2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)
Banff Center, Banff, Canada, October 5-8, 2017

Compiling Static Software Metrics for Reliability and Maintainability from GitHub Repositories

Jeremy Ludwig, Steven Xu
Stottler Henke Associates, Inc.
San Mateo, CA

Frederick Webber
Air Force Research Laboratory
711th HPW/RHAS
WPAFB, OH

Abstract— **This paper identifies a small, essential set of static software code metrics linked to the software product quality characteristics of reliability and maintainability and to the most commonly identified sources of technical debt. A plug-in is created for the Understand code visualization and static analysis tool that calculates and aggregates the metrics. The plug-in produces a high-level interactive html report as well as developer-level information needed to address quality issues using Understand. A script makes use of Git, Understand, and the plug-in to compile results for a list of GitHub repositories into a single file. The primary contribution of this work is to describe an open-source plug-in to measure and visualize architectural complexity based on the propagation cost and core size metrics, which are not currently found in other tools. The plug-in should be useful to researchers and practitioners interested in these two metrics and as an expedient starting point to experimentation with metric collection and aggregation for groups of GitHub repositories. The plug-in was developed as a first step in an ongoing project aimed at applying case-based reasoning to the issue of software product quality.**

Keywords— software product quality, technical debt, reliability, maintainability, architecture, metrics, static code analysis

I. INTRODUCTION

There is a consistent push to improve software quality, especially for components that are designed to be heavily re-used and extended, e.g., as part of a software product line. Software quality models are a way of articulating what is meant by 'software quality.' These models define the desired characteristics and sub-characteristics of software and the relationship between these characteristics and measurable properties of the software [1]. The ISO-IEC 25010: 2011 [2] quality model defines eight desired characteristics of software product quality. An objective of this paper is to identify a small, essential set of static software code metrics linked to these product quality characteristics. While all the characteristics are important, the paper focuses specifically on Reliability and Maintainability, as the bulk of existing research linking software quality to static code analysis uses these characteristics [3].

Technical debt is a measure of how much work would be needed to move from the current code to higher-quality code [4]. The source of technical debt during development and sustainment stems primarily from making design, implementation, documentation, and testing decisions that are focused on short-term value [5]. As technical debt increases, changes to the software become more difficult, error-prone, and time-consuming, and this threatens the reliability and maintainability characteristics of the software.

This is an especially important take-away for software product lines, where long-lived, reusable modules are intended to be shared by multiple systems. Each module will want to invest in high software quality (low technical debt) initially and maintain this investment in quality over time as it is extended and updated. That is, as part of planned re-usability, each module commits to making a long-term investment to software quality. The likely alternative is that the software quality will gradually degrade, until the problems become overwhelming [6].

There are several practical software quality models and tools that have been recently developed and that generally include an automated measurement of technical debt [7]. However, by default, the measured technical debt may not be an accurate measure of product quality issues [8]. While some technical debt is unavoidable [9], a large survey of software engineers and architects across multiple organizations provides a practical view of the causes and sources of avoidable technical debt [5]. Their results indicate that architectural decisions, overly complex code, and lack of code documentation are the top three avoidable sources of technical debt in practice.

In the remainder of this paper, the Related Work section briefly covers existing tools in this area. Following this, the Methods section describes the work performed on metric identification, metric calculation, and compilation of results for GitHub repositories. The Results section illustrates use cases for managers, developers, and GitHub compilation. Finally, the Discussion and Conclusion sections evaluate and summarize the results.

A. Related Work

Related software quality and automated code review tools are based on a software quality model that identifies specific source code metrics, and describes how the measurements of these metrics are aggregated, and how the aggregations are used to assess characteristics of software quality and technical debt [10]. Some examples of state-of-the-art models (and tools) discussed are Software Quality Enhancement (SQUALE), Quamoco Benchmark for Software Quality, Columbus Quality Model, Software Improvement Group

(SIG) Maintainability Model, and Software Quality Assessment based on Lifecycle Expectations (SQALE). Codacy is another example of an automated code review and analysis tool. As an open-source project, SQUALE provides a veritable treasure trove of information on understanding and developing a software quality model. SQALE [11] differs from the others in that it is an open methodology that defines the software quality and technical debt model and is implemented by tools such as SonarQube.

Contribution: The primary contribution of this work is to describe an open-source plug-in to measure and visualize architectural complexity metrics not currently found in other tools. The work reported in this paper also differs from this existing work in that it begins to develop a quality model that focuses on architectural complexity and relies on only a small set of essential software metrics that address the primary sources of technical debt.

II. METHODS

The Methods section includes metric identification and calculation, metric aggregation, and GitHub compilation.

A. Metric Identification and Calculation

This section discusses three types of static source code metrics related to software quality characteristics and how they are calculated. The first measures architectural complexity, the second code complexity and coupling, and the third, code commenting. The Understand code visualization and static code analysis tool, developed by Scientific Toolworks, is used to perform all metric measurement and calculation.

1) Architectural Metrics

The two selected visibility metrics define architectural complexity based on how reachable one file in the code base is from all the other files. These two metrics are measured using the algorithm defined in [12]. The code for performing these measures is included in the open source plug-in. While there are numerous measures of software architecture [13] – [15], these two were specified by the research agenda.

The first architectural metric is **propagation cost**. This is a system-wide metric that describes the proportion of software files that are directly or indirectly linked to each other. If component A makes a call to B, A and B are directly linked. If file B makes a call to C, then B and C are directly linked and A and C are indirectly linked. Given this configuration, if C is changed, B might need to be changed as well. If B is changed, then A might need to be changed. So, a change in C might require both B (direct link) and A (indirect link) to be changed. The propagation cost provides a single, system-wide measure of how linked the code is.

The second architectural metric is the **core size**. This metric involves classifying every component (class or file) into one of five architecture groups based on the number of direct and indirect links of the components: core, shared, control, peripheral, and isolate. The algorithm also divides the overall architecture into one of four types: core-periphery, borderline core-periphery, multi-core, and hierarchical. The core group represents the largest cyclic group, i.e., the largest set of components that are interdependently linked to each other.

The utility of the propagation cost and core size measures of architectural complexity has been demonstrated in a number of studies [16] – [18]. These measures have been shown to relate significantly to defect density, programmer productivity, and programmer retention. Core files have been found to contain more defects and cost more to maintain [17].

2) Complexity Metrics

A small set of complexity, size, and coupling metrics were selected based on evidence of supporting correlation with, or prediction of, the software characteristics of reliability and maintainability. Given the vast available literature on software metrics and an equally large variety of metrics, a reasonable starting point is systematic literature reviews. [19] reviews 99 primary studies and compares their work to several other surveys and systematic literature reviews (e.g., [20] [21]). The results indicate that the link from metric to reliability and maintainability across studies is strongest for: LOC, WMC-Unweighted/WMC-McCabe, RFC, and CBO. A high level definition of each of these metrics from [1] is:

- LOC (Lines of Code): The number of lines containing source code, including inactive regions, in a component. LOC is a measure of size/volume.
- WMC Unweighted (Weighted Method Count - Unweighted): A simple count of the number of methods implemented in a component. WMC is a measure of complexity.
- WMC-McCabe (Weighted Method Count – McCabe): Like WMC-Unweighted, except that each method implemented in the component is weighted by its McCabe Cyclomatic complexity value. Cyclomatic complexity is the number of independent paths through a method. WMC is a measure of complexity.
- RFC (Response for Class**):** The sum of the number of methods in a component plus the sum of all the methods it directly references in other components. RFC is a measure of complexity.
- CBO (Coupling Between Objects**):** This is a measure of how many other components are relied on by a given component. CBO is a measure of coupling.

We define each of these metrics based on existing calculations performed by Understand. The plug-in relies on these existing calculations, concretely defined in [22].

3) Comment Metrics

The Code-To-Comment ratio is used as an initial measure of code commenting. This metric has been well studied as part of earlier work on quality models [23]. Anecdotally, it is also one of the metrics most-used by developers utilizing the Understand software. The plug-in relies on existing functionality within Understand to calculate this metric.

B. Metric Aggregation Model

Propagation cost and core size metrics generate a project-wide measurement. For all other metrics, an aggregation model is needed to generate a project-level description from the component-level description. In this initial work, two different types of aggregation were used. The first was to report median values, e.g., median LOC per component. The second was to report the number of components that exceeded a threshold, e.g., number of components that contained more than 200 LOC. Default thresholds were taken from [24], [25] and from thresholds previously defined within Understand. There are numerous difficulties with threshold values; the plug-in supports users easily changing to different values. Counts are also included in the report (e.g., Number of Classes), to convert from absolute number of thresholds exceed to percentage of files exceeding a threshold. This aggregation model represents a first step in ongoing work.

C. GitHub Compilation

Software quality has also been studied specifically in the context of open-source code [26]. Open-source repositories, such as that available on GitHub, include information on the software product, the development process, and their popularity. The available information includes the software code, the associated version control system, the list of issues and their status, and the list of contributions, as well as the number of stars and forks [27]. All this information is available through the GitHub API.

Taking advantage of this, Python and Bash scripts running on a Linux box were created to automatically pull and process data from a list of specified GitHub projects using the API. The first script clones each repository into a local directory and runs the Understand metrics on it, while also querying relevant data fields (e.g., stars/watches/forks, languages used, number of issues) using the GitHub API, storing the results in a JavaScript Object Notation (JSON) file. A second script extracts the essential fields from both the JSON files and generated metrics of each project and compiles them into a single combined Comma Separated Values (CSV) file. This allows the results for groups of repositories to be easily viewed and manipulated in a spreadsheet.

III. Results

The Results section describes the results of the plug-in for project management, developer, and GitHub use cases.

A. Project Management

From a management perspective, the goal is to use the plug-in to calculate the current metric values. Once the plug-in is installed, the user opens a software code base in Understand and then selects a "Core Metrics" report menu item. At this point the user can elect to change the default thresholds. An interactive HTML report is generated as shown in Figure 1. The report includes all the generated metrics on the left and the various architecture groupings in a Design Structure Matrix graph in the center, as described in [12]. The plug-in was tested on a variety of open-source and proprietary software in Java, C++, C, and Web languages.

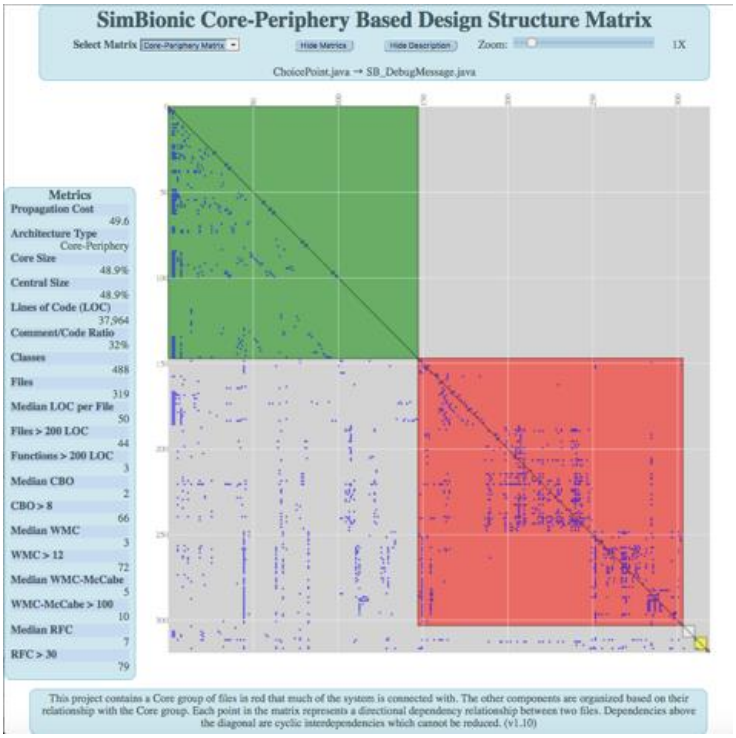

Figure 1. Interactive report for a single GitHub Project.

name	arrayfire	MITIE	WorldWind Java
stars	1570	1191	110
languages	C++	C++	Java
Architecture Type	Core-Periphery	Core-Periphery	Core-Periphery
Core Size	9.80%	9.70%	19.80%
Central Size	77.20%	19.20%	15.40%
Lines of Code (LOC)	91204	253591	337041
Comment/Code Ratio	26%	25%	34%

Figure 2. Partial results compiled from three GitHub repositories.

B. Software Development

For the software developer example, the goal is to reduce technical debt. Understand already supports this goal for the complexity and comment metrics. The plug-in takes advantage of existing Understand functionality to address architectural complexity as well. This is accomplished by importing the architecture partitions, to support tracking down and refactoring complexity and cyclic dependencies as shown in Figure 3 and Figure 4.

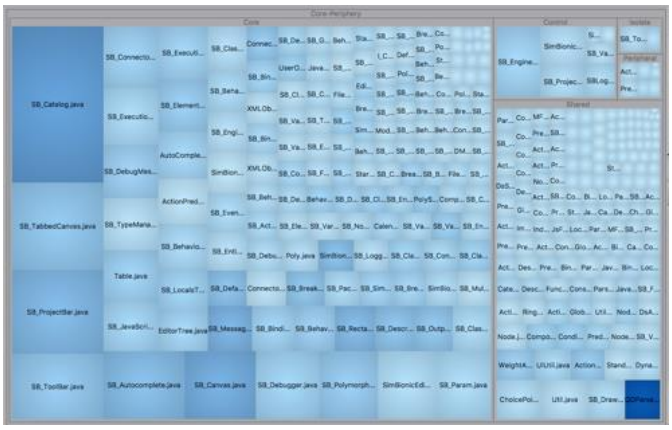

Figure 3. A tree map in Understand where the map is divided by architecture partition, the size of each component is determined by LOC, and color indicates the amount of WMC-McCabe complexity (darker blue is more complex).

C. GitHub Results

The GitHub compilation scripts were tested against 8 C++ and 8 Java projects from GitHub, choosing projects with high numbers of stars that spanned different application areas. When given the list of projects, the system was able to automatically download, analyze, and collect meta-information on each project. These projects represent a total of 5M+ lines of code, analyzed in about 40 minutes. The results are presented in CSV format as shown in Figure 2. Complete results are available on github.com/StottlerHenke.

IV. Discussion

The methods section focused on identifying metrics that are linked to software product reliability and maintainability and also to the most relevant types of technical debt facing developers. The results demonstrate that the developed plug-in is able to calculate the identified metrics and how this information can be used within Understand to help address technical debt. The plug-in is freely available at: scitools.com/support/gui-plugins/. A licensed copy of Understand is needed to use the plug-in.

The results also demonstrate the ability to write scripts that use the plug-in to download the code and meta-information for a repository from GitHub, analyze the code, and then compile the results. The generated report includes context information such as the description, stars, contributors, and issues, in addition to the metrics generated from the source code. Additionally, the script serves as a template to use the system in other ways, for example measuring and compiling results as part of an automated build process. The compilation scripts are available at: github.com/StottlerHenke.

There are limitations in the current plug-in. First, the object-oriented measures (CBO, WMC, WMC-McCabe, and RFC) are not currently calculated for C files. Second, the plug-in has been tested on code only up to 4.5M LOC, which took over an hour to analyze.

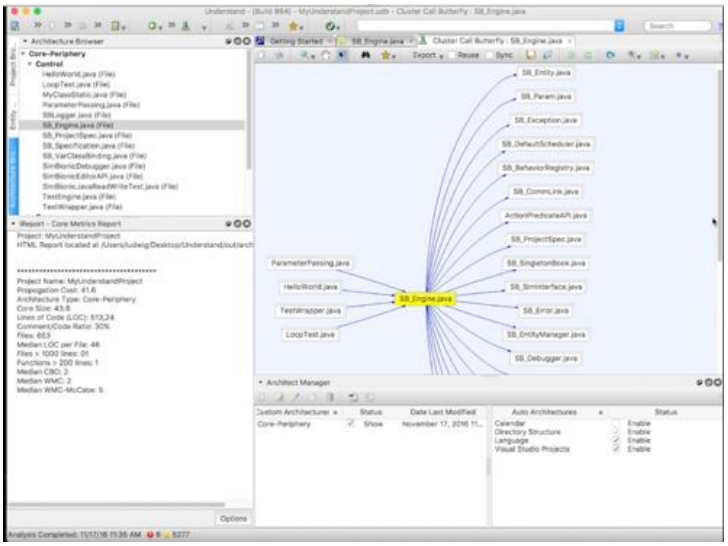

Figure 4. Viewing the Core-Periphery architecture partition in Understand.

V. Conclusion

Software code quality and technical debt have significant impact on a software product's reliability and maintainability. This paper identifies a small, essential, set of static software code metrics linked to reliability and maintainability and to the most commonly identified sources of technical debt. A plug-in is created for the Understand code visualization and static analysis tool that calculates and aggregates the metrics and produces a high-level interactive html report as well as developer-level information needed to address quality issues. A script makes use of Git, Understand, and the plug-in to compile results for lists of GitHub repositories into a single file.

While the plug-in is useful as-is, it was developed as a first step in an ongoing project aimed at applying case-based reasoning to the issue of software product quality. The next step in this project aims to use the described plug-in as part of a research effort to define and validate the aggregation of these metrics as part of a software quality model.

Acknowledgment

This material is based upon work supported by the United States Air Force Research Laboratory under Contract No. FA8650-16-M-6732. The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the AFRL.

DISTRIBUTION A. Approved for public release: distribution unlimited (case 88ABW-2017-2167).

References

[1] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, 3rd ed. Boca Raton, FL, USA: CRC Press, Inc., 2014.

[2] Organización Internacional de Normalización, *ISO-IEC 25010: 2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Geneva: ISO, 2011.

[3] R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," *Empir. Softw. Eng.*, vol. 20, no. 3, pp. 640–693, Jun. 2015.

[4] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In Search of a Metric for Managing Architectural Technical Debt," in *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, Washington, DC, USA, 2012, pp. 91–100.

[5] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2015, pp. 50–60.

[6] E. Lim, N. Taksande, and C. Seaman, "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Softw.*, vol. 29, no. 6, pp. 22–27, Nov. 2012.

[7] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *IEEE Softw.*, vol. 29, no. 6, pp. 34–42, Nov. 2012.

[8] I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, and B. Williams, "The Correspondence Between Software Quality Models and Technical Debt Estimation Approaches," in *Proceedings of the 2014 Sixth International Workshop on Managing Technical Debt*, Washington, DC, USA, 2014, pp. 19–26.

[9] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, Nov. 2012.

[10] R. Ferenc, P. Hegedűs, and T. Gyimóthy, "Software Product Quality Models," in *Evolving Software Systems*, T. Mens, A. Serebrenik, and A. Cleve, Eds. Springer Berlin Heidelberg, 2014, pp. 65–100.

[11] J.-L. Letouzey, "The SQALE Method for Managing Technical Debt Definition Document," 31-Mar-2016. [Online]. Available: http://www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf. [Accessed: 04-Feb-2017].

[12] C. Baldwin, A. MacCormack, and J. Rusnak, "Hidden Structure: Using Network Methods to Map System Architecture," 2014.

[13] R. Kazman *et al.*, "A Case Study in Locating the Architectural Roots of Technical Debt," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, Piscataway, NJ, USA, 2015, pp. 179–188.

[14] S. Stevanetic and U. Zdun, "Software Metrics for Measuring the Understandability of Architectural Structures: A Systematic Mapping Study," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, New York, NY, USA, 2015, p. 21:1–21:14.

[15] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and Quantifying Architectural Debt," in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA, 2016, pp. 488–498.

[16] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Manag. Sci.*, vol. 52, no. 7, pp. 1015–1030, Jul. 2006.

[17] A. MacCormack and D. J. Sturtevant, "Technical debt and system architecture: The impact of coupling on defect-related activity," *J. Syst. Softw.*, vol. 120, pp. 170–182, Oct. 2016.

[18] D. J. Sturtevant, "System design and the cost of architectural complexity," Thesis, Massachusetts Institute of Technology, 2013.

[19] R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," *Empir. Softw. Eng.*, vol. 20, no. 3, pp. 640–693, Mar. 2014.

[20] M. Riaz, E. Mendes, and E. Tempero, "A Systematic Review of Software Maintainability Prediction and Metrics," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, Washington, DC, USA, 2009, pp. 367–377.

[21] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, Aug. 2013.

[22] "Metrics | SciTools.com." [Online]. Available: https://scitools.com/feature/metrics/. [Accessed: 07-Mar-2017].

[23] D. Coleman, B. Lowther, and P. Oman, "The application of software maintainability models in industrial software systems," *J. Syst. Softw.*, vol. 29, no. 1, pp. 3–16, Apr. 1995.

[24] Ö. F. Arar and K. Ayan, "Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies," *Expert Syst. Appl.*, vol. 61, pp. 106–121, Nov. 2016.

[25] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *Empir. Softw. Eng.*, vol. 16, no. 6, pp. 812–841, Dec. 2011.

[26] D. Spinellis *et al.*, "Evaluating the Quality of Open Source Software," *Electron. Notes Theor. Comput. Sci.*, vol. 233, pp. 5–28, Mar. 2009.

[27] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, and A. Wierzbicki, "GitHub Projects. Quality Analysis of Open-Source Software," in *Social Informatics*, 2014, pp. 80–94.