Challenges in Explaining Source Code Quality Assessment

Jeremy Ludwig & Devin Cline
Stottler Henke Associates, Inc.
San Mateo, CA 94402
ludwig, dcline @ stottlerhenke.com

Abstract— **Creating and maintaining high-quality source code is especially important for critical systems such as those made for aerospace and military domains as well as for software product lines where long-lived, reusable modules are intended to be shared by multiple systems. CBR-Insight is an open-source automated assessment tool that relies on data science and interface design to provide an objective and understandable measure of source code quality. CBR-Insight supports the ability of technical and non-technical decision makers to verify that a project's software implementation follows through on promises around developing and sustaining reliable and maintainable software while managing technical debt. The primary contribution of this work is an in-depth discussion on how source code quality assessment is communicated to the entire development team in CBR-Insight, spanning from non-technical decision makers overseeing the project to the software developers responsible for source code. Specific issues discussed include understandability of the explanations provided by the user interface along with transparency and trust in the underlying, technically complex, calculations and scoring algorithms.**

TABLE OF CONTENTS

1. INTRODUCTION

Creating and maintaining high-quality software is especially important for critical systems such as those made for the aerospace and defense as well as for software product lines where long-lived, reusable modules are intended to be shared by multiple systems. A vital component of software development is creating high-quality source code that strengthens the reliability and maintainability of the software and has limited technical debt. High-quality source code is an investment that saves time and money in the long run: users will find fewer bugs, the bugs will be easier to fix, and new features will be easier to add.

Software development teams employ a variety of design techniques, processes, and tools to continually work towards quality code while balancing the overall time and budget demands of the project. CBR-Insight (CBRI) is an open-source automated code assessment tool that relies on data science and design to provide an objective and understandable measure of source code quality that can help guide decisions and direct limited resources during software acquisition, development, and sustainment. CBRI supports the ability of technical and non-technical decision makers to verify that a project's software implementation follows through on promises around developing and sustaining reliable and maintainable software while managing technical debt.

There is a long history of software engineering research in the area of source code quality, and numerous existing tools aim at performing automated code quality assessment and review. What makes CBRI a complementary addition to existing tools is: (i) the calculation of a small, essential set of static code metrics associated with software maintainability, reliability, and source code technical debt, (ii) using a customizable set of peer projects to provide the context needed to understand the metric results, and (iii) presenting the information in a format preferred by decision makers. This paper begins with an overview of related work and the CBRI source code assessment tool. Following this is an in-depth discussion on how source code quality assessment is communicated to the entire development team in CBRI, spanning from non-technical decision makers overseeing the project to the software developers responsible for source code. The paper concludes with a description of ongoing work on CBRI and links to the source code on GitHub.

2. RELATED WORK

There is a consistent push to improve software quality for critical systems and software product lines. The related work spans several areas, including software quality models, technical debt, and automated code review tools.

978-1-6654-3760-8/22/$31.00 ©2022 IEEE

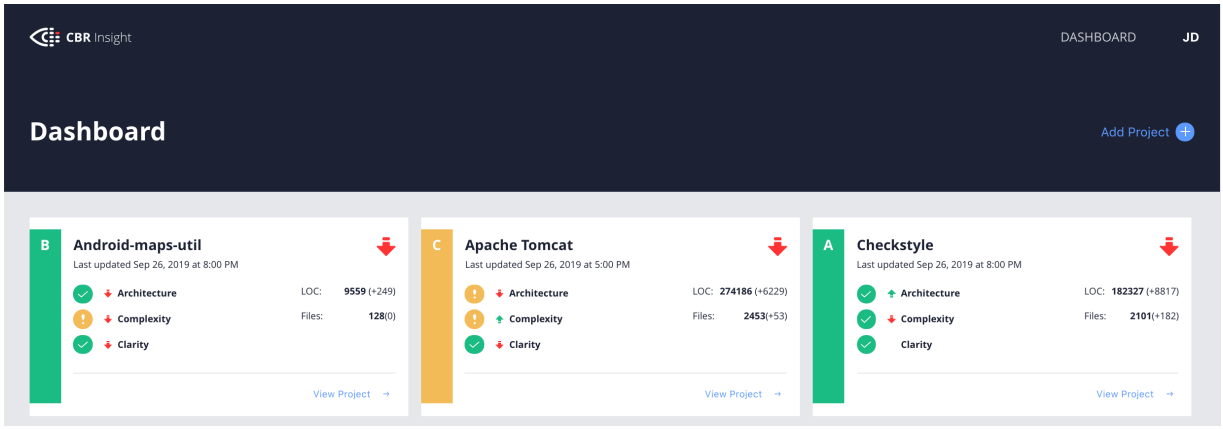

Figure 1. CBR-Insight Dashboard with three example projects

Software quality models articulate 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: Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, and Portability. While all of these characteristics are important, this paper focuses specifically on reliability and maintainability as the bulk of existing research linking software quality to static code analysis uses these characteristics [3]. Reliability and maintainability play a supporting role for other characteristics such as usability and security.

Software quality models based on static source code analysis generally follow a three-step pattern. They identify specific source code metrics to be calculated, describe how the measurements of these metrics are aggregated, and define how the aggregations are used to assess characteristics of software quality (e.g. Reliability) [4]. Some examples of models and tools 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). As an open-source project, SQUALE [5] provides a veritable treasure trove of information on understanding and developing a software quality model and visualizing the results. SQALE [6] 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. CBRI builds on all of this prior work in creating its underlying software quality model.

Technical debt is a measure of how much work would be needed to move from the current code to higher-quality code [7]. The source of technical debt during development and sustainment stems primarily from making design, implementation, documentation, and testing decisions that focus on short-term value [8]. As technical debt increases, changes to the software become more difficult, error-prone, and time-consuming, and this threatens the reliability, maintainability, and security software characteristics.

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, eventually, the problems become overwhelming [9].

While some technical debt is unavoidable [10], a large survey of software engineers and architects across multiple organizations provides a practical view of the causes and sources of avoidable technical debt [8]. 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. CBRI focuses on these three areas of source code technical debt in order to support software reliability and maintainability.

There are several practical tools aimed at improving source code quality and reducing technical debt. Of these SonarQube [11] appears to be the most widely used [12]. This and other similar automated code review tools use rules to analyze every line of code to identify likely bugs, maintainability issues, and security flaws— encouraging developers to correct these issues with each code commit.

Automated code review provides an invaluable service, assisting developers in catching these issues early. However, automated code review based strictly on rule violations may not present a complete picture of overall code quality [13]. Additionally the (generally long) list of violations generated for existing systems can be overwhelming for developers and is not necessarily helpful in providing a high-level view of the health of the code base [14]. CBRI aims to complement automated code review systems by highlighting overall

software quality trends in the areas of architecture, complexity, and clarity as well as by providing the context required to interpret and utilize the results. A detailed look at CBRI, including metrics, methods, and a real-world DevOps use case, can be found in [15].

3. CBR-INSIGHT

The CBRI web application is composed of two main user interface components: Dashboard and Project View. The Dashboard in Figure 1 provides an at-a-glance summary across a number of projects, while the Project View in Figure 2 enables a deep look into a single project. Behind the scenes, CBRI uses the Understand static source code analysis tool developed by SciTools [16] to generate the data displayed in the web app. The guiding design principle of the user interface is to include accessible explanations whenever possible.

The Dashboard focuses on highlighting software code quality across multiple projects in three important areas: architecture, complexity, and clarity. Intuitive symbols and colors indicate the relative score, from red/alert to green/check. An overall letter grade (A – F) is also assigned, each with a corresponding color. Trending icons indicate how the area and overall scores have changed relative to a baseline measurement.

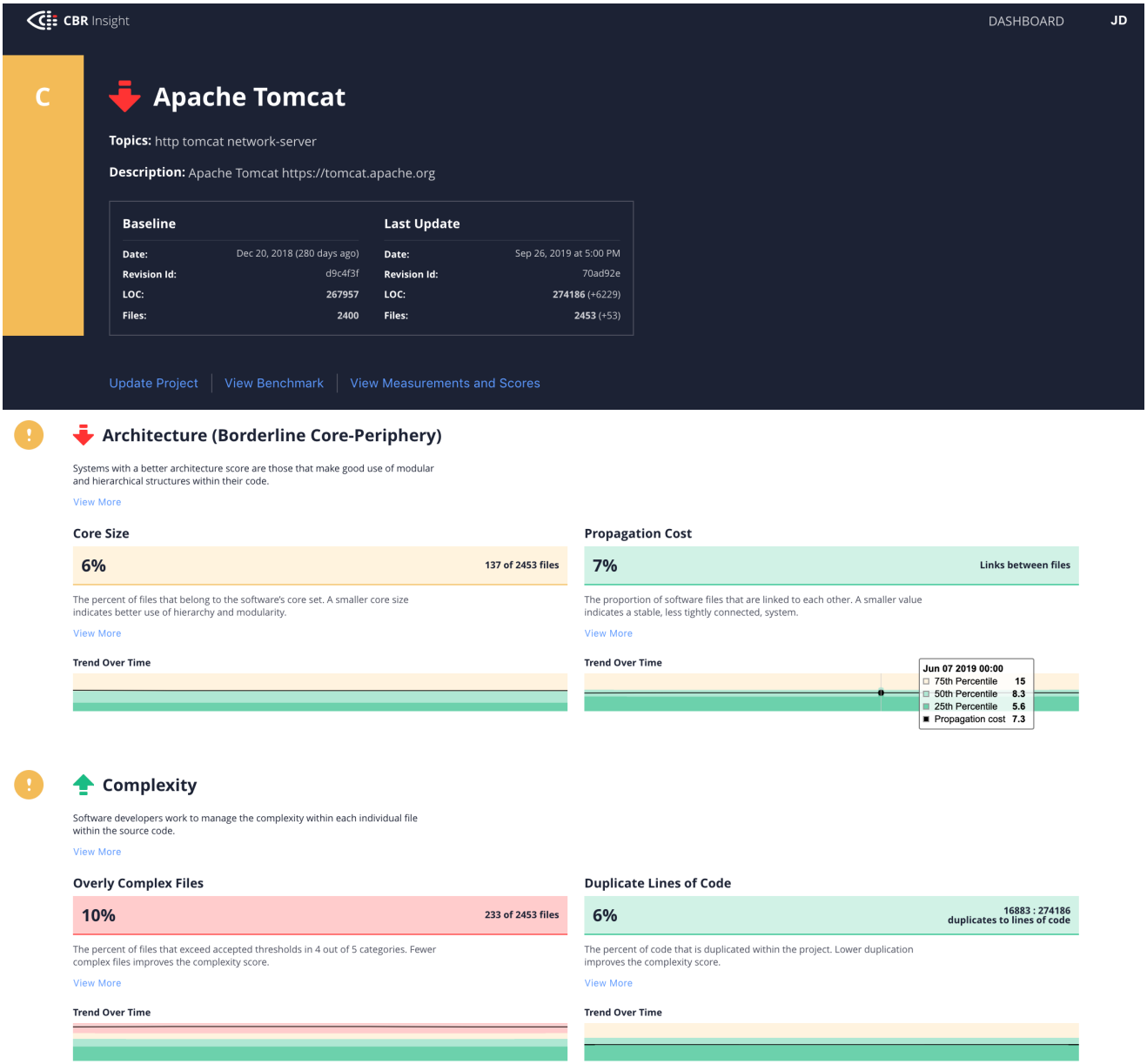

Figure 2. Project View general information, architecture, and complexity sections

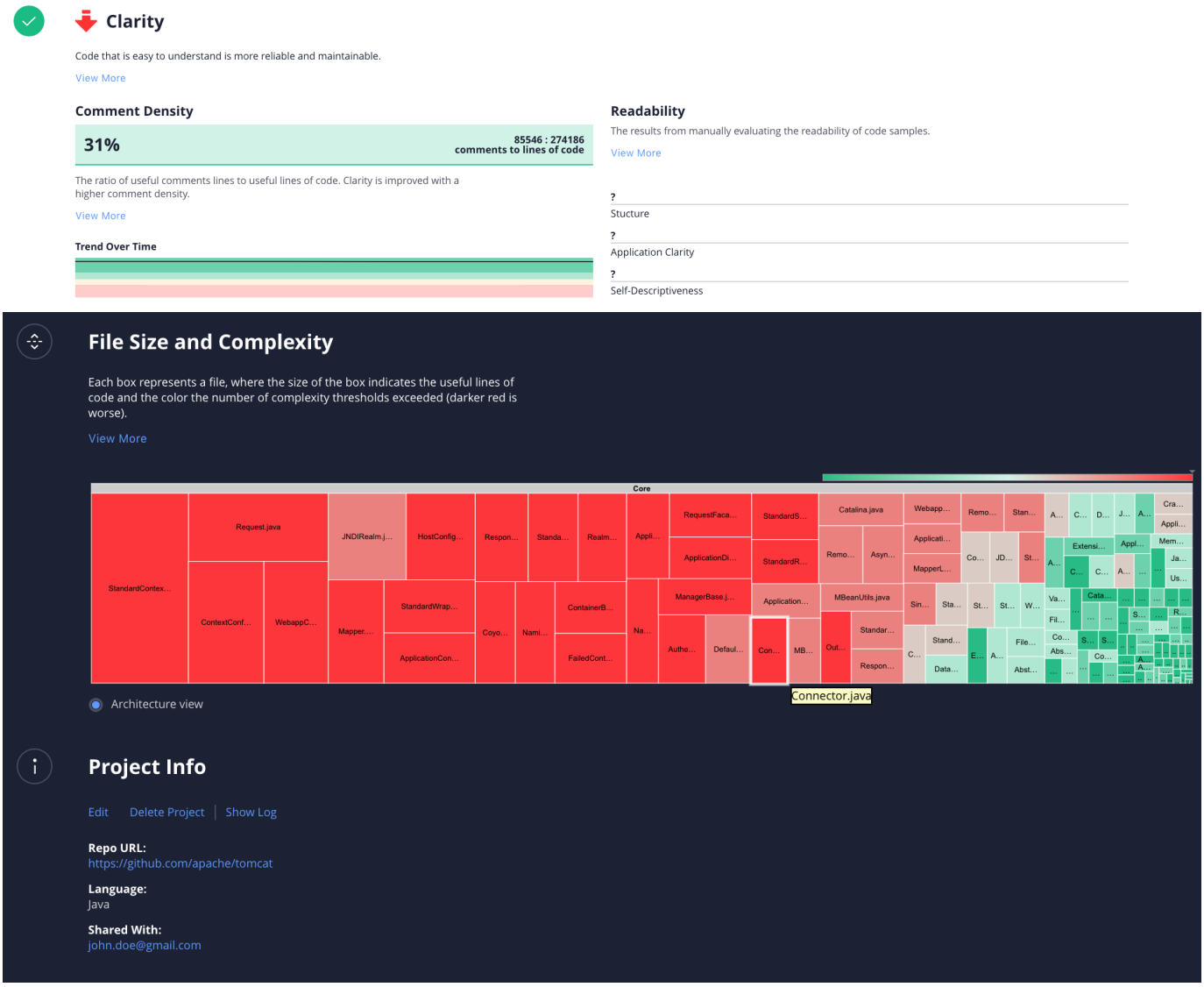

Figure 3. Project view clarity, treemap, and project info sections

The Project View provides a description of the underlying metrics used to generate the scores for the project and visualizes the calculations over time. The visualizations include color-coded target ranges determined by analyzing peer projects along with a tree-map of file size and complexity organized by the Core Size architecture set. Every section contains descriptive text to assist the user in understanding the scores and measurements.

The top portion of the Project View in Figure 2 provides general information such as topics (e.g., machine-learning) and a brief description. Following this are the date, revision, lines of code (LOC), and number of files in the baseline and latest measurement. Links enable the operator to update the project and to view additional details on the benchmarks and measurements.

The next three sections of the Project View are the architecture, complexity, and clarity (continued in Figure 3) measurements. Each section includes a description of the measurement and a graphical representation of the measurement over time, relative to peers. Hovering over a graph brings up a popup that shows the calculated metric value compared to the 25th, 50th, and 75th percentile values from the selected peer projects. Each section also includes visual indicators of change (good, bad, none), relative to the project baseline. For this very mature project, the trend lines have remained consistent over time; the visual indicators provide insight into the slowly creeping changes to code quality.

The following section of the Project View in Figure 3 contains a treemap, where each box is a file, the size of the box indicates the lines of code, and the color the number of complexity thresholds exceeded (darker red is worse). The files are organized by their determined core size architecture grouping. Click on architecture group to see the files with the group; right click to navigate back to the architectures.

Addressing complexity issues in the core group will likely have the greatest impact. The final section of the Project View supports the viewing and editing of detailed information such as the repository location, the analysis language, and the users with whom this repository is shared. A log is also included to troubleshoot repository connection issues.

4. EXPLANATIONS

There are four different types of results that CBRI attempts to explain to the end user: metrics, metric calculations, peer projects, and generated scores.

Metrics

The first challenge is explaining the five metrics that are calculated by performing a static analysis of the source code. This is primarily done with textual explanations for each metric embedded in the UI. For example, the short (two sentence) explanation for propagation cost is seen in Figure 2 under the percentage value. The *View More* link of this explanation expands to show a longer definition as seen in Figure 5.

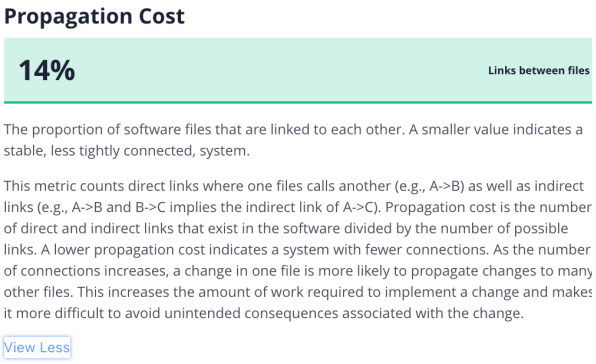

Figure 5. Expanded explanation for propagation

Note that these explanations do not contain all of the details that would be required to replicate the measurement. This was the result of a design decision to streamline the interface rather than provide complex algorithm descriptions and examples that most users would have little reason or desire to read. This design decision is a point of debate – while most users find these explanations sufficient there are times when more details are requested.

The second metrics-related challenge is explaining why these metrics were selected from hundreds of available static source code metrics [1]. Metric selection for CBRI is based on [17], which identifies architecture, complexity, and comment metrics related to software reliability and maintainability. The user interface builds on this, and the notion of preventable technical debt described in [8], to highlight architecture, complexity, and clarity in both the Dashboard and Project View. Ultimately while there is prior research [17] and quantitative results [15] that provide evidence for the selected metrics, users familiar with static code analysis tend to approach the selected metrics as a starting set that they would like customize with additional metrics.

Metric Calculations

The are three challenges associated with a project's calculated metrics results. First, users have difficulty with interpreting calculated values other than that more (or sometimes less) is bad. For example, if 2% of a project's files are over complex is that good or bad? CBRI uses values calculated from similar peer projects [15] to develop the interquartile range (IQR) in which to interpret the results as shown in Figure 4. Graphing the calculations for a project over time, against the backdrop of peer-generated IQR color bands, provides an intuitive explanation of the metric calculation result and direction.

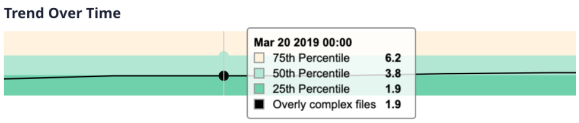

Figure 4. Overly complex files over time

Second, users familiar with static code analysis tend to want to see the calculated results for their projects within the web application (in addition to within the Understand integrated development environment as currently supported). For overly complex files, this information is mostly contained in the treemap. Results of other metrics such as duplicate lines of code and comment density could usefully be displayed in a table. However, the architecture metrics are problematic, involving large matrices showing connections between files. For example, even a relatively modest design structure matrix used to determine the core size [18] as shown in Figure 6 is difficult to make use of within the web application. Regardless, to ensure transparency we plan to include detailed results for all metrics as part of future work.

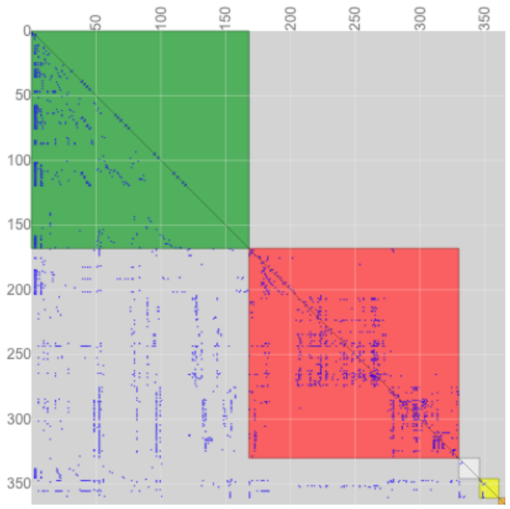

Figure 6. Design structure matrix for 350+ files

Third, users frequently ask to see which files were included in the analysis results, to verify that generated and test source code is not included. Explicit inclusion and exclusion of source directories is planned as part of future work.

Peer Projects

Peer projects play a large role in CBRI, where they are used to determine the IQR bands against which each metric is graphed and in the scoring algorithm. Peer projects are selected from a vetted library drawn from the 1,000 most popular Java, C, C++, and C# repositories on GitHub. The custom set of selected peer projects, along with all of their lines of code, topics, metric calculations, etc. are available in a table by following the *View Benchmark* link at the top of the Project View. What is missing from the user interface is any explanation of how the peers were chosen from the library. The lack of selection criteria hurts overall system transparency and trust and would be relatively easy to address with a text explanation.

Scoring Algorithm

CBRI calculates four aggregated scores – one for the overall score and one in each area of architecture, complexity, clarity. The scores are generated directly from the calculated metric percentile scores. For example, if a project's propagation cost score is in the 95th percentile relative to its peers a score of 0.95 is assigned for propagation cost. The area scores are the sum of the metric scores in that area; the overall score is the sum of the area scores. Higher scores indicate better performance on the underlying metrics relative to peers. These scores are used to generate the trending arrows as well as letter grades.

Letter grades are presented in the user interface as colors and symbols and are based on the percentile of the score relative to the scores of all of the projects in the library of the same language. To be precise, grades are assigned by looking at the distribution of numeric scores across all projects in the library of the same language (not just peers) and then assigning letter grades based on this distribution with the following percentile cutoffs: A (0.9), B (0.7), C (0.3), D (0.1), and F (< 0.1). For example, all of the projects with an "A" will be those that scored the best against their peers; a "C" represents an average score. This is different from letter grade assignment schemes used by some other code quality tools (e.g., [19]) that awards A and B grades to a majority of projects. There is no explicit explanation of the scoring mechanism to help users understand the grade results. For example, users commonly interpret a "C" as a poor score rather than as average with respect to other production-level projects. This explanation gap is addressed as part of future work.

5. Conclusion

Software code quality has a significant impact on a software product's reliability, maintainability, and security. This paper describes CBR-Insight, an open-source web application designed to measure and visualize source code quality. CBRI provides an objective and understandable measure of software quality that can help guide decisions and direct limited resources during software acquisition, development, and sustainment. One specific challenge that CBRI attempts to address is how source code quality assessment is communicated to the entire development team, with the idea that the assessment needs to be valuable to everyone involved in performing and overseeing software development.

There is a long history of software engineering research in the area of software product quality, and numerous existing tools aim at performing automated code quality assessment. What makes CBRI a complementary addition to existing tools is: (i) the calculation of a small, essential set of metrics associated with maintainability, reliability, and technical debt, (ii) using peer projects to set the targets associated with each metric, and (iii) presenting the information in a format preferred by decision makers. CBRI components are available at: https://github.com/StottlerHenkeAssociates.

Future efforts on CBRI related to explanations includes the issues highlighted in this paper. First, while CBRI strives to provide a high-level view, there is a need to add more details on the metric algorithms and the raw data from calculations. Second, the user needs a well-defined way to define the directories to be included and excluded from the source analysis. Third, clear explanations for peer project selection and the scoring mechanism should be included. All of these efforts will further the usability of the system and trust in the assessment results.

Acknowledgements

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.

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 Software Eng*, vol. 20, no. 3, pp. 640–693, Jun. 2015, doi: 10.1007/s10664-013-9291-7.

[4] 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. doi: 10.1007/978-3-642-45398-4_3.
[5] Squale Consortium, "Visualization of Practices and Metrics," Mar. 2010. Accessed: Feb. 07, 2017. [Online]. Available: http://www.squale.org/quality-models-site/research-deliverables/WP1.2_Visualization-of-Practices-and-Metrics_v1.1.pdf
[6] J.-L. Letouzey, "The SQALE Method for Managing Technical Debt Definition Document," Mar. 31, 2016. http://www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf (accessed Feb. 03, 2017).
[7] 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. doi: 10.1109/WICSA-ECSA.212.17.
[8] 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. doi: 10.1145/2786805.2786848.
[9] E. Lim, N. Taksande, and C. Seaman, "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Software*, vol. 29, no. 6, pp. 22–27, Nov. 2012, doi: 10.1109/MS.2012.130.
[10] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, Nov. 2012, doi: 10.1109/MS.2012.167.
[11] "Code Quality and Security | SonarQube." https://www.sonarqube.org/ (accessed Sep. 24, 2019).
[12] "The 2020 State of Code Review Report," *SmartBear Software*. https://smartbear.com/resources/ebooks/the-state-of-code-review-2020/
[13] 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. doi: 10.1109/MTD.2014.13.
[14] C. Chen, R. Alfayez, K. Srisopha, B. Boehm, and L. Shi, "Why is It Important to Measure Maintainability, and What Are the Best Ways to Do It?," in *Proceedings of the 39th International Conference on Software Engineering Companion*, Piscataway, NJ, USA, 2017, pp. 377–378. doi: 10.1109/ICSE-C.2017.75.
[15] J. Ludwig, D. Cline, and A. Novstrup, "A Case Study Using CBR-Insight to Visualize Source Code Quality," Big Sky, MT, Mar. 2020.
[16] "SciTools.com." https://scitools.com/ (accessed Dec. 24, 2018).
[17] J. Ludwig, S. Xu, and F. Webber, "Compiling static software metrics for reliability and maintainability from GitHub repositories," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct. 2017, pp. 5–9. doi: 10.1109/SMC.2017.8122569.
[18] C. Baldwin, A. MacCormack, and J. Rusnak, "Hidden Structure: Using Network Methods to Map System Architecture," 2014. [Online]. Available: http://www.hbs.edu/faculty/Publication%20Files/13-093_3011858c-cd52-494b-b58b-f46af7331e85.pdf
[19] "LGTM code quality: Tools to measure quality of your source code." https://blog.semmle.com/code-quality-tools/ (accessed Sep. 09, 2019).

BIOGRAPHY

Jeremy Ludwig, *PhD, is a principal engineer at Stottler Henke Associates, Inc. He directs teams of computer scientists and conducts research in artificial intelligence, applying reasoning, knowledge representation, and machine learning to create solutions for complex, real-world, problems.*

Devin Cline *is a software engineer at Stottler Henke Associates, Inc. He has built case based reasoning, image analysis, behavior analysis, and visualization components for scheduling and decision support systems.*