ISSN:
Print - 2277 - 0593
Online - 2315 - 7461
© FUNAAB 2021

Journal of Natural Science, Engineering and Technology

A MULTILANGUAGE COMPLEXITY MEASUREMENT TOOL FOR CODE QUALITY ASSESSMENT OF SOFTWARE USING CYCLOMATIC COMPLEXITY APPROACH

***[1]M. A. OGUNRINDE AND [2]O. S. AKINOLA**

[1]Department of Mathematical and Computer Sciences, Fountain University, Osogbo, Nigeria.
[2]Department of Computer Science, University of Ibadan, Ibadan, Nigeria.
***Corresponding Author:** bogunrinde@gmail.com **Tel:** +2348054280030

ABSTRACT

Code complexity or quality has been a focus point by software stakeholders, and on several occasions, has led to the abandonment of codes that has consumed time and money to develop. However, tools that measure code complexity and predict future maintenance across some development platforms before deployment are inadequate. This study was designed to develop a Complexity Measurement Tool (CMT) for assessing code quality in different platforms and compare its performance with that of an existing complexity tool. McCabe cyclomatic complexity approach was adopted and the CMT was developed using C# language to support four programming languages: C, C++, C# and JavaScript. The tool adopted source codes written in any of the above-mentioned programming languages as input, scanned through and reported names of each method contained in the source program, their code lines, the complexity of each of the method and also specified the equivalent category of the complexity value. The performance of CMT was compared with Code Metrics (CM), an existing complexity equivalent tool embedded in Visual Studio (VS) environment using System's Computational Time (SCT) and result representations. The average SCT obtained from CMT and CM for all the codes were 1.0±0.01 and 3.0±0.01 minutes. The complexity measurement tool with cyclomatic complexity category had better speed and result interpretation. This will assist software developers in building quality into their products. The result from the tool can also be used in making critical decisions by software stakeholders.

Keywords: Code Quality, Software Maintenance, Multi-language, Software Stakeholders, Complexity Tool, Software Maintainability, Quality Assessment

INTRODUCTION

One of the major objectives of software development is the system quality. Since this is the most desired of by stakeholders in the domain, it is therefore essential to identify the quality factors and improve them. Programming code quality can be termed as a code characterized with a wide range of attributes which includes ease of maintenance, testability, reusability, difficulty, reliability, interoperability, etc. (IEEE Computer Society, 2014). The significant method to look at the quality is by ceaseless and early assessment of the program codes as it progresses. The program codes must have high quality to address the business issues in today's market and therefore becomes a focal point for developers of software. The presence of certain factors that influence the quality of program code, software engineers has come to conclusions that there is a link between unseen attributes of program code such as cost, strength, Lines of code (LOC), speed etc. and those attributes of program code which has direct impact on the stakeholder such as usefulness, quality, complexity, effectiveness, reliability or viability. For instance, a greater number of code lines may prompt more prominent software complexity, etc (Tashtoush, Almaolegi, & Arkok, 2014).

Similarly, the complexity of software may have effects on maintenance activities like software testability, reusability, understandability, and modifiability (Tashtoush et al., 2014). According to IEEE (2014) Software complexity is defined as the degree to which a system or component has a design or implementation that is difficult to understand, verify and change in future. Many Programmers define software complexity as "Difficult to perform a task such as testing, maintenance, debugging and to change the software in future" (Ardito et al., 2020; De Silva, Kodagoda, and Perera, 2012). Variables that make the program codes hard to comprehend are accountable for its complexity. As a result, it is important to find way of reducing the effects of the complexity on program codes and guarantee the quality simultaneously. Hence, the important task is keeping quality of the program codes in light of the requisite functionalities. The focus of this work was to develop a Complexity Measurement Tool (CMT) based on McCabe approach for measuring the complexity of Software before deployment. The tool has the capacity to be incorporated into Integrated Development Environments (IDEs) and also presented the decision-makers with some information about the target solution. The remaining part of the paper is arranged as follows; section 2 discussed the McCabe Cyclomatic Complexity approach followed by related works in section 3, section 4 talked about the materials and methods used while section 5 presented the results and discussion.

MCCABE CYCLOMATIC COMPLEXITY APPROACH

Three major metrics for measuring complexity of program codes have been reported in the literature which includes Lines Of Code (LOC), Halstead's measure of complexity and Cyclomatic Complexity metric (Gujar, 2019 ; Tashtoush et al., 2014). LOC metric only counts the number of lines in the program's source code. It is calculated at the end of the application completion, ignores the complexity of decision statements and considers the complexity of each code line (Gujar, 2019;Ostberg and Wagner,2014; Ogheneovo, 2013). In solving the difficulty of LOC metric, a new metric called Halstead's complexity metric was introduced. This metric is used to measure the complexity of the program or modules directly from the program source code but there was also some problem with this metric. Halstead classifies a program as a collection of tokens, either operands or operators but only includes the complexity from data flow and is also calculated at the end of the whole program's implementation (Delange, 2015; Tashtoush et al., 2014; Madi, Zein, & Kadry, 2013; Serebrenik, 2011).

Cyclomatic complexity has been the oldest and the widely acceptable tool for assessing software quality out of the three code complexity tools in the literatures, (Alenezi &

Zarour, 2020; Gujar, 2019;Ukić, Maras, & Šerić, 2018; Shanthi, et al., 2018; Vard Antinyan et al., 2014; Garg, 2014; Madi et al., 2013;Mohamed, et al., 2013). The reason as to why the McCabe's metrics is used is because it gives a quantitative measure of the results on the risk assessments of the software component (Alenezi & Zarour, 2020; Ardito et al., 2020; Gujar, 2019; Tombe and Oliveira Okeyo, 2014).

McCabe's Cyclomatic Complexity was introduced by T. J. McCabe in 1976 to overcome the problems associated with LOC, Halstead's Measures of complexity. It is computed before the application completion and can be calculated at the early stage of software development life cycle as compared to Halstead's metric (Omri, Montag, & Sinz, 2018; Oliveira, Black, & Fong, 2017; Gil & Lalouche, 2016; Meirelles et al., 2010) McCabe's Cyclomatic Complexity is an indication of a program module's control-flow complexity and has been found to be 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 because these two parameters are directly related to software complexity (Tombe and Okeyo, 2014).

The Significance of the McCabe Number

Research has shown that program with McCabe Cyclomatic complexity number above 10 has a higher complexity which implies that the program likely to contain bugs and flaws then it will be extremely hard to comprehend such program. Likewise, the number of experiments required to test such program code will be on the high side too. The higher the Cyclomatic number, the higher will be the bugs. Similarly, the higher will be the maintenance time required for extending the code to accommodate more features as well as the number of test cases required during testing. Software Engineering Institute, categorized risk associated with the software or its component in term of the measured Cyclomatic Complexity (Table 1). It shows that module and/or program component that has its cyclomatic complexity measured between 1 and 10 is said to be free of risk; which means the code is well structured and require less maintenance cost and effort, Software and/or component whose Cyclomatic Complexity measured between 11 and 20 is said to be of moderate risk, less structured, require more maintenance effort and cost. Software components whose Cyclomatic Complexity measured fall between 21 and 50 are said to be of high risk, less structured and require more effort and cost while those of 50 and above are regarded as very high risk, unstructured and untestable software and require grate maintenance effort and cost (Surendra, 2020).

Table 1: Cyclomatic Complexity Categories with equivalent Number (Surendra, 2020)

Cyclomatic Complexity	Risk Evaluation (Complexity Category).
1-10	Software considered as risk-free software
11-20	Software considered having a moderate-risk
21-40	Software considered as of high-risk
41 and above	Considered as very high risk and untestable software,

RELATED WORKS

According to Gujar (2019), the use of Cyclomatic Complexity as a software metric in software developments an not be over emphasised. The author referred to the metris as a White box and structural testing required in the life of an upcoming software. The purpose of the paper is to describe the Use and Analysis on Cyclomatic complexity in Software development with an example. The result of the research shows that the high the decision points within the program, the higher the complexity.

Another expert, Liu et al. (2018), said programming development influence the complexity of a software and its component in an object-oriented systems emphasised. They built a supporting tool based on cyclomatic complexity which was used to assess some programming codes. Their conclusion was that though cyclomatic approach to measuring complexity in program codes has been projected over forty years ago, it remains a significant guide to measuring quality in program code and determining ease of future maintenance for all types of programming dialects. At the same time pointed out that there are lots of advantages for developers and other stakeholders involved in making critical decisions using the tool.

Malhotra (2015) is of the opinion that Complexity in software is constantly considered as an undesired property since it is a dynamically worsen program code quality. Some measures have been created and utilized by different software development companies for assessing and ensuring values in program codes; processes that produce it are kept at a minimal level, and guarantee ease of maintenance. The focus of the study was the development of an application using Python programming language based on Cyclomatic approach and computes complexity for program codes written in python.

Miguel et al., (2014) conducted a survey of Software Quality Models for the evaluation of Software Products. They are of the opinion that since software product is used in an aspect of our life, measuring and evaluating its quality should be given utmost importance. A few models have been proposed to help different kinds of clients with quality issues. They classified software developed as at 2000 has depended on produced or factory-made components and offered to meet people's high expectations for evaluating quality such as low configurability, unreusability, un-maintainability, under quality and lower-cost products. Subsequently, grouped into fundamental models are models which were created until 2000, and segments or custom-made quality models. Their article also described the strengths of each model together with their weaknesses. They concluded that in the present age, interaction between software stakeholders neither play a significant role in producing software that is of high standard program code or product.

In a work titled "The Correlation among Software Complexity Metrics with Case Study", Tashtoush et al. (2014) observed that interest of researchers for programming quality is developing progressively, hence various scales for the product are developing quickly to deal with the nature of programming. The product complexity metric is one of the estimations that utilization a portion of the inner traits or attributes of programming to know how they impacted on the product quality. In this paper, the Author discussed how increasingly productive programming complexity measurements, for example, Cyclomatic multifaceted nature, a line of code and Hallstead complexity metric and their effects on the product quality. It equally examined and breaks down the relationship between them. The authors finally concluded that all are connected with the quantity of error incurred by utilizing a genuine dataset as a contextual analysis (Tashtoush et al., 2014).

According to Antinyan et al. (2014), managing Complexity has turned into an essential measure in constant programming improvement. Their research was aimed at developing a measurement system that keeps tracks of features that can cause difficulty in software development through being part of a research conducted in two big programming houses. They evaluated three complexity methods and two change properties of code for two big industrial products. They also monitored complexity progress for five consecutive releases of the products. The outcomes demonstrated that observing cyclomatic complexity values, capacities development and number of revisions of function concentrated the attention of the designers to possibly dangerous functions and capacities for evaluation and improvement.

Antinyan et al. (2013) observed in a work titled "Monitoring Evolution of Code Complexity in Agile/Lean Software Development". The authors said that one of the unique features of Agile and Lean programming process is the "grow" of program code which is relative to new functionality with moderately little augmentations as the client keep on requesting for an extension of the initial features. The capacity of the organizations to convey on those requests has two main thrusts behind it which have serious effect on software evolution. The objective of this study was to introduce an estimation framework for observing these driving forces, during their contextual analysis. Two major domains were investigated using development of size, complexity, corrections, and number of creators of two enormous programming items. Their outcome revealed that McCabe's complexity vale and version corrections are two key features that should be properly observed to have the complexity of the program code under threshold.

Debbarma, Debbarma, Debbarma, Chakma, & Jamatia, (2013) reviewed and analysed Software Complexity Metrics focusing on Structural Testing. They said one of the fundamental goals of programming measurements is that applying it to both product and processes that produced it. Software components with high complexity are generally considered worse than those with low complexity. In their work, they reported that complexities in program code are factors that should be determined, pursued, and controlled frequently at any stage of development which includes testing. Various parts of measurements in structural testing which helps in calculating efforts needed for testing program code were evaluated and carefully studied. They concluded that measuring the complexity of the software could prompt reduction in development estimated cost and improve testing adequacy.

Mohamed et al. (2013) worked on using Cyclomatic Complexity Metrics in assessing student programming performances as a student always believe that Programming courses are usually difficult and complex. The complexity of a source program may be controlled by the number of the linearly autonomous track within the code. Their study focused on deliberating the consistencies of the complexity of the code produced by students based on the programming questions in the practical assessment throughout a semester of studies and then compared the level of complexity and the students' grade for various assessments materials done. Based on this, they observed a relationship and concluded that a certain pattern of the expected complexity of a programming code has an effect on students performance.

In a work titled "Assessment of Software Quality: Choquet Integral Approach", Pasrija, Kumar, & Srivastava (2012) applied another approach to software quality assessment based on the fact that various initially proposed software quality model could not qualify the software parameter efficiently. Another strategy was proposed for contrasting distinctive programming arrangements dependent on the SRS to a typical issue. The study concluded that the introduction of Choquet Integral with fuzzy measures should be taken into cognizant, provided significant concepts of cooperation among software criteria exist.

Based on the results from the review of related works, it can be concluded that despite the popularity of McCabe Cyclomatic complexity, its automation is scarce in the literature, and available complexity tools are inadequate.

METHODOLOGY

The focus of this work is to develop a software quality metric that can be useful for both academic researchers and industrial practitioners. The tool was developed using c# technology and the Research methodology (Figure 1).

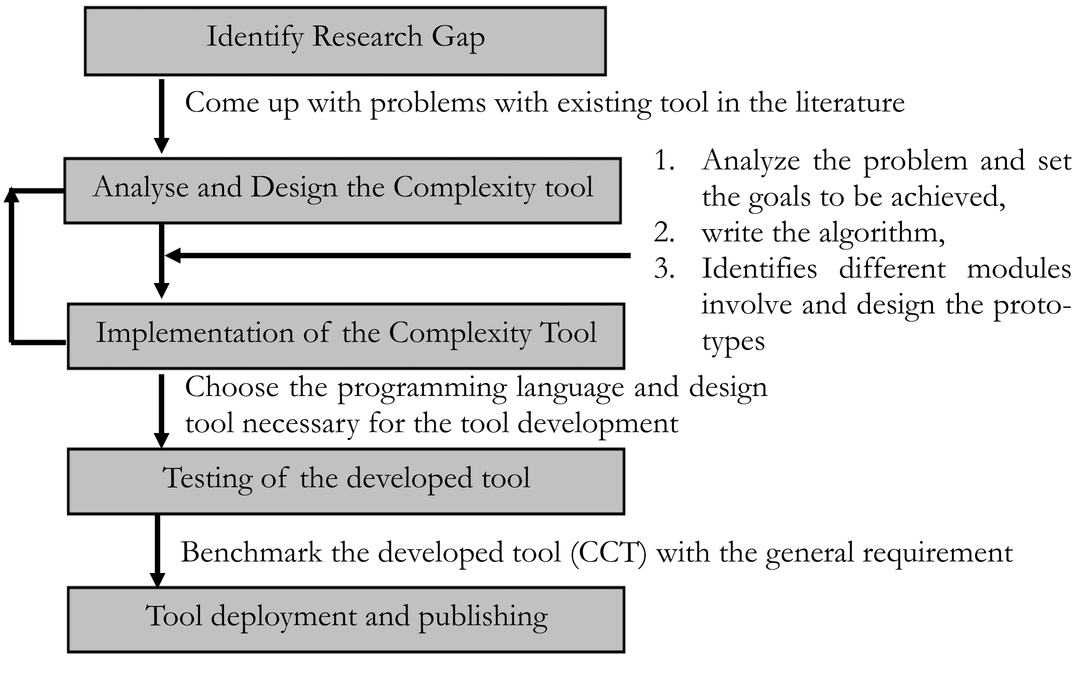

Identify Research Gap
Come up with problems with existing tool in the literature
Analyse and Design the Complexity tool
1. Analyze the problem and set the goals to be achieved,
2. write the algorithm,
3. Identifies different modules involve and design the prototypes
Implementation of the Complexity Tool
Choose the programming language and design tool necessary for the tool development
Testing of the developed tool
Benchmark the developed tool (CCT) with the general requirement
Tool deployment and publishing

Figure 1: Research Methodology/Framework

Brief about the Developed Tool

Complexity Measurement Tool (CMT) is a metric tool that analyzes C, C++, C#, and JavaScript code and produces a list of the methods or functions in the analyzed code. The developed tool is a desktop-based application which is inherently distributive. This distributive characteristic of the tool helps in getting different programming languages to share the metrics and make analysis within them easier and also improves the efficiency of software developers' work. The developed tool takes source codes written in the above-mentioned programming languages as its input, it identifies each method or function in the code, the number of code lines that form each method, then measure the complexity of each of the method and also specify the category of the complexity value. The developed tool will assist developers to produce reliable and efficient software products with good maintenance property. The result from the tool can also be used in making critical decisions such as predicting ease of maintenance, production cost and effort required.

Modules involved in the Developed System

CMT TEST: This module carries out various tests such as the supported languages, block analyzer, configuration file, driver, file analyzer, integration and program tests. The main function of this module is to ensure that all program is well tested before considered for analysis.

CMT ENGINE: This module consists of the engine that carries out the metrics on each supported languages, updates and forward result analysis of the CMT user's interface (CMT GUI).

CMT GUI: This module is the main entry and exit point into the CMT application which defines the user interfaces. The output consists of a form that specifies the category of the source code, unit (Method) been measured, complexity generated for the source code, lines of codes and the file path.

CMTvsPLG-IN: This module provides an add-in extension for Visual Studio Integrated Development Environment (IDE) to enables source codes to be tested from these environments using defined CCM controls. It has been tested with VS 2008 and 2010.

The interrelated modules that make up the developed Complexity Measurement Tool (CMT) with the engine taking the central stage to which all other modules have a specified relationship (Figure 2).

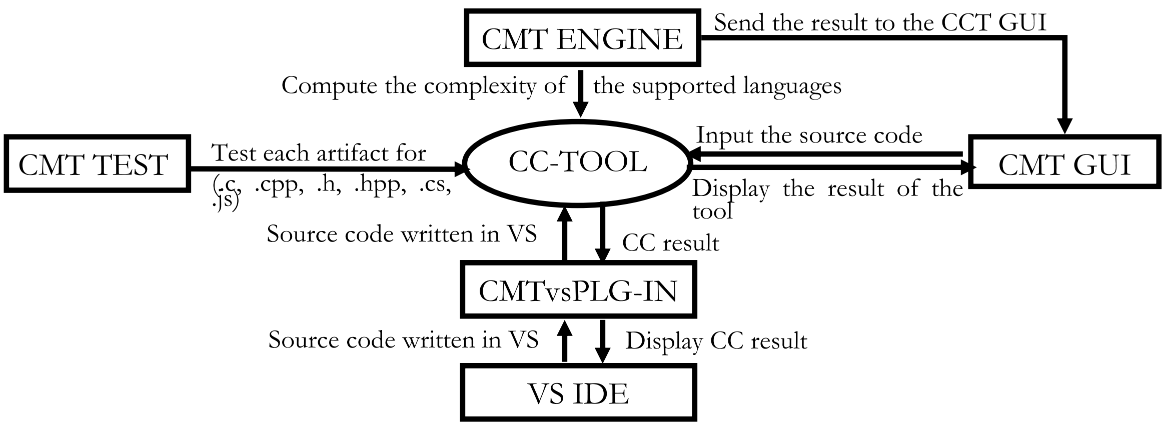

CMT ENGINE
Send the result to the CCT GUI
Compute the complexity of the supported languages
CMT TEST
Test each artifact for (.c, .cpp, .h, .hpp, .cs, .js)
CC-TOOL
Input the source code
Display the result of the tool
CMT GUI
Source code written in VS
CC result
CMTvsPLG-IN
Source code written in VS
Display CC result
VS IDE

Figure 2: The Context Diagram of the Developed CMT Application

Complexity Measurement Tool Algorithm

The algorithm for the developed tool is shown below:

```
Start
Input the source code
if(filename.ToLower().EndsWith(".cpp") | |
filename.ToLower().EndsWith(".cs") | |
filename.ToLower().EndsWith(".h”) | |
filename.ToLower().EndsWith(".hpp") | |
filename.ToLower().EndsWith(".c") | |
filename.ToLower().EndsWith(".js") | |
filename.ToLower().EndsWith(".ts"))
{
Call FileAnalyzer
Call FilePaser
{
Call Calculate ccmNo
{
Call Classification
{
Call ResultOutputer
}
}
}
Call errorMessage (“File not supported”)
}
Call errorMessage (“File not supported”)
End
```

Complexity Measures

This research employed a McCabe Cyclomatic Complexity measure in measuring the code complexity in order to test the efficiency of the tool developed. As discussed earlier in Chapter Two, there are three ways of calculating the software Complexity using McCabe Cyclomatic Complexity measure; the second method (Number of binary decisions + 1) produced by Gill and Kemerer (1991) was adopted in the development of the tool and the results were benchmarked against the standard provided by the Software Engineering Institute, SEI. The complexity results show that the code produced during software development is either risk-free software, having a moderate risk, high risk, or high risk and even not testable software based on the complexity number they produced. The formula and the algorithm are as follow;

Cyclomatic Complexity (CC) = number of decision statements + 1

Where the numbers of decision statements are calculated as follows;

i. Sum of all selection statements which includes if/then, else if but do not count the else statements in the program.
ii. Identify all the switch statement and sum the cases in the program but do not count the default in the program.
iii. Sum all the repetition like for, while and do-while statements and also all the try/ catch statements in the program.
iv. Sum all conditional operator “and” and “or” operator and also ternary operators like?: from the statement.

Then, whatever the value of CC is will be tested as:

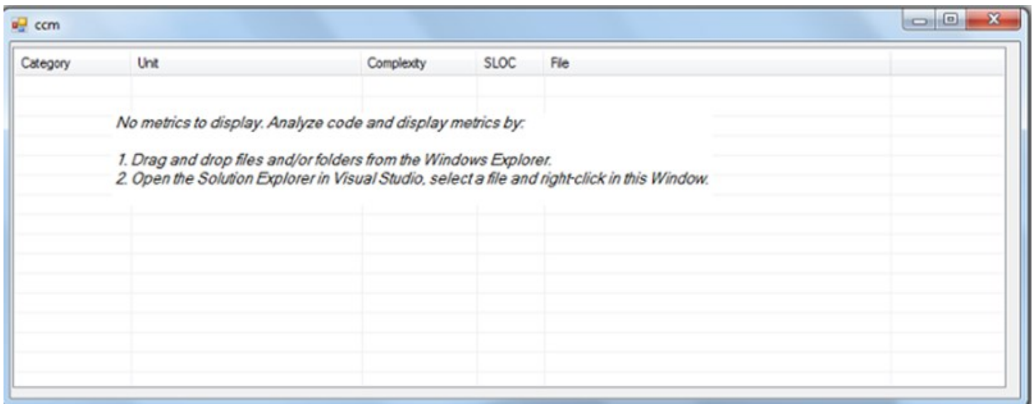
```
if (1< = CC< = 10)  Then
“The program code is categorized as risk-free software”
Else if (11< = CC< = 20)  Then
“Software considered having a moderate-risk”
Else if (21< = CC< = 50)  Then
“The software is having high-risk”
Else
“Considered as of high-risk and even not testable software”
End if.
```

RESULTS AND DISCUSSION

The system works by taking source code artefact written in any of the programming languages as an input, this was scanned to identify the Number of Class/ Method (NOM) contained, the Source Line Of Code (SLOC), the Cyclomatic complexity value, the equivalent category and the acquired file path. This is displayed as an output for future decision making.

CCT Tool Home Page

This is the entry point (interface) of the application where users interact with it. It is the first page that shows up when the application is launch. The page also contains the some information on how to use the tool (Figure 3).



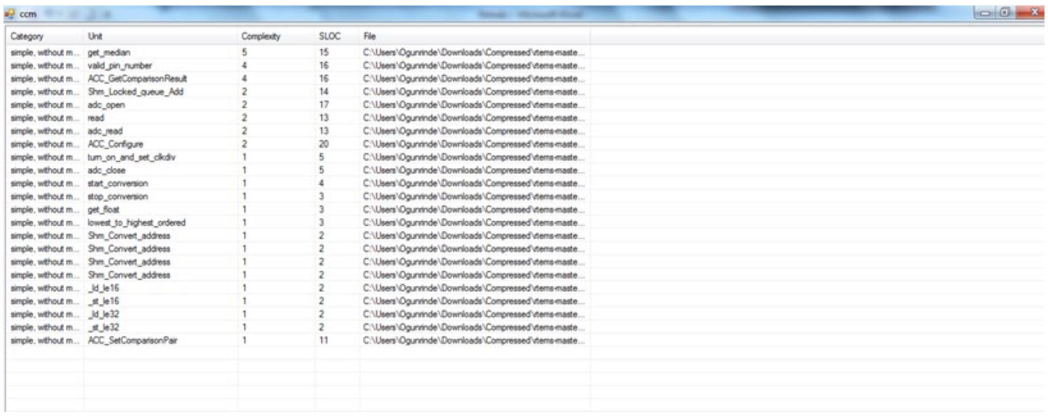
No metrics to display. Analyze code and display metrics by:

1. Drag and drop files and/or folders from the Windows Explorer.
2. Open the Solution Explorer in Visual Studio, select a file and right-click in this Window.

Figure 3: CCT Home Page

Metric Result

The tool accept input by drag and drop the folder containing the source code on the CCT tool home page for processing. The result of the analyses is displayed as shown (Figure 4).



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 4: Interface of complexity result of the inputted Source Code

The part of the tool where the input is accepted and at the same time displays the result of the metrics after the processing (Figure 4). 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

Comparison with an Existing Tool

The CMT was evaluated with an equivalent tool Code Metric (CM) embedded In Visual Studio using some open-source code, the following was observed;

1) The developed tool accepts folder containing all the files in a solution and sort for the appropriate file extension then measures the complexity while the Code Metrics of Visual Studio expects the solutions alone.
2) The developed tool and Code Metric in Visual Studio calculate the complexity the same way but Code Metric does not calculate for JavaScript language and did not present the category in which the complexity value of each method belongs.
3) The developed tool can calculate the complexity of more than one solution or files that contain the targeted programming language at once which proves the efficiency of the tool while in embedded Visual Studio's code metric; only one solution can be calculated at a time.
4) In term of speed, the developed tool calculates at a faster rate compares to Code Metrics in Visual Studio.

Comparisons of Findings with Related Existing Works in the Literature

Results obtained in this study are in consonance with the work of Liu, et al., (2018) who showed that despite Cyclomatic complexity approach to finding the complexity of programming code been old, it remains a significant index to evaluate program code quality and ease of maintenance for wide range of programming languages. The Authors also concluded that the measured cyclomatic complexity values will assist the software stakeholders in making critical decisions such as predicting ease of maintenance, production cost and effort required.

CONCLUSIONS

In today's software development businesses, producing quality and maintainable software takes a great step. Maintainability attributes are one of the factors of software quality. The complexity of code has effects on maintenance activities like software testability, reusability, understandability and modifiability. Software metrics should be able to determine the complexity of code and give reports on the modules that are likely to have problems in future. This research has developed and tested a Complexity Measurement

Tool (CMT) that supports four different programming languages which are C/C++, C# and JavaScript. The tool has useful implications that could assist software designers, programmers, testers and other stakeholders in building quality software, right from the design stage till the maintenance stage. By using the developed complexity tool, the reliability, quality, and ease of maintenance of software product will be greatly enhanced. Equivalent Cyclomatic Complexity Tool for web programming languages could be of interest for future research. This tool should be able to adjust to dynamically to changes in team configuration and requirement at different phases during software lifecycle.

REFERENCES

Alenezi, M., Zarour, M. 2020. On the Relationship between Software Complexity and Security. *International Journal of Software Engineering & Applications* 11(1): 51–60. https://doi.org/10.5121/ijsea.2020.11104

Antinyan, V., Staron, M., Hansson, J., Meding, W., Österström, P., Henriksson, A. 2014. Monitoring evolution of code complexity and magnitude of changes. *Acta Cybernetica* 21(3): 367–382. https://doi.org/10.14232/actacyb.21.3.2014.6

Antinyan, V., Staron, M., Meding, W., Österström, P., Bergenwall, H., Wranker, J., Hansson, J., Henriksson, A. 2013. Monitoring evolution of code complexity in agile/lean software development: A case study at two companies. *13th Symposium on Programming Languages and Software Tools, SPLST* pp.1–15.

Ardito, L., Coppola, R., Barbato, L., Verga, D. 2020. A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review. *Scientific Programming* https://doi.org/10.1155/2020/8840389

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

Debbarma, M. K., Debbarma, S., Debbarma, N., Chakma, K., Jamatia, A. 2013. A Review and Analysis of Software Complexity Metrics in Structural Testing. *International Journal of Computer and Communication Engineering* 2(2): 129–133.

Delange, J. 2015. Evaluating and Mitigating the Impact of Complexity in Software Models. http://www.sei.cmu.edu

Gil, J. Y., Lalouche, G. 2016. When do software complexity metrics mean nothing?- When examined out of context. *Journal of Object Technology* 15(1): 1–25. https://doi.org/10.5381/jot.2016.15.5.a2

Gujar, C. R. 2019. Use and Analysis on Cyclomatic Complexity in Software Development. *International Journal of Computer Applications Technology and Research* 8(5): 153–156. https://doi.org/10.7753/ijcatr0805.1002

IEEE Computer Society 2014. IEEE Standard for Software Quality Assurance Processes- IEEE Std 730™-2014 (Revision of IEEE Std 730-2002)

Liu, H., Gong, X., Liao, L., Li, B. 2018. Evaluate How Cyclomatic Complexity Changes in the Context of Software Evolution. 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 02, 756–761. https://

doi.org/10.1109/COMPSAC.2018.10332

Madi, A., Zein, O. K., Kadry, S. 2013. On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and Its Applications* 7(2): 67–82.

Malhotra, M.P. 2015. Python Based Software for Calculating Cyclomatic Complexity. *International Journal of Innovative Science, Engineering & Technology* 2(3): 546–549.

Meirelles, P., Santos, C., Miranda, J., Kon, F., Terceiro, A., Chavez, C. 2010. A study of the relationships between source code metrics and attractiveness in free software projects. In Proceedings - *24th Brazilian Symposium on Software Engineering, SBES* pp. 11–20. https://doi.org/10.1109/SBES.2010.27

Miguel, J. P., Mauricio, D., Glen, R. 2014. A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications* 5(6): 31–53. https://doi.org/10.5121/ijsea.2014.5603

Mohamed, N., Fitriyah, R., Sulaiman, R., Rohana, W., Endut, W. 2013. The Use of Cyclomatic Complexity Metrics in Programming Performance ' s Assessment. *Procedia - Social and Behavioral Sciences*, 90 (InCULT 2012): 497–503. https://doi.org/10.1016/j.sbspro.2013.07.119

Ogheneovo, E. 2013. Software Maintenance and Evolution: The Implication for Software Development. *West African Journal of Industrial and Academic Research* 7(1): 81–92. Retrieved from http://www.ajol.info/index.php/wajiar/article/view/91395

Oliveira, C. D. De, Black, P. E., Fong, E. 2017. Impact of Code Complexity On Software Analysis. *National Institute of Standards and Technology Kent Rochford.*

Omri, S., Montag, P., Sinz, C. 2018. Static Analysis and Code Complexity Metrics as Early Indicators of Software Defects. *Journal of Software Engineering and Applications* 11: 153–166. https://doi.org/10.4236/jsea.2018.114010

Ostberg J., Wagner, S. 2014. On automatically collectable metrics for software maintainability evaluation, in Proceedings of the 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process And Product Measurement, Rotterdam, 'e Netherlands, October 2014

Pasrija, V., Kumar, S., Srivastava, P.R. 2012. Assessment of Software Quality: Choquet Integral Approach. *2nd International Conference on Communication, Computing & Security (ICCCS)*, 1: 153–162. https://doi.org/10.1016/j.protcy.2012.10.019

Serebrenik, A. 2011. Software metrics. 2IS55 *Software Evolution*, (2). Retrieved from http://www.win.tue.nl/~aserebre/2IS55/2013-2014/9.pdf

Surendra, K. 2020. Cyclomatic Complexity in Software Development, *International Journal of Engineering Research & Technology (IJERT)* 8 (16): 46-47, NCSMSD – 2020.

Tashtoush, Y., Al-maolegi, M., Arkok, B. 2014. The Correlation among Software Complexity Metrics with Case Study. *International Journal of Advanced Computer Research* 4 (2): 414–419. https://arxiv.org/pdf/1408.4523.pdf

Tombe, R., Okeyo, G. 2014. Cyclomatic Complexity Metrics for Software Architecture Maintenance Risk Assessment. *International Journal of Computer Science and Mobile Computing* 3(11): 89–101.

Ukić, N., Maras, J., Šerić, L. 2018. The influence of cyclomatic complexity distribution on the understandability of xtUML models. *Software Quality Journal*, 26(2): 273–319. https://doi.org/10.1007/s11219-016-9351-5

(*Manuscript received: 18th June, 2020; accepted: 24th August, 2021*)