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

Automated Code Analyzer Based on the ICB Metric

D.I. De Silva[1], M.V.N. Godapitiya[2], F.M. Sudhais[3], M.K.F. Saara[4], M.J.S Ahmed[5], M.M.M.S. Saraf[6]

Faculty of Computing, Sri Lanka Institute of Information Technology,New Kandy Road, Sri Lanka.

Abstract:-In the rapidly evolving domain of software development, the imperative for reliable tools to gauge code quality and offer invaluable insights has reached a critical juncture. This research paper introduces an ingenious Automated Code Analysis (ACA) system meticulously crafted to cater to the unique requisites of both developers and students. ACA transcends the conventional boundaries of code evaluation by not only scrutinizing source code but also bestowing users with the gift of automated software metrics, actionable feedback, and profound analyses. This paper presents an exhaustive panorama of ACA, emphasizing its fundamental attributes, expansive capabilities, and the transformative influence it holds over software development methodologies. The dynamic nature of modern software development demands tools that not only adapt but also innovate. ACA embodies this spirit of adaptability and innovation, delivering an all-encompassing solution that empowers its users to elevate their coding prowess. Beyond conventional code scrutiny, ACA harnesses the power of data-driven metrics, facilitating informed decisions and insights into codebase enhancements. This paper embarks on a journey through the intricate realm of ACA, unearthing its potential to redefine code analysis practices, embolden developers and students alike, and cultivate a culture of code excellence. The study delves deep into ACA's intricate architecture, exploring how it transforms software development paradigms by equipping practitioners with the means to not just evaluate code but to perfect it. With ACA at the helm, software development enters an era where precision, efficiency, and excellence converge, propelling the industry towards uncharted horizons of innovation.

Keywords*: automated code analysis, software metrics, code quality, code complexity, software development tools*

Introduction

The realm of software development is witnessing unprecedented growth and innovation, with an increasing emphasis on software quality and measurement. As the software landscape becomes more intricate, the demand for automated tools to assess and enhance code quality has surged. This paper introduces an original Automated Code Analysis (ACA) system designed to address the unique challenges faced by developers and students in today's software development ecosystem.

Unlike conventional code analysis tools, ACA stands out as a comprehensive solution that offers a wide array of functionalities tailored to the needs of its users. It not only evaluates source code but also equips users with automated software metrics, providing actionable insights to optimize their coding practices. ACA's capabilities extend beyond mere code analysis; it serves as a valuable resource for developers and students seeking to improve their coding skills and produce higher-quality software.

The existing landscape of code analysis tools presents certain limitations, including a lack of structured metrics, limited user interaction, and an inability to relate metrics to external software quality attributes. ACA was developed to overcome these challenges and provide a more holistic approach to code analysis. By categorizing metrics into key areas such as code complexity, size, and maintainability, ACA offers a comprehensive view of software quality, enabling users to make informed decisions and improvements in their codebase.

This paper will delve into the core features and functionalities of ACA, illustrating how it can enhance the software development process. The study will explore its potential to revolutionize code analysis practices, empower developers and students, and ultimately contribute to the creation of higher-quality software in an increasingly complex software development landscape.

A background study of the software metrics is given in the next section. Complexity metrics which have been proposed mainly based on the cognitive aspect are given insection3. The methodology that was used to build the web application is discussed in section 4. Section 5 proposes the overview of the system. And section6 proposes the result and discussion of the paper. Finally, the conclusion of the paper is given in section 7.

1. Background Study

In the ever-evolving landscape of software development, the need for robust tools to assess code quality and provide valuable insights has become paramount. As the software landscape becomes more intricate, the demand for automated tools to assess and enhance code quality has surged. This paper introduces an original Automated Code Analysis (ACA) system designed to address the unique challenges faced by developers and students in today's software development ecosystem.

A. Importance of Software Complexity Metrics

Software Complexity metrics offer valuable insights into the quality of codebases. They provide a means of detecting issues and improving software quality, facilitating more effective planning and decision-making throughout the development lifecycle by reducing time and effort. However, the existing landscape of code analysis tools presents certain limitations, including a lack of structured metrics, limited user interaction, and an inability to relate metrices to external software quality attributes.

B. The Weighted Composite Complexity (WCC) Measure

To address the limitation of single-factor complexity metrics, Chhillar and Bhasin proposed the Weighted Composite Complexity (WCC) measure. This innovative approach considers multiple factors to measure code complexity effectively. WCC incorporates four key factors: Inheritance level of classes (Wi), Type of control structures in classes (Wc), nesting level of control structures (Wn), and the size of a class in terms of token count. By assigning different weights to these factors, the WCC measures provides a more accurate representation of code complexity.

C. Integration of WCC Metrics with ACA

In the context of the project, the study aims to leverage the Weighted Composite Complexity (WCC) metrics in a unique and innovative manner through the Automated Code Analysis (ACA) system. The project seeks to enhance the coding experience by introducing syntax highlighting based on WCC values. This approach aligns with recent research by Chhillar and Bhasin, who demonstrated the importance of understanding program complexity.

By integrating WCC metrics with syntax highlighting within ACA, developers can gain real-time insights into the quality of their code. This integration allows them to identify potential complexity hotspots and make informed decisions during development. Such as approach aligns with the principles of developer-centric analysis, emphasizing the importance of effective decision-making throughout the software development process. The integration of WCC- based syntax highlighting can significantly enhance the coding experience, making ACA a valuable resource for developers and students alike.

In summary, software complexity metrics, particular the Weighted Composite Complexity (WCC) measures, offer a promising avenue for improving code quality assessment. By integrating WCC metrics into the Automated Code Analysis (ACA) system and its integration with the Weighted Composite Complexity (WCC) metrics. It provides a more structured and comprehensive overview, helping readers understand the significance of our research in the field of software development.

2. Cognitive Based Complexity Metrics

In the realm of software development, Cognitive Based Complexity metrics have emerged as crucial tools for evaluating the software systems. These metrics, introduced over the years by various research, aim to quantify software complexity by considering a range of factors. They encompass Cognitive weights assigned to basic control structures, the cognitive impact identifiers, operators, and lines of codes, as well as architectural aspects and inheritance. Notable metrics such as Cognitive Functional Size (CFS), Cognitive Weight Complexity Measure (CWCM), and Weighted Class Complexity (WCC).

In 2011, Sanjay Misra, Akman and Koyuncu presented the Cognitive Code Complexity measure (CCC), designed to evaluate the design of object-oriented (OO) programs. CCC operates in three stages, with the initial stage calculating complexity at the method level using cognitive weights assigned to Basic Control Structures (BCS). Notably, CC also incorporates the inheritance aspect, which influence the overall program complexity assessment in the third stage [6]. In the same year J.K. Chhabra introduced the Code Cognitive Complexity measure (CCC). This metric considers factor such as the absolute distance in terms of Lines of the Code (LOC) between a module's call and its use, input and output parameters, and Cognitive Weights. Significantly, Chhabra expanded the cognitive weights to encompass variable and constant data types [7]. The other metric was Chhillar and Bhasin CB measure [8]. It computed the complexity of a program using the following four factors.

D. Inheritance

A weight of zero, denoting executable statements suited in the base class. As ascend the inheritance hierarchy, a weighted of one is attributed to statements within the first derived class. Subsequently, this weigh increases incrementally, with each higher-level derived class receiving a weighted that is one unit higher than its predecessor. This hierarchy-based weight assignment method provides a means of quantifying the influence of inheritance on program complexity.

E. Type of control structures

Distinct weightings are assigned to different categories of control structures, which serves to reflect their inherent complexity. Sequential statements, considered relatively straightforward, are assigned a weight of zero. In contrast, conditional control structures are ascribed a weight of one, denoting a higher degree of complexity. Iteratively control structures are allocated a weight of two, signifying their even greater complexity.

F. Nesting level of control structures

Sequential statements are ascribed a weight of zero, reflecting their position as the least complex. Nested control structures, statements at each subsequent inner level receive higher weightings. This hierarchical weight assignment mechanism captures the increasing intricacy associated with statements embedded within nested constructs.

G. Size

The size of an executable statement is another vital dimension of program complexity assessment. This metric is determined by quantifying the constituents within a statement, including operators, operands, functions, and strings. Counting these elements provides an objective measure of the statement's complexity.

The CB metric is a good measure, it could be transformed into a more accurate measure by incorporating another three factors. The ICB measure introduces additional factors to assess code complexity beyond what the CB measure accounts for.

H. Compound conditional statements

Which involve "&&" or "||" logical operators. In the ICB measure, each of these operators is assigned a weight of one. This adjustment aims to reflect that each such operator increases the number of conditions checked by a decision statement. Unlike the CB measure, which treats all decision statements equally, the ICB measure distinguishes between simple and compound conditions.

I. Threads

In contrast to the CB measure, which adds a constant value of one for each operator, operand, method or function, and string in a statement, the ICB measure assigns a constant value of two to statements involving thread invocations. This distinction allows the ICB measure to capture the added complexity associated with threads.

J. Recursion

When assessing the impact of recursion, one first multiplies the size (S) of each statement within the recursive function by their respective weight (W) values. Following this calculation, the resulting values are then added to the overall ICB value of the program under evaluation.

The ICB measure provides a more nuanced evaluation of software complexity by considering compound conditional statements recursion, and accounting for the impact of threads, offering a potentially more accurate assessment of program intricacy than the CB measure alone.

3. Methodology

The utilization of IEEE standards in the creation of the code analyzer was imperative. This decision was driven by the recognition that in the pursuit of crafting software that is both of exceptional quality and meticulously organized, strict adherence to established industry standards is indispensable.

Across the entire software product lifecycle, software quality engineering is characterized as the systematic and structured administration of quality standards. To effectively endorse this concept, any alternative methodology should possess the capability to replicate the delineation of quality criteria, generate reports, assess code, and enhance the overall quality of the software.

The choice of the IEEE Standard 1061's "Software Quality Metrics Methodology" as the foundation for this project was made due to its highly appropriate framework for modeling software quality engineering practices.

Illustrated in Figure 1, the software quality metrics framework, as outlined in IEEE Standard 1061, exhibits a hierarchical arrangement and is intentionally crafted to offer adaptability. The initial phase within this framework involves the commencement of quality requirements, which serve to define the software's quality. These requirements are subsequently associated with specific quality attributes. Importantly, the framework allows for the inclusion, removal, and adjustments of quality factors and metrics [10].

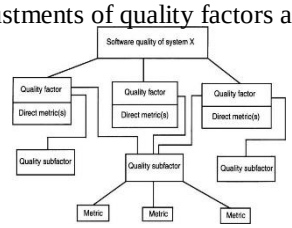

Fig. 1. Software quality metrics framework

The software quality metrics methodology provides organizations with a structured five-step approach for handling software processes. The specific details of these five steps and the process employed during the implementation of ACA are outlined below[10].

K. Initiate software quality requirements

- Identify quality requirements.
- Determine quality requirements.
- Quantify each quality factor.

During the process of identifying quality requirements, the study has emphasized that the following aspects of the software hold the utmost significance for evaluation.

- Size of the Statement. (Based on tokens)
- Type of control structures.
- Nested level of control structures.
- Inheritance level of statements.
- Conditional compound statements.
- Threads.
- Recursion.

L. Determining Software Quality Metrics

- Utilize the software quality metrics framework
- Conduct a cost-benefit analysis

For each of the identified quality aspects, the decision was made to calculate the following Software Metrics to measure and assess the software's quality.

M. Implement the Metric

- Establish the procedures for data collection.
- Create a prototype of the measurement process.
- Gather the necessary data and calculate the metric values.

N. Examine the Metrics Results

- Interpret and analyze the outcomes.

Compare the computed metric values with the user-defined threshold metric values.

O. Validate the Software Quality Metrics

- Execute the validation process.
- Employ specific validity criteria.

To ensure the quality metrics' validity, tests were conducted, and comparisons were made between the ACA's metric values and the real metric values.

4. System Overview

In accordance with the five steps outlined previously, ACA was specifically crafted for the purpose of determining the ICB [1] metric. The primary functions of this research can be summarized as follows:

- Develop a web application for analyzing source code using the ICB metric.
- Calculate metric values for a given code section based on the established metric framework.
- Present numerical representations of the computed metric values.
- Visualize measurement results through graphs for a more interactive user experience.
- Provide a comprehensive interpretation of the analysis outcomes, ensuring users gain a thorough understanding of the characteristics of the specified code portion.
- Allow users to input threshold values for factors, enabling them to apply their own criteria for evaluating their work.

- Display the source code and the relevant metric value in tabular format to make it easy for the users to understand the code.

To implement the proposed web application Visual studio code, ReactJs, NodeJS, ExpressJS, MongoDB, and Chevrotain were used.

5. Result and Discussion

The implemented ACA will show the result of the analysed code that was given to the system. And it performs many other functions. Some of these functions are demonstrated in Fig 2, Fig 3, and Fig 4.

```
public class Result{

  public void res(int marks){
   if(marks > 0 && marks < 50)
        System.out.println("Fail");
   else
        System.out.println("Pass");
  }

  public static void main(String args[]){
     Result r = new Result();
     r.res(50);
  }
}
```

Fig. 2. Sample code to analyse.

Class Name	Method Name	Line No	Statement	S	Wi	Wc	Wn	W	S*W
Result	res	2	public void res(int marks)	2	1	0	0	1	2
Result	res	3	if (marks >0 && marks <50)	8	1	2	1	4	32
Result	res	4	System.out.println('Fail')	6	1	0	1	2	12
Result	res	5	System.out.println('Pass')	6	1	0	1	2	12
Result	main	6	public static void main(String args[ ])	4	1	0	0	1	4
Result	mian	7	Result r = new Result()	5	1	0	0	1	5
Result	main	8	r.res(50)	3	1	0	0	1	3
			ICB Value						70

Fig. 3. Main interface after analysing the code and showing the output of the result.

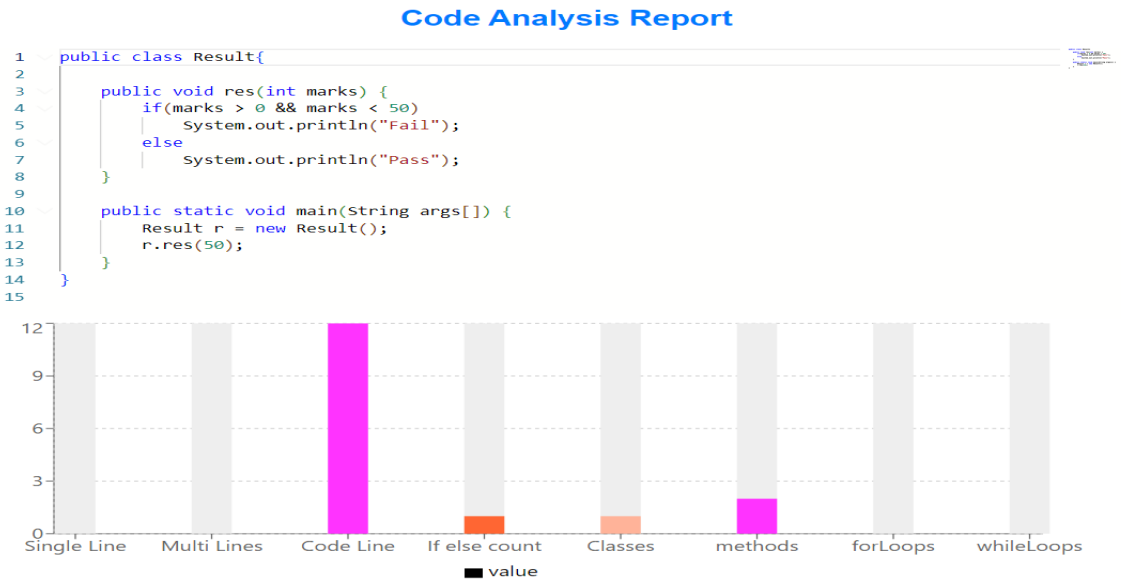

Fig. 4. Sample graphical interface

In Fig 2, the provided sample code serves as the test case for Automated Code Analyzer. Figs 3 and 4 represent the outcomes of the code analysis.

Fig 3 details the systematic analysis process, wherein the system evaluates each statement individually. It computes several metrics, including the statement's size, inheritance attributes, type of control structure employed, nesting level within control structures, the cumulative weight associated with each statement, and the product of statement size and weight. By aggregating these details for all statements, the system calculates the total ICB value for the analyzed code.

Moving on to Fig 4, it presents a graphical representation in the form of a bar chart. This chart provides insights into various aspects of the analyzed code, such as the count of single-line and multiline statements, code lines in general, occurrences of if-else constructs, the presence of classes and methods, as well as the utilization of for and while loops. These visualizations offer a concise overview of the code's structural characteristics and assist in comprehending its complexity and composition.

6. Conclusion

The overall goal of this research project can be summarized by saying that project members have created a well-structured web application for source code analysis based on the ICB metric calculation, especially for developers, which will help them a carry out their functions more effectively.

The study's aim was to construct a precise code- analyzing tool with the calculation of thread statements and compound statements in addition to inheritance levels of statements, control structure types, nesting levels of control structures, and program size. With the help pf the ACA, we were able to accomplish this goal. The following can be done to enhance ACA release in the future:

- Extend the application to handle programming languages such as C, C++, JavaScript, PHP, and Python.
- Improve the efficiency, reliability, and user friendliness of the application.

Refrences

[1] D. I. De Silva, N. Kodagoda, S. R. Kodituwakku, A. J. Pinidiyaarachchi, "Analysis and enhancements of a cognitive based complexity measure," IEEE International Symposium on Information Theory (ISIT), Aachen, Germany, 2017, pp. 241-245.

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

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

[4] D. I. De Silva, "Analysis of Weighted Composite Complexity Measure," International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT), New Delhi, India, March 11 - 13, 2016, pp. 59-63.

[5] D. I. De Silva, S. R. Kodituwakku, A. J. Pinidiyaarachchi, N. Kodagoda, "Enhancements to an OO Metric: CB Measure," Journal of Software vol. 13, no. 1, pp. 72-81c, 2018.

[6] S. Misra, I. Akman, and M. Koyuncu, "An inheritance complexity metric for object-oriented code: A cognitive approach", Sadhana Academy Proceedings in Engineering Sciences, 36 (3), 2011, pp. 317 – 377.

[7] J. K. Chhabra. "Code Cognitive Complexity: A New Measure", World Congress on Engineering, July 2011, Vol. 2, London, U. K.

[8] U. Chhillar and S. Bhasin, “A new weighted composite complexity for object-oriented systems”, International Journal of Information and Communication Technology Research, July 2011, 1(3), pp.101-108.

[9] S. Misra, F. Cafer, and I. Akman, “Multi-Paradigm Metric and its Applicability on Java projects”, Journal of Applied Sciences, 10(3), 2013, pp. 203-220.

[10] Software Engineering Standards Committee of the IEEE Computer Society, IEEE Standard for a Software Quality Metrics Methodology, Revision of IEEE Std 1061-1992, December 1998, pp. 3-10.