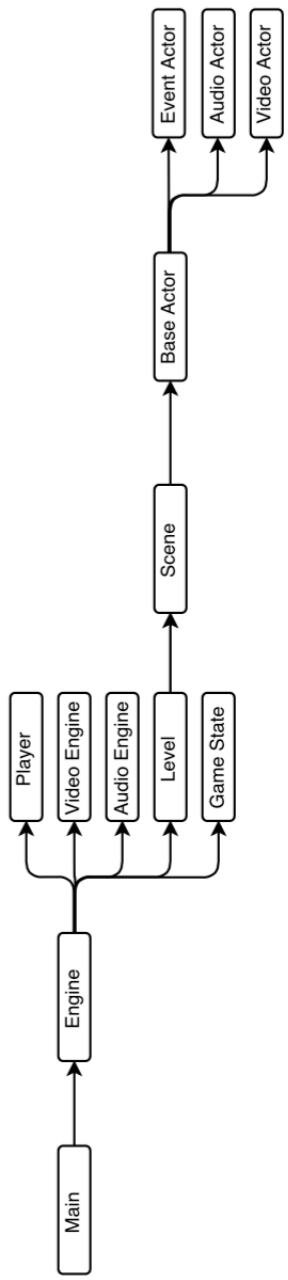
8 Design Selection & Rationale

9 Appendix A: Diagrams & Figures

This appendix contains all of the relevant diagrams for each design methodology discussed in the SDD previously.

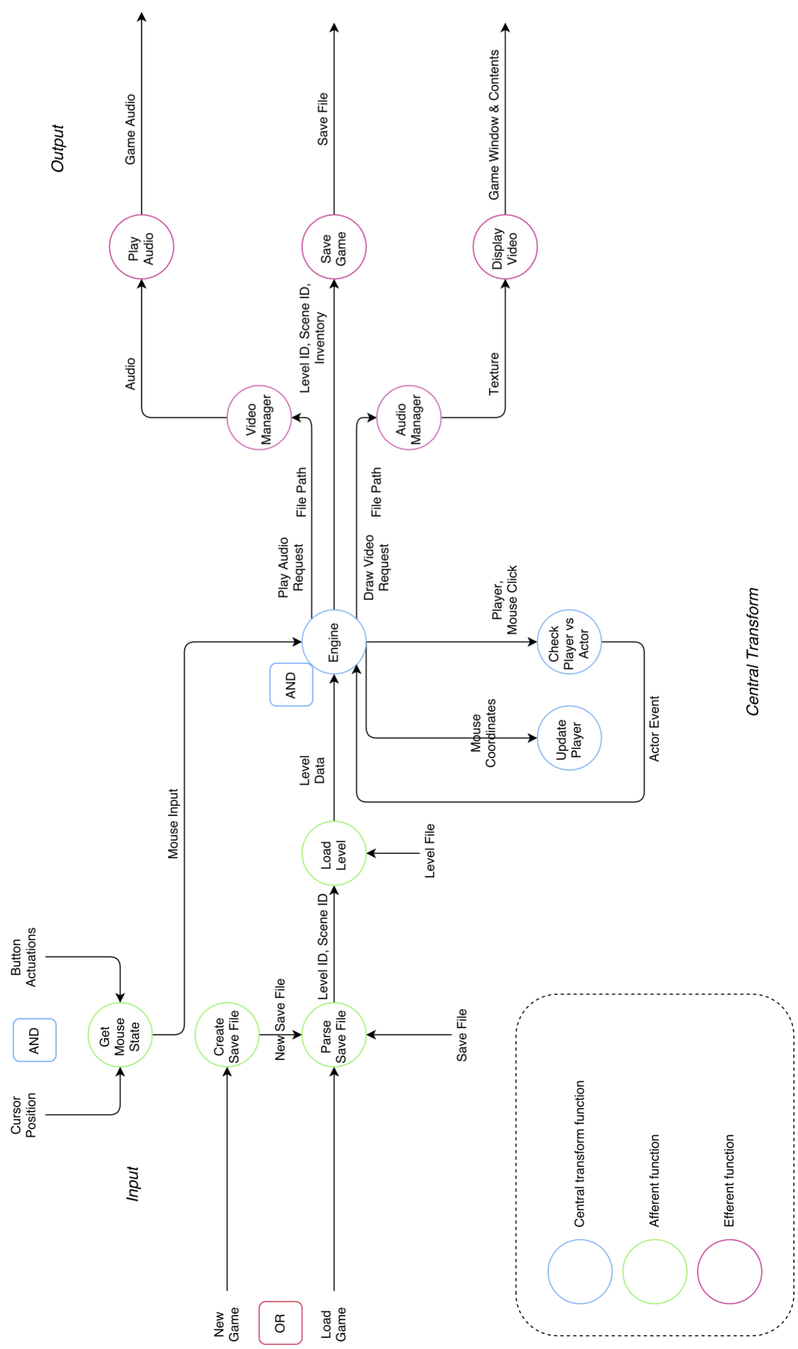
9.1 Top Down Iterative Refinement

9.1.1 Component Diagram

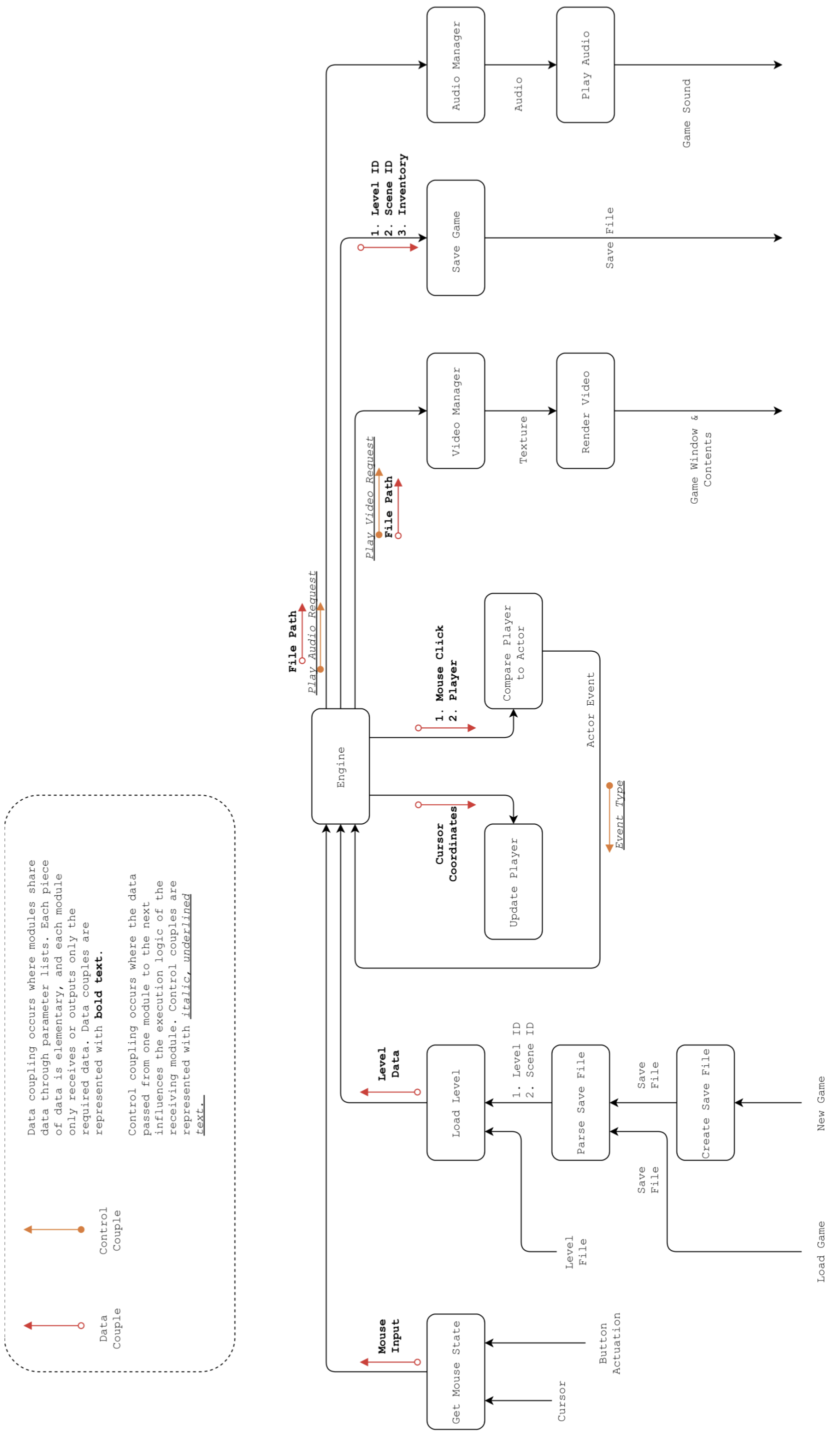


9.2 Dynamic Data Flow Analysis

9.2.1 Flow Diagram

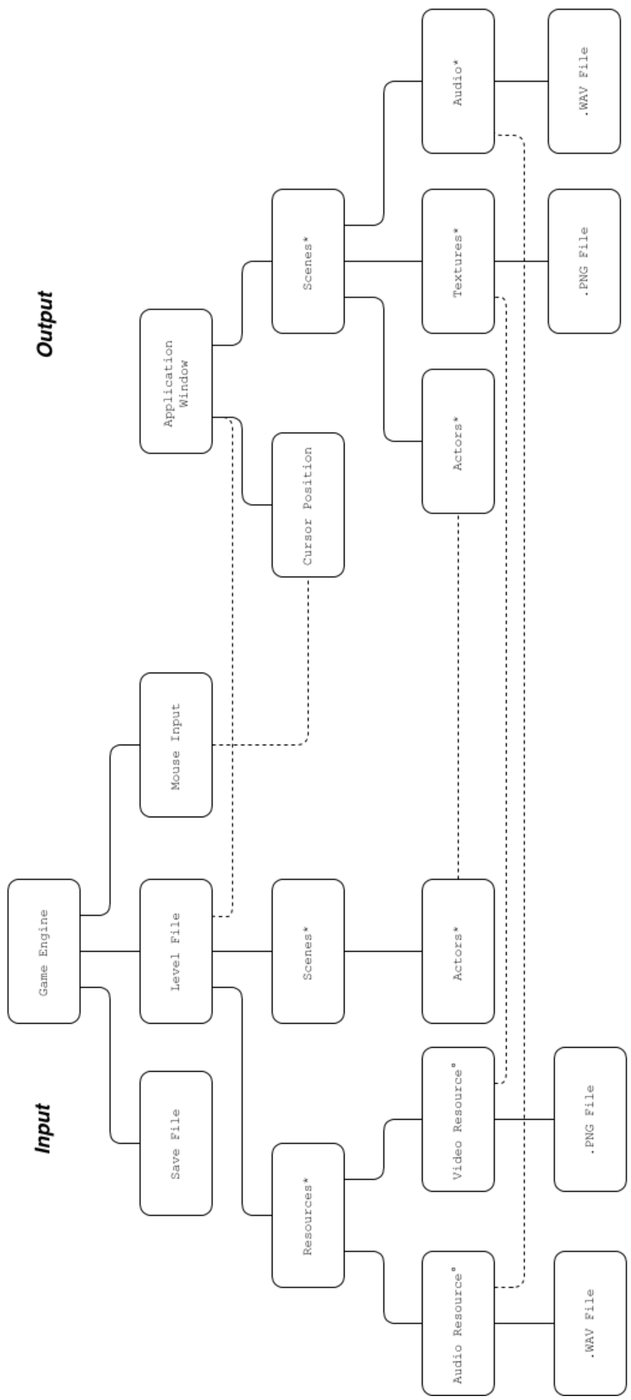


9.2.2 Factored Diagram

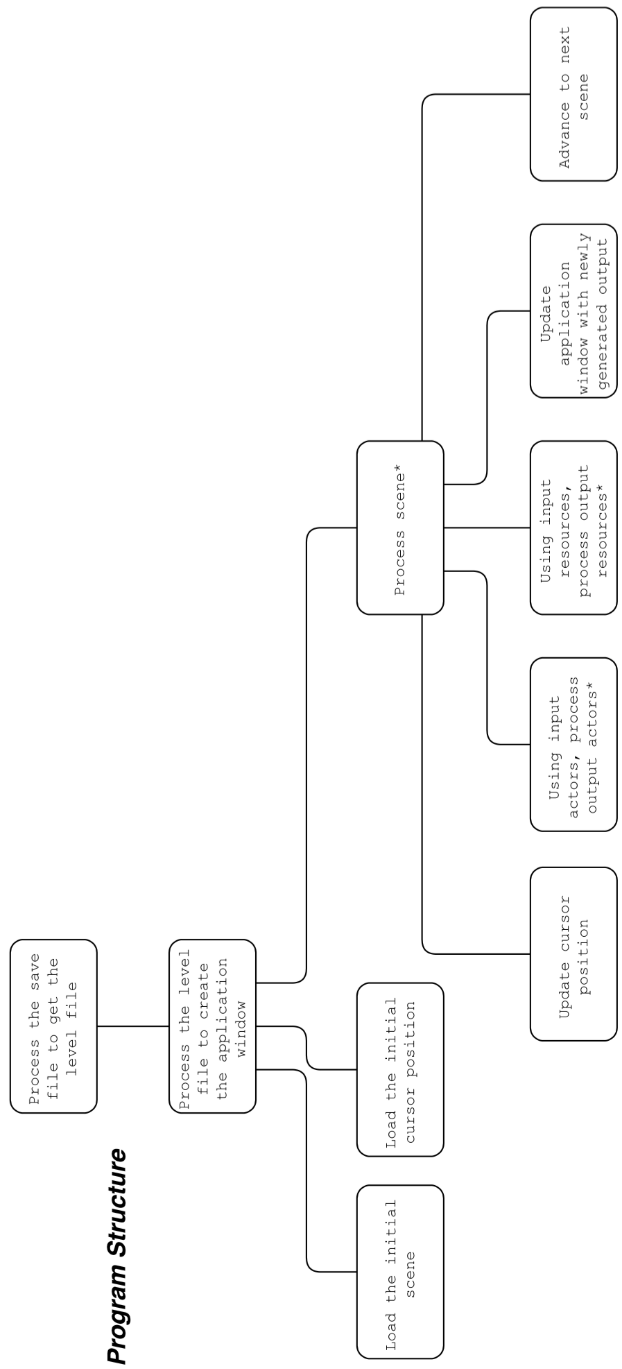


9.3 Static Data Flow Analysis

9.3.1 Data Structure Diagram



9.3.2 Program Structure Diagram



9.3.3 Nonprocedural Program Tasks & Task Allocation

Tasks to be performed:

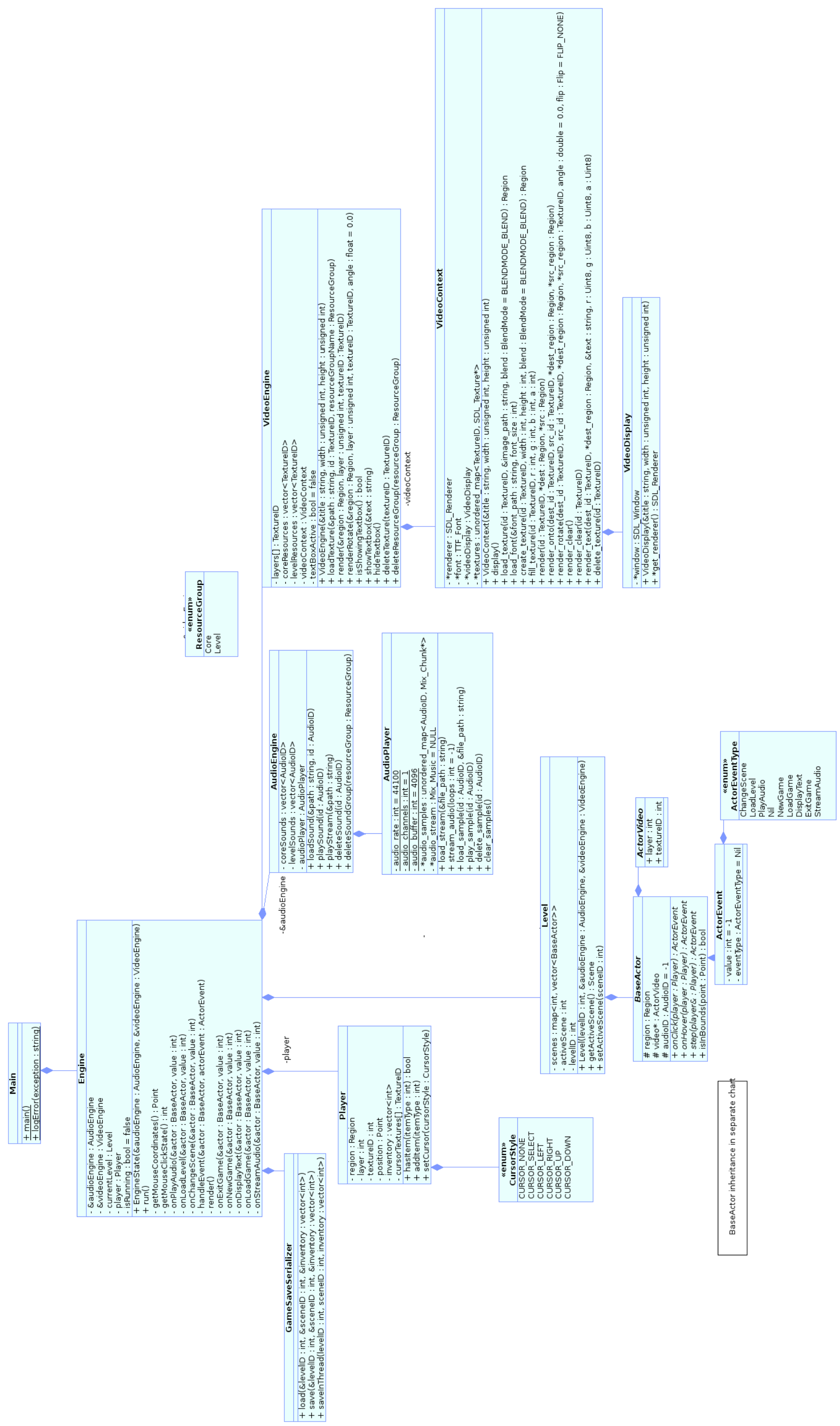
```
0) Initialize engine
1) Load a save file (create one if necessary)
2) Load a level file
3) Load level resources & data
4) Send initial scene to engine
5) Poll mouse input
6) update player coordinates
7) Compare player to scene actors
8) Get actor event
9) Respond to the actor event
10) Check player inventory
11) Advance to next scene
12) Write a save file if necessary
```

Task Allocation:

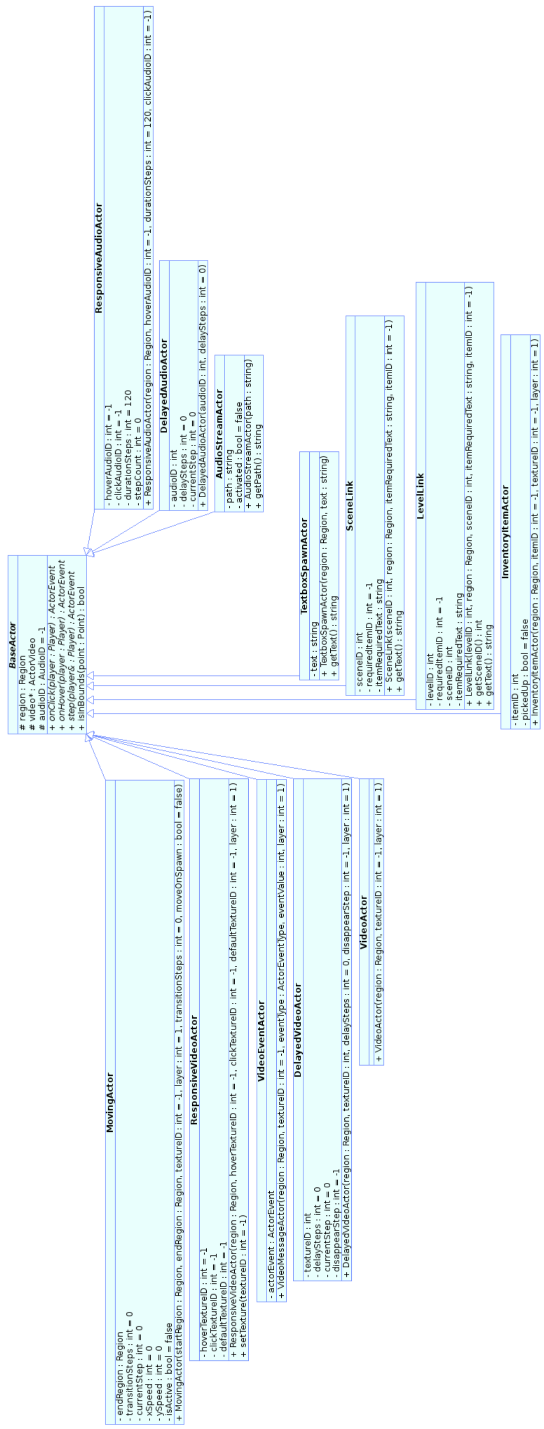
```
Main:
      Initialize engine module
Serializer:
      Load a save file (or create one)
      Get player inventory
      Get level file
      Parse level data
      Send data to engine
Engine:
      Initialize required objects
      Load initial scene
      Loop:
            Poll for mouse input
            update player coordinates
            compare player to scene actors
            get actor event
            respond to actor event
            check player inventory
            advance to next scene
```

9.4 Object Oriented + Unified Modeling Language

9.4.1 UML Diagram

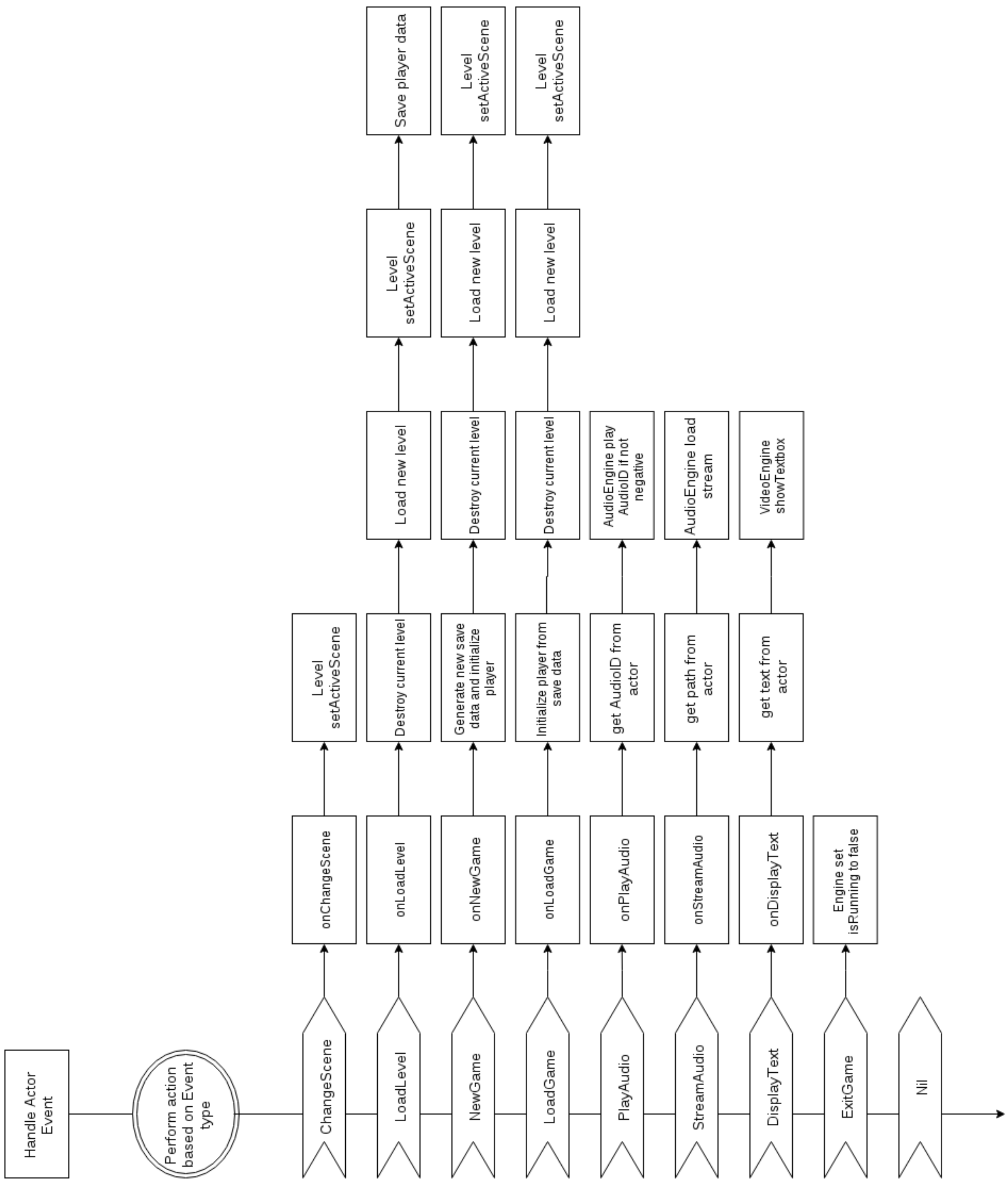


9.4.2 Actor Inheritance Sub-diagram

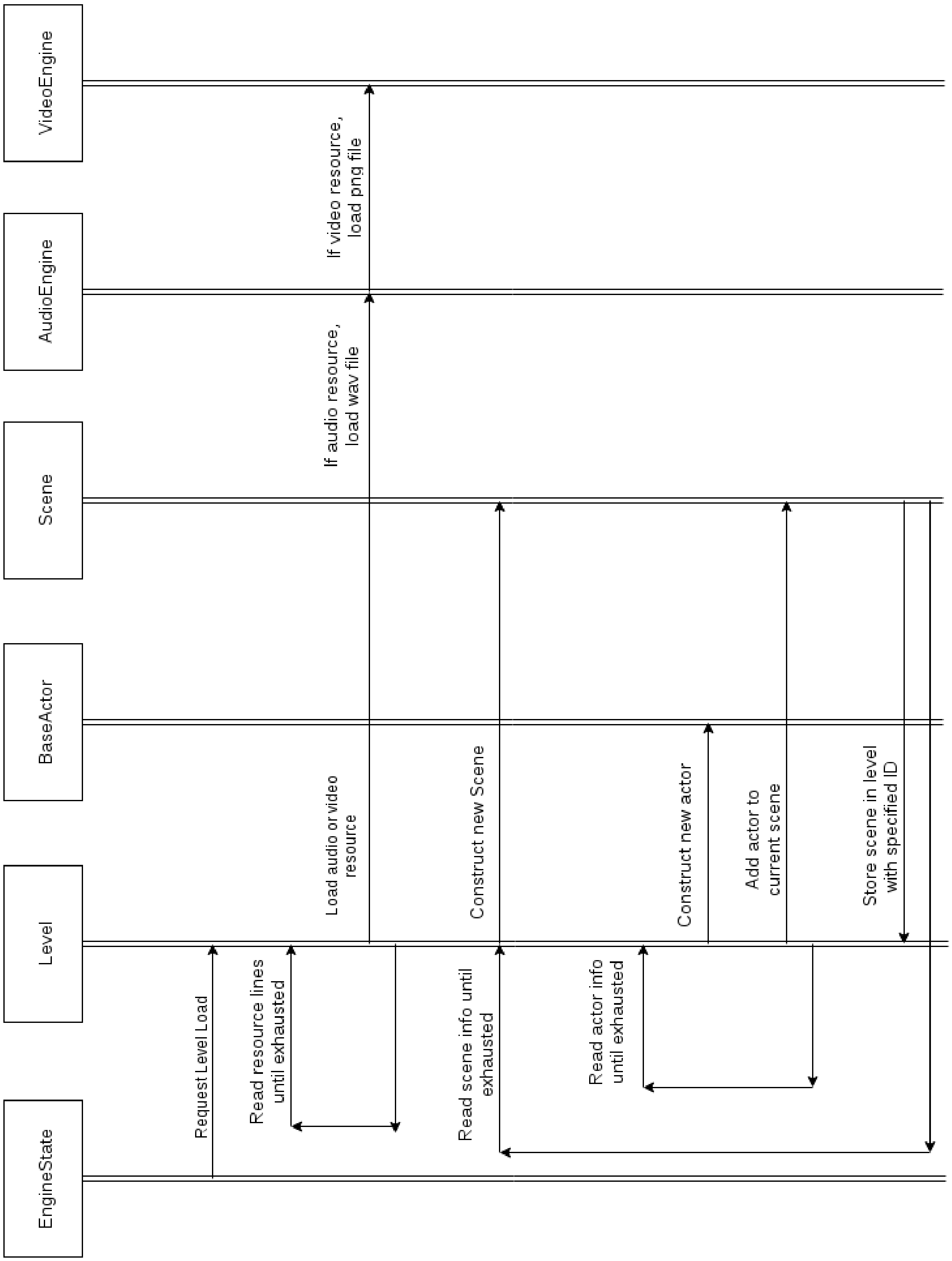


9.4.3 Timing Diagrams

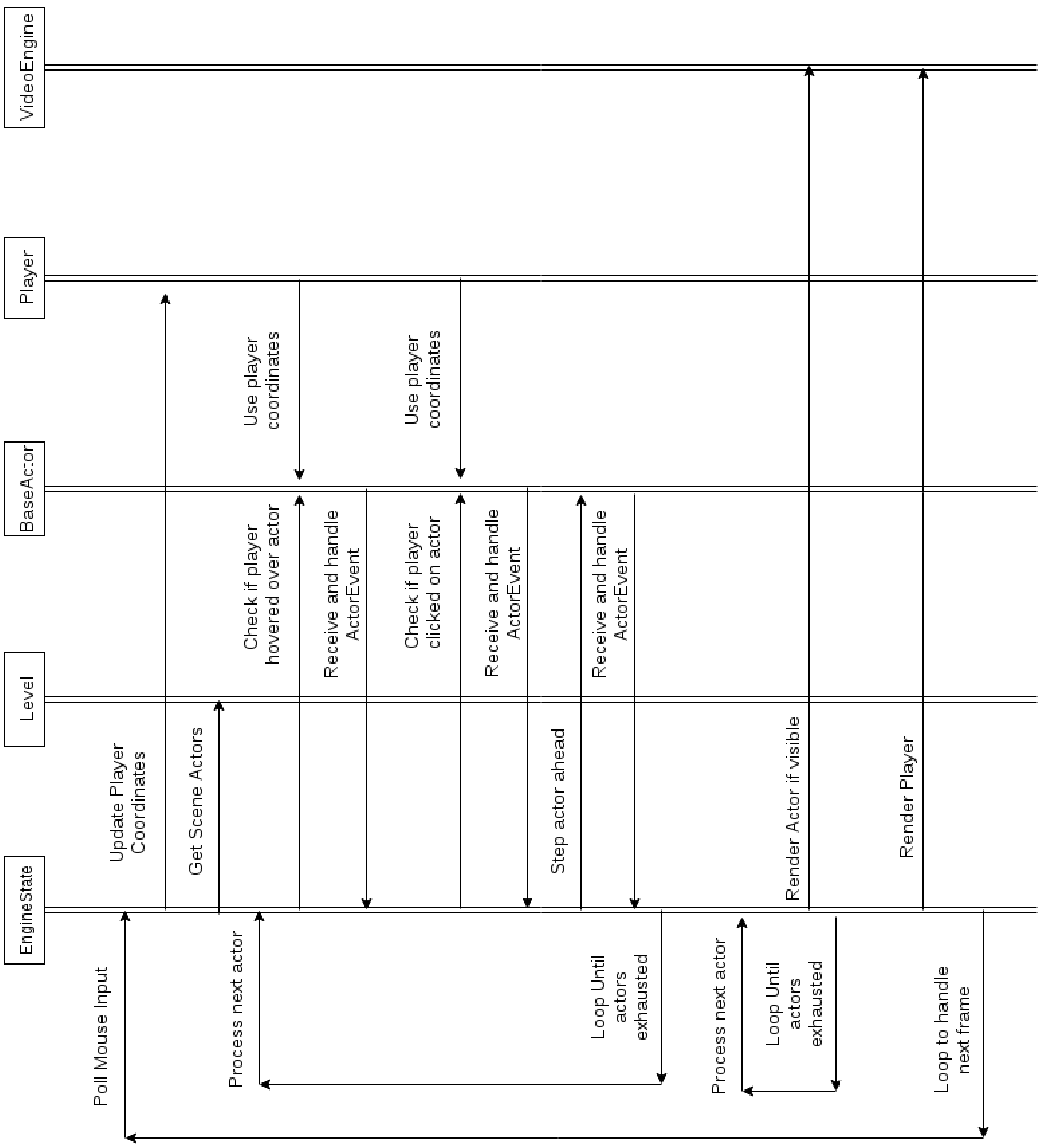
Handle Actor Event



Load Level

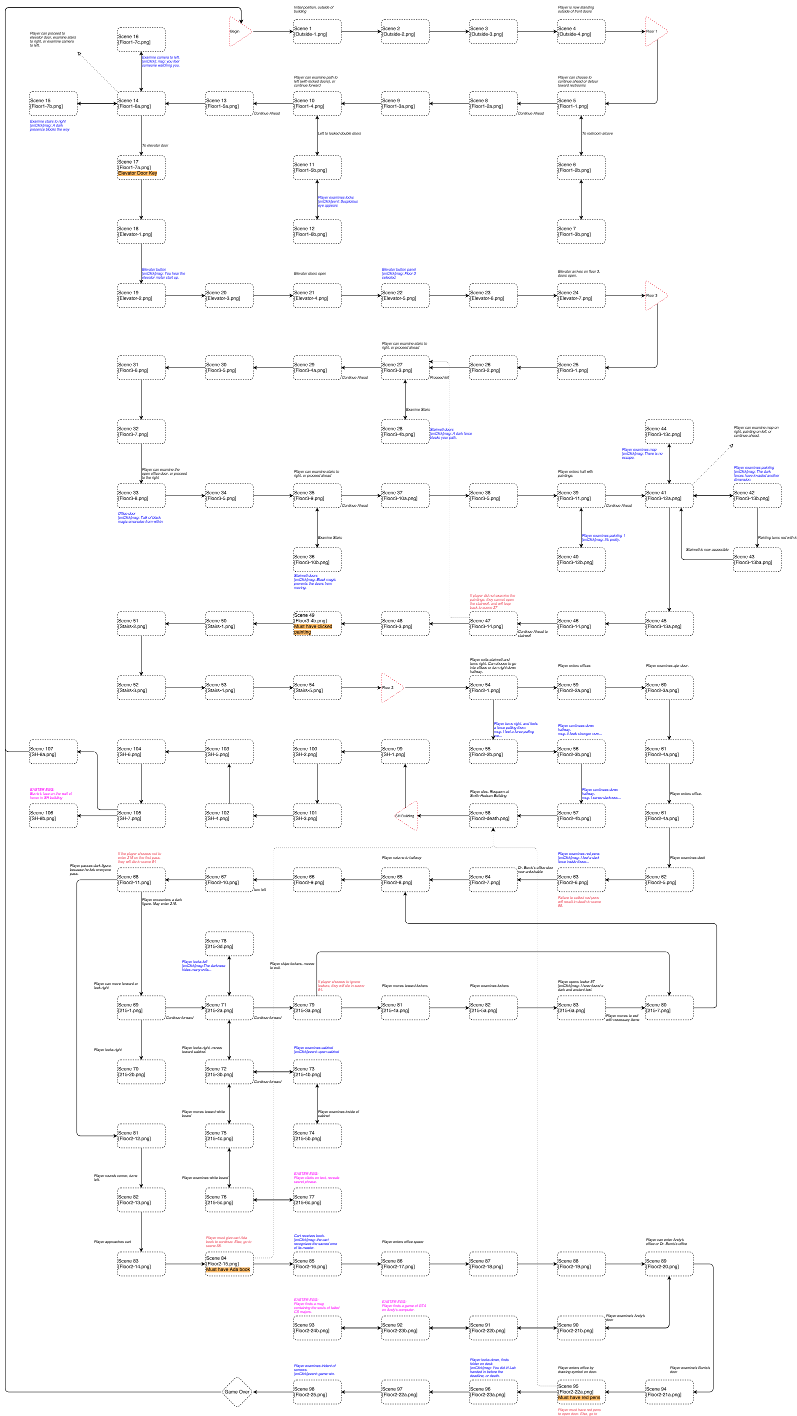

VideoEngine
AudioEngine
Scene
BaseActor
Level
EngineState
Request Level Load
Read resource lines until exhausted
Load audio or video resource
If audio resource, load wav file
If video resource, load png file
Read scene info until exhausted
Construct new Scene
Read actor info until exhausted
Construct new actor
Add actor to current scene
Store scene in level with specified ID

Main Loop


VideoEngine
Player
BaseActor
Level
EngineState
Poll Mouse Input
Update Player Coordinates
Get Scene Actors
Process next actor
Check if player hovered over actor
Use player coordinates
Receive and handle ActorEvent
Check if player clicked on actor
Use player coordinates
Receive and handle ActorEvent
Step actor ahead
Receive and handle ActorEvent
Loop Until actors exhausted
Process next actor
Render Actor if visible
Loop Until actors exhausted
Render Player
Loop to handle next frame

10 Appendix B: Level Progression & Miscellany

10.1 Level Progression Diagram



A full sized level progression diagram has been attached separately. It is included here merely for completeness, though it can be magnified and viewed using a suitable PDF viewer on a computer.

10.2 Miscellany