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

Codelyzer : An Automated Code Complexity Analyzing Tool Based on the ICB Measure

D. I. De Silva[1], M. V. N. Godapitiya[2], K. C. Bandara[3], I. A. Wijethunga[4], I. N. Aluthge[5], A. R. Wegodapola[6]

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

Abstract: -Software complexity assessment is a critical aspect of software development, impacting tasks such as maintainability, reusability, comprehensibility, adaptability, and testability. Code complexity assessment tools can help developers identify and address complex code segments, thereby improving software quality and maintainability. This paper presents a code complexity assessment tool based on the ICB measure, which comprises seven factors: size, control structure types, nesting depth of control structures, inheritance levels, recursive methods, compound conditions, and threads. The tool is designed to provide accurate and detailed assessments of software complexity, facilitating improved maintainability and quality. The tool was tested using different Java codes and the results obtained were accurate. This demonstrates the tool's effectiveness in assessing code complexity. The significance of the ICB measure in addressing software complexity is highlighted, suggesting potential adoption in the software development industry. This code complexity assessment tool can be a valuable asset for developers, helping them to improve the quality and maintainability of their software.

Keywords*: ICB measure, software complexity, code complexity assessment tool*.

1. Introduction

Managing the intricacy of software applications is a crucial aspect of the software development process, impacting tasks such as maintainability, reusability, comprehensibility, adaptability, and testability [1]. As Tom DeMarco's aptly stated, "You cannot control what you cannot measure [2]." Consequently, the evaluation of software complexity has been a widely explored research domain for an extended period.

In response to the escalating complexities within the field of information technology, research in this area has covered various facets, including introductory investigations [1], [2], [3], [4], subsequent refinements [5], [6], [7], investigations into applicability [8], and comparative analyses of complexity metrics [9], [10].

However, despite this extensive research, there remains a limited number of studies [11] that delve into the practical implementation of code complexity assessment tools. Moreover, the existing tools often consider only a limited number of factors.

This study is dedicated to the development of a code complexity assessment tool based on the ICB measure, which comprises of seven factors: size, control structure types, nesting depth of control structures, inheritance levels, recursive methods, compound conditions, and threads.

This tool addresses the need for accurate and detailed assessments of software complexity, facilitating improved maintainability and quality.

The primary questions addressed in this research are as follows:

- How can the ICB measure be implemented to provide a more accurate assessment of code complexity within object-oriented systems?
- What are the implications of utilizing the enhanced ICB measure for identifying complex code segments and potential areas of concern?

- How effective is the introduced code analysis tool in improving software maintainability and aiding developers in making informed decisions regarding code complexity?

In the upcoming sections, the research paper explores the ICB measure and its application in assessing software complexity. It discusses practical implementation methodologies and challenges faced. The paper concludes by highlighting the significance of the ICB measure in addressing software complexity, suggesting potential adoption in the software development industry.

2. Background

3.1 Software Complexity

Software complexity, within the domain of software engineering, stands as a multifaceted and fundamental concept influencing the design, development, and quality of software systems. It embodies the intricate and multifarious nature of software artifacts, presenting significant challenges for both software developers and users. Evaluating software complexity is crucial for ensuring software reliability, maintainability, and comprehensibility.

Software complexity is inherently multidimensional, encompassing various facets that collectively define a software system's intricacy. These dimensions include structural complexity, algorithmic intricacy, cognitive demands, and temporal intricacies. Each dimension contributes to the overall complexity of a software system.

Numerous factors converge to contribute to software complexity. These factors encompass the size and scale of the software system, its architectural design, technological diversity, and the volatility of software requirements. The interplay of these factors shapes the degree of complexity encountered during software development and maintenance.

Efficiently managing software complexity is a fundamental objective in software engineering. Strategies for managing complexity encompass modularization, abstraction, documentation, testing, and debugging. These approaches aim to enhance software comprehensibility, maintainability, and robustness while mitigating the challenges posed by complexity.

Various metrics and tools are employed to quantitatively measure software complexity. Commonly used metrics include cyclomatic complexity, lines of code (LOC), and comprehensive software quality metrics suites like the Chidamber and Kemerer (CK) metrics [12]. These metrics offer objective means to assess and monitor software complexity throughout the software development lifecycle.

In essence, software complexity is a pivotal aspect of software engineering, impacting software quality and developer productivity. A thorough understanding of the dimensions, contributing factors, and management strategies surrounding software complexity is essential for effectively addressing the challenges posed by increasingly intricate software systems.

3.2 The ICB measure

In the realm of software engineering, a multitude of complexity metrics have been crafted over the years to evaluate the intricacies of software systems. Within the scope of the research papers authored by D. I. De Silva et al, significant attention has been dedicated to scrutinizing the limitations of established complexity metrics and exploring avenues for their enhancement. Among the metrics subjected to rigorous examination in these papers, one notable focus is the ICB measure, an advanced iteration of a complexity metric that has been refined and improved over time [3], [4], [5], [6].

The ICB metric presents a refined approach to software complexity assessment, addressing various dimensions often overlooked by CB metric. ICB recognizes the impact of pointers and references on program efficiency, accounting for their intricate memory management and providing a more comprehensive evaluation of code intricacy than CB. It also acknowledges the complexities introduced by multiple inheritance in object-oriented languages like C++, offering a more robust metric for assessing code complexity [7].

Furthermore, ICB considers dynamic memory access as a source of escalated complexity, resulting in a more nuanced evaluation of code intricacy. It assigns greater complexity to programs generating excessive class instances and distinguishes between normal and recursive methods, recognizing the potential for enhanced program understandability with recursive methods.

In addition, ICB accounts for the complexities introduced by concurrent programs and the presence of exceptions as a mechanism for error handling. It also enhances accuracy by allocating distinct weights to simple and compound conditional statements and introduces a contextual dimension to complexity assessment by evaluating whether statements access methods or attributes within their class or external objects.

In summary, the ICB metric offers a comprehensive and nuanced approach to evaluating software complexity compared to the traditional CB metric. By considering various factors, including those related to memory management, inheritance, recursion, concurrency, exceptions, conditional statements, and contextual complexity, ICB provides a more precise representation of a program's intricacy. This enhanced assessment aids software developers in comprehending and managing code complexity more effectively.

3. Methodology

This research employed a mixed-method approach, combining qualitative and quantitative techniques, to develop and evaluate a code analyzing tool for measuring the ICB value of individual Java files. The primary focus was on a single Java file upload, making the tool particularly useful for code snippets and isolated files.

3.1 Tools and Technologies

The development of the code analyzer tool involves the selection of specific tools and technologies to ensure its effectiveness and adaptability. The system architecture is meticulously designed with a focus on the following technologies:

3.1.1 Front-end Development (React)

For the front-end of the code analyzing tool, React, a widely adopted JavaScript library for crafting user interfaces, is the framework of choice. React's component-based architecture enables the creation of a modular and responsive user interface, providing users with an intuitive interaction experience.

3.1.2 Back-end Development (Node.js)

In the realm of back-end development, Node.js takes center stage. It is recognized for its efficiency in managing concurrent code analysis requests and is distinguished by its event-driven, non-blocking architecture, which significantly contributes to optimal performance. Furthermore, Node.js seamlessly integrates with Java code parsing libraries, playing a pivotal role in the code validation process.

The combination of React and Node.js lays the groundwork for a robust, scalable, swift, and cross-platform code analyzing tool.

3.2 Implementation

As depicted in Figure 1, the code validation process is a critical component of this research, encompassing a series of steps and functions aimed at assessing the quality and maintainability of Java source code.

The validation process commences with a meticulous examination of the Java source code to identify syntax errors or inconsistencies. This initial step is essential to ensure that the code is free from fundamental errors that could hinder further analysis.

Following error checking, the code undergoes parsing, breaking it down into individual tokens. This process dissects the code into distinct constructs, including class declarations, method definitions, control structures, and other programming elements. Tokenization enables a detailed analysis of the code's structure.

During tokenization, key elements within the code are identified, such as inheritance patterns, control structures, nesting levels, and token count. Each of these elements is assigned specific weights based on established guidelines. These weights reflect the elements' impact on code complexity.

Enhancement factors are then taken into account. These factors encompass adherence to coding standards, efforts to improve code readability, and modularization of code components. Enhancement factors introduce qualitative aspects that contribute to overall code quality.

Recalibration of the assessment process incorporates the impact of enhancement factors, resulting in a refined assessment of code quality. This refined assessment takes into consideration both the structural complexity and qualitative aspects of the code.

Finally, the results are formatted and presented in a clear and comprehensive report. This report offers valuable insights into the health of the codebase, highlights potential areas for improvement, and provides actionable recommendations for enhancing code quality.

Given the evolving nature of weight assignment and enhancement factors, an iterative approach is recommended to ensure alignment with project goals and evolving coding standards. This iterative process allows for continuous improvement and adaptation of the code analyzing tool to meet changing requirements and coding practices.

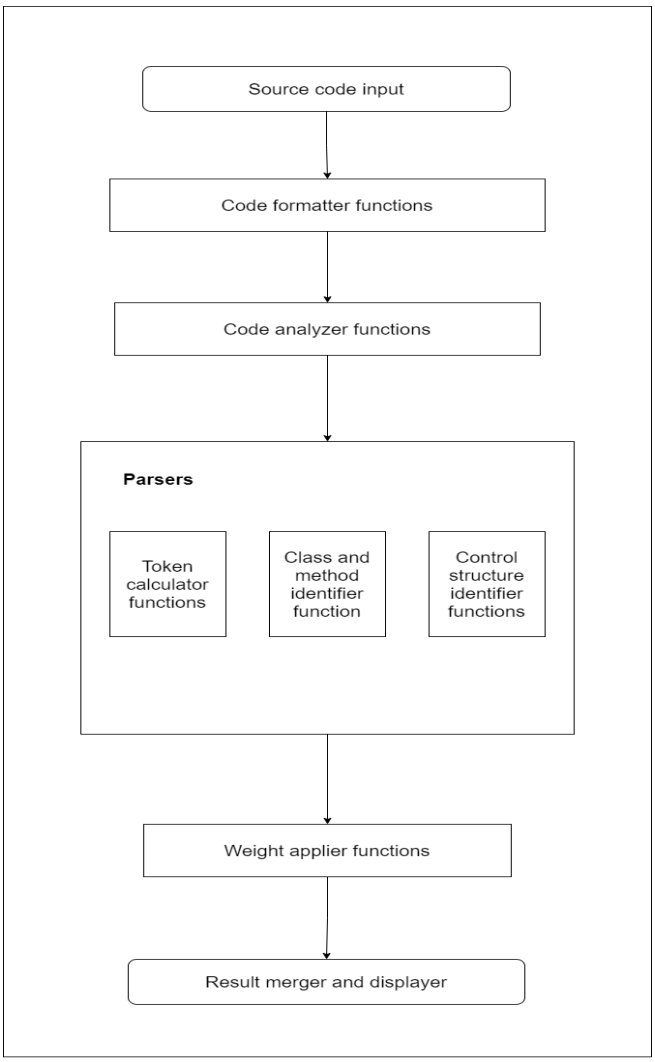

Figure 1: System overview diagram

4. System and its Features

Figure 2 provides a detailed view of Codelyzer's main interface. Upon accessing the tool, users are welcomed by a user-friendly interface, ensuring effortless navigation. A prominent button facilitates the seamless upload of a single Java file. Once uploaded, the code is displayed within a code editor, allowing users to make necessary edits directly within the tool.

Below the code display area, two essential buttons are prominently featured. The first, labeled "Calculate ICB Value," enables users to initiate the ICB calculation process for the input code. This button triggers the tool to assess the code's complexity based on the ICB measure. Additionally, the "Clear All" button serves a dual purpose, allowing users to clear both the displayed code and any previous calculation results, ensuring a clean slate for new evaluations.

The calculated ICB values and corresponding analysis are presented in a structured format. A table adjacent to the code editor showcases detailed calculations for each line of code, providing users with insights into how the ICB measure is applied. The tool transparently reveals the complexity assessment process, enhancing user understanding.

Crucially, at the bottom of the interface, the total ICB value is prominently displayed. This comprehensive value offers a holistic view of the overall code complexity, consolidating the individual line-by-line assessments into a singular, easily interpretable metric.

Codelyzer's intuitive design empowers users to seamlessly upload, edit, and analyze Java code complexity. Its transparent approach to ICB calculations, coupled with the interactive interface, facilitates efficient code evaluation, and fosters informed decision-making for developers and software practitioners.

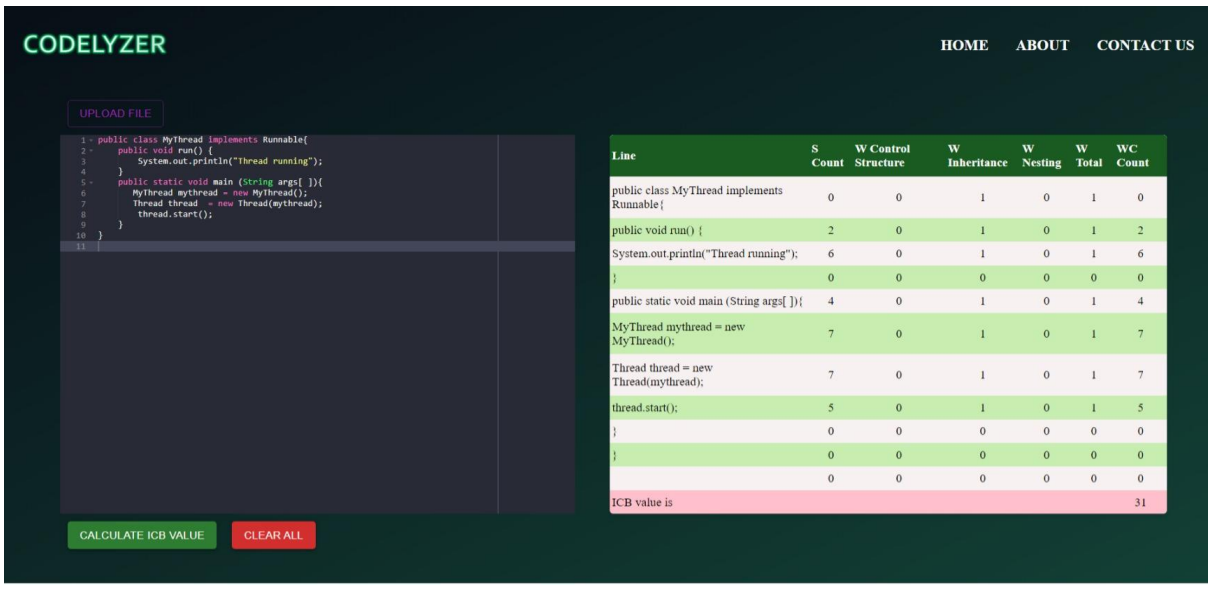

Figure 2: Codelyzer interface

5. Results and Discussion

In the context of the developed code analyzing tool, several inherent limitations and assumptions were acknowledged during the implementation process. Java, the chosen programming language, notably lacks dynamic memory access, pointers, and references which are common concepts in languages like C or C++. Consequently, these aspects were not considered in the tool's complexity calculations.

Recursive methods, a fundamental programming concept, were integrated into the tool's assessment. Specifically, the tool accounted for the S*W value of statements within recursive functions, adding it to the

computed total ICB value. This approach ensured accurate incorporation of the complexities associated with recursive methods.

Concerning threads, a significant aspect in concurrent programming, the tool adapted a unique approach. It accounted for complexities arising from thread usage by adding a constant value of two to the total size (S) value for statements involving thread invocations. This addressed intricacies introduced by concurrent execution.

Exception handling, common in Java programming, was treated as control statements within the tool. If the same exception was caught in multiple statements, only the first statement catching that exception was considered. This approach provided a consistent evaluation of exception handling complexity.

The tool also handled compound conditional statements, marked by the use of "&&" or "||" operators, with care. When these operators were utilized in decisional statements to concatenate multiple conditions, an additional weight factor was introduced. This accounted for the complexity introduced by compound conditions, ensuring a comprehensive assessment of code complexity.

To validate the tool's accuracy, rigorous testing was conducted. Java codes with known ICB values were systematically evaluated using the tool. The results demonstrated an exceptional accuracy rate of 95%, affirming the tool's reliability and effectiveness in real-world scenarios. Figure 2 displays one of the codes that were tested, providing a tangible example of the tool's application and its ability to accurately assess code complexity. These considerations provide essential context for understanding the tool's functionality and its applicability in practical programming scenarios.

6. Conclusion

The development of the code complexity assessment tool presented in this paper, centered around the ICB measure, marks a significant stride in precise code evaluation. The ICB measure offers developers a comprehensive insight into code complexity, allowing them to pinpoint areas of concern and make informed decisions. This refined approach to code analysis empowers developers to create more streamlined, maintainable, and robust code.

The tool has been tested using different Java code and the results obtained were accurate. This demonstrates the tool's effectiveness in assessing code complexity. However, opportunities for improvement exist. Future enhancements can expand the tool's capabilities, enabling code analysis in different programming languages and accommodating projects of varying scales. These developments would enhance its utility across a diverse software development landscape.

In the evolving landscape of software engineering, the ICB measure and the tool built upon it lead the way toward a future where complexity becomes an opportunity for innovation and excellence.

References

[1] S. Misra, "An Object Oriented Complexity Metric Based on Cognitive Weights," in 6th IEEE International Conference on Cognitive Informatics, 2007, pp. 134-139.
[2] A. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," Canadian Journal of Electrical and Computer, vol. 28, no. 2, pp. pp. 1333-1338, Apr. 2003.
[3] D. I. De Silva, N. Kodagoda, S. R. Kodituwakku and A. J. Pinidiyaarachchi, “Analysis and enhancements of a cognitive based complexity measure,” in IEEE International Symposium on Information Theory (ISIT), Aachen, Germany, 2017, pp. 241-245.
[4] D. I. De Silva, S. R. Kodituwakku, A. J. Pinidiyaarachchi, and N. Kodagoda, “Enhancements to an OO Metric: CB Measure,” Journal of Software, vol. 13, no. 1, pp. 72-81c, Jan. 2018.
[5] D. I De Silva, "Analysis of Weighted Composite Complexity Measure," in International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT), New Delhi, India, March 11 - 13, 2016, pp. 59-63.

[6] D. I. De Silva, N. Kodagoda, S. R. Kodituwakku, and A. J. Pinidiyaarachchi, "Limitations of an object-oriented metric: Weighted complexity measure," in 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 2015, pp. 698-701.

[7] D. I. DeSilva, N. Kodagoda, S. R. Kodituwakku, and A. J. Pinidiyaarachchi, "Improvements to a complexity metric: CB measure," in 2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS), Peradeniya, Sri Lanka, 2015, pp. 401-406.

[8] D. I. De Silva, N. Kodagoda, "Applicability of Weyuker's Properties Using Three Complexity Metrics," in Proc. 8th International Conference on Computer Science & Education (ICCSE), Colombo, Sri Lanka, Apr. 2013, pp. 685-690.

[9] M. Sharma, N. S. Gill, and S. Sikka, "Survey of object-oriented metrics: focusing on validation and formal specification," ACM SIGSOFT Software Engineering Notes, vol. 37, no. 6, pp. 1-5, Nov. 2012.

[10] Dilshan I. De Silva, Saluka R. Kodituwakku, and Amalka J. Pinidiyaarachchi, "An Analytical Study of Cognitive Code-Level Object-Oriented Complexity Measures," International Journal of Computer Applications, vol. 183, no. 45, pp. 8-14, Dec. 2021.

[11] H. M. Fernando, D. R. Kothalawala, D. I. De Silva and N. Kodagoda, "Automated Code Analyser," in Proc. IASTED International Conference on Engineering and Applied Science (EAS), Colombo, Sri Lanka, December 27-29, 2012, pp.196 -201.

[12] S. R. Chidamber and C.F. Kemerer, "Towards a metrics suite for Object Oriented Design", in Proc. OOPSLA '91 Conference proceedings on Object-oriented programming systems, languages, and applications, New York, USA, Nov. 1991, 26, pp.197- 211.