Would Static Analysis Tools Help Developers with Code Reviews?

Sebastiano Panichella[1], Venera Arnaoudova[2], Massimiliano Di Penta[3], Giuliano Antoniol[2]
[1] University of Zurich, Department of Informatics, Switzerland
[2] Polytechnique Montréal, Canada
[3]Dept. of Engineering, University of Sannio, Italy

***Abstract*—Code reviews have been conducted since decades in software projects, with the aim of improving code quality from many different points of view. During code reviews, developers are supported by checklists, coding standards and, possibly, by various kinds of static analysis tools. This paper investigates whether warnings highlighted by static analysis tools are taken care of during code reviews and, whether there are kinds of warnings that tend to be removed more than others. Results of a study conducted by mining the Gerrit repository of six Java open source projects indicate that the density of warnings only slightly vary after each review. The overall percentage of warnings removed during reviews is slightly higher than what previous studies found for the overall project evolution history. However, when looking (quantitatively and qualitatively) at specific categories of warnings, we found that during code reviews developers focus on certain kinds of problems. For such categories of warnings the removal percentage tend to be very high, often above 50% and sometimes up to 100%. Examples of those are warnings in the *imports*, *regular expressions*, and *type resolution* categories. In conclusion, while a broad warning detection might produce way too many false positives, enforcing the removal of certain warnings prior to the patch submission could reduce the amount of effort provided during the code review process.**

***Keywords*—Code Review, Static Analysis, Mining Software Repositories, Empirical Study**

I. Introduction

Modern Code Reviews (MCR) [6] represent a less formal way of conducting code peer reviews, which has been a software engineering consolidated practice since several decades [4], [12], [21]. During code reviews, developers try to improve software quality in different ways, for example, fixing bugs or making the code easier to be maintained.

Developers often use tools that facilitate the code review process. *Gerrit* [2] for example, is one of the tool most used in open source projects for such purpose. When performing MCR, developers can also use other forms of support, for example checklists or organizational coding standards, or, very simply, just rely on their own experience. One category of tools that can serve as an aid to developers during code reviews are static analysis tools, which analyze either source code or bytecode (in the case of Java), as they are able to provide various sorts of warnings to developers. Warnings may be related to the software design (*e.g.*, high coupling between objects), code style (*e.g.*, missing space after a comma), documentation (*e.g.*, incomplete Javadoc comments), etc. Nevertheless, a weakness of these tools is that they might provide a too extensive list of recommendations, most of which might be irrelevant (and noisy) for developers. For this reason, previous work has studied how static analysis tool warnings [14] or vulnerabilities [11] are taken into account during software evolution. However, to the best of our knowledge, nobody has studied to what extent developers take care of such warnings during code reviews.

This paper aims at investigating (i) whether warnings raised by static analysis tools are removed during code reviews; (ii) what kinds of warnings developers pay more attention on during code reviews; and (iii) whether there is qualitative evidence of dealing with warnings in software repositories and/or reviews' comments. To this end, we use two static analysis tools—namely CheckStyle [1] and PMD [3]—and mine data from the Gerrit repositories of six Java open source software projects—in three of which developers actually used static analysis tools. While previous studies investigating the use of static analysis tools mainly focus on a quantitative analysis of how warnings evolve across the software evolution history [14], [11], this paper puts its attention on a specific phase—*i.e.*, code review rather than the whole development process. It also complements the quantitative analysis with a qualitative one, providing evidence that developers take actions to assure the removal of such warnings.

Results of this study indicate that, on the one hand, if we look at the overall set of warnings highlighted by static analysis tools, their density in the source code is only slightly reduced during each review, while the overall percentage of removed warning is slightly above (between 6% and 22%) what previous studies have found (about 10%) for bug fixes [14]. On the other hand, when performing a deeper analysis, we found that developers tend to focus their review activity on specific kinds of problems, and that such problems tend to be removed with percentages in many cases above 50% and in some cases 100%. In summary, if properly used, static analysis tools can provide a useful support during the development process. In other words, the removal of certain warnings before submitting a patch could help reducing the burden during the review process. Finally, developers can benefit from our analysis as it can be used to provide suggestions for improvement of their warning configurations, in case one exists, or for creating one, if none exists.

Replication package. A replication package is available online[1]. It includes (i) information about the path sets of the analyzed code reviews; (ii) comments posted during the

[1]http://ser.soccerlab.polymtl.ca/ser-repos/public/tr-data/2015-saner-code-reviews.zip.

code reviews and when uploading patches; and (iii) warnings detected by CheckStyle and PMD in the first and last patch set of all analysed code reviews.

Paper structure. Section II provides an overview of the capabilities and features offered by static analysis tools used in this study. Section III details the study definition, research questions, and planning, including the data extraction process. Section IV discusses the results, while threats to its validity are discussed in Section V. Section VI discusses the related literature concerning studies on code review, while Section VII concludes the paper and outlines directions for future work.

II. Static Analysis Tools

Static analysis tools perform analysis of source code or, in some cases, of byte code or binaries, with the aim of complementing the work of the compiler in highlighting potential issues that can arise in a software system. Examples of issues are related to uninitialised or unused variables, empty catch blocks, but also poorly commented and organized code (too long lines or methods). In this work we consider two widely adopted static analysis tools, namely, CheckStyle [1] and PMD [3]. Their choice is motivated by different factors: besides being quite popular, the two tools differ in the kinds of analyses they perform, *i.e.*, CheckStyle focuses more on readability problems, whereas PMD tends to highlight suspicious situations in the source code. In addition, as we are only interested in partial source code, *i.e.*, source code that underwent a review, we used tools that do not require the code to compile. For this reason, tools such as FindBugs [13] that require the code to be compiled were not considered in this study.

Both CheckStyle and PMD are highly configurable tools that help users to enforce the adherence to coding standards by reporting violations to those standards. Such tools come with a set of predefined configurations some of which are general, while others can be customized for a particular project. For example, the *basic* category of PMD contains a collection of rules that report violations, or warnings, to general good practices. Examples of violations are the use of empty catch blocks and an inappropriate use of a for loop (instead of a while loop). Other rules in the *basic* category report possible errors in the code such as misplaced null check (*e.g.*, in a conditional statement the null check is placed after an access to the object) and jumbled incrementer (*i.e.*, when in two nested for loops the second loop increments the first index rather than the second). An example of a rule that must be configured for a particular project is the header rule of CheckStyle that ensures that a source file begins with the specified header.

For the purpose of this study, we analyzed only warnings that do not require customization and can be executed on any project. Tables I and II show the analyzed categories and the number of analyzed warnings in each category for CheckStyle and PMD, respectively. It is worth noting the diversity of the considered warnings: they spread from warnings regarding style (*e.g.*, whitespace), documentation (*e.g.*, annotations and comments), and naming conventions through warnings regarding the code (*e.g.*, possible coding errors, unused code, and code optimization) to warnings dealing with design (*e.g.*, defining an interface without any behavior and high coupling between objects).

TABLE I. CheckStyle - analyzed warnings.

Category	# warnings	Category	# warnings
Annotations	5	Metrics	6
Block Checks	5	Miscellaneous	13
Class Design	8	Modifiers	2
Coding	43	Naming Conventions	10
Duplicate Code	1	Regular expressions	1
Imports	7	Size Violations	8
Javadoc Comments	6	Whitespace	12

TABLE II. PMD- analyzed warnings.

Category	# warnings	Category	# warnings
Android	3	Javabeans	2
Basic	23	Junit	12
Braces	4	Logging-jakarta-commons	4
Clone	3	Logging-java	5
Code size	13	Migrating	14
Comments	3	Naming	20
Controversial	23	Optimizations	12
Coupling	5	Strict exception	12
Design	54	Strings	16
Empty	11	Sun security	2
Finalizers	6	Type resolution	4
Imports	6	Unnecessary	7
J2ee	9	Unused code	5

III. Study Definition and Design

The *goal* of this study is to analyze code reviews, with the *purpose* of understanding how static analysis tools could have helped in dealing with warnings developers solved during code reviews. The *quality focus* is, on the one hand, software maintainability and comprehensibility that code reviews aim at improving and, on the other hand, reducing developers' effort during the code review task.

The *context* consists of two static analysis tools for Java, namely CheckStyle and PMD, and code reviews and history changes related to six Java open source projects, namely Eclipse CDT, Eclipse JDT Core, Eclipse Platform UI, Motech, OpenDaylight Controller, and Vaadin. Eclipse CDT is an Integrated Development Environment for C and C++, part of the Eclipse development environment, whereas Eclipse JDT core is the basic infrastructure of the Eclipse Java IDE, and Eclipse Platform UI provides the basic building block for building user interfaces within Eclipse. Motech is an open source suite for developing mobile health (mHealth) applications. OpenDaylight Controller is a component of an open platform to enable Software-Defined Networking (SDN), and Network Functions Virtualization (NFV). Vaadin is a framework to build Java Web applications. These projects were mainly chosen due to the availability of review information through Gerrit, but also, due to their different domain and size. Finally, and more important, we have evidence, for three of the projects, of the use of static analysis tools in the context of their development process. Specifically, while we did not find evidence that the three Eclipse projects use static analysis tools—either the tools we considered or other tools—OpenDaylight Controller and Vaadin used CheckStyle, while Motech used both. The evidence of the usage of such tools was found in the version control repositories of the projects with the presence of configuration XML files for the static analyzers. The main characteristics of the analyzed projects are reported in Table III.

TABLE III. CHARACTERISTICS OF THE ANALYZED PROJECTS.

Project	URL	Observed period	Size range (KLOC)	# of reviews analyzed	Uses CheckStyle	Uses PMD
Eclipse CDT	www.eclipse.org/cdt	2013-11-29–2014-09-22	1,500–1,550	309	✗	✗
Eclipse JDT core	www.eclipse.org/jdt/core	2013-06-24–2014-09-09	2,092–2,305	16	✗	✗
Eclipse Platform UI	www.eclipse.org/eclipse/platform-ui	2013-05-23–2014-09-24	2,736–2,554	113	✗	✗
Motech	http://motechsuite.org	2014-01-01–2014-09-24	149–171	161	✓	✓
OpenDaylight Controller	www.opendaylight.org/software	2013-07-04–2014-09-24	586–1,909	209	✓	✗
Vaadin	https://vaadin.com/home	2013-06-01–2014-09-24	6,174–6,114	180	✓	✗

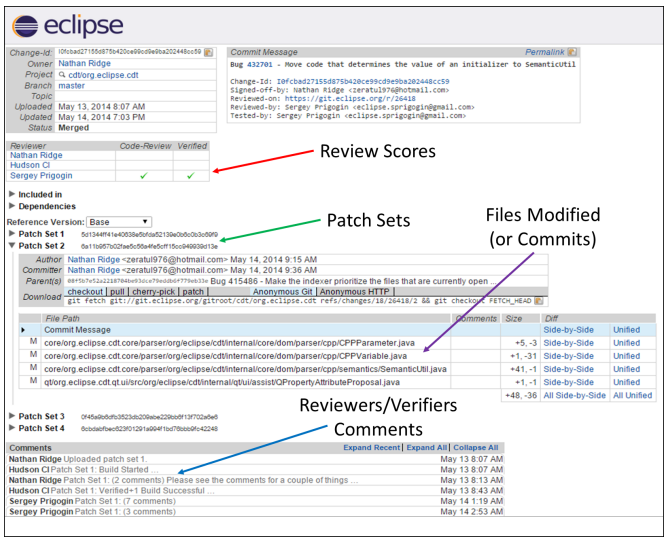

Fig. 1. Gerrit: Eclipse CDT - code review #26418.

This study addresses the following research questions:

- **RQ$_1$:** *To what extent warnings detected by static analysis tools are removed during code reviews?* The rationale of this research question is to analyze whether code reviews, overall, contribute to affect the presence of problems in the code highlighted by static analysis tools. In this circumstance, we do not distinguish among different kinds of warnings, but we rather observe the overall phenomenon.
- **RQ$_2$:** *What kinds of warnings detected by static analysis tool are mainly considered during code reviews?* This research question is a follow-up of the previous one. However, while in **RQ$_1$** we look at the overall variation of the warnings, here we study whether some categories are considered by developers more than others in the context of code reviews. We perform both quantitative and qualitative analyses.

A. Data Extraction

In the following, we describe how we extract data with the aim of addressing the research questions formulated above. Specifically, we describe (i) how we extract review information stored in Gerrit, and (ii) how we identify the presence of code issues using static analysis tools.

1) Review Analysis: Gerrit [2] is a tool, integrated with the git versioning system, that supports developers during the code review process. Gerrit allows developers to perform local changes (release of new patches) and then submit these changes in the Gerrit repository to be reviewed by other developers. During code review, developers can play the role of *reviewers* and of *verifiers*. *Reviewers* have the responsibility to give feedbacks and comments to improve the quality of the proposed patches. In addition, *verifiers* also evaluate whether the patches are really useful to fix problems/defects without breaking the behavior of the system. Thus, if patches meet the defined criteria, they are automatically integrated into the master repository and their status is changed to *merged*.

Similarly to Mukadam *et al.* [17] we collect the code reviews data from Gerrit for all projects considered in our study relying on the Gerrit API. The example in Fig.1 shows a review[2] for Eclipse CDT. In particular, each Gerrit review has an ID, a set of review scores, the patch sets, the set of files modified, and the reviews/verifiers comments. Specifically, a review change is represented by a set of files called *patch set* and corresponds to a commit. Until the patch set is not accepted, the author of the patch may submit several *patch sets*. The final and accepted version of the patch is the one that will be integrated in the master repository of the project. The data related to the patch sets allows us to study how warnings detected by static analysis tools change during code reviews for the files modified between the various releases of a patch (**RQ$_1$** and **RQ$_2$**). Reviewers/verifiers' comments can be of two types: (i) comments inline in the patch and (ii) general comments not included in the patch. We considered both of these comments for the qualitative analysis performed to address **RQ$_2$**.

Note that after collecting the code reviews, we filter out those that do not modify Java files as we only detect warnings in Java source code. In addition, we only retain those code reviews for which at least one Java file could be mapped between the initial and final patch sets. This filtering is needed because we are interested to compare the warnings detected in two versions of the same file—*i.e.*, the initial version that is submitted for review and the final version that is integrated in the master repository of the project.

2) Static Analysis: Using the process described above, we download from the Gerrit git the snapshot corresponding to the submitted patch—*i.e.*, the first patch in Gerrit that is submitted for review—and the one corresponding to the last commit in Gerrit—*i.e.*, the one that is integrated in the master repository of the project. Once such snapshots have been downloaded, we analyze, for both snapshots, the classes involved in the submitted patch using CheckStyle and PMD[3]. As mentioned in Section II, we executed the tools with the available warnings that did not require a customization for each project. For warnings that require thresholds we used the default values.

[2]https://git.eclipse.org/r/26418

[3]We used version 5.7 of CheckStyle and version 5.1.3 of PMD.

B. Analysis Method

To address $\mathbf{RQ}_1$, we perform two kinds of analyses. First, we study in detail what happened in each code review, *i.e.*, how warnings changed in such reviews. Then, we provide an overview, for each project and for each static analysis tool, of the percentage of warnings removed in the context of code review. To study in detail each code review, we compare the distribution of observed number of removed warnings. However, one limitation of such an analysis is that different reviews could impact code base of different size, and therefore involve a highly variable number of warnings. For this reason, we also compare the density of warnings in the source code before and after reviews. Given a review r_i, the warning density before the review wd_i^{pre} is defined as:

$$wd_i^{pre} = \frac{|w_i^{pre}|}{LOC_i^{pre}}$$

where $|w_{pre}|$ is the number of warnings a tool detects—in the system snapshot corresponding to the patch sent for review—limiting the attention to the classes affected by the patch only. LOC_i^{pre} is the size, in LOC, of the classes affected by the patch. Similarly, the warning density after the review wd_i^{post} is considered when applying the accepted patch to the system and is defined as:

$$wd_i^{post} = \frac{|w_i^{post}|}{LOC_i^{post}}$$

where $|w_i^{post}|$ and LOC_i^{post} are defined similarly to $|w_i^{pre}|$ and LOC_i^{pre}, with the difference that the measurements of warnings and LOC refer to the last commit of the review.
Once $|w_i^{pre}|$ and $|w_i^{post}|$ have been measured for each review, we compare their distributions across all reviews statistically using Mann-Whitney U test (two-tailed). We use such a non parametric test after verifying, using the Shapiro-Wilk procedure, that data significantly deviate from normal distribution. In addition to such a statistical comparison, we estimate the magnitude of the observed differences using the Cliff's d effect size measure.

To address $\mathbf{RQ}_2$, we study whether the observed differences in terms of warning density vary depending on the kinds of warnings (described in Section II) that CheckStyle and PMD detect. To this aim, we use the Permutation test, a non-parametric equivalent to the Analysis of Variance (ANOVA) [7] to check whether the density difference vary across different warnings. Wherever the permutation test indicates the presence of a significant difference, we identify which pair(s) of warning categories difference by using the Tukey's HSD (Honest Significant Differences) test [27]. This allows us to observe which category of warning is taken more into account during code reviews. Finally, in addition to the statistical procedures described above, we use box plots to depict the distributions of the differences observed for the different categories of warnings.
Besides the quantitative analysis described above, $\mathbf{RQ}_2$ is also addressed qualitatively by performing manual analysis, for each project and for each static analysis tool, on 10% randomly sampled code reviews that resolved at least one warning. Note that the goal of the manual analysis is not to validate the precision of static analysis tools but to gather qualitative evidence on the kinds of warnings that developers deal with during code reviews. Thus, we did not strive for a statistically significant sample. Moreover, in addition to the random sample, we also search in all reviewers' comments for such evidence. The manual analysis has been performed by relying on different sources of information, namely (i) the changes performed during the reviews, (ii) the output of the static analysis tools on the first and last versions of the patch, and (iii) the reviewers' comments.

TABLE IV. CHANGES IN CHECKSTYLE WARNING DENSITY AND NUMBER DURING CODE REVIEWS.

	Density of warnings		# of warnings	
Project	p-value	Cliff's d	p-value	Cliff's d
Eclipse CDT	0.028 (*)	0.002	0.009 (*)	-0.002
Eclipse JDT core	0.351	-0.008	0.624	0.004
Eclipse Platform UI	0.011 (*)	0.000	0.200	0.002
Motech	0.614	-0.010	3E-5 (*)	-0.010
OpenDaylight Controller	0.205	0.012	1.95E-4 (*)	-0.009
Vaadin	0.148	-0.002	0.209	0.000

TABLE V. CHANGES IN PMD WARNING DENSITY AND NUMBER DURING CODE REVIEWS.

	Density of warnings		# of warnings	
Project	p-value	Cliff's d	p-value	Cliff's d
Eclipse CDT	0.074 (.)	0.004	0.025 (*)	-0.001
Eclipse JDT core	0.450	0.012	0.919	0.000
Eclipse Platform UI	0.132	0.002	0.857	0.000
Motech	0.080 (.)	0.003	8.56E-5 (*)	-0.009
OpenDaylight Controller	0.005 (*)	0.025	0.002 (*)	-0.004
Vaadin	NA	0.000	NA	0.000

IV. ANALYSIS OF THE RESULTS

In the following, we report results of our study, with the aim of answering the research questions formulated in Section III.

A. RQ_1: To what extent warnings detected by static analysis tools are removed during code reviews?

Tables IV and V report, for all projects and for the two tools (CheckStyle and PMD respectively), the results of the paired Mann-Whitney test and the Cliff's delta (d) effect size obtained when comparing the distribution of warning density, as well as absolute number of warnings, for all the analyzed code reviews. Significant and marginally significant p-values are shown with (*) and (.), respectively. We observe a statistically significant difference between the density of CheckStyle warnings in the first and last patch of the analyzed code reviews for Eclipse CDT and Eclipse Platform UI. For PMD, results are statistically significant for OpenDaylight Controller and marginally significant for Eclipse CDT and Motech. No change in the density is observed for Vaadin. In all cases, the observed effect size is negligible.

Fig. 2 shows the density of CheckStyle (white) and PMD (grey) warnings in the first patch sets for the different projects. Comparing, on the one hand, projects that use static analysis tools and, on the other hand, projects that do not use such tools, there is no noticeable difference in the density distribution of warnings in the first patch set between the two groups of projects—a Mann-Whitney test comparing the two groups of

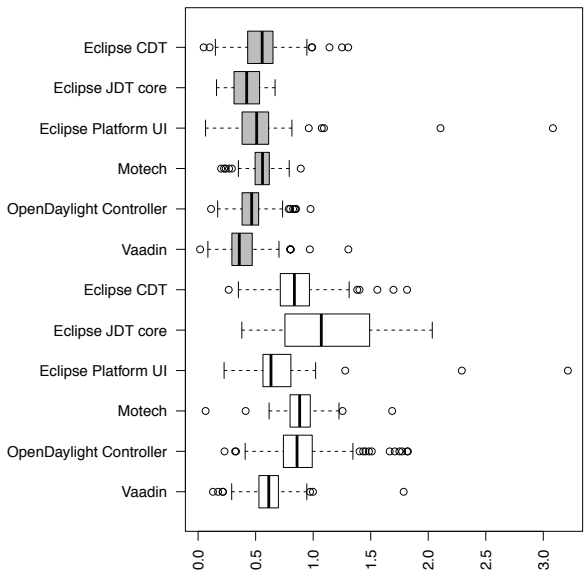

Fig. 2. Density of CheckStyle (white) and PMD (grey) warnings in the first patch sets.

projects was not significant ($p-value = 0.5038$). Thus, based on the available evidence, we cannot conclude that projects using static analysis tools analyse the source code prior to a code review. When comparing the cumulative percentages of fixed warnings during code reviews (Tables VI and VII), interesting insights emerge. First, as it can be noticed from both tables, the observed percentages are inline with (and often higher than) results of previous studies—showing that during software evolution about 10% of warnings are being removed [14]. Also, as it can be noticed, such percentages tend to be particularly high (at least higher than other cases) for projects that relied on static analysis tools during their evolution. For example, for CheckStyle warnings we observe 15.06% for Motech, 12.94% for OpenDaylight Controller, and 13.02% for Vaadin, while for PMD we observe 15.18% in the case of Motech. The only exception is represented by OpenDaylight Controller, exhibiting an overall removal of 22.78% for PMD warnings even if such a tool was not used. None of the PMD warnings were resolved in the Vaadin project; recall that the project does not use the tool.

TABLE VI. CHECKSTYLE - CUMULATIVE PERCENTAGES OF REMOVED WARNINGS.

Project	# of warnings in first patch set	# of warnings in last patch set	Percentage of resolved warnings
Eclipse CDT	21,087	18,684	11.40%
Eclipse JDT Core	3,260	3,045	6.60%
Eclipse Platform UI	4,409	4,009	9.07%
Motech	11,162	9,481	15.06%
OpenDaylight Controller	26,774	23,310	12.94%
Vaadin	484	421	13.02%

RQ1 summary: On the one hand, the density variation as well as the absolute variation of warnings in each code review is very small and in many cases not statistically significant. On the other hand, when looking at the cumulative percentage of removed warnings, such percentages are slightly higher than findings of previous studies and higher for projects using static analysis tools.

TABLE VII. PMD - CUMULATIVE PERCENTAGES OF REMOVED WARNINGS.

Project	# of warnings in first patch set	# of warnings in last patch set	Percentage of resolved warnings
Eclipse CDT	13,652	12,112	11.28%
Eclipse JDT Core	1,382	1,309	5.28%
Eclipse Platform UI	2,587	2,295	11.29%
Motech	8,649	7,336	15.18%
OpenDaylight Controller	9,201	7,105	22.78%

B. RQ_2: What kinds of warnings detected by static analysis tool are mainly considered during code reviews?

Figs 3– 6 show the distributions of the log differences in the density of warnings between the first and last patch sets. Due to the lack of space, we do not show boxplots for all projects. In the following we discuss the results for each project. Wherever we mention the presence of statistically significant differences, these refer to p-values < 0.05 obtained by the Tukey's HSD test (significance level $\alpha = 5\%$). Such test was executed after the permutation test indicated that in all projects the density of warnings is significantly different for different categories of warnings (p-values < 0.001). Due to limited space, we do not report detailed results of permutation test and Tukey's HSD.

a) Eclipse CDT: Warnings from the *regular expressions* category are the ones with the most tangible decrease of CheckStyle warning density. The difference with respect to all other categories is statistically significant and in terms of cumulative percentage, the fixed warnings represent 17% (349 out of 1,717). A warning from this category is for example one that detects trailing spaces at the end of lines. Although we did not find evidence that developers of Eclipse CDT rely on static analysis tools, our qualitative analysis confirms that developers of this project fix several issues highlighted by CheckStyle during code reviews. Developers seem to pay particular attention on the quality of the comments (fixed 161 out of 1,241 warnings from the *comments* category, *i.e.*, 13%). For example, in code review #20026 the first version of the patch contains method `EditorUtility.getTextEditor();` its javadoc does not provide a description of the parameter `IEditorPart editor` (the tag `@param` must be used for this purpose). The parameter description was added in the Javadoc in the last version of the patch. The CheckStyle warning `JavadocMethod` of the category *Javadoc comment* allows to identify such issues. Developers also fix 25% (73 out of 286) of the *duplicate code* warnings. For example, during code review #24522, one of the reviewers commented: *"Instead of sending commands directly to GDB, we should use the DSF services. In this case, IStack.getFrameData(). This is important to avoid duplicate code and enable caching"*. Developers removed the duplicated portion of the source code in the last version of the patch. Code duplication problems can be detected by CheckStyle, thus reducing the effort required by reviewers to find them manually.

The only statistically significant difference of the density of PMD warnings is observed for warnings from the *comments* category—statistically significant difference with half of the categories. However, in terms of cumulative percentage, we observe 100% removal for *imports* and *typeresolution* categories, where 19 and 9 warnings were fixed, respectively

(see Table VIII). Qualitative analysis also suggests that PMD may help developers with code reviews by highlighting several problems. For example, PMD detects for code review #23550 that the conditional statement `if(p != null && p instanceof ProcessFailureException)` needs to be simplified (*design* category). A reviewer commented: *"p != null part is redundant here... the second part of the condition covers the "null" case"*. Thus, in the last version of the patch the condition was simplified to `if (p instanceof ProcessFailureException)`. Moreover, PMD also highlights if statements that are not followed by braces (category *braces*). This is the case for an if statement in `PDOMExternalReferencesList.java` submitted for code review (#20972). The reviewer commented: *"If statements with 'else' branches should have curly braces around both branches"*.

TABLE VIII. ECLIPSE CDT - CUMULATIVE PERCENTAGES OF REMOVED PMD WARNINGS.

Category	# of warnings in first patch set	# of warnings in last patch set	Percentage of resolved warnings
basic	8	2	75%
braces	477	410	14%
codesize	82	34	59%
comments	1,750	1,559	11%
controversial	1,117	986	12%
coupling	3,800	3,424	10%
design	144	84	42%
empty	86	58	33%
imports	19	0	100%
j2ee	17	9	47%
javabeans	133	98	26%
junit	96	60	38%
logging-jakarta-commons	0	0	NA
logging-java	45	22	51%
migrating	0	0	NA
naming	876	749	14%
optimizations	4,015	3,704	8%
strictexception	0	0	NA
strings	761	725	5%
sunsecure	3	1	67%
typeresolution	9	0	100%
unnecessary	202	182	10%
unusedcode	12	5	58%

b) Eclipse JDT core: Fig. 3 shows that warnings from the *regular expressions*, *duplicate code*, and *annotations* categories of CheckStyle have the highest change in the density. The difference is statistically significant only for *regular expressions* and *duplicate code* and their respective cumulative percentages of fixed warnings are 31% (85 out of 276) and 72% (21 out of 29). An example of fixed warning from the *regular expressions* category can be found in code review #17926 where a reviewer commented in `DefaultCodeFormatterOptions.java`: *"... trailing whitespace must be removed"*. Another category from which developers resolved warnings is *miscellaneous* (34 out of 348, *i.e.*, 10%). For example, in review #6848 Checkstyle highlights an instance of `TodoComment` in `ReusableFileSystem.java`: *"// TODO: cleanup current instance"*. A reviewer, in accord with the identified warning, asks the author of the patch: *"Fix the TODO"*. Warnings belonging to the *imports* and *class design* categories are also fixed by developers. For example, CheckStyle highlighted a star import in `JavaCorePreferenceModifyListener.java` for code review #17299. Indeed, the initial patch set contained a starred import: `org.eclipse.jdt.core.*`. Since this import is actually using a single class of the package `import org.eclipse.jdt.core`, it was replaced with

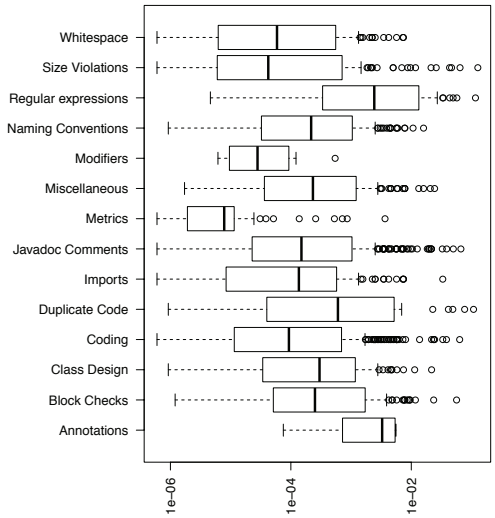

Fig. 3. Eclipse JDT core - density variation of CheckStyle warnings.

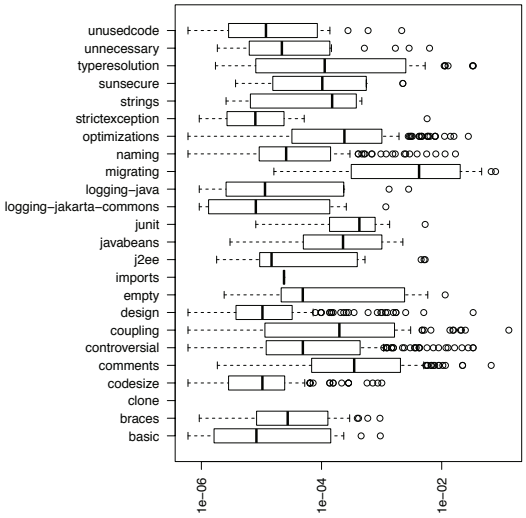

Fig. 4. Eclipse JDT core - density variation of PMD warnings.

`org.eclipse.jdt.core.JavaCore` in the final patch set. From Fig. 4 we note that the highest change of density of PMD warnings concerns the *migrating* category—the difference is statistically significant—and this category has a 100% cumulative fix (for a total of 60 warnings).

c) Eclipse Platform UI: Also for Eclipse Platform UI, the density of *regular expressions* warnings changed the most compared to the other categories of CheckStyle warnings; the cumulative percentage the fixed warnings represent 9% (93 out of 993). An example of fixed warning from this category can be found in code review #31002 where a reviewer commented on the need to fix trailing white spaces. Instead, in code review #24230 CheckStyle detected several warnings related to the *whitespace* category. Indeed, various reviewers' comments say: *"Whitespace please remove"*. The cumulative percentage of fixed warnings from this category is 44% (11 out of 25).
We observe a change in the density of PMD warnings in only few categories, namely *optimizations*, *design*, *coupling*, *con-*

troversial, and *comments*—there is no statistically significant difference among them. In terms of cumulative percentages we observe high numbers for *controversial* (41%, which is 18 out of 44 warnings) and *braces* (39%, 7 out of 18).

d) Motech: Fig. 5 shows that the highest change in the density of CheckStyle warnings is again from the *regular expressions* category—the difference is statistically significant and the percentage of fixed warnings is 100% (representing 13 warnings). Table X shows the CheckStyle and PMD configurations of Motech. We observe that *coding*, *naming conventions*, and *whitespace* are the categories with the highest number of configured CheckStyle warnings. A high percentage of the *whitespace* warnings are actually fixed during code reviews (89%, which represents 74 fixed warnings). The cumulative percentages for *coding* and *naming conventions* are 12% (531 fixed warnings out of 4,448) and 13% (145 out of 1,093), respectively. Interestingly, *regular expressions* have only 1 configured check (see Table X). An example of fixed warning from this category can be found in code review #4126. After the initial patch was submitted, the owner of the patch commented: *"Patch Set 1: Do not submit. I have a few trailing whitespace issues - will fix and upload a new patch."*
Design and *unused code* are the two categories from PMD with the highest number of configured warnings for Motech. The cumulative percentages of fixed warnings in those categories during code reviews are also high—77% (30 out of 39) and 100% (4 fixed warnings). An example of warning that is not part of the project configuration but that developers took care of during the code review process is *confusing ternary* from the *design* category. The rule recommends to avoid negation in an if statement when the else clause is present. In code review #3873, in `Lookup.java`, the code `lookupFieldDto.getId() != null` of the initial patch was replaced with `lookupFieldDto.getId() == null` in the final l patch and the else blocks were reversed. Adding *confusing ternary* would allow developers to take care of other instances of the warning.
Qualitative analysis of the code review comments also show that developers take care of warnings reported by static analyzers. For example, in code review #4383, one of the reviewers commented *"use spaces after commas (don't we have a checkstyle rule for that?)"*. The reviewer is actually right—there is a configured CheckStyle rule for missing spaces after commas but the severity of the rule is low—it is set to *info*[4]. Developers may benefit from updating the current configuration.

e) OpenDaylight Controller: None of the CheckStyle categories has a noticeable statistically significant difference of the delta density of warnings compared to the other categories. However, from Table IX we observe that for three categories, namely *metrics*, *modifiers*, and *regular expressions*, all warnings have been fixed—*i.e.*, 100% cumulative. Table XI shows the CheckStyle configuration of the project. From the configured checks, we observed that during code reviews developers fix trailing spaces and unused imports (e.g., in code review #11400). The number of configured warnings is not very high. Developers would probably benefit from extending their CheckStyle configuration. For example, checking for file headers may be useful as during code review #11048 a reviewer commented: *"I forgot to tell you, that all new files need to contain a copyright header."*. Three warnings are configured from the *imports* category, but they do not deal with the order of imports. The `ImportOrderCheck` could be added as instances of this warning (resolved in code review #9490, for example).
Fig. 6 shows the change in density of PMD warnings. We observe that warnings from the *migrating* category have an important statistically significant decrease in the density compared to the rest of the categories. Qualitative analysis also suggests that adding a PMD configuration may help developers. For example, many occurrences from the *optimization* category of PMD (*e.g.*, local variables and method arguments could be final) were fixed during code review #11048. Other examples are warnings from the *logging* category: The `GuardDebugLogging` warning recommends that *"when log messages are composed by concatenating strings, the whole section should be guarded by a `isDebugEnabled()` check to avoid performance and memory issues."* Multiple instances of this warning were fixed in `RaftActor.java` during code review #11182.

f) Vaadin: We observe a change in the density of warnings in only two of the CheckStyle categories—*Javadoc*

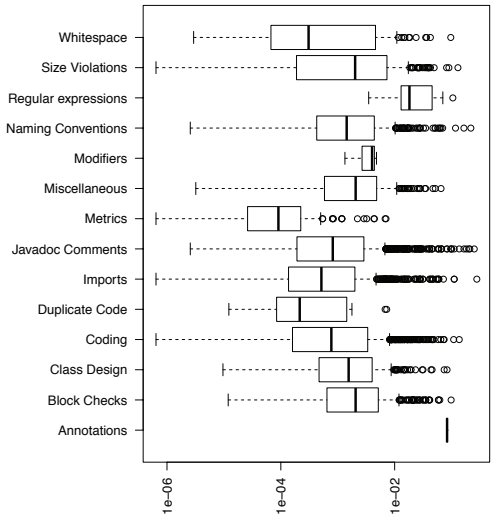

Fig. 5. Motech - density variation of CheckStyle warnings.

TABLE IX. OPENDAYLIGHT CONTROLLER - CUMULATIVE PERCENTAGES OF REMOVED CHECKSTYLE WARNINGS.

Category	# of warnings in first patch set	# of warnings in last patch set	Percentage of resolved warnings
Annotations	10	6	40.00%
Block Checks	2,423	2,173	10.32%
Class Design	262	165	37.02%
Coding	5,401	4,611	14.63%
Duplicate Code	31	18	41.94%
Imports	118	55	53.39%
Javadoc Comments	2,511	2,198	12.47%
Metrics	11	0	100.00%
Miscellaneous	9,914	9,141	7.80%
Modifiers	13	0	100.00%
Naming Conventions	1,525	1,285	15.74%
Regular expressions	27	0	100.00%
Size Violations	3,478	3,110	10.58%
Whitespace	1,050	548	47.81%

[4]CheckStyle provides four severity levels, namely, *ignore*, *info*, *warning*, and *error*.

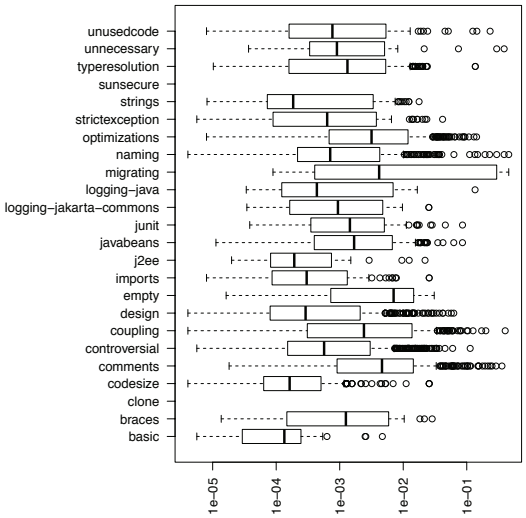

Fig. 6. OpenDaylight Controller - density variation of PMD warnings.

TABLE X. MOTECH - CHECKSTYLE AND PMD CONFIGURATIONS.

CheckStyle category	# warnings
Coding	17
Naming Conventions	10
Whitespace	9
Block Checks	5
Class Design	4
Miscellaneous	4
Imports	3
Size Violations	3
Metrics	2
Modifiers	2
Javadoc Comments	1
Regular expressions	1

PMD category	# warnings
Design	4
Unused code	4
Strict exception	3
Custom	2
Strings	2
Sun security	2
Code size	1
Coupling	1
Javabeans	1
Logging-java	1
Migrating	1

TABLE XI. OPENDAYLIGHT CONTROLLER - CHECKSTYLE CONFIGURATION.

CheckStyle category	# warnings
Imports	3
Coding	2
Miscellaneous	1

CheckStyle category	# warnings
Regular expressions	1
Whitespace	1

comments and *duplicate code*. In terms of cumulative percentages the fixed warnings correspond to 100% (4 fixed warnings) and 12% (59 out of 480 warnings), respectively. Table XII shows the CheckStyle configuration for the project. We observe that developers have configured a large amount of warnings from a large set of categories. In contrast to the other projects that we studied, we found very little quantitative and qualitative evidence that developers took care of the warnings reported by static analysis tools during code reviews. There is no change in the density of PMD warnings.

In the previous research question, we looked for a change of the density of all warnings that we consider in this work. One may argue that not all of those warnings are important for developers. As discussed in the above paragraphs, different projects stress on different categories of warnings. Thus, we perform the same analysis but this time we consider only the warnings that have been resolved in each project. In other words, we filter out warnings that are never taken care of during code reviews. Results are very similar to the results in Tables IV and V and thus we do not show them in details.

Finally, for the projects that have a CheckStyle and–or PMD configuration we perform a Mann-Whitney U test considering only the configured checks. Again results are similar to those reported in RQ_1 except for CheckStyle warnings in two projects. We found marginally significant difference for the number of warnings in the Vaadin project and statistically significant difference for the density of warnings for Motech—both with negligible effect sizes.

RQ_2 Summary: We found that some specific categories of warnings, for example *imports*, *regular expression*, and *type resolution* were in general removed more than others across all projects, and likely deserve more attention by developers. Noticeably, warnings from categories for which developers have specifically configured static analysis tools were largely removed. Thus, results confirm that static analysis tools can be used to support developers during code reviews. Enforcing the removal of certain warnings prior to submitting a patch for review could 1) reduce the time and effort provided by reviewers/verifiers and 2) speed up the code review process, thus, decreasing the time before a patch is integrated in the master repository of the project. To decide which categories of warnings need to be checked prior to a patch submission, developers can rely on prior warnings that have been removed—*i.e.*, using the historical data of the project—and on the preconfigured warnings of the project.

V. THREATS TO VALIDITY

This section discusses threats to the study validity. Threats to *construct validity* concern relationship between theory and observation, and are related to imprecisions in our measurements. We do not have issues concerning the presence of false positives in the tools' output, because we are interested to study warnings *as* they are highlighted by tools.

Threats to *internal validity* concern factors, internal to our study, that could have influenced the results. When we study how the density of warnings vary over time, we are aware that this can happen for various reasons, *e.g.*, because a source code fragment is removed when it is no longer needed. We supply to this limitation with the qualitative analysis carried out in $\mathbf{RQ}_2$, in which we perform a manual analysis on a subset of the reviews. Also, as mentioned in Section III, while we know that in some projects developers used static analysis tools, it is possible that some tools were also used in the other projects, *e.g.*, Eclipse, but we do not have trace of that.

Threats to *conclusion validity* concern the relationship between experimentation and outcome. While part of the analyses of $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ are supported by appropriate statistical procedures, other findings of $\mathbf{RQ}_2$ mainly have a qualitative nature, hence no statistical procedure is used.

Threats to *external validity* concern the generalization of our findings. Admittedly, the study is limited to six Java projects, three of which belonging to the Eclipse ecosystem. Although we are aware that further studies are needed to support our findings, our investigation was, intendedly, relatively limited in size to allow us to complement the quantitative analysis with a manual, qualitative analysis on 10% of the code reviews (for each project) in which developers remove

TABLE XII. VAADIN - CHECKSTYLE CONFIGURATION.

CheckStyle category	# warnings
Whitespace	11
Coding	10
Naming Conventions	9
Miscellaneous	6
Block Checks	5
Class Design	5

CheckStyle category	# warnings
Imports	5
Javadoc Comments	5
Size Violations	3
Regular expressions	2
Headers	1
Modifiers	1

warnings highlighted by static analysis tools. Also the choice of static analysis tools is limited to two of them, not only for simplicity's sake, but also for the need of analyzing the code without compiling it (required, for example, by FindBugs).

VI. RELATED WORK

This section discusses related literature about modern code reviews, as well as empirical studies investigating problems detected by static analysis tools in the software evolution.

A. Code Review

Rigby *et al.* [24], [23] empirically investigated the use of code reviews in open source projects. They identified several guidelines that developers should follow in order to make their patches accepted by reviewers. For instance, they suggested small, independent, complete patches. Weißgerber *et al.* [29] found that the probability of a patch to be accepted is about 40% and that, in line with Rigby *et al.* 's findings [25], [24], [23], smaller patches have higher chance of being accepted than larger ones. Baysal *et al.* [8] discovered that patches submitted by casual contributors are more likely to be abandoned (not reviewed) compared to the patches submitted by core contributors. Also, Baysal *et al.* [9] investigated factors that can influence the review time and patch acceptance. Their investigation highlighted that the most important factors are (i) the affiliation of the patch writer and (ii) the level of participation within the project. In relation to the above works, our study is relevant because the removal of certain warnings developers have to deal with during reviews would also contribute to the acceptance of patches.

Other studies focused on how developers perform code reviews. Rigby *et al.* [26] compared two peer review techniques, *i.e.*, RTC (Review Then Commit) and CTR (Commit Then Review), and discovered that (i) CTR is faster than RTC, and (ii) CTR is used mainly by core developers. Nurolahzade *et al.* [20] confirmed these findings and showed that reviewers not only try to improve the code quality, but they also try to identify and eliminate immature patches. Mantyla *et al.* [15] analyzed the code review process in both commercial and FLOSS projects, and observed that the type of defects that are discovered by the code reviewers are related, in the majority of the cases (75%) to non-functional aspects of the software. Nurolahzade *et al.* [20] showed that non-functional defects are hard to identify using automatic tools and thus, they should be identified through manual inspections performed by the owners of the modules. Bacchelli and Bird [6] studied the code review process across different teams at Microsoft by surveying developers and manually classifying review comments. They showed that the available tool support does not always meet developers' expectations. McIntosh *et al.* [16] discovered that the large degree of freedom that code reviewers have impacts modern reviewing environments and software quality. We share many findings with the above authors and focus specifically on work done during code review on aspects related to warnings highlighted by static analyzers. Related to what Bacchelli and Bird [6] found, our study indicates that (static analysis) tools can indeed be useful when properly configured.

B. Static Analysis Tools

The use of static analysis tools for software defect detection is becoming a common practice for developers during software development and has been studied by several researchers.

Kim and Ernst [14] studied how warnings detected by static analysis tools (JLint, FindBugs, and PMD) are removed during the project evolution history, finding that about 10% of them are removed during bug fixing, whereas the others are removed in other circumstances or are false positives. As a consequence, warning prioritization done by tools tend to be pretty ineffective, with precision below 10%. They suggested to prioritize warnings using historical information, improving warning precision in a range between 17% and 67%. While we share with Kim and Ernst the analysis of warning removal, our perspective is different and complementary to theirs. Rather than building a model to prioritize warnings, we deeply investigate—from a qualitative but also and above all from a qualitative point of view—what happens to warnings in the context of code reviews. A related analysis, focusing on vulnerability, was also performed by Di Penta *et al.* [11] who studied what kinds of vulnerabilities developers tend to remove from software projects. Our work differs from the work of Di Penta *et al.* as we focus on generic warnings instead of specific ones (vulnerabilities) and, as said before, we study warning removals in the context of code reviews.

Thung *et al.* [28] manually examined the source code of three projects to evaluate the precision and recall of static analysis tools. Their results highlight that static analysis tools are able to detect many defects but a substantial proportion of defects is still not captured. Nanda *et al.* [19] performed a similar study focusing on evaluating null pointer defects; they obtained similar findings. Zheng *et al.* [30] evaluated the kinds of errors that are detected by bug finder tools and their effectiveness in an industrial setting. The majority of the defects found are related to programmers' errors and some of the defects can be effective for identifying problematic modules. Ayewah *et al.* [5] showed that the defects reported by FindBugs are issues that developers are really interested to fix. Rahman *et al.* [22] compared statistically defect prediction tools with bug finder tools and demonstrated that defect prediction achieves (i) better results than PMD and (ii) worst results than FindBugs. Instead, Nagappan *et al.* [18] found that the warning density of static analysis tools is correlated with pre-release defect density. Moreover, Butler *et al.* [10] found that, in general, poor quality identifier names are associated with higher density of warnings reported by FindBugs.

VII. CONCLUSION AND FUTURE WORK

This paper empirically investigated how warnings detected by two static analysis tools—CheckStyle and PMD—are removed in the context of code reviews conducted on six Java open source projects, three of which (Motech, OpenDaylight

Controller and Vaadin) rely on static analyzers during the software development and three of which (Eclipse CDT, JDT core and UI Platform) do not. The analysis has been conducted from both (i) a quantitative point of view, by analyzing changes occurred during code reviews using data available through Gerrit and by running the analyzers on the first (initial) and last (accepted) change sets, and (ii) a qualitative point of view, by manually analyzing 10% of the code reviews (for each project) to investigate how and whether developers took care of warnings highlighted by static analyzers.

Results of the study indicate that the overall percentage of warnings removed during code reviews ranges between 6% and 22%, which is slightly higher but in line with the 10% that Kim and Ernst [14] found to be removed in the context of bug fixing changes. This is the result of a gradual removal of warnings, since we found that the warning density does not significantly decrease in each code review and that in some cases such an observed difference is marginally significant but in any case with a negligible observed effect size. However, by analyzing specific categories of warnings separately, we found that in many cases the percentage of removed warnings is particularly high, and above 50%. Examples of such warnings include the *imports*, *regular expression*, *type resolution* categories. Last, but not least, we found that projects using static analysis tools fixed a higher percentage of warnings than other projects.This result suggests that an appropriate configuration of static analysis tools—with the aim of highlighting only warnings relevant in a particular context—might help developers to focus on relevant warnings only without being overloaded.

Work-in-progress investigates to what extent enforcing the removal of certain warnings before submitting a patch can help reducing developers' burden during code reviews. Also, the analysis of past data from code reviews could possibly be used to support the configuration of static analysis tools towards warnings that are considered relevant in a particular project or organization. Finally, we plan to perform in-depth studies on specific kinds of problems, for example on how issues related to source code lexicon are dealt with during code reviews.

REFERENCES

[1] "CheckStyle," http://checkstyle.sourceforge.net, accessed: 2014-10-27.

[2] "Gerrit," https://code.google.com/p/gerrit/, accessed: 2014-10-27.

[3] "PMD," http://pmd.sourceforge.net, accessed: 2014-10-27.

[4] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: An effective verification process," *IEEE Software*, vol. 6, no. 3, pp. 31–36, May 1989.

[5] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2007, pp. 1–8.

[6] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721.

[7] R. D. Baker, "Modern permutation test software," in *Randomization Tests*, E. Edgington, Ed. Marcel Decker, 1995.

[8] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The secret life of patches: A firefox case study," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 447–455.

[9] ——, "The influence of non-technical factors on code review," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 122–131.

[10] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 156–165.

[11] M. Di Penta, L. Cerulo, and L. Aversano, "The life and death of statically detected vulnerabilities: An empirical study," *Information & Software Technology*, vol. 51, no. 10, pp. 1469–1484, 2009.

[12] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems*, vol. 38, no. 2-3, pp. 258–287, June 1999.

[13] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, December 2004.

[14] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2007, pp. 45–54.

[15] M. V. Mantyla and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 3, pp. 430–448, 2009.

[16] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 192–201.

[17] M. Mukadam, C. Bird, and P. C. Rigby, "Gerrit software code review data from android," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 45–48.

[18] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005, pp. 580–586.

[19] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making defect-finding tools work for you," in *Proceedings of the International Conference on Software Engineering (ASE) - Volume 2*, 2010, pp. 99–108.

[20] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal, "The role of patch review in software evolution: An analysis of the mozilla firefox," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, 2009, pp. 9–18.

[21] D. L. Parnas and M. Lawford, "The role of inspection in software quality assurance," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 674–676, 2003.

[22] F. Rahman, S. Khatri, E. T. Barr, and P. T. Devanbu, "Comparing static bug finders and statistical prediction," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014, pp. 424–434.

[23] P. C. Rigby, "Understanding open source software peer review: Review processes, parameters and statistical models, and underlying behaviours and mechanisms," Ph.D. dissertation, University of Victoria, BC, Canada, 2011.

[24] P. C. Rigby and D. M. German, "A preliminary examination of code review processes in open source projects," University of Victoria, Tech. Rep. DCS-305-IR, January 2006.

[25] P. C. Rigby, D. M. Germán, L. Cowen, and M. D. Storey, "Peer review on open-source software projects: Parameters, statistical models, and theory," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 35, 2014.

[26] P. C. Rigby, D. M. German, and M. D. Storey, "Open source software peer review practices: a case study of the apache server," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008, pp. 541–550.

[27] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[28] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2012, pp. 50–59.

[29] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2008, pp. 67–76.

[30] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 4, pp. 240–253, 2006.