Examining Software Design Projects in a First-Year Engineering Course Through Different Complexity Measures

Laine E. Rumreich
Department of Engineering Education
The Ohio State University
Columbus, OH, USA
rumriech.1@osu.edu

Krista M. Kecskemety
Department of Engineering Education
The Ohio State University
Columbus, OH, USA
kecskemety.1@osu.edu

***Abstract*—This Innovative Practice Work in Progress paper examines student game development software projects that replaced a prior end of term project at a large Midwestern university. The use of end of term projects for introductory software courses can increase student engagement and provide a way to summarize and synthesize topics learned in the course. Previous research on a random sample of data has shown that game design projects require significantly higher student effort in the area of programming fundamentals (input/output, repetition structures, selection structures) when compared to prior end of term projects. This current work uses a custom automated Python script to analyze a much larger data set, and examines how game design projects differ from a previous project option in the areas of programming fundamentals and complexity, where complexity is measured using Lines of Code, Cyclomatic Complexity, Halstead Complexity, and Maintainability Index. The results show that when examining these metrics, the game project requires significantly higher effort as compared to the prior end of term project. This preliminary work will lead to a more in-depth analysis of the results from the automated script, as well as an analysis of additional end of course projects in first-year engineering courses.**

Keywords—software design project, first-year engineering, software complexity measures, Cyclomatic Complexity, Halstead Complexity, Maintainability Index

I. Introduction

Developing software for computer games as a teaching tool in programming classrooms is increasing in popularity both in high school programming courses and in engineering classrooms [1-5]. Research has shown that game design projects as part of a software engineering curriculum can lead to increased class participation and performance [6]. Game modding, modification of existing computer games, has also been shown to benefit software engineering students by increasing their motivation to acquire and practice specific computer science skills [7]. Part of this shift towards game development in the classroom is due to the flexibility and creativity of open-ended game development projects, as well as the ability to promote learning in higher levels of Boom's taxonomy [8]. Game development is also a way for students to understand and showcase their programming knowledge in a tangible and visual way. Students are more likely to learn programming techniques when programming a game because they are likely to be interested in the subject matter, and the visual component of the project allows students to see mistakes in their code as manifested in the resultant graphics [9]. Another reason open-ended game development projects are successful in teaching students is that they allow students to choose their own project. Choice can be used as a way to increase student motivation and interest [10]. This added element of choice also allows for creativity in selecting a topic and requires initial project planning to generate unique ideas.

At a large Midwestern university, first-year engineering (FYE) students learn introductory programming during their first semester engineering course. This course covers programming languages MATLAB, C, and C++. The course culminates in a software design project where students can assimilate and showcase their programming knowledge from the semester. The prior student software design project involved using an infrared (IR) sensor to compute the modulation frequency of an IR light source. Recently, faculty replaced this project with a game design project. There were no substantive differences between the two projects in terms of requirements and difficulty for students with little to no prior programming knowledge. This study focuses on comparing the programming content of the student software produced from these projects. This work in progress study works towards answering the research question: *Does using a game software design project with FYE students increase the level of programming complexity and the amount of practiced programming fundamentals?* This is one piece of the larger research on the impact of game development on the knowledge retention of programming fundamentals beyond FYE courses.

II. Background

A. Course Background

The FYE course that is the focus of this work centers around problem solving through computer programming. The course topics covered in this course are as follows.

978-1-7281-1746-1/19/$31.00 ©2019 IEEE

- MATLAB topics
- C/C++ Input/Output
- C/C++ File Input/Output
- C/C++ Pointers
- C/C++ Repetition
- C/C++ Selection Structures
- C/C++ Functions
- C/C++ Classes Structures

B. Past Work

Previous work has studied student perceptions of the end of term projects [11, 12] and concluded that the game project is more engaging than the IR project. Another previous work analyzed a sample set of end of term student projects to determine if the projects met more learning objectives and promoted deeper understanding and learning [13]. The results of this preliminary work indicate that the game project increases overall programming fundamentals and effort. However, the previous work was limited in the metrics used and did not examine various software complexity metrics.

C. Description of Metrics Used in this Study

Source code can be analyzed in a number of ways to determine the programming fundamentals present, complexity, and control flow. The end of term project in the FYE course that is the focus of this study allows students to synthesize programming elements learned throughout the course. This study uses various programming metrics to quantify the quality of software design projects in terms of both programming fundamentals learned and overall complexity.

a) Programming Fundamentals: In this study, the techniques used to identify and quantify the programming fundamentals learned throughout the first-year engineering course involve counting the number of comments, Input/Output statements, repetition structures, selection structures, and user-defined functions.

b) Complexity: The four metrics used in this study to measure the complexity of source code files are Lines of Code, Cyclomatic complexity, Halstead complexity, and Maintainability Index. Source Lines of Code counts the number of lines of code, and provides a limited measure of a program's complexity [14].

Cyclomatic complexity is a common metric of code complexity that measures the number of linearly independent paths through the program, resulting in a normalized complexity value score [15]. A program with a high Cyclomatic complexity score is typically more difficult to adequately test and is more prone to having undiscovered bugs because of the number of possible paths through the program [16]. Cyclomatic complexity is limited in that it can only be used to measure control flow and has no dependence on the data flow of the program. This means that a program with no selection structures would be scored as very simple to test, independent of the content or number of lines of code.

Halstead complexity is another commonly used software metric [17], and it measures the number of distinct operators, number of distinct operands, and the total of both in the source code. From these values, program difficulty, effort, estimated time required to program, and estimated number of delivered bugs are computed. Limitations of Halstead complexity are that it emphasizes computational complexity rather than control flow [16].

The Maintainability Index metric is a combination of the Cyclomatic complexity and Halstead complexity metrics, and is designed to give a better view of the overall complexity of the program [18].

These complexity metrics cannot directly measure what a human would define as complexity, especially in terms of readability and understandability. However, using multiple metrics helps reduce this problem because it gives a more robust view of the overall complexity of the program. Another limitation of these complexity measures is that the correct implementation of these metrics is vague, so different implementations often result in different measured complexity values. However, since only one implementation was used in this study, and the values are relative to each other and not absolute data, this limitation is not as relevant.

III. Methods

A total of 227 student projects in PDF format were collected for this study. While direct text source code would have been easier to process, the method students submitted the final code varied and the most common was as a PDF and thus that became the standard used in this study. In order to anonymize the files for processing, sections of the PDF were redacted in they contained identifying information. Of the student projects, 39 are game design programs, and the rest are IR receiver programs completed during a similar timeframe. This imbalance was due to the game project being piloted in a few sections of the course, while the majority of the sections continued to complete the IR project. In order to run an automated analysis on these projects, they were converted to text files. This process resulted in damaged code, meaning that added, removed, or malformed characters resulted in code that could not be run using a compiler.

For this study, a Python script with two major components was written to run an automated analysis of the student design projects. The first component repairs damaged code from the text files to make it conform to the standards of a compiler and remove unwanted characters, line breaks, and spaces when possible. The second component analyzes the repaired source code files for programming fundamentals and complexity. To search for programming fundamentals, the input source code was parsed to search for key sets of strings such as '//', 'for', 'while', 'if', and '{' to count particular components and identify nested structures. A similar parsing strategy was used to compute complexity measures.

This Python script was verified for accuracy in two ways. The first verified the accuracy of the programming

fundamentals, and it was done by randomly selecting 20 projects and comparing the results of manually counting values with the results generated from the automated analysis. The manually counted results were verified by two independent researchers. The second verification method was used for the computation of complexity metrics, and it involved running sample code with known complexity values through the script to validate the output.

IV. INITIAL FINDINGS

To begin this analysis, a sample of the source code submissions consisting of 227 of the full 425 source code submission dataset was analyzed for programming fundamentals and complexity. The submissions were processed by the automated analysis script, producing a spreadsheet containing the frequency of various programming fundamentals and the complexity of each submission. The resulting data were then consolidated into the categories of Input/Output, Selection Structures, Repetition Structures, and the four categories of complexity discussed in the Introduction of this paper. With the data from these categories, box-and-whisker plots (Fig. 1-8) were constructed to illustrate the differences in the number of programming fundamentals and the complexity of the game design projects compared to the IR receiver projects. To test the relationship between the project outputs, a Mann-Whitney U test was performed for each metric. Results indicate a significant difference for each metric ($p < .00001$).

Students who completed the game design projects, on average, included a greater number of programming fundamentals in their programs, and wrote programs with higher complexity than the IR receiver counterparts. Due to the median differences displayed on the box-and-whisker plots, it can be assumed that students are gaining additional practice using the defined fundamentals with the game project as opposed to the IR project. In addition, due to the difference in Cyclomatic complexity, Halstead complexities, and Maintainability Index median values, it can be assumed that students are writing more complex code with the game project compared to the IR project. Due to the inverse relationship between code complexity and maintainability in the measures used in this study, it can be assumed that students are writing less maintainable code with the game project compared to the IR project. A detailed discussion of each measured element follows.

A. Lines of Code and Comments

Fig. 1 shows that the game projects have a greater number of lines of code than the IR receiver projects. This is likely because developing code for a unique game would require more code to initialize the graphics, accommodate for user input, and accomplish a unique task. It is clear that the game projects have more lines of comment blocks than the IR receiver projects. This is reasonable because the number of lines of comments tends to increase as the number of lines of code increases.

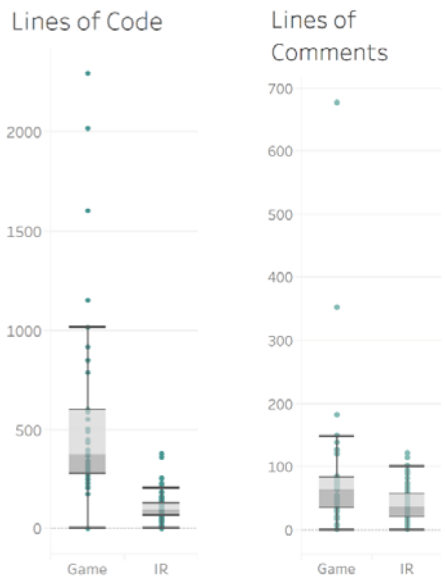

Fig. 1. Source Lines of Code

B. Selection, Repetition, and I/O Elements

The number of selection structures, repetition structures, and Input/Output elements are noticeably higher in the game project compared to the IR receiver counterparts, as displayed in Fig. 2. This trend is related to the number of lines of code and the overall complexity of the game projects. The increased number of repetition structures may relate to the repetitive nature of many games, such as tic-tac-toe, Tetris, or card games. Similarly, the higher number of selection structures may have to do with the interactive element of the game projects as well as the number of possible user paths to follow through the game. The number of calls to Input/Output functions is also likely related to the complex graphics and interactive elements of the game project.

In contrast, the IR receiver project requires students to write a program to read a signal and display the corresponding output. This program would likely follow a more straightforward and uniform path, which would result in lower values for repetition, selection, and Input/Output.

C. Cyclomatic Complexity

Based on Fig. 3, it can be seen that the Cyclomatic Complexity is, on average, higher for the game design projects than for the IR receiver projects. This is likely due to the fact that games tend to have more possible paths of outcomes for the user. This corresponds to a higher number of linearly independent paths in the code, which increases the Cyclomatic Complexity. In contrast, the IR receiver project requires fewer paths because it requires the same type of result each time it is run.

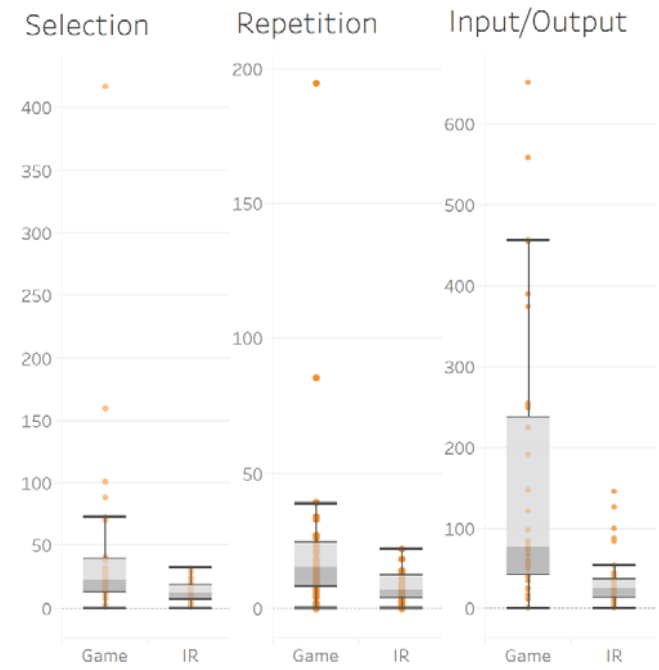

Fig. 2. Numbers of programming fundamentals

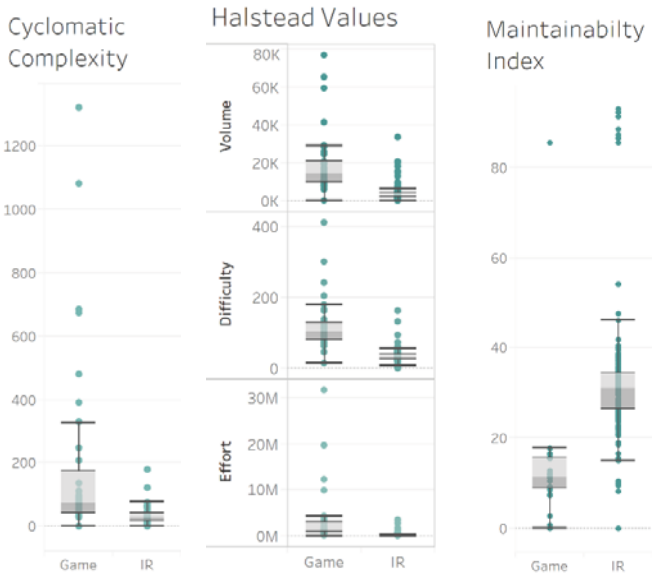

Fig. 3. Measures of Complexity

D. Halstead Complexity Measures

Also in Fig. 3, the Halstead complexity is, on average, higher for the game design projects than for the IR receiver projects. The Halstead Complexity for this study was measured in terms of volume, difficulty, and effort, which measure the amount of code, how difficult it would be to write or understand the code, and the amount of effort or time it would take to write it, respectively. This increase is likely due to the fact that games tend to be longer, and they tend to have more variables, Input/Output statements, function calls, and structures. Halstead metrics are directly based on the number of operands and operators, so all of these factors contribute to higher values for each metric.

E. Maintainability Index

The Maintainability Index value shown in Fig. 3 is on average lower for the game design projects than for the IR receiver projects. The Maintainability Index measures how maintainable the source code is, and it is inversely related to the Cyclomatic complexity and Halstead volume of the code. Since the game design projects tend to be more complex, it follows that they are also less maintainable and therefore likely also harder to debug and understand.

V. Conclusions and Future Research

The results of this study suggest that as compared to the IR project, the game projects meet the goals of increasing overall programming practice, effort, and practice with programming fundamentals when examining the number of instances of fundamentals and source code complexity. The projects were not substantially different in terms of requirements, making these findings noteworthy. This result, combined with the increased student engagement determined in previous studies, helps support the full implementation of the game project throughout these FYE courses. A more thorough analysis will be conducted with the full 425 source code submission dataset. This analysis will involve an automated script that is modified to parse source code in other languages, which is necessary to analyze the remaining data. It will use a modified process for identifying the programming elements. Researchers are working on supplementing the existing software script to analyze the additional source code submissions.

This work could be expanded by adding two additional elements to the automated analysis of the source code. The first element is to record the nesting depth of the source code as an additional complexity metric. Nesting depth is the maximal number of repetition or selection structures that are nested within each other. Although simple, this metric relates to the context that a reader would have to understand to comprehend the function. The second element that could be added to expand this study would be to automate the identification of poor programming practices in the source code. This could be done by defining a few poor practices, such as a large number of function arguments, including comments that are full lines of code that were likely meant to be deleted, or frequently repeating code. The program could then search for the defined patterns in the source code to identify the defined poor practices. Future research could include student performance, such as grade comparisons for the projects, and the collection of student feedback such as perceived usefulness of game design based projects for learning software engineering concepts.

REFERENCES

[1] B. Maxim, "Serious games as software engineering capstone projects," Proceedings of the 2008 American Society of Engineering Education Annual Conference, Pittsburgh, Pennsylvania, June 2008.

[2] J.K. Estell, "Writing card games: an early excursion into software engineering principles," Proceedings of the 2005 American Society of Engineering Education Annual Conference, Portland, Oregon, June 2005.

[3] E. Helber, M. Brockman, and R. Kajfez, "Gaming with LabVIEW: an attempt at a novel software design project for first-year engineers," First Year Engineering Experience Conference, August 7-8, College Station, TX, 2014.

[4] T.R. Hamrick, and R.A. Hensel, "Putting the fun in programming fundamentals - robots make programs tangible," Proceedings of the 2013 American Society of Engineering Education Annual Conference, Atlanta, Georgia, June 2013.

[5] M.K. Thomas, X. Ge, B.A. Greene, B. A., "Fostering 21st century skill development by engaging students in authentic game design projects in a high school computer programming class," Journal of Educational Computing Research, 44(4), 2011, pp 391–408.

[6] K. Claypool, M. Claypool, "Teaching software engineering through game design," In Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '05). ACM, New York, NY, USA, 2005, pp 123-127.

[7] M.S. El-Nasr, B. Smith, "Learning through game modding," Computer Entertainment 4, 1, Article 7, January 2006.

[8] L. Anderson, and D. Krathwohl, A Taxonomy for Learning, Teaching, and Assessing. Addison Wesley Longman, Inc., Illinois, 2001.

[9] S. Leutenegger, and J. Edgington, "A games first approach to teaching introductory programming," Proceedings of the 38th SIGCSE technical symposium on Computer science education, March 07-11, 2007, Covington, Kentucky, USA

[10] T. Shepard, "Implementing first-year design projects with the power of choice," Proceedings of the 2013 American Society of Engineering Education Annual Conference, Atlanta, Georgia, June 2013.

[11] K.M. Kecskemety, A.B. Drown, L.N. Corrigan, "Examining software design projects in a first-year engineering course: how assigning an open-ended game project impacts student experience," 124th American Society for Engineering Education Annual Conference & Exposition, June 25-28, Columbus, OH, 2017.

[12] K.M. Kecskemety, L.N. Corrigan, "End of semester software problem solving and design projects in a first-year engineering course," Frontiers in Education, October 18-15, Indianapolis, IN, 2017

[13] K.M. Kecskemety, Z. Dix, B. Kott, "Examining software design projects in a first-year engineering course: the impact of the project type on programming complexity and programming." 2018, pp 1-5.

[14] K. Bhatt, V. Tarey, and P. Patel, "Analysis of source lines of code (SLOC) metric," International Journal of Emerging Technology and Advanced Engineering, 2(5), May 2012.

[15] T.J. McCabe, "A complexity measure," IEEE Transactions on Software Engineering, SE-2(4), December, 1976.

[16] T. Yahya, A. Mohammed, and A. Bassam, "The correlation among software complexity metrics with case study," International Journal of Advanced Computer Research, 4(2), 15 June 2014.

[17] M.H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc, New York, NY, 1977.

[18] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," Computer, vol. 27 (8), pp, 44-49, 1994.