Does code review really remove coding convention violations?

DongGyun Han[§] Amazon Germany

Chaiyong Ragkhitwetsagul SERU, Faculty of ICT Mahidol University Thailand

Jens Krinke University College London UK

Matheus Paixao University of Fortaleza Brazil

Giovanni Rosa University of Molise Italy

***Abstract*—Many software developers perceive technical debt as the biggest problems in their projects. They also perceive code reviews as the most important process to increase code quality. As inconsistent coding style is one source of technical debt, it is no surprise that coding convention violations can lead to patch rejection during code review. However, as most research has focused on developer's perception, it is not clear whether code reviews actually prevent the introduction of coding convention violations and the corresponding technical debt.**

Therefore, we investigated how coding convention violations are introduced, addressed, and removed during code review by developers. To do this, we analysed 16,442 code review requests from four projects of the Eclipse community for the introduction of convention violations. Our result shows that convention violations accumulate as code size increases despite changes being reviewed. We also manually investigated 1,268 code review requests in which convention violations disappear and observed that only a minority of them have been removed because a convention violation has been flagged in a review comment. The investigation results also highlight that one can speed up the code review process by adopting tools for code convention violation detection.

***Index Terms*—code review, coding conventions, coding style**

I. Introduction

The adoption of coding conventions, or programming style guidelines, is one of the most widely accepted best practices in software development. It is assumed that adherence to coding conventions increases not only readability [1], [2], [3] but also maintainability of software. Moreover, inconsistent coding style is one source of technical debt, specifically code debt. A survey of 682 developers [4] showed that technical debt is the biggest concern of developers and they spend 17% of their time addressing technical debt.

Code review, a popular method for maintaining high-quality software, is the process of conducting reviews of source code changes by developers other than the change author. During code review, a change in the form of a patch may undergo multiple revisions until it is finally accepted (or abandoned). Diverse empirical studies report benefits of code review, such as detecting defects [5], improving code quality [6], and sharing knowledge among team members [7]. Code review has widely spread in both open source projects and industrial projects [8], [9], [10], and has several supporting code review platforms such as Gerrit [11], CodeFlow and Fabricator. Moreover, the same suvery of 682 developers also showed that the adoption of code review had the biggest impact to code quality [4].

[§]DongGyun Han participated in this work prior to joining Amazon.

Tao et al. [12] reported that developers believe coding conventions are important criteria to apply in evaluating a patch, i.e., a code change, during code review. If a patch does not follow coding conventions, the patch will be rejected during the review. Furthermore, Tao et al. [12] observed that 21.7% of patches in Eclipse and Mozilla projects are rejected if they violate coding conventions (e.g., poor naming) or contain missing documentation). Their results show that developers are still struggling with checking coding conventions manually, although automated tools for checking coding conventions are available such as Checkstyle [13] and PMD [14]. Previous studies have reported that developers ignore about 90% of warnings generated by automated tools [15], [16].

Panichella et al. [16] investigated how coding convention violations flagged by automated tools disappear during code review. They mainly focused on how a patch under review affects the density of convention violations at the beginning and end of a code review. They found that the number of convention violations usually drops. However, they also found that the density of violations (i.e., number of violations per line) does not significantly decrease in each code review. Although Panichella et al. perform a manual investigation of 10% of code reviews for which a violation disappeared, their paper does not investigate whether the violations disappear because of the reviewers' comments. Instead, all disappearing violations are assumed to be resolved. Moreover, their approach compared the number and the density of violations in the files affected by the patch under review in the version after the initial and after the final patch. However, due to the effect of rebasing, the number and density of violations can be different also in the parts of the files that have not been changed by the patch. Rebasing a patch is necessary if the branch it is relative to has progressed in the development, so that the patch is relative to the current state of the branch. Paixao et al. [17] demonstrate that empirical studies which do not account for rebasing may report skewed, biased and inaccurate observations, as on average 34% of all reviews are affected by modifications in the reviewed files due to rebasing. Since Panichella et al. do not take rebasing into account, their observations and conclusions may be affected.

The purpose of this paper is to complement Panichella's results by (a) taking the effects of rebasing into account and (b) distinguishing violations that are resolved in response to reviewers' comments from violations that disappear coincidentally. We used the code review open platform (CROP) data [18], a data set created from project repositories and their corresponding Gerrit code review repositories [11]. The CROP data contains code review data, the changes that were reviewed, and the entire source code for every patch revision which is usually not recorded in a Git repository. In total, we analysed 16,442 code review requests from four Java subprojects of Eclipse with a further detailed investigation of 1,268 review requests and 2,172 reported convention violations. It is important to note that every change to the Eclipse projects undergoes code review, i.e., only approved changes in their final version are present in the project's Git repositories.

The contribution of this paper is an in-depth investigation of coding convention violations during code review. We investigate what kind of coding convention violations are introduced, removed, and addressed during code review. Furthermore, we investigate whether manual coding convention checks done by reviewers delay the code review process. We also discuss possible explanations for our observations, based on our examination of coding conventions in the four open source projects and how automatic tools are used. The discussion highlights how developers adopt coding conventions and what difficulties they face.

II. Experimental Design

A. Research Questions

The goal of this paper is to investigate in detail how developers deal with coding convention violations in real-world software projects during code review. To this end, we want to answer the following research questions (the first two research questions correspond to the research questions of Panichella et al. [16]):

RQ1. *How many coding convention violations are introduced despite code review?* As a preliminary research question, we investigate how many convention violations are introduced in the initial (i.e., first) patch of a code review request and how many convention violations are introduced in the final (i.e., last) patch. If manual code review is effective, the final patch should have fewer introduced convention violations than the initial patch.

Note that we focus on mandatory code review, i.e., every change undergoes code review. Therefore, the convention violations that are still present in the final (approved) patch end up in the code base.

RQ2. *What kinds of convention violations are addressed during code review?* While RQ1 investigates how many convention violations are introduced in the initial and in the final patch, RQ2 investigates convention violations that are introduced in the initial patch but are no longer presented in the final patch – that is, introduced convention violations that have disappeared. We want to know how many convention violations disappeared because they are fixed according to a code review comment that points out a convention violation. A code snippet which contains a convention violation can disappear during code review for several reasons. For example, it may be deleted to fix a defect, not because of a coding convention issue. Thus, it is important to perform a detailed investigation and pin point the convention violations that are removed only because of code review comments. This research question identifies convention violations that are important to developers and flagged during code review. Then, they are fixed in the subsequent patches under review. In other words, we also want to identify the violation types that were never flagged by a reviewer.

To answer this research question, we need to extract the set of coding conventions that are perceived as being important to developers during code review and study such convention violations in more detail. Previous studies have reported that developers ignore about 90% of warnings generated from automated tools [15], [16], and the answer to this research question can help to identify important violations so that unimportant violations can be filtered out, preventing 'Static Analysis Fatigue' [19].

RQ3. *Do convention violations delay the code review process?* As can be seen in Section IV and as also reported by others [15], [16], Checkstyle and other automated coding convention checking tools are not widely adopted in practice. However, if such tools were used only to check for violations that are important to developers, giving the tools a low false positive rate, their use could eliminate the need for human developers to check and fix violations and speed up the code review process. We investigate the delay caused by manual checking for convention violations that might be mitigated by adopting automated tools.

B. The CROP Data Set

To answer the research questions, we need not only to analyse source code for coding convention violations, but also to analyse every patch submitted for code review and investigate the review comments related to those patches. Instead of mining code review data ourselves, we used CROP [18], a data set that links code review data to software changes. All projects in CROP are open source and use Gerrit for their code review process. The data set contains code review data (e.g., description, changed files, and comments) and the complete code base before and after a change revision. Please note that the Git repositories of the projects contain only accepted revisions (i.e., the changes in the final patches in a code review request), whereas CROP stores all revisions.

We selected all projects that had Java as their primary language. The statistics of the four projects are displayed in Table I. The table shows the date of the first review ('Start Date') in the data set and the number of code review requests ('Reviews') for each project in the original data set. The last review dates in November 2017 are the same for all four projects. The table also shows the number of code review requests that were finally merged ('Merged Reviews') and how many of those code review requests consisted of more than a single revision ('Multiple Patches'). Only merged reviews with multiple patches are of interest to us because the initially submitted patch was revised based on reviewers' comments, and the patch was finally accepted and merged into the codebase. The last two rows show how many reviews a rebasing occured, i.e., a new revision of the patch under review is relative to a different commit than the previous revision of the patch, and how many reviews tampering occurs, i.e., when external commits due to rebasing modify the same files involved in the code review [17].

TABLE I
CODE REVIEW DATA SETS STATISTICS

Item	Platform UI	EGit	JGit	Linux Tools
Start Date	Feb'13	Sep'09	Oct'09	Jun'12
Reviews	4,756	5,336	5,382	5,105
Merged Reviews	3,802	4,502	4,463	4,559
Multiple Patches	2,985	2,899	2,533	4,301
Rebased Patches	2,798	2,464	1,941	3,141
Tampered Patches	593	1,060	845	533

C. Extracting Introduced Violations

Diverse automated tools have been introduced to support developers in adhering to coding conventions. To identify coding convention violations, we used a well-known tool, Checkstyle [13] version 8.8. Checkstyle is flexible and extensible, making it adjustable to any deviation from a Java coding convention. Checkstyle validates all Java source code against a set of rules (encoded conventions) and reports all cases in which the code is not compliant with the coding conventions, i.e., coding convention violations. Checkstyle can check for 153 different types of coding convention violations, grouped into 14 categories. However, only 62 conventions in 11 categories are enabled for Sun's Java coding conventions [20]. Based on the coding conventions of the four projects in our study, we set up our own Checkstyle configuration that complies with the conventions used by the projects. As the four systems have adopted coding conventions that have only minor deviations from Sun's Java coding conventions [20], only minor changes to the default configuration were necessary (e.g., tab width and maximal line length) [21][1].

First of all, we needed to understand what kinds of convention violations are introduced at the beginning of a code review request. Since analysis tools such as Checkstyle require the entire source code, we extracted violations from two different versions of source code: before and after applying the initial patch of a code review request. Figure 1 shows the process we used to extract violations that appeared or disappeared during code review. By comparing the lists of violations, we could extract those violations that were newly introduced by the initial patch of a code review request (Diff$_{initial}$). In matching the lists of violations, we ignored line numbers, file names, etc. because those elements may be changed by the

[1]Full info can be found on our study website: https://disclosed.after.review

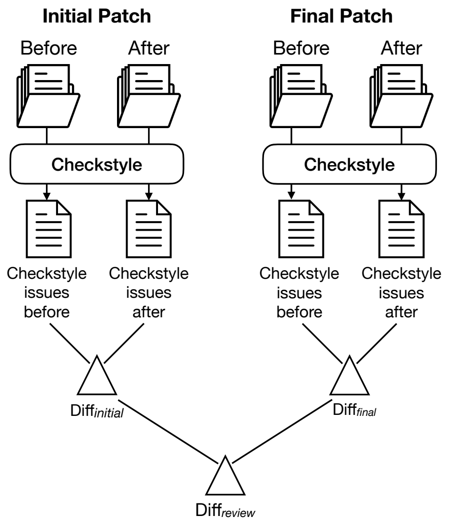


Fig. 1. Extracting introduced violations during code review by comparing violations before and after a patch

patch. Moreover, only the added—that is, newly introduced—violations were considered, because code review should prevent the introduction of convention violations. It is important to note that the system may have changed while the patch was being revised and the patch has been rebased. Therefore, the state before the final revision may be different from the state before the initial revision. Table I shows that not only rebasing is common, but also often affects the same files that are involved in the reviewed change. Changes in violations between the state before the initial revision and the state before the final revision need to be ignored. Therefore, we repeated the same steps as above on the final patch of a code review request to extract the list of introduced violations at the end of the code review (Diff$_{final}$).

The next step (RQ2) was to determine whether the newly introduced violations disappeared during the code review. A violation was considered to have disappeared if it was introduced in the initial patch but it was no longer present in the final patch. We determined what kinds of violations had been introduced in the initial patch (Diff$_{initial}$), and then disappeared in the final patch (Diff$_{final}$) by comparing the difference between Diff$_{initial}$ and Diff$_{final}$. The comparison result (Diff$_{review}$) contains the violations that were introduced (appeared) and deleted (disappeared) during the code review. For RQ2, only the disappearing violations are needed.

Table II shows the overall statistics for our violation extraction from code review data. The 'Unchanged' row presents the number of code review requests that have no violation changes between the initial and final patch (i.e., Diff$_{initial}$ = Diff$_{final}$). The 'Changed' row shows the number of code review requests that contain violation changes (violations either appeared or disappeared) between the initial and final patch (i.e., Diff$_{initial} \neq$ Diff$_{final}$). The '(Improved)' row states the number of code review requests containing violations that disappeared at the end of the code review.

TABLE II
OVERALL CONVENTION VIOLATION VARIATIONS FOUND BETWEEN INITIAL AND FINAL PATCHES OF A CODE REVIEW REQUEST.

Item	Platform UI	EGit	JGit	Linux Tools
Unchanged	2,636	2,138	1,799	2,886
Changed (Improved)	349 (199)	761 (375)	734 (397)	552 (314)

TABLE III
THE NUMBER OF MANUALLY INVESTIGATED CONVENTION VIOLATIONS AND THE NUMBER OF CONFLICTS BETWEEN HUMAN INVESTIGATORS. ALL CONFLICTS WERE RESOLVED BASED ON DISCUSSION.

	Platform UI	EGit	JGit	Linux Tools
Violations	313	622	640	597
Conflicts	16 (5.11%)	55 (8.84%)	80 (12.50%)	73 (12.23%)

From the 4,502 merged EGit reviews, we extracted Diff$_{initial}$ and Diff$_{final}$ for 2,899 code review requests that have multiple patches. Among the 2,899 reviews, 2,138 have no changes in violations, while only 761 reviews have differences in violations between the initial and final patch. A total of 375 reviews out of these 761 contain violation changes in which a violation was introduced in the initial patch but is no longer present in the final patch (i.e., the code review has improved the change).

Although we have the list of convention violations that appeared and disappeared during code review, we do not know whether a coding convention violation disappeared because it had been flagged by a reviewer during the code review and addressed by the patch author. For example, EGit has 375 improved code review requests based on the definition above. The 375 code review requests contain 622 violations that were introduced in the initial patch and disappeared in the final patch. Among these 622 violations, however, we do not know how many were addressed during code review (i.e., they did not coincidentally disappear because of other addressed issues). To determine whether reviewers detected and flagged the convention violations during code review, we manually investigated the review comments from the four projects for all patches of such reviews. We focused primarily on finding whether a reviewer's comment (e.g., '*please remove the trailing whitespace*') flagged the convention violation that appeared in Diff$_{initial}$, but is no longer present in Diff$_{final}$. If we could locate a comment flagging the convention violation, we labelled the violation as 'Confirmed'; otherwise, we labelled it as 'No Evidence'. To mitigate subject bias of the manual investigation, two of the authors (investigators) inspected the comments independently. They read the review comments and looked for mentions of the reported convention violations. Sometimes they also checked the patch to see how a violation disappeared. After finishing the independent rounds of investigation, the two investigators resolved all conflicts. They discussed a conflict until both agreed on the same label.

As shown in Table III, from 313 (Platform UI) to 640 (JGit) convention violations were introduced in the initial patch and resolved in the final patch of code review requests. Please note that we did not investigate code review requests that had only one patch or that had no convention violations introduced and removed (i.e., we investigated only those code review requests that introduced and removed at least one coding convention violation). As shown in the table, the investigators initially had conflicting results in 5.11%–12.50% of the cases and resolved the conflicts by discussing until both of them agreed.

III. RESULTS

The results from the above described experiment are used to investigate and answer the three research questions.

RQ1. How many convention violations are introduced during code review?

Figure 2 shows the accumulated numbers of violations appearing in the initial and final patch of a code review request for the four projects. The figure shows the number of violations for each violation category. There are 11 categories for 62 convention violation types overall. For example, a violation of type RegexpSingleline belongs to the Regexp category. The grey bar shows the number of violations in the initial patch, while the black bar shows the number of violations in the final patch. As shown in Figure 1, we ran Checkstyle before and after applying the initial patch and the final patch, respectively. By computing the differences between before and after applying a patch, we derived the list of introduced violations in the patch (i.e., Diff$_{initial}$ and Diff$_{final}$ for the violations introduced in the initial and the final patch, respectively).

Although the number of introduced convention violations in the final patch is larger than the number in the initial patch for all convention categories except for the Regexp convention category (as shown in Figure 2), a Mann-Whitney U test showed no statistically meaningful difference between the numbers of convention violations introduced between the two versions of the patch (p-value > 0.05) in each violation type.

One cannot conclude that code review introduces more violations than it prevents, because the number of violations naturally increases as the code base grows larger. Figure 3 shows the distribution of the numbers of added and deleted lines in the initial and final revisions of a code review request. We conducted a Mann-Whitney U test to determine the statistical difference between the initial and the final patches in terms of the numbers of added and deleted lines (i.e., the difference between the number of added lines in the initial and the final patches and also the difference between the number of deleted lines in the initial the and final patches). All four projects show a statistically meaningful difference between the initial and the final patches for both added and deleted lines (p-value < 0.05). As shown in Figure 3, the final patch usually has more added lines than the initial patch. This gradually increasing pattern is similar to the pattern shown in Figure 2. It shows that many violations appear in the initial patch of

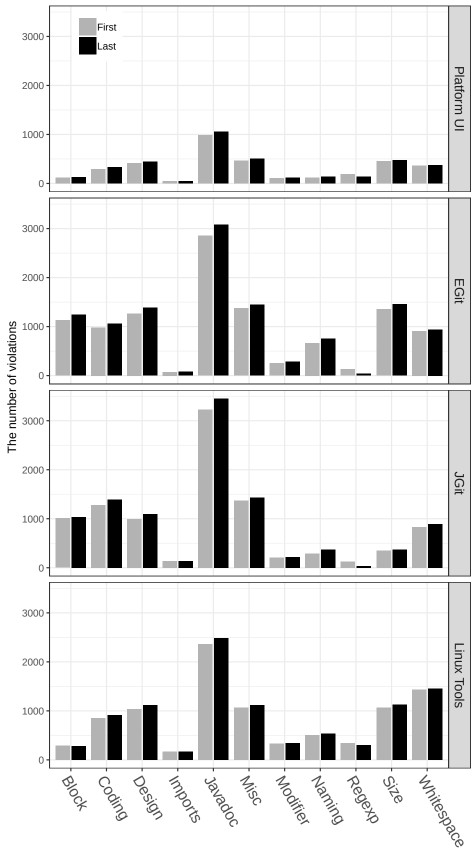


Fig. 2. The accumulated numbers of convention violations detected in the initial (grey) and final (black) patches

a code review request, the violations do not disappear, and more violations appear until the final patch, which is also usually larger than the initial patch. Therefore, the number of violations increases as the number of added lines increases.

Figure 4 shows how the number of appearing violations increases in the number of added lines during code review. For each code review request, we extracted the number of appearing violations in the initial and the final patches. The y-axis shows the difference (increase) from the initial to the final patches, and the x-axis shows the difference (increase) in the number of added lines between the initial and the final patches. We consider only the number of added lines, since deleted lines usually cannot introduce convention violations[2]. Please note the number of added lines can be negative, since the initial patch can add more lines than the final patch. Moreover, the initial patch can have more appearing violations than the final patch. Although there is a variance, this confirms that the number of violations generally increases according to the number of added lines.

Considering the different violation categories in Figure 2, the Javadoc category shows the largest number of violations in all four projects. This observation is in line with a previous study that showed that developers tend to treat missing documentation as a relatively less important issue than other conventions [12].

[2]It might be possible that deleting an existing Javadoc introduced a violation. During our manual investigation, however, we could not find any such case.

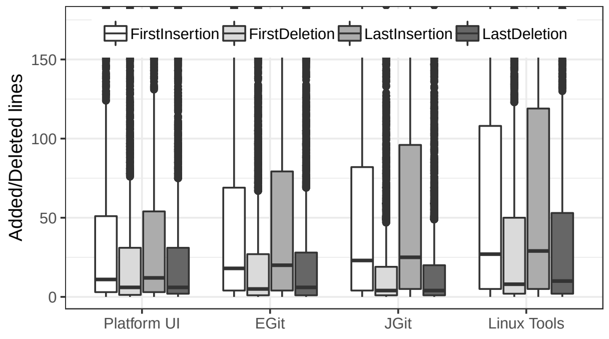


Fig. 3. Number of lines added or deleted in the initial and final patches

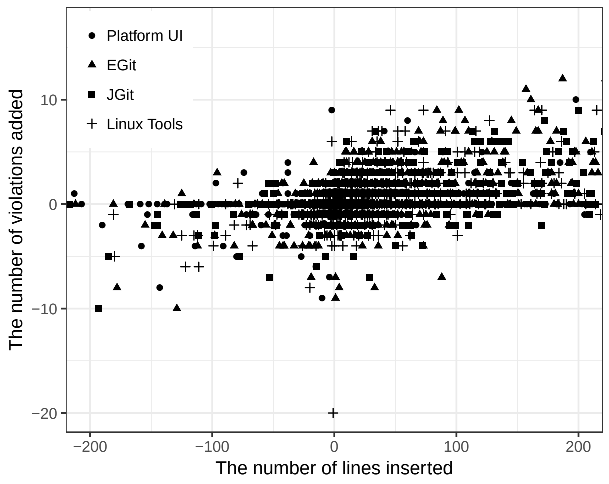


Fig. 4. The number of violations varies over the number of added lines between the initial and the final patches

The Regexp category shows the lowest number of violations in both initial and final patches. In addition, the number of introduced Regexp violations in the final patch is smaller than the number of violations in the initial patch (i.e., $|\text{Diff}_{initial}| > |\text{Diff}_{final}|$). This shows that the violations in this category may have been addressed during code review (i.e., spotted by reviewers, flagged, and fixed). The Regexp convention violation category contains the RegexpSingleline convention violation, a configurable check that is usually set for checking trailing whitespace (although most whitespace-related conventions belong to the Whitespace category). No other violation of the Regexp category has been reported. Thus, in this paper, the Regexp solely represents the trailing whitespace violations. We discuss the trailing whitespace issue in detail in Section IV-C.

Please note that the conventions in the Whitespace category provide finer-granularity checks for whitespace (e.g., GenericWhitespace checks the whitespace around the Generic tokens '<' and '>'). On the other hand, the convention violations might simply have disappeared when issues other than convention violations were addressed (e.g., bug fixes or removal of redundant code). We therefore investigate in RQ2 whether a convention violation disappeared because it had been addressed during the code review.

To answer RQ1, we found that for 10 out of 11 coding convention violation categories, the number of introduced violations in the final patch (after review) is larger than the number of introduced violations in the initial patch submitted for code review. The Javadoc category contributes the highest number of convention violations and the Regexp category, which represents trailing whitespace, has the lowest number of violations overall and has fewer violations in the final patch.

Furthermore, a key observation in this research question is that the number of newly introduced convention violations usually increases during code review as the size of the patches increases. While this key observation is not a surprising result, it suggests a correlation between the number of introduced violations and the size of a patch. This indicates that manual code review is not effective in preventing the introduction of convention violations, except for trailing whitespace.

RQ2. What kinds of convention violations are addressed during code review?

Figure 5 shows the number of code review requests in which the violations of a specific category disappeared during the code review between the initial and final patches. The reported numbers are the result of the manual investigation of all 2,172 disappearing convention violations, which affect 1,268 review requests. For 55 out of the 62 violation types, we found at least one disappearing violation.

The x-axis presents different categories of coding conventions, while the y-axis shows the number of violations for each category. The 'Confirmed' bars (grey) show the numbers of reviews in which a convention violation of a specific category appeared in the initial patch, the convention violation was flagged by a reviewer in a code review comment, and the final patch no longer contains the violation that appeared in the initial patch. The 'No Evidence' bars (black) show the numbers of reviews in which a convention violation of the specific category appeared in the initial patch, and the final patch no longer contains the violation, but in which the reviewers did not flag concerns about the violation (i.e., the investigators could not locate any code review comments regarding the violation). Mann-Whitney U tests for all four projects show statistically significant differences (*p-value* < 0.05) between the numbers of investigated violations and confirmed violations in each project. The statistical tests indicate that the numbers of violations confirmed in our manual investigation were not affected by the total number of investigated violations.

In 55 out of 62 convention violation types, at least one violation disappeared in the final patch. We could manually confirm that there was at least one violation flagged during the code review for 38 out of the 55 removed violation types. In other words, for 24 out of 62 violation types, not a single violation was flagged by a reviewer. It seems that developers did not care about such coding convention violations.

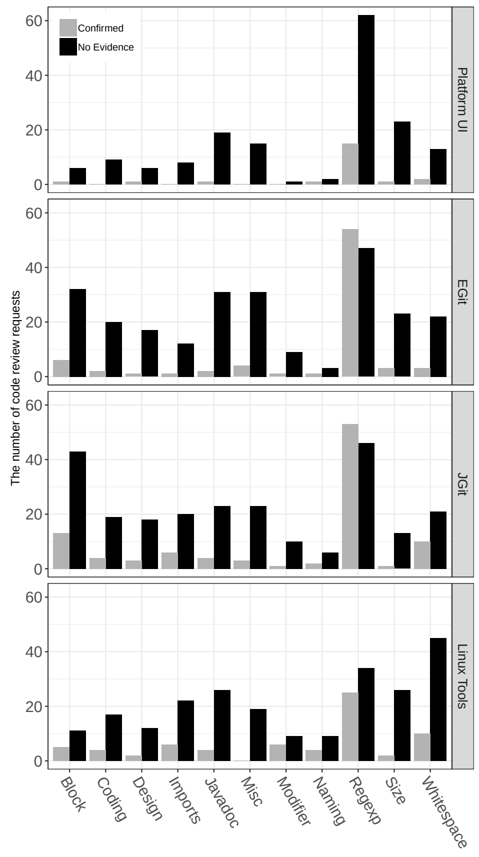


Fig. 5. The numbers of code review requests in which violations were addressed during code review ('Confirmed' – grey) and violations disappeared without evidence ('No Evidence' – black)

It is interesting to see that the majority of cases in which violations of a specific category were introduced in the initial patch but disappeared in the final patch were not due to reviewers flagging the issue ('No Evidence' in the figure). We found two scenarios illustrating why a violation disappeared even though no review comment had mentioned the violation. In the first scenario, a patch author changed a patch to address a reviewer's comment, and although the comment was not about the convention violation, the changed patch no longer contained the violation. For example, a reviewer points out a bug in a patch which also contains a convention violation. While fixing the bug, the patch author coincidentally removes the violation. Please note that if the reviewer had pointed out the convention violation in the comment, we would have labelled this occurrence as 'Confirmed'. In the second scenario, the change was not in response to a reviewer's comment, but the new patch no longer includes the convention violation. In this scenario, it is unclear why the patch has been changed. Most often this was due to self-review (i.e., the patch was created and reviewed by the same developer without a review comment).

For the 'Confirmed' category in Figure 5, it becomes clear that the problems of trailing whitespaces (i.e., Regexp) were pointed out the most by the reviewers. Trailing whitespace violations also disappeared during code review without being pointed out by reviewers ('No Evidence' category) the most for all the projects except Linux Tools.

For JGit and EGit, the second largest number of flagged and fixed convention violations was the Block category. This comes from the requirement that blocks always be enclosed in braces (i.e., the NeedBraces rule). As will be discussed in Section IV-D, since 27 January 2015, EGit and JGit force braces around single-line blocks, and not enclosing a block in braces became a violation. Before this update of the coding convention, among the 31 and 29 disappearing NeedBraces violations in EGit and JGit respectively, none were confirmed. After the new convention was adopted, 6 out of 7 disappearing NeedBraces violations were confirmed in EGit and 13 out of 27 were confirmed in JGit. For Linux Tools and Platform UI, the second largest number of flagged and fixed convention violations was the Whitespace category. It is also the third largest category for JGit and EGit.

For the violations that reviewers care about, one would expect that not only the violations introduced in the initial revision were removed in the final revision, but also no violation should further be introduced. However, for most of the violation types (33), this was not the case. There were only five violation types, in the four projects, that had fewer violations in the final revision compared to the initial revision. The RegexpSingleline violation (trailing whitespace) is the violation that the reviewers cared about most since it was the most often flagged and fixed in the four projects. Nonetheless, a significant number of this violation type still appeared in the final revision of a patch as can be seen in Figure 2. It seems that although reviewers cared somewhat about the 33 violation types, they did not take actions to ensure that no new violations occur during code review. They also did not apply consistent and rigorous checking.

One possible reason that coding convention violations are ignored is that flagging a coding convention violation requires the patch author to fix the violation and resubmit the patch for review. This delays the integration of the change and consumes more developer and reviewer time.

To answer RQ2, we found that the trailing whitespace (Regexp) is the convention violation that is most often flagged and fixed during code review, followed by surrounding a code block with braces (Block), and other convention violations regarding whitespace (Whitespace). We observed that many coding convention violations are ignored by developers and reviewers as we could only find 38 violation types in which a violation has been flagged and fixed during code review, i.e., only 38 out of 62 violation types were addressed.

RQ3. Do convention violations delay the code review process?

Although automated convention checking tools support developers by detecting convention violations instantly, they are often not adopted in practice. As a result, many convention violations are still detected through manual inspection by a reviewer. We investigate the delay caused by manually detecting a convention violation during code review.

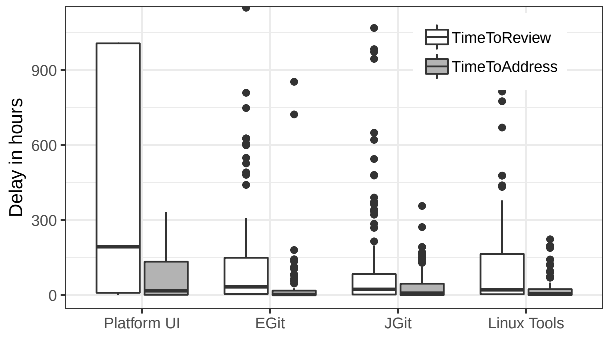


Fig. 6. Delay in detecting ('TimeToReview') a convention violation during code review and delay in addressing ('TimeToAddress') a convention violation

During the manual investigation for RQ2, we recorded the timestamps of review comments that flag convention violations and the timestamps of the subsequent patches. Figure 6 shows the time in hours, not all outliers are shown, from the introduction of a convention violation in the initial patch to a review comment that points out the violation (i.e., 'TimeToReview') and the time from the review comment to the next patch that removes the violation (i.e., 'TimeToAddress'). It takes more than 24 hours (the median is 24.21 hours) to receive a review comment that points out a convention violation. We found that it then takes more than 6 hours (the median is 6.90 hours) until the violation is addressed and fixed in the next patch. If developers used an automated convention checking tool such as Checkstyle, they could immediately detect violations and address them even before submitting a patch for review.

Discussion

The results for the three research question suggest some key observations about coding conventions during code review.

1) Convention violations accumulate as code size increases: As we presented in Section III, convention violations not only accumulate as code size increases, but the number of violations that a change introduces also increases during code review. The only exception over the four analysed projects was the Regexp category (i.e., trailing whitespace). The accumulated convention violations decrease code readability and maintainability, i.e., they become technical debt. Therefore, our results call for further studies on how to prevent accumulating convention violations as code size increases.

2) Developers manually check convention violations during code review: Although diverse convention checking tools are available, developers still manually check for convention violations during code review. Since code review is the gatekeeping process to assure code quality, the check for convention violations seems natural. Our results in Section III show that many convention violations are detected manually by developers, including very simple trailing whitespace violations. The results also show that humans are neither effective nor consistent in preventing the introduction of convention violations. For almost all violations that we confirmed of being flagged by a reviewer, we observed that the number of violations introduced by a change was higher at the end of the code review than in the initial version of the change. The notable exception was again trailing whitespace.

3) Convention violation checking delays code review: The answer to RQ3 highlights that the manual convention violation checking can cause delays during code review. Surprisingly, a patch author faces a median wait of more than 24 hours to receive a reviewer's comment flagging a convention violation. More importantly, even if the change is otherwise ready to be accepted, the presence of a convention violation flagged by a reviewer will not only require time to fix the violation but will also require another round of code review, including a CI run. We believe this delay can be reduced if developers can employ a trustworthy automated convention violation checking tool.

4) Comparison with Panichella et a. [16]: Our results differ significantly from the results of Panichella et al. [16]. They report a drop in violations while we report an increase in the results for RQ1. We are unable to find a convincing explanation for the difference. However, we speculate that it may be connected to the different approaches to measure the number of violations. While we extract the number of introduced violations in the initial and the final patch, Panichella et al. compare the absolute number of violations presented in the files affected by the initial and the final patch. They are likely affected by rebasing [17].

In the detailed analysis of violations that are disappearing between the initial and final patch, our results for RQ2 show that the majority of disappearing violations are not intentionally removed because of a reviewer's comment. Although mentioned in their discussion of threats to validity, Panichella et al. do not take this observation into account as they consider all disappearing violations as being 'resolved'.

However, there are some shared observations, e.g., we also observe that trailing whitespace was in general removed more than others. In Section IV-C we will discuss the case of trailing whitespace in more detail. Moreover, we share the suggestion that the adoption of automatic checking (and fixing) of convention violations can reduce the burden of reviewers and speed up the code review time.

IV. Coding Conventions in Practice

In this section, we discuss some examples of using coding conventions in practice from our observations of the four studied open source software projects and other related projects.

A. Adoption of Coding Convention Checking Tool

Our experiment uses Checkstyle, which is often used for convention violation checking. However, none of the four projects we investigated showed evidence that Checkstyle had been used. In May 2011, there was a discussion among Eclipse developers about adopting Checkstyle in their projects[3], but the tool does not seem to have been widely adopted. Instead, one advice is to use the Eclipse Formatter. A reason why Checkstyle is not introduced into existing codebases is that the source code does not adhere to coding conventions. As one Eclipse developer expressed it, '*I would never impose Checkstyle on an existing code base. Even in well behaved code one would get thousands (more likely tens of thousand) warnings/errors.*'.

According to JGit's and EGit's coding conventions, both projects integrate FindBugs [22], [23] and PMD's copy-paste-detector [14] as part of their build process. However, JGit developers currently do not use these tools.

The Eclipse Platform UI project has adopted SonarQube [24] to conduct code quality analysis. However, it does not seem to be used regularly, because, as of 9 January 2019, 32,796 issues had been reported[4]. 757 of them are categorised as critical. SonarQube was mentioned only once[5] during code review. This suggests that the SonarQube quality analysis may not be important enough to Eclipse Platform UI developers.

Although Checkstyle is not used in the Eclipse projects that we investigated, the Spymemcached project contained in the CROP dataset uses it. Spymemcached is a lightweight Java implementation of a memory caching system. The project was terminated and became the groundwork for the Java client for the Couchbase NoSQL database [25]. While the project is no longer maintained, we found an interesting case regarding coding conventions. At the beginning of the project, spymemcached developers did not employ any automated coding convention checking tools in their development process. Therefore, developers spent considerable time during code review discussing and fixing convention issues. In August 2011, the project integrated Checkstyle into its development process. As an integration step, a developer executed Checkstyle for the complete project code and fixed all the convention violations that were reported in a single commit.[6] Because of this 'big-bang' style change, most of the files in the code base were changed by a single developer in a single commit, and the change history was tainted. Developers can no longer track when the last change was introduced and who made the change in a file farther back than the 'big-bang' commit. To avoid this problem, the projects that we investigated have decided against fixing present convention violations. Moreover, we have seen during our investigation that code reviewers often reject changes that fix only convention violations. Instead, changes should not introduce new coding violations.

B. Adoption of Coding Convention Fixing Tool

Assuming that Eclipse developers themselves use Eclipse, it was surprising to see a large number of convention violations that could have been prevented by Eclipse's automatic formatting system. For example, the Eclipse Platform UI project has documented coding conventions together with instructions on how to adhere to them. It has adopted the Eclipse Coding Conventions [21], and it provides IDE configurations for

[3]https://bugs.eclipse.org/bugs/show_bug.cgi?id=347666

[4]https://sonar.eclipse.org/dashboard/index/eclipse.platform.ui:eclipse.platform.ui

[5]https://git.eclipse.org/r/#/c/65695

[6]http://review.couchbase.org/#/c/8644/

code formatting that a contributor needs to import. Clear instructions are given on code formatting:

> *Avoid formatting whole files – as this can generate pseudo-changes (whitespace related) when committing changes to existing source files. The easiest way, for Java files, is to have 'Format edited lines' activated*

Given these explicit instructions on how to use automatic code formatting, one would assume that violations to the corresponding documented conventions do not occur. As presented in the previous sections, however, there was an abundance of code reviews in which violations were discussed by developers. Some discussions even mentioned that the Eclipse automatic formatting system was not working correctly at some point.[7]

C. Trailing Whitespace

The introduction of trailing whitespace was the only convention violation that decreased in number during code review. One might conclude that trailing whitespace is a convention violation that developers are specifically interested in. A developer contributing to the Eclipse Platform UI even mentioned trailing whitespace during code review:

> *The general rationale behind coding style, namely improving readability, is very important to code reviews, because the main task of code reviews is to read code. In Eclipse Platform, my experience is that coding style is handled quite strictly. Even a trailing space at the end of the line can lead to a rejection of the change set and needs to be fixed in order to be included.*

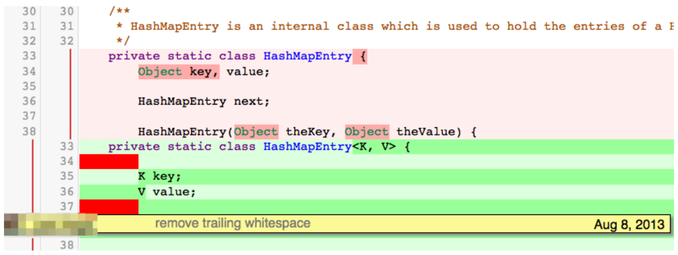
However, the explanation for the role of trailing whitespace may be much simpler because Gerrit's visualisation of changes highlights trailing whitespace in red. Figure 7 shows an example from the Eclipse Gerrit repository that illustrates how a reviewer[8] pointed out trailing whitespace in a change. The patch author introduced two meaningless trailing whitespaces (highlighted in red) while making a change. The reviewer pointed out the issue. The patch author revised the change and the trailing space disappeared in the later revision. Given the striking visualisation of trailing whitespace in Gerrit, it is no wonder that reviewers often flag the violation explicitly.

D. Enclosing Blocks in Braces in JGit and EGit

For JGit and EGit, the second largest number of flagged and fixed convention violations related to the requirement to always enclose blocks in braces (i.e., the NeedBraces violation in the Block category). Both JGit and EGit deviated from Eclipse coding conventions [26]. Their coding conventions flag two particular cases. First, they highlight that trailing whitespace should be automatically removed (similar to Eclipse Platform UI). Second, they discuss the need for braces around one-line statements. The second element conflicted with the Eclipse coding convention. Before 27 January 2015,

[7]https://bugs.eclipse.org/bugs/show_bug.cgi?id=477476

[8]The reviewer's photo and name are blinded for privacy.



```
    /**
     * HashMapEntry is an internal class which is used to hold the entries of a
     */
    private static class HashMapEntry {
        Object key, value;

        HashMapEntry next;

        HashMapEntry(Object theKey, Object theValue) {
    private static class HashMapEntry<K, V> {

        K key;
        V value;

```

[redacted] remove trailing whitespace Aug 8, 2013

Fig. 7. A reviewer pointed out trailing whitespace during code review. Gerrit highlights trailing whitespace in red.

EGit and JGit did not allow braces around single-line blocks, and many reviewers requested the removal of braces in cases where single-line blocks were enclosed in them. For example, in code review request #11558, the reviewer pointed out that the patch author should remove the braces around a single line. A proposal to change this deviating coding convention was flagged in Bugzilla (Bug #457592) by one of the main JGit/EGit developers.

> *I found this rule pretty annoying for many reasons: the rule itself is not only an exception from the general rule to have braces around any blocks but also contains another 2 exceptions that you *have to* use braces in some special cases. The code containing multiple if/else block with and without braces looks inconsistent, refactoring often leads to left-over braces.*

Based on this discussion, the developers updated the contributor guide [26]. The guide currently says:

> *Starting with 3.7.0 braces are mandatory independently of the number of lines, without exceptions. The old code will remain as is, but the new changes should use the style below:*

```
if (condition) {
    %doSomething();
}
```

> *The main reason for the change was **to simplify the review process**, coding guidelines and to make them more consistent with Eclipse code formatter.*

The developers updated their coding conventions to reduce confusion. However, we found that updating the coding conventions failed to reduce confusion because our investigation showed that the developers were still confused and spent manual effort detecting and fixing the violations of this convention.

Deviations from generally accepted coding conventions may lead to confusion and unnecessary discussions during code review and should be avoided. Removal of such deviations can also cause confusion and unnecessary discussions.

V. Threats to Validity

We selected four projects (i.e., Platform UI, EGit, JGit, and Linux Tools) from the Eclipse Foundation. Although the projects have large amounts of code review data, the results may not be generalisable to other commercial or open source projects. Moreover, all four projects use Gerrit as the code review platform, and this has only limited support for including automatic checking tools. We focused only on the initial and the final patch of a code review request. However, it is possible that we may have missed convention violations in intermediate patches. For example, a developer violated a coding convention in the second patch, a reviewer spotted it, and the violation was addressed in the third patch. However, we assume there will be no significant difference between the initial patch and the following patches in a code review request, since Eclipse and other open source and proprietary projects limit the size of patches [27]. The study is only based on the patches that Checkstyle reported coding convention violations. We did not investigate the reviews where there might be a discussion about convention violations but were not detected by the tool.

Since our investigation for RQ2 was a manual process, it risked of being subjective. To mitigate this, two authors investigated the data independently. If the two investigators found a conflict between their results, they discussed the conflict until they agreed on the same decision. The delay reported in this paper might not be representative, since a code review comment and the patch may address multiple issues at once. Therefore, the time we measured between them may not be solely spent on the checking and fixing of convention violations.

VI. Related Work

Panichella et al. [16] investigated how developers handle static analysis results such as coding convention violations during code review. While their approach to the analysis is very similar to the approach we use in this paper, their evaluation was limited and mainly focussed on quantitative analysis. While they manually investigated only a small sample of candidate reviews, we manually investigated all reviews. They concluded that static analysis tools can be used to support developers during code reviews. Our analysis similarly demonstrated that developers were not effective—and more importantly, not consistent—in detecting violations, suggesting that automated checking (and fixing) should be used to reduce the burden on reviewers and make the code review more rigorous in terms of catching violations that developers actually care about. In addition, we focused in detail on what violations are introduced and removed during the review, improving on the diversity of those already present.

Balachandran et al. [28] suggested the Review Bot, which is an extension of the Review Board and can recommend appropriate reviewers for a submitted review issue. The Review Bot uses line-level change history to determine the proper reviewer. The author evaluated his approach by using proprietary data from VMware. When the Review Bot recommended only one reviewer, its accuracy was 59.92%-61.17%. When it recommended five reviewers, it showed an 80.85%-92.31% accuracy rate. Similar to Balachandran's work, Henley et al. [29] integrated a CI tool, CloudBuild [30], that covers builds, test, and code analysis within a code review tool, CodeFlow. The authors showed that integrating static analysis tools within the code review leads to more communication between developers. The process increased coding convention discussions by about 50%.

Singh et al. [31] found that PMD can reduce the workload of code reviewers. Beller et al. [32] empirically found that static analysis tools are adopted in projects, but their use is not strictly enforced. Czerwonka et al. [33] reported benefits and costs of reviewing practices at Microsoft. They pointed out the high cost of code reviews and the fact that reviews are not always used efficiently. Vassallo et al. [34] investigated developers in both industry and open source projects who use static analysis tools. They observed that developers configure static analysers at least once, and the configuration is rarely changed during a project. They also stated that developers assign different priorities to warnings from static analysers based on different contexts. Later, Vassallo et al. [35] reported that the perceived relevance of static analysis tools varies between different projects and domains, showing that using static analysis tools is still not a common practice.

Zampetti et al. [36] empirically investigated the integration of static analysis tools and CIs. They found that a failure caused by a convention checker is one of the main reasons for build failure. Sarkar et al. [37] argued that human efforts imply mental fatigue, which causes an increase in coding convention violations. Smit et al. [38] examined whether convention adherence is a proxy measurement for maintainability. They observed that adopting coding convention checking tools does not lead to a reduction in the number of violations. Elish et al. [39] also show that Java programmers find it difficult to comply with coding conventions.

VII. Conclusion

In this paper, we described how developers handle coding convention violations during code review. First, we investigated how many convention violations were introduced in the initial patch of a code review request and disappeared in the final patch. Then, we investigated whether violations disappeared because of a fix in response to reviewer comments. We found that many coding convention violations are ignored by developers and reviewers. 24 out of 62 violation types did not have any violation flagged and fixed. Moreover, there were only five violation types for which there are fewer violations in the final revision compared to the initial revision. Our results indicate that humans are neither effective nor consistent in preventing the introduction of convention violations. Finally, the results show that it can take around 24 hours for a human reviewer to pick up coding convention violations, which can be detected instantly by an automated tool.

Our study calls for future work on coding conventions. For example, the majority of coding convention violations can easily be detected by automated tools, rather than by a reviewer's manual inspection. It is also important to improve tools to provide fewer false-positives (i.e., violation warnings that are unimportant or unnecessary) to developers. It is also necessary for the tools to analyse changed code only. Based on the results presented in this paper, automated tool support can save developers' time and boost development speed.

References

[1] T. Lee, J. B. Lee, and H. P. In, "A study of different coding styles affecting code readability," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 5, pp. 413–422, Sep 2013.

[2] L. Tysell Sundkvist and E. Persson, "Code styling and its effects on code readability and interpretation," Ph.D. dissertation, KTH Royal Institute of Technology, 2017.

[3] R. M. dos Santos and M. A. Gerosa, "Impacts of coding practices on readability," in *Proceedings of the 26th Conference on Program Comprehension (ICPC '18)*, 2018, pp. 277–285.

[4] *The ultimate guide to code reviews*, 2016. [Online]. Available: https://www.codacy.com/ebooks/guide-to-code-reviews

[5] M. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering*, 2009.

[6] 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 11th Working Conference on Mining Software Repositories (MSR '14)*, 2014, pp. 191–201.

[7] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. San Francisco, CA, USA: IEEE, 2013, pp. 712–721.

[8] J. Shimagaki, Y. Kamei, S. Mcintosh, A. E. Hassan, and N. Ubayashi, "A study of the quality-impacting practices of modern code review at Sony Mobile," in *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C '16)*, 2016, pp. 212–221.

[9] C. Sadowski, J. v. Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, 2015, pp. 598–608.

[10] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at Facebook," *IEEE Internet Computing*, 2013.

[11] M. Mukadam, C. Bird, and P. C. Rigby, "Gerrit software code review data from Android," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, 2013, pp. 45–48.

[12] Y. Tao, D. Han, and S. Kim, "Writing acceptable patches: An empirical study of open source project patches," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME '14)*, 2014.

[13] Checkstyle, "Checkstyle." [Online]. Available: http://checkstyle.sourceforge.net

[14] PMD, "PMD – an extensible cross-language static code analyzer." [Online]. Available: https://pmd.github.io

[15] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '07)*, 2007, p. 45–54.

[16] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER '15)*, 2015, pp. 161–170.

[17] M. Paixao and P. H. Maia, "Rebasing in code review considered harmful: A large-scale empirical investigation," in *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM '19)*, 2019, pp. 45–55.

[18] M. Paixao, J. Krinke, D. Han, and M. Harman, "CROP: Linking code reviews to source code changes," in *International Conference on Mining Software Repositories (MSR '18)*, 2018.

[19] J. Regehr, "Static Analysis Fatigue," 2010. [Online]. Available: https://blog.regehr.org/archives/259

[20] Sun Microsystems, "Code conventions for the Java programming language," 1999. [Online]. Available: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

[21] Eclipse, "Eclipse coding conventions." [Online]. Available: http://wiki.eclipse.org/Coding_Conventions

[22] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, 2007, pp. 9–14.

[23] 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 (PASTE '07)*, 2007, pp. 1–8.

[24] SonarQube, "Sonarqube." [Online]. Available: https://wiki.eclipse.org/SonarQube

[25] Couchbase, "Couchbase NoSQL database." [Online]. Available: https://www.couchbase.com/

[26] EGit, "Contributors' guide for Egit." [Online]. Available: https://help.eclipse.org/mars/topic/org.eclipse.egit.doc/help/EGit/Contributor_Guide/Contributing-Patches.html

[27] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proceedings of the 2008 International Workshop on Mining Software Repositories (MSR '08)*, 2008, pp. 67–76.

[28] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*, 2013, pp. 931–940.

[29] A. Z. Henley, K. Muçlu, M. Christakis, S. D. Fleming, and C. Bird, "CFar: A tool to increase communication, productivity, and review quality in collaborative code reviews," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*, 2018, pp. 1–13.

[30] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's distributed and caching build service," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE-C '16)*, 2016, pp. 11–20.

[31] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '17)*, 2017, pp. 101–105.

[32] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, 2016, pp. 470–481.

[33] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs: How the current code review best practice slows us down," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, 2015, pp. 27–28.

[34] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[35] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, pp. 1419–1457, Nov 2019.

[36] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. D. Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR '17)*, 2017, pp. 334–344.

[37] S. Sarkar and C. Parnin, "Characterizing and predicting mental fatigue during programming tasks," in *Proceedings of the IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion '17)*, 2017, pp. 32–37.

[38] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, "Code convention adherence in evolving software," in *International Conference on Software Maintenance (ICSM '11)*, 2011, pp. 504–507.

[39] M. O. Elish and J. Offutt, "The adherence of open source Java programmers to standard coding practices," in *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA '02)*, 2002.