Tuijin Jishu/Journal of Propulsion Technology
ISSN: 1001-4055
Vol. 44 No 4 (2023)

Code Complexity Calculator Based on Improved Cognitive Based (ICB) Complexity Metrics

H. J. Sudusinghe[1], B. A. P. De Silva[2], I. U. Abeysinghe[3], G. H. G. T. S. De Silva[4], D. I. De Silva[5], M. V. N. Godapitiya[6]

Faculty of Computing, Sri Lanka Institute of Information Technology, Malabe, Sri Lanka.

Abstract: -This research paper introduces an innovative approach to assessing code complexity in object-oriented programming, focusing on enhancing the Improved CB (ICB) metric. Code complexity significantly impacts software quality and maintainability. The study presents an advanced complexity measurement tool that refines the ICB metric by considering factors like inheritance depth, control structure types, nesting levels, statement size, compound conditionals, threads, and recursion. These refinements improve the accuracy of code complexity assessment, aiding informed decision-making in software development and maintenance. The paper reviews related cognitive-based complexity metrics, emphasizing their contributions to code complexity evaluation. It discusses the importance of code complexity assessment in software engineering. The proposed system, a real-time Java code complexity calculation desktop application, offers insightful metrics, including code size and ICB values, presented through charts for better code comprehension. It underscores the necessity of entering a code for meaningful results. In conclusion, the Code Analyzer Tool provides a systematic, data-driven approach to code quality assessment, facilitating code optimization, and clear communication. It promises to be an indispensable asset for software development teams dedicated to delivering maintainable and high-quality software solutions.

Keywords*: ICB Measure, Code Complexity, Complexity Metrics.*

1. Introduction

In the field of software engineering, assessing the complexity of code stands as a crucial aim, directly impacting software quality and maintainability. The present study explores this essential aspect, particularly focusing on code complexity measurement within the context of object-oriented programming. The complexity of software code is a well-recognized concern in the software engineering domain. As modern software systems become more complex, it becomes compulsory to develop powerful metrics that effectively capture the various nature of code. Object- oriented metrics have emerged as an influential approach in this pursuit, offering a comprehensive view of code complexity by considering factors such as control structures, nesting levels, and inheritance relationships. The ICB measure is one such metric that has gained prominence due to its ability to encapsulate the complexities inherent in object-oriented code.

The significance of this research lies in its potential to enhance software quality, maintainability, and overall system performance. By refining the ICB measure using advanced tool functions, the accuracy of code complexity assessment can be elevated, leading to better decision-making in software development and maintenance. The significance of this research lies in its potential to enhance software quality, maintainability, and overall system performance. By refining the ICB measure using advanced tool functions, the accuracy of code complexity assessment can be elevated, leading to better decision-making in software development and maintenance. This study seeks to introduce a novel complexity measuring tool based on the ICB metric These functions of the proposed system address specific complexities associated with token analysis, inheritance

depth, type and nesting level of control structures, threads, compound conditions, and recursion, to provide a precise of code complexity assessment.

The following sections of this paper e systematic approach taken to achieve this enhancement and the potential indication for software engineering practices are discussed in Section 3. The functions of the proposed system are discussed in Section 4. Furthermore, the paper concludes by highlighting the broader significance of our findings in the context of code complexity evaluation.

2. Related Works

The Cognitive Functional Size (CFS) metric, developed by Wang and Shao in 2003[1], served as the foundation for the idea of measuring complexity in software development using cognitive-based metrics. This metric was created to assess how difficult it is for people to comprehend a piece of software. Basic software control structures are given importance values by the CFS metric. It depends on the number of keyboard inputs and outputs as well as the combined cognitive significance of these control structures. The field of cognitive informatics uses empirical and verifiable methods, just like other scientific disciplines.

In January 2006, Kushwaha and Misra introduced the Cognitive Information Complexity Measure (CICM) to grasp how complex the information in a program is and how effectively it's coded [2]. It was determined by considering the cognitive weight of internal basic control structures, identifiers, operators, and the number of lines of code.

Sanjay Misra introduced two complexity measures in the same year. The Cognitive Weight Complexity Measure (CWCM) was simple to understand and compute [3] since it only considered the cognitive significance of internal basic control mechanisms. Later that year, he unveiled the Modified Cognitive Complexity Measure (MCCM), which included the total number of operands and operators [4] in addition to the significance of fundamental control structures. A metric known as Class Complexity (CC) was developed by Sanjay Misra in 2007 to evaluate the complexity of an object-oriented (OO) system [5]. Other proposed metrics for object-oriented programming do not consider the internal architecture of the class, subclass, and member methods like the proposed measure does. By considering basic control structures (BCSs), this metric computed the complexity of a method. Next, it totalled up the complexities of all the methods inside a class to calculate the complexity of that class. The overall complexity of the complete OO system was then calculated by adding the total complexity of all the classes.

The Weighted Class Complexity measure (WCC), developed by Sanjay Misra and Akman in 2008 [6], is used to assess the complexity of object-oriented (OO) systems. WCC calculated complexity similarly to the Class Complexity (CC) measure by adding up the complexity of each method within a class to establish class complexity, and then combining these class complexities to determine the overall OO system complexity. However, WCC went further by not only considering the cognitive weight of internal basic control structures (BCSs) but also factored in complexity stemming from global attributes and message calls. Unlike previous metrics, which typically assigned a fixed weight of 2 to method calls between classes, WCC accounted for both the cognitive weight of the method call and the complexity of the called method when evaluating message calls between classes. For calls occurring within the same class, only the method call weight was considered in the complexity calculation.

Gupta and Chhabra proposed cognitive-spatial complexity measures in 2009[7] to evaluate the complexity of software classes and objects. These metrics used a program's spatial and architectural characteristics to determine its complexity. The architectural aspect considered the significance of basic control structures (BCSs), while the spatial aspect considered how far away program parts were in terms of the number of lines of code.

To evaluate the architecture of object-oriented (OO) systems, Sanjay Misra, Akman, and Koyuncu proposed the Cognitive Code Complexity measure (CCC) in 2011. The CCC uses a three-stage methodology similar to their prior metrics, CC and WCC [8]. CCC used cognitive weights for basic control structures (BCSs) to calculate

complexity at the method level while taking the inheritance aspect into account. Notably, when assessing the program's overall complexity in the third step, CCC also took inheritance into account. Two more cognitive-weight-based complexity metrics were introduced that same year. J. K. Chhabra's Code Cognitive Complexity measure (CCC) considered input/output parameters, cognitive weights, and the number of lines of code between module calls and their definitions or uses [9]. Chhabra enhanced Shao and Wang's cognitive weights for BCSs by defining cognitive weights for variable and constant data types, in contrast to earlier metrics that mainly used Shao and Wang's cognitive weights for BCSs.

The Chhillar and Bhasin CB measure, another metric, evaluated program complexity based on four criteria: inheritance, control structure type, control structure nesting level, and statement size, which was determined by the number of operators, operands, functions/methods, and strings contained in each executable statement [10]. Source [11] provides enhancements for the CB measurements, while source [12] shows limitations of the CB statistic. Additionally, a number of research [13], [14], [15]and suggest improvements to this metric.

The Code Comprehending Measure (CCM) was developed by Gurdev, Satinderjit, and Monika in February 2012 to evaluate complexity based on three factors: data volume, structural complexity, and data flow [16]. The unique variables and operators in a basic control structure (BCS) and how frequently they appeared were considered to assess data volume. In terms of structural complexity, BCSs' cognitive weight was important. The data flow factor considered how data was transferred across BCSs using variables. Sanjay Misra and his team unveiled a set of cognitive metrics in June of the same year to assess the complexity of methods, messages, attributes, classes, and code in object-oriented (OO) programming. Later on in the year, Aloysius and Arockiam proposed their cognitive complexity metric [17], which evaluated the complexity brought on by different kinds of coupling between classes, including Data Coupling (DC), Global Data Coupling (GDC), Internal Data Coupling (IDC), Lexical Content Coupling (LCC), and Control Coupling (CC).

The Multi-Paradigm Complexity (MCM) measurement was developed by S. Misra and colleagues in 2013 and incorporates a number of elements from both procedural and object-oriented (OO) programming, including attributes, variables, basic control structures (BCSs), objects, method invocations using objects, cohesion, and inheritance [18]. For the purpose of evaluating program complexity, K. Jakhar and K. Rajnish presented the New Weighted Method Complexity (NWMC) metric in November 2014 [19] It made use of keyboard inputs, outputs, local and formal parameters, as well as cognitive weights given to BCSs. A new weighted complexity metric that considered cognitive weights assigned to flow chart controls, the quantity of operations, the number of variable declarations, external libraries and functions, function arguments, and locally called functions was introduced in 2015 by M. A. Shehab and colleagues [20].

A large number of software complexity measures have been introduced around the world. Selecting the right complexity measure remains a challenge due to each measure's distinct advantages and drawbacks. Researchers continually seek a comprehensive measure that encompasses most software parameters. In addition to proposing complexity metrics there have been several studies by comparing the existing metrics including cognitive based complexity metrics as well. Source[21], source[22] and source[23] have been comparatively studied that metrics.

In addition to proposing complexity metrics, complexity measuring tools have also been suggested based on those measures. These tools help users by automatically assessing the quality of source code and estimating project schedules using a hierarchical metrics model. It provides quality notifications, analyses metric results, offers insights into the code, and suggests ways to address issues. This tool categorizes metrics into Object Oriented, Complexity Oriented, Size Oriented, and Maintainability Oriented, aiming to improve software engineering practices and provide valuable information about source code.

3. ICB Measure

The ICB measure was introduced as an improvement to the CB metric in 2017 by De Silva et al[14]. It computes complexity based on the following factors:

3.1 Inheritance level of classes

Chhillar and Bhasin came up with a way to measure the importance of statements in different classes within a group. They gave a weight of zero to statements in the main class, a weight of one to statements in the first new class, and added one for each new class after that. This helps us understand which statements are most important in a program as we move from the main class to the subclasses.

3.2 Type of control structures in classes

The complexity of a class or program depends on the type of control structure it employs. Consequently, they assigned different weights to these structures: sequential statements received a weight of zero, conditional control structures like if-else and if-else if conditions were assigned a weight of one, iterative control structures such as for, while, and do-while loops received a weight of two, and switch-case statements with 'n' cases were given a weight of 'n'. This approach allowed for the assessment and management of complexity within various parts of a program based on the control structures used.

3.3 Nesting level of control structures

The number of nesting levels in control structures impacts a program's clarity and adds to its complexity. In light of this, Chhillar and Bhasin introduced a system for measuring this complexity. They assigned a weight of zero to sequential statements, a weight of one to statements at the outermost level of nesting, and a weight of two to statements at the next inner level of nesting, and so on. This method offers a way to quantitatively assess a program's complexity based on the depth of its control structure nesting.

3.4 Size of class in terms of token count

Chhillar and Bhasin thought that when a class or program gets bigger, it also gets more complicated. They looked at the size of a class or program as the last thing to consider in their measurement. To figure out how big a statement was, they counted things like the operators, operands, methods/functions, and strings used in that statement.

3.5 Concurrent programs - threads

De Silva devised a method for assessing the significance of threads in concurrent programs. He introduced a way to distinguish the size of a typical statement from one that initiates a thread. In the case of statements involving thread invocation, he consistently increased the total size of each statement by a constant value of two.

3.6 Compound conditional statements

The ICB measure considers compound conditional statements as a factor. To address this, the ICB measure assigns a weight of one for each "&&" or "||" operator in a conditional statement. This approach is based on the idea that including one "&&" or "||" operator signifies an increase of one in the number of conditions checked by a decisional statement. Since the CB measure assigned a weight of one for each decision statement, regardless of the number of conditions it had, the authors decided to also add a weight of one for each "&&" or "||" logical operator to account for this added complexity.

3.7 Compound conditional statements

Initially, the size (S) of each statement within the recursive function is multiplied by the corresponding weight (W) assigned to those statements. Subsequently, the resulting values are combined and incorporated into the program's final ICB value.

An equation to represent the overall Improved Cognitive Based complexity (ICB) calculation for the provided code. The ICB is calculated by summing up the weighted sums of all executable statements in the code.

The equation for ICB can be represented as follows,

$$C_w(P) = \sum_{j=1}^{n} (S_j) * (W_t)_j \quad (1)$$

Where:

- P is the Improved Cognitive Based complexity.
- Σ represents the summation symbol, indicating that it sums up the following expression for each executable statement.
- S represents the Executable Statement Count for each statement.
- Wt represents the total weight assigned to each statement based on factors like inheritance, control structure type, nesting level, and size.

This ICB measure outlines the process of Improved CB (ICB) is an enhanced complexity assessment metric in software engineering that goes beyond traditional code complexity evaluation. ICB considers not only inheritance, control structures, and statement size but also considers additional factors such as compound conditional statements, threads, and recursion. This comprehensive approach provides a more holistic view of software complexity, helping developers identify and address critical areas for improvement in code readability and maintainability.

4. Methodology

The few steps and the process that was followed for the implementation of Code Complexity Calculator are as follows:

To identify Java code in an input area, one can examine it for specific Java language keywords such as "class," "public," "private," "void," "import," and "package." Additionally, Java code typically terminates each statement with a semicolon, employs curly braces to delineate code blocks, and defines methods using parentheses. The presence of these characteristics aids in determining if the input constitutes Java code.

Methods are located by searching for lines beginning with access modifiers (e.g., public, private) followed by a return type or a class/interface name and containing parentheses ().

Classes are identified by lines starting with access modifiers followed by the class keyword and a class name, as well as lines that contain the class keyword followed by a class name, even if they don't start with access modifiers like public or private.

Variable declarations are identified in lines where a variable name is followed by a data type and possibly an assignment operator (=) or initialization.

Threads are identified by searching for classes that extend the Thread class or implement the Runnable interface. Thread object declarations usually involve creating instances of these classes using the new keyword.

Nesting levels are identified by monitoring the opening and closing of control structures like loops (for, while), conditional statements (if, else), and code blocks (curly braces {}). A stack data structure is used to count the depth of nested structures, with the size of the stack representing the current nesting level.

Control structures are identified via keywords such as "if," "else," "for," "while," and "switch." Additionally, the presence of opening and closing curly braces "{}" is checked to define code blocks. These structures control the flow of execution in the code.

Inheritance levels are identified by checking class declarations to see if they extend or implement other classes or interfaces. In Java, the "extends" keyword is used to indicate inheritance from a superclass, and the "implements" keyword is used to specify implemented interfaces. This information can help determine the inheritance hierarchy within the code.

Compound conditional statements are detected by searching for occurrences of logical operators such as "&&" (logical AND) and "||" (logical OR) within if statements or loops. These operators combine multiple conditions, creating compound conditional statements that control program flow based on multiple criteria simultaneously.

Recursive statements are located in functions or methods that call themselves within their own definitions. Recursive statements are those where a function invokes itself either directly or indirectly, creating a loop of function calls until a base case is met to terminate the recursion.

4.1 Tools and technology

WindowBuilder is an Eclipse IDE plugin makes it much more convenient to create complex GUIs for Java desktop applications by providing a visual interface for designing the application's user interface. One can continue to enhance and customize the application by adding logic, event handling, and additional GUI components as needed.

Regex library utilize the regex library in code, import the relevant library. Then, create a `Pattern` object with a desired regular expression pattern and compile it. Finally, use the `Matcher` class to apply the pattern to input data, allowing for finding, matching, or manipulating specific text patterns within the code.

JFreeChart library use the JFreeChart library in code, begin by adding the JFreeChart JAR files to the project's classpath. Next, import the required classes from the library. Then create a chart using the provided factory methods, customize it, and then display it in the GUI application using a ChartPanel or a similar component.

4.2 System functionality illustration

The document references a visual representation of the overall system functionalities as "Fig. 1" This figure likely provides a visual overview of how these functionalities are integrated into the system architecture, but the actual details of this illustration are not provided in the text.

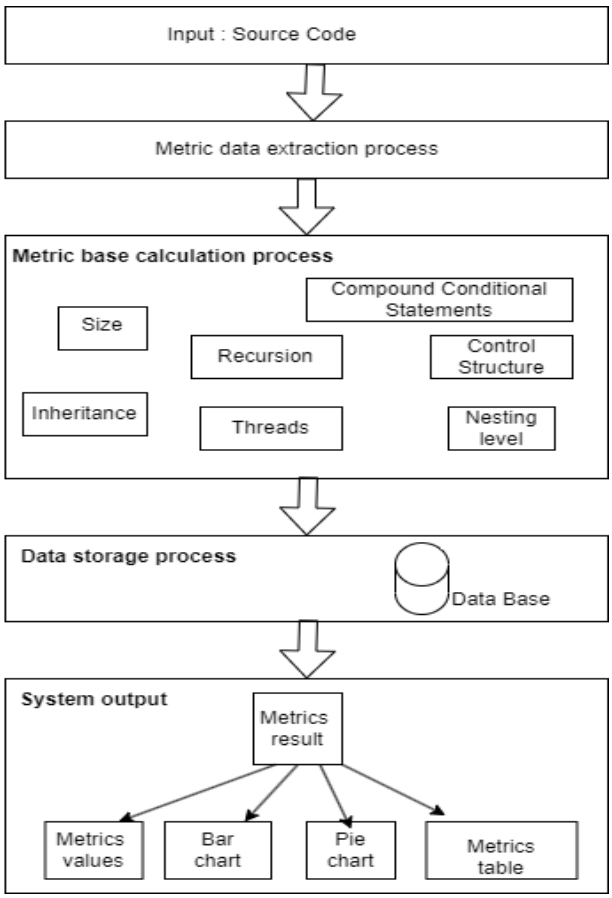

Fig.1. System functionality illustration

The implementation of the code complexity calculator was successful. Not only it provides the code complexity measuring results, but also provides some graphical charts to analyses the result and code elements as well. Some of graphs and charts are demonstrated “Fig. 2”, “Fig. 3”, “Fig. 4”, “Fig. 5” and "Fig. 6”. To implement the proposed functionalities the above-mentioned methodology was followed.

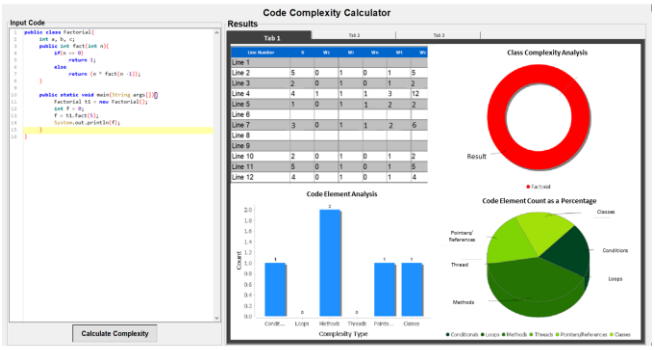

Fig.2. Interface with resulted graphs for the factorials

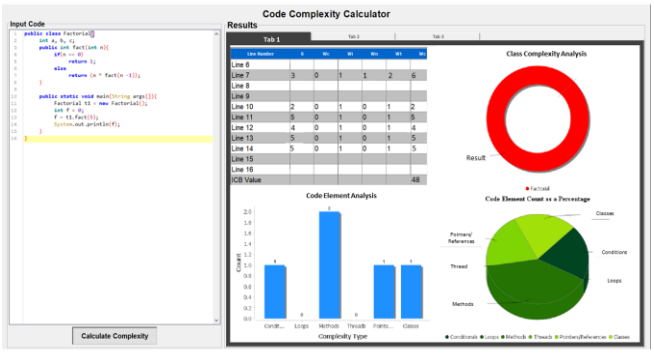

Fig.3. Interface with resulted graphs for the factorials

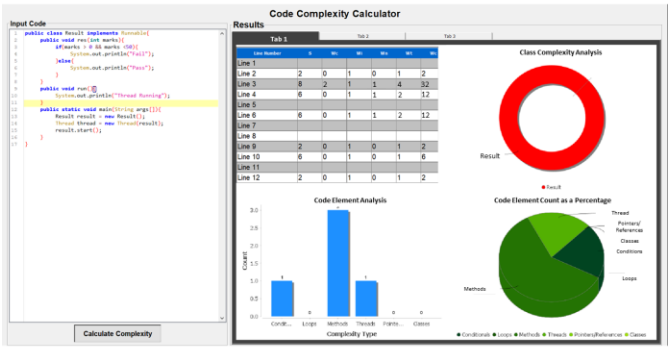

Fig.4. Interface with resulted graphs for the threads

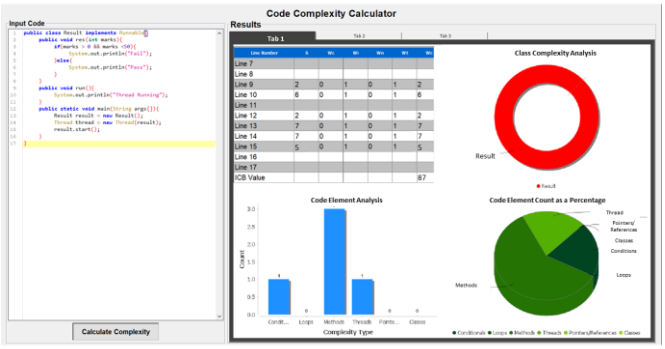

Fig.5. Interface with resulted graphs for the threads

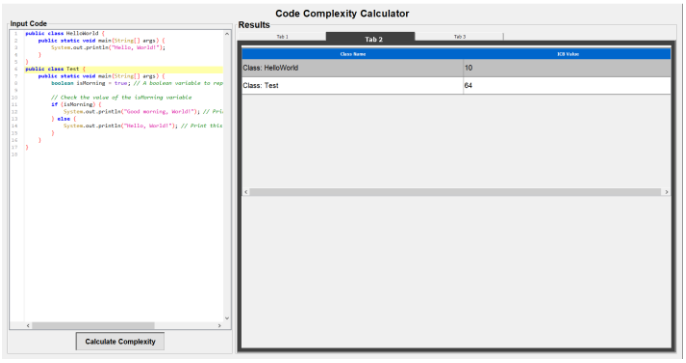

Fig.6. Interface with class wise complexity measurers

The tool was tested against many source codes from a variety of code models and it was evident that the results generated by the code complexity calculator are 100% accurate base on used ICB metrics. The sample test cases used to evaluate the accuracy and reliability of the code complexity calculator are displayed in “Fig. 7” and “Fig. 8”.

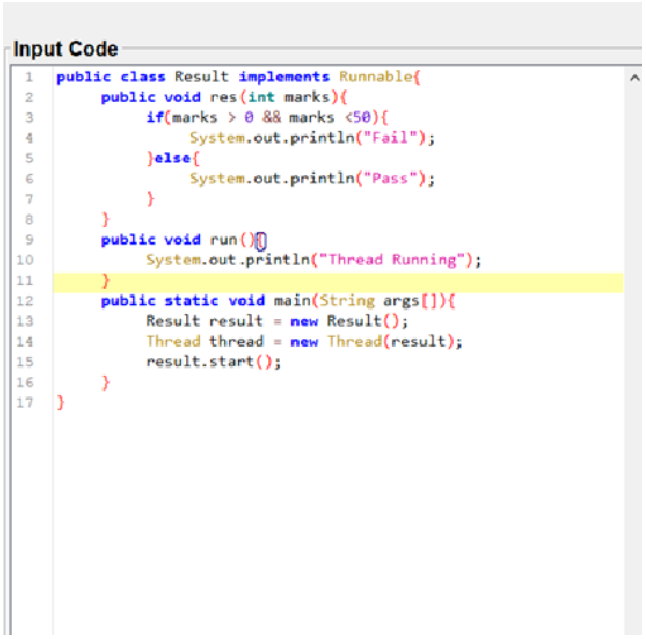

Fig.7. Sample program to demonstrate the calculation if ICB value.

Line Number	S	Wc	Wi	Wn	Wt	Wc
Line 1						
Line 2	2	0	1	0	1	2
Line 3	8	2	1	1	4	32
Line 4	6	0	1	1	2	12
Line 5						
Line 6	6	0	1	1	2	12
Line 7						
Line 8						
Line 9	2	0	1	0	1	2
Line 10	6	0	1	0	1	6
Line 11						
Line 12	2	0	1	0	1	2
Line 13	7	0	1	0	1	7
Line 14	7	0	1	0	1	7
Line 15	5	0	1	0	1	5
Line 16						
Line 17						
ICB Value						87

Fig.8. Complexity Calculation of above sample code

Above results proves the accuracy and the reliability of the code complexity calculator and it is also evident that this tool supports all the ICB metrics calculations which are not available in any other tools. Some of the major aspects of code complexity calculator are illustrated in the “Fig. 9”. Throughout these major functions, the tool provides more accurate results. Mainly this kind of tools are used by the higher management of the software engineering industry. So, this can acquire a huge market-share in the context of code complexity measuring tools in software engineering industry.

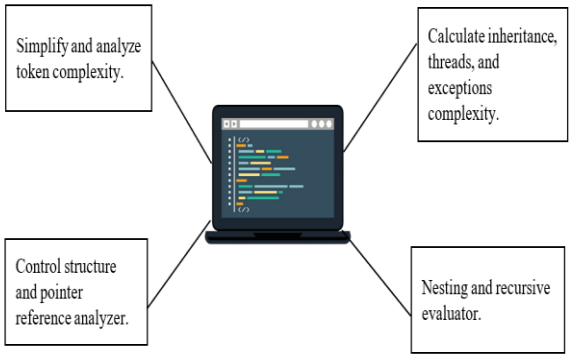

Fig.8. Main functions of tool

6. Conclusion

In conclusion, the Code Analyzer Tool emerges as a pivotal resource in the realm of software development. Its systematic approach to code quality assessment, underpinned by a comprehensive array of Improved CB (ICB) metrics, provides developers and teams with a robust framework for evaluating and enhancing code quality. This tool excels in efficiently identifying intricate areas of code complexity, including size, control structures, inheritance, nesting, and more. By illuminating these aspects, it empowers teams to embark on targeted efforts to optimize their codebase, ultimately leading to code that is not only easier to maintain but also poised for future scalability. Moreover, the tool's ability to render data tables and pie charts fosters clear communication among developers and stakeholders, facilitating informed decision-making on code refactoring and structural improvements. With a focus on reducing technical debt and improving code quality, the Code Analyzer Tool exemplifies modern software development best practices and promises to be an indispensable asset for teams committed to delivering robust, maintainable software solution.

References

[1] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," in *Canadian Journal of Electrical and Computer Engineering*, Apr. 2003, pp. 69–74. doi: 10.1109/CJECE.2003.1532511.

[2] Singh, Dharmender, Misra, and Arun, "A modified cognitive information complexity measure of software. ACM SIGSOFT Software Engineering Notes," 2006, doi: 31. 1-4. 10.1145/1108768.1108776.

[3] S. Misra, "A Complexity Measure Based on Cognitive Weights," c) GBS Publishers and Distributors, 2006.

[4] S. Misra, "Modified cognitive complexity measure," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, 2006, pp. 1050–1059. doi: 10.1007/11902140_109.

[5] Misra and Sanjay, "An Object Oriented Complexity Metric Based on Cognitive Weights. Proceedings of the 6th IEEE International Conference on Cognitive Informatics," 2007.

[6] S. Misra and K. I. Akman, "Weighted Class Complexity: A Measure of Complexity for Object Oriented System Achieving Sustainable Development Goals through ICT/Software Engineering View project Modeling and Managing Compliance Requirements at Design and Runtime View project Weighted Class Complexity: A Measure of Complexity for Object Oriented System," 2008.

[7] J. K. Chhabra, "Object-Oriented Cognitive-Spatial Complexity Measures," 2009.

[8] S. Misra, I. Akman, and M. Koyuncu, "An inheritance complexity metric for object-oriented code: A cognitive approach," *Sadhana - Academy Proceedings in Engineering Sciences*, vol. 36, no. 3, pp. 317–337, Jun. 2011, doi: 10.1007/s12046-011-0028-2.

[9] Chhabra and Jitender, "Code Cognitive Complexity: A New Measure. Lecture Notes in Engineering and Computer Science," 2011.

[10] S. Bhasin and U. Chhillar, "A New Weighted Composite Complexity Measure for Object-Oriented Systems," 2015.

[11] De Silva D. I, Kodagoda N, Kodithuwaku S. R, and Pinidiyaarchchi A. J, *Improvements to a Complexity Metric: CB Measure*, 10th ed. 2015.

[12] D. I. De Silva, N. Kodagoda, S. R. Kodituwakku, and A. J. Pinidiyaarachchi, "Limitations of an Object-Oriented Metric : Weighted Complexity Measure.", 6th ed. 2015

[13] D. I. De Silva, "Enhancements to an OO Metric: CB Measure," *Journal of Software*, vol. 12, no. 12, pp. 72–81, Jan. 2018, doi: 10.17706/jsw.13.1.72-81.

[14] D. I. De Silva, N. Kodagoda, S. R. Kodituwakku, and A. J. Pinidiyaarachchi, *Analysis and enhancements of a cognitive based complexity measure*.

[15] De Silva D. I, *Analysis of Weighted Composite Complexity Measure*. 2016.

[16] G. Singh, S. Singh, and M. Monga, "Code Comprehending Measure (CCM)," *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY*, vol. 2, no. 1, pp. 9–14, Feb. 2012, doi: 10.24297/ijct.v2i1.6733.

[17] A. Aloysius and L. Arockiam, "Coupling Complexity Metric: A Cognitive Approach," *International Journal of Information Technology and Computer Science*, vol. 4, no. 9, pp. 29–35, Aug. 2012, doi: 10.5815/ijitcs.2012.09.04.

[18] S. Misra, I. Akman, and F. Cafer, "A Multi-paradigm Complexity Metric (MCM)," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011, pp. 342–354. doi: 10.1007/978-3-642-21934-4_28.

[19] A. K. Jakhar and K. Rajnish, "Measuring Complexity, Development Time and Understandability of a Program: A Cognitive Approach," *International Journal of Information Technology and Computer Science*, vol. 6, no. 12, pp. 53–60, Nov. 2014, doi: 10.5815/ijitcs.2014.12.07.

[20] M. A. Shehab, Y. M. Tashtoush, W. A. Hussien, M. N. Alandoli, and Y. Jararweh, "An Accumulated Cognitive Approach to Measure Software Complexity," *Journal of Advances in Information Technology*, pp. 27–33, 2015, doi: 10.12720/jait.6.1.27-33.

[21] S. Misra and A. K. Misra, "Evaluation and comparison of cognitive complexity measure," *ACM SIGSOFT Software Engineering Notes*, vol. 32, no. 2, pp. 1–5, Mar. 2007, doi: 10.1145/1234741.1234761.

[22] Y. Tashtoush, M. Al-Maolegi, and B. Arkok, "The Correlation among Software Complexity Metrics with Case Study." 2014

[23] A. A. Khan, A. Mahmood, S. M. Amralla, and T. H. Mirza, "Comparison of Software Complexity Metrics," 2016.