Performance Evaluation of a Code Complexity Measurement Tool: An Empirical Approach

Mutiat A. Ogunrinde
Department of Mathematical and Computer Sciences, Fountain University, Osogbo, Nigeria.
Email: Ogunrinde.mutiat@fuo.edu.ng, bogunrinde@gmail.com
and
Solomon O. Akinola
Department of Computer Science, University of Ibadan, Ibadan, Nigeria.
Email: Solom202@yahoo.co.uk

ABSTRACT

Quality is an important goal desired by stakeholders in every Program Code (PC) developed to meet its functional requirements. However, empirical evidence on tools that measure code complexity and predict future maintenance before deployment is a problem. This work evaluates a Complexity Measurement Tool (CMT) and compares its performance with Code Metrics (CM), an equivalent existing complexity tool using Cyclomatic Complexity Value (CCV), Cyclomatic Complexity Category (CCC) and System's Computational Time (SCT). The CMT and Code Metric (CM) were subjected to evaluations by conducting two sets of controlled experiments involving each of 25 Student Programmers (SP) and Professional Programmers (PP). The quality of code artifacts obtained from the experiments was assessed using computed complexity values obtained with CMT and CM. Some other metrics obtained from the experiments were Effort/time range for Coding (EC) and Number of Bugs (NB). Data analysis was carried out using descriptive statistics and correlation, tested at $\alpha_{0.05}$. The mean CCV reported by CMT for SP and PP respectively, were 7.5±0.1 and 9.2±0.1, while for CM were 8.5±0.1 and 9.0±0.1 respectively. The CCC reported by CMT for both SP and PP were 'risk-free' and 'moderate-risk'. The average SCT obtained from CMT and CM for all the codes were 1.0±0.01 and 3.0±0.01 minutes respectively. A strong correlation exists between Line of Code (LoC) and CCV while a weak correlation between NB and CCV as measured by CMT for SP and PP. No CCC was reported by CM; also no CCV was computed by CM for codes with bugs. The included CCC makes CMT result interpretation better. The speed also gave it an upper hand over the code metrics embedded in Visual Studio environment.

Keywords: *Software development, Code quality, Cyclomatic complexity, Software measurement, Maintainability, Programming Code*

African Journal of Computing & ICT Reference Format:

Mutiat A. Ogunrinde and Solomon O. Akinola (2020), Performance Evaluation of a Code Complexity Measurement Tool: An Empirical Approach, *Afr. J. Comp. & ICT*, Vol. 13, No. 3, pp. 55 – 69 .

© Afr. J. Comp. & ICT, September 2020; P-ISSN 2006-1781

I. INTRODUCTION

Programming is the act of producing computer code in a specific programming language. This code, in turn, gives birth to software that targets a specific problem or used in directing the way a computer system works. The principal objective of any system is its quality. Since this is the desire of most stakeholders in the domain, it is therefore important to identify the quality factors and improve them. Program's code quality can be termed as a code characterised with a wide range of attributes that includes ease of maintenance, testability, reusability, difficulty, reliability, interoperability, etc. [3]. Program codes must have high quality to address the business issues in today's market. For that reason, it becomes a focal point for developers of software. Software engineers have concluded that there is a link between unseen attributes of program code such as cost, strength, Line of code (LOC), speed etc. and those attributes that have a direct impact on the stakeholders such as usefulness, quality, complexity, effectiveness, reliability or viability. For instance, a greater number of code lines may prompt more prominent software complexity [14].

Although research on software complexity, software attribute started around the mid-70s, a critical number of issues identified with unpredictability and fundamentally boosting programming efficiency stay to be unravelled. To be sure, the inability of top unpredictability is the reason for a considerable lot of present issues in programming advancement [4]. The main way to study the qualities and advancement is by constant and timely assessment of program code as it progresses [4, 16] and by employing an automated tool to calculate the complexity of the code makes it easier and faster. Complexity Measurement Tool (CMT) is one of the available tools to measure the code complexity in terms of quality; it is developed based on the cyclomatic approach and predicts future maintenance before deployment. This work evaluates the performance of CMT, compared it with an equivalent Code Metric (CM) in the Visual Studio environment and provides empirical evidence for its adoption into the development environment.

1.1 Software Measurement and Quality

Software is a generic term for a collection of instructions that controls output and also dictates for hardware which is the physical component of a computing system. Software is used in virtually all disciplines for various purposes, such as accounting, aviation, medicals, engineering, etc. and upgrades are constantly provided as needs arise which may be as a result of consumers' data, improved coding methods, and software developers' survey or measurement. Program estimation is a significant research subject in Computer Science. It is a basic piece of understanding, controlling, observing, foreseeing and assessing programming advancement and tasks support [8]. Software measurement must be an issue situated procedure that measures, assesses, modifies, and lastly improves the current program for upgraded execution. All through the whole programming items life cycle, quality, power, and yields are assessed utilizing the estimation procedure [17]. Program estimation has become a key part of good programming building practice [5]. As indicated by Srinivasan and Devi [23], a portion of the product building approaches are utilizing the measure and estimation frameworks in everyday exercises for the generation of upgraded programming items, yet a few associations build up their product without impeccable estimation framework [13]. Software metric manages the estimation of programming item advancement process. It likewise directs and assesses advancement models and apparatuses. Olabiyisi, et. al. [10] likewise expressed that a building discipline based on the experience of programming experts, is a vital piece of program estimation, much the same as for issue definitions.

Hence, a software measure ought to be subjective and quantitative, frequently evaluated with numerical qualities registered from a lot of information. Complex programming is viewed as the purposes behind the nearness of imperfections hence make programming multifaceted nature answerable for poor programming quality. Three significant measurements for estimating the multifaceted nature of code have been accounted for in the writing, which incorporates Lines Of Code (LOC), Halstead's proportion of unpredictability and Cyclomatic Complexity [14].

1.2 Software Complexity

According to Silva *et. al.*, [26], Software complexity, can be characterized as "the product, which is hard to comprehend and change in future". Numerous Programmers characterize complexity as " a task difficult to perform, for example, tasks such as testing, ease of maintenance, debugging as well as future expansion of the product" [21, 22]. The Institute of Electrical and Electronics Engineers (IEEE) characterizes software complexity nature as "how much a framework or segment has a structure or execution that is hard to comprehend and confirm". To quantify the product multifaceted nature, different programming measurements are utilized like LOC, Halstead's Measures of unpredictability, McCabe Cyclomatic Complexity [12]

In 1976, McCabe T. J. presented McCabe's Cyclomatic Complexity to defeat the above issues raised by LOC and Halstead's Measures of complexity [7]. It is processed before the application fruition and can be determined at the beginning times of the software development life cycle when contrasted with Halstead's metric [9, 12, 13, 19]. McCabe's Cyclomatic Complexity is an indication of a program module's control-flow complexity and is a reliable indicator of complexity in large software projects. It is used to measure the complexity of software via the number of independent path or number of independent flows through the graph. It is a direct indicator of software cost and quality which are parameters proved to be directly related to software complexity [15]. Cyclomatic complexity is the most seasoned and the generally acknowledged tools for surveying source program quality [2, 11, 16, 17, 18, 19] and gives a quantitative proportion of the outcomes on the risk evaluations of the product source code [15, 18].

1.2.1 Nature of Software Complexity

Software complexity can be viewed in different ways depending on who is writing. The nature of complexity in this research was viewed in terms of source program organisation. It has to do with the internal structure and arrangement of the source program. Many times in software development, control structures are used to direct the way the program works to achieve the desired goal. Too many unstructured control structures can cause havoc to the program (end product) and in turn, lead to complexity. To produce reliable software, its complexity must be controlled by suitably decomposing the software system into smaller subsystems.

1.2.2 Effects of Complexity on Software

Effects of Complexity can be seen on different aspects of the software, ranging from design, development, testing and maintenance. Unmanaged software complexity can lead to technical debt, high cost and excessed budget, and eventually abandoned solution. In recent years the focus has shifted towards software development approaches that are designed to improve the system's maintainability by introducing automated testing at the earliest stage, writing small modularised units of code and working in short 'sprints' of work, often with a sprint set aside for 'refactoring' or reworking an area of code that may have been considered, or become overly complex.

II. RELATED WORKS

Adewumi, *et. al.* [1] looked into six existing open-source programming quality models and exhibited the trademark highlights, novel strength(s) and confinements of the models. Three chosen criteria which include: regardless of whether any assessment of open source ventures/items utilizing the model has been distributed on the Web, the inception of the model and whether it gives device backing to the assessment procedure, were utilized for a similar report on the quality model. They found that the root of the models can be arranged into four and that every one of the models has aftereffects of their assessment distributed on the Web. Also, it was discovered that four out of the models gave apparatus support. It was inferred that the work will fill in as a guide for open source programming evaluators when they plan to pick a model with which to assess open-source programming choice.

Work done on the correlation of programming complexity nature of linear and binary search algorithms assess, rank aggressive object-oriented applications (Visual Basic, C#, C++ and Java dialects). The algorithms utilized code based complexity measurements such as the line of codes, McCabe cyclomatic unpredictability measurements and Halstead unpredictability measurements. The algorithms estimated programs utilizing length (in lines) of the program, line of code (LOC) without comments, LOC with comments, McCabe strategy, difficulty in the program using Halstead technique. The outcome recommended that McCabe strategy has the least complexity number for Visual Basic, C#, C++ and Java dialects for straight quest and comparative qualities for binary search. The outcome of the research also said any of the four (4) programming dialects can be utilized in actualizing linear and binary search algorithm [10].

Mitre-Hernández, *et al.* [9] talked the consistencies of the complexity of the code produced by students based on the programming exercises found in their programming assessment throughout a semester of studies. The study did comparisons of the level of complexity and the students' grade for various assessments materials and observed a relationship. The conclusion was that the consistencies in producing a certain pattern of expected complexity of a programming code affect students' performance. How the complexity was estimated with the instrument utilized was not captured in the paper.

In a work titled "The Correlation among Software Complexity Metrics with Case Study", Tashtoush, *et al.* [14] said that interest for programming quality is developing progressively, in this manner various scales for the program code are developing quickly to deal with the nature of programming. The product complexity metric is one of the measurements that utilized a portion of the inner traits or attributes of programming to know how it influences the quality of the source program. Tashtoush, *et al.* [14] covers some more efficient software complexity metrics, for example, Cyclomatic complexity, a line of code and Hallstead multifaceted nature metric and additionally examines and breaks down the relationship between them and later uncovers their connection with the number of bugs employing a genuine dataset as a case study yet neglected to develop a device that can be utilized by programming engineers to make a critical decision during implementations.

Malhotra [6] accepted that Complexity in programming is continually considered as an undesirable property in programming since it is a powerfully more regrettable program code quality. A few measures have been made and used by various programming advancement organizations to survey and guarantee esteems in program codes, forms that produce it, and simplicity of the support. The focal point of the study was the improvement of an application utilizing Python programming language dependent on Cyclomatic approach which registers multifaceted nature for program codes written in python. The author failed to test the performance of the tool.

III. RESEARCH METHODOLOGY

In carrying out this research work, Controlled Experiment method was employed. A controlled experiment is a research method that tests hypotheses. There is usually manipulation of one or more *independent variables* and *dependent variables* in an experiment to gauge the influence on one another. The controlled experiment allows researchers to measure precisely in what way these parameters are connected and if there are direct relations between them. Every mix of standards of independent parameters is called *treatment*. The most straightforward experimentations have only two treatments, signifying two degrees of one independent variable (for example utilizing a device versus not utilizing a device) [26]. Increasingly, mind-boggling test plans emerge when there are multiple levels or when more than one independent parameter is utilized. *Subjects* are used to carry out tasks in Software Engineering (SE) experiments and the impacts of the treatments on the subjects are measured.

3.1 Conducting the Experiment

3.1.1 Subjects of the Experiment

The subjects used for this research were volunteered Students (200 level of the Department of Computer Science, University of Ibadan, Nigeria and Fountain University, Osogbo) and software practitioners. The students were chosen as part of the subjects for this research work because a large number of students can be put together to have a controlled experiment. Practitioners working in the software industry were later used to carry out a confirmatory experiment to validate the expected results. These subjects were chosen because:

1. It is appropriate to use the two types of subjects since software users normally get their software developed by either experienced Students due to cost and time frame or contract it to the known software industry.
2. It also helps to compare the quality of software produced by both types of subjects.

3.1.2 Experimental Design

Two separate control experiments were carried out for this study. The first experiment involved 25 Students (in ratio 15 to 10) while the second experiment also involved 25 software practitioners; no attention was paid on gender. The two groups had experimented in a controlled manner. The first experiments were carried out after the students have heard eight (8) weeks of lectures in their Structural Programming course coded as CSC232 and CPS201 courses in their respective institution respectively. The average times spent on the task and some variables were identified and documented by each participant.

The planned instruments for the first set of subjects (students) used for the experiment, real-world programming questions which entailed using of the object-oriented design approach, databases and structures together with control structures were given. Every participant was given a recording sheet to fill in the parameters and observations during the stages of software development. Parameters/variables such as start and end time for the analysis, coding, compilation and time used for fixing errors and the number generated during the experiment were recorded by each participant.

The first experiment with the students was carried out on the 5th of July, 2016 at the Department of Computer Science, University of Ibadan, Nigeria with 15 participants. It was on for about five hours, starting from 11 a.m. This was repeated for the second experiments held on 19th July 2016 at Department of Computer Science, Fountain University, Osogbo with 10 participants. The experiments were overseen and directed by the researcher who also acts as the technologist on duty for the duration of the experiments. Each student's matriculation number was used as the unique number that was filled on the Recording Sheet. The matriculation number served as a means of identification for each subject. To bring out data recorded by each participant, the researchers did thorough analyses of each recording sheet.

The researcher ensured that there were no communications among the participants while the experiments were ongoing to prevent the sharing of ideas among them. Also, all code errors and durations were recorded by each participant on the reporting forms and their outputs were saved on the external device.

The procedure above was repeated for software practitioners in the industry in February 2017 in one of the popular software houses in Nigeria. The experiment also involved 25 practitioners, without paying attention to their gender compositions. The same problem solved by the student-subjects was given after which record sheets were served to them to record their observations.

3.1.3 Nature of the Experiment
The experiment was done in two (2) phases. Phase 1 was carried out using the student as subjects while phase 2 was done as a confirmatory experiment using practitioners working in the industry. The same task was given to the participants in each phase to solve, and all necessary figures were recorded on the record sheet given.

Aim:
The experiment aimed at determining the performance of the developed Complexity Measurement tool by accessing the quality of the code produced by the two subjects used during the experiment.

Language:
The languages used for the experiments were C/C++ and / or C#.

Task Given:
Over the years, the security of data transported over the network has always been susceptible to different types of threats. Similarly, different algorithms have been designed for encrypting the data to strengthen its security which also has associated problems. Develop a data encryption algorithm tagged PQR employing the following characteristics:

1. A piece of information to be sent along over a system is taken as a solitary string. Be that as it may, the base number of characters in the string must be three (3)
2. The string is then partitioned into two equivalent substrings, A and B.
3. Each of the substrings A and B are then encoded in a turn-around request.
4. To further scramble the string information, vowels in the substrings are encoded as pursues:

Vowel	Encoding Number
a	1
e	5
i	3
o	2
u	4

5. To send the first string information from its source, the scrambled substring An and B are connected with A coming after B.

Implement a program in C or C++ or C# to solve the above-stated problem.

3.2 Variables / Metric Used In the Experiment
Dependent and independent variables were manipulated in this study. The independent variables manipulated in this work are mainly two- the means of getting software developed (student, and professionals) and the complexity of the code produced by each group.

The dependent variables measured as adapted from [25] were:

1. Effort/time for analysis: this is the period used by each subject in understanding the problem. These centred on their:
   • Interpretation of the programming problem;
   • Analyses plan and figuring out the inputs cum outputs parameters, corresponding data types and their modes
   • Selection of suitable control structures for the problem
   • Presentation of result
   • Usage of selected object-oriented programming concepts such as classes and objects.
2. Effort/time range for Coding (EC): this is the time taken by each participant to implement their plans, using the selected programming language.
3. Effort/time range for Compilation: this is the time used by the compiler in interpreting the program code into the target object code.
4. Effort/time range for Debugging: this is the time used for correcting or fixing the error(s) by the participant.
5. Number of Bugs (NB): this is the number of errors incurred during different stages of compilation.

The other metrics used in the research work as attached to the efficiency of the tool include:

6) NOM (Number of Method) **-** this lists the methods/classes present in each system.
7) NSLOC (Source Line of Code) –counts the code length/line per each method/class.
8) CCV (Cyclomatic Complexity Value) – measures the complexity of each method by presenting the associated value.
9) CC (Cyclomatic Complexity Category) – measures the level of cyclomatic complexity value.

(NOTE: Nos 1-5 above were measured and recorded by the participants during experiments while nos 6-9 were measured by the tool.)

3.3 Data Analysis strategy

This research employed qualitative data analysis techniques in its data analysis. **R** statistical tool was used in the analysis. R was chosen because of its robustness and openness. It has the capacity to handle a variety of specific statistical analyses as well as an interactive programming language. Correlations between the variables under study were done to know their dependencies; Analysis of Variance (ANOVA) which is a rigorous statistical tool used in making inferential decisions in experimental design studies to ensure the equivalence of comparative groups even when number per group differed across the group was employed. Some of the data gotten from this study were also subjected to independent test at $\alpha_{0.05}$. Descriptive statistics such as mean, median standard deviation, etc. were also methods of analysis used by the researcher.

.

3.4 Code Complexity Measure

This research employed Complexity Measurement Tool (CMT) for assessing code quality and also compared with Code Metrics (CM), which is embedded in Visual Studio (VS) environment. The two tools were developed based on McCabe Cyclomatic Complexity measure approach in measuring the code complexity.

3.4.1 *Language Scanner*

The language scanner is one of the major functional units of the quantity assessment process. It reads characters from the source file, group input characters into meaningful units, called tokens. A token is a logical entity described as part of the syntax of a language. The scanner does the removal of comments and white spaces within the source program, keeps track of current line number which is required for reporting error messages, interpretation of directives, communication with the symbol or literal table among others. The language scanner designed in this study uses a regular expression to generate a token from the inputted source code and put them in their respective groups such as identifiers, operators, keywords, etc. Finite Automata are used to recognise the tokens specified by a regular expression and converted to an algorithm for matching input strings. This process is shown in Figure 1.

3.4.2 *Code quality Assessment*

The process of code quality assessment takes source code artifact from the experiment as an input; pass it to language scanner which scans and test if the artifact syntax belongs to C/C++ (.c, .cpp, .h, .hpp), C# (.cs) or javascript. If yes, it identified the classes or methods that makeup of the whole source code and then determines the complexity values of each of the methods with the appropriate categories it belongs to by displaying the results on the output panel. The program component that has its Cyclomatic Complexity measured between 1 and 10 is said to be free of risk, Software and/or component whose Cyclomatic Complexity measured between 11 and 20 is said to be of moderate risk, Software and/or component with Cyclomatic Complexity measured between 21 and 50 is said to be of high risk while those of 50 and above are regarded as very high risk and untestable software. The methods with the complexity less or equals to ten (10) are considered as having low complexity while those methods with the complexity of more than ten (10) are written and suggested to the complexity tool till they have low complexity that is deployable and testable. The framework for code quality assessment is shown in figure 2.

IV. RESULTS AND DISCUSSION

The results of the analysis of the dependent variables recorded by the participants (SP and PP) during the experiments were summarized in Figure 3.

DISCUSSION

Analysis Efforts: The leading stage in writing computer programs is the analysis of the current issue. The problem must be well understood and carefully analyzed for the input and output parameters expected as well as the processing logic to follow in solving the problem at hand. At the analysis phase, program design also comes into play. The design of the inputs and outputs, user interface, data structures, file/database design as well as test cases writing are carried out. The effort exhausted by the programmers in the analysis and design is compared to the total time spent doing the analysis. As shown in Figure 3, it is shown that time taken by the student programmers to comprehend the problem ranged from 4.2 (minimum) to 132(maximum) minutes while that of the professional programmer ranges from 6 (minimum) to 69(maximum) minutes which is almost half of the time take by student programmers.

Coding and Debugging (Programming) Effort: This is the stage where the real work (coding) is done. Many programmers tend to go straight to coding without a well understanding of the problem which makes them spend more time and effort trying to figure out how the logic and the flow of the program should be. Figure 3 shows that the time taken by the student programmers to do the coding and debugging ranged from 76.8 (minimum) to 126 (maximum) minutes while that of the Professional programmer ranges from 48 (minimum) minutes to 148.2 (maximum) minutes.

Programming Efforts: This is the overall time spent from when the analysis started until when the desired output was gotten, which invariably means the time spent on the experiment. From Figure 3, it can be inferred that the time spent by the Professional programmer ranges from 66 (minimum) minutes to 129 (maximum) minutes while that of the Student programmers ranges from 91.2 (minimum) minutes to 131.6 (maximum) minutes.

Number of bugs: This is the number of errors incurred by each participant as recorded during the practical sections. As shown on Figure 3, the bugs incurred by Professional programmers' ranges from 0 (minimum) to 10 (maximum) while that of the Student programmers' ranges 2 (minimum) to 7 (maximum).

Efficiency: The performance of the whole program (output of the experiment) was measured in terms of the complexity of the code produced and the number of bugs incurred by each of the groups. The developed Complexity Measurement Tool (CMT) was adopted for the research. The CMT reported some methods which include Number of Source Lines of Code (NSLOC), CCV and CCC as shown in figure 3 while Code Metrics (CM) in Visual Studio only reported NSLOC and CCV for all the codes. Two metrics CCV and NSLOC were assumed to have a great effect on program code complexity and so, the correlation between NSLOC and CCV were done in the R environment. Section 4.1 shows the details of the correlation results by each group of subjects.

Figure 3 shows the part of the tool where the input is accepted and at the same time displays the result of the metrics after the processing. The interface is partitioned into columns, each of which holds the results based on the following attributes:

Category: The category column specified the level of complexity of the methods in the analysed code. It also gives a more detailed description of the complexity value.
Unit: The Unit specifies and lists the name of methods as identified in the source code program.
Complexity: This column classifies the complexity value of each of the method as identified in the source using the McCabe‘s Cyclomatic complexity approach.

SLOC: SLOC (Source Line of Codes) indicates the total number of code lines for each of the methods in the inputted source program.
File: The file column indicates the paths to which the file containing the source program is located

4.1 The Correlation between CCV AND SLOC among Students Programmer (SP) Professionals Programmer (PP)

As said earlier, the performance of the tool was measured by finding the relationship between variables (metrics) as measured by the **CMT**, which are **SLOC** and **CCV**. The results for both categories of subjects (SP and **PP)** as analysed in R environment were summarized in Table 1.

From Table 1, the Correlations between the SOC and CCV for both SP and PP are close to 1 (i.e. 0.72 and 0.83) with the P-values of ($6.776e^{-06}$ and $2.529e^{-13}$). This is interpreted as a strong positive relationship between the Source Length of Code (SLOC) and the Complexity of the Code (CCV). Statistically speaking, the smaller the P-value, the greater the evidence against the null hypothesis. Consequently, the P-values of ($6.776e^{-06}$ and $2.529e^{-13}$) leads to the rejection of the null hypothesis $H_{02.}$ This implies that the two parameters (SLOC and CCV) are key factors in predicting the quality of any software product and so will have a strong effect on the Software development environment.

4.2 Two-Sample T-Test for SP and PP using complexity Value
T-test Result for SP and PP **using their Complexity Value:**

```
Welch Two Sample t-test
t = 1.7036, df = 54.311, p-value = 0.09418
alternative hypothesis: true difference in mea
95 percent confidence interval:
          -0.6439187  7.9316080
sample estimates:
          mean of x mean of y
          11.41935   7.77551
```

The results above show the student's t-test results for two groups of the subject used; the t-test statistics value (t=1.70), df is the degrees of freedom df=54.311), the p-value is the significant level of the t-test (p-value=0.094). The confidence interval (conf.int) of the mean differences at 95% is also shown (conf.int=[-0.64, 7.93]); and finally, we have the sum of the two groups of samples (average CCV_SP = 11.42, average CCV_PP =7.78). This is interpreted as strong significant differences between the two subjects used.

This implies that there are differences in the performance of the code produced by both SP and PP which means that software product developed through any of these subjects will have different complexity capacities, and the rate at which they are maintainable, user-friendly and reliable also differs.

4.3 Comparison between CMT and CM using descriptive statistics
Some of the descriptive statistics of all the dependent variables and metrics used in the study for in comparing the CMT and CM were shown at a glance in Table 2.

V. CONCLUSIONS

The efficiency of the CMT was subjected to evaluation by conducting two separate experiments using two subjects- Student Programmers (SP) and Professional Programmers (PP). The experiments were conducted under a controlled experiment for both SP and PP at different time and location while metrics such as Analysis Effort (AE), Coding Effort (EC), Compilation Effort (CE), Debugging Effort (DE), and Number of Bugs (NB) were recorded in the record sheet by each subject.

Program code artifacts were gotten after experimenting in two categories for SP and PP. The quality of the program code was examined using the developed CMT, which calculates the complexity of the software product on modular bases and reported the Number of Class/ Method (NOM) contained, the Source Line of Code (SLOC) and the acquired file path. The results from the tool were analyzed using the R language and were later interpreted.

The result showed that both categories have tendencies of producing software whose complexity varies across the four levels of complexity. This implies that software products produced by either student programmer (SP) or professional programmer (PP) can either be easy to maintain, reusable and easily customize to specific users' desire or otherwise are difficult to maintain, and customize to specific users' desire and may eventually fail if not given special attention. Testing this software also may be a lot easier and at the same time difficult, because the complexity values indicate the number of test cases that should be employed during testing.

Similarly, the result also shows that there are strong relationships between the Source Length of Code (SLOC) and the Cyclomatic Complexity Value of the Code (CCV), implying that the two parameters are key factors in predicting the quality of any software product. So, this will have a strong effect on the Software development environment; hence, software developers will benefit immensely by knowing the complexity of the code being produced. Equivalent complexity tool for web programming languages should be of future interest. This tool should be able to adjust to dynamically changing team configurations at different phases in the software development lifecycle. Future work can focus on the development of the tool that can bridge the geographical locations among the developers to increase productivity and facilitate location coordination.

REFERENCES

[1] A. Adewumi, S. Misra, and N. Omoregbe, "A Review of Models for Evaluating Quality in Open Source Software," *2013 Int. Conf. Electron. Eng. Comput. Sci.*, vol. 00, no. 2012, pp. 1–5, 2013.

[2] V. Antinyan, M. Staron, J. Hansson, W. Meding, P. Österström, and A. Henriksson, "Monitoring evolution of code complexity and magnitude of changes," *Acta Cybern.*, vol. 21, no. 3, pp. 367–382, 2014.

[3] IEEE Computer Society, "IEEE Standard for a Software Quality Metrics Methodology - IEEE Std 1061^{TM}-1998 (R2009)," *IEEE*, vol. 1998, pp. 1–26, 2009.

[4] C. Ikerionwu, "Cyclomatic Complexity as a Software Metric," *Int. J. Acad. Res.*, vol. 3, no. 2, pp. 117–121, 2010.

[5] M. C. Lee and T. Chang, "Software measurement and software metrics in software quality," *Int. J. Softw. Eng. its Appl.*, vol. 7, no. 4, pp. 15–34, 2013.

[6] M. P. Malhotra, "Python Based Software for Calculating Cyclomatic Complexity," *Int. J. Innov. Sci. Eng. Technol.*, vol. 2, no. 3, pp. 546–549, 2015.

[7] P. Meirelles, C. Santos, J. Miranda, F. Kon, A. Terceiro, and C. Chavez, "A study of the relationships between source code metrics and attractiveness in free software projects," in *Proceedings - 24th Brazilian Symposium on Software Engineering, SBES 2010*, 2010, pp. 11–20.

[8] H. A. Mitre-Hernández, G. Javier, D. A. Antonio, and V. Perla, "Designing a Strategic Measurement Program for Software Engineering Organizations: Discovering Difficulties and Problems," *Ing. Investig. y Tecnol.*, vol. 15, no. 2, pp. 253–269, 2014.

[9] N. Mohamed, R. Fitriyah, R. Sulaiman, W. Rohana, and W. Endut, "The Use of Cyclomatic Complexity Metrics in Programming Performance's Assessment," *Procedia - Soc. Behav. Sci.*, vol. 90, no. InCULT 2012, pp. 497–503, 2013.

[10] S. O. Olabiyisi, E. O. Omidiora, and K. A. Sotonwa, "Comparative Analysis of Software Complexity of Searching Algorithms Using Code Based Metrics," *Int. J. Sci. Eng. Res.*, vol. 4, no. 6, pp. 2983–2993, 2013.

[11] C. D. De Oliveira, P. E. Black, and E. Fong, "Impact of Code Complexity on Software Analysis," *Nistir 8165*, 2017.

[12] S. Omri, P. Montag, and C. Sinz, "Static Analysis and Code Complexity Metrics as Early Indicators of Software Defects," *J. Softw. Eng. Appl.*, pp. 153–166, 2018.

[13] K. P. Srinivasan, "Unique Fundamentals of Software Measurement and Software Metrics in Software Engineering," *Int. J. Comput. Sci. Inf. Technol.*, vol. 7, no. 4, pp. 29–43, 2015.

[14] Y. Tashtoush, M. Al-maolegi, and B. Arkok, "The Correlation among Software Complexity Metrics with Case Study," *Int. J. Adv. Comput. Res.*, vol. 4, no. 2, pp. 414–

419, 2014.

[15] R. Tombe and G. Okeyo, "Cyclomatic Complexity Metrics for Software Architecture Maintenance Risk Assessment," *Int. J. Comput. Sci. Mob. Comput.*, vol. 3, no. 11, pp. 89–101, 2014.

[16] N. Ukić, J. Maras, and L. Šerić, "The influence of cyclomatic complexity distribution on the understandability of xtUML models," *Softw. Qual. J.*, vol. 26, no. 2, pp. 273–319, 2018.

[17] V. Shanthi, P. Jeevana, G. K. Chaithanya, and C. Thirumalai, "Evaluation of McCabe's cyclomatic complexity metrics for secured medical image," *Proc. - Int. Conf. Trends Electron. Informatics, ICEI 2017*, vol. 2018–Janua, no. June, 2018.

[18] A. Garg, "An approach for improving the concept of Cyclomatic Complexity for Object-Oriented Programming," 2014.

[19] A. Madi, O. K. Zein, and S. Kadry, "On the improvement of cyclomatic complexity metric," *Int. J. Softw. Eng. its Appl.*, vol. 7, no. 2, pp. 67–82, 2013.

[20] J. Y. Gil and G. Lalouche, "When do software complexity metrics mean nothing?- When examined out of context," *J. Object Technol.*, vol. 15, no. 1, pp. 1–25, 2016.

[21] H. De Silva, D. I. , Kodagoda, N. and Perera, "Applicability of three complexity metrics," in *" International Conference on Advances in ICT for Emerging Regions (ICTer2012), Colombo*, pp. 82–88 , 2012.

[22] U. Chhillar, and S. Bhasin, " A new weighted composite complexity measure for object-oriented systems", *International journal of information and communication technology*, 2011, pp 101-108.

[23] K.P. Srinivasan, and T. Devi, "Software Metrics Validations Methodologies in Software Engineering," *International Journal of Software Engineering and Applications*, Vol. 5, No. 6, November, 2014, pp.87-102.

[24] S. Easterbrook, J. Singer, M. A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*, Springer, London, 2008, pp. 285–311.

[25] S. O. Akinola, and B. I. Ayinla, An Empirical Study of the Optimum Team Size Requirement in a Collaborative Computer Programming/Learning Environment. *Journal of Software Engineering and Applications*, Vol. 7:1008-1018, 2014.

[26] H. De Silva, D. I. Kodagoda, and N. Perera, "Applicability of three complexity metrics," in International Conference on Advances in ICT for Emerging Regions (ICTer2012), Colombo, 2012, pp. 82–88.

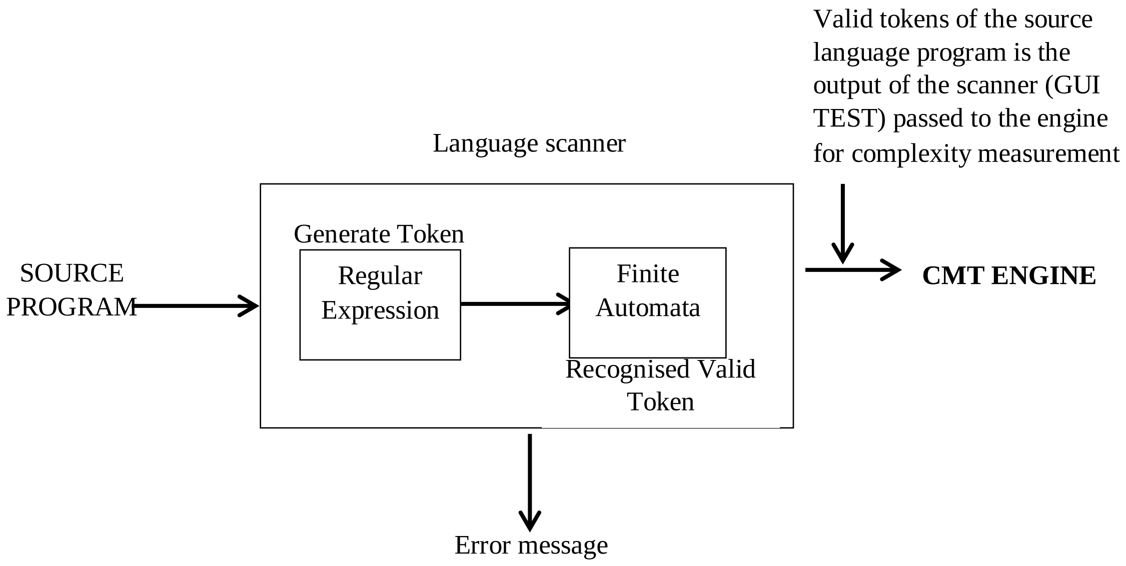

Valid tokens of the source language program is the output of the scanner (GUI TEST) passed to the engine for complexity measurement
Language scanner
Generate Token
SOURCE PROGRAM
Regular Expression
Finite Automata
CMT ENGINE
Recognised Valid Token
Error message

Figure 1: Language scanning process

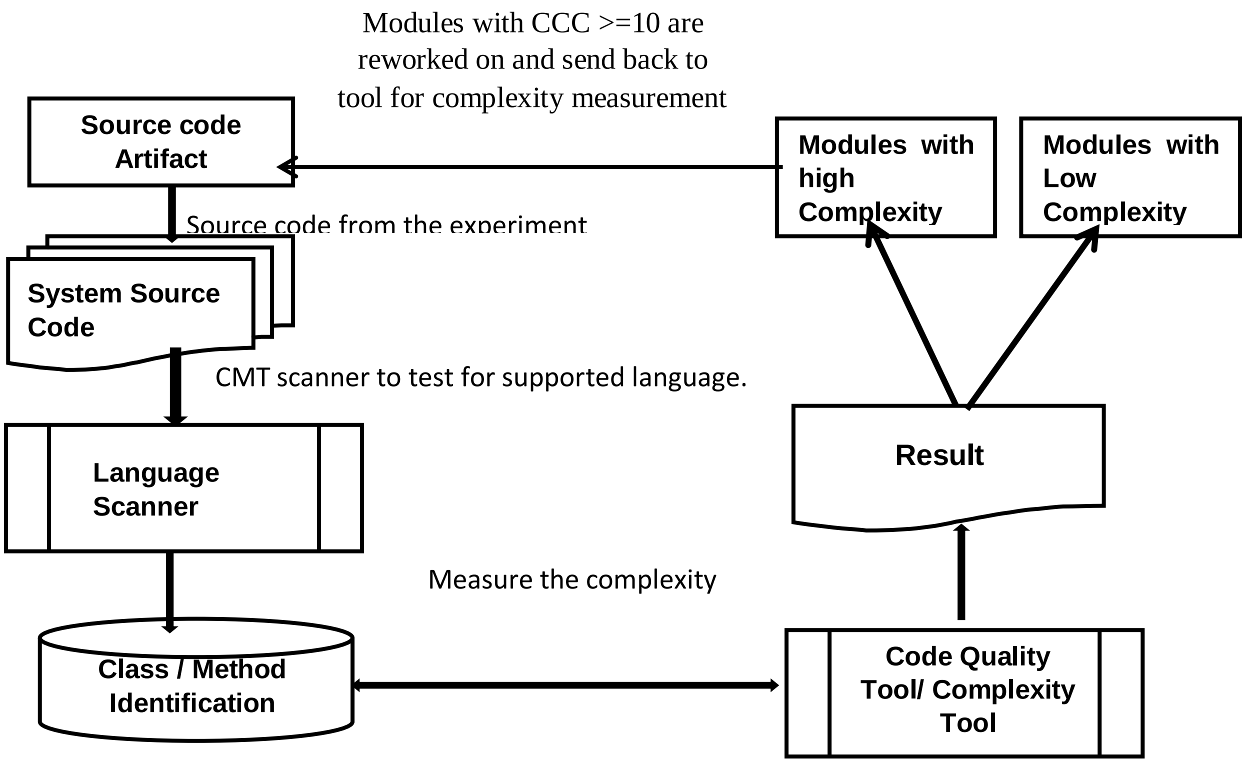

Modules with CCC >=10 are reworked on and send back to tool for complexity measurement
Source code Artifact
Modules with high Complexity
Modules with Low Complexity
Source code from the experiment
System Source Code
CMT scanner to test for supported language.
Language Scanner
Result
Measure the complexity
Class / Method Identification
Code Quality Tool/ Complexity Tool

Figure 2: Framework for Code Quality Assessment Process

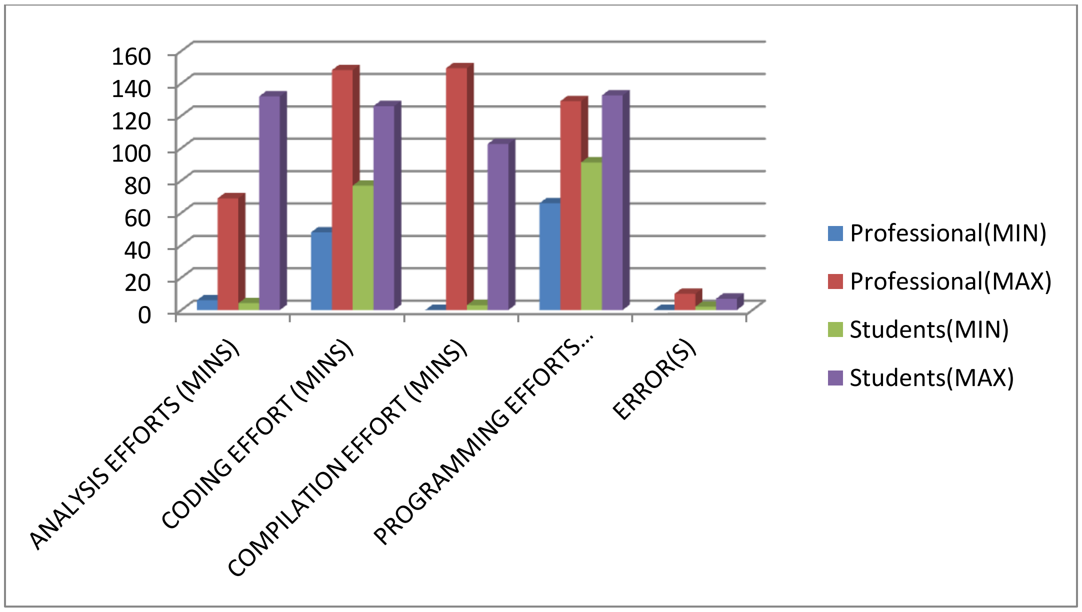
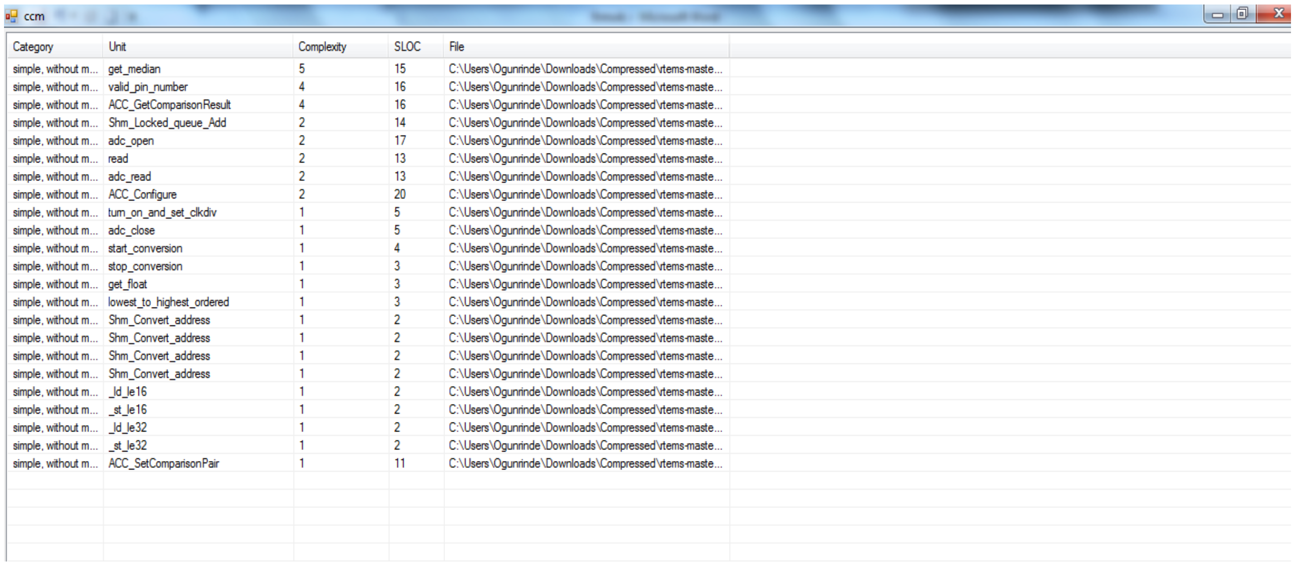


Figure 3: Summary of the Experimental Results for both Professionals and Students programming



Category	Unit	Complexity	SLOC	File
simple, without m...	get_median	5	15	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	valid_pin_number	4	16	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	ACC_GetComparisonResult	4	16	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	Shm_Locked_queue_Add	2	14	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	adc_open	2	17	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	read	2	13	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	adc_read	2	13	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	ACC_Configure	2	20	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	turn_on_and_set_clkdiv	1	5	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	adc_close	1	5	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	start_conversion	1	4	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	stop_conversion	1	3	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	get_float	1	3	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	lowest_to_highest_ordered	1	3	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	Shm_Convert_address	1	2	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	Shm_Convert_address	1	2	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	Shm_Convert_address	1	2	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	Shm_Convert_address	1	2	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	_ld_le16	1	2	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	_st_le16	1	2	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	_ld_le32	1	2	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	_st_le32	1	2	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...
simple, without m...	ACC_SetComparisonPair	1	11	C:\Users\Ogunrinde\Downloads\Compressed\items-maste...

Figure 3: Interface showing Complexity result of the inputted Source Code

Table 1: Summarized Correlation results between CCV and SLOC between SP and PP

Methods	**t-value**	**Cor2cor**	**Level of sign**	**P-value**
Students	5.5164	0.7217	95%	= 6.776e^{-06}
Professionals	10.073	0.8267	95%	= 2.529e^{-13}

Table 2: Descriptive statistics for SP, PP and Open Source System using CMT and CM

Variables **Metrics**	**CMT**	**CM**
Mean NSLOC (SP)	25.5±0.2	23.3±0.2
Mean NSLOC (PP)	26.2±0.2	26.4±0.2
Mean CCV (SP)	7.5±0.1	8.5±0.1
Mean CCV(PP)	9.2±0.1	9.0±0.1
Mean CCC (SP)	Risk-Free	Not Available
Mean CCC (PP)	Moderate Risk	Not Available
Standard Deviation CCV (SP)	12.3±0.1	11.0±0.1
Standard Deviation CCV (PP)	9.87±0.1	11.8±0.1
Standard Deviation NSLOC (SP)	32.4±0.2	30.57±0.2
Standard Deviation NSLOC (PP)	40.7±0.2	31.5±0.2
Average System's Computational Time (SCT)	1.0 (Seconds)	3.0 (Seconds)

APPENDIX I

RECORDING SHEET:

The recording sheet was given to each participant to record the following variables:

Participant No: ______________ **Name**: ____

Experiment start Time: _______________

Analysis Effort i.e. record the time analysis of the problem start and time finished

Analysis Start Time: __________________

Analysis End Time: __________________

Coding Effort i.e. record the time coding start and time finished

Coding Start Time: __________________

Coding End Time: __________________

Compilation Effort i.e. records the time compilation start and time finished (record the first, second and the subsequent ones)

Compilation Start Time: __________________

Compilation End Time: __________________

Debugging Effort i.e. record the time debugging start and time finished (record the first, second and the subsequent ones)

Debugging Start Time: __________________

Debugging End Time: __________________

Number of Bugs. i. e. record the number of bugs at each compilation

Number of Bugs: ________________

Record the time the experiment finished

Experiment End Time: _______________