Examining Software Design Projects in a First-Year Engineering Course:

The Impact of the Project Type on Programming Complexity and Programming Fundamentals Required

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

Zachary Dix
Department of Engineering Education
The Ohio State University
Columbus, OH, USA
dix.47@osu.edu

Benjamin Kott
Department of Engineering Education
The Ohio State University
Columbus, OH, USA
kott.8@osu.edu

***Abstract*—This Innovative Practice work-in-progress paper examines a game software design project implemented at a large Midwestern university to replace prior end of term projects. 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 has shown student's perceptions of the game project to be positive. This current work examines a sample set of programming source code submissions for the Source Lines of Code metric and counts of the instances of certain programming fundamentals (input/output, repetition structures, selection structures). The results show that the game projects had significantly higher effort required when examining these metrics compared to the prior end of term projects. This preliminary work will lead to additional analysis of the full set of source code submissions, as well as, different metrics to measure complexity of the submissions.**

Keywords— software design project, source code analysis, first-year engineering

I. Introduction

Using game software development to engage students and practice programming fundamentals is something that has increased in popularity in engineering and computer science classrooms, for both upper level courses [1] and first-year engineering courses [2-4]. Games can be used to teach students programming skills similar to how they have been used to teach students about reading or math [5]. Games also allow creativity in software design and can promote learning in higher levels of Boom's taxonomy [6]. One of the other advantages of using a game for an end of term project is that it lends itself to open-ended problems and giving students a choice in their project. Choice can be used as a way to increase student motivation and interest [7].

At a large Midwestern university, first-year engineering students learn introductory programming during their first semester engineering course. The programming languages include MATLAB programming and C/C++ programming depending on the course sequence. This first-year engineering course concludes with a software design project. Prior student software design projects have included sensing an infrared (IR) sensor to compute the frequency and programming a train track. Over the last few years, faculty have implemented and scaled up a game software design project for all students. This current study focuses on the programming content of the software that was developed in these various end of term projects. This work in progress study is the first step to answering the research question: *Does using a game software design project with first-year engineering students increase the level of programming complexity required and the amount of practiced programming fundamentals?* This is a small part of the larger research question centered on the impact of these projects on the knowledge retention of programming fundamentals beyond the first-year engineering courses.

II. Background

A. Course Background

This first-year engineering course focuses on problem solving through computer programming. The course topics that concentrate on programming are included in Table 1. The standard first-year engineering course covers MATLAB programming and the honors first-year engineering course covers the same topics in MATLAB and also topics in C/C++ programming.

TABLE I. Course Programming Topics

Standard Course	Honors Course
MATLAB Script Files	MATLAB topics
MATLAB Input/Output	C/C++ Input/Output
MATLAB File Input	C/C++ File Input/Output
MATLAB Pointers	C/C++ Pointers
MATLAB Repetition	C/C++ Repetition
MATLAB Selection Structures	C/C++ Selection Structures
MATLAB Functions	C/C++ Functions
	C/C++ Classes and Structs

B. Our Past Work

Previous work has looked at student perceptions of the end of term projects [8,9]. Based on these studies students found the

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

game projects more engaging than the train/IR projects. Additionally, students mentioned that they could engage more in creativity with the game projects. However, there had yet to be an analysis of the student products developed from these projects to see if the projects met more learning objectives and promoted deeper understanding and learning.

C. Source Code Metrics

There are many metrics commonly used to analyze source code. One of the most common is the Source Lines of Code [10] where the number of lines of code are counted. This metric provides a limited measure of a program's complexity. For example, if comparing two projects of different length, A and B, it can be difficult to claim a reasonable distinction in complexity if the difference in length is insignificant. However, if the difference in length between projects A and B is significant, then the longer program is generally notably more complex. The Source Lines of Code metric directly relates to the amount of effort that was needed to write the code, however this does not always equate to highly efficient code since one software developer could do something in a more efficient way with less lines than another [10].

The Halstead complexity measure is another commonly used software metric [11]. This measure, classified a composite complexity metric, looks at the total number of operators and operands in the source code and also the distinct number of operators and operands. From these values, various measures can be computed including the program difficulty, effort, and time required to program. It can also be used to compute a reasonable expectation of the bug density of the program. The Halstead complexity measure's is limited in that the complexity computed is dependent on the program's data and not its control flow [12].

Another common metric is Cyclomatic complexity [13]. This measure of complexity considers the maximum number of linearly independent paths through a program, taking into account the number of repetition paths and selection structures in its source code, and gives a score relating the program's complexity. A program which is measured to have high Cylcomatic complexity is usually more difficult to adequately test and is more prone to having undiscovered bugs than a simple method with a lower number of paths [12]. The limitation of using Cyclomatic complexity is the inverse of that of the Halstead complexity – it is used to measure control flow only and has no dependence on the data flow of the program. Thus, a sequential-execution program consisting of thousands of lines of code measures the same as a single code line according to Cyclomatic complexity. In practice, Hallstead complexity and Cyclomatic complexity metric are usually used together. Hallstead complexity metric is used to relate the complexity from the data flow, while Cyclomatic complexity metric measures the control flow. However, Graylin et al. suggested that the Cyclomatic complexity metric provides no more information on complexity than the Source Lines of Code metric [14].

All of these measures could be used to analyze the source code submissions to compare the different projects.

D. Importance of Practice in Programming Education

Practice is commonly cited as something that is necessary and useful when teaching programming. Berglund and Eckerdal examined how theory and practice are intertwined when teaching programming [15]. They found that students use practice to determine gaps in their knowledge which they then can fill using theory. Additionally, Höök and Eckerdal specifically looked at the impact of practice and time spent coding and exam grades [16]. They found that high amounts of time coding correlated with high exam grades. This helps demonstrate that practice helps support learning programming and a metric like Source Lines of Code could be an appropriate metric to demonstrate increased practice and opportunities for learning.

E. Description of Metrics Used in this Study

In teaching the first-year engineering course the motivation to have a final course project was to allow students to synthesize elements they had learned throughout the course, gain additional practice with programming fundamentals that had been taught, troubleshoot a large program, learn new commands, and demonstrate good programming practices. Because one of the intended elements of this project was for students to gain additional practice with programming fundamentals, this lead to an analysis that counted instances of each programming fundamental. Therefore, through this analysis we can determine which project produced more practice with programming fundamentals and also which project required the most effort based on the Source Lines of Code metric. Future studies will include additional complexity measures.

III. INITIAL FINDINGS

To begin this analysis, a sample of the source code submissions of all 4 projects (10 of each, 40 total) have been analyzed to look for specific programming elements out of the full 425 source code submission dataset. To do this, each submission was examined to look for items like the amount of 'for' loops or the number of lines of code. The source code data was analyzed individually by searching for each feature in the source code. Each desired operation was recorded by hand. In order to consider the total number of comments, '%' was searched for in the MATLAB submissions, while '//' was searched for in the C/C++ submissions. Each block of code containing '%' or '//', that was not used for an operation, was recorded as a comment. Finally, to count the total number of lines of code used, every line in the program was recorded and the previously-recorded number of comments was subtracted from this number.

After completing this process with many individual operations/commands, the data was combined into the following themes: Input/Output, Conditional/Selection Structures, and Repetition Structures. With this raw data, the minimum, first quartile, median, third quartile, and maximum

values were found for each category. Box-and-whisker plots (Figures 1-5) were then constructed using these values to illustrate the differences in the number of programming fundamentals found across the four software design projects. Students who completed the C/C++ honors game project wrote, on average, the highest number of total lines of code and used the highest number of input/output statements and repetition elements. The standard MATLAB game project followed the C/C++ honors game project in each of these categories. Students who were assigned the standard MATLAB train and C/C++ honors IR projects wrote, on average, much shorter codes than either of the game projects. It followed that these projects were observed to require a smaller amount of each type of programming fundamental considered in this study. Due to the median differences displayed on the box-and-whisker plots, it can be assumed that students are gaining additional practice using these fundamentals with C/C++ or MATLAB with the game projects as opposed to the MATLAB train and C/C++ honors IR projects. A detailed discussion of each element is given below.

A. Overall Code and Comments

Based on Fig. 1 it is seen that the game projects have a greater amount of code compared to the IR receiver and train projects, with the C/C++ honors game project incorporating more code than the MATLAB game project. This is reasonable as developing code for a unique game would lead to more code being used to accomplish the task at hand. It also is understandable that the honors game project would incorporate the most code, too, as C/C++ as a language contains different elements like declaring variables compared to what is required with MATLAB. The IR receiver and train projects are shown to use a similar amount of code, most likely due to the simpler nature of these projects even though they used different programming languages.

Transitioning to Fig. 2, it is seen that the two game projects, again, use the most comments; followed by the train and IR receiver projects. This is reasonable as more comments would be necessary to explain the workings of uniquely different game project submissions, compared to the similar code structures of the IR receiver and train projects. A difference is apparent though, with the MATLAB game and train projects utilizing more comments than the C/C++ honors game and IR project, respectively. As such, it is noted that the two projects incorporating MATLAB tended to use more comments. It is not clear what caused this difference. It could be that commenting was stressed more in the standard course, or students felt that they needed to explain their code more thoroughly in that course compared to the honors course. While the number of comments does not directly correlate to overall complexity, it does provide a sense of what students felt they needed to explain from their code.

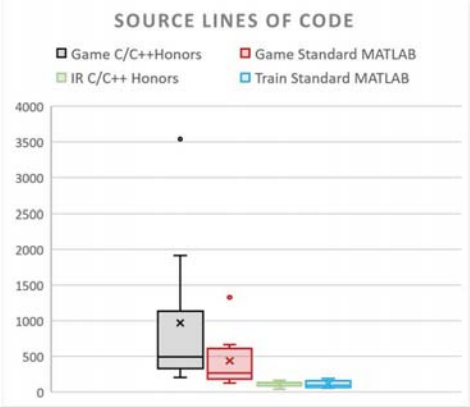

Fig. 1. Source Lines of Code for all four projects.

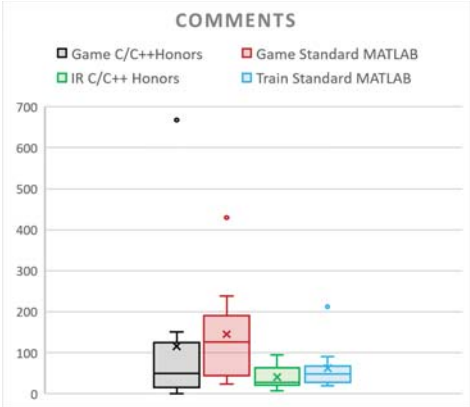

Fig. 2. Number of comment statements for all four projects.

B. Repetition Elements

It is noticeable that the two game projects have the highest instances of repetition in Fig. 3, with the honors C/C++ having the most. Students who completed the MATLAB game project used a comparable median number of repetition elements to those who completed the C/C++ honors game project. This is followed by the IR receiver project and the train project. This trend is reasonable as all projects require repetition of some sort, but the IR and train projects require less repetition compared to the game projects. This may be due to the game projects being interactive as well, meaning that more repetition may be needed, such as in the game blackjack. Blackjack is fairly simple to understand and quite repetitive, with the purpose of the game to rack up enough points in a given hand to be close to 21. In this case, a player can repeatedly gain a card until they either decide to stop or bust. This repetitive nature, too, will continue indefinitely until the player wishes to quit the game. This type of repetition was common in the game programs and would not have been seen in the train and IR projects.

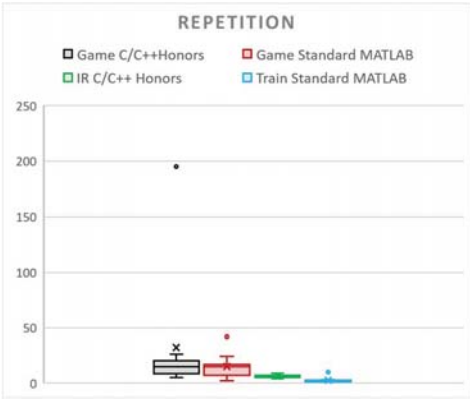

Fig. 3. Number of repetition commands for all four projects.

C. Conditional/Selection Structures

In Fig. 4 it is clear that the four projects follow a general trend of the C/C++ honors game project incorporating the most conditional/selection functions followed by the MATLAB game project, the IR receiver project, and the train project. This is reasonable as interactive games require different outcomes for different possible decisions. In contrast, the IR receiver project only required students to code for reading in a signal and giving the appropriate output, thus limiting the number of outcomes that could result from a given action. The same was true for the train project; the program only had to dictate whether or not to open a gate or change the speed of the train.

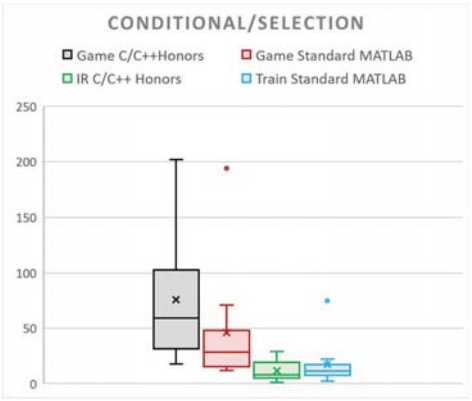

Fig. 4. Number of conditional/selection commands for all four projects.

D. Input/Output

As explained in the previous sections, games are, by nature, interactive, therefore we would expect that they would require substantial amounts of input and output commands. Based on Fig. 5 it is clear that the standard MATLAB and honors C/C++ game projects provided students more opportunity to use input and output commands. The honors C/C++ game project has the highest instance of input and output commands followed by the standard MATLAB game project. This makes sense since most games require user input, as well as instructions and other visuals that would need to be written to the screen to make the game playable. The input/output extent of the IR receiver project, on the other hand, was to read in an infrared signal and output the determined frequency to a display. The train project read in sensor data but required no user communication with input and output. Therefore, it is not a surprising result that this project was found to use the lowest number of input and output commands. Another thing to note is that the overall number of input and output elements is much higher than the other programming fundamentals counted in this study.

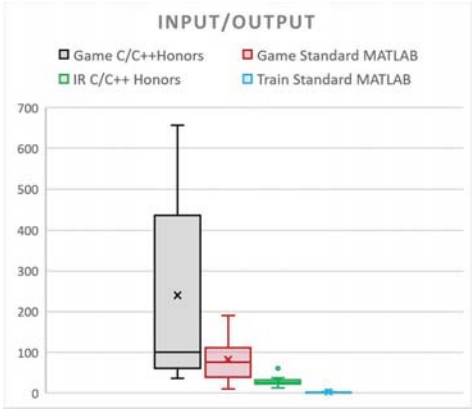

Fig. 5. Number of input and output statements for all four projects.

IV. Conclusions and Future Research

The results of this study suggest that the game projects met the goals of increasing overall programming practice, effort, and practice with programming fundamentals when looking solely at the number of these instances and source code length. This, coupled with the increased student engagement as found in previous studies, helps initially support the full implementation of the game project throughout these first-year engineering courses beyond the pilots. A similar analysis will be conducted with the full 425 source code submission dataset, using a modified process for identifying the programming elements. Researchers are working on developing a software program to analyze each source code submission for these elements. This software program will be verified by two independent researchers for a sample of the submissions.

One element that is missing in this work is identifying poor programming practices in the source code. A separate analysis will go through the submissions to look for elements that are considered poor programming practices (copying and pasting lines of code rather than using a loop, etc.). Additionally, it should be noted that because only a Source Lines of Code metric approach was used, and because the projects were distinct in theme and differed in coding language used, it is somewhat inconclusive whether a difference in overall programming complexity was present across the different project groups in this preliminary analysis. In the larger study we will include the investigation of other source code metrics including Cyclomatic and Halstead's metrics of complexity.

REFERENCES

[1] Maxim, B., "Serious Games as Software Engineering Capstone Projects," Proceedings of the 2008 American Society of Engineering Education Annual Conference, Pittsburgh, Pennsylvania, June 2008.

[2] Estell, J.K., "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] Helber, E., Brockman, M., and Kajfez, R., "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] Hamrick, T.R., and Hensel, R.A., "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] Baibak, T, and Agrawal, R., "Programming Games To Learn Algorithms", Proceedings of the 2007 American Society of Engineering Education Annual Conference, Honolulu, HI, June 2007.

[6] L.W. Anderson, et al., A Taxonomy for Learning, Teaching, and Assessing. Addison Wesley Longman, Inc., Illinois, 2001.

[7] Shepard, T., "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.

[8] Kecskemety, K.M., Drown, A.B., and Corrigan, L.N., "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.

[9] Kecskemety, K.M. and Corrigan, L.N., "End of Semester Software Problem Solving and Design Projects in a First-Year Engineering Course," Frontiers in Education, October 18-15, Indianapolis, IN, 2017.

[10] Bhatt, K., Tarey, V., and Patel, P."Analysis of Source Lines of Code (SLOC) Metric", International Journal of Emerging Technology and Advanced Engineering, 2(5), May 2012.

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

[12] Yahya, T., Mohammed, A., and Bassam, A., "The Correlation among Software Complexity Metrics with Case Study", International Journal of Advanced Computer Research, 4(2), 15 June 2014.

[13] McCabe, T.J. "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2(4), December, 1976.

[14] G. Jay, et al., "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship", J. Software Engineering & Applications, 2009.

[15] Berglund, A. and Eckerdal, A., "Learning Practice and Theory in Progamming Education: Students' Lived Experience", 2015 International Conference on Learning and Teaching in Computing and Engineering, Taipei, Taiwan, 2015.

[16] Höök, L.J. and Eckerdal, A.,"On the Bimodality in an Introductory Programming Course: An Analysis of Student Performance Factors." 2015 International Conference on Learning and Teaching in Computing and Engineering, Taipei, Taiwan, 2015.