SCMA: A Lightweight Tool to Analyze Swift Projects

Fazle Rabbi
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
bsse0725@iit.du.ac.bd

Syeda Sumbul Hossain
Department of Software Engineering
Daffodil International University
Dhaka, Bangladesh
syeda.swe@diu.edu.bd

Mir Mohammad Samsul Arefin
Information Engineering
Chalmers University of Technology
Gothenburg, Sweden
s.arefin@outlook.com

***Abstract*—In global software engineering, practitioners use code metrics analyzers to measure code quality to detect code smells or any technical debt early at the development phase. Different tools exist to evaluate these metrics to ensure the maintainability and reliability of any codebase. This paper presents a tool SCMA (Swift Code Metrics Analyzer) which analyzes swift code considering ten code metrics for analyzing software architecture to ensure code quality. We have used the native swift parser to implement this tool. This tool suggests refactoring the codebase by giving a final score averaging the score of all ten metrics. We have validated the accuracy of each metric measured by this tool by analyzing the codebase manually. This tool can help the developers to inspect the swift modules of iOS projects and give an insight into the improvement area of each project.**

***Index Terms*—Code metrics, Code Metrics Analyzer, Swift Language, Code Quality, Code Smell**

I. Introduction

Nowadays, software companies are evaluating their projects in terms of different software metrics. These metrics are categorized into three different types- Process metrics, Project metrics, and Product metrics [3]. These metrics are evolving to measure the software development processes to enhance the development models. Numerous studies further have been done on these metrics. A systematic mapping study [4] has been done on software process metrics where process metrics had categorized into three different types. Software Product metrics are also studied over time which is stated in another mapping study [1]. Among those metrics, product metrics define product quality. Source code metrics are one of the product metrics that helps to measure the code quality of any codebase [8]. There are around 300 code metrics found [8] in different literature.

Clean architecture becomes a buzzword among the software engineering practitioners motivated by R. Martin (uncle Bob) [7]. To ensure the code quality, architectural analysis of the codebase is crucial to avoid code smells or any other technical debts. There are different types of code smells are stated in different literature. These code smells tends to code refactoring [2].

This paper presents a tool SCMA (Swift Code Metrics Analyzer) which analyzes swift code considering ten code metrics for analyzing software architecture to ensure code quality.

DOI reference number: 10.18293/SEKE2022-006

The rest of the paper is organized in accordance: Background Study at Section II followed by Tool Description and Discussion at Section III and Section IV respectively. We have reviewed the validity of our study in Section V. Finally Section VI furnishes our contribution.

II. Background Study

A. Technical Debt (TD)

Technical debts are short-term benefits that occurred by the software engineers unintentionally throughout the software development processes. There are different types of technical debts stated in studies. A systematic mapping study [6] had been done on 94 studies where technical debt is classified into 10 different types as Requirements TD, Architectural TD, Design TD, Code TD, Test TD, Build TD, Documentation TD, Infrastructure TD, Versioning TD, Defect TD. This paper also stated that Code TD had studied most.

B. Code Metrics

Code metrics are being used to get the insight of source code written by the developers in terms of some measurement units. As these metrics reveal the code health of any codebase at the development phase, practitioners can easily get the idea of improvement areas of their codebase. Considering the nature of projects and processes in global software development, practitioners are using different code metrics. Kitchenham, B. [5] had conducted a preliminary mapping study on software metrics. A systematic mapping study [8] has been done on software code metrics where a total of 226 studies have been gone through to map almost 300 code metrics of 13 different programming languages (Java, AspectJ, C++, C, C#, Jak, Ada, Cobol, Javascript, Pharo, PHP, Python, and Ruby). This study also summarizes 41 different software metrics tools into two categories: commercial tools which are free or paid tools and others that are developed by the authors. Another systematic mapping study [9] had been performed on dynamic software code metrics which illustrated that coupling, cohesion, complexity, method invocation, memory allocation and usages were mostly focused research topics.

C. *Tool used within our organization*

In our organization we are using our own tool, to measure the different code metrics associated with their organization. This tool summarizes a score using the code metrics of LOC (Line of Code), Global Variables, Predefined processors, Cyclomatic Complexity, Duplicate Code, and Modular Circular Dependency. This tool supports projects with languages like Java, C, C++, C# excluding iOS supported languages, especially swift. To be aligned with the organization's coding culture, we are motivated to develop a code metrics analyzer that supports swift language.

D. *Code Metrics Tools for Swift*

Swift language has evolved later in 2014 and becomes more popular for iOS development. Clean Swift [1], a set of rules, is introduced with XCode for maintaining better architecture for swift projects. As per the best of our language, there are a few software code metrics tools that exist for swift language which measures some basic code metrics. SonarSource [2] is a static analyzer that checks 119 predefined rules for swift language which covers unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in SWIFT code. It also supports other 26 languages and has specific rule sets based on the languages.

Code Climate [3] is a repository-based metric analyzer for swift language that checks duplication, cognitive and cyclomatic complexity and some others basic checks.

Swift Code Metrics is another tool for swift projects [4] by which some basic code metrics (the overall number of concrete classes and interfaces, the instability and abstractness of the framework, the distance from the main sequence, LOC (Lines Of Code), NOC (Numbers Of Comments), POC (Percentage Of Comments), NOM (Number of Methods), Number of concretes (Number of classes and structs), NOT (Number Of Tests), NOI (Number Of Imports), and Frameworks dependency graph (number of internal and external dependencies)) can be analyzed.

Taylor [5] is another tool for analyzing swift code which considers the code metrics Excessive Class Length, Excessive Method Length, Too Many Methods, Cyclomatic Complexity, Nested Block Depth, N-Path Complexity, and Excessive Parameter List. SwiftLint [6] checks over 200 rules, including 12 code metrics for swift languages.

III. Tool Description

In this section, we presented the overall activities of our tool from parsing source codes to generating html reports. The score calculation method from metrics is also shared.

[1]https://clean-swift.com/
[2]https://rules.sonarsource.com/
[3]https://codeclimate.com
[4]https://github.com/matsoftware/swift-code-metrics
[5]https://github.com/yopeso/Taylor
[6]https://github.com/realm/SwiftLint

A. *SOURCE CODE PARSING*

At the very beginning, the source code files with .swift extensions are selected from a project. The contents are read one by one and converted into Abstract Syntax Trees (AST). To parse these files and AST preparation, SwiftSyntax [7] is used as a parser. Each of the AST contains a class, methods under the class, global variables and variables under methods, and other elements from a source code file as branches in hierarchy order. Figure 1 illustrates the parsing procedure of source codes.

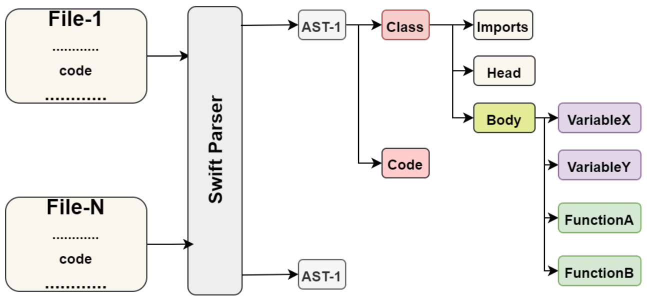


Fig. 1. Source Code Parsing

B. *METRICS DETAILS*

After ASTs are generated from source code files, information such as class names, lines of codes, method names, variables names are collected from the ASTs. From that information, a call graph matrix is prepared for a project where every node (methods, variables) is linked to other nodes with whom they have a connection. Each of the properties as well as each of the relations in a project is considered to calculate metrics. After this step, the following metrics are captures. For every metric, a violation count is considered. To keep the project well managed and keep the score good, violations must be avoided. The metrics to be considered with their brief details in SCMA tool are as follows:

1) **Line of Code by Classes (LOCC):** Total line of code without comments in a class are considered the first information to be used as a metric. More lines in a class hampers the maintaining activities and we considered it as a negative impact to our score calculation.
2) **Weighed Method Count by Classes (WMCC):** Summation of Cyclometic complexities of all methods in a class. Classes with high Weighted Method Count have less readability.
3) **Number of Methods by Classes (NOMC:** Count of methods in a class. More methods in a class increases the complexity in a class. The number of methods should be kept low.
4) **Number of Global Variables by Classes (NOGC):** Count of global variables in a class. Number of Global Variables must be kept as low as possible to make a good score by the tool.

[7]https://github.com/apple/swift-syntax

5) **Number of Couplings by Classes (NOCC):** A Coupling is considered when two classes have at least one mutual connection in any direction. This metrics represents the number of total couplings in a project. To maintain a good readability, couplings should not be presented very high between classes. Low number of couplings will help to make a good score.
6) **Number of Accessed Methods by Variables (NOAV):** Total number of accessible global variables inside a method. More usage of global variables by methods increases the cohesion and contributes to the score.
7) **Line of Code by Functions (LOCF):** Total line of code in a method body. As like LOC for classes, LOC for function also should be kept as low as possible.
8) **Cyclomatic Complexity by Functions (CCF):** The quantitative measure of the number of linearly independent paths through a program's source code. To keep good readabilities, cyclometic complexity must be small in number.
9) **Number of parameters by Functions (NOPF):** Number of total parameters in a method signature. It should be kept as low as possible.
10) **Duplicate Code (DC):** Same lines of code between two or more methods/class/blocks introduces code duplicacy. Duplicate codes are generated by copying and pasting codes and they introduce bugs while editing the codes later. In our tool we use lizard[8] to count the duplicate blocks of codes.

The overall procedure of this step is shown in Figure 2

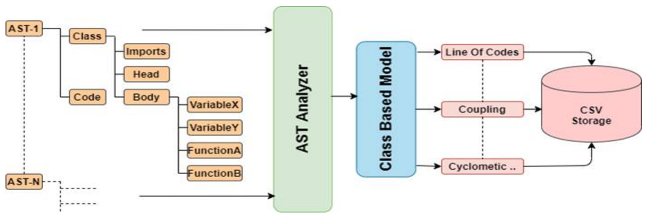


Fig. 2. Metrics Calculation

C. SCORE CALCULATION

In this section, the score calculation from metrics is described briefly. Table III-C illustrates the details of violations and formula we used for score calculation. From each of the metrics we get a score ranging from 0 to 5. Using all of these scores, a final score is calculated by averaging the values of these scores. Alike the individual scores from metrics, the final score is also remains from 0 to 5. Score close to 5 represents the project as good conditioned close to 0 means the project contains smells which needs to be refactored.

D. REPORT GENERATION

At the final step after metrics calculation, a report needs to be generated. To generate a report, an HTML page is generated through a python server. In that report, the summary of every metrics is illustrated. Users can also get the details from a CSV file for the corresponding metric. From each of the metrics (See Figure 3), a score (0 to 5) is calculated from the violation count for few metrics.

After that, an overall score is calculated using the average value from all of the metrics scores. Figure 4 illustrates an output of the overall score.

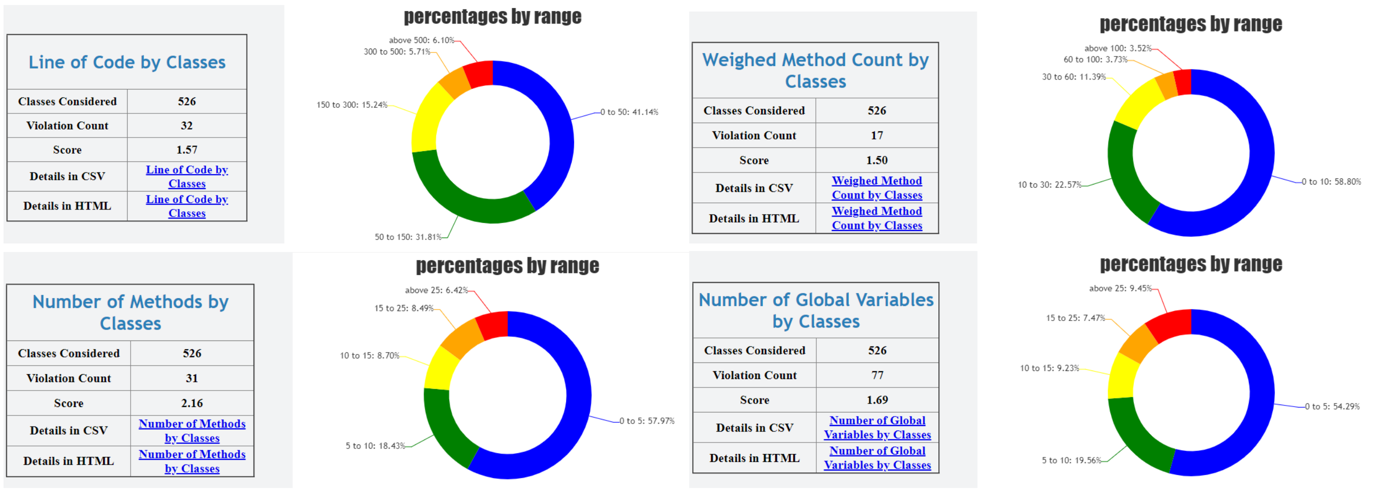
TABLE I
SCORE CALCULATION FROM METRICS

Metrics	Violation	Score
Line of Code by Classes	over 500 lines	Score = $\frac{1000}{maxLOC+1500}$
Weighed Method Count by Classes	over 200 count	R = $\frac{maxWMC*violations}{totalWMC}$; Score = $\frac{2}{R+0.4}$
Number of Methods by Classes	N/A	Score = $\frac{500}{maxmethodsinaclass+75}$
Number of Global Variables by Classes	N/A	Score = $\frac{500}{maxGlobalsinaclass+75}$
Number of Couplings by Classes	N/A	Score = $\frac{1000}{maxcoupling+150}$
Number of Accessed Methods by Variables	N/A	Score = $\frac{2}{methods/globals+0.4}$
Line of Code by Functions	N/A	Score = $\frac{2000}{maxLOC+300}$
Cyclomatic Complexity by Functions	over 20 count	R = $\frac{maxCC*violations}{totalCC}$; Score = $\frac{2}{R+0.4}$
Number of parameters by Functions	over 10 params	R = $\frac{maxParm*violations}{totalParms}$; Score = $\frac{2}{R+0.4}$
Duplicate Code	over 10 lines	R = $\frac{duplicatedlines*totallines}{totalParms}$; Score = $\frac{2}{R+0.4}$

IV. DISCUSSION

The proposed tool (SCMA) is developed considering the standard of global SW engineering tools, internal organiza-

[8]https://github.com/terryyin/lizard



Line of Code by Classes

Classes Considered	526
Violation Count	32
Score	1.57
Details in CSV	Line of Code by Classes
Details in HTML	Line of Code by Classes

Weighed Method Count by Classes

Classes Considered	526
Violation Count	17
Score	1.50
Details in CSV	Weighed Method Count by Classes
Details in HTML	Weighed Method Count by Classes

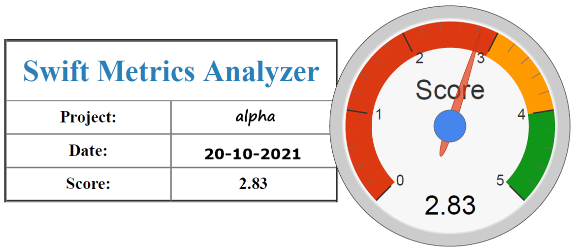
Number of Methods by Classes

Classes Considered	526
Violation Count	31
Score	2.16
Details in CSV	Number of Methods by Classes
Details in HTML	Number of Methods by Classes

Number of Global Variables by Classes

Classes Considered	526
Violation Count	77
Score	1.69
Details in CSV	Number of Global Variables by Classes
Details in HTML	Number of Global Variables by Classes

Fig. 3. Individual Metrics Score



Swift Metrics Analyzer

Project:	alpha
Date:	20-10-2021
Score:	2.83

Fig. 4. Final Score

tion's guildlines and expert opinions. There is some resemblance and dis-resemblance exit among the metrics calculated by different tools. Modular Circular Dependency (MCD) and Predefined Preprocessor Metrics are not considered in this tool as they are not compatible with swift. Table II illustrates SCMA tool specification.

TABLE II
SCMA TOOL

Key	SCMA
Language	Swift
Scoring	0-5
Build Dependency	No
Report	CSV, HTML

V. Threats to Validity

a) Internal Validity: We have built this tool based on the code metrics that are being considered within our company.

b) External Validity: For analyzing the result of our tool, we have not considered any other open-source projects as we could not trace the changes in metric scores.

c) Construct Validity: We considered the scoring formula and threshold values used as standards of global software engineering and our company. After having the scores from every sections, we calculate the overall score by averaging the individual scores of all of the metrics.

d) Reliability: We have run our tool on 11 iOS based software projects. All the applications are from real-life projects available on the app store. After the first run, we have run this tool several times after refactoring the code as per the suggestions provided by our SCMA tool.

VI. Conclusion

In this paper, we proposed a tool named SCMA to automatically score a swift project using ten software code metrics. We run this tool on swift-based software projects of our company and manually validated the result for some cases. To our knowledge, this tool provides valid output. The violation counts used here are chosen from the global code smell standards, our internal organization's guidelines and expert opinions. Currently, we are working with other metrics and we will adapt those in our future work.

References

[1] Colakoglu, F.N., Yazici, A., Mishra, A.: Software product quality metrics: A systematic mapping study. IEEE Access (2021)
[2] Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
[3] Honglei, T., Wei, S., Yanan, Z.: The research on software metrics and software complexity metrics. In: 2009 International Forum on Computer Science-Technology and Applications. vol. 1, pp. 131–136. IEEE (2009)
[4] Hossain, S.S., Ahmed, P., Arafat, Y.: Software process metrics in agile software development: A systematic mapping study. In: International Conference on Computational Science and Its Applications. pp. 15–26. Springer (2021)
[5] Kitchenham, B.: What's up with software metrics?–a preliminary mapping study. Journal of systems and software **83**(1), 37–51 (2010)
[6] Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. Journal of Systems and Software **101**, 193–220 (2015)
[7] Martin, R.C.: Clean architecture: a craftsman's guide to software structure and design. Prentice Hall (2018)
[8] Nuñez-Varela, A.S., Pérez-Gonzalez, H.G., Martínez-Perez, F.E., Soubervielle-Montalvo, C.: Source code metrics: A systematic mapping study. Journal of Systems and Software **128**, 164–197 (2017)
[9] Tahir, A., MacDonell, S.G.: A systematic mapping study on dynamic metrics and software quality. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM). pp. 326–335. IEEE (2012)