Development of an Enhanced Automated Software Complexity Measurement System

***Sanusi B.A.*[1], *Olabiyisi S.O.*[2], *Afolabi A.O.*[3], *Olowoye, A.O.*[4]**
[1,2,4]*Department of Computer Science,*
[3]*Department of Cyber Security,*
Ladoke Akintola University of Technology, Ogbomoso, Nigeria.
Corresponding Author
E-mail Id:- [1]*sanusibashiradewale90@gmail.com*
[2]*soolabiyisi@lautech.edu.ng*
[3]*aoafolabi@lautech.edu.ng*
[4] *aoolowoye@pgschool.lautech.edu.ng*

ABSTRACT
Code Complexity measures can simply be used to predict critical information about reliability, testability, and maintainability of software systems from the automatic measurement of the source code. The existing automated code complexity measurement is performed using a commercially available code analysis tool called QA-C for the code complexity of C-programming language which runs on Solaris and does not measure the defect-rate of the source code. Therefore, this paper aimed at developing an enhanced automated system that evaluates the code complexity of C-family programming languages and computes the defect rate. The existing code-based complexity metrics: Source Lines of Code metric, McCabe Cyclomatic Complexity metrics and Halstead Complexity Metrics were studied and implemented so as to extend the existing schemes.
The developed system was built following the procedure of waterfall model that involves: Gathering requirements, System design, Development coding, Testing, and Maintenance. The developed system was developed in the Visual Studio Integrated Development Environment (2019) using C-Sharp (C#) programming language, .NET framework and MYSQL Server for database design. The performance of the system was tested efficiently using a software testing technique known as Black-box testing to examine the functionality and quality of the system. The results of the evaluation showed that the system produced functionality of 100, 100, 75, 75, and 100 %, and quality of 100, 100, 75, 75, and 100 % for the source code written in C++, C, Python, C# and JavaScript programming languages respectively. Hence, the tool helped software developers to view the quality of their code in terms of code metrics. Also, all data concerning the measured source code was well documented and stored for maintenance and functionality in the possibility of future development.

Keywords: *Software Metrics, Code-based Complexity, Sources Line of Code metric, McCabe Cyclomatic, Halstead Complexity.*

INTRODUCTION

Computational Complexity hypothesis focuses on identifying computational issues as claimed by their built-in problems and correlate these categories to one another [15]. These issues are task resolved by a system using other problem-solving operations mainly by a computer [15]. One of the functions of the hypothesis is establishing feasible limitation of what the computer can execute.

In software lifetime development, one of the significant concerns is software complexity. The source code complexity metrics are used to measure the quality of software properties. Software complexity in order words predicts analytical information about reliability, testability, and maintainability of program source code from the automatic measurement of the code [12]. A complex source code will therefore be difficult to develop and maintain by the software developer.

Recently, it's important for software engineering to correctly predict the complexity of a program source code to save millions of people in maintaining time and effort [16]. A software metric can be defined as the quality of measuring the level to which a software system or operation possesses some attribute. However, the metrics and measurements are often used but metrics are said to be basis while measurements are the figures obtained by the application of metrics. According to literature quantitative measurements are vital in all sciences, these leads to ceaseless effort by computer science practitioners and theoreticians to bring similar perspective to software development [10]. In this paper the code-based complexity metrics, Source Lines of Code metrics (SLOC), McCabe Cyclomatic Complexity metrics and Halstead Software Science metrics were used to obtain the objective reproducible measurements that can be useful for quality assurance, performance, debugging management, and defects rate. The Software metrics have been accepted under the idea that before any source code can be measured or quantified, it needs to be translated into numbers. There are several areas where software complexity metrics are found to be useful. These areas simply include phases from software planning to the steps that are meant to improve the quality of certain software. However, the software system can't perform on its own without human interaction. Therefore, the software complexity metric is also a measure of a person's relation to the software that he or she is handling [14].

RELATED WORK

In [8], Jetter conducted a research on how to apply the Bansiya software quality model to evaluate the development of 19 Azureus versions by making comparisons using the object-oriented metrics rather than the metrics proposed by Bansiya and his colleagues [1]. The researcher demonstrated the ability of this model to track the evolution of design quality in several versions by providing access to important information in the internal life of the software. This information can simply support design decisions at higher levels of abstraction. His suggested model may require additional inputs to cover the highest levels of abstraction to assess all aspects of the quality model at this level [8].

Zhang and Baddoo [18], studied the performance of three complexity metrics. They selected McCabes Cyclomatic Complexity, Halstead's Complexity, and Douces Spatial Complexity. Their experiment is established on four hypotheses using dataset from Eclipse JDT which is an open-source application. Sequel to the result, they conclude that the three complexity metrics show different performance results during their hypotheses testing, and finally they recommended combining Cyclomatic complexity metric and Halstead metrics for better decisions on software complexity.

Another study by Panovski [13] established a new assessment of software product quality, which focused on assessing the quality of the external features of the software product, which means evaluating the behavior of the software product when implemented. In addition, the study focused on the development of the quality model (ISO / IEC 9126) at the level of software metrics. The study is based on seven samples of the software product and assesses them using

ISO / IEC 9126-2 quality model. In the research, Panovski [13] concluded that external product quality attributes are an area or category that can be used, and that the metrics provided by ISO / IEC 9126-2 can be regarded as a starting point for the definition of standards, but are not ready to use in their present form. The software metrics of the software product need to be more adapted to give better information [13].

Borchert [3] examined the technique of code profiling by using a static analysis. The study was done on 19 industrial samples and 37 samples of students' programs. He has analyzed software samples through software metrics. The results of this study recommended that the code pattern could be a useful method for rapid program comparisons and quality observation in the field of industrial application and education.

Jay *et al*. [7] compared two metrics to calculate the complexity of code. They used McCabe Cyclomatic Complexity metric and Line of code metric (LOC) to prove the stable linear relationship between these two techniques. They used 5 NASA datasets with different programming languages, such as C, C++, and Java, to be the dataset for their research.

Bhatti [2] explored the research area occupied by the software complexity metrics. He made use of a public available QA-C tool to automatically measure software metrics on the code written in C programming language by expressing the association between software metrics and the complexity of the source code. He also attempted to evaluate the values of these metrics graphically only, without considering the quality features and threshold limits relationship [2].

Another work in [17] is the impact of code complexity and usability, either in monitoring software complexity during development, or in evaluating the complexity of legacy software system. The researchers of this study, Widheden and Goran [17] suggested a new coupling metrics (Ecoup), and introduced the Java met tool, which works in a static analysis of programs written in Java programming language with respect to coupling, flow control, complexity and coherence. In the same year, Chandra *et al*. suggested the use of Object Oriented metrics that introduced by Chidamber and Kemerer [5] to assess program quality at the class level. The suggested tool can be used to verify the class design conforms to the design specifications of the Object Oriented programming, through using the threshold for each metric [4].

According to Silva et al. [16 who worked on the relevance of three software complexity metrics: McCabe's Cyclomatic complexity, Halstead's complexity, and Shao and Wang's cognitive functional size. In the research work, 10 different programs of the same Java programming language were used and evaluate which one of the three complexity metrics is most suitable for software manufacturing. In order to calculate manually the ranking of the ten programs complexity based on the three metrics, quota sampling method was applied by selecting five different companies and randomly asked six programmers from each company to rank the complexity of the ten programs.

Hence, this research studied and investigated code-based complexity metrics, Source Lines of Code metrics (SLOC), Halstead, and McCabe Cyclomatic Complexity metrics, and designed a system that will automatically measure the software complexity of C-family programming languages as well as compute the defect rate.

METHODOLOGY

The waterfall model of software engineering was adopted in this paper. It is a linear sequential procedure that flows down in respect to its importance or an organized process to make sure that every stage is followed carefully before moving on to the following step. The steps taken are hereby highlighted:

1. Requirement Engineering**:** This phase involved the gathering of required information online from different websites, eBooks and reviewing of existing software metrics applications.
2. System Design**:** The requirements gathered during phase one was the primary directing tool that was used in designing the system which measures the complexity as well as compute the defect rate of a code fragment.
3. System Coding: This phase deals with programming or development of the designed tool and creation of the database using C# programming language, .NET framework, and MYSQL server for database design.
4. System Testing: The system was tested efficiently by making sure no invalid data has been entered in forms in other not to disorganize the system, all pages were duly tested and the system was tested against exceptions and how they are handled. Black-box testing was adopted in this research since it focuses on the functionality of the system.
5. Maintenance: The system maintenance was done on different phases during the development stage of the system while the code was maintained whenever new information is gathered with respect to the objects used on the system.

SELECTION OF METRICS

In this paper, the goal is to examine and implement a way in which automated measurement of source code complexity is possible to perform. One of the objectives of this paper is to find suitable metric or a group of metrics that can indicate the complexity of a system. Therefore, some of software code metrics have been studied.

Source Line of Code (SLOC)

This metric is used to measure the quantitative characteristics of program source code. This metric is based on counting the lines of the source code. The limitations of SLOC may give an impression of a less useful metric, but wise use of this metric can still be valuable. Specifically, to the requirements of this paper, SLOC can simply be used to monitor change in terms of lines of code metric between different build versions of the software.

This software metric can be calculated for individual functions of a program. For instance, if a function is too large in comparison with the average length per function, it may indicate that the function is hard to maintain and hence complex in terms of maintainability. The unit of this metric is a simple number that indicate the number of lines in a source code file. This number can simply be used with other code metrics to derive new complexity indicator that can be more comprehensive.
LOC is usually represented as:

1. kLOC: thousand lines of code metrics
2. mLOC: million lines of code metrics

Halstead Software Science

A suite of metrics was introduced by Halstead in 1977 [6]. This suite of metrics is known as Halstead software science or as Halstead metrics. Most of the product metrics typically apply to only one particular aspect of a software product. In contrast, Halstead set of metrics applies to several aspects of a program, as well as to overall production effort.

The Halstead metrics are established on the following indices:

1. n_1- distinct number of operators in a

program
2. n_2 - distinct number of operands in a program
3. N_1- total number of operators in a program
4. N_2 - total number of operands in a program

Halstead formulas

On the basis of above-mentioned indices (n_1, n_2, N_1, N_2), Halstead derived more than one formulas relating to the properties of program code. These formulas can measure Program Vocabulary (n), Program Length (N), Program Volume (V), Program Difficulty (D), Program Level (L), Total Effort (E), Development Time (T) and Number of Delivered Bugs (B). Halstead named his formulas as "Halstead's Software Science Metrics".

Program vocabulary $n = n_1 + n_2$ (1)

Program length $N = N_1 + N_2$ (2)

Program volume $V = N * log_2 n$ (3)

Potential Difficulty $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$ (4)

Program level $L = 1 / D$ (5)

Total effort $E = D \times V$ (6)

Development time $T = {}^{E}/_{S}$ (7)

Value of S is usually taken as 18 for these calculations.

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation.

$$B = \frac{E^{\frac{2}{3}}}{3000} \quad (8)$$

McCabe Cyclomatic Complexity

McCabe [11] established a metric based on the control flow structure of a program. This metric is known as McCabe cyclomatic complexity metric and it has been famous code complexity metric throughout since it was first introduced. The McCabe cyclomatic complexity metric is based on measuring the linearly independent path through a program and gives cyclomatic complexity of the program which is represented by a single number.

McCabe noted that a program consists of code chunks that execute according to the decision and control statements, e.g. if-else and loop statements. McCabe metric ignores the size of individual code chunks when calculating the code complexity but counts the number of decision and control statements. McCabe cyclomatic complexity method maps a program to a directed, connected graph. The nodes of the graph represent decision or control statements. The edges indicate control paths that define the program flow. Cyclomatic complexity is calculated as:

$$M = E - N + P \quad (9)$$

where
M = McCabe metric,
E = the number of edges of the graph of a program,
N = the number of nodes of the graph and
P = the number of connected components.
P can also be considered as the number of exits from the program logic. The recommended ranges are shown in Table 1

***Table 1:-** McCabe Cyclomatic Complexity Ranges*

Cyclomatic Complexity	**Code Complexity**
1 – 10	A simple program, without much risk
11 – 20	More complex, moderate risk
21 – 50	Complex, high risk
50+	Untestable, very high risk

Requirement Analysis

The system was developed with the goal of automated measurement of software complexity and increasing software reliability by measuring its defect rate. A defect rate is the percentage of output that fails to meet a quality target and also calculated by testing output for noncompliance to a quality target [9]. The following formula can be used to calculate defect rate;

$$Defect\ rate = \frac{Number\ of\ Defects}{Lines\ of\ code} \times 100 \qquad (10)$$

Code defects are commonly measured as defects per one thousand lines of code, and can be calculated with the following formula;

$$Defects\ per\ thousand\ lines\ of\ code = \frac{Number\ of\ Delivered\ bugs}{1} \times \frac{Lines\ of\ code}{1000} \qquad (11)$$

The requirements needed to gather for every analysis carried out from the software being developed take a main requirement which was a source project or source code for another C# or other C-family programming languages where the software application run some metrics and calculations with respect to the source code supplied. Figure 1 represents the block diagram for the developed system to work effectively and carry out a particular task. In this paper, the net satisfaction index ranging from 0 to 100% was discussed and agreed with the software developers in the underlying test. The net satisfaction index rating scale maps ratings between completely dissatisfied and completely satisfied to numbers between 0 and 100% as seen in Table 2.

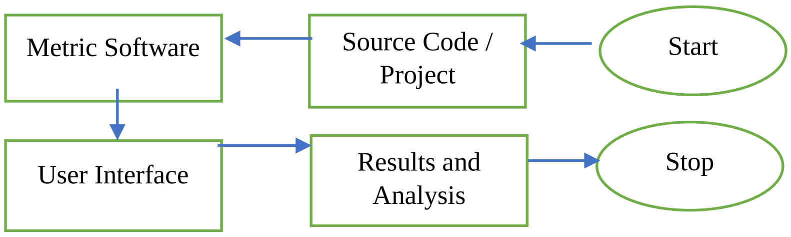


Fig.1:-*The Block diagram of software metric application*

Table 2:-*The Net Satisfaction Index Ranges*

Net Satisfaction Index Rating	**Remark**
100%	Completely Satisfied
75%	Satisfied
50%	Neutral
25%	Dissatisfied
0%	Completely Dissatisfied

RESULTS AND DISCUSSION

The selection of the measured source code was made after consulting with some software developers. Each software developers have different view about their software package before the complexity measurement. As stated, code-based metric Source Lines of Code metrics (SLOC), Halstead Software Science metrics and McCabe Cyclomatic Complexity metrics are used during the development of this tool. However, the implementation of each complexity metric is discussed in the below sub-sections. Table 3 represents the complexity values of the measured source code.

Source Line of Code Metric (SLOC)

In evaluating the Source Line of Code metric (SLOC), this tool automatically counts and calculate the number of Blank Lines of Code (BLOC), Comment Lines of Code (CLOC) and Source Lines of Code (SLOC) which includes the actual code (logic and computation) and declarations in the source code while BLOC and CLOC improves readability and helps in understanding the code during maintenance. SLOC metric is used to calculate the quantitative attributes of the program source code and gives the software size estimations. However, in increasing the software package reliability the defect rate was measured with the calculated number of delivered bugs and the lines of code.

Halstead Software Science Metric
In evaluating the Halstead set of metrics, this tool counts the distinct number of operators, a distinct number of operands, the total number of operands, and the total number of operators in a program source code. Considering a C program, examples of unique operators are main, ( ), { }, int, scanf, &, =, +, /, printf and so on. Also, examples of operands are a, b, c, 3, avg, %d among others. The tool searches for all the distinct operands and operators in the software package and counts how many times each distinct operator or operands appears in the source code, then calculate the total and gives the output as the total number of operands and operators. Once these results have been computed then the values are used to calculate the Halstead Complexity metrics as stated in each formula discussed in the previous chapter. The accuracy of the tool is determined while evaluating every possible place in the software package that could have a predefined operator.

However, SLOC metric is typically applied to only one particular aspect of the source code. In contrast, the Halstead set of metrics applies to several aspect of a program source code as it is good to have more than one metric for the quantitative measure of a program and not to fully rely on one. Although, Halstead set metrics are difficult to compute manually as it is not easy to count all the operands and operators if a program is using a large number of operands and operators. Hence, in this research automatic measurement of source code is achieved by developing a tool that solves the limitations of the existing tool.

McCabe Cyclomatic Complexity
McCabe is a measure of the complexity of a module's decision structure. In evaluating the McCabe Cyclomatic Complexity metrics, this developed tool looks for IF, FOR, WHILE, CASE statements in the source code and counts them to calculate the McCabe complexity value. It counts the number of decision or control structures and maps a program to a directed, connected graph. The branches of the graph are regarded as decision or control statements and the edges indicate a control path that defines the program flow. Nevertheless, due to the fact that no organization or software company will want to release their source code and because of the license time restrictions two senior software programmers were consulted. The selected source code for performing the code analysis is Python and C#.

The software developer's opinion about Python was that the package is well structured and designed which should possibly contain less complex code. In other words, the software package C# was supposed to be more complex as said by the software developer. Hence, the McCabe Cyclomatic metric used in this tool results for a fast assessment and from the calculated results the Python software package has less complex code while the C# software package has more complex code which relates to the software developer's opinion. Figure 2 represents the calculated result of the Python software package and Figure 3 shows the calculated result of the C# software package.

Evaluation of the Developed Tool
This developed tool has made it possible for software testers to indicate which part of the code needs refactoring and stores the generated report in the database so as to monitor the code changes between the releases. Also, this tool has the functionality to check up to 500,000 lines of code.

However, black-box testing method was used to test the tool based on requirements and functionality by the software developers because it is a software testing method which evaluates the functionality of an application without looking into its internal structure or workings that is, in the testing method, the structure and design of

the source code are not disclosed to the software tester, software developers and end-users perform this test on software.
The software programmers were chosen in the software testing phase and the test was conducted on the system. Hence, this black-box testing method attempts to find errors in functionality and quality of the developed system as represented in figure4. The functionality is performed by the end-users on providing the input and the output equals the desired results while the quality is the level to which the system meets specified requirements. The overall average for functionality and quality of the developed system results to 90% and 90% respectively.

***Table 3:**-Complexity Values of the Measured Source Code*

Metrics Attributes		**C++**	**C**	**Python**	**C#**	**JavaScript**
Source Lines of Code Metric	BLOC	4	314	291	245	316
	CLOC	0	315	210	156	293
	SLOC	82	2005	1807	1162	1828
	Total LOC	86	2634	2308	1563	2437
Halstead Metric	Program Length (N)	135	348	15881	11714	290
	Vocabulary(n)	59	69	1411	706	49
	Program Volume (V)	794.157	2125.767	166154.999	110855.725	1628.266
	Difficulty (D)	28.517	56.649	256.429	501.995	63.542
	Program Level (L)	0.035	0.018	0.004	0.002	0.016
	Total Effort (E)	22646.975	120422.575	42606960.240	55645915.710	103463.278
	Program Time (T)	1258.165 Sec.	6690.143 Sec.	2367053.347 Sec.	3091439.762 Sec.	5747.960 Sec.
	Delivered Bugs (B)	0.268	0.816	40.902	48.875	0.738
Cyclomatic Complexity	No. of Methods	3	2	98	62	2
	Defect rate	0.327	1.636	73.910	56.793	1.349

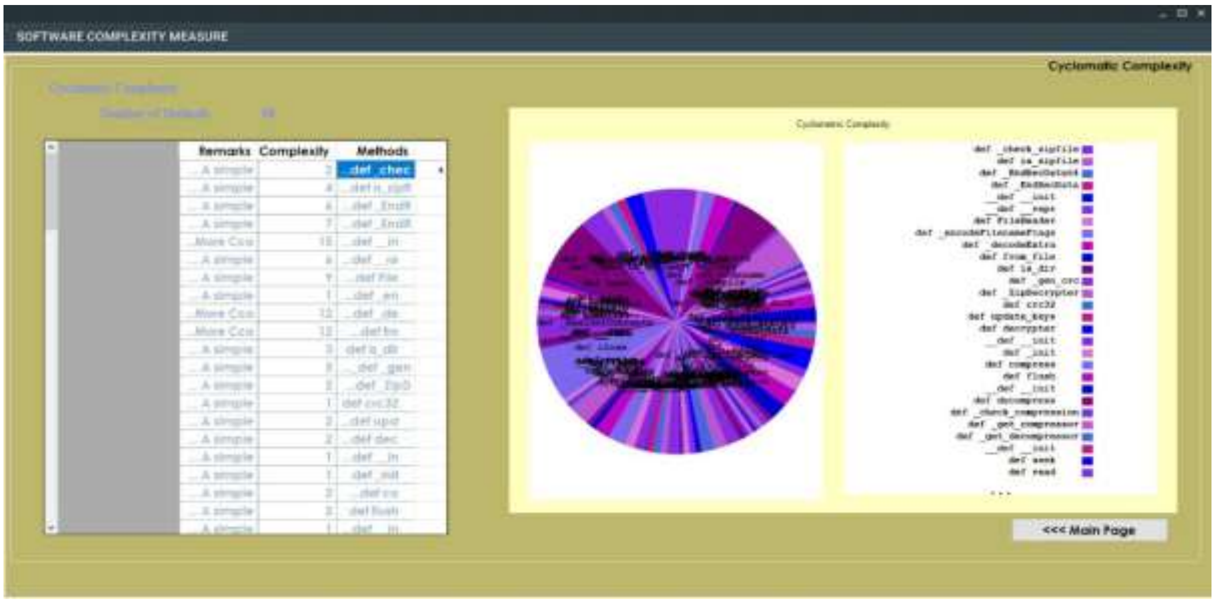
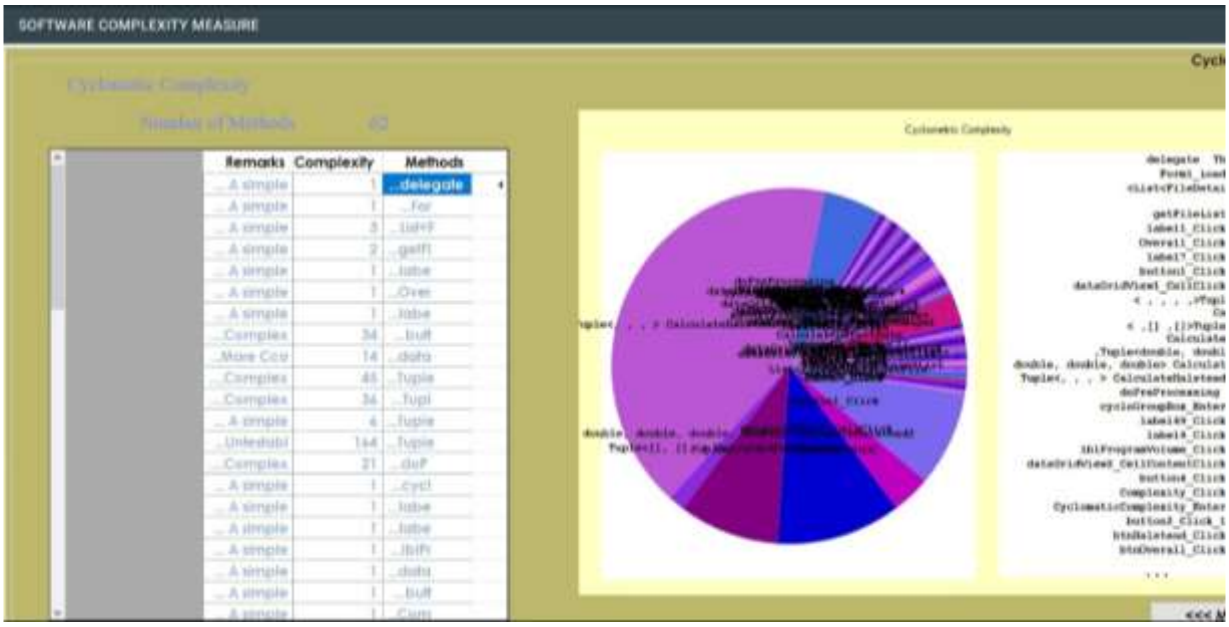


***Fig.2:**-Desktop Interface Showing Python Software Package Result*


SOFTWARE COMPLEXITY MEASURE

Cyclomatic Complexity

Number of Methods 40

Remarks	Complexity	Methods
...A simple	1	...delegate
...A simple	1	...For
...A simple	3	...List<F
...A simple	2	...getFi
...A simple	1	...labe
...A simple	1	...Over
...A simple	1	...labe
...Complex	34	...butt
...More Co	14	...data
...Complex	45	...Tuple
...Complex	36	...Tupl
...A simple	6	...Tuple
...Unleshabl	144	...Tuple
...Complex	21	...doP
...A simple	1	...cycl
...A simple	1	...labe
...A simple	1	...labe
...A simple	1	...lblPr
...A simple	1	...data
...A simple	1	...butt
...A simple	1	...Com

Cyclomatic Complexity

```
delegate  Th
Form1_Load
<List<FileDetai

getFileList
label1_Click
Overall_Click
label7_Click
button1_Click
dataGridView1_CellClick
< , , , ,>Tupl
Ca
< ,[] ,[]>Tuple
Calculate
,Tuple<double, doubl
double, double, double> Calculat
Tuple<, , , > CalculateHalstead
doPreProcessing
cycleGroupBox_Enter
label49_Click
label8_Click
lblProgramVolume_Click
dataGridView2_CellContentClick
button4_Click
Complexity_Click
CyclomaticComplexity_Enter
button2_Click_1
btnHalstead_Click
btnOverall_Click
...
```

<<< M

Fig.3:-*Desktop Interface Showing C# Software Package Result*

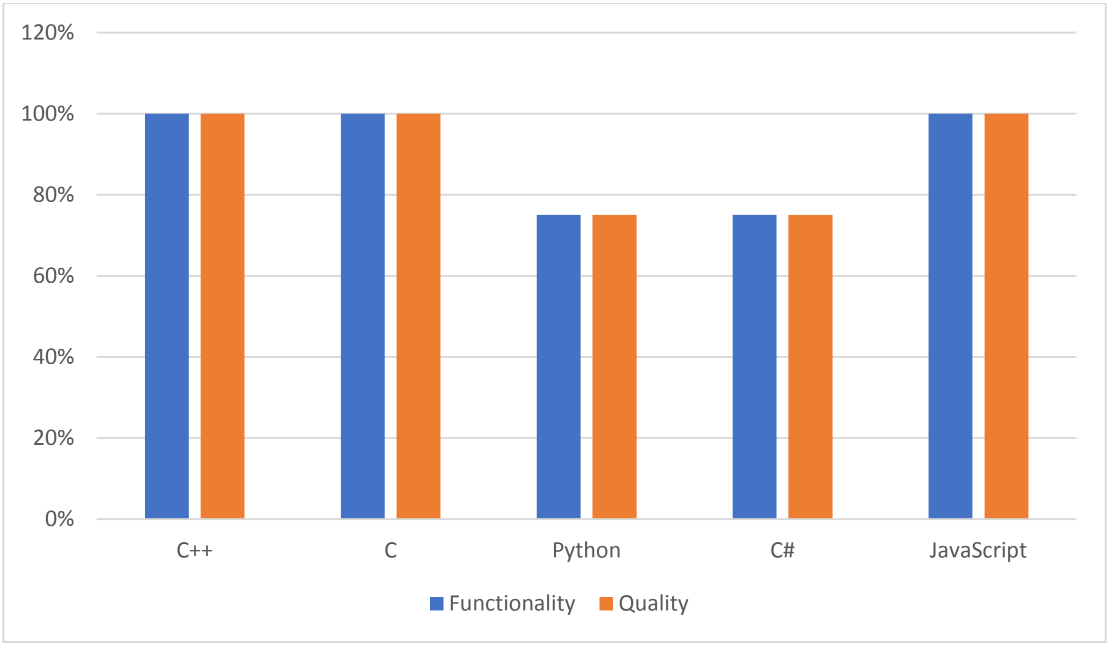


Fig.4:-*Summary of Black-box Testing for each Software Developer*

CONCLUSION

As the demand for software is growing at an exponential rate, the complexity of software is also increasing. The source code complexity measure seems to be the most capable measure for both the quantitative and control flow of the software project. In order words, understanding and measuring the quality and functionality of a software is significant to relate it to measurable quantities. Furthermore, establishing a reason for gathering metrics is essential to ensure relevant metrics are gathered because too many metrics become confusing if they are not required.

In this paper, code-based metrics Source Line of Code metrics (SLOC), McCabe Cyclomatic Complexity and Halstead Complexity Metrics were studied and used to achieve the goal of developing an automatic software complexity measurement tool. The metrics provide visibility and control for the complex software development procedure and therefore, they are a valuable tool for providing guidance on making the software development process better as well as meeting organizational goals to enhance software productivity and quality.

This tool was successfully developed and tested efficiently to run on Microsoft Windows operating system. However, the software testing is a significant stage of the software development life-cycle. In order words, it is necessary to optimize test cases and generate them automatically so as to reduce testing time, effort and cost. Hence, the black-box testing method was used in this research because it focuses on evaluating the functionality of the system.

REFERENCES

1. Bansiya, J. (2000). A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*.2000.28(1):4-17p.
2. Bhatti, H. R. (2010). *An Automatic Measurement of Source Code Complexity*. Master's Thesis, Department of Computer Science, Electrical and Space Engineering, Lulea University of Technology.2010:1-14p.
3. Borchert, T. (2008). *Code Profiling: Static Code Analysis*. Master's Thesis, Department of Computer Science, Karlstad University, Sweden.2008:12-16p.
4. Chandra, E., Linda, P. and Edith, A. (2010). *Class Break Point Determination Using CK Metrics Thresholds*, Global Journal of Computer Science and Technology.2010.10(14):73-77p.
5. Chidamber, S. R. and Kemerer, C. F. (1994*). A metrics suite for the object-oriented design*. IEEE Transactions on Software Engineering.1994.20 (6):476-498p.
6. Halstead, M. (1977). *The Elements of Software Science, Operating and Programming Systems Series*, Elservier Computer Science Library North Holland N. Y. Elsevier North-Holland, Inc. ISBN 0-444-00205-7.1977:1–6p.
7. Jay, G., Hale, J. E., Smith, R. K., Hale, D., Kraf, N. A. and Ward, C. (2009). *Cyclomatic complexity metric and lines of code: Empirical evidence of a stable linear relationship*, J. Software Engineering & Applications.2009:7p.
8. Jetter, A. (2006). *Assessing Software Quality Attributes with Source Code Metrics*. Diploma Thesis, Department of Informatics, University of Zurich. 2006:45-48p
9. John Spacey (2017). How Defect Rate is Calculated. https://simplicable.com/new/defect-rate.
10. Lincke, R., Lundberg, J. and Löwe, W. (2008). Comparing software metrics tools. *In Proceedings of the 2008 International Symposium on Software Testing and Analysis. ISSTA'08*. 131–142p. New York, NY:ACM.
11. McCabe, T. J. (1976). *A Software Complexity Measure.* IEEE Transactions on Software Engineering.1976.SE-2(4):308-320p.
12. Olabiyisi, S. O., Omidiora, E. O. and Sotonwa, K. A. (2013*). The Comparative Analysis of Software Complexity of Searching Algorithms Using Code Based Metrics. International Journal of Scientific and Engineering Research.* ISSN: 2229-5518. 2013.4(6):1-2p.
13. Panovski, G. (2008). *Product Software Quality*. Master's Thesis, Department of Mathematics and Computing Science, Technische Universiteit Eindhoven. 2008:1-24p.

14. Sam, M. (2008). *Areas of Use for Software Metrics*. [Online: accessed on 2010-06-16 from http://www.articlesbase.com/management-articles/areas-of-use-for-software-metrics306127.html]
15. Sanjeev, A. and Barak, B. (2009). Computational Complexity. A Modern Approach, Cambridge. ISBN 978-0-521-42426-4. Zbl 1193.68112.2009:1-8p.
16. Silva, D., Koadagoda, N. and Perera, H. (2012). Applicability of three complexity metrics, in Proc. *International Conference on Advances in ICT for Emerging Regions*. 2012:12-16p.
17. Widheden, K and Göran, J (2010). Software Complexity: Measures and Measuring for Dependable Computer Systems ", MASTER'S THESIS, Department of Computer Science and Engineering, University of Gothenburg, Goteborg, Sweden. 2010:1-14p.
18. Zhang, M. and Baddoo, N. (2007). Performance comparison of software complexity metrics in an open source project, *In Proc. 14th European Conference Software Process Improvement*, Potsdam, Germany.2007:1–16p.