Hybrid Technique for Java Code Complexity Analysis

[1]Nouh Alhindawi
[1]Faculty of Science and Information Technology
Jadara University
Irbid, Jordan

[2]Mohammad Subhi Al-Batah
[2]Faculty of Science and Information Technology
Jadara University
Irbid, Jordan

[3]Rami Malkawi
[3]Faculty of Information Technology and Computer Science
Yarmouk University
Irbid, Jordan

[4]Ahmad Al-Zuraiqi
[4]Faculty of Science and Information Technology
Jadara University
Irbid, Jordan

***Abstract*—Software complexity can be defined as the degree of difficulty in analysis, testing, design and implementation of software. Typically, reducing model complexity has a significant impact on maintenance activities. A lot of metrics have been used to measure the complexity of source code such as Halstead, McCabe Cyclomatic, Lines of Code, and Maintainability Index, etc. This paper proposed a hybrid module which consists of two theories which are Halstead and McCabe, both theories will be used to analyze a code written in Java. The module provides a mechanism to better evaluate the proficiency level of programmers, and also provides a tool which enables the managers to evaluate the programming levels and their enhancements over time. This will be known by discovering the various differences between levels of complexity in the code. If the program complexity level is low, then of the programmer professionalism level is high, on the other hand, if the program complexity level is high, then the programmer professionalism level is almost low. The results of the conducted experiments show that the proposed approach give very high and accurate evaluation for the undertaken systems.**

Keywords*—Complexity; java code; McCabe; Halstead; hybrid technique

I. Introduction

Java language is considered as one of the languages that has various advantages, these advantages includes its simplicity, safety, strength, impact, high level object-oriented ability, and many other advantages [1]. Complexity can here be defined as, the relationship between the internal parts of the program and how these parts can be interacted with each other, some of these parts will be connected to other parts of the program to make the program more complex and difficult to be analyzed or maintained. However, if these parts are less cohesive then the program will be less complex, in this case, the analysis would be easier to be analyzed and maintained [2]. The benefits of complexity measurement can be summarized as follows:

a) Complexity analysis of code can even be estimated from a design (whenever the design is easy and simple then the code will be less complex, in contrast, if the design is more complex and unclear, then the program will be more complex).

b) The ability to distinguish between the simple and more complex program (allow the programmers to write a program in a way that has the following features: high quality, easy to understand, has few mistakes, easy to use and re-use, easy to maintain, easy to test, saves time and lower cost).

c) Good Complexity Measure provides continuous feedback (allowing us to follow the program continuously and to avoid most of the expected mistakes or problems).

TABLE I. Level of Complexity by Mccabe Measure

Complexity	Risk Evaluation
1-10	A simple module without much risk
11-20	A more complex module with moderate risk
21-50	A complex module of high risk
51 or more	An untestable program of very high risk

Categorizing any source code complexity into good or bad will be helpful for code maintenance and evolution. Typically, the source code with good complexity is more maintainable, testable, understandable, and have less errors. On the other hand, any source code with bad complexity will be complex to be maintained, tested, understood by developers, and it will have a lot of errors.

Shrivastava [3] presented a measurement to provide a single ordinal number to be used to compare the program's complexity with other programs. This measurement used McCabe Complexity measures to analyze the system and find the complexity of the program, as follows:

$$CC=E - N+ P$$

where

CC = Cyclomatic Complexity

E = Number of edges of the graph

N = Number of nodes of the graph

P = Number of connected components

The following is an example

```
publicvoid ProcessPages()
{
while(nextPage !=true)
 {
if((lineCount<=linesPerPage) && (status !=
Status.Cancelled) && (morePages == true))
} }
```

As shown in the above example, the routine is starting by adding 1 to the *while loop*, adding 1 to the if *statement*, and adding1 to each && for a total calculated complexity of 5.

Davis and LeBlanc [4] studied a predictive value of various syntax-based problem complexity measures; they discussed McCabe and Halsted Complexity measures and analyzed the system to find the complexity of the program. Sheppard et al [5] compared three types of existing standard measures to find the complexity of the program, they used Halstead, McCabe's, and the length that measured by number of statements to analyze the system and find the complexity of the program.

Prabhu [6] applies McCabe's cyclomatic complexity and the Halstead metrics to evaluate the complexity of Simulink models. Prabhu notes that, the challenge of switching from programming languages to models is that, metrics have to be tailored and values obtained at the code or model level so that computed values are different. Olszewska [7] introduced new metrics specific to high-level design. They focus primarily on model counting, such as the average number of blocks per layer or the stability of the number of inputs/outputs across the model. Toularkis [8] distinguished between two classes of complexity measures which are: dynamic complexity measure and static complexity measure. Dynamic complexity to measure the amount of resources consumed during computation and static complexity to measure the size (e.g. program length) or structural complexity. Olabiyisi et al. [9] applied different software complexity metrics to searching algorithms, and the result showed that for both linear and binary search techniques, the languages do not differ significantly, therefore it is concluded that any of the programming languages is good to code linear and binary search algorithms.

Software complexity is different at the architecture level, where it is defined by how components communicate and are integrated, than at the code or behavior level, where it is defined by how components are implemented [10]. Delange et al. [11] demonstrate that maintaining low-complexity components and delivering high-quality models reduce maintenance activities and associated costs. Banker [12] estimates that software complexity itself can increase maintenance costs of commercial applications by 25% and increase the total lifecycle costs by 17%. Considering not only that safety-critical applications have stringent quality requirements but also that both the software and models of such applications must be maintained for decades, the real costs could be higher than these estimates for critical applications.

There is substantial evidence that cyclomatic complexity is linearly correlated with product size [13]. Evidence shows that software complexity has increased significantly over time not only because of the increase in number of functions but also because of a paradigm shift in which more functions are realized using software rather than hardware [14]. The SEI's experience with high-reliability systems has been that a high-quality process leads to a low number of defects and reduces rather than increases cost [15], [16]. Nonetheless, actual industry practice and estimates of cost for high-reliability software vary widely [17]. Shull reports increases to development costs ranging from 50% to 1,000% due to more coding constraints and certification requirements (e.g., testing, validation) [18].

To the best knowledge, most of the previous modules used only one technique or one theory to measure the ratio of complexity of the programs. So the contribution of this paper is integrating two theories that are called Halstead and McCabe. In this way, the ratio of complexity will be more accurate, which helps programmers to make sure that their programs will be better and their work is more efficient. Whenever the ratio of complexity is more complex, then it will increase the mistakes and errors in the program, thus, the difficulty in maintenance and testing will be increased and the cost of the program will also be increased [19], [20]. For this reason, the ratio of complexity must accurately be measured to be more efficient, contains less errors, easy to test, easy to understand, easy to maintain and test, then this will decrease the cost of the program.

Microsoft visual studio 2010 with language (C#) are used for building the program which has been written to allow users to open any program written in Java, analyze the code, extract all Operators and Operands, all number of edges, number of nodes, and number of connected components, then finding the complexity measurement that allows to identify both the program and programmer levels is done.

This paper consists of five other sections organized as follows: Section 2 discusses McCabe and Halstead complexity measures, Section 3 includes the proposed approach, Section 4 contains the evaluation and discussion, Section 5 about the related work, and finally, Section 6 presents the conclusion and recommendations.

II. MCCABE AND HALSTEAD COMPLEXITY MEASURES

This paper focuses on McCabe [21] and Halsted Measures [22], here each mechanism will be discussed in more details.

A. McCabe Complexity

This theory is being used widely since it was issued; it depends on computing and controlling flow graph of the program, and measuring the number of linearly-independent paths [23]. Tables 1, 2 and 3 show an example about McCabe along with the complexity, $C = E - N + 2P$, where E is the number of edges, N is the number of nodes, and P is the number of connected components.

The following is an example of McCabe complexity Measure as shown in Fig. 1.

```
public static void sort(int x []) {
    for (inti=0; i< x.length-1; i++) {
        for (int j=i+1; j <x.length; j++) {
            if (x[i] > x[j]) {
                int save=x[i];
                x[i]=x[j]; x[j]=save
} } } }
```

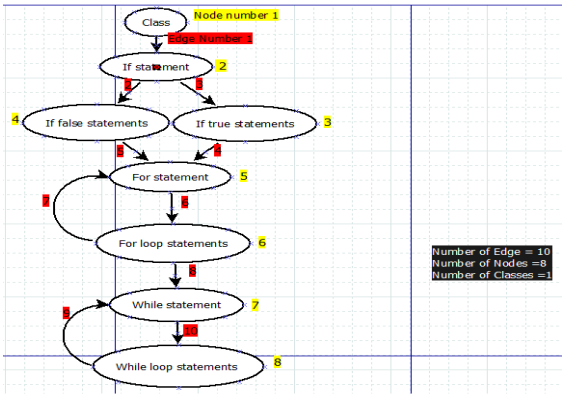

Fig. 1. Main steps for McCabe complexity.

TABLE II. MCCABE EXAMPLE

Measure	Symbol	Result
number of edges	E	10
number of nodes	N	8
number of connected components	P	1
C=E-N+2P C=10-8+(2*1) = 4 4 mean a simple module without much risk		

TABLE III. MCCABE EXAMPLE

Measure	Symbol	Result
number of edges	E	13
number of nodes	N	11
number of connected components	P	1
C=E-N+2P C=13-11+(2*1) = 4 4 mean a simple module without much risk		

The following is an example about the ratio of nested condition statements as shown in Table 4.

TABLE IV. NESTED CONDITION EXAMPLE

Over all condition statements	4
Nested condition statements	2
Ratio = Nested condition statements / Overall condition statements Ratio = 2/4	

B. Halstead Complexity

This theory is used to analyze and measure the complexity of the code; it relies on code division into two parts: Operators and Operands. In this way, the theory of Halstead that he believes can be interpreted as the followings: the program is a collection of operations performed on data, so in this case, each code in the program is either operation or operand. The following notations are used:

By using these parameters, Halsted theory can be defined as a set of complexity measures, including the program volume, program difficulty, program development time, and program bug fixing effort. Table 5 shows the symbol equation

n1= number of unique or distinct operators appearing in a program.
n2= number of unique or distinct operands.
n= n1+n2, this is the vocabulary.
N1= total number of operators (implementation).
N2= total number of operands (implementation).
N= N1+N2

for Halsted measure. Tables 6 and 7 show the operators and operand example, respectively.

The following is an example for Halsted complexity.

```
public static void sort(int x []) {
    for (inti=0; i< x.length-1; i++) {
        for (int j=i+1; j <x.length; j++) {
            if (x[i] > x[j]) {
                int save=x[i];
                x[i]=x[j]; x[j]=save
} } } }
```

TABLE V. SYMBOL EQUATION FOR HALSTED MEASURE

Measure	Symbol	Formula
Program length	N	N= N1 + N2
Program vocabulary	N	n= n1 + n2
Volume	V	V= N * (LOG n)
Difficulty	D	D= (n1/2) * (N2/n2)
Effort	E	E= D * V

TABLE VI. OPERATOR EXAMPLE

Operator	Number of Occurrences
Public	1
Sort()	1
Int	4
[]	7
{}	4
for {;;}	2
if ()	1
=	5
<	2
n_1 = 17	N_1 = 39

TABLE VII. OPERAND EXAMPLE

Operand	Number of Occurrences
X	9
Length	2
I	7
J	6
Save	2
0	1
1	2
$n_2 = 7$	$N_2 = 29$

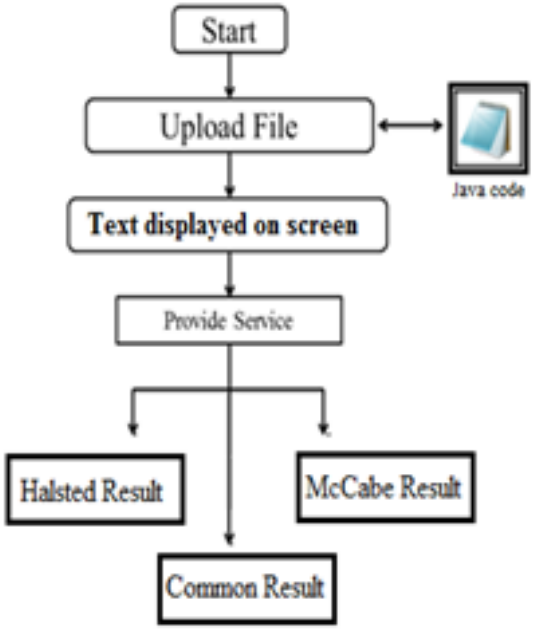

Fig. 2. Flow chart for the complexity analysis of JAVA code system.

III. PROPOSED APPROACH

The goal of this paper is to build a tool that measures the complexity of code to distinguish between the programs which have a little or more complexity, this can be made for the following reasons: to have a high-quality program, easy to be understood by the other programmers, has few mistakes, easy to use, easy to re-use, easy maintenance, easy to test, less of execution time, and lower cost.

Fig. 2 displays a flow chart for the complexity analysis of JAVA code system and working process. This system contains the main process of the first screen which uploads the file that contains Java code and then Enter, when the user start the code is displayed in the report, then the user selects what he/she needs. In this project there are 3 cases: Halsted Result, McCabe Result, and Common Result.

In order to create database for this program, all constants in the program must be selected, these constants such as all Java reserved words, and all Operators used in Java. Table 8 lists all words that are reserved, and Table 9 lists all Operators that are used.

TABLE VIII. JAVA RESERVED WORDS

abstract	Continue	For	new	switch
assert***	Default	goto*	package	synchronized
boolean	Do	If	private	this
Break	Double	implements	protected	throw
Byte	Else	import	public	throws
Case	enum****	Instance of	return	transient
Catch	Extends	int	short	try
Char	Final	interface	static	void
Class	Finally	long	strictfp**	volatile
const*	Float	native	super	while

TABLE IX. JAVA OPERATORS

Category	Operator	Name/Description
Arithmetic	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
	%	Modulus
	++	Increment
	--	Decrement
Logical	&&	Logical “and”
	II	Logical “or”
	!	Logical “not”
Comparison	==	Equal
	!=	Not equal
	<	Less than
	<=	Less than or equal
	>	Greater than
	>=	Greater than or equal
String	+	Concatenation(join two string)

IV. EVALUATION AND DISCUSSION

The main objective of this paper is to build a tool that measures the ratio of the complexity of JAVA programs. Typically, the best way to test the program is to have an example for it, in other words, a copy of the program must be available to have full evaluation for the program. This analysis is a dynamic based technique, where the program has been traced and inspected at running time. An example along with detailed steps about how the proposed approach works are presented and explained in this section.

Fig. 3 shows the Main window of the system which appears after clicking Enter in the Welcome window. It contains many buttons and empty space, these buttons such as: Browse of the project, Browse by Class, Clear Code area, McCabe Result, and Halsted Result. The main objectives of the buttons are as follows:

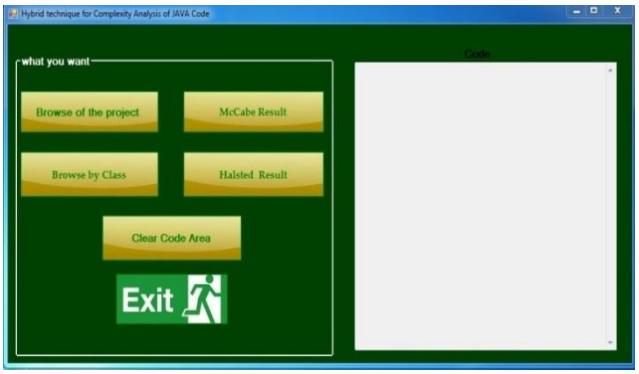

Fig. 3. Main window.

- **Browse of the project**: to open new screen in order to look for a folder containing some of classes written in Java,
- **Browse by Class**: to open new screen to look for any file containing some of codes written in Java.
- **Select code**: If you select a folder from (Browse of the project), this folder contains some of classes (message of number of file founded)
- **Clear Code area**: When the button is pressed, then any code in the code area is deleted.
- **McCabe Result**: the results screen is as shown in Fig. 4. It is designed for the following reasons: 1) Extract all number of edges, number of nodes, and number of connected components, 2) Make the necessary calculations, and 3) Find a level of complexity.

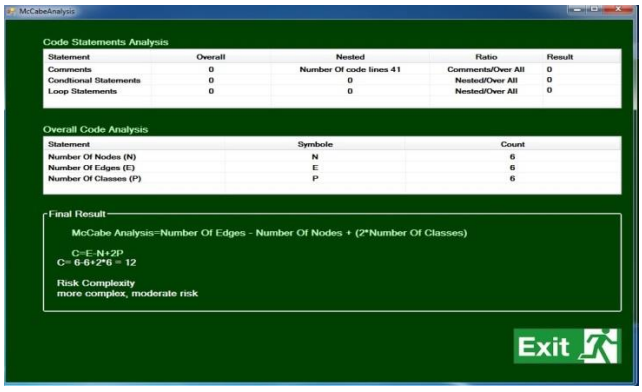

Fig. 4. Results screen.

In this window (Fig. 4), there are three main parts: Code Statement Analysis, Overall Code Analysis, and Final Result.

1) Code Statement Analysis: to extract number of comments, number of conditional statements, and number of loop statements in the project.

2) Overall Code Analysis: to extract number of edges**,** number of nodes, and number of connected components in the project**.**

3) Final Result: to calculate the complexity of the project using C=E-N+2P equation, then find the level of complexity using Table 1.

- **Halsted Result:** is designed to extract all Operators and Operands, make the necessary calculations, and find a level of complexity.

In this window (Fig. 5) there are three main parts: Operators, Operands, and result of Halted equation

1) Operators: to extract Operators with total number of each one.

2) Operands: to extract Operands with total number of each one.

3) Result of Halted equation: to calculate the Complexity of the project.

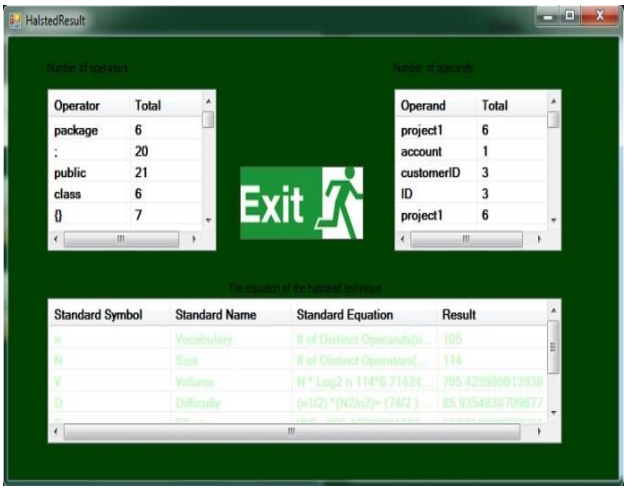

Fig. 5. Halsted result window.

- **Exit:** In all previous windows, click on the exit button to close the window or close the program.

Typically, code complexity correlates with the defect rate and robustness of the application program. In practice, the process of calculating the time complexity of a large program would be unproductive. Therefore, the developers must just focus on understanding the time complexity of the main functions of the program. Since that the time complexity of any program is considered as strong evidence and analysis for the complexity.

As shown in the above result, the output of the tool gives a detailed data about the undertaken code. Thus, by comprehending this data, the developers can know the exact complexity. This complexity can be used by the developers for any update or maintenance over the code specially when performing refactoring [24], [25]. The refactoring process over any source code is considered as a challenge for the developers, where the developers need to know previously the exact complexity information for the code. By presenting this information, the refactoring process will be easier and safer.

Moreover, the presented tools give a very accurate categorization for complexity risk. Furthermore, the presented approach helps the developers to find coding errors and programmers mistake if it exists. The presented approach was also evaluated by 10 master students, the student tried and evaluated the tool over two four open source software which are OpenCms which is website content management, Gwen view which is for 3D Modeling, K-3D which is for image viewer, and OLAT which is for Online Learning and Training. For each system, the students tried ten different test cases that mainly contains nested if statement and loops with all operators. The results show that having an analysis for Java programs using McCabe and Halstead theories together is very helpful for the developer. Moreover, the results can be used efficiently as a guide for software refactoring process, predicting effort, rate of error and time, and in scheduling projects.

V. Related Work

A survey about software testing was presented by [26] which describes and presents the current approaches for software testing; the paper also presents an overview about the used models in software analysis and testing.

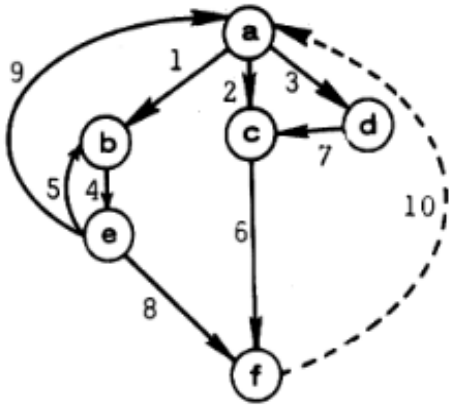

Fig. 6. McCABE example.

In 1976, Thomas McCABE used graph-theory to explain programming complexity [27], this method made it easier to trace the code paths within the program using algebraic expression to solve the infinite backward loops as shown in Fig. 6 as an example for a control graph.

On the other hand, Halstead [28] in 1977 has defined the way that metrics should affect the software implementation or expression despite of the type of the language that the developers have been used, but at the same time it won't affect the platform that has been used on the code execution time. The main idea was to find out a relation between all measurable properties for the software, this will measure the easiness of understanding the software code.

The complexity of coding issues has been raised especially with the appearance of object-oriented programming languages, Java was and still one of the most object-oriented languages that is used especially with the arise of mobile programming, mobile and other Java dependencies like Linux and Unix repositories needs away to find out the reachability issue for a dead repositories code [29] to reduce the time missed in seeking a dead source.

VI. Conclusion

Complexity measures can be used to predict critical information about testability, reliability and maintainability of the software systems from automatic analysis of the source code. There are many code complexity measurements as Lines of Code metrics, McCabe, Halstead Metrics, Maintainability Index, and other code complexity measurements. In this paper, a tool has been developed to analyze the complexity of JAVA code using two complexity measures, Halsted and McCabe. The Halstead and McCabe theories has been explained, and the way which used to analyze code and find the complexity rate. The results show that the presented approach gave very useful and understandable results that can be used for developers assisting.

It has been concluded that this issue is very helpful to distinguish between the program which has a complexity ratio if it is high or not, because if there was less complexity ratio then the program is in its best case, easier to be understood, and easier to re-use and maintenance. Moreover, the focus in the paper has been made on analyzing codes written in Java, however in the future work there is a decision to expand this project to be negotiable on programs written in other languages such as C++, C# and/or any other languages.

In this paper, McCabe and Halstead theories have been only used, there is a hope to extend the program and add other metrics in the future work such as Zage metrics, McClure etc. This program is widely used to help the instructor to check the code, at the university for example, and compare codes written by programmers or students at the university or company. This program can be used by any person using Java to check his work quality and performance. The plan is to make the proposed technique useful for predicting the complexity of the program while designing phase by adding new features and statistical data.

References

[1] Arnold, K., Gosling, J., Holmes, D., & Holmes, D. The Java programming language (Volume 2). Reading: Addison-wesley. 2000.

[2] Sanchez, S. M., & Lucas, T. W. Exploring the world of agent-based simulations: simple models, complex analyses: exploring the world of agent-based simulations: simple models, complex analyses. In Proceedings of the 34th conference on Winter simulation: exploring new frontiers. 2002. Pages 116-126.

[3] Shrivastava, S. V., & Shrivastava, V. Impact of metrics based refactoring on the software quality: A case study. In TENCON 2008-2008 IEEE Region 10 Conference. 2008. Pages 1-6.

[4] Davis, J.S., & LeBlanc, R.J. A Study of the Applicability of Complexity Measures. IEEE Transactions on Software Engineering, Volume 14. Number 9. 1988.

[5] Sheppard, S. B., Curtis, B., Milliman, P., Borst, M. A., & Love, T. First-year results from a research program on human factors in software engineering. In afips (p. 1021). IEEE. 1899

[6] Prabhu, Jeevan. Complexity Analysis of Simulink Models to Improve the Quality of Outsourcing in an Automotive Company. Manipal University. 2010.

[7] Olszewska, Marga. Simulink-Specific Design Quality Metrics. TUCS Technical Report 1002. Turku Centre for Computer Science. 2011.

[8] Tourlakis G. J. Computability, Reston, Virginia. Volume 12. 1984. Pages 39-42.

[9] Olabiyisi S.O, Omidiora E. O and Sotonwa K. A. Comparative Analysis of Software Complexity of Searching Algorithms Using Code Based Metrics. International Journal of Scientific & Engineering Research. Volume 4. Number 6. 2013.

[10] Aiguier, Marc et al. Complex Software Systems: Formalization and Applications. International Journal on Advances in Software. Volume 2. Number 1. 2009. Pages 47–62.

[11] Delange, J., Hudak, J., Nichols, W., McHale, J., Nam, M. Y., (2015) Evaluating and Mitigating the Impact of Complexity in Software Models. CMU/SEI-2015-TR-013, Software Engineering Institute Carnegie Mellon University.

[12] Banker, R. D. et al. A Model to Evaluate Variables Impacting the Productivity of Software Maintenance Projects. Management Science. Volume 37. Number 1. 1991. Pages 1–18.

[13] Jay, Graylin et al. Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. Journal of Software Engineering and Applications. Volume 2. Number 3. 2009. Pages 137–143.

[14] Nolte, Thomas. Hierarchical Scheduling of Complex Embedded Real-Time Systems. Ecole d'Ete Temps-Réel 2009 (ERT09). Paris, France. September 2009.

[15] Nichols, William R. Plan for Success, Model the Cost of Quality. Software Quality Professional. Volume 14. Number 2. 2012. Pages 4–11.

[16] Obradovic, Alex. Using TSP to Develop and Maintain Mission Critical IT Systems. TSP Symposium. 2013.

[17] Banker, Rajiv D. et al. Software Complexity and Maintenance Costs. Communications of the ACM. Volume 36. Number 11. 1993. Pages 81–94.

[18] Shull, Forrest et al. What We Have Learned About Fighting Defects. Proceedings of the 8th IEEE Symposium. 2002. Pages 249–258.

[19] Zage, W. M. & Zage, D. M. Evaluating Design Metrics on Large-Scale Software. IEEE Software. Volume 10. Number 4. July 1993. Pages 75–81.

[20] Nam, Min-Young. ERACES: Complexity Metrics Tool User Guide. 2015.

[21] McCabe T.J. A Complexity Measure. IEEE Transactions on Software Engineering. Volume 2. Number 4. 1976. Pages 308-320.

[22] Halstead M.H. Elements of software science: Published by North Holland Amsterdam and N.Y. 1977.

[23] Harrison, W. A., & Magel, K. I. A complexity measure based on nesting level. ACM Sigplan Notices, Volume 16. Number 3. 1981. Pages 63-74.

[24] Meqdadi, O, Alhindawi, N, Collard, ML, Maletic, JI, "Towards understanding large-scale adaptive changes from version histories" in International Conference onSoftware Maintenance (ICSM), 2013 29th IEEE.

[25] Alhindawi, N, Alsakran, J, Rodan, A, and Faris, H, "A Survey of Concepts Location Enhancement for Program Comprehension and Maintenance" in: Journal of Software Engineering and Applications, 7:5 (2014), pp. 413–421.

[26] Lee, J, Kang. S, and Lee. D "Survey on software testing practices," in *IET Software*, vol. 6, no. 3, pp. 275-282, June 2012.

[27] McCABE, T. J. (n.d.). A Complexity Measure - IEEE Xplore Document. Retrieved August 26, 2017.

[28] Hamer, P. G. (1992). Advertisements. *Environmental Science & Technology*,*26*(12).

[29] Buchsbaum, A, Yih-Farn Chen, Huale Huang, E. Koutsofios, J. Mocenigo, A. Rogers, M. Jankowsky, S. Mancoridis, "Visualizing and analyzing software infrastructures", *Software IEEE*, vol. 18, pp. 62-70, 2001.