A Case Study Using CBR-Insight to Visualize Source Code Quality

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

***Abstract*—Creating and maintaining high-quality source code is especially important for critical systems such as those made for NASA and the DoD, and for software product lines where long-lived, reusable modules are intended to be shared by multiple systems. CBR-Insight is an automated code assessment tool developed for the US Air Force, and released as open source on GitHub, to provide an objective and understandable measure of software quality. CBRI-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 contributions of this work include advancing the state of the art in assessing software code quality, presenting a method to communicate code quality to decision makers, and examining a case study where these methods are applied to develop software in the aerospace domain.**

TABLE OF CONTENTS

1. INTRODUCTION 1
2. RELATED WORK 1
3. CBR-INSIGHT 2
4. METHODS 5
5. RESULTS AND DISCUSSION 7
6. CASE STUDY 9
7. CONCLUSION 10
ACKNOWLEDGEMENTS 11
REFERENCES 11
BIOGRAPHY 12

1. INTRODUCTION

Creating and maintaining high-quality software is especially important for critical systems such as those designed for NASA and the DoD, and 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 is reliable, maintainable, and has limited technical debt. Software development teams generally employ a variety of design techniques, processes, and tools to continually work towards quality code while balancing the overall time and budgetary demands of the project. CBR-Insight (CBRI) is an automated code assessment tool developed for the US Air Force and released as open source on GitHub. 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. 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. What makes CBR-Insight a complementary addition to existing tools is: (i) the calculation of a small, essential set of static code metrics associated with maintainability, reliability, and technical debt, (ii) using a customizable set of peer projects to determine the target ranges associated with each metric, and (iii) presenting the information in a format preferred by decision makers. This paper begins with an overview of related work and an in-depth look at CBRI. Following this high-level review, we analyze the data that underlies CBRI in the Methods and Results & Discussion sections. Next, we present a real-world case study that illustrates how CBRI is applied as part of a suite of tools and processes to the development of critical software for scheduling and deconflicting satellite communications. The paper concludes with a description of ongoing work on CBRI and provides links to the source code on GitHub.

The primary contributions of this work include advancing the state of the art in assessing software code quality, presenting a method to communicate code quality to decision makers, and examining a case study where these methods are applied to develop software in an aerospace domain.

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.

978-1-7821-2734-7/20/$31.00 ©2020 IEEE

Software Quality Models

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 [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. 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 using tools such as SonarQube. CBRI builds on all of this prior work in creating its underlying software quality model.

Technical Debt

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 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, 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 technical debt in order to support software reliability and maintainability.

Automated Code Review

There are several practical tools aimed at improving source code quality and reducing technical debt, such as SonarQube [11] and Codacy [12]. These and other 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 the 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 in which to interpret and make use of the results.

3. CBR-INSIGHT

CBRI is a web application equipped with two main user interface components: Dashboard and Project View. The Dashboard provides an at-a-glance summary across a number of projects, while the Project View enables a deep look into a single project. Behind the scenes, CBRI uses the Understand static source code analysis tool developed by SciTools to generate the data displayed in the web app.

Dashboard

The CBRI Dashboard shown in Figure 1 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.

Architecture—One especially important technique to reduce complexity is developing software in a modular and hierarchical fashion. The term architectural complexity is used to describe how a software architecture makes use of modularity and hierarchy. Modularity and hierarchy reduce the dependencies between different pieces of the source code, so a change in one file doesn't propagate changes to many

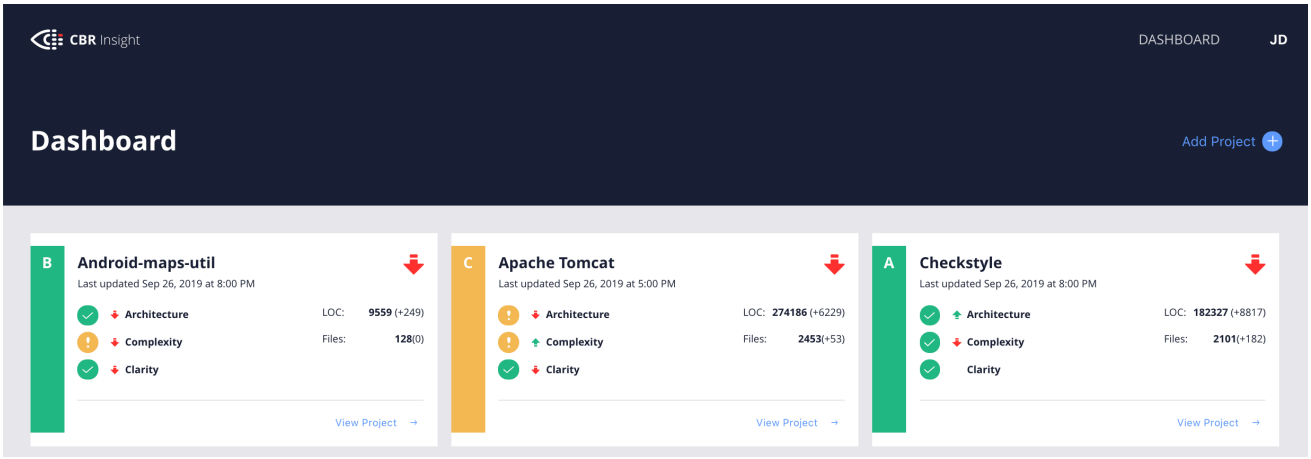

Figure 1. CBRI Dashboard with three example projects

others. Similarly, a developer can make a change in one file without having to arrive at a detailed understanding of all of the other affected files. Systems with a good architecture score are those that make better use of modular and hierarchical structures than their peers.

Complexity—Software developers also work to manage the complexity within each individual component (class or file) of the source code. Simply put, components that contain less logical complexity, less coupling to other files, fewer methods, and less code to deal with are more reliable and maintainable. Despite this general guidance, some complexity is always expected—there will necessarily be some number of overly complex, highly coupled, and lengthy components in all but the simplest of projects. Systems with better complexity scores have fewer overly complex components than their peers.

Clarity—Software developers (hopefully) strive to write code that is simple and readable rather than clever. They use descriptive names for classes, methods, and variables aimed at making code easy to understand. They add comments to their source code to provide an overview or to describe the intent of the code. While difficult to objectively measure, the clarity of source code has a marked impact on reliability and maintainability. Systems with better clarity scores are those found to be well commented. Additional clarity measures for readability are discussed as part of Future Work.

Scores and grades are calculated by comparing the calculated metrics of the project against the values from a set of peer projects, as described in the Methods section. For interpreting the grades, it is important to know that they are distributed on a bell-shaped curve. The most common score is a C, with fewer Bs and Ds and even fewer As and Fs. For example, getting a C indicates that reliability, maintainability, and technical debt should be about average with other production-level projects. Improvements to the scores can be made by addressing the areas with red and yellow scores. An F on the other hand indicates this project is significantly worse off than other production-level efforts. In this case, one would expect more than usual difficulty in making changes (maintainability) and keeping the project running well (reliability) until the underlying code quality issues have been addressed.

Project View

The Dashboard is the starting point for the user to drill down into the details of each project. 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 accessible descriptions 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 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.

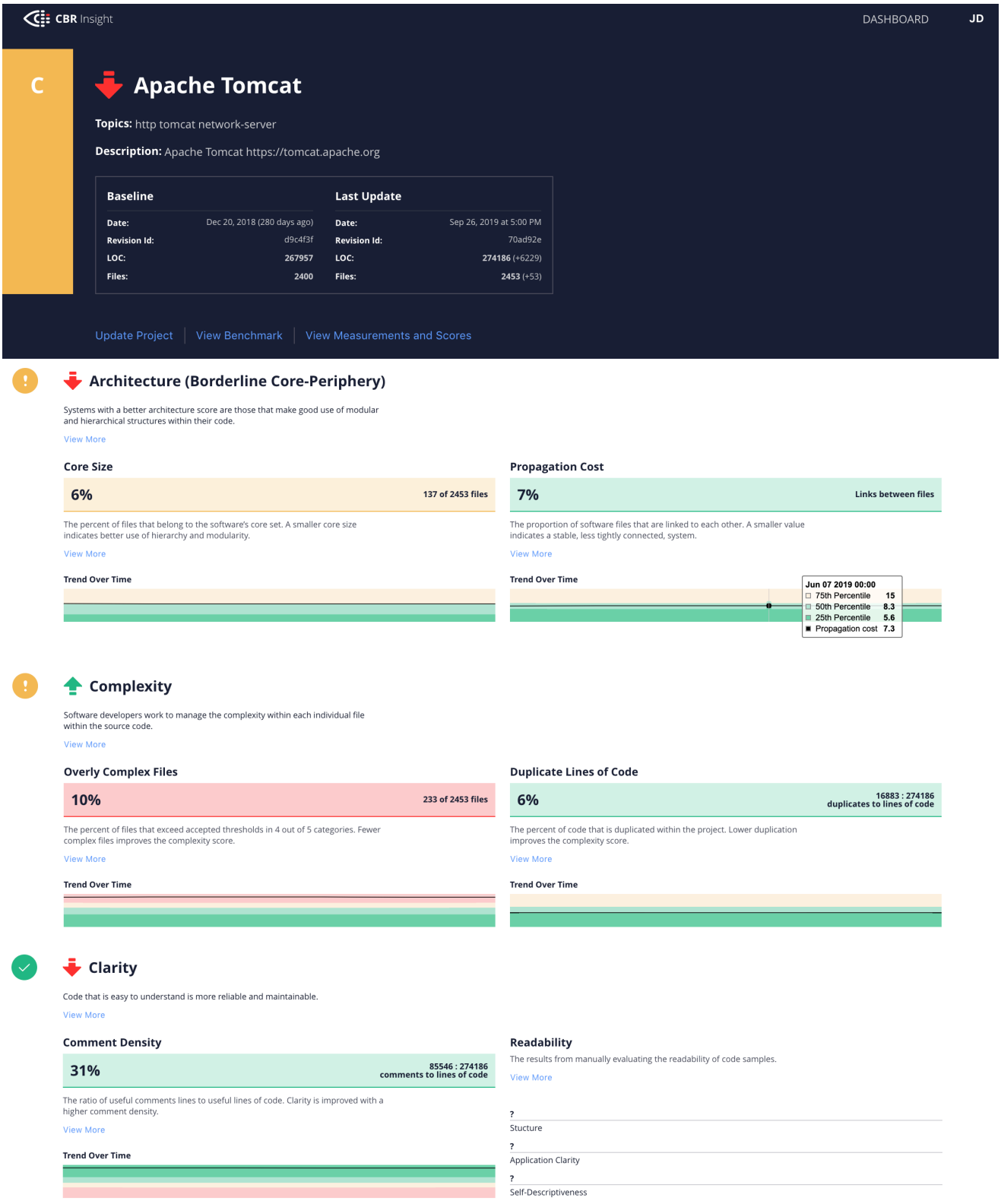

Figure 2. CBRI Project View general information and metric details

The following section of the Project View in Figure 3 contains a treemap, where each box is a file, while 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 (core, shared, control, peripheral, and isolate). 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 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.

Understand

CBRI uses a plugin to the proprietary Understand static source code analysis software [15] to calculate the architecture, complexity, and clarity metrics. While CBRI focuses on presenting decision makers an overview, software developers can use Understand and the plugin directly to calculate the same metrics and address identified deficiencies. The plugin is included as part of the source code hosted on GitHub.

4. METHODS

One of the most useful aspects of CBRI is providing a context in which to understand the metric calculation results for a particular project. For example, a project manager might ask: "Is a propagation cost of 7% good or bad? If it is bad, what is a reasonable number?" The context used to create the graphs and scores in the web application is driven by the creation of a library of peer projects. In order to trust the user interface, we need to examine the underlying data. The methods section offers details on how the project library was developed (i.e., by selecting and analyzing open source projects from GitHub). A replication package is available online as part of the open source release (https://github.com/StottlerHenkeAssociates).

Measurements

A number of measurements were gathered through the GitHub Application Programming Interface (API) by examining the commits in the version control system as well as by analyzing the source code using the Understand static source code analysis tool developed by SciTools. These measurements were used to develop the project library and to assess the utility of the CBRI metrics.

Stars, open and closed issue counts, number of releases, and topics were gathered via the GitHub API. Stars are assigned by GitHub users and serve as a measure of a project's popularity. Topics are self-assigned project descriptors (e.g., 'machine-learning'). Number of commits and contributors

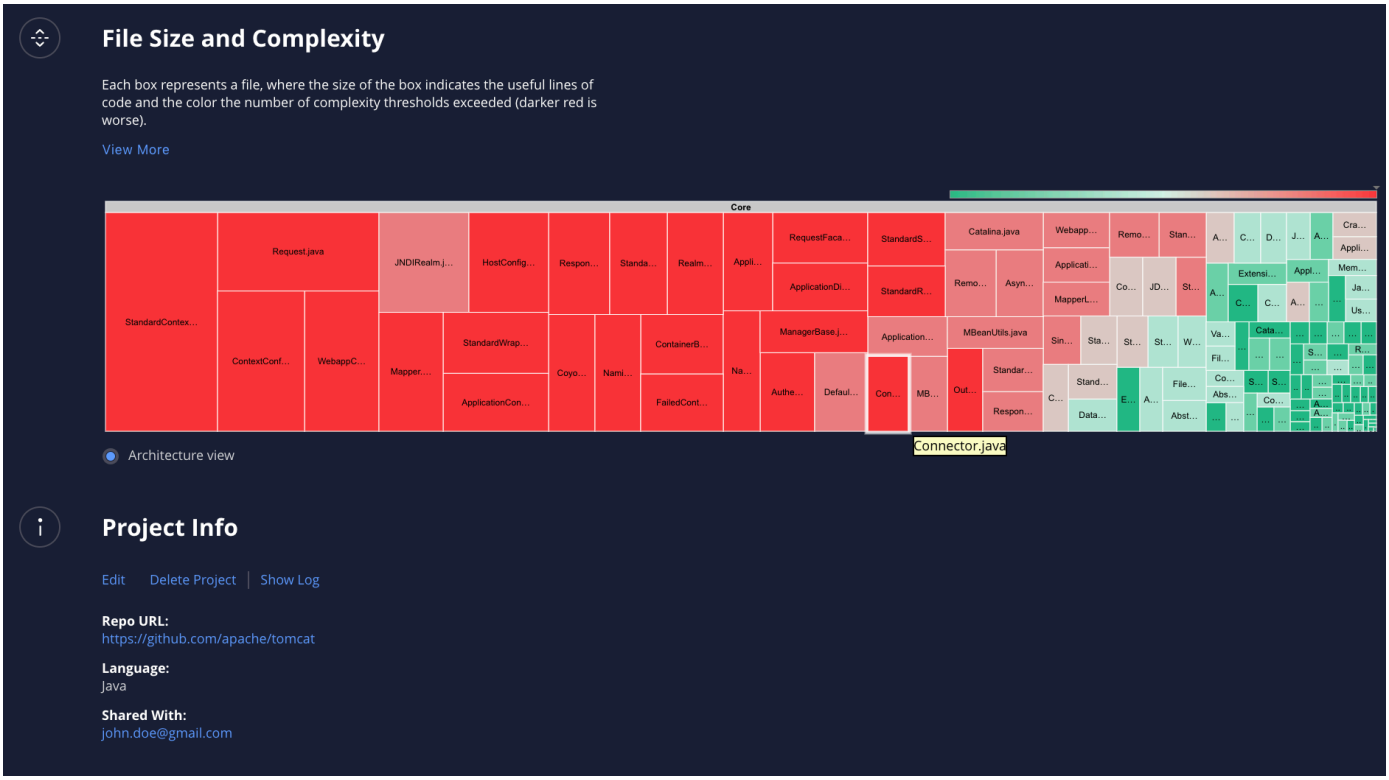

Figure 3. CBRI Project View architecture treemap and project information

were determined by examining the Git repository. Finally, the number of classes and files, lines of code, and lines of comments were measured by Understand. The lines of code measure attempts to capture the number of lines developers would actually need to review to comprehend the code. Similarly, the lines of comments measure attempts to weed out license headers and comments that aren't meaningful.

Metrics

A brief summary is given below of the calculated metrics; see [16] for a more detailed discussion of the specific metrics selected for use in CBRI along with details on how they are calculated.

Architecture—The architecture metrics are Core Size and Propagation Cost [17]. The Core is the largest set of components (classes or files) that are interdependently linked to each other*; Core Size* refers to the size of the Core relative to the total number of components. *Propagation Cost* is a system-wide metric that describes the proportion of software files that are directly or indirectly linked to each other. Both of these metrics provide a single, system-wide measure of how interconnected the source code is and therefore how extensive/expensive a change might be on average.

Complexity—The two complexity metrics are Percent Overly Complex and Percent Duplicate LOC. An overly complex file is one that exceeds 4 of 5 thresholds from a set of standard software metrics [1], including LOC, WMC-Unweighted, WMC-McCabe, RFC, and CBO. The reasoning is that any component that fails the majority of these metrics is likely to actually be complex, not just large. Duplicate lines of code are defined as blocks of ten or more lines that are exactly repeated in more than one location. This was selected as a reasonable threshold where abstraction should be used rather than copy-and-paste.

Complexity—Code-To-Comment ratio is used as an initial measure of clarity. This metric has been well studied as part of earlier work on quality models [18].

Project Libraries

For C, C++, C#, and Java, a project library was selected by identifying the top 1000 GitHub repositories in each language, sorted by number of stars. In all cases, the projects needed to be at least 200 KB. Each repository was analyzed with Understand and the plugin to generate a table of measurements and metrics. For C, files were used instead of classes for object-oriented metrics. Up to 2 hours was allowed for analysis with Understand and for running the plugin (i.e., up to 4 hours total per project). Projects that did not complete either step within 2 hours were not included in the library (25 for C, 19 for C++, 5 for C#, and 1 for Java).

All project libraries were then processed to remove repositories that were not likely to be actual software projects. The filter removed repositories with: < 100 stars, < 30 commits, < 1 release, < 1000 lines of code, <= 0 propagation cost, >=100% comment density, >= 100% duplicate code.

The result is a library that contains the most popular and successful projects available in GitHub for each language. Our assumption is that by selecting only the most successful open source projects, the libraries will include primarily production-quality source code, which is the target population of CBR-Insight.

Defect Proneness

While stars are at best a measure of a repository's popularity on GitHub [19], CBR-Insight is attempting to score projects based on their reliability and maintainability. To address this, we use a measure of defect proneness based on [20]. A bug fix (defect) commit is one that includes any of the following key words: "error," "bug," "fix," "issue," "mistake," "correct," "fault," "defect," or "flaw." Defect proneness is the ratio of defect commits to all commits.

This measure is obviously not going to discriminate bug fixes from feature commits perfectly. Defect proneness is also only a proxy for reliability, not maintainability. That said, defect proneness shows a bell-shaped distribution across projects for all languages. This matches the expectation that among successful projects, some have numerous defects, while others have few, and most a medium amount. Defect proneness is also significantly associated with LOC, which matches the general observation that more LOC leads to more bugs. Finally, defect proneness is not significantly associated with stars. Given these findings, we have opted to continue the analysis using defect proneness as a better proxy for reliability than number of stars among these successful projects.

Peer Project Selection

Peer projects are used to provide a context in which to understand the metric values generated for a project. For example, a core size of 17% is by itself difficult to understand. Peer projects support comparing that value to the scores of similar, successful projects (e.g., the median core size is 20%, so 17% is a reasonable number).

Peer projects are selected from the project library for a target project in accordance with the following criteria. First, the primary language must be the same as in the target project. Second, a project must share at least one topic with the target project. Third, the project must be within +/- 40% LOC of the target project. Finally, a minimum of 25 peer projects is required. In cases where there are not enough projects that share a Topic, then all projects are considered within the LOC range. If there are still not enough projects, then the 25 nearest projects in terms of LOC are selected.

Aggregated Scoring

Four project scores are created by comparing the metrics of the target project to the selected peers. The four score components are Architecture, Complexity, Clarity, and

Overall. The definitions of the numeric scores are given below. For the numeric score, M_p is the percentile of the metric relative to the peer projects. Some of the percentiles are inverted; a higher score is better.

- Architecture = (1- $CoreSize_p$) + (1- $PropagationCost_p$)
- Complexity = (1- $PercentComplexFiles_p$) + (1- $PercentDuplicateLOC_p$)
- Clarity = $UsefulCommentDensity_p$
- Overall = Architecture + Complexity + Clarity

Each score component (Architecture, Complexity, Clarity, and Overall) is also assigned a letter value in addition to the numeric. The definition of all score components is based on $Score_L$, the percentile of the score relative to the scores of all of the projects in the library of the same language. $Score_L$ > 0.9 = A, > 0.7 = 'B,' > 0.3 = 'C,' > 0.1 = 'D,' and <= 0.1 = 'F.' As described, 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. So, for example, all of the projects with an "A" will be those that scored the best against their peers.

Statistical Methodology

We used Spearman correlation and partial correlation to compare the strength and direction of associations between variables. Log_2 transformation was performed on variables with a long-tailed distribution (e.g., LOC, Contributors, Commits). Significance is determined by p <= 0.05. Coarsened Exact Matching (CEM) is used for effect estimation. Intuitively, what CEM does is compare projects that are similar (e.g., similar LOC and contributors). That is, for each value of the "treatment" variable (e.g., core size), it finds groups of examples that are similar in terms of the potential confounders (LOC and contributors) but that differ on the treatment variable. Observations that can't be matched that way are discarded. The ones that are "matched" are weighted appropriately to reduce imbalance within *and* across groups.

5. Results And Discussion

After filtering, the project library contains the following number of projects per language: C 664, C++ 700, C# 756, and Java 669. While a few projects had 1M+ LOC, the bulk were less than 500k. We use the project library to address several research questions and then discuss threats to validity.

Q1: How do the metrics relate to defect proneness?

The metrics were selected based on evidence in the literature of being indicators of reliability or maintainability. Based on this, it is expected that core size, Propagation Cost, and Percent Overly Complex will be significantly related to defect proneness (reliability). Percent Duplicate LOC and Comment Density are primarily related to maintainability (not reliability) and therefore are not expected to relate significantly to defect proneness.

The results generally meet these expectations as shown in Figure 4, which analyzes all languages at once (with similar findings for each language individually). Slight but significant associations were found (as expected) for the reliability measures—with one exception: Percent duplicate LOC was found to be negatively associated with defect proneness in some languages. This does not go against our expectations (there is no positive association), but it is an unexpected result that warrants further study.

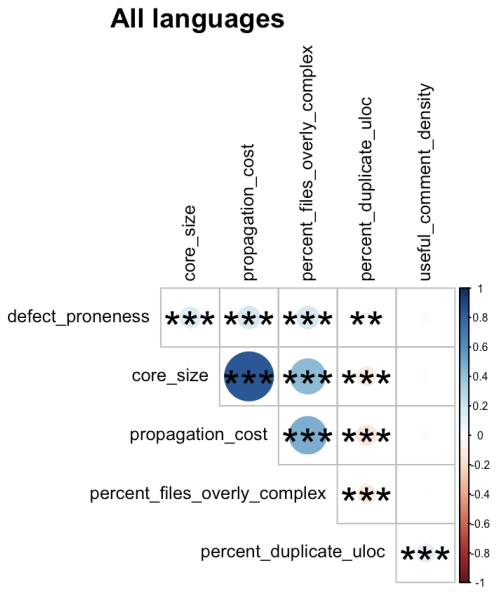

Figure 4. Partial correlation plot, accounting for LOC and Contributors (p < 0.05 *, < 0.01 **, < 0.001 *)**

Q2: How do the Architecture, Complexity, Clarity, and Overall scores relate to defect proneness and LOC?

Based on the metrics used to create the aggregated scores, it is expected that Architecture will be most strongly related to defect proneness, followed by Complexity and Overall. The Clarity score is not expected to be related to defect proneness. Additionally, it is expected that all of the scores are measuring something other than LOC, so there should be no significant relationship between the scores and LOC.

The results generally met expectations as shown in Table 1. The Architecture and Overall scores had small, significant associations with defect proneness, accounting for LOC and contributors. The complexity score was not associated with defect proneness, which was unexpected and may be due to the unexpected findings with duplicate LOC. Clarity performed as expected as it was unrelated to defect proneness. We do still expect that Complexity and Clarity are associated with maintainability, based on metric selection. None of the scores are significantly related to LOC or stars.

Table 1. Partial correlations between Architecture and Overall scores and defect proneness, accounting for LOC and contributors

Score	Language	Estimate	p-value	Statistic	N	Significance
Architecture	All	-0.132	2.81E-12	-7.018	2789	***
	C	-0.127	1.09E-03	-3.282	664	**
	C++	-0.132	4.51E-04	-3.525	700	***
	C#	-0.183	4.52E-07	-5.090	756	***
	Java	-0.100	9.74E-03	-2.592	669	**
Overall	All	-0.120	1.82E-10	-6.399	2789	***
	C	-0.142	2.55E-04	-3.677	664	***
	C++	-0.123	1.13E-03	-3.270	700	**
	C#	-0.135	2.08E-04	-3.728	756	***
	Java	-0.101	8.77E-03	-2.629	669	**

Significance: p < 0.10 '.' , < 0.05 *, < 0.01 **, < 0.001 ***

Table 2. Effect size estimate for Architecture and Overall score using CEM, relative to $\log_2$ LOC

Score	Langauge	Var	Estimate	Std. Error	t value	p-value	Significance
Architecture	C	architecture_score_letter	-0.869	0.422	-2.059	4.03E-02	*
	C	log_uloc	0.109	0.081	1.340	1.81E-01	NA
	C++	architecture_score_letter	-1.639	0.433	-3.782	1.84E-04	***
	C++	log_uloc	0.254	0.101	2.523	1.21E-02	*
	C#	architecture_score_letter	-1.560	0.361	-4.316	1.97E-05	***
	C#	log_uloc	0.254	0.084	3.014	2.73E-03	**
	Java	architecture_score_letter	-0.796	0.358	-2.227	2.65E-02	*
	Java	log_uloc	0.149	0.077	1.944	5.25E-02	.
Overall	C	overall_score_letter	-0.479	0.441	-1.087	2.78E-01	NA
	C	log_uloc	0.160	0.094	1.702	8.98E-02	.
	C++	overall_score_letter	-1.730	0.418	-4.134	4.38E-05	***
	C++	log_uloc	0.241	0.097	2.471	1.39E-02	*
	C#	overall_score_letter	-1.097	0.403	-2.719	6.79E-03	**
	C#	log_uloc	0.409	0.093	4.413	1.27E-05	***
	Java	overall_score_letter	-1.472	0.397	-3.711	2.41E-04	***
	Java	log_uloc	0.305	0.096	3.171	1.65E-03	**

Significance: p < 0.10 '.' , < 0.05 *, < 0.01 **, < 0.001 ***

Even though the strength of association between Architecture and Overall scores and the proxy reliability measure are small [21], the estimated effects are significant from a software development standpoint. Table 2 estimates the effect of moving a project's Architecture and Overall letter grade up by one. For example, in C++, moving up one letter grade in Architecture would result in 1.6 percentage point fewer defects on average, with a standard error 0.43 away from that average. In contrast, this is much more than the 0.254 percentage point decrease that would be expected by cutting the size of the code base in half ($\log_2$ LOC). The effect for Architecture is significant in all languages and is larger than the estimated effect of cutting the code size in half in all languages with a correspondingly larger amount of error. The effect is significant for Overall in all languages except C and larger than the estimated effect of cutting the code size in half in C++ and Java. The error is correspondingly higher than for the LOC estimated effect in all significant cases. While a significant effect was not found for the letter grade in C, significant effects were found for the Overall score in all languages (not pictured). These estimates illustrate the likely effects of improving the Architecture and Overall scores on reliability, tempered by the large standard error which indicates the variability in the effect.

Discussion

The takeaway is that as predicted by the literature, the underlying metrics and the Architecture and Overall scores built with these metrics are significantly associated with defect proneness, are providing information on reliability independent of LOC, and have a significant estimated effect on reliability in all languages.

Threats to Validity

Project Library—First, the 4,000 selected repositories represent a tiny slice of the available repositories on GitHub. More data would improve the analysis, especially given the loss of projects to filtering. In particular, more projects are needed below 4k LOC and above 260k LOC. Second, the four-hour processing limit may have contributed to fewer larger projects in C and C++. Third, the libraries were generated from open source software. It is possible that metrics and scores generated from the code of popular open source projects might not be similar to the code of proprietary production-level projects.

Defect Proneness—While this appears to work reasonably well, it is clearly an imperfect proxy for reliability.

Metric Selection—A small set of metrics for reliability, maintainability, and preventable technical debt was identified during the Phase I research, based on a thorough literature review. It is possible that different metrics, or a larger number of metrics, would have yielded better results.

Peer Project Selection—Peer project selection involves a number of free parameters (same language, shared topic, +/- 40% LOC). We selected these values based on experience developing software as reasonable defaults. There are likely different values that would better fit defect proneness. Given that defect proneness is only a loose proxy of reliability and there is no measure of maintainability, we did not attempt to tune peer project selection parameters to improve results.

Topics—Matching shared topics across projects relies on self-reported GitHub topics. Many projects have no topic at all or use different terminology than other, similar topics. Additionally, some topics such as 'android' are extremely broad. In grading the peer project library, topic-based peer selection was used infrequently in C++/C#/Java and never used in C.

Aggregated Score—A purposely simple aggregation method was used to create the Architecture, Complexity, Clarity, and Overall scores, where the focus was on ease of description as opposed to maximizing association with popularity, reliability, etc. There are likely more complex ways to combine this information that would yield a score that is more highly correlated with metrics such as defect proneness. Given that defect proneness is only a loose proxy of reliability and there is no measure of maintainability, we did not attempt to tune scoring.

Bias in causal inference—Inferences regarding the effect of software design/engineering choices on reliability are inherently causal in nature, and causal inference from observational data is prone to bias. We attempted to mitigate this threat by employing matching techniques (specifically, Coarsened Exact Matching) to approximate the results of a blocked experiment. However, just as in a blocked experiment, bias can remain if there are any unobserved/unblocked confounders. Unlike a blocked experiment, we have no opportunity to randomize units into treated and control groups in order to avoid bias caused by unknown or uncontrollable confounders.

Maintainability—A large flaw in the analysis is that there is no reliable measure of maintainability against which metrics and scores could be compared, as was done for reliability. Automated maintainability measures (e.g., Maintainability Index) can be useful to prioritize what parts of the software need attention, but they do not seem to correlate well with the amount of maintenance-related effort required [10]. Given that the metrics were selected based on their association with reliability and maintainability in the literature, and we did show a relationship between scores and reliability, we fully expect similar relationships to exist between scores and maintainability.

6. CASE STUDY

Stottler Henke is applying CBRI to the development of critical software for scheduling and deconflicting satellite communications for the US Air Force [22]. It is important to note that this software, like many projects, started as a rapid prototype to demonstrate proof of concept and then began transitioning into a production-level system. This section provides a high-level view of how CBRI is helping to improve source code quality over time as part of a software quality pipeline.

A software quality pipeline is a combination of a team's or organization's culture, processes, and tools aimed at producing high quality software – sharing much in common with their DevOps pipeline [23]. Just like there is no single DevOps solution that works in all contexts, there is also no single software quality pipeline that works everywhere. In this use case, the culture includes a focus on developing reliable, maintainable, and secure software as a long-term investment, the processes are those common to lean and agile software development, and the tools are primarily the same as those hosted by D2IE DevTools [24] for US DoD software development.

Jira, by Atlassian, is used for issue tracking and serves as the primary interface point between project management and development. *Issue* in this case is a nebulous term—it can mean a requirement, feature, bugfix, test, etc. Bitbucket, also by Atlasssian, is used as the version control system. Jenkins, an open source project, is used to compile and test the software, and build software releases, to support continuous delivery. Both Fortify and SonarQube are used for automated code review, though in different ways. CBRI provides a birds-eye view of code quality and helps guide software development and refactoring efforts. Finally, Zephyr is used for software test management. Jira, Zephyr, Bitbucket, Jenkins, SonarQube, and CBRI are all integrated into an automated pipeline via plugins.

Beginning work on an issue involves starting a new source code branch in Bitbucket, which we will call the feature branch. A software developer(s) then makes any changes in

this branch. Once the changes are made, the developer issues a pull request that signals the feature branch is ready to merge the change into the main development branch. This triggers several actions. First, Jenkins compiles the feature branch and runs automated tests on this branch. The developer needs to correct any compilation or failed test issues to proceed. Second, the feature branch is analyzed by SonarQube. Any problems that SonarQube finds in the new code are posted to the pull request in Bitbucket. Software developers are expected to correct all identified problems before proceeding to the next step. Third, the feature branch undergoes a manual code review by another software developer. The reviewer will create tasks in Bitbucket to be addressed. Once the reviewer has verified all of the tasks have been corrected, then the feature branch is finally merged with the main development branch.

The objective of this process is to write clean code going forward. The combination of automated and peer review has three main benefits. First, the resulting code is more reliable and maintainable. Once in a while, a bug is found, but more often what is addressed are future maintainability issues. Second, these reviews mentor less experienced software developers. Requiring developers to fix all issues, whether from SonarQube or the team lead, before merging in their code generally encourages them to start doing the right thing the first time. Third, the manual reviews spread knowledge of the code out across the development team. While developed independently, this lightweight review system is very similar to that used by Google in terms of tools, process, and motivation [25].

While this process focuses on good, clean code going forward, it does not address all of the historic technical debt. First, the team uses Fortify to identify and fix the high-priority security-related issues. Second, CBRI is used to guide software development and refactoring efforts. Guidance for software development includes maintaining a high level of quality comments, managing the complexity of individual classes, and developing modular components as part of the day-to-day development process. Refactoring is taken on as time allows. There is never enough time to address the full backlog of technical debt; CBRI is used to identify the most complex files in the Core architecture group as most likely to yield a return on investment.

During the last six months, this development environment has resulted in slow but steady improvement along nearly all dimensions: a 3% reduction in core size, a 1% reduction in overly complex files and duplicate code, and a 1% increase in comment density. The result was an improvement in the complexity, clarity, and overall scores. This was during a time period when sprints were devoted to feature development and testing, with little time assigned to refactoring and a 10% increase in the LOC. In addition to suggesting refactorings, CBRI is used by the project manager to track this reduction in technical debt over time and provide evidence that the development environment was working as expected.

However, it was not all good news. Propagation cost rose 6% in this same timeframe. This points to a fundamental challenge of architecture issues—they are not easy to fix. Without a concerted refactoring effort to make the code base more modular and hierarchal, adding new features across the existing architecture tends to increase the technical debt. In this case, CBRI is used to document the increase and highlight the need for architecture refactoring along with new feature development.

7. CONCLUSION

Software code quality and technical debt have significant impact on a software product's reliability, maintainability, and security. This paper described the open source tool CBRI, built for the US Air Force to measure and visualize source code quality. The CBRI web-application provides an objective and understandable measure of software quality that can help guide decisions and direct limited resources during software acquisition, development, and sustainment. The analysis of the data behind CBRI provided evidence to support this claim and also clearly lays out threats to validity. Finally, a concrete use case illustrated how CBRI was used as part of a development environment to improve code quality in software for scheduling and deconflicting satellite communications.

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 CBR-Insight 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 and a replication package are available at: https://github.com/StottlerHenkeAssociates.

Future efforts on CBRI focus on three main areas. First is identifying additional metrics that gauge the understandability of software or the clarity measure. Not surprisingly, software developers are better judges of the understandability of code than automated systems [14], [26]. Our current approach is derived from [14], which involves including a questionnaire on readability and understandability as part of each manual code review. A different approach suggested by [26] is to investigate deep learning methods to see if they are able to accurately and automatically assess understandability. Second is improving the project library. This could be accomplished by including more popular open-source projects, especially those with more than 260k LOC, and by making it easier for organizations to augment the library with their own projects. Third is increasing the visibility of the CBRI results. Currently the CBRI results are presented in a stand-alone web application, which involves an extra step to view the results. In our case study, we found this one extra step to present a significant barrier at times. The solution is to build a Jira

dashboard plugin that would properly situate the CBRI results within the familiar, and extensively used, Jira dashboard.

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.

[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.

[5] Squale Consortium, "Visualization of Practices and Metrics," Mar. 2010.

[6] 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: 03-Feb-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.

[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.

[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.

[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.

[11] "Code Quality and Security | SonarQube." [Online]. Available: https://www.sonarqube.org/. [Accessed: 24-Sep-2019].

[12] "Automated code reviews & code analytics." [Online]. Available: https://www.codacy.com/. [Accessed: 24-Sep-2019].

[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.

[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.

[15] "SciTools.com." [Online]. Available: https://scitools.com/. [Accessed: 24-Dec-2018].

[16] 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)*, 2017, pp. 5–9.

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

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

[19] H. Borges and M. Tulio Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, Dec. 2018.

[20] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A Large Scale Study of Programming Languages and Code Quality in Github," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2014, pp. 155–165.

[21] C. J. Ferguson, "An effect size primer: A guide for clinicians and researchers.," *Professional Psychology: Research and Practice*, vol. 40, no. 5, pp. 532–538, Oct. 2009.

[22] R. Stottler and R. Richards, "Managed intelligent deconfliction and scheduling for satellite communication," in *2018 IEEE Aerospace Conference*, 2018, pp. 1–7.

[23] A. Wiedemann, N. Forsgren, M. Wiesche, H. Gewald, and H. Krcmar, "Research for practice: the DevOps phenomenon," *Commun. ACM*, vol. 62, pp. 44–49, 2019.

[24] "DI2E DevTools," *DI2E*. [Online]. Available: https://www.di2e.net/display/DI2E/DI2E+DevTools. [Accessed: 27-Sep-2019].
[25] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern Code Review: A Case Study at Google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, New York, NY, USA, 2018, pp. 181–190.
[26] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Automatically Assessing Code Understandability," *IEEE Transactions on Software Engineering*, pp. 1–1, 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.*

Aaron Novstrup *is an artificial intelligence software engineer and researcher at Stottler Henke Associates, Inc. He has practical experience with a broad range of AI applications, including information extraction, decision support, and knowledge representation.*