Evaluating How Static Analysis Tools Can Reduce Code Review Effort

Devarshi Singh, Varun Ramachandra Sekar, Kathryn T. Stolee
Department of Computer Science
North Carolina State University
{dsingh4, vramach, ktstolee}@ncsu.edu

Brittany Johnson
College of Information and Computer Science
University of Massachusetts, Amherst
bjohnson@cs.umass.edu

***Abstract*—Peer code reviews are important for giving and receiving peer feedback, but the code review process is time consuming. Static analysis tools can help reduce reviewer effort by catching common mistakes prior to peer code review. Ideally, contributors would use static analysis tools prior to pull request submission so common mistakes could be addressed first, before invoking the reviewer.**

To explore the potential efficiency gains for peer reviewers, we explore the overlap between reviewer comments on pull requests and warnings from the PMD static analysis tool. In an empirical study of 274 comments from 92 pull requests on GitHub, we observed that PMD overlapped with nearly 16% of the reviewer comments, indicating a time benefit to the reviewer if static analyzers would have been used prior to pull request submission. Using the non-overlapping set of comments, we identify four additional rules that, if implemented, could further reduce reviewer effort.

I. Introduction

Peer code review is a form of code inspection that helps developers assess code for style, defects, and other standards prior to integration into a codebase [4]. The code reviewer has to be knowledgeable in both the possible defect patterns and coding standard violations, and apply that knowledge in reviewing code written by others. The importance of maintaining coding standards has been shown in previous studies [9], [13], [14]. The biggest challenge of the review process is the manual effort required from reviewers. The reviewers potentially have to look at hundreds of lines of code to find possible flaws and write comments to communicate the issues.

Static analysis tools like FindBugs [21] and PMD [22], have proven to be effective in checking for design flaws and violations in coding standards in an automated fashion [2], [19]. Existing research has explored integrating static analysis into the code review process *after submitting* code for review to reduce the effort required by code reviewers [4], [23]. Static analysis can help reduce manual effort by automatically checking for standard coding and style violations so that the code reviewer can focus on more important tasks, such as finding logical issues in code. Even when leveraging static analysis, code reviewers still must sift through the static analysis warnings to determine those that matter and add comments for issues that static analysis did not find.

978-1-5386-0443-4/17/$31.00 © 2017 IEEE

We suggest that static analyzers, such as PMD, should be used *before submitting* code reviews, placing the burden of sifting through static analysis warnings on the contributor rather than the reviewer. The goal of this work is to study the effectiveness of PMD in code reviews made on GitHub pull requests to help reduce the manual effort of reviewers.

The main contributions of this study are as follows:

- An empirical evaluation using PMD on Java files in 92 pull requests from 23 popular open source Java repositories in GitHub, and exploration of the overlap between PMD warnings and 274 code review comments.
- Identification of future PMD rules based on reviewer comments that are disjoint from PMD warnings.

Through this work, we lay a foundation to encourage more research on studying the effectiveness of static analyzers on reducing code reviewer effort, prior to code review.

II. Background and Related Work

Pull requests are used in GitHub repositories as part of *pull-based development* [7] to make a project team aware of changes attempting to be pushed to a GitHub repository. Pull-based development includes peer code review as part of assessment and acceptance processes [7], [8], as illustrated in Figure 1. The code review process consists of a reviewer from the project team who makes *reviewer comments* evaluating a *submitted pull request* from a contributor. The reviewer decides whether to approve, suggest changes to make (leading to *contributor modifications*), or reject the pull request. This cyclic process involving reviewer comments and contributor modifications can take several iterations prior to acceptance, especially since a majority of reviewers cite maintaining quality of code as a major challenge when working with pull-based development model [8]. Once approved, the reviewer merges the pull request into the project repository.

Code review introduces delays between submission and merging [7], and the number of comments is moderately correlated with time to merge [7]. Some pull requests are fully merged in an hour while others sit for days without resolution [7]. It is important to consider how to reduce the review effort. Tools can assist reviewers with making comments, such as Gerrit [3] or Codeflow [15]. Our proposed solution focuses on reducing the number of comments made by reviewers by leveraging static analysis tools.

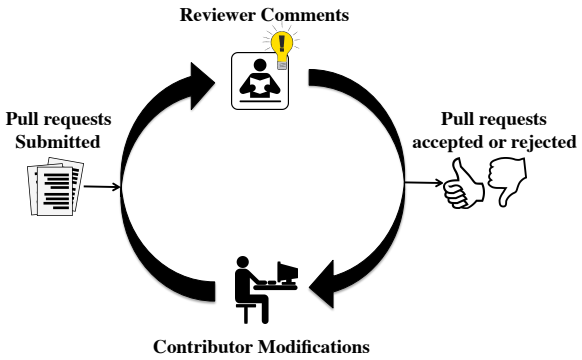

Fig. 1. An overview of code review in pull-based development.

Static analysis tools automatically analyze source code for defects and style issues [1], [5], [6]. Existing research suggests that developers occasionally use such tools in an attempt to improve or expedite the code review process [3], [15]. Review Bot extends Review Board [23] and uses multiple static analyzers, such as PMD, Checkstyle [20] and FindBugs to perform code review [4]. Closest to our work, Balachandran evaluated the effectiveness of Review Bot in the context of code reviews in VMware [24]. However, that study lacks an empirical evaluation over multiple open source projects.

III. Methodology

To understand how static analysis tools can help reduce the manual effort associated with code reviews, we explore:

- **RQ1:** *How often can PMD warnings preempt reviewer comments on pull requests?*
- **RQ2:** *Do the non-overlaps suggest new rules PMD could use to preempt reviewer comments?*

To address RQ1, we sample reviewer comments from 23 GitHub projects, run PMD against the files on which these comments were made, and identify overlaps between the reviewer and PMD for the same line(s) of code. With the non-overlapped set, RQ2 explores opportunities for future static (and non-static) rules that could further reduce reviewer effort.

A. Artifacts

We selected 23 popular open source Java projects on GitHub, which have between 619 and 11,084 forks and have between 29 and 4,841 kLOC. Projects were chosen by consulting the GitHub Rank website[1].

From each project, we selected four closed pull requests, each of which had at least one Java file changed. Among the 500 most recent closed pull requests, we filtered out those that lacked reviewer comments, leaving 10–15 pull requests per repository. From this small subset, we randomly selected four pull requests for analysis. Bot comments were removed, resulting in 274 comments analyzed from 92 pull requests in 23 projects; experimental artifacts are available[2].

[1]http://github-rank.com/fork?language=Java
[2]http://go.ncsu.edu/vlhcc_stolee_2017

B. Infrastructure and PMD configuration

We automated the task of fetching pull requests and reviewer comments on the 23 targeted projects. We used REST to query GitHub's v3 API and fetch the pull request data. We fetched pull request comments along with the diff of code on which the comment was made. After collecting all the data from the pull requests, we ran PMD version 5.5.1.

The PMD static analyzer scans Java source code and detects potential flaws in design and inefficient coding practices. PMD uses a set of rules organized under rulesets. These rulesets are easily extensible to meet the needs of a particular organization who wishes to follow a particular coding standard. We invoked PMD from the command line and passed the `all-java.xml` file as an argument to run all its rulesets.

The selection of PMD was influenced by the work done by Mäntylä, et al. who illustrate that code reviews discover *evolvability defects* (e.g., failing to include comments, not conforming to style, long subroutines, dead code) more than *functional defects* (e.g., logical issues in code that may affect the visible functionality of the application) [11]. PMD's bug finding capabilities were studied in alongside other common static analyzers [17], revealing that PMD and FindBugs [21] have a high correlation in warnings. Thus, we have reason to believe PMD is somewhat representative of other static analyzers. Furthermore, PMD is a mature, open source software tool that can run without compilation, making it convenient for exploring a large number of open source projects.

C. Analysis

To answer RQ1, two authors manually compared the warnings generated by PMD against the reviewer comments for each pull request. If the reviewer comment identified the same issue on the same line(s) of code, or provided the same solution as the PMD warning, it was marked as an overlap. PMD warnings are generally descriptive enough to convey their meanings precisely. In rare cases, the authors consulted the online documentation to better understand so that they could accurately compare PMD warnings with reviewer comments. If the comments required background knowledge of source code, the authors consulted the commits related to the pull requests to better comprehend the meaning implied. Thought inter-rater reliability was not measured, disagreements between the authors were rare.

Figure 2 shows an example of a PMD ruleset that was triggered on source code in the `Openhab` project, along with the reviewer comment and the portion of the source code diff relevant to the comments. Since both the PMD warning and the reviewer comment indicate the `equals()` method should be used for string comparisons, this was marked as an overlap.

If a PMD warning or a reviewer comment was missing for code in a pull request, then no overlap could be recorded. That is, if PMD did not produce a warning for code, but there was a reviewer comment, this was marked as a non-overlap. Conversely, if there was no reviewer comment for code in a pull request but PMD produced a warning, this did not make it into our data set.

PMD Ruleset:

`Strings: Use equals() to compare strings instead of '==' or '!='.`

Reviewer Comment:

"You should always compare strings using the 'equals' method in Java. Testing using the '==' method might only return 'true' accidentally."

Source Code Diff:

```
1  -    .getTransformationService(TCPActivator.
       getContext(), transformationType);
2  -  if (transformationService != null) {
3  -    transformedResponse = transformationService.
       transform(transformationFunction, response);
4  +  if (transformationType == "") {     // test
       for empty type first to avoid the WARN from
       getTransformationService
```

Fig. 2. Example overlapping comment to the PMD Strings ruleset from the Openhab project (link)

PMD Ruleset:

`Design: Private field 'fruitBowl' could be made final; it is only initialized in the declaration or constructor.`

Reviewer Comment:

"the private member are only set in the constructor so they should be final"

Source Code Diff:

```
1  +. public class Customer extends Thread {
2  +
3  +  private String name;
4  +  private FruitShop fruitShop;
5  +  private FruitBowl fruitBowl;
```

Fig. 3. Example overlapping comment to the PMD Design ruleset from the Java-design-patterns project (link)

IV. Results

Here, we address RQ1 and RQ2 in turn.

A. RQ1: PMD and Reviewer Comments Overlap

Of 274 comments, 43 overlapped with PMD warnings, representing 15.7% of the comments. For four of the pull requests, PMD overlapped with all of the reviewer comments (seven comments in total).

The primary reason for a lack of overlap between a comment and PMD warning was that PMD did not produce any warning at all for the commented code. Of the 231 non-overlapping comments, 227, 98.3% of them were because PMD produced no warnings. This provides an opportunity to use the non-overlapping comments to guide possible additional static analysis warnings to extend PMD and other similar static analyzers, as discussed in Section IV-B. For the remaining four (1.73%) of the reviewer comments, PMD produced a warning that was different from what the reviewer suggested.

The most common overlapping PMD ruleset category concerns code `comments`, representing 10 of the 43 overlapping comments This is on par with prior research showing that developers do not write enough, or descriptive enough comments [12], [16]. The next most common category is `Design`, representing seven of the 43 overlapping comments. An example is shown in Figure 3, where the reviewer and PMD suggest that one or more of the fields on the diff should be made `final`. In concert, the top two categories cover nearly 40% of all overlapping reviewer comments.

With static analysis tools, false positive rates are a concern for adoption [18]. Flipping around the analysis, we also looked at the entire set of PMD rules that were triggered on the evaluated pull requests. In total, 21 rules form 15 rulesets were triggered. There were six rules (28.6%) in the PMD rulesets such that, whenever triggered, were always caught by the reviewers. That is, for those six rules, every time it is triggered by PMD, it is correct and on-par with the reviewer comments. These rules are: **`RedundantFieldInitializer`**, **`UnusedImports`**, **`UseEqualsToCompareStrings`** (Figure 2), **`AvoidPrintStackTrace`**, **`AtLeastOneConstructor`**, and **`ReplaceHashtableWithMap`**.

Summary: After applying PMD to projects, 15.7% of the reviewer comments overlapped with a PMD warning. When there was no overlap, it was most commonly because PMD produced no warning at all. Of the 34 overlapping reviewer comments, 35% resided in just two of the 23 projects that we evaluated, showing not all projects would benefit equally from our proposed process. For six of the 21 rules triggered by PMD, the warnings always overlapped with reviewer comments, indicating no false positives for those rules.

B. RQ2: Non-Overlapping Comments Analysis

We stored non-overlapping comments to analyze the issues that code reviewers found but PMD did not detect. An example of the non-overlapping case involved PMD determining that the use of `printf` is a questionable usage of a logger but the reviewer suggests to use another form of print statement (`println` instead of `printf`). Hence we categorized this example as a non-overlapping case.

We utilised the non-overlapping comments to propose modifications to existing rulesets of PMD so that it can improve its efficacy at detecting more issues relevant to reviewers. Table I shows four proposed modifications to the rulesets for PMD. The *ID* column provides a reference point for each row to facilitate discussion. The *Rule Set* column identifies existing PMD rule sets to extend, *Rule* names the specific rule, *Description* describes the rule, an *Example Comment* illustrates via a reviewer's comment, and *# Comments* counts the number of comments that would fit this rule from the non-overlapping set.

A large number of reviewer comments (rule 1; 19 comments) focused on fixing indentation and spacing issues. Many reviewers commented on the naming of methods and variables that were not descriptive enough or that did not follow the camel-case standard (rule 2; 18 comments). Our above two

TABLE I
SUGGESTED ADDITIONAL RULES BASED ON REVIEWER COMMENTS

ID	Rule Set	Rule	Description	Example Comment	# Comments
1	Basic	`Indentation/ formatting/ spelling`	Issues with indentation on complex boolean formulas in if-statements	*"Indent this line to the right, to be visible that is not on the same level" (link)*	19
2	Basic	`Best practices`	Changes or practices suggested based on project/team norms	*"Maybe the message should be more descriptive in what is the header and what is the uri?" (link)*	18
3	Comments	`Commented code`	Commented code not removed	*"remove commented out code" (link)*	5
4	Logging (Java)	`Parameterized logging instead of string concatenation`	Use parameterized logging instead of string concatenation due to performance issues	*"Use parameterized logging instead of string concatenation." (link)*	4

observations are in line with the findings of Gousios, et al. who describe that most reviewers look for style conformance when they evaluate the quality of code in a pull request [8].

We further observed that some review comments asked contributors to remove commented code that was left in the pull request (rule 3; five comments). A new rule inside the `Comments` ruleset would serve as a warning to contributors to avoid this mistake and save precious reviewer time from not having to point this out repeatedly.

Summary: Through a manual analysis of the non-overlapping comments, we distilled a set of four possible future static rules that cover nearly 17% of the comments.

V. DISCUSSION

In this work, we look at *potential* efficiency gains that could be realized by using the PMD static analyzer prior to pull request submission, as a means to save reviewer time when the tool and the reviewer make the same comment on the same code change. While reviewers will still need to look over the pull requests, using PMD could have pre-empted nearly 16% of the 274 comments made by reviewers. Our suggested process change would impact the contributor the most. While it is generally the responsibility of the contributor to deliver quality code, there needs to be a balance between how much manual effort the code reviewer is willing to take up and how much effort can be pushed onto the contributor.

We do not speculate on the difficulties associated with getting contributors in open source software to change their processes and adopt static analysis tools. In fact, studies have already recognized this challenge, showing that static analysis tools are under-utilized, in part because the output is hard to understand and false positives are often abundant [17], which can deter developers from integrating the tool into their regular workflow [10], [18].

We observe that PMD generates the same class of warning repeatedly at different places in code which adds to its false positive count. For instance, the *Comments* ruleset of PMD warns developers to add comments for each public method and field. To reduce the number of false positives, we suggest configuring PMD's rulesets such that it generates a limited set of relevant warnings for each ruleset. For instance, in the case of the *CommentRequired* rule of the *Comments* ruleset, PMD can be configured to generate a single warning on the entire code, which would remind contributors that reviewers of the project regard comments as important. Projects could suggest configurations for such tools so that the warnings presented are most likely to be similar to the comments given by reviewers. Using learning algorithms to automatically configure static analyzers so they are more consistent with reviewer comments for a specific project is left for future work.

A. Threats to Validity

We looked at a limited number of open source Java projects hosted on GitHub. The pull requests examined were selected randomly and may not be representative of other pull requests. To mitigate, we chose the 23 most-forked Java projects on GitHub so that we could examine a diverse range of issues and reviewer comments. Furthermore, we only ran PMD on projects developed in Java; our results might not be representative of projects written in other languages.

For the manual comparison of reviewer comments with PMD reports, we minimized subjectivity issues by employing the double verification technique. Disagreements and discrepancies were rare and resolved by discussing the issues.

We designed our study around the idea of displacing effort from the code reviewer to the contributor by suggesting the PMD static analysis tool be invoked prior to pull request submission. However, possibly due to the experimental configurations, the overlapping comments map to a small percentage of the PMD warnings; of the 735 total PMD warnings, only 43 overlapped with reviewer comments. This added effort for the contributor is possibly amplified by the need to sift through possible false positives in the warnings. The overall impact on the code review process efficiency needs to be studied further.

VI. CONCLUSION

We present an empirical study of PMD and code reviews on 92 pull requests from 23 open source Java projects. Results show that PMD could find nearly 16% of all review comments. We conclude that PMD may prove to reduce the workload of code reviewers, in particular, six rules were always matched by reviewer comments. To reduce the number of false positives that the contributor sees from PMD, we propose to configure PMD rulesets such that it generates a limited set of warnings for each of its rulesets. Evaluating the effectiveness of this approach would be an interesting future work.

ACKNOWLEDGEMENTS

This work was supported in part by NSF Award #1645136.

REFERENCES

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.

[3] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721, 2013.

[4] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 931–940, 2013.

[5] T. Bruckhaus, N. Madhavii, I. Janssen, and J. Henshaw. The impact of tools on software productivity. *IEEE Software*, 13(5):29–38, 1996.

[6] M. Gegick and L. Williams. Towards the use of automated static analysis alerts for early identification of vulnerability- and attack-prone components. In *Proc. ICIMP*, pages 18–23, 2007.

[7] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355, 2014.

[8] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 358–368, 2015.

[9] D. Huizinga and A. Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[10] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, 2013.

[11] M. V. Mäntylä and C. Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[12] M. Michlmayr, F. Hunt, and D. Probert. Quality practices and problems in free software projects. In *Proceedings of the First International Conference on Open Source Systems*, pages 24–28, 2005.

[13] G. O'Regan. *Introduction to software process improvement*. Springer Science & Business Media, 2010.

[14] A. Reddy et al. Java™ coding style guide. *Sun MicroSystems*, 2000.

[15] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, 2013.

[16] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 34th International Conference on Software Engineering*, pages 255–265, 2012.

[17] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256, 2004.

[18] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 598–608, Piscataway, NJ, USA, 2015.

[19] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.

[20] Checkstyle. http://checkstyle.sourceforge.net/.

[21] FindBugs. http://findbugs.sourceforge.net/.

[22] PMD. https://pmd.github.io/.

[23] Review Board. https://www.reviewboard.org/.

[24] VMWare. http://www.vmware.com/.