UDC 3338.01
DOI: 10.20535/SRIT.2308-8893.2022.2.02

DEVELOPMENT OF A HYBRID METHOD FOR CALCULATION OF SOFTWARE COMPLEXITY

T.H. KAZIMOV, T.A. BAYRAMOVA

Abstract. The use of code metrics allows software developers and project managers to evaluate various features of the software (to be built or already in existence), predict workload, determine software complexity and reliability, and quantify the quality of software systems being developed. Articles written in recent years have proposed various methods for solving this problem. However, there is still no very effective approach to measuring software complexity. This article provides a brief overview of existing software complexity metrics and proposes a new hybrid method for computing software complexity. The proposed hybrid method for evaluating software complexity combines the key features of the Halsted, Maccabe, and SLOC metrics and also allows for a more efficient assessment of complexity.

Keywords: software engineering, software complexity, complexity metrics, hybrid method.

INTRODUCTION

One of the most dynamically developing spheres of modern life is information technology. Today, people's lives are organized in such a way that even people who are far from information technology use them to achieve their goals. The introduction of automated information systems not only reduces the number of operations performed by a person, but also creates new problems. As software grows in size and complexity, the number of bugs increases. Software projects that at first glance appear to be successful may be stopped due to such errors, or the code may be rewritten. This leads to a slight increase in budget expenditures. The world's leading software companies are working on these problems. Various standards organizations (ISO/IEC, IEEE) have developed hundreds of standards covering all stages of software life to improve its quality. Software quality is a complex and multifaceted concept. After the advent of computers in the defense industry in the 1970s, metrics began to emerge to evaluate software performance and quality management.

Software measurement is one of the most important issues in developing quality software. In the words of Tom DeMarco [1], "you can't control what you can't measure".

At the moment, a large number of different software metrics have been developed. Software metrics fall into three categories (Fig.1):

- ***Product metrics*** are product characteristics such as performance, design, size, quality level, and complexity.
- ***Process Metrics***. These metrics are used to improve software development and maintenance processes.
- ***Project Metrics***. These metrics describe the performance and characteristics of the project.

Among these indicators, software complexity indicators are of particular importance. In the 10th edition of Sommerville's book Software Engineering, he noted that one of the most important tasks in the development of the modern software industry is the management of software complexity [4]. Research shows that when the project size approaches 5 million lines, the number of defects begins to increase dramatically (Fig. 2). This can be explained by the significant complication of large-scale projects [5].

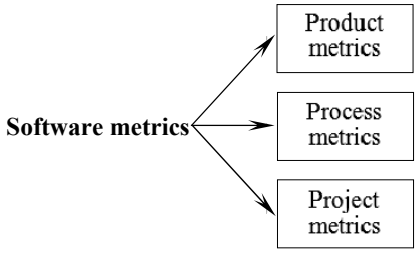

Fig. 1. Software metrics

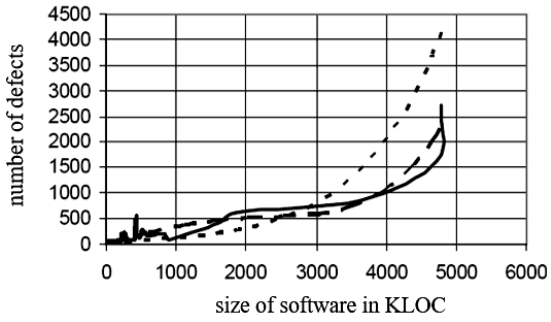

Fig. 2. Dependence of KLOC

Complexity is the degree where it is difficult to verify and understand the design or application of a system or component. The problem of software complexity estimation is widely studied in the field of software engineering. The complexity of the software may vary depending on the choice of algorithm, design, choice of programming language, writing code. Research shows that increasing complexity increases the number of bugs in software, which makes it difficult (in some cases impossible) to maintain and improve programs and makes it difficult to test certain modules. The legibility of the program code directly depends on the level of complexity of the program. The relationship between indicators of software complexity and various attributes of a software system is shown in Fig. 3.

The importance of program complexity indicators can be illustrated as follows [7]:

✓ Difficulty metrics can help people predict and sustain projects.

✓ Complexity metrics can help estimate the amount of programming and development costs, and estimate maintenance costs.

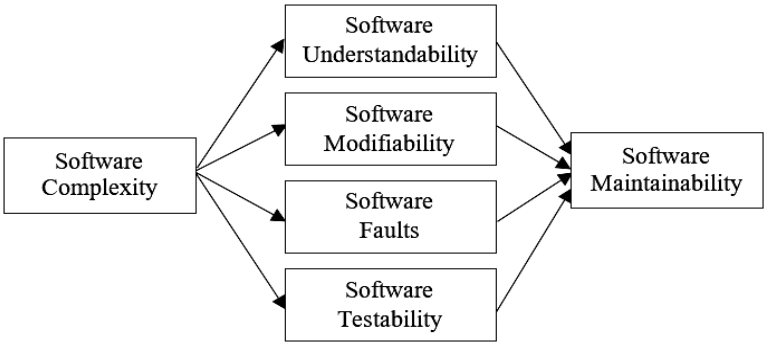

Fig. 3. Software complexity and various attributes of a software

✓ Difficulty scores can help you choose the most suitable program with the same functionality. The less complex the programs, the better they are.

✓ Complexity measures can be used to predict defects or errors [8].

✓ By determining the complexity of software systems, the overall workload and personal contribution of each person working on software modules to the overall work can be assessed in evaluating the performance of software developers [9].

RELEATED WORK

Application of program code metrics allows professionals who work on the project to evaluate various feauters of existing or to be created software, to predict the scope of work, quantitatively characterize these or other project solutions, to evaluate quality of prepared systems, complexity and realiability of software [9–13].

The large amount of data and functionality required in today's enterprise systems presents many challenges for software developers. It is difficult to maintain a balance between software complexity and ease of use, which are indicators of the quality of software, as the complexity of the software increases and its use becomes more difficult. To ensure the quality of the software and the high level of project management, it is necessary to control the complexity and other related characteristics. Large companies such as SAP and Oracle are currently losing market share due to the cost and complexity of the software product [14].

Numerous studies show that complexity leads to an increase in the number of vulnerabilities in software. Programmers face a number of challenges when trying to make changes to complex software components. Software developers should strive for minimal complexity, as increasing complexity creates security risks that not only damage the business, but also damage its reputation. Violation of safety measures can also pose a serious threat to human health. Some experts deliberately complicate the same program so that they do not write it twice. It is necessary to simplify the management of software security. The authors substantiate the relationship between security and software complexity (Fig. 4) [15].

Different software development organizations use different metrics to measure and maintain the quality of software code. In [16, 17], the authors

conducted comparative analytical studies using Halstead metrics and proved that software complexity metrics can be used to measure various characteristics. The authors analyzed the program code written for the same function in the programming languages Python, C, JavaScript, and Java. The complexity of these program codes was calculated based on the Halstead scores. Experience has shown that python is more convenient and simpler, while Java is a modern, powerful, but complex programming language. They showed that program performance is important throughout the life of a project, and they concluded that early calculations have a direct impact on fixing errors and thus saving time and budget.

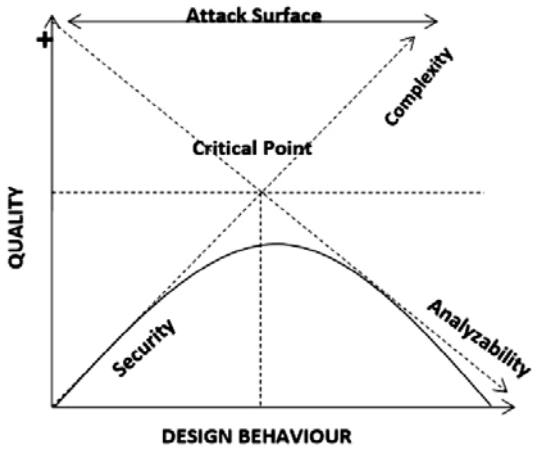

Fig. 4. Security and software complexity

In [18], the authors point out that the main reason for software vulnerabilities is its complexity. Failure to discover software vulnerabilities in a timely manner can jeopardize confidentiality, completeness, and availability. Studies of 12 programs, differing in characteristics and sizes, have shown that the complexity of all software components (size, structure, etc.) is important for predicting vulnerabilities.

In their articles, Shin and William examined code complexity metrics that can be used to predict software vulnerabilities, and concluded that the more complex the software code, the more vulnerabilities. Their results showed a correlation between difficulty scores and vulnerabilities in the Mozilla JavaScript Engine [19].

In [20], the authors investigated the influence of complexity, coupling and cohesion on the number of vulnerabilities in a program. Experiments with Mozilla Firefox have empirically proven that there is a relationship between these parameters and vulnerabilities in the program. Vulnerabilities cannot be discovered in the early stages of software development. Metrics such as complexity, coupling and cohesion can be calculated early. Calculating these metrics can help software developers identify potential vulnerabilities.

In [21], the authors presented a new approach to measuring the quality of software using fuzzy metrics obtained as a result of software design. This metric begins with an assessment of the difficulty rating for each class, which is itself assessed based on the difficulty rating of the class attributes and the difficulty rating of the class methods. Then, to assess the quality metrics of the fuzzy code,

they followed a pattern to explain the relationship between class complexity and LOC for that class, and between method complexity and LOC for that method.

The paper [23] examines the problem of software complexity and its impact on software errors. Having studied various studies, the authors came to the conclusion that with the increase in the complexity of the software system, the threats to its security also increase. While most studies have shown that bugs in software systems are related to their complexity, some researchers have noted that the relationship between complexity and the occurrence of bugs is weak [24].

The software development process, including documentation, design, software, testing, and support, can be measured statistically. Thus, the quality of the software can be effectively controlled. Software metrics are very important in software engineering research [25] provides a summary of software metrics and their types. The authors note that measuring the complexity of software is an integral part of software performance and affects the price and reliability of software products.

Software users assume there are no bugs in the program, but software developers know that it is very difficult, and in some cases impossible, to write program code without bugs. This complexity is mainly due to the intrinsic complexity of the program and the problems that arise when developing and testing the program. In [26], the authors demonstrated the relationship between the complexity of software and its reliability in each specific case.

In [27], the authors analyzed aspects influencing the complexity of the program based on various metrics. A large amount of program code to some extent affects the programmer's thought process and leads to more errors in the program. At the same time, if the number of modules and branches in the program is large, the volume of information exchange is large, and the program is very complex, this will cause problems for the program testers and reduce the quality of testing.

Software complexity plays an important role in reducing the effort required to build and maintain software, and in improving testing efficiency and software quality. The more complex the solution of the program, the more errors it creates [28] examines four software metrics, their importance, strengths and weaknesses. The authors come to the conclusion that each method covers a part and takes into account a certain group of parameters. Therefore, it is important to use a combination of these metrics to measure software complexity.

HYBRID SOFTWARE COMPLEXITY CALCULATION METHOD

The hybrid software complexity method presented in this study combines the advantages of the SLOC, Halstead, Maccab metrics and the module connectivity metric.

Number of lines of code (LOC). When assessing the complexity of a software product, three groups of indicators are usually used: indicators that determine the size of the program, indicators of the complexity of the program flow, and complexity of the data flow of the program. The first group is more common because the metrics are simple. The program size traditionally means the number of source lines of the program (SLOC – Source Lines Of Code) [29].

Lines of Code (LOC) or Source Lines of Code (SLOC) are used to measure the size of a program by counting the number of lines in the text of the program's source code. LOC measures the amount of code, can be used to compare or evaluate programs that use the same programming language and are coded using the same coding standards. Lines of code are completely dependent on the programming language and may differ during the conversion of the program code to any other programming language. This indicator cannot be considered effective in assessing complexity.

Halsted metric. M.H. Halsted was the first researcher to write a mathematical formulation of software metrics [30]. Table 1 shows the main Halsted metrics.

T a b l e 1. Halsted metrics

Program dictionary	$h = h_1 + h_2$
Program length	$N = N_1 + N_2$
Program scope	$V = N \log_2 h$
Program complexity	$D = (h_1/2)(N_2/h_2)$
Implementation effort or program comprehensibility	$E = DV$
Expected number of errors in the program	$B = E0{,}667/3000$

The Halsted metric is based on four measurable program characteristics:

h_1 — the number of unique program operators;

h_2 — the number of unique software operands;

N_1 — total number of program operators;

N_2 — total number of program operands.

McCabes cyclomatic complexity. McCabe proposed this metric in 1976. McCabe's cyclomatic complexity metric is one of the most common indicators for evaluating software complexity. This indicator is calculated based on the control-flow graph of the program and is an indicator of the complexity of the control structures.

The formula for calculating cyclomatic complexity is as follows:

$$C = e - n + 2p,$$

where e — the number of edges of the graph; n — the number of nodes of the graph; p — the number of connected components.

This method takes into account all kinds of cyclic and conditional operators, as well as the complexity of their logical predicates. Linear operators are ignored here. Despite the passage of 45 years, it remains relevant because it more accurately expresses the complexity of the fundamental structures of software. The author of the metric notes the high correlation of the metric with errors. Therefore, the use of this metric to evaluate errors is reasonable and logical. Cyclomatic complexity allows not only to estimate the labor costs and the cost of software projects, but also to make the necessary management decisions taking into account risks [32, 33].

The program module coupling metric distinguishes between data communication, data structure, control, common area (global data), content, as well as external communication, message communication, subclass communication, time communication, and no communication. The advantages of these metrics are that they were developed back in the 80s, are well studied, are often used, and the calculation of values for metrics is automated. These metrics are intended for structured programming, but can be applied to object-oriented programming.

A hybrid method for calculating the complexity of a program code. The proposed method provides an integrated approach to the complexity of the software system and takes into account the following disavantages of the McCabe and Halsted metrics and LOC:

- it cannot be argued that a program with the maximum number of lines of code is more complex, sometimes a small program can be quite complex;
- linear, cyclic, conditional operators in the system of Halsted metrics have the same complexity, which is not true;
- all cyclic, conditional operators in the McCabe metrics system have the same complexity, which is not true. It is assumed that the more nested cyclic and conditional (multiple branching) statements, the more complex the program code becomes and the greater the likelihood that the programmer can make a logical error;
- models do not take into account the inter-module structural complexity of the software, which generates a significant number of defects.

In this method, the main parameters affecting the complexity are the algorithmic complexity of each program module, the number of standard called programs ready for use, and the total number of intermodule links.

When solving problems algorithmically, it often becomes necessary to create a cycle containing another cycle in its body. Such loops are called nested loops. Sometimes you have to check several conditions in a row. To solve the problem of redundancy, you can nest conditional statements inside each other. This is called multiple branching of conditional statements. Nested loops and multiple branches in the code complicate it. At the same time, the number of interacting modules and standard software applications available in these modules directly affects the complexity of the program.

The complexity of the program code can be calculated using the formula (higher value is considered better):

$$C = \log_2 E + \log_2 (N - n) + r,$$

where N — the number of all operators in the program code; n — total number of conditional and cyclic operators; r — the number of intermodular couplings; E — parameter depending on the number of cyclic and conditional operators in the program code. This parameter is calculated using the following formula:

$$E = \sum_{i=0}^{n} (i+1) m_i, \quad (i = 0,1,...,n),$$

m_i is the total number of conditional and cyclic operators in which conditional and cyclic statements of number i are nested. (For example, if there are 2 cyclic statements in the program code and three nested cyclic statements in each of them, then $i = 3$, $m_3 = 2$).

Let's calculate the complexity of two programs written in C++ (Fig. 5 and 6). First, we received an expert assessment of the complexity of these programs based on the assessment of 3 experts on a 10-point scale. All experts rated the algorithm in Fig. 6 as complex, since this algorithm has more nested cyclic and conditional operators. The calculation results are shown in Table 2.

```
 1  #include<iostream>
 2  using namespace std;
 3   int main {}
 4  {
 5   int n=3;
 6   int a[n][n];
 7   int m=0
 8  for {int i=1; i<=n;i++}
 9   for {int j=1; j<=n;j++}{
10         cin>>a[i][j];
11        if (a[i][j]==i+j)
12        m=m+1;
13       else
14        m=m;
15  }
16
17  if(m>0)
18   for {int i=1; i<=n;i++}
19    for {int j=1; j<=n;j++}{
20
21       if (a[i][j]==i+j)
22         cout<<a[i][j];
23       else
24         continue;
25  }
26  else
27   cout<<"No such element";
28  return 0;
29  }
```

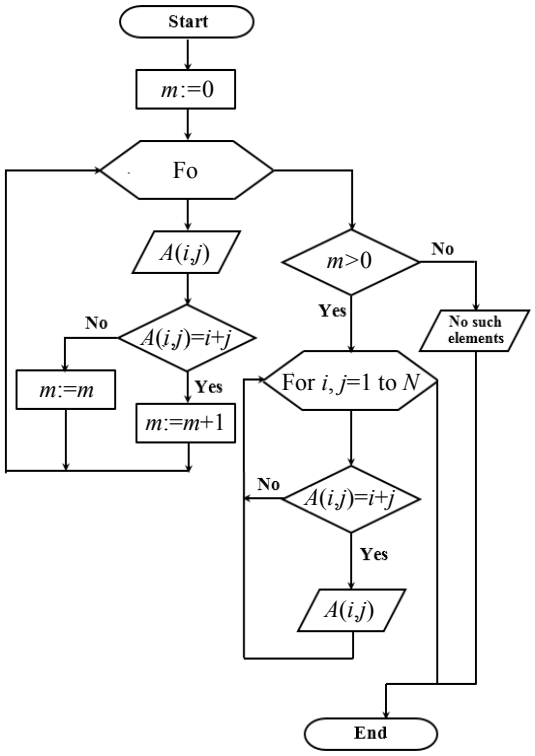

Fig. 5. Finding complexity of program 1

Table 2. Software complexity metrics

Complexity metrics	Calculation	Program 1		Program 2	
LOC	Number of lines of code		27		30
Expert assessment			3		4
MacCabe	For a single program $C = e - n + 2$	e=15; n=12	6	e=17, n=13	6

Continued Tabl. 2

Complexity metrics	Calculation	Program 1		Program 2	
Hybrid	$E=\sum_{i=0}^{n}(i+1)m_i$ $C=\log_2 E+\log_2(N-n)+r$	N=108, \ n=5, r=0, m_0=3 (i=0) m_1=2 (i=1),	9,49	N=101, n=5, r=0, m_0=3 (i=0), m_1=0 (i=1), m_2=1 (i=2), m_3=1 (i=3)	9,58

```
 1  #include<iostream>
 2  using namespace std;
 3  int main {}
 4  {
 5    int n;
 6    cin>>n;
 7    int z[n];
 8    for (int i=1; i<=n;i++){
 9      cin>>z[i];
10    }
11  if (n>1)
12    for (int i=1; i<=n-1;i++){
13    int v=z[i];
14    int I=i;
15    for (int j=i+1;<=n;j++)
16    {
17      if(z[j]<v)
18      {
19        v=z[j];
20        I=j
21      {
22    {;
23    if (I==1)
24    continue;
25    else{
26    z[I]=z[i]
27    z[i]=v;
28    }}
29    return 0;
30    }
```

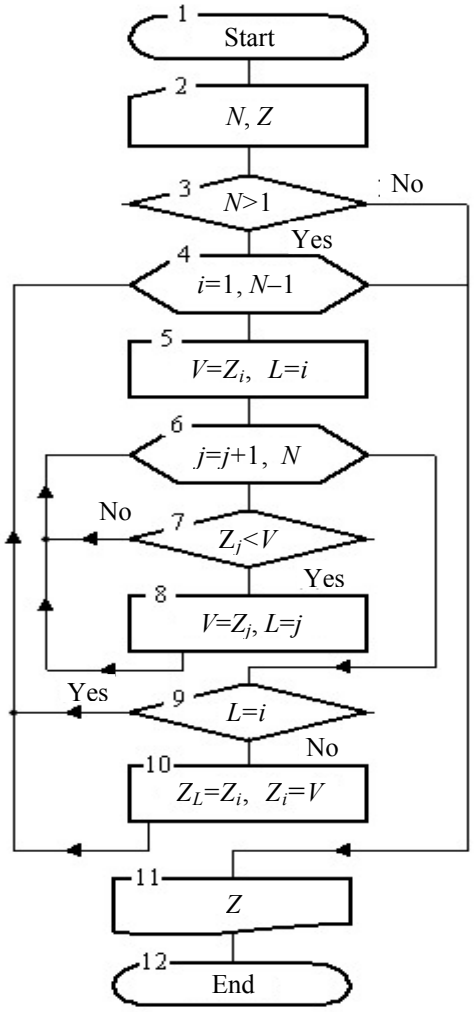

Fig. 6. Finding complexity of program 2

As can be seen from Table 2 according to McCabe, the complexity of programs is the same, because in this method all conditional and cyclic operators have the same level of complexity. The hybrid method takes into account the nesting of cycles, so the result is closer to the expert assessment.

In another example, let's calculate the complexity of a program code written for the same function in C++ and Java using the metrics mentioned above (Fig. 7 and 8). Depending on the chosen programming language, the complexity may increase. Calculating the complexity will help you choose the simpler of the two programs. The calculation results are shown in Table 3.

```
 1  Public Class
 2  {
 3
 4  Public static void main
                (String[]args)
 5
 6  {
 7    int arr[10]={1,9,0,5,6,7,
                  8,2,4,3};
 8    int length = 10;
 9    int result[]=new int (10);
10   result=QuickSort (arr,0,
                  length-1);
11  }
12
13  Static int [] QuickSort (int
                  []a,int r)
14
15    if (I<r)
16
17  {
18    int i=I;
19    int j=r;
20    int k = (int)((I+r)/2);
21    int pilot = a[k];
22
23    do
24    {
25       while (a[i].less (pilot))
26         i++
27       while (pilot.less (a[i]))
28         j--;
29
30       if (i<=j)
31       {
32          int t = a[i];
33          a[i] = a[j];
34          a[j] = t;
35
36          i++;
37          j++;
38
39       }
40    }
41    while (i<j)
42
43  a=QuickSort{a,i,j};
44  a=QuickSort{a,i,r};
45  }
46
47  return a;
48
49  } //end of QuickSort
50
51
52  } // end of class QS
```

Fig. 7. Quick sort implementation code in C++

```
 1  #include<iostream>
 2  using namespace std;
 3
 4  int main()
 5
 6  {
 7
 8    int A[]={}1,9,0,5,6,7,8,2,3,4
 9    intlength = 10;
10   quickSort(A,0,length-1);
11
12
13  void quicksort(int A[],int
                  F,int L)
14  {
15    int pivotIndex;
16
17    if (F<L)
18
19    Partition(A,F,L,pivotindex;
20    quicksort(A,F,pivotindex-1);
21    quicksort(A,pivotindex+1,L);
22    }
23  }
24
25  void partition(int A[],int F,
          int L,int & pivotindex)
26  {
27    int piv = A[F];
28    int lastS1 = F;
29    int firstUnknown = F+1;
30
31    for(firstUnknown = F+1;++
              firstUnknown)
32    {c
33    if (A[firstUnknown]<pivot)
34      {++lastS1;
35        Swap(A[firstUnknown],
              A[lastS1]);
36    }
37    }
38    Swap(A[F],A[lastS1]);
39    pivotindex=lastS1;
40
41  }
42
43  void Swap(int & x,int & y)
44  {
45    int temp = x;
46    X=y;
47    Y-temp;
48  }
```

Fig. 8. Quick sort implementation code in Java

Table 3. Software complexity of C++ and Java programs

Complexity metrics	Calculation	C++		Java	
LOC	Number of lines of code		48		52
MacCabe	For a single program $C = e - n + 2$	e=28, n=26	4	e=29, n=25	6
Halstead	$D = (h_1/2) * (N_2/h_2)$	N_1=142 N_2=57 h_1=25 h_2=21	33,92		39,75
Hybrid	$E = \sum_{i=0}^{n} (i+1)m_i$ $C = \log_2 E + \log_2 (N-n) + r$	N=142, n=3, r=0, m_1=1 (i=1), m_0=2 (i=0)	9,11	N=133, n=5, r=0, m_0=3 (i=0), m_1=1 (i=1), m_2=1 (i=2)	10

CONCLUSİON

Software complexity metrics are one of the key aspects of software process management. Complexity makes software difficult to understand, which creates problems when maintaining a software system and adding new functions to it. In the articles of recent years, various methods of solving this problem have been proposed. Numerous studies in the field of software complexity metrics suggest that there is no universal metric for assessing the complexity of any program code. The use of any metric, hybrid metric, or multiple metrics depends on the specific problem. This article proposes a new hybrid method for calculating the complexity of software, the effectiveness of which is substantiated by experiments. The proposed metric makes it possible to clarify (due to additional indicators) the complexity of software modules of a large class separately from the known metrics. Based on the above, it is concluded that software metrics are important at the stages of the project life cycle and, with early application of program metrics, help to largely overcome the presence of errors and, thus, save time and money.

REFERENCES

1. T. DeMarco, *Controlling Software Projects*. Yourdon Press, New York, 1982, 816 p.
2. S. Reddivari and J. Raman, "Software Quality Prediction: An Investigation Based on Machine Learning", *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pp. 115–122. doi: 10.1109/IRI.2019.00030.
3. A.V. Smirnov, "Methods for assessing and managing the quality of software", *Izvestia ETU "LETI"* no. 2, pp. 20–25, 2019.
4. I. Somerville, *Software engineering*, 10th edition. Pearson, 2015, 816 p.
5. S.A. Yaremchuk, "Method for estimating the number of software defects using complexity metrics", *Radioelectronic and computer systems*, no. 5, pp. 212–218, 2012.
6. P.A. Laplante, *Dictionary of computer science, engineering and technology*. CRC Press, 2017, 560 p.
7. J. Rashid, T. Mahmood, M.W. Nisar, "A Study on Software Metrics and its Impact on Software Quality", *Technical Journal, University of Engineering and Technology (UET), Taxila, Pakistan*, vol. 24, no. 1, pp. 1–14, 2019.

8. T. Honglei, S. Wei and Z. Yanan, "The Research on Software Metrics and Software Complexity Metrics", *2009 International Forum on Computer Science-Technology and Applications*, pp. 131–136, 2009. doi: 10.1109/IFCSTA.2009.39.
9. T.H. Kazimov and T.A. Bayramova, "Evaluating Key Performance Indicators for Software Development", *IV International Congress on New Trends in Science, Engineering and Tehcnology*, pp. 99–105, 2020.
10. Shweta, S. Sharma, and R. Singh, "Analysis of correlation between software complexity metrics", *International Journal of Innovative Science, Engineering & Technology*, vol. 2 issue 8, August 2015.
11. S. McIntosh et al., "An empirical study of the impact of modern code review practices on software quality", *Empirical Software Engineering*, 21, pp. 2146–2189, 2016. Available: https://doi.org/10.1007/s10664-015-9381-9.
12. S. Bhatia and J. Malhotra, "A survey on impact of lines of code on software complexity", *2014 International Conference on Advances in Engineering & Technology Research (ICAETR – 2014)*, pp. 1–4, 2014. doi: 10.1109/ICAETR.2014.7012875.
13. A. Ghazarian, "A Theory of Software Complexity", *2015 IEEE/ACM 4th SEMAT Workshop on a General Theory of Software Engineering*, pp. 29–32, 2015. doi: 10.1109/GTSE.2015.11.
14. M. Khan, F. Ahmad, and M.A. Khanum, "Literature review on software complexity, software usability and software deliverability", *International Journal of Advanced Research in Computer Science*, vol. 9, no. 2, pp. 438–441, 2018. doi: 10.26483/ijarcs.v9i2.5853.
15. M. Alenezi and M. Zarour, "On the relationship between software complexity and security", *International Journal of Software Engineering & Applications (IJSEA)*, vol. 11, no. 1, pp. 51–60, 2020.
16. S.A. Abdulkareem and A.J. Abboud, "Evaluating Python, C++, JavaScript and Java Programming Languages Based on Software Complexity Calculator (Halstead Metrics)", *2nd International Scientific Conference of Engineering Sciences (ISCES 2020) 16th-17th December, Diyala, Iraq, IOP Conference Series: Materials Science and Engineering*, vol. 1076, pp. 1–9, 2020. doi: 10.1088/1757-899X/1076/1/012046.
17. Nikhil Govil, "Applying Halstead Software Science on Different Programming Languages for Analyzing Software Complexity", *Proceedings of the Fourth International Conference on Trends in Electronics and Informatics (ICOEI 2020)*, pp. 939–943, 2020. doi: 10.1109/ICOEI48184.2020.9142911
18. Y. Javed, M. Alenezi, M. Akour, and A. Alzyod, "Discovering the relationship between software complexity and software vulnerabilities", *Journal of Theoretical and Applied Information Technology*, vol. 96, no. 14, pp. 4690–4698, 2018.
19. Y. Shin, "Exploring Complexity Metrics as Indicators of Software Vulnerability", in *the 3rd International Doctoral Symposium on Empirical Software Engineering, Kaiserslautern, Germany,* October 8, 2008.
20. I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?", *Proceedings of the 2010 ACM Symposium on Applied Computing – SAC '10*, 2010. doi:10.1145/1774088.1774504.
21. O. Masmali and O. Badreddin, "Towards a Model-based Fuzzy Software Quality Metrics", *8th International Conference on Model-Driven Engineering and Software Development*, pp.139–148, 2020.
22. Tong Yi and Chun Fang, "A Novel Method of Complexity Metric for Object-Oriented Software", *International Journal of Digital Multimedia Broadcasting*, vol. 2018, pp. 1–9, 2018. Available: https://doi.org/10.1155/2018/7624768
23. S. Moshin Reza, M. Mahfujur Rahman, H. Parvez, O. Badreddin, and S. Al Mamun, "Performance Analysis of Machine Learning Approaches in Software Complexity Prediction", *Proceedings of International Conference on Trends in Computational and Cognitive Engineering*. *Advances in Intelligent Systems and Computing*, vol. 1309, pp. 27–39, 2021. doi: 10.1007/978-981-33-4673-4_3.
24. P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models", in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pp. 1–9, 2015.

25. Mohd. Kamran Khan et al., "Literature review on software complexity, software Usability and software deliverability", *International Journal of Advanced Research in Computer Science*, 9 (2), pp. 438–441, 2018.
26. M. Devon Simmonds, "Complexity and the Engineering of Bug-Free Software", *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS), Athens*, pp. 94–100, 2018.
27. T. Hariprasad, G. Vidhyagaran, K. Seenu, and C.Thirumalai, "Software complexity analysis using halstead metrics", *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, *Tirunelveli*, pp. 1109–1113, 2017. doi: 10.1109/ICOEI.2017.8300883.
28. A. Athar Khan, M. Amjad, Sajeda M. Amralla, and Tahera H. Mirza, "Comparison of Software Complexity Metrics", *International Journal of Computing and Network Technology*, no. 1, pp.19–26, 2016.
29. G.R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault prediction", *Computers & Electrical Engineering*, vol. 67, pp. 15–24, 2018. Available: https://doi.org/10.1016/ j.compeleceng. 2018.02.043.
30. N. Govil, "Applying Halstead Software Science on Different Programming Languages for Analyzing Software Complexity", *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, pp. 939–943, 2020. doi: 10.1109/ICOEI48184.2020.9142911.
31. L. Pudovkina and V. Sinyaiev, "Applying empirical models and halstead metrics to evaluate the quality of application software", *Open Information and Computer Integrated Technologies*, no. 86, pp.190–197, 2019. doi: 10.32620/oikit.2019.86.14
32. T.J. McCabe, "A Complexity Measure", in *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. doi: 10.1109/TSE.1976.233837.
33. H. Liu, X. Gong, L. Liao and B. Li, "Evaluate How Cyclomatic Complexity Changes in the Context of Software Evolution", *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pp. 756–761, 2018. doi: 10.1109/COMPSAC.2018.10332.

Received 22.09.2021

INFORMATION ON THE ARTICLE

Tofig Kazimov (Hasanaga), ORCID: 0000-0001-9245-6731, Institute of Information Technology of Azerbaijan National Academy of Sciences, Azerbaijan, e-mail: tofig@mail.ru

Tamilla Bayramova (Adil), ORCID: 0000-0002-8377-3572, Institute of Information Technology of Azerbaijan National Academy of Sciences, Azerbaijan, e-mail: toma_b66@mail.ru

РОЗРОБЛЕННЯ ГІБРИДНОГО МЕТОДУ ОБЧИСЛЕННЯ СКЛАДНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ / Т.Г. Казімов, Т.А. Байрамова

Анотація. Застосування показників програмного коду дозволяє розробникам програмного забезпечення і керівникам проектів оцінювати різні функції програмного забезпечення, яке буде створено або вже існує, прогнозувати робоче навантаження, визначати складність і надійність програмного забезпечення і надавати кількісні характеристики якості розроблюваних програмних систем. У працях останніх років пропонуються різні методи вирішення цієї проблеми. Утім досі немає ефективного підходу до вимірювання складності програмного забезпечення. Подано стислий огляд існуючих метрик складності програмного забезпечення. Запропоновано новий гібридний метод обчислення складності програмного забезпечення, який об'єднує ключові характеристики метрик Холстеда, Маккабі і SLOC, а також дозволяє більш ефективно оцінювати складність.

Ключові слова: програмна інженерія, складність програмного забезпечення, метрики складності, гібридний метод.