SPL-1 Project Report, 2023

CMC: Code Metric Calculator

SE 305: Software Project Lab – 1

Submitted by

Umme Kulsum Tumpa

BSSE Roll no: 1307

BSSE Session: 2020-21

Supervised by

Abdus Satter

Assistant Professor

Institute of Information Technology

Institute of Information Technology

University of Dhaka

[21 May, 2023]

Project Name: Code Metric Calculator

Prepared by:

Umme Kulsum Tumpa

Roll – 1307

BSSE 13th batch

Supervised by:

Abdus Satter

Assistant Professor

Institute of Information Technology,
University of Dhaka

Date submitted: 21 May, 2023

Supervisor's Approval:

_____

(Signature of Abdus Satter)

Table of Contents

1. Introduction

The Code Metric Calculator (CMC) is a software tool designed to measure and assess the quality and performance of Java source code. It provides developers with valuable insights into code metrics such as Lines of Code (LOC), Cyclomatic Complexity (CC), Weighted Methods per Class (WMC), Number of methods in a class, and Number of fields in a class. By analyzing these metrics, CMC helps developers identify areas for code improvement, optimize performance, and enhance overall code quality. With an intuitive interface and informative reports, CMC aims to assist developers in making informed decisions to ensure the maintainability and efficiency of their software projects.

1.1 Background Study

The following background study was necessary for building up CMC:

1.1.1 Code Metrics

- Definition and importance of code metrics in software engineering.
- Detailed explanation of key code metrics, including Lines of Code (LOC), Cyclomatic Complexity (CC), Weighted Methods per Class (WMC), and more.
- Methods of calculating each code metric and their interpretation in assessing code quality and complexity.

1.1.2 Software Quality

- Overview of software quality assurance principles and practices.
- Discussion on various dimensions of software quality, such as maintainability, readability, and testability.
- Examination of how code metrics contribute to evaluating and improving software quality.
- Exploration of industry-standard coding guidelines and best practices that align with code metric analysis.

1.1.3 Performance Analysis

- Introduction to performance analysis and its significance in software development.
- Discussion on profiling, benchmarking, and identifying performance bottlenecks.
- Explanation of performance metrics relevant to Java applications, such as CPU usage, memory consumption, and response time.
- Illustration of how code metric analysis can contribute to optimizing performance.

1.1.4 Java Programming

- Overview of the Java programming language, its syntax, features, and object-oriented principles.
- Explanation of Java-specific concepts, including classes, methods, fields, inheritance, and polymorphism.
- Relevance of Java programming knowledge for implementing code metric calculations and analyzing Java source code effectively.

1.1.5 Existing Tools and Libraries

- Examination of existing code metric calculators and software analysis tools in the industry.
- Evaluation of their features, limitations, and common use cases.
- Identification of relevant open-source libraries or frameworks for parsing and analyzing Java code.

1.2 Challenges

1.2.1 Creating the Control Flow Graph(CFG)

Constructing a CFG requires accurately identifying control flow structures, such as loops and conditionals, within the code.

So I introduced a basic graph representation with adjacency list data structure and represented each node having relation with some statement to represent the code's execution paths.

1.2.2 Handling Complex Control Flow

Dealing with complex control flow structures, such as nested loops or conditional statements, can introduce challenges in constructing an accurate CFG.

I read line by line of the source code to handle nested control flow structures and ensure the CFG accurately represents the code's control flow, accounting for all possible execution paths.

1.2.3 Exception Handling and Method Calls

Capturing exception handling blocks and correctly modeling method calls within the CFG was challenging.

I extended the parsing process to identify exception handling blocks and handled method invocations appropriately, ensuring that they are captured and represented accurately within the CFG.

1.2.4 Identifying Predicate Statements

Here the main challenge was identifying and distinguishing predicate statements, such as conditional statements (if, switch), ternary operators, and loops with boolean conditions.

I employed a combination of lexical and syntactic analysis techniques to identify and extract predicate statements accurately.

1.2.5 Counting decision points

Here I faced challenge to accurately counting the decision points within the predicate statements, which directly contribute to Cyclomatic Complexity (CC) calculation.

I implemented an algorithm that traverses the parsed predicate statements and increments a counter for each decision point encountered. Decision points can include branches within conditional statements, cases in switch statements, and loop conditions. By systematically analyzing the code's control flow, I accurately determined the number of decision points.

1.2.6 Handling Complex Predicates

Here the challenge was dealing with complex predicate structures, such as nested if-else statements, multiple conditions within a single predicate, or complex boolean expressions.

I carefully counted the nested and complex predicates by considering all possible execution paths.

1.2.7 Exception Handling and Error Cases

Ensuring accurate decision point counting in the presence of exception handling blocks or error cases within predicate statements.

I carefully accounted for exception handling blocks and error cases to avoid miscounting decision points.

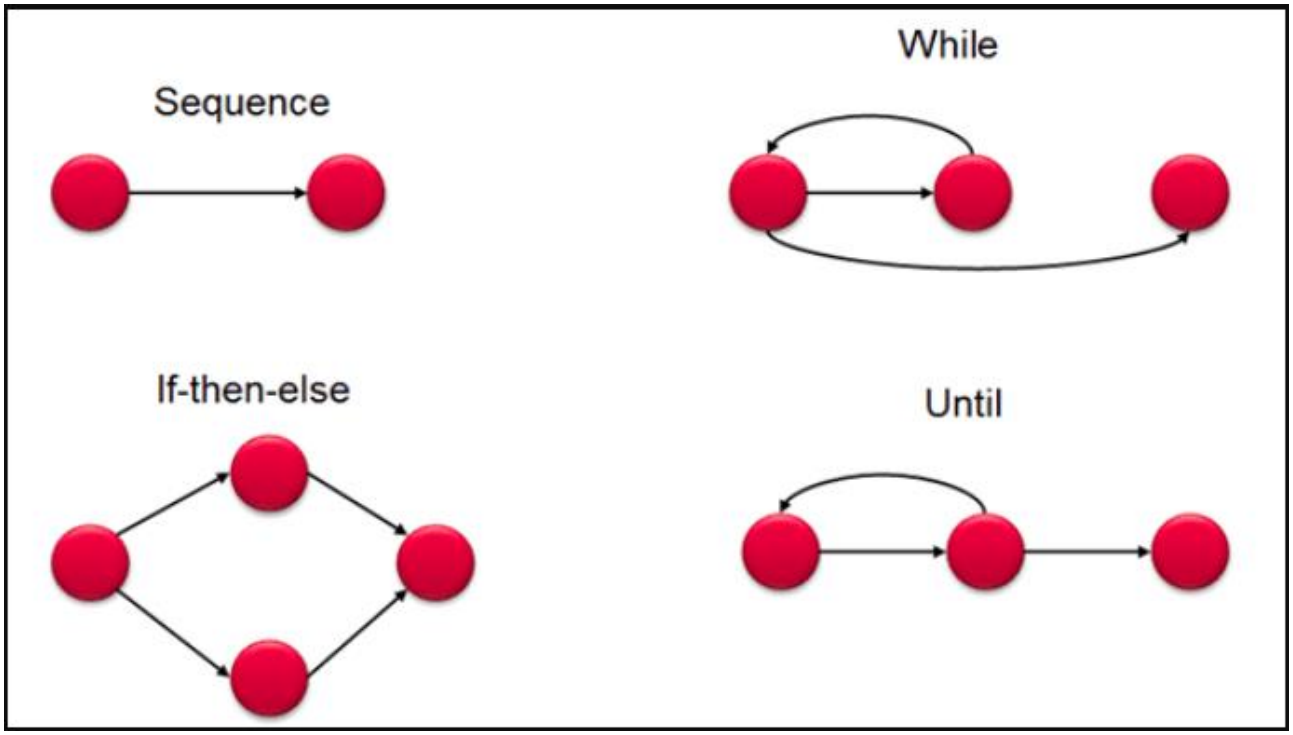
2. Project Overview

The Code Metric Calculator (CMC) is a software tool designed to measure various aspects of the quality and performance of Java source code. It provides a set of basic code metrics that can be used to evaluate the codebase and identify areas that may require improvement or optimization.

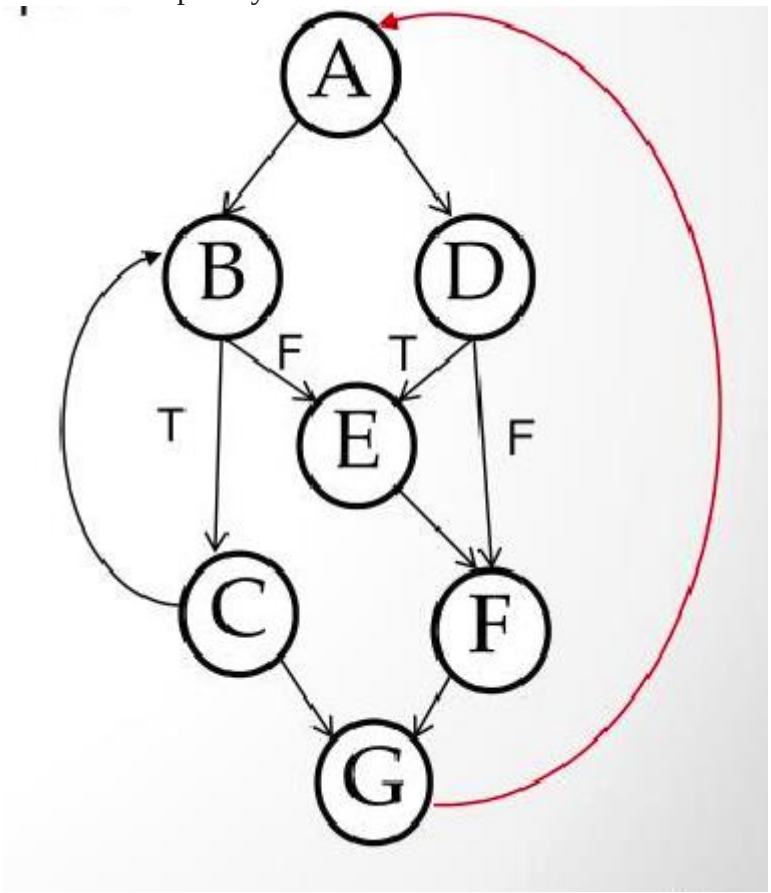
1. The first metric calculated by CMC is Lines of Code (LOC), which gives an indication of the size and complexity of the codebase. This metric helps developers understand the overall scale of the project and can be useful for measuring progress or estimating development effort.

2. The second metric is Cyclomatic Complexity (CC), which measures the complexity of the code's control flow. It counts the number of decision points and helps identify code segments with high complexity, which may be harder to understand, test, and maintain. By identifying such areas, developers can focus on simplifying the code and reducing potential bugs.

Im programming, a predicate node represents a decision point or a branching condition in the code, typically expressed through conditional statements which creates multiple possible paths execution, which contributes to the overall Cyclomatic Complexity.



Path Analysis: Cyclomatic complexity



If flow graph has link between sink node (G) to source node(A) which is called a strongly conected graph.

CC = e-n+1

e is the number of edges and n is the number of nodes
Here, for the following graph, CC = 11-7+1 = 5

3. The third metric, Weighted Methods per Class (WMC), calculates the number of methods within a class, taking into account their complexity. It provides an estimate of the overall complexity and potential maintainability challenges within a class.

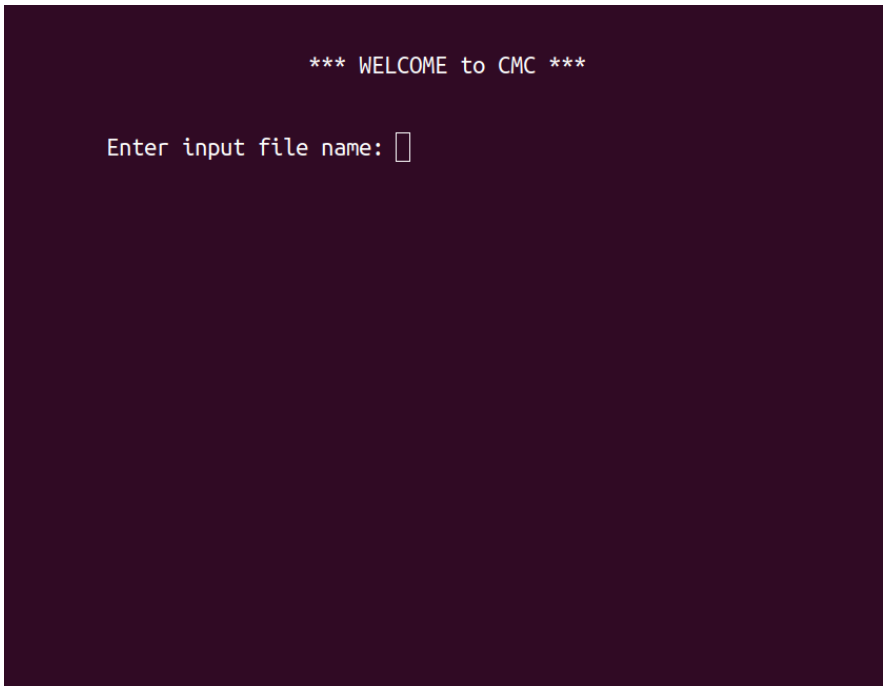
4. CMC also measures the number of methods and fields in each class separately, giving developers insights into the structure and organization of their code. This information can help identify classes that have too many methods or fields, which may indicate a need for refactoring or splitting the class into smaller, more manageable units.

Overall, the Code Metric Calculator aims to provide developers with quantitative measurements of code quality and performance. By analyzing these metrics, developers can make informed decisions about code refactoring, optimization, and improving overall code quality.

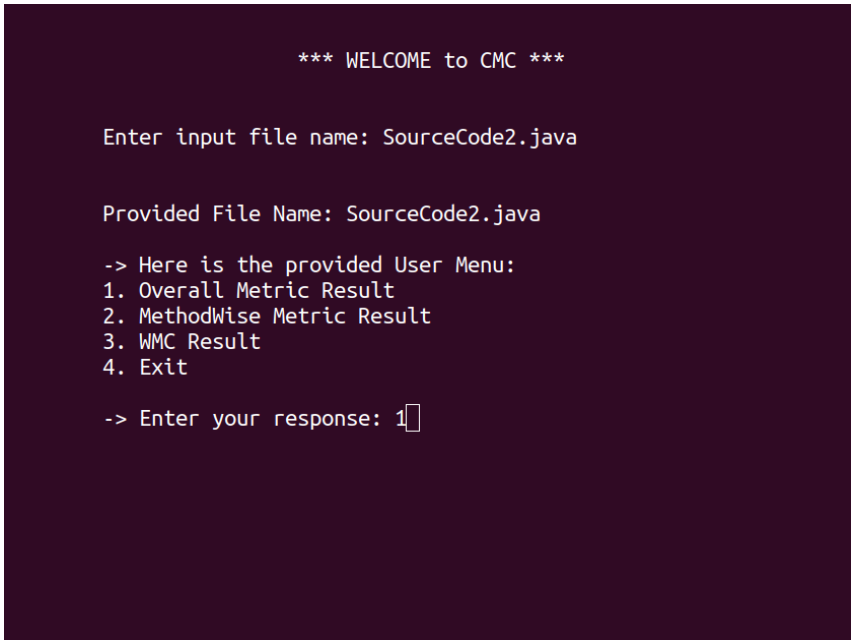
3. User Manual

This user manual provides instructions on how to use CMC effectively to assess code quality, identify areas for improvement, and optimize performance.

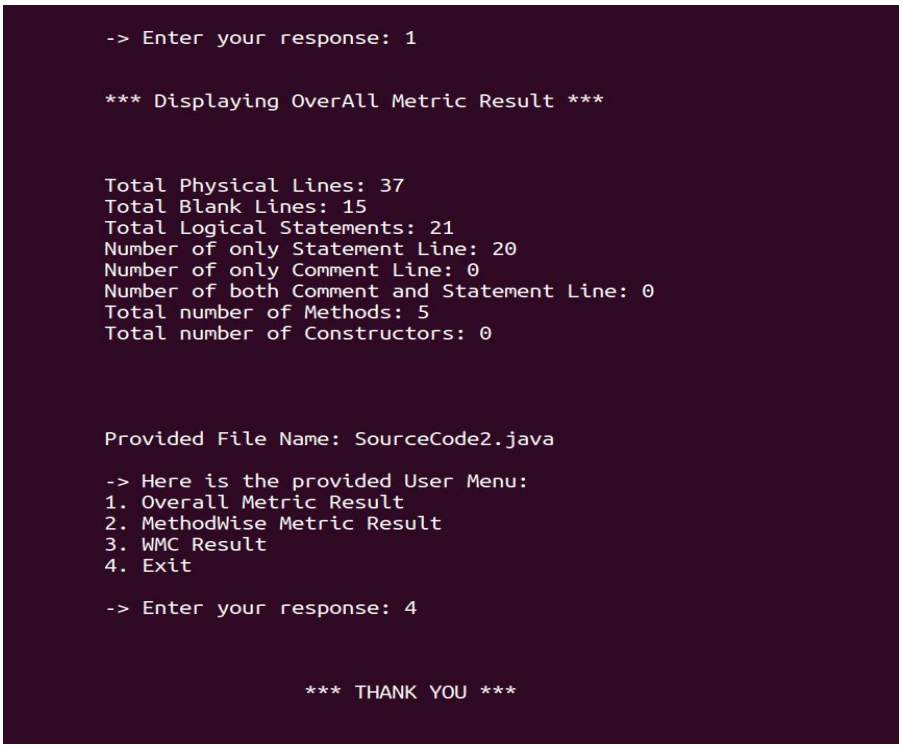
- Install the project
- Create a java source code file, remember the source code must be in .java format to execute the CMC tool properly, otherwise it will pop up error message
- Compile the executable file main.cpp file which includes all the necessary headers
- Run the compiled file, console would look like below



- Input the name of the source file which the metrics would be calculated
- A user manual list would appear and select the desired action by their corresponding key



- Result based on selected key would appear



- To exit CMC tool, select the corresponding key

4. Conclusion

In conclusion, the Code Metric Calculator (CMC) project has successfully developed a software tool that measures and analyzes code metrics for Java source code. By providing insights into metrics such as Lines of Code (LOC), Cyclomatic Complexity (CC), Weighted Methods per Class (WMC), Number of methods in a class, and Number of fields in a class, CMC assists developers in improving code quality and optimizing performance. With its user-friendly user manual and informative reports, CMC facilitates better decision-making and enhances the overall efficiency and maintainability of software projects. CMC proves to be a valuable asset for developers seeking to maintain efficient and high-quality code.

References

1. *McCabe, T. J. (1976). A Complexity Measure. IEEE Transactions on Software Engineering, 2(4), 308-320.*
2. *Li, H., Wang, S., & Zhang, Y. (2015). Investigating the Relationship between Object-Oriented Metrics and Fault-Proneness in Open-Source Software Systems. Information and Software Technology, 58, 52-63.*
3. *Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering, 20(6), 476-493.*
4. *Seaman, C. B. (1994). Measuring and Assessing Rapid Application Development. IEEE Software, 11(2), 42-49.*
5. *Rombach, H. D., & Basili, V. R. (1993). Goal-Driven Software Measurement. IEEE Transactions on Software Engineering, 19(8), 787-793.*