University of Alabama in Huntsville

LOUIS

Honors Capstone Projects and Theses

Honors College

4-22-2022

Software Metric Analysis of Robot Vacuum (RoVac) Simulator

Alec Braden Mueller

Follow this and additional works at: https://louis.uah.edu/honors-capstones

Recommended Citation

Mueller, Alec Braden, "Software Metric Analysis of Robot Vacuum (RoVac) Simulator" (2022). *Honors Capstone Projects and Theses*. 722.
https://louis.uah.edu/honors-capstones/722

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.

Software Metric Analysis of Robot Vacuum (RoVac) Simulator

by

Alec Braden Mueller

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

Date

Honors Capstone Director: Dr. Harry Delugach

Associate Professor of Computer Science

Alec Mueller

Student (signature) Date

4-22-2022

Director (signature) Date

Department Chair (signature) Date

Honors College Dean (signature) Date

Honors College
Frank Franz Hall
+1 (256) 824-6450
(voice)
+1 (256) 824-7339
(fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis. In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Alec Mueller

Student Name (printed)

Alec Mueller

Student Signature

04/22/22

Date

Table of Contents

Abstract 4

Introduction 5

Requirements 7

Development 8

Software Metrics 11

- Visual Studio Integrated Code Metric Analyzer 11
- Source Lines of Code 12
- Cyclomatic Complexity 15
- Depth of Inheritance 18
- Class Coupling 19

Discussion 21

Conclusion 23

Appendix 24

Abstract

The roVac Simulator Program is a standalone application that was developed using the Unity game engine. It was designed and developed for the Senior Design Project class, CS 499, from August 2021 to December 2021. The requirements for the features of this project were provided in a statement of work from the professor, Dr. Harry Delugach, with slight modifications made as necessary at the request of the group. The requirements for the Honors Capstone project were completed by Alec Mueller under the guidance of Dr. Harry Delugach.

During this project, I learned a lot about the software development process that I have since been able to apply in the rest of my coursework as well as the workplace at my internship. Due to the project being produced by a team, I was able to improve my leadership, communication, and time management skills through the organization of the project and the scheduling of its development. My team meshed well together which allowed each of us to make contributions to the project without fear of judgment. This relaxed cooperative environment helped me feel more comfortable trying to step into an organizational role than I had in the past. Since I was the only team member completing the honors thesis, I was tasked with time recording and notetaking, which gave me experience pinpointing what was important in each of our meetings and discussions. Many of our software ideas would originate in the meeting discussions, which meant that they had to be recorded or we might forget about them. These notes would often drive the software development tasking and scheduling done between team meetings.

Introduction

The roVac cleaning simulator provides both roVac users and manufacturers to test roVacs in a customizable simulation environment. The simulator will allow the user to create a room in their desired configuration, with the option for different flooring materials and the placement of obstacles. Four different path-following algorithms will be available for testing to find the optimal path-following method that the roVac can use given its battery life or usage time.

The testable path-following algorithms will be random path-following, spiral path-following, snaking path-following, and wall-following. The roVac in the simulation will have a built-in battery life that will be used as a stopping point for the simulation if it is not ended manually by the user. Upon user request, or after the completion of a run of the simulation, different pieces of data will be saved into a database and become available for future analysis. The recorded data will include a time and date stamp of the start of the run, the path-following algorithm chosen and its current version, a unique identification for the floor plan, the floor coverage percentage, and the total vacuum run time.

This paper will first analyze the development lifecycle of the project and the contributions and interactions between project team members in order to get a better idea of the process behind the creation of the product. Then, various software metrics measuring the actual software written for this simulation will be discussed and analyzed in order to provide insight into the benefit of their use not only for this project but software development in general. Ideas will be discussed for possible future

improvements that can be made to the product based on the knowledge gained from the metrics.

Requirements

The purpose of my project within the CS 499 - Senior Team Project course was to design and produce software capable of providing a user with the ability to model a house or room from scratch, place a robot vacuum cleaner (roVac) within the created environment, select a path-following algorithm for the vacuum to use, and then run a simulation of the cleaning process that results from the selections. Furthermore, run data needed to be saved and displayed if desired in order to allow for the comparison of various runs of the simulation.

There were three primary focus areas present in the requirements: the house, the vacuum, and the simulation. There were 5 house requirements that specified details about the modification of the layout of the house and its constraints, traits of placeable obstacles, and floor cleaning difficulties. The vacuum had 6 requirements specifying its characteristics, dimensions, and features. Finally, the simulation itself had 9 requirements which included how the user would be able to interact with the application and a list of the information that would be tracked and recorded for each run.

Development

In order to fairly divide the project work among members and effectively devise a development plan for it, the requirements had to be individually analyzed and brainstormed. We took each requirement and discussed how long its implementation was estimated to take and possible approaches to its fulfillment. Then, each member's skills were considered and the tasks were assigned accordingly with the time estimation in mind. For example, members with experience in graphical user interface design would be prioritized for tasks related to that. This dissection of the requirements allowed us to envision a rough timeline for the project that we could use to guide us as we progressed through development. Of course, plans never play out exactly as expected but it is far better to take this approach than do nothing to prepare.

Taking a look at the development of the project and the labor contributions made to it can help with understanding the final product of the roVac simulation project. At the beginning of the brainstorming process in the initial phase of the project, the team decided to use Visual Studio and the Unity game engine in 2D to implement the simulation that we envisioned. Visual Studio is an integrated development environment (IDE), which are applications that provide a framework for users to organize and develop code for their projects in a centralized location with features such as editing and debugging code. It also handles the conversion of code from human-readable programming languages down to less readable binary 0s and 1s, which is all that the computer can understand. Essentially, Visual Studio makes the programmer's job much easier by handling nearly all of the complicated processes that need to happen behind the scenes in order for programs to run. Unity on the other hand is an example of a

game engine. A game engine provides the user the means to create three-dimensional and two-dimensional games, as well as interactive simulations, in the case of this project [1]. The use of a game engine for this project allowed us to build an end product simulation without worrying about programming a full physics engine (gravity, shadows, and object collisions for example) or displaying the simulation on the screen. A large amount of functionality was provided to us due to it already existing in the Unity game engine. We will see later just how much code the team was able to avoid writing by using Unity.

We chose to use Discord as the primary means of team communication, GitHub (a website commonly used to share and host code) to collaborate on the code, Google Drive to store files and documents, and ClickUp as our project management tool.

As the project began development, we realized certain aspects of the work that could be made easier with some changes. On October 3, 2021, we decided to transition to Unity 3D instead of 2D to achieve a better-looking end product. Additionally, on November 11, 2021, we switched to Visual Studio Code as the primary code development platform due to its increased customizability.

For this project, the team was required to mark down the time that was spent on the project each week. This tracking helped members keep up with each other in progress and not fall behind in participation. It also helped determine who would be best suited for each task, given how long it was estimated to take them.

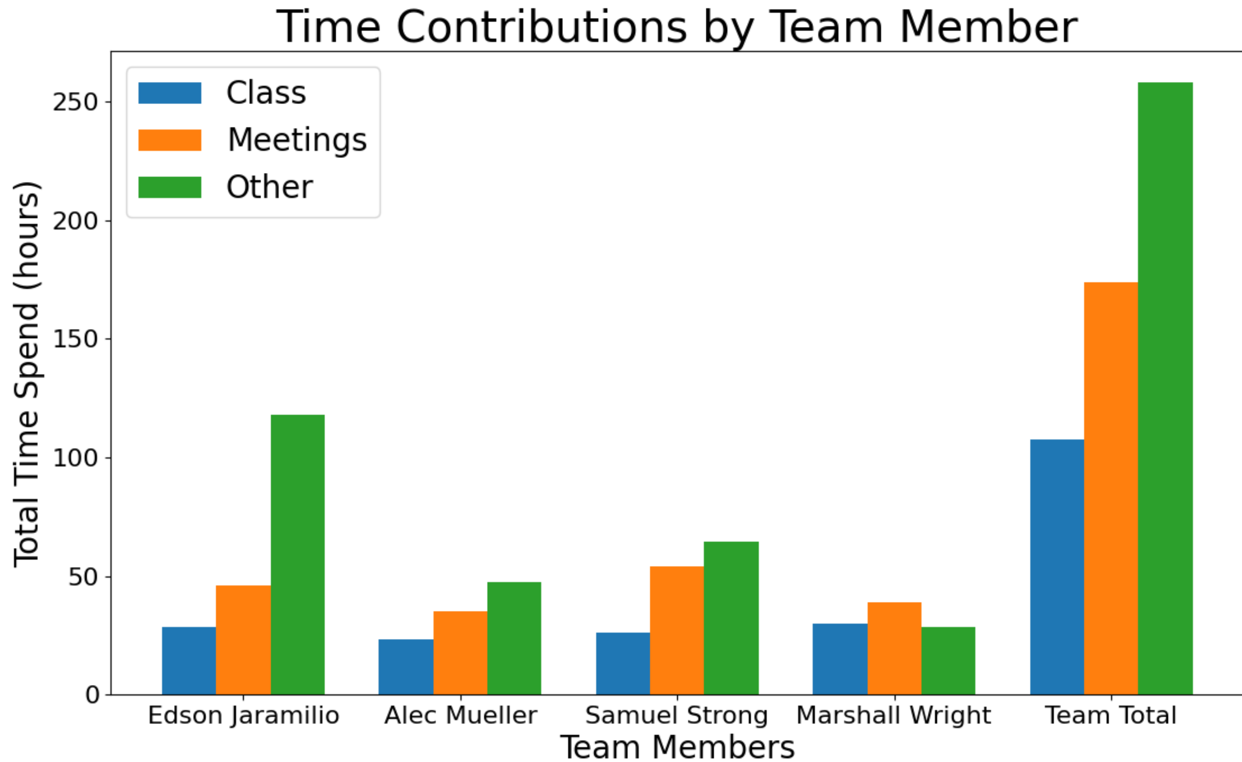


Figure 1. Time Contributions by Team Member

The team worked collectively on some parts of the project, and individually on others. Any time spent coding, testing, working on documentation, or researching was noted under “other”. Despite the distribution of time spent, each member definitely pulled their own weight and cooperated to produce the best final product possible. The programming and testing took by far the most time, which is expected due to the complexity of the desired functionality of the software. Meetings were held at minimum once a week, and often after class. Nearing the expected completion date of the project, more frequent and longer meetings were held to get everything tested and prepared to be deemed complete, so a lot of the meeting hours come from this period.

Software Metrics

Now that we have a good understanding of the project as a whole, the code can be more closely analyzed through the lens of software metrics. Software metrics are standards of measuring the degree to which software systems or processes possess a property or characteristic[2]. This definition is fairly broad, but specific examples of these metrics provide deeper insight into their explanation and use. Metrics can be gathered for both code and design, and provide useful information that can help improve software design and efficiency, especially for large-scale projects. A large number of these metrics exist, so only some of them will provide an actual benefit to this project in particular.

Visual Studio Integrated Code Metric Analyzer

Within Visual Studio (Visual Studio 2019 Version 16.9.4 used here), there is a built-in code metric analyzer that can be used to generate results for a few common metrics. This is a good way to take a look at the roVac software and metrics related to it. Now, it must be noted that Unity, due to it being a game engine and development environment, has a large number of native features, including physics engines, UI managers, and rendering capabilities. These features will be accounted for in the code metric analysis, with notations provided to differentiate between group written software and built-in software. I ran the Visual Studio metric analyzer and thought it would be interesting to visualize some of the metrics using graphs that I generated using Python.

Source Lines of Code

To start, an obvious metric for software is simply the number of lines of source code. I first took a look at the total number of lines of code required for the project to function as desired, which includes both group-written and Unity native code.

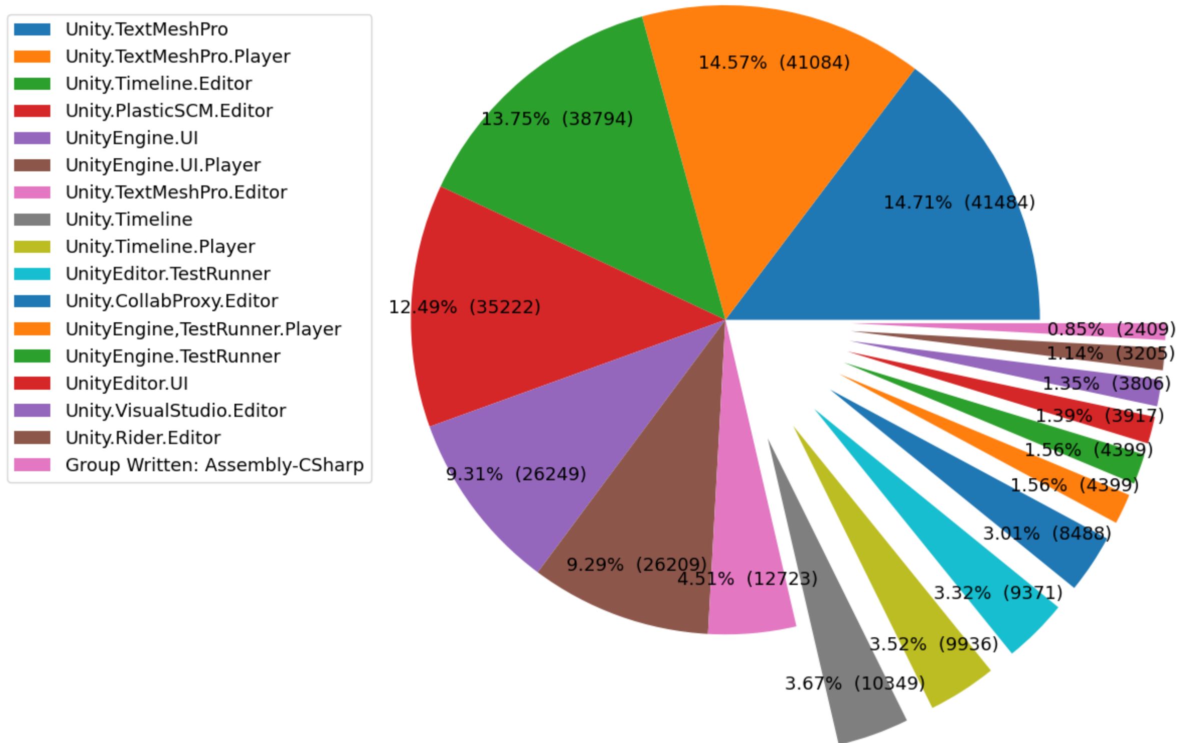


Figure 1. Total Lines of Project Code with Unity Code Included

The group written code makes up a mere 0.85% of the total project source code. This is unsurprising and goes to show just how much code is required on the backend of a game engine that the users never even see. It is a nested hierarchy of views since the user of the roVac simulator does not see the code that went into creating the application, and the team does not see the code that went into creating the game engine we used to develop it. The code my group wrote is essentially interfacing with

the Unity engine to use the built-in features in a way that produces our desired result. This chart only accounts for the pieces of Unity directly used for this project, such as the UI, meaning that the engine has even more code under the hood that goes unappreciated by unaware users.

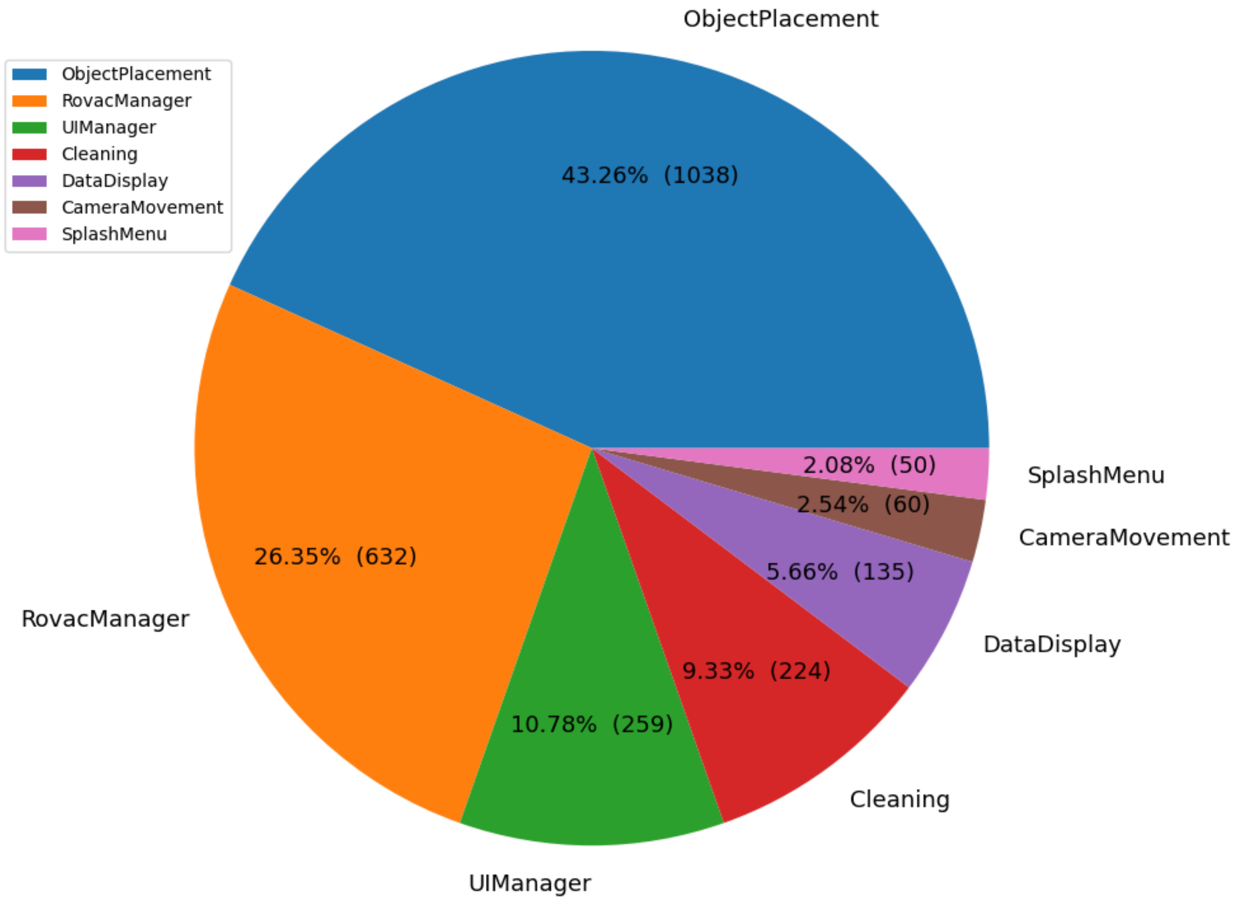


Figure 2. Lines of Group Written Source Code

Taking a closer look at the group written source code in particular, which was the 0.85% of the total project code seen in Figure 1, nearly 70% of it is comprised of the ObjectPlacement and RovacManager scripts. The majority of the functionality of the application relies on the features provided by these scripts, so it makes sense that they would comprise the largest portion of the code.

The number of lines of source code is a fairly surface-level metric and does not provide much insight into the software other than a general idea of where the major features of the application are located in the code. For example, this metric can provide

less useful information if certain scripts are more optimized than others. If the Cleaning script had very poorly written and inefficient code, it would falsely appear to make up a larger portion of the source code and would skew the perception of which scripts provide the most functionality. Despite this pitfall, codebases today tend to be quantified by the number of lines of their source code, especially when being referred to by parties outside of the industry. I often see articles online that bring up how many lines of code a given video game or software is comprised of in order to point out how complex or impressive it is, such as an estimation that Facebook has 65 million lines of code[3]. Obviously this large of a number is indicative of the complexity of Facebook, but on a smaller scale for example, with two programs having 1000 and 2000 lines of code respectively, it is not necessarily reasonable to assume that the 2000 line code is twice as functional or complex as the 1000 line code. Additionally, the definition of a “line” of code is subjective and can vary from the inclusion of every single line that is written in a program to the exclusion of comments, blank spaces, and wrapped lines. These are some of the reasons that using lines of code as a software metric can often yield misleading results about the efficiency or functionality of a given program.

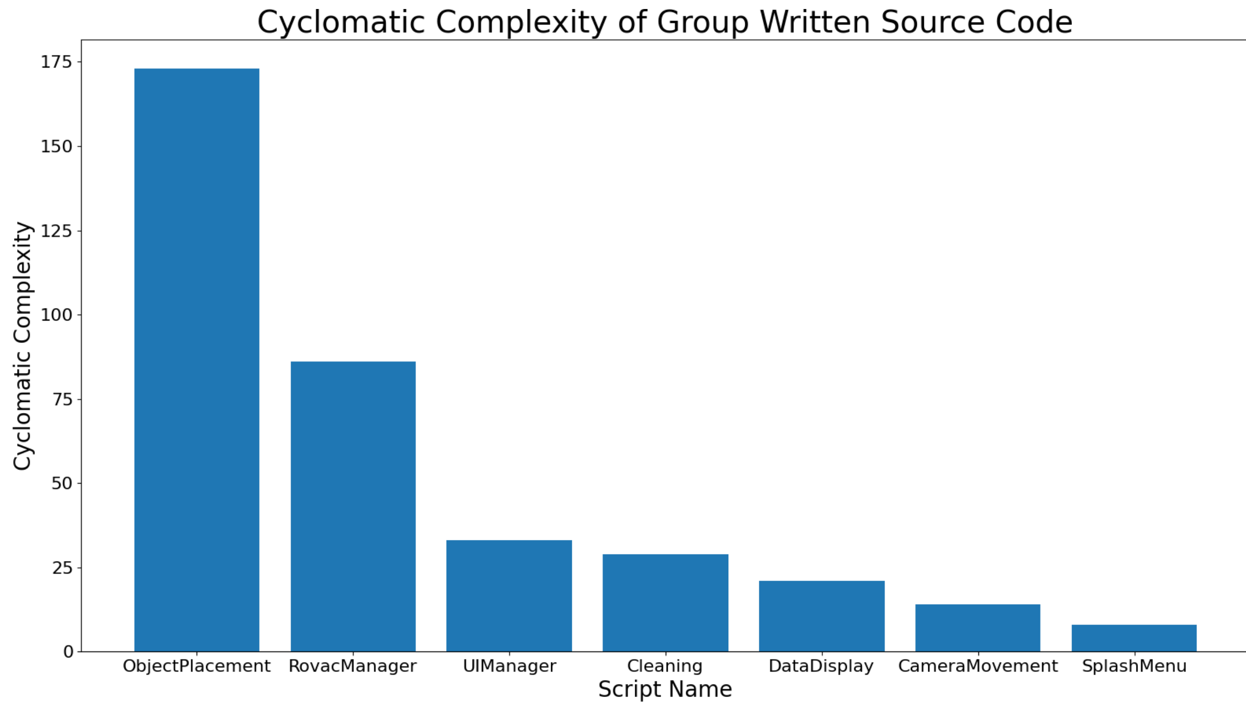
Cyclomatic Complexity

A more useful metric for software analysis is called cyclomatic complexity. This metric is also calculated with the Visual Studio metric analyzer. The official documentation for the Visual Studio code metrics values[4] notes that cyclomatic complexity is a measure of the structural complexity of the code. It comes from calculating the number of different code paths in the flow of the program. Programs with

complex flow require more testing to achieve good code coverage, and is less maintainable as a result.

To help contextualize the values of cyclomatic complexity, program control flow must be understood. Control flow is the order that instructions in a program are executed by the computer. This flow becomes more complex as a result of the inclusion of conditional statements or branches in the code. In the case of cyclomatic complexity, if the source code has no conditionals or branches then the complexity value would be 1. If a single *IF* statement was introduced to the code, then there would be two possible paths of code execution, one where the *IF* is true and one where it is false, which raises the complexity value to 2. With this in mind, it is obvious why longer codes would often have a higher cyclomatic complexity value due to the presence of more conditionals and branches. This would also lead to an increase in the difficulty of testing such programs since ideally, you want to test every possible flow path of the code. Testing such codes can end up taking thousands of test cases, so reducing that number in any way possible is strongly desired. Even in our project, with a cyclomatic complexity of 365, we had 42 test cases, which arguably still was not enough!

In the Visual Studio code metric results, cyclomatic complexity is given as a number value, where lower values represent a less complex code. For this metric, only the group written code will be analyzed, but it is interesting to note that the highest cyclomatic complexity value was for Unity.Timeline.Editor with a value of 8,164 for its 38,794 lines of source code. Larger programs are bound to be more complex, but the degree to which this is true can be visualized with this metric.



For the group-written source code, the highest cyclomatic complexity value was only 173 for the ObjectPlacement script. The cyclomatic complexity values of the group written code happen to be directly proportional to the lines of source code, with less complexity for less code. This is not the case with the Unity code, as the highest cyclomatic complexity value is held by a script that is not the longest, and some other scripts are more complex than longer counterparts. It might seem strange that the longer scripts are sometimes measured to be less complex according to this metric, but it can be explained when thinking about a specific simple example. Let us say we created a program to take a letter input from the user, and output the next letter in the alphabet (given a, it prints b). We might have a cyclomatic complexity value of at least 26, one for each of the *IF* statements that check the input. Since each input would have a unique output, we need at least one different path of execution for each input that will lead to its corresponding output. Say this program is arbitrarily 50 lines long, to account

for asking the user what letter they want to use, empty lines, and comments on the code. Then let us have another program that causes a box with predefined dimensions to bounce up in the air at a certain speed and then fall back down. Using the features provided by Unity, this code can easily be written in less than 10 lines of code, despite it being far more complex to us, and it will have a cyclomatic complexity of 1. These same principles can be scaled up to the tens of thousands of lines of code making up some of the scripts found in Unity. Just because the code only has 1 possible result, it does not mean that it is less complex than the code that has 26 results.

The cyclomatic complexity metric provides valuable insight into the maintainability and testability of a program. In a perfect world, you would test the code once and know that it works for every scenario, but this is clearly impossible in reality. The cyclomatic complexity value can be used to pinpoint which scripts will be the most difficult to fully test, and those scripts can be targeted for revisions to attempt to decrease this value by making more efficient code, resulting in fewer tests being required later on.

Depth of Inheritance

Depth of inheritance is an indication of the number of classes inheriting from each other, back to the base class. A higher number for this metric is bad because it means that a lot of classes are inheriting from each other, and more inheritance means that more consequences will result if a part of the code is changed. The group written project code has a depth of inheritance of 5, meaning that the longest path from one class to the base class is 5 inheritances. For the furthest child class, for example, it has likely inherited a number of different methods from parent classes, and it will be difficult

to predict its behavior. This unpredictability leads to increased complexity and a higher probability of errors in the code. In some cases, a high number for this metric can be a good thing as it indicates that there is more opportunity for code reuse, but this still comes with the drawback of complexity. We can understand the impact of this kind of inheritance when we think of the code as a tree. If the root, the base class, is changed or “shaken”, then all of its children classes, or leaves, are also shaken. It can be difficult to know how much they will be shaken, increasingly so for longer chains of inheritance. On a smaller scale, such as with shorter code written by one person or a small group of people, the complexity can be accounted for due to there not being as much code to understand in order to predict its behavior, but with millions of lines of code and hundreds of classes, it is unrealistic to expect everyone working on the project to understand the entire path of inheritance and the characteristics of each class.

Class Coupling

Similar to depth of inheritance, class coupling is a software metric that is used to measure the coupling between classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration[5]. Where depth of inheritance is the measure of direct inheritance between classes, class coupling is a measure of the usage of classes within other classes. The coupling for the group written project code is 54, which means that in the code, 54 references are made to classes outside of that file. This metric only measures each class once even if the class is used more than once. Examining this metric allows you to see the max number for a desired class and inspect it individually member by member. Class coupling can

serve as a predictor of software failure due to the number of interdependencies on other types, so it is useful to be able to pick apart which sections of the code have the most dependencies and work to reduce those dependencies in order to decrease code coupling.

Discussion

After taking a look at the roVac simulation project from a new perspective provided by its software metrics, we can begin to formulate improvements that can be made to the software in the future in order to enhance its maintainability and functionality.

The lines of code metric that was discussed can only really tell us what sections of the codebase need the most attention to have sufficient quality assurance. Other than simply showing where the highest quantity of code is housed, not much other useful information can be gleaned from this metric.

Cyclomatic complexity provides more insight into possible directions for future improvements. By knowing how many paths of execution exist for each given section of our software, we can attempt to reduce the number of paths by making the code flow more coherently. This could include downsizing the amount of branching that occurs, removing unnecessary *IF* statements and clauses or consolidating necessary ones, or functionizing pieces of code that are frequently repeated. These changes would reduce the amount of testing that needs to be done when releasing new software versions, as well as reduce the likelihood of untraceable bugs resulting from the many different execution paths that would need to be checked over.

Similarly, the depth of inheritance metric gives another view of the complexity of our code by showing how interconnected each of the classes is within the software. We can use this to find deeply seeded classes with long chains of inheritance and follow them to their roots searching for links in the chain that may be able to be removed. If classes can be combined or removed in a way that avoids further confusion, then the

readability of the software will vastly improve, and changes to the code will be able to be made more safely with less worry of unintended side effects further down the inheritance tree.

Class coupling provides an extension to the information that depth of inheritance provided to us. It gives more details regarding the interconnection of classes through their sharing of parameters, variables, return types, method calls, and other pieces of information. We can use this metric to find which classes are most closely connected and attempt to separate them to support the encapsulation and independence of parts of the software. This also allows for changes to the code to be made due to decreasing the vulnerability of the code to side effects and cascading consequences.

All of these metrics can be used to improve our code in a number of ways, and then can be again used in the same process once the next round of improvements is made. These are only a fraction of all of the different software metrics that exist for use in the improvement of productivity and functionality of software projects. The larger the project at hand, the more important it is for software metrics to be considered as the project progresses due to the increased likelihood of mistakes to be made during its development.

Conclusion

Software metrics, when used properly, provide an excellent way for programmers to evaluate their projects and programs, and work on improvements to reduce complexity and inefficiency within them. It is interesting the insight into a project that their analysis provides, and I think that our project could see a number of improvements if refactored using the knowledge gained from understanding the metrics. In the future, I will definitely be taking a look at the software metrics for any code that I write, and aiming to improve my projects using what I have learned about them.

Appendix

[1] Wikimedia Foundation. (2022, April 1). Unity (Game Engine). Wikipedia. Retrieved April 4, 2022, from https://en.wikipedia.org/wiki/Unity_(game_engine)

[2] Wikimedia Foundation. (2021, December 28). Software metric. Wikipedia. Retrieved 2022, from https://en.wikipedia.org/wiki/Software_metric

[3] Desjardins, J. (2017, February 9). Here's how many millions of lines of code it takes to run different software. Business Insider. Retrieved 2022, from https://www.businessinsider.com/how-many-lines-of-code-it-takes-to-run-different-software-2017-2

[4, 5]]Mike Jones, "Calculate Code Metrics," Calculate code metrics - Visual Studio (Windows) | Microsoft Docs, August 5, 2021, https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022.