Concerns identified in code review: A fine-grained, faceted classification

Sanuri Gunawardena[a,*], Ewan Tempero[a], Kelly Blincoe[b]

[a]*School of Computer Science, The University of Auckland, Auckland, New Zealand*
[b]*Department of Electrical, Computer, and Software Engineering, The University of Auckland, Auckland, New Zealand*

Abstract

Context: Code review is a valuable software process that helps software practitioners to identify a variety of defects in code. Even though many code review tools and static analysis tools used to improve the efficiency of the process exist, code review is still costly.
Objective: Understanding the types of defects that code reviews help to identify could reveal other means of cost improvement. Thus, our goal was to identify defect types detected in real-world code reviews, and the extent to which code review can be benefited from defect detection tools.
Method: To this end, we classified 417 comments from code reviews of 7 OSS Java projects using thematic analysis.
Results: We identified 116 defect types that we grouped into 15 groups to create a defect classification. Additionally, 38% of these defects could be automatically detected accurately.
Conclusion: We learnt that even though many capable defect detection tools are available today, a substantial amount of defects that can be detected automatically, reach code review. Also, we identified several code review cost reduction opportunities.

Keywords: code review, code inspection, concerns, types, defects, decisions, manual classification, detection method, detection expertise, non-programmers

1. Introduction

Code review, where code contributions of a software practitioner are evaluated by other members of the team, is a common practice in software teams [1, 2, 3]. This practice brings significant advantages such as improved code quality and knowledge sharing [4, 5, 6, 7, 8]. During code review, reviewers may identify functional issues with the code, suggest changes to the design, check for adherence to coding conventions, suggest alternate implementations, or simply provide praise [1]. Despite its popularity, a common complaint is the cost of performing code review, in particular the time it takes [1, 8, 9, 10, 11]. To this end, tools have been proposed that automatically provide reviewers the information they need [12], enable effective collaboration [13, 14, 15, 16], and detect defects using static analysis [17, 18, 19, 20]. Static analysis tools would typically be run prior to the code review so reviewers can focus on identifying more complex potential issues during code review. Despite the wide availability of such tools, developers have reported low adoption of these tools due to false positives and other barriers [21, 22] and complaints are still being made on the cost of code review [11].

Our goal is to further reduce the cost of code review without compromising its reliability. To this end, code review cost reduction opportunities could be identified through examination of defect types usually identified during code reviews. By understanding which defects are common, we may tailor cost-reduction approaches to such defects. Previous work has investigated defects identified during code reviews [23, 24, 25, 8, 26, 18, 27]. Several classifications of code review findings and changes have been proposed (e.g., [27] [24]). However, the current classifications are not fine-grained enough to enable identification of all automation opportunities.

For example, the comment "seems like these need

*Corresponding author
Email addresses:
`sanuri.gunawardena@auckland.ac.nz` (Sanuri Gunawardena), `e.tempero@auckland.ac.nz` (Ewan Tempero), `k.blincoe@auckland.ac.nz` (Kelly Blincoe)

some javadocs" extracted from CROP [28] refers to a set of missing Javadocs. This can be automatically detected by performing a text search to see if all methods include a Javadoc each. The comment "Javadoc? It looks like there's just enough here to pass checkstyle?", also from CROP [28] refers to missing Javadoc information. To determine that Javadoc lacks information, an understanding of the what the Javadoc is stating and an understanding of the related code and its purpose are essential. Thus, the detection of such concerns cannot be automated. However, both these defects fall into the same "Comments" class in the most fine-grained classification existing currently, CRAM [27]. This means that CRAM does not differentiate the detection automate-ability of defects. Thus, existing classifications do not provide an easy way to identify all code review cost reduction opportunities.

In this study, we develop a fine-grained classification for defects reported in code review comments with the goal of identifying automation opportunities. In addition, we examine which code review defects could have been automatically identified using popular existing tools to better understand the adoption of existing automation techniques. We focus on all *potential* problems with the code identified in code review comments that may or may not be actual defects because our goal is to understand the entire effort involved in code review, not just the comments that lead to code changes. Thus, in the remainder of this paper, we will refer to these potential problems as *concerns*.

Our study was guided by the following research questions:

RQ1: What types of concerns are identified in real-world code reviews?
RQ2: To what extent could the concerns identified during code review be automatically and accurately identified?

The "accurately" in RQ2 refers to detecting a concern without producing false positives and false negatives. We use "accurately" with the same meaning throughout the article.

We extracted and classified 417 comments from a subset of code review discussions of 7 open-source projects obtained from CROP [28], using thematic analysis [29]. The main contribution of this study is a detailed code review concern classification (see Figure 5) consisting of 116 concern types that are grouped into 15 concern groups.

To understand which of the code review concerns could have been identified using existing automated tools, we considered five popular static analysis tools and tested whether or not they could have automatically identified the concerns described in the code review comments. Our results suggest that many concerns (22%) could have been identified using existing automated tools, showing there is significant potential to reduce the effort of code review through better adoption of tools. The detection of another 16% of concerns were not supported by the tools we studied but also could have been automated accurately as new heuristics could be easily defined to enable their detection. In our discussion, we reflect on the types of concerns that are identified during code review and suggest future research avenues to improve the cost of code review including both automation and other ways.

The remainder of this paper is organized as follows: We describe the related work in Section 2. Section 3 describes our methodology and the research questions. Section 4 presents the results. Section 5 discusses the implications of our results, future research directions, and the threats to validity. Finally, Section 6 offers a brief conclusion.

2. Related Work

2.1. Modern Code Review

Modern code review (MCR) is a light-weight, tool-assisted, and asynchronous review process where code written by a developer is read through and evaluated by other team members [8, 1, 2, 3]. MCR tools highlight the differences between two versions of code and provide collaboration features. Popular MCR tools include Gerrit Code Review [13], Collaborator [14], Crucible [15], and Github Code Review [16].

The use of static analysis tools to reduce code review effort and increase code review quality has also been explored both in research and practice [17, 18, 19, 20]. There is a wide range of static analysis tools available (Eg: SonarQube [30], Checkstyle [31], FindBugs^{TM} [32], PMD [33], IntelliJ IDEA Code Inpection [34]) and they have a wide range of capabilities. Static analysis tools can either fully or partially automate the detection of defects, prior to code review. The extent to which the detection can be automated and the accuracy of detection, however, depends on the nature of the defect being detected.

A defect that can be well-defined, such as "trailing white-spaces" (unwanted whitespaces at the end of code statements) can be detected accurately using static analysis tools. False positives and false negatives are not produced in such cases. For example Checkstyle can detect trailing whitespaces using its "NoWhitespaceAfter" rule [35] which simply looks for the presence of a whitespace at the end of each code line. Thus, the detection of such defects can be *fully* automated.

Defects inherently manual to assess [36], such as "non-self-explanatory identifier names", do not have objective definitions, and cannot be accurately detected by tools; a tool can only detect a set of *potentially* bad identifier names, check an identifier name against a regular expression (class names should comply with a naming convention (SonarQube) [30], Type name(Checkstyle) [37], Class naming convention(IntelliJ) [38]), and recommend *potentially* better alternatives [39, 40, 41] but a human (i.e. the reviewer) must make the decision of whether the identifier name is bad enough to raise the concern. The detection of such defects can only be *partially* automated as described above.

Thus, it is impossible to completely remove human intervention in code review. However, human intervention can be reduced by identifying the concerns that can be fully detected using static analysis tools and giving the responsibility of such concerns to static analysis tools rather than reviewers. One of our attempts is to identify those defects and understand the role of automation in code reviews to pave the path to reducing code review cost.

2.2. The Modern Code Review Process

Here, we describe a typical modern code review process. A review starts when a developer modifies the original codebase in the repository and submits a new patch in the form of a commit. This person is typically called the *author* of the code. Next, one or more other developers of the project will review the submitted source code and provide feedback in the form of comments. These developers are the *reviewers*. Then, the author will modify the current revision of the patch if required, according to the review feedback and submit the improved code as a new revision. There may be multiple iterations of this step. If agreed, the proposed revision is merged into the code base. If rejected, the revision is abandoned.

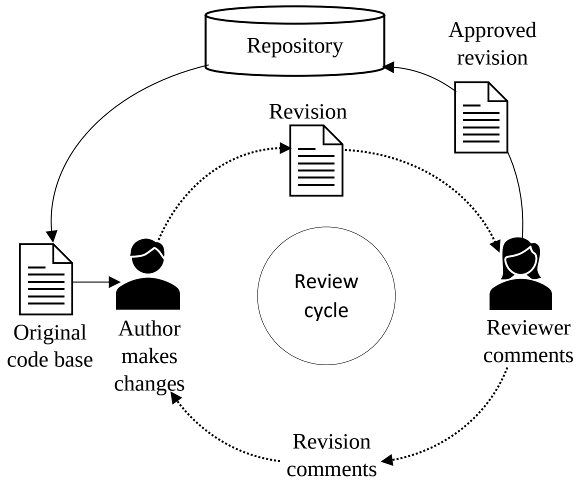


Figure 1: A typical modern code review process (adapted from Paixao et al. [28])

2.3. Defect Classifications

There are several existing *code review* defect classifications: Beller et al. [42], Runeson et al. [26], Bacchelli et al. [8], Lei et al. [25], Siy et al. [23], Mäntylä and Lassenius [24], and Panichella and Zaugg [27]. The two main goals of these classifications are to identify the value of code review and tool needs of reviewers. All of these studies categorize code review defects into broad categories, Panichella and Zaug's Code Review chAnges Model (CRAM) having the most fine-grained categories among the classifications. One of the lowest level categories of CRAM is "Comments", defined as "Explanations of complex code fragments, classes, methods. Issues include wrongly placed comments, missing comments, missing or wrong Javadoc, etc.".

The categories of these classifications do not provide a way to identify fine-grained automation opportunities, but only high-level tool needs. For example, a documentation comment that is unnecessarily placed on a private method can be detected by a simple text search and this opportunity for using automation cannot be identified by looking at the categories of CRAM or any other existing classification due to their high level of abstraction. Therefore, our classification was designed to carry concern categories that are further fine-grained so

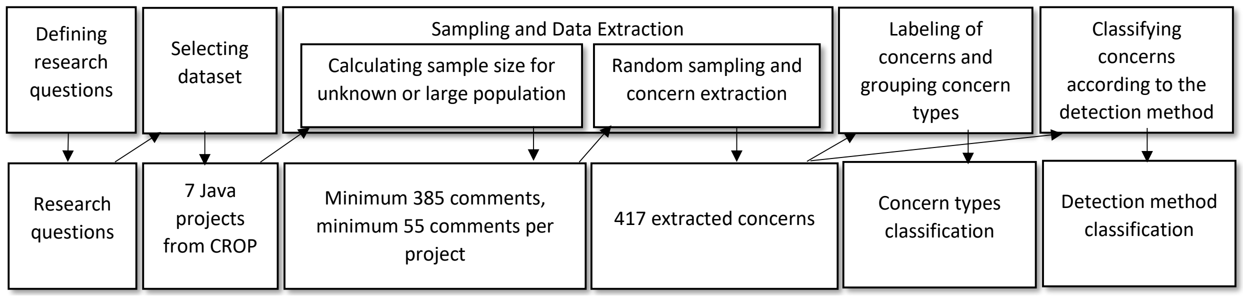

Defining research questions
Selecting dataset
Sampling and Data Extraction
Calculating sample size for unknown or large population
Random sampling and concern extraction
Labeling of concerns and grouping concern types
Classifying concerns according to the detection method
Research questions
7 Java projects from CROP
Minimum 385 comments, minimum 55 comments per project
417 extracted concerns
Concern types classification
Detection method classification

Figure 2: Methodology

that automation opportunities can be conveniently identified. Since CRAM is the most detailed classification existing, we compared our classification to CRAM in detail in section 5.4.

There are many other *manual* defect classifications available [43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58]. They extract defects from various resources such as test reports and customer feedback reports [53]. None of these classifications extract defects from code review data and thus do not qualify to answer our research questions. There are also many automatic defect classification methods that make defect classification efficient. They either use an existing classification such as ODC [59, 60], classify defects broadly as bug or other request [61, 62], or create their own classification [63, 64, 65]. Similar to the manual defects classifications we discussed before, none of these classifications either provide fine-grained defect details or classify defects based on their detection automate-ability.

Thus, the classification we created is the first of its kind. To properly encode data and to not be biased by the existing classifications, we designed our concern classification by applying thematic analysis to code review data instead of building on existing classifications.

There are several uses to the concern classification we created. It can be used as a reference to create organizational standards and checklists for future code reviews. Also, the classification provides a new way of viewing the code under review if rigorousness of the review is a goal: While reviewing code for all aspects of code can be overwhelming, considering a single concern group at a time modularizes and simplifies the code review process. Additionally, the classification we produced is the only existing classification of its kind that can potentially help to reduce code review cost effectively.

3. Methodology

This section describes the methodology we followed in our study. Fig.2 illustrates this process.

3.1. Research Questions

The focus of this study is to identify the types of concerns raised during MCRs and the part that existing static analysis tools can play in detecting concerns to identify code review cost reduction opportunities. Our research questions are inspired by this focus:

RQ1: What types of concerns are identified in real-world code reviews?
RQ2: To what extent could the concerns identified during code review be automatically and accurately identified?

3.2. Data Set

A formally published open-source data set of code review data intended for software engineering researchers and practitioners, "Code Review Open Platform (CROP)" was used for this study [28]. This data set contains data coming from the Gerrit Code Review and thus, review comments are available in both file-level and code line-level. This study analysed only the Java projects from CROP, which includes 7 of its 11 technical projects. Also, we analyzed code review comments on Java files only.

Fig.3 illustrates the structure of the CROP data set. At the highest level, it consists of a set of folders, each folder corresponding to a single *project*. Within each project folder, there is a set of subfolders, each corresponding to an entire code review *discussion* on a unique patch.

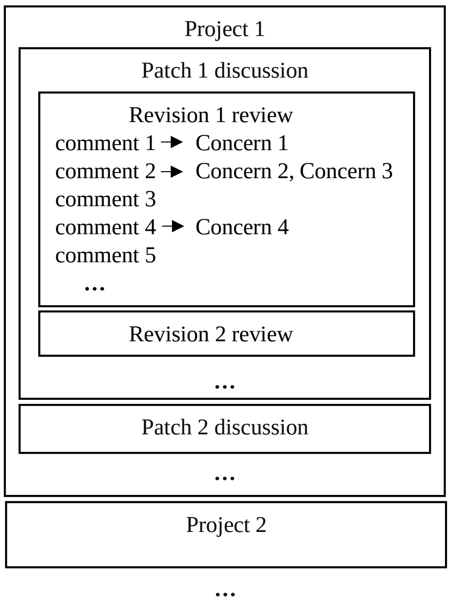


Figure 3: The Structure of CROP Data Set

A unique patch undergoes one or more conceptual revisions as a result of the code review process, before it is accepted or abandoned. Each revision of the patch produces a *revision review* file containing the author-reviewer communication made on the revision of the patch. Therefore, a single discussion folder contains one or more revision review files.

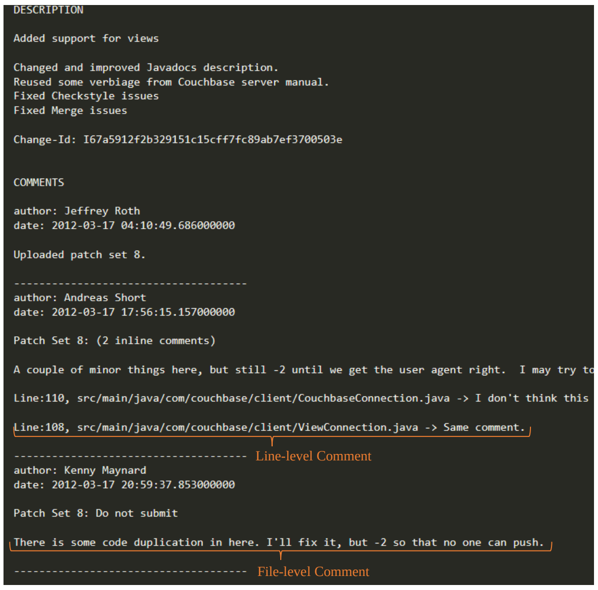
A revision review file may or may not contain *code review comments* based on the outcome of the code review performed on its patch revision. A code review comment is a comment made on a code line or a file in the patch by the reviewer. It may identify something that needs to be changed (e.g., a functional defect), question something that works as is but could be improved (e.g., a design defect or non-adherence to code conventions), or suggest alternatives (e.g., a different algorithm). A comment may also acknowledge a point raised by another reviewer, or praise the code being reviewed.

Our interest is in comments that *may lead to a change in the code*. As noted in the introduction, we examine these comments to identify *concerns*.

The CROP data set has a total of 50,959 code review discussions and 144,906 revision reviews, according to the CROP website [66]. The 7 projects we selected for this study had 22,859 code review discussions and 63,051 revision reviews. Table 3.2 summarizes the characteristics of the data available for these 7 projects in the CROP dataset [28] (population) and the sample we extracted as described below. The project descriptions were obtained from Paixao and Maia [67].

3.3. Sampling and Data Extraction

We calculated a sample size that would be representative of the concerns in the data set while still being reasonable for manual analysis. As described above, not all code review comments are describing concerns. An initial observation of discussions was conducted to get an idea of the density of comments that contain concerns. Only 29 out of 100 randomly selected comments were reporting concerns. Thus, it was not possible to calculate the exact population size of concerns in our data set. We, therefore, chose to be conservative and calculated a sample size for a unknown, large population size. Studies have shown that as the population increases, to obtain a sample with a confidence level of 95% and a margin of error of 5%, the sample size increases at a diminishing rate and remains relatively constant at 384.16 after a population size of 1,000,000 [68]. Thus, we considered 385 as the target sample size.


```
DESCRIPTION

Added support for views

Changed and improved Javadocs description.
Reused some verbiage from Couchbase server manual.
Fixed Checkstyle issues
Fixed Merge issues

Change-Id: I67a5912f2b329151c15cff7fc89ab7ef3700503e


COMMENTS

author: Jeffrey Roth
date: 2012-03-17 04:10:49.686000000

Uploaded patch set 8.

------------------------------------
author: Andreas Short
date: 2012-03-17 17:56:15.157000000

Patch Set 8: (2 inline comments)

A couple of minor things here, but still -2 until we get the user agent right.  I may try to

Line:110, src/main/java/com/couchbase/client/CouchbaseConnection.java -> I don't think this

Line:108, src/main/java/com/couchbase/client/ViewConnection.java -> Same comment.

------------------------------------
author: Kenny Maynard
date: 2012-03-17 20:59:37.853000000

Patch Set 8: Do not submit

There is some code duplication in here. I'll fix it, but -2 so that no one can push.

------------------------------------
```

Figure 4: File-level Comment Vs. Line-level Comment and Revision Review File Content

Our goal was to extract a similar number of concerns from each project to represent them equally. Thus, from each project, first, a discussion was randomly selected and concerns were extracted by manually reading through all revision review files

Project	Description	Language	Population				Sample			
			Time span	Number of discussions	Number of revisions	Number of developers	Time span	Number of discussions	Number of revisions	Number of developers
1	Couchbase's driver implementation in Java	Java	01-12 to 11-17	917	2638	35	02-12 to 10-17	40	124	14
2	Low-level API mostly used by java-client	Java	04-14 to 11-17	841	2301	18	04-14 to 11-17	45	156	8
3	Building blocks for user interfaces in Eclipse	Java	02-13 to 11-17	4756	14118	290	03-13 to 11-17	44	147	31
4	Integration of jgit into the Eclipse IDE	Java	10-09 to 11-17	5337	13231	184	10-09 to 08-17	34	72	22
5	Java implementation of Git	Java	09-09 to 11-17	5384	14043	251	04-10 to 11-17	32	54	21
6	C/C++ IDE for linux developers	Java	06-12 to 11-17	5105	15337	82	08-12 to 03-17	33	82	16
7	Implementation of a memory caching system	Java	05-10 to 07-17	519	1383	36	05-10 to 10-11	42	88	11

Table 1: Population and sample summaries

under the selected discussion. Next, another discussion was randomly selected and the same extraction process was executed on its revision review files. This was repeated until at least 55 concerns (target sample size 385 / Number of projects 7) were extracted from each project. Since we extracted all concerns from each randomly selected discussion, each project had different numbers of extracted concerns within the range of 56 and 71.

We chose to extract all concerns from a single discussion, rather than randomly sampling concerns across each project, to ensure that for each discussion, we captured each type of concern that was identified by the reviewers i.e. we extracted both file-level and line-level concerns in each discussion. Fig.4 shows the structure of a revision review file and the difference between line and file-level comments.

When a single comment was referring to more than one concern, the comment was split in a way that each part of the comment was referring to only one concern. Then each part of the comment was labeled independently. There were 22 comments that we had to split into 2 or more concerns. Similarly, if there were more than one comment referring to the same concern, they were grouped as a single comment. There were 9 such concerns.

3.4. Labeling Concerns and Grouping Concern Types

To answer RQ1, we built a classification of *concern types* by applying thematic analysis [29] to the code review data described above. The concerns described in code review comments and the associated code were analyzed to identify the type of concern and each concern was labeled with a suitable *concern type* label.

Concern types were then grouped based on common themes that emerged among them. For example, "why dropping public on these? we don't seem to do that elsewhere" was labeled "Missing access modifier", "Fields that sub-classes might want to update should be either protected or have protected setters (protected methods), for example fStatisticsData." was labeled "Unexpected access modifier", and both these concerns were grouped under "Modifiers" because they were both concerns related to modifiers used in code.

Some concerns were lacking information on their root cause. In such cases, we assigned them labels with higher-levels of abstraction. For example, "This will not generate a pom file with the correct dependencies." refers to a functional issue in the code, but does not reveal what causes this issue. We labeled this concern as "Unexpected logic and functionality". These labels together with other root cause-level labels constitute the lowest layer of our classification.

We used consistent wording across the labels we created. "unexpected" was used to mean that the reviewer observed something different compared to what they were expecting in the code but it may or may not be incorrect (e.g., unexpected functionality and logic). "Better <ITEM> exists" was used to mean that the reviewer felt that the current subject of interest can be further improved (e.g., better design exists). "Unnecessary" was used to mean that something not required is present in the code (Eg: Unnecessary modifier). "missing" was used to mean that a required something is not present in the code (Eg: Missing comment). "unconventional" was used to mean that something in the code is against the project-conventions (Eg: Unconventional identifier name, Unconventional license header pattern).

During the labeling process, the first author was mainly responsible of labeling the concerns. After 100 concerns were labeled and grouped, to ensure the reliability of the classification, the other two authors categorized 10 concerns each in 5 rounds (total 50 comments), into the existing labels created by the first author or new labels as required. Each round was followed by a discussion among the three authors where the labels were changed, updated, and moved into different groups appropriately.

The concern extraction resulted in 397 concerns. Once this was completed, additional 20 concerns (equivalent to 5% of the original sample size) were extracted by the first author from the data set and categorized to check for theoretical saturation. All of these concerns could be categorized into the existing concern types, suggesting that saturation may have reached. These concerns were also included in the classification bringing the total number of concerns to 417.

Project	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Extracted number of concerns	60	56	58	61	71	67	56
Unclear concerns	4	2	4	2	1	4	0
Discussions including concerns	3	9	12	8	2	5	10
Total evaluated discussions	40	45	44	34	32	33	42

1 = couchbase-java-client, 2 = couchbase-jvm-core, 3 = eclipse.platform.ui, 4 = egit, 5 = jgit, 6 = org.eclipse.linuxtools, 7 = spymemcached

Table 2: Concern Extraction Summary

Table 3.4 lists the number of concerns extracted from each project, the number of concerns that were difficult to understand i.e. unclear concerns, the number of discussions that included concerns, and the number of total discussions that were evaluated in each project. A concern was marked as unclear if it was concluded to be incomprehensible among the authors. "Document in exec and connect that exec is called, then connect. It doesn't make sense that this is the behavior, but it is." is an example of an unclear concern.

As a final reliability check, each author other than the first author categorized 15 randomly selected concerns from the 417 categorized concerns, followed by a discussion. At this stage, there were no conflicts among the authors regarding the labeling and grouping. This gave us the confidence that the classification was reliable.

3.5. Detection Method Classification

To answer RQ2, we classified each concern based on whether it can be automatically detected accurately or it must be manually detected by applying the criteria given in table 3.4. The code containing each concern was analyzed using five popular static analysis tools: SonarQube, Checkstyle, FindBugs^{TM}, PMD, and IntelliJ IDEA Code Inspection, and Grammarly to detect grammar-related issues in API documentation. The tools were used to check whether the concern could be detected without producing any false positives or false negatives. These tools were selected because they have been used commonly and therefore improved throughout the past decade based on user feedback.

If these tools could not detect the concern, the concern was discussed among the authors to decide if accurate automation was currently possible. For example, "Do you think we should be addressing some of this in the class documentation rather than repeating it next to each method? It feels a bit repetitive." referring to identical text blocks in API documentation comments can be detected by performing a simple text search in the code for identical Javadocs. Thus, it should be categorized as an automatically detected concern even though the standard rules of the studied tools could not detect it. The complete list of concerns with their "detection method" classification, evidence on the automatability of concern detection, and the versions of tools studied are available in our supplementary material [69].

This classification of automatically and manually detected concerns was done in parallel with the con-

Category	**Definition of Category**
Automatically detected	These concerns can be accurately detected automatically with existing tools or new heuristics could be easily defined to enable their detection. "Accurately" implies that false positives and false negatives will never be produced during the detection of the concern. (Eg: Missing braces, Code lines longer than 80 characters)
Manually detected	The detection of these concerns cannot be automated accurately. (Eg: Poor design, Low class cohesion)

Table 3: Automatically Vs. Manually Detected Concerns

cern type labelling described above to ensure that the level of detail of the concern type labels was suitable to differentiate between automatically and manually detected concerns. Our aim was that for each concern type label, all concerns with that label should also share the same automatically or manually detected label. In nearly all cases, by creating low-level labels, this was possible. There were nine of the 116 concern types where it was not desirable to differentiate between the concerns that could be automatically detected and those that needed to be manually detected. For example, "The ID *correspond* to the package in which this class is embeded." labeled "Documentation Grammar" could be automatically detected by Grammarly while "The list of *the* traces to add in the tree", also labeled "Documentation Grammar" could not be automatically detected using Grammarly [70]. Splitting such concern types further would have increased the number of concern types undesirably, and have made the classification unnecessarily complex.

4. Results

4.1. Concern Types Classification

We identified 116 concern types, which are listed in Fig.5. The concern type that had the largest distribution was **Trailing whitespace** (22/417, 5%). Trailing whitespace is an unnecessary single space introduced at the end of the code statements during code changes. This is different from Unnecessary whitespace which refers to a whitespace that is not required, but present within a code statement and not at the end of the statement.

The next most common concern type was **Missing Documentation Info** (16/417, 4%). Many of the concern types (43% or 50/116) occurred only once in the sample. The complete descriptions of each concern type and how each concern was classified into these concern types is available in our online supplementary material [69].

These 116 concern types were grouped into 15 concern groups as shown in Fig.5. Table 4.1 lists the distribution of concern types and concerns within each concern group. The column, "Number of Concern Types" provides the count and percentage of concern types under each group. "Number of Concerns" column provides the count and percentage of individual concerns under each group. Concern groups are discussed below in the order of most number of concerns to least number of concerns in the sample.

concern group	Number of Concern Types (Total 116)		Number of Concerns (Total 417)	
	Count	%	Count	%
API Documentation	12	10.4	68	16.3
Implementation	19	16.4	62	14.9
Appearance	11	9.5	61	14.6
Logic and functionality	15	12.9	59	14.1
Design	16	13.8	31	7.4
Modifiers	6	5.2	25	6.0
Comments	8	6.9	20	4.8
Identifier naming	4	3.4	19	4.6
Errors, warnings, and logging	7	6	14	3.4
Performance	1	0.9	14	3.4
Header comments	7	6	14	3.4
Annotations	4	3.4	11	2.6
Test cases	3	2.6	10	2.4
Literals	2	1.7	8	1.9
Threads	1	0.9	1	0.2

Table 4: Distribution of Concerns and Concern Types

API Documentation represents concerns related to a special group of comments, API documentation comments and contains the most concerns in the sample (68/417 concerns , 16.3%). "Here the timeout is in milliseconds, but the API Javadoc suggested seconds. That's why I wanted to clarify it." labeled "Unexpected documentation info" is an example of a comment that belongs to this group.

Implementation (62/417 concerns, 14.9%) reports shortcomings of the current way of implementation. "My preference is to use org.eclipse.core.runtime.Platform.getWS() and getOS()" labeled "Better library use exists" is an example which suggests that a better library method call can be used instead of what is currently used in the implementation to achieve the same outcome.

Appearance (61/417 concerns, 14.6%) includes concerns related to the way code looks, such as unnecessary braces, trailing whitespaces, and incorrect indentation. "Style-nit: In JGit we don't put curly braces around single line statements." labeled "Unnecessary braces" is an example.

Logic and functionality groups concerns related to logic that cause the functionality to be different from the expected behaviour (59/417 concerns, 14.1%). "This will not generate a pom file with the correct dependencies." labeled "Unexpected logic and functionality" is an example of such a concern.

Design (31/417 concerns, 7.4%) includes concerns related to the architecture of the software under review. An examples is "I'm pretty sure this

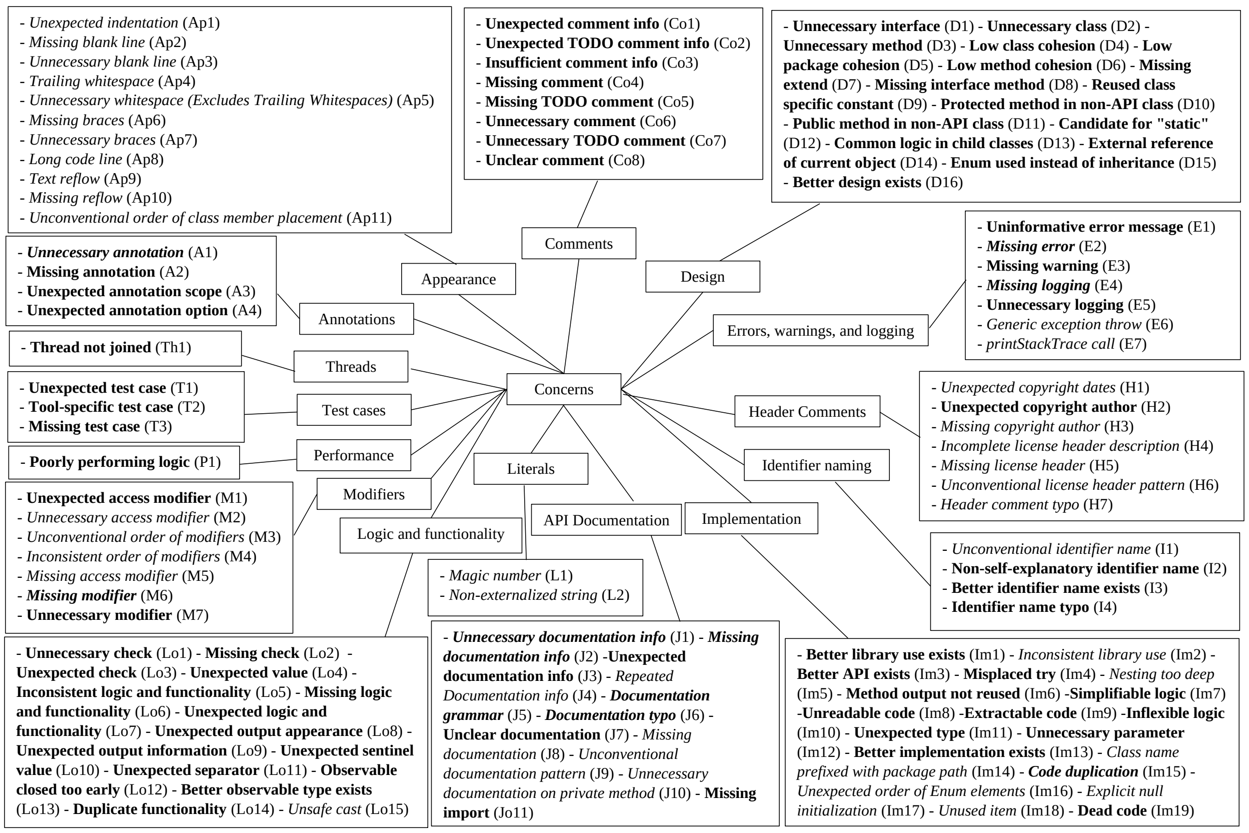

- Unexpected indentation (Ap1)
- Missing blank line (Ap2)
- Unnecessary blank line (Ap3)
- Trailing whitespace (Ap4)
- Unnecessary whitespace (Excludes Trailing Whitespaces) (Ap5)
- Missing braces (Ap6)
- Unnecessary braces (Ap7)
- Long code line (Ap8)
- Text reflow (Ap9)
- Missing reflow (Ap10)
- Unconventional order of class member placement (Ap11)
- Unexpected comment info (Co1)
- Unexpected TODO comment info (Co2)
- Insufficient comment info (Co3)
- Missing comment (Co4)
- Missing TODO comment (Co5)
- Unnecessary comment (Co6)
- Unnecessary TODO comment (Co7)
- Unclear comment (Co8)
- Unnecessary interface (D1) - Unnecessary class (D2) - Unnecessary method (D3) - Low class cohesion (D4) - Low package cohesion (D5) - Low method cohesion (D6) - Missing extend (D7) - Missing interface method (D8) - Reused class specific constant (D9) - Protected method in non-API class (D10) - Public method in non-API class (D11) - Candidate for "static" (D12) - Common logic in child classes (D13) - External reference of current object (D14) - Enum used instead of inheritance (D15) - Better design exists (D16)
- Unnecessary annotation (A1)
- Missing annotation (A2)
- Unexpected annotation scope (A3)
- Unexpected annotation option (A4)
Comments
Appearance
Design
Annotations
Errors, warnings, and logging
- Uninformative error message (E1)
- Missing error (E2)
- Missing warning (E3)
- Missing logging (E4)
- Unnecessary logging (E5)
- Generic exception throw (E6)
- printStackTrace call (E7)
- Thread not joined (Th1)
Threads
Concerns
- Unexpected test case (T1)
- Tool-specific test case (T2)
- Missing test case (T3)
Test cases
Header Comments
- Unexpected copyright dates (H1)
- Unexpected copyright author (H2)
- Missing copyright author (H3)
- Incomplete license header description (H4)
- Missing license header (H5)
- Unconventional license header pattern (H6)
- Header comment typo (H7)
- Poorly performing logic (P1)
Performance
Literals
Identifier naming
Modifiers
API Documentation
Implementation
Logic and functionality
- Unexpected access modifier (M1)
- Unnecessary access modifier (M2)
- Unconventional order of modifiers (M3)
- Inconsistent order of modifiers (M4)
- Missing access modifier (M5)
- Missing modifier (M6)
- Unnecessary modifier (M7)
- Magic number (L1)
- Non-externalized string (L2)
- Unconventional identifier name (I1)
- Non-self-explanatory identifier name (I2)
- Better identifier name exists (I3)
- Identifier name typo (I4)
- Unnecessary check (Lo1) - Missing check (Lo2) - Unexpected check (Lo3) - Unexpected value (Lo4) - Inconsistent logic and functionality (Lo5) - Missing logic and functionality (Lo6) - Unexpected logic and functionality (Lo7) - Unexpected output appearance (Lo8) - Unexpected output information (Lo9) - Unexpected sentinel value (Lo10) - Unexpected separator (Lo11) - Observable closed too early (Lo12) - Better observable type exists (Lo13) - Duplicate functionality (Lo14) - Unsafe cast (Lo15)
- Unnecessary documentation info (J1) - Missing documentation info (J2) -Unexpected documentation info (J3) - Repeated Documentation info (J4) - Documentation grammar (J5) - Documentation typo (J6) - Unclear documentation (J7) - Missing documentation (J8) - Unconventional documentation pattern (J9) - Unnecessary documentation on private method (J10) - Missing import (Jo11)
- Better library use exists (Im1) - Inconsistent library use (Im2) - Better API exists (Im3) - Misplaced try (Im4) - Nesting too deep (Im5) - Method output not reused (Im6) -Simplifiable logic (Im7) -Unreadable code (Im8) -Extractable code (Im9) -Inflexible logic (Im10) - Unexpected type (Im11) - Unnecessary parameter (Im12) - Better implementation exists (Im13) - Class name prefixed with package path (Im14) - Code duplication (Im15) - Unexpected order of Enum elements (Im16) - Explicit null initialization (Im17) - Unused item (Im18) - Dead code (Im19)

Figure 5: Code Review Concerns Classification

class doesn't belong here at all. It looks like it's for testing??" labeled "Low package cohesion".

Implementation, Logic and functionality, and Design-related concerns included both statement-level concerns (Lo1, D12, Im11) and code block-level concerns (Lo6, D1, Im13). The block-level concerns were the complex and larger concerns that were labeled with a higher-level of abstraction. It is up to the author to investigate the root causes of such concerns and correct them.

Modifiers (25/417 concerns, 6.0%) groups concerns related to modifiers used in Java. "Should this be public? I'm not quite sure what I'd do with this." labeled "Unexpected access modifier" is an example. Modifier-related concerns were either access modifier-related (M1, M2), non-access modifier-related (M6, M7,) or related to both (M3,M4). "final" and "static" in Java were examples of non-access modifiers. The concerns related to both modifier types were regarding the order in which the modifiers were placed with respect to each other.

Comments (20/417 concerns, 4.8%) group contains concerns related to regular inline comments that describe the code. "I find this comment pretty confusing. What is 'the changes feed'. I know this isn't in this commit, but needs to be somewhere." refers to an Unclear comment which falls under this category. Comment-related concerns were either related to TODO comments (Co2, Co5, Co7) or normal code comments (Co1, Co3). TODO comments were in fact code comments that documented parts of implementation that was yet to be implemented. Concerns related to TODO comments were much less (3 concerns) compared to the normal code comments (17 concerns).

Identifier naming (19/417 concerns, 4.6%) includes concerns related to identifier names such as "This is a meaningless variable name. I can't tell what it is without digging further." that was labeled "Non-self-explanatory identifier name".

Errors, warnings, and logging (14/417 concerns, 3.4%) includes concerns related to either errors, warnings or logging i.e. concerns related to the error communication of a program. "I realize this isn't a new change, but we shouldn't be calling e.printStackTrace anywhere." labeled "printStack-Trace call" is an example of a concern that belongs to this group.

Performance (14/417, 3.4%) represents performance-related concerns of program code. "This one is likely to be a lot more expensive." is an example. Additionally, this is one of the groups that contain the least variety of concern types: just 1 concern type labeled "Poorly-performing logic".

Header comments (14/417, 3.4%) was another group that demanded a separate concern group due to the variety of header comment-related concern types we found in the sample. "Comma here too ;)" referring to a header comment typo is an example. Observing several random files of the data set showed that header comments are written using either Javadoc notation or a regular comment notation in practice.

Annotations (11/417 concerns, 2.6%) represents annotation-related concerns. "We should add @noimplement" labeled "Missing annotation" is an example of a concern that is annotation-related. This concern group is language specific and may not be applicable to a language other than Java.

Test cases (10/417 concerns, 2.4%) contain concerns related to test cases used to test the software. "also assertFalse(second.hasSubscribers())" referring to a Missing test case is an example of a test case-related concern.

Literals (8/417 concerns, 1.9%) contain concerns related to literals used in code. An example of Literal-related concern is "maybe extract as a constant?" referring to a Magic number.

Threads (1/417 concerns, 0.2%) included only 1 concern, thus only 1 concern type: Thread not joined ("Looks good. But there is another issue in GitRepositoriesViewRemoteHandlingTest. Checkout job needs to be joined.").

4.2. Manually Vs. Automatically Detected Concerns

We defined "automatically detected concerns" as concerns of which the detection can be automated accurately. The remaining concerns were categorized as "manually detected concerns". In Fig.5 automatically detected concerns are marked in *italics* while manually detected concerns are marked in **bold**.

From the 417 concerns, the detection of 22% concerns were supported by the tools we studied and another 16% (36/417 concerns) were not supported by the tools we studied but could be detected automatically and accurately by defining new heuristics easily. Thus, a total of 38% (157/417) concerns were categorized under "automatically detected concerns" and the remaining 62% (260/417) of concerns were categorized under "manually detected concerns". Out of the 116 concern types, 32% (37/116) of the concern types were categorized under automatically detected concerns only, 60% (70/116) of the types under manually detected concerns only, and 8% (9/116) contained both types of concerns (fig.5). The concern types that included both automatically and manually detected concerns were Unnecessary annotation, Missing error, Missing logging, Code duplication, Documentation grammar, Documentation typo, Missing Documentation info, Unnecessary Documentation info, and Missing modifier.

An example of an **automatically detected** concern is "missing braces" of a control structure which can occur in Java when it is project's convention to use mandatory braces marking control structure bodies, but the developer decided not to use braces because the control structure body was only a single code line (Java allows this). Many tools provide checks to enforce control structure braces (Eg: SonarQube - Control structures should use curly braces [30], Checkstyle - NeedBraces [71], PMD - ControlStatementBraces [72], IntelliJ - Control flow statement without braces [73]) and can automatically detect such cases without producing any false positives or negatives.

Fig.6.a illustrates the distribution of automatically detected concerns among the concern groups. From the 157 automatically detected concerns, the majority of concerns belonged to Appearance (61/157 concerns, 39%). "Trailing whitespace" was the most common (22/157 concerns, 14%) concern type among automatically detected concerns. The lowest distribution of automatically detected concerns was under Annotations group (1/157, 0.6%). The only Annotations-related, automatically detected concern was "This is a test plugin, NLS warnings are not enabled, so these annotations are not needed." labeled "Unnecessary annotation".

An example of a **manually detected** concern is "I think we should find a different name for this, because an unsubscribe could be because of a timeout but may also be for different reasons. What about something like isActive()? if unsubscribed then its just not active and can be dropped

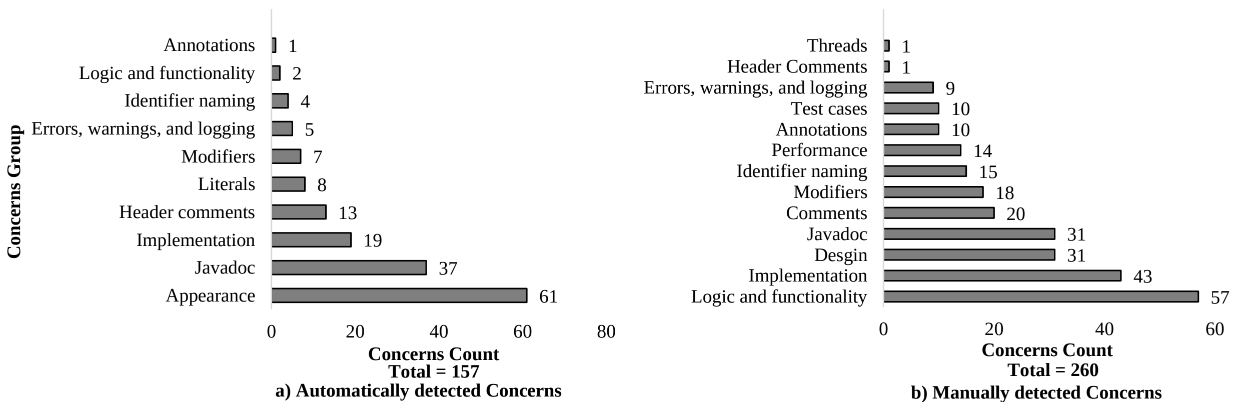


Figure 6: High-level categorization of concerns - Detection Method

for various reasons." labeled "Non-self-explanatory identifier name". Even though existing technology can be used to identify potentially bad identifier names [39], whether the identifier name is bad enough to raise the concern depends on the reviewer.

Fig.6.b illustrates the distribution of manually detected concerns among concern groups. From the 260 manually detected concerns, the highest distributed was related to Logic and functionality (57/260, 22%), and the lowest was Threads and Header comments-related concerns (1/260, 0.4%). The only manually detected, Header comment-related concern was "We generally assign copyright to either the author or employer, depending on what is most appropriate. don't attribute copyright to Eclipse Egit Team, instead use your name and email or your company" labeled "Unexpected copyright author".

5. Discussion

5.1. RQ1: Concerns Identified in Real-world Code Reviews

The study revealed 116 concern types identified in real-world code reviews and 15 concern groups that they could be grouped into. Fig.5 is an overview of the concern types that were present in the sample. The API Documentation group contained the most number of concerns (68/417 concerns, 16.3%). The least concerns were observed in the Threads group (1/417, 0.2%). Implementation group contained the highest number of concern types (19/116 concern types, 16.4%). The least variety (1/116, 0.9%) of concern types was in Performance ("Poorly performing logic") and Threads ("Thread not joined"). The most common concern type in the sample was "Trailing whitespace", present 22 times (22/417, 5%). 43% (50/116 concern types) of the concern types were present only once in the sample.

We did not find any security-related concerns in the sample. Several possible reasons behind this are explained in the literature. The diversity of the security issues is overwhelming for general developers and the lack of effectiveness of the security assurance tools is a challenge and thus it usually leads to security review avoidance or bringing in expensive external resources to the organization. Also, the lack of expertise and security being a non-functional requirement adds to its invisibility during code reviews. [74, 75, 76]. There are many existing static analysis tools that are designed to support security defect detection [77, 78, 79, 80, 3]. However, due to barriers like usability problems, these tools are not well-adopted [81]. The lack of usable and useful security tools was reflected in a survey where most programmers ideally wanted to see security warnings on static analysis tools [82].

We found only one concern related to multi-threaded programming: Thread not joined. The presence of this one concern shows that at least one of the projects is making use of multi-threaded programming. The rarity of concurrency concerns in the sample could be due to the nature of the projects, the nature of the considered sample, or the complexity of concurrency programming compared to sequential programming. A study at Microsoft has shown that Concurrency bugs take on average several days to detect, reproduce, debug and fix [83]. There are many forms of support to help concurrency bugs detection in code [84, 85, 86, 87, 88]. However, their spurious results and effectiveness still need to be improved further [89].

Some of the concern types discovered made it apparent that coding conventions can be different from project to project. For example, the two inverse conventions "unnecessary braces" (in project "Jgit") stating that braces should not be used in control structures with single statement bodies and "missing braces" (in project "Egit") that mandates to always use braces in control structures were discovered in the sample implying differences of project-level conventions. "unnecessary braces" also shows that sometimes project conventions can be contrasting to the generally accepted standards (Oracle Java Coding Conventions [90]). Header comments is another element that we observed differences of. While some programmers used Javadoc notation for header comments, some used regular comments. There are numerous language-specific [90] as well as general [91] coding conventions created by different authors, organizations, and projects [92]. Due to their objective nature, most coding conventions are well-supported by existing tools, compared to security and concurrency bugs [93, 71, 94].

The experience of producing this classification disclosed that diff tools (code review tools) partially automate the detection of two of the simplest concern types, unnecessary trailing white spaces and blank lines, as a side-effect. The introduction of the two concern types could be due to the specific development tools or their configurations used in the considered projects. Since the diff view of code review tools highlights newly added blank spaces in red, it is convenient for the reviewers to spot them. Otherwise, the usual goals of code review tools are to support review information management and collaboration [16, 13].

Additionally, we discovered two forms of subjectivity related to code review decisions. Concern types such as "Nesting too deep" and "Long code line" contain a subjective component that depends on project conventions i.e. *project subjectivity.* For example, the maximum depth of nesting allowed would be based on the project conventions and may differ from project to project. The other form is the *reviewer subjectivity* that has an effect on the identification of concern types such as "Non-self-explanatory identifier name". The meaningfulness and suitability of an identifier name and the requirement to improve an identifier name quality depends on the reviewer's perspective and may differ from reviewer to reviewer. The presence of these two forms of subjectivity is a major barrier for effectively reducing code review cost using static analysis tools.

5.2. RQ2: Code Review Automation

The results showed that MCR identifies more than concerns that are inherently manual to assess. A substantial number of concerns (157/417 concerns, 38%) and concern types (46/116 concern types, 40%) could be detected automatically without producing any false positives or false negatives. Attempting to capture these concerns by consistently using defect detection tools prior to code review would reduce the code review cost considerably.

Of the 38% of concerns that can be automatically detected, 22% are supported by the popular tools we studied, and 16% concerns were not supported by those tools but could be accurately detected automatically by defining new rules. The latter type of concerns are often concerns that are project-specific as general rules of the tools cannot detect them. "We need to clean up the logging. Some is JDK logging, some is spy logging." is an example.

Many existing tools allow teams to create quality profiles, which could be used to handle project-specific rules [95]. Software teams could create project-specific custom rules and add them to a profile at the tool configuration stage. Then, the quality profile can be assigned to a project, so that those rules will run only against that particular project. Software teams can create such a profile for each project they work on. We believe that having custom rules can standardize the project conventions further and ensure consistency throughout the project. Quality profiles may help to manage the resulting large number of custom rules. Future work can investigate this further.

The automatically detected concerns in our classification do not include concerns that can be partially detected using existing tools because such concerns cause false positives and false negatives in static analysis tool results and therefore result in low adoption of static analysis tools [96, 18, 97]. The low adoption of static analysis tools lead to manual defect detection which is a barrier to reducing the code review cost. By improving existing static analysis tools further (to minimize false positives and other barriers), we believe that the cost of code review can be further reduced. However, for now, we recommend using tools only for those concerns that can be identified *accurately*. Partial automation and manual code review should be compared in future to determine which is more cost effective.

To provide tool recommendations, we looked at the number of concern types that each tool could support from the concern types we identified. IntelliJ IDEA Code Inspection detected the highest number of concern types (16) accurately. Grammarly detected the 2 concerns that it is designed to detect: API Documentation Typo and API Documentation Grammar, when the Javadocs were extracted and tested on it. SonarQube, Checkstyle, PMD, and FindBugs^{TM} supported 15, 14, 10, and one concern types respectively. However, this does not mean that one tool has more features than the other.

For example, we observed that FindBugs^{TM} supported many complex design-related checks but false positives and false negatives were inevitable in such cases. Thus, they were not counted as automatically detected concerns. An example is "Return value of method without side effect is ignored" rule supported by FindBugs^{TM} [98]. Such checks can be called "partial automation" because human intervention is still required to make the final decision on whether a concern is present or not.

5.3. Opportunities for Reducing Code Review Cost

Conducting this study helped us identify several opportunities for reducing the code review cost.

5.3.1. Automatically detected concerns

Automatically detected concerns evidently still reach code review sessions. Consistent and thorough use of existing defect detection tools prior to code reviews can help prevent this, reducing the code review cost by a substantial amount (22% concerns of the considered sample). Another 16% concerns were not supported by the popular tools we studied, but their detection could be automated accurately. Using new custom rules together with the standard rules of existing tools, and using these tools consistently and thoroughly, code review cost can be reduced. We have identified the concerns that can be automatically detected i.e. the opportunities where human effort and resultant cost can be prevented (Concern types in italics in fig.5).

5.3.2. Partially Automated Concerns

There are still many concerns identified in code reviews that can be partially detected by existing defect detection tools. These are currently categorized under "manually-detected concerns". We are expecting to differentiate these from fully manually detected concerns in our future studies so that human effort required in such cases also can be minimized. Additionally, investigating the effect of partial automation on code review cost is an interesting future research avenue.

5.3.3. Functional Concerns

From the sample, 26% concerns were functional concerns. Some of them may have been detected comparatively inexpensively by having a high-quality test suite instead of putting code review effort into the task. An example of such a concern is "what if the ICommitMessageProvider returns null? ..." that could be detected by having a test case that invokes the related code with ICommitMessageProvider being null.

5.3.4. Code Review and Expertise

Category	**Definition of Category**
No programming expertise	These concerns can be detected with no programming expertise or training of any kind. The review instructions and the source code representation may have to be modified to support non-programmers. (Eg: "Detect grammatical mistakes in the following documentation texts extracted from a software system.")
No programming expertise with training	These concerns can be detected with no programming expertise, but requires training on other concepts related to the concern. The review instructions and the source code representation may have to be modified to support non-programmers (Eg: "Detect low-quality error messages from the following list of error messages extracted from a software system" will instruct the reviewer to detect low-quality error messages such as "Something went wrong!". The reviewer needs training on what a low-quality error message is.)
Programming expertise	These concerns require programming expertise and the original source code to be detected. (Eg: Logical and functional concerns, Design-related concerns)

Table 5: Detection Expertise Categories

Code review cost (time and effort) has been a persistent problem in code review [99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 10, 110, 111, 2, 112]. Many researchers have attempted to solve this problem by suggesting different guidelines [113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 97, 124]. Several common measures are suggested in these guidelines: determining reviewers based on the available human resources rather than fixing the number of reviewers by process, promoting tool usage, and not having review meetings or having them only when required. All these measures achieve cost reductions by ultimately reducing the number of programmers that have to be involved in the review process. On the contrary, we suggest, reducing the amount of money spent on human resources or in other words, recruiting less expensive human resources (i.e., non-programmers) may help to reduce the code review cost.

The domain-specific expertise and technological-expertise are two major factors considered in modern reviewer recommendation systems because of their effect on the code review quality [125, 126, 127, 128, 129, 130, 122]. Thus, it is common for domain and technological experts to review code in practice. However, sometimes, the code review decision is as simple as spotting a meaningless variable name such as "objRef" and thus might be able to be identified by a non-programmer as well. By delegating such simple code review tasks that are usually considered as nitpicking by developers to non-programmers who are less expensive, the code review cost can be further reduced. In fact, at Microsoft, a preliminary "code improvement" review round is conducted prior to the actual code review session to identify maintainability issues [8]. This can be an excellent point at which non-programmers can contribute to code review by identifying nitpicks that they can so that programming experts can focus on more complex issues in code. Organizations with cross-functional teams can utilize their existing human resources more effectively in the code review process and improve the programming productivity of expert programmers by reducing the workload imposed on them by code review. Another possibility is to crowd-source these tasks as micro-jobs.

As the first step to study the feasibility of having non-programmers perform some elements of code review, we created a non-programmer-centric concerns classification, "Detection Expertise". The categorization was done according to the criteria defined in table 5.3.4. In creating the categories for this classification we considered 1) whether a non-programmer can detect the concern or programming expertise (i.e. domain and technological expertise acquired by a programmer over time) is required, and 2) whether providing the non-programmer instructions on what to detect is sufficient or training on concepts related to the concern is required. Additionally, while verifying this classification in the future studies, we will also have to consider whether the original source code and review instructions can be used for review or modified representations of the source code and modified instructions understandable by non-programmers are more appropriate to get the task done effectively by the non-programmers.

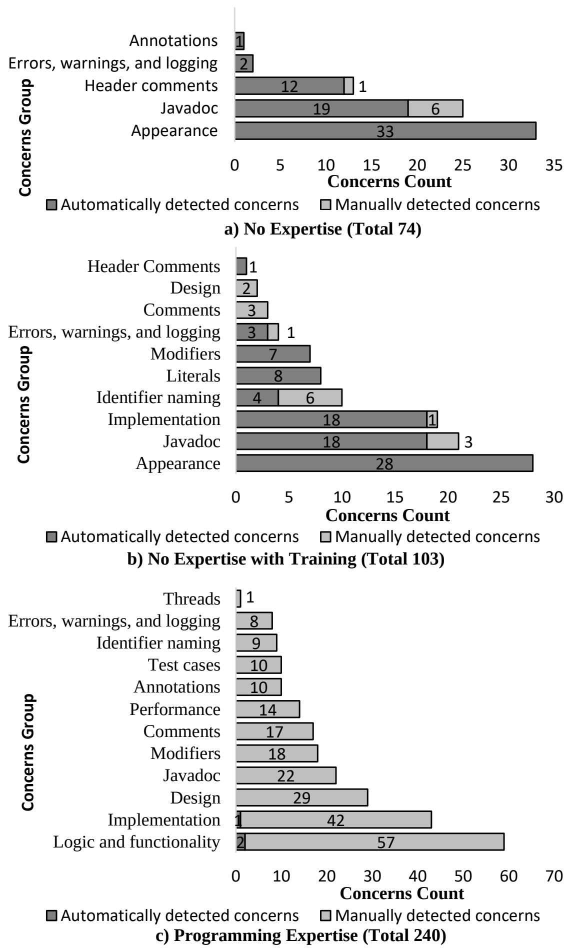


Figure 7: High-level categorization of concerns - Detection Expertise

To classify, each code review comment and the associated code were studied by the first author to understand the nature of the concern: whether programming expertise is a prerequisite and whether concept-training is required to detect the presence of the concern. Once the first author had completed this classification, the other two authors categorized 15 randomly selected comments from the sample. A Fleiss Kappa value of 0.624 implying a substantial agreement was obtained among the 3 authors. Obtaining a perfect agreement is difficult for this kind of categorization where the authors had to depend on their subjective understanding about the concerns reported in the classified comments. However, following this categorization, the authors had a discussion where they reached a consensus. Based on this, the first author re-categorized all comments one last time. The examples given below explain the 3 categories in this classification further.

The detection of concerns related to the logic of a program requires the ability to read and understand the code, thus, should be categorized under "Programming Expertise".

A missing Javadoc (documentation comment) on a method can be detected by a person who can identify method blocks. This does not require the code to be understood, rather training on identifying method blocks based on the indentation and the placement of curly brackets. Thus, the concern "missing documentation" should be categorized under the "No Expertise with Training" category.

To detect the code lines longer than 80 characters, the reviewer does not need to read and understand the code i.e. requires no programming expertise and does not have any complex concepts to learn i.e. requires no training. The reviewer simply has to identify the text lines longer than 80 characters in the code file. Thus, "long code line" should be categorized under the "No Expertise" category.

Fig.7 depicts the distribution of concerns in each expertise level. The complete classification is available online [69].

Programming expertise category with the most number of concerns in the sample (240/417 concerns, 57%) contained 12 concern groups (Fig.7.c). Logic and Functionality (59/240, 25%) included the most number of concerns. This category is the only instance that Logic and functionality, Performance, and Test cases groups can be observed. This is because they are the most complex aspects of programming and thus they need programming expertise to detect. Naturally, the majority of concerns that require expertise to identify could not be detected automatically either, due to their "inherently manual to assess" nature.

No programming expertise with training category (103/417 concerns, 25%) (Fig.7.b) contained 10 concern groups of which the majority was Appearance-related (28/103, 27%). The manually detected concern types that belong to this category are 3.8% of the sample (16/417 concerns) which is another promising cost reduction opportunity because they can be detected with no programming expertise and with non-extensive concept training.

No programming expertise category (74/417 concerns, 18%) contained 5 concern groups. The majority of concerns were Appearance-related (33/74, 45%)(Fig.7.a). The most effective code review cost reduction opportunity here is in the manually detected concerns because their detection cannot be automated accurately and therefore human involvement is required. 1.7% (7/417 concerns) of the concerns in the sample are manually detected and require no programming expertise to detect. If a person with no expertise (i.e a less expensive human resource) can detect those concerns, the overall code review cost can be effectively reduced.

There is also another 37% (154/417 concerns) concerns in the sample that belong to the last two categories explained above and are also *automatically detected* concerns. For organizations that prefer manual code review, this is a significant opportunity to reduce the cost of their code review process by utilizing less expensive human resources.

Thus, we estimate that in total 42.5% of the concerns in our sample could be identified by someone without programming expertise, which could enable another avenue for cost savings in software code review. Programmers will agree that the concerns in this group are mostly "nitpicks". A study conducted at Microsoft implies that reviewers tend to miss more complex issues in code due to these nitpicks [8]. Another study has shown that fixing soft maintenance issues lowers the cost of future changes [23]. Therefore, nitpicks are important to be discovered. Also, they do carry a certain identification cost. However, this cost is unknown. Future studies should examine the cost of identifying such concerns to see whether they are worth being delegated to non-programmers. Future studies also should validate this classification.

5.4. A Comparison of Classifications

Code review defect classifications existing today are the products of an evolutionary process of researchers attempting to make classifications more informative and more inclusive of the many possible code review defect types. Most of these classifications either adapt a previous classification or integrate a previous classification. For example, the Panichella and Zaugg classification [27] published recently integrates the Beller classification [42] which in turn adapts the Mäntylä and Lassenius classification [24]. We compared our classification to the most recent and most detailed existing classification, Panichella and Zaugg classification, also called CRAM [27].

Since our lowest level categories were more fine-grained than CRAM, we grouped our concern types under the lowest-level categories of the Panichella and Zaugg classification (the details of this grouping is available in our supplementary material [69]). Based on the definitions of the low-level categories of CRAM, we found it difficult to fit 10.34% of our concern types into their classification categories. From the 15 concern groups in our classification 14 overlapped with the categories of CRAM. The remaining group, "Annotations" could not be placed in CRAM because CRAM data set did not contain any code review comments on code-related annotations. However, according to Mäntylä classification that CRAM is indirectly based on, "Annotations" belongs in the "Language supported documentation" high-level category. The other concern types for which we could not find matching CRAM low-level categories are better implementation exists, inflexible logic, better observable type exists, observable closed too early, unexpected logic and functionality, unexpected sentinel value, and unexpected separator.

Conversely, CRAM contained some low-level categories that we did not include. We did not differentiate, for example, "semantic duplication" and "duplicate code" whereas in CRAM they were differentiated. "Semantic dead code" and "Dead code" were another example. CRAM did not differentiate API Documentation-related changes and Comment-related changes whereas we did. Also, CRAM does not separate automatically detected concerns from manually detected concerns.

Our classification was an attempt to cleanly separate automatically detected concerns and manually detected concerns. During the process we learnt that this was not entirely possible. However, we were able to minimize the number of concern types that contained both automatically and manually detected concerns to just 9 concern types out of 116 (8%). When concerns in CRAM are considered, 15 "detailed changes" out of 45 (33%) contain both automatically and manually detected concerns. Thus, we have been able to improve the separation of automatically and manually detected concerns further.

The existing classification studies report a striking 75:25 ratio of evolvability to functional defects or changes in industrial and OSS projects [24, 42]. Here, *evolvability defects* are the defects that affect future development efforts instead of runtime behavior. *Functional defects* are the defects that affect runtime behavior. We also classified our concern types as evolvability concerns and functional concerns [69] and found a similar ratio of 74:26. Thus, our study also supports their implication that code review is superior to software testing as it not only finds the same amount of functional defects as testing but also identifies a large number of evolvability (non-functional) defects. However, it should be noted that in contrast to these other studies, we did not categorize only true positives. Rather, we considered all reported concerns in the sample that represent the entire code review effort involved.

5.5. Threats to Validity

5.5.1. Construct Validity

Researcher bias: When a single person (primary author) is categorizing a large number of concerns, it is difficult to completely eliminate the researcher bias. To minimize the effect of researcher bias, all categories were defined and the definitions were discussed among the authors. During the categorization process, 10 concerns each in 5 rounds were categorized by the other two authors and discussed among the three authors. The labels were updated and moved during these sessions to better represent the concern types. Once the categorization was completed, 15 randomly selected concerns from the classification were categorized by the other two authors followed by a discussion. While creating the detection method categorization, in addition to defining the categories, 5 well-known defect detection tools were studied to back up the concerns that we categorized as "automatically detected". The concerns that existing tools could not detect but obviously could be accurately and automatically detected were thoroughly discussed among the authors. To minimize the bias during data analysis, we present the data to backup our discussion and conclusions. Also, once the primary author had completed the data analysis, the resulting implications and conclusions were discussed among the authors.

Sampling bias: Sampling bias is a possibility in any study that works with a sample from a population. To minimize this bias, we performed random sampling at the code review discussion level. Furthermore, once the classification was completed, another 5% concerns of the sample size were extracted and categorized for saturation check which demonstrated that the saturation may have reached.

The sample sizes we used to check the saturation and the reliability of our classification have a possibility of being insufficient for a thorough check. However, we discovered the 75:25 evolvability to functional concerns ratio that have been observed in the previous code review defect classification studies [27, 42, 24]. This provides us some confidence that the data we categorized is saturated and reliable.

5.5.2. External Validity

The data set selected for this study had been extracted from a popular code review tool Gerrit Code Review and consists of 7 OSS Java projects. Thus, the generalizability of our results towards closed-source projects, OSS projects that are not considered in this study, and projects created in other programming languages might be limited. However, all of our categories overlap with the categories of other existing classifications (see section 5.4) and obtained the ratio of 75:25 evolvability to functional defects similar to other studies [27, 42, 24]. Due to these reasons, our results may be applicable to many other scenarios. Future research can validate this further.

Our sample size was 385 with 55 or more code review comments extracted from each project. This sample size maybe too small to identify rare and highly project-specific concerns. However, we found that 16% of our sample concerns required custom rules to be detected. These may be highly project-specific concerns and representative of such concerns in the population. Future work can explore project-specific concerns more by classifying larger samples of code review comments.

6. Conclusion

We have presented a classification of concerns identified in MCRs of 7 OSS Java projects [28]. We extracted a sample of 417 code review comments, and, using thematic analysis [29], we identified 116 concern types which we grouped into 15 groups. Additionally, we categorized the concerns based on the automatability of their detection.

The first RQ of this study was to identify the types of concerns that were detected during MCRs. We identified 116 concern types and 15 concern groups. The API Documentation group had the largest number of concerns (68/417 concerns, 16.3%). The Implementation group had the most number of concern types (19/116 concern types, 16.4%). The least number of concerns were related to Threads (1/417, 0.2%). The entire list of concern types and concern groups are available in fig.5.

The second RQ was aimed at identifying the extent to which the concerns identified during MCR could have been automatically detected accurately using existing tools. The results suggested that this was a substantial amount, 22% of concerns in the sample. Additionally, 16% of concerns were not supported by existing, popular tools but automating the detection of these concerns using custom rules was possible. This is another opportunity for further reducing the cost of code review.

Not all concerns require an expert to detect them (see section 5.3.4). Using less expensive non-programmers to conduct code review where possible could also improve the cost of code review. In our sample, 42.5% concerns were categorized as detectable by non-programmers with or without training.

There are concerns that can be partially automatically detected using existing tools. This is another cost reduction opportunity where the application of defect detection tools could help with to a certain extent.

As future work, we will explore the last two cost reduction opportunities discussed above. We expect to answer the research question “Can non-programmers contribute to code review?”. The verified classification may allow organizations to dispatch code review tasks to appropriate expertise-levels and possibly use less expensive resources to conduct code review tasks. Additionally, the extent to which partial automation capabilities of existing tools can help with code review tasks will be explored so that the use of automation and appropriate expertise-level can work together to reduce the code review cost effectively.

References

[1] A. Bosu, M. Greiler, C. Bird, Characteristics of useful code reviews: An empirical study at microsoft, in: in Proc. IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 146–156.

[2] C. Sadowski, E. Söderberg, L. Church, M. Sipko, A. Bacchelli, Modern code review: a case study at google, in: in Proc. 40th International Conference on Software Engineering: Software Engineering in Practice, 2018, pp. 181–190.

[3] D. Distefano, M. Fähndrich, F. Logozzo, P. W. O’Hearn, Scaling static analyses at facebook, Communications of the ACM 62 (8) (2019) 62–70.

[4] S. Nazir, N. Fatima, S. Chuprat, Modern code review benefits-primary findings of a systematic literature review, in: in Proc. 3rd International Conference on Software Engineering and Information Management, 2020, p. 210–215.

[5] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, C. Chockley, Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft, IEEE Transactions on Software Engineering 43 (1) (2017) 56–75.

[6] T. Baum, O. Liskin, K. Niklas, K. Schneider, Factors influencing code review processes in industry, in: in Proc.24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, p. 85–96.

[7] J. Wang, P. C. Shih, J. M. Carroll, Revisiting Linus's law: Benefits and challenges of open source software peer review, International Journal of Human-Computer Studies 77 (2015) 52–65.

[8] A. Bacchelli, C. Bird, Expectations, outcomes, and challenges of modern code review, in: in Proc. 35th International Conference on Software Engineering, 2013, pp. 712–721.

[9] J. Czerwonka, M. Greiler, J. Tilford, Code reviews do not find bugs. how the current code review best practice slows us down, in: in Proc. 37th IEEE International Conference on Software Engineering, Vol. 2, 2015, pp. 27–28.

[10] T. Baum, O. Liskin, K. Niklas, K. Schneider, Factors influencing code review processes in industry, in: in Proc. 24th acm sigsoft international symposium on foundations of software engineering, 2016, pp. 85–96.

[11] C. Staff, Codeflow: Improving the code review process at microsoft, Communications of the ACM 62 (2) (2019) 36–44.

[12] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, A. Bacchelli, Information needs in contemporary code review, Proceedings of the ACM on Human-Computer Interaction 2 (CSCW) (2018) 1–27.

[13] Gerrit code review.
URL `https://www.gerritcodereview.com/`

[14] Collaborator.
URL `https://smartbear.com/product/collaborator/overview/`

[15] Crucible.
URL `https://www.atlassian.com/software/crucible`

[16] Github code review.
URL `https://github.com/features/code-review/`

[17] V. Balachandran, Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation, in: in Proc. 35th International Conference on Software Engineering, 2013, pp. 931–940.

[18] S. Panichella, V. Arnaoudova, M. Di Penta, G. Antoniol, Would static analysis tools help developers with code reviews?, in: in Proc. 22nd International Conference on Software Analysis, Evolution, and Reengineering, 2015, pp. 161–170.

[19] M. Fadhel, Towards automating code reviews (2020).

[20] D. Singh, V. R. Sekar, K. T. Stolee, B. Johnson, Evaluating how static analysis tools can reduce code review effort, in: in Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, 2017, pp. 101–105.

[21] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: in Proc. of the International Conference on Software Engineering, 2013, p. 672–681.

[22] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, A. Zaidman, How developers engage with static analysis tools in different contexts, Empirical Software Engineering (2019) 1–39.

[23] H. Siy, L. Votta, Does the modern code inspection have value?, in: in Proc. International Conference on Software Maintenance, 2001, pp. 281–289.

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

[25] Q. Lei, Z. He, H. Fuqun, L. Bin, Classification of air on-board software code defects and investigations, Procedia Engineering 15 (2011) 3577–3583.

[26] P. Runeson, A. Stefik, A. Andrews, Variation factors in the design and analysis of replicated controlled experiments, Empirical Software Engineering 19 (6) (2014) 1781–1808.

[27] S. Panichella, N. Zaugg, An empirical investigation of relevant changes and automation needs in modern code review, Empirical Software Engineering (2020) 1–40.

[28] M. Paixao, J. Krinke, D. Han, M. Harman, Crop: Linking code reviews to source code changes, in: International Conference on Mining Software Repositories, MSR, 2018.

[29] V. Braun, V. Clarke, Using thematic analysis in psychology, Qualitative research in psychology 3 (2) (2006) 77–101.

[30] Sonarqube rules.
URL `https://docs.sonarqube.org/latest/user-guide/rules/`

[31] Checkstyle standard checks.
URL `https://checkstyle.sourceforge.io/checks.html`

[32] Findbugs - bug descriptions.
URL `http://findbugs.sourceforge.net/bugDescriptions.html`

[33] Pmd.
URL `https://pmd.github.io/`

[34] Intellij idea code inspection.
URL `https://www.jetbrains.com/help/idea/code-inspection.html`

[35] Checkstyle standard checks - nowhitespaceafter.
URL `https://checkstyle.sourceforge.io/config_whitespace.html\#NoWhitespaceAfter`

[36] N. Cassee, B. Vasilescu, A. Serebrenik, The silent helper: the impact of continuous integration on code reviews, in: in Proc. 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, 2020, pp. 423–434.

[37] Checkstyle - type name.
URL `https://checkstyle.sourceforge.io/config_naming.html\#TypeName`

[38] Intellij idea - identifier naming.
URL `https://www.jetbrains.com/help/clion/naming-conventions.html`

[39] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, B. Vasilescu, Dire: A neural approach to decompiled identifier naming, in: in Proc. 34th IEEE/ACM International Conference on Automated Software Engineering, 2019, pp. 628–639.

[40] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, M. Lanza, Investigating the use of code analysis and nlp to promote a consistent usage of identifiers, in: in Proc. IEEE 17th International Working Conference on Source Code Analysis and Manipulation, 2017, pp. 81–90.

[41] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, Learning natural coding conventions, in: in Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 281–293.

[42] M. Beller, A. Bacchelli, A. Zaidman, E. Juergens, Modern code reviews in open-source projects: Which problems do they fix?, in: in Proc. 11th working conference on mining software repositories, 2014, pp. 202–211.

[43] J. Agnelo, N. Laranjeiro, J. Bernardino, Using orthogonal defect classification to characterize nosql database defects, Journal of Systems and Software 159 (2020) 110451.

[44] X. Xia, X. Zhou, D. Lo, X. Zhao, Y. Wang, An empirical study of bugs in software build system, IEICE TRANSACTIONS on Information and Systems 97 (7) (2014) 1769–1780.

[45] U. Hunny, Orthogonal security defect classification for secure software development, Ph.D. thesis (2012).

[46] F. Thung, S. Wang, D. Lo, L. Jiang, An empirical study of bugs in machine learning systems, in: in Proc. 23rd International Symposium on Software Reliability Engineering, 2012, pp. 271–280.

[47] N. Li, Z. Li, X. Sun, Classification of software defect detected by black-box testing: An empirical study, in: 2010 Second World Congress on Software Engineering, Vol. 2, 2010, pp. 234–240.

[48] M. Grottke, A. P. Nikora, K. S. Trivedi, An empirical investigation of fault types in space mission system software, in: in Proc. International Conference on Dependable Systems & Networks (DSN), 2010, pp. 447–456.

[49] Ieee standard classification for software anomalies, IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) (2010) 1–23.

[50] R. S. Pressman, Software engineering: a practitioner's approach, Palgrave macmillan, 2005.

[51] R. C. Michael Inies, Fault links: identifying module and fault types and their relationship, Master's thesis, University of Kentucky (2004).

[52] B. Beizer, Software testing techniques, Dreamtech Press, 2003.

[53] X. Huang, Software reliability, safety and quality assurance, Beijing: Publishing House of Electronics Industry 112 (2002).

[54] C. Kaner, J. Falk, H. Q. Nguyen, Testing computer software, John Wiley & Sons, 1999.

[55] W. S. Humphrey, A discipline for software engineering, Addison-Wesley Longman Publishing Co., Inc., 1995.

[56] R. B. Grady, Practical software metrics for project management and process improvement, Prentice-Hall, Inc., 1992.

[57] L. H. Putnam, W. Myers, Measures for excellence: reliable software on time, within budget, Prentice Hall Professional Technical Reference, 1991.

[58] V. R. Basili, R. W. Selby, Comparing the effectiveness of software testing strategies, IEEE transactions on software engineering (12) (1987) 1278–1296.

[59] F. Lopes, J. Agnelo, C. A. Teixeira, N. Laranjeiro, J. Bernardino, Automating orthogonal defect classification using machine learning algorithms, Future Generation Computer Systems 102 (2020) 932–947.

[60] J. Mabrey, Automated defect classification using machine learning, Ph.D. thesis, North Carolina Agricultural and Technical State University (2020).

[61] I. Chawla, S. K. Singh, An automated approach for bug categorization using fuzzy logic, in: in Proc. of the 8th India Software Engineering Conference, 2015, pp. 90–99.

[62] N. Pingclasai, H. Hata, K.-i. Matsumoto, Classifying bug reports to bugs and other requests using topic modeling, in: in Proc. 20th Asia-Pacific Software Engineering Conference (APSEC), Vol. 2, 2013, pp. 13–18.

[63] C. Liu, Y. Zhao, Y. Yang, H. Lu, Y. Zhou, B. Xu, An ast-based approach to classifying defects, in: in Proc.International Conference on Software Quality, Reliability and Security-Companion, 2015, pp. 14–21.

[64] X. Xia, D. Lo, X. Wang, B. Zhou, Automatic defect categorization based on fault triggering conditions, in: in Proc. 19th International Conference on Engineering of Complex Computer Systems, 2014, pp. 39–48.

[65] L. Yu, C. Kong, L. Xu, J. Zhao, H. Zhang, Mining bug classifier and debug strategy association rules for web-based applications, in: in Proc. International Conference on Advanced Data Mining and Applications, 2008, pp. 427–434.

[66] Code review open platform (crop).
URL `https://crop-repo.github.io/\#structure`

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

[68] R. V. Krejcie, D. W. Morgan, Determining sample size for research activities, Educational and psychological measurement 30 (3) (1970) 607–610.

[69] S. Gunawardena, Supporting documentation - code review cost reduction opportunities.
URL `https://github.com/sgun571/Code-Review-Cost-Reduction-Opportunities`

[70] grammarly.
URL `https://www.grammarly.com/`

[71] Checkstyle - needbraces.
URL `https://checkstyle.sourceforge.io/config_blocks.html\#NeedBraces`

[72] Pmd - controlstatementbraces.
URL `https://pmd.github.io/latest/pmd_rules_java_codestyle.html\#controlstatementbraces`

[73] Intellij idea - control flow statement without braces.
URL `https://www.jetbrains.com/help/idea/list-of-java-inspections.html`

[74] A. M. Jamil, L. b. Othmane, A. Valani, M. Abdelkhalek, A. Tek, The current practices of changing secure software: an empirical study, in: in Proc. 35th Annual ACM Symposium on Applied Computing, 2020, pp. 1566–1575.

[75] M. Tahaei, K. Vaniea, A survey on developer-centred security, in: in Proc. IEEE European Symposium on Security and Privacy Workshops, 2019, pp. 129–138.

[76] T. Thomas, Exploring the usability and effectiveness of interactive annotation and code review for the detection of security vulnerabilities, in: in Proc. IEEE Symposium on Visual Languages and Human-Centric

Computing, 2015, pp. 295–296.

[77] B. Chess, J. West, Secure programming with static analysis, 2007.

[78] V. B. Livshits, M. S. Lam, Finding security vulnerabilities in java applications with static analysis., in: USENIX Security Symposium, Vol. 14, 2005, pp. 18–18.

[79] R. K. McLean, Comparing static security analysis tools using open source software, in: in Proc. Sixth International Conference on Software Security and Reliability Companion, 2012, pp. 68–74.

[80] A. Masood, J. Java, Static analysis for web service security-tools & techniques for a secure development life cycle, in: 2015 IEEE International Symposium on Technologies for Homeland Security (HST), 2015, pp. 1–6.

[81] J. Smith, L. N. Q. Do, E. Murphy-Hill, Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security, in: Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020), 2020, pp. 221–238.

[82] L. N. Q. Do, J. Wright, K. Ali, Why do software developers use static analysis tools? a user-centered study of developer needs and motivations, IEEE Transactions on Software Engineering (Jun 2020).

[83] P. Godefroid, N. Nagappan, Concurrency at microsoft: An exploratory survey, in: CAV workshop on exploiting concurrency efficiently and correctly, 2008.

[84] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, Y. Liu, {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs, in: 29th {USENIX} Security Symposium ({USENIX} Security 20), 2020, pp. 2325–2342.

[85] F. Eichinger, V. Pankratius, P. W. Große, K. Böhm, Localizing defects in multithreaded programs by mining dynamic call graphs, in: International Academic and Industrial Conference on Practice and Research Techniques, 2010, pp. 56–71.

[86] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, S. Ur, Framework for testing multi-threaded java programs, Concurrency and Computation: Practice and Experience 15 (3-5) (2003) 485–499.

[87] M. A. Al Mamun, A. Khanam, H. Grahn, R. Feldt, Comparing four static analysis tools for java concurrency bugs, in: Third Swedish Workshop on Multi-Core Computing (MCC-10), 2010, pp. 18–19.

[88] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, C. Tian, Dcatch: Automatically detecting distributed concurrency bugs in cloud systems, ACM SIGARCH Computer Architecture News 45 (1) (2017) 677–691.

[89] D. Kester, M. Mwebesa, J. S. Bradbury, How good is static analysis at finding concurrency bugs?, in: 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, 2010, pp. 115–124.

[90] Code conventions for the java tm programming language - if.
URL `https://www.oracle.com/java/technologies/javase/codeconventions-statements.html#449`

[91] R. S. Laramee, Bob's concise coding conventions (c3), Advances in Computer Science and Engineering (ACSE) 4 (1) (2010) 23–26.

[92] Coding conventions.
URL `https://en.wikipedia.org/wiki/Coding_conventions`

[93] Checkstyle standard checks - linelength.
URL `https://checkstyle.sourceforge.io/config_sizes.html\#LineLength`

[94] Intellij idea - line is longer than allowed by code style.
URL `https://www.jetbrains.com/help/phpstorm/general-line-is-longer-than-allowed-by-code-style.html`

[95] Quality profiles - sonarqube.
URL `https://docs.sonarqube.org/latest/instance-administration/quality-profiles/`

[96] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: in Proc. 35th International Conference on Software Engineering (ICSE), 2013, pp. 672–681.

[97] D. Singh, V. R. Sekar, K. T. Stolee, B. Johnson, Evaluating how static analysis tools can reduce code review effort, in: in Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2017, pp. 101–105.

[98] Findbugs - return value of method without side effect is ignored.
URL `http://findbugs.sourceforge.net/bugDescriptions.html#RV_RETURN_VALUE_IGNORED_NO_SIDE_EFFECT`

[99] T. Gilb, D. Graham, Software inspections, Addison-Wesley Reading, Masachusetts, 1993.

[100] D. O'Neill, Issues in software inspection, IEEE Software 14 (1) (1997) 18–19.

[101] T. Hall, D. Wilson, N. Baddoo, Towards implementing successful software inspections, in: Proceedings International Conference on Software Methods and Tools. SMT 2000, 2000, pp. 127–136.

[102] M. Ciolkowski, O. Laitenberger, S. Biffl, Software reviews, the state of the practice, IEEE software 20 (6) (2003) 46–51.

[103] Y.-K. Wong, An exploratory study of software review in practice, in: PICMET'03: Portland International Conference on Management of Engineering and Technology Technology Management for Reshaping the World, 2003., 2003, pp. 301–308.

[104] J.-S. Oh, H.-J. Choi, A reflective practice of automated and manual code reviews for a studio project, in: Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05), 2005, pp. 37–42.

[105] L. Harjumaa, I. Tervonen, A. Huttunen, Peer reviews in real life-motivators and demotivators, in: Fifth International Conference on Quality Software (QSIC'05), 2005, pp. 29–36.

[106] P. Sliz, A. Morin, Optimizing peer review of software code, Science 341 (6143) (2013) 236–237.

[107] S. Jayatilake, S. De Silva, U. Settinayake, S. Yapa, J. Jayamanne, A. Ruwanthika, C. Manawadu, Role of software inspections in the sri lankan software development industry, in: 2013 8th International Conference on Computer Science & Education, 2013, pp. 697–702.

[108] G. Gousios, A. Zaidman, M.-A. Storey, A. Van Deursen, Work practices and challenges in pull-based development: The integrator's perspective, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, 2015, pp. 358–368.

[109] O. Kononenko, O. Baysal, M. W. Godfrey, Code review quality: How developers see it, in: Proceedings of the 38th international conference on software engi-

neering, 2016, pp. 1028–1038.
[110] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, J. Czerwonka, Code reviewing in the trenches: Challenges and best practices, IEEE Software 35 (4) (2017) 34–42.
[111] T. Baum, H. Leßmann, K. Schneider, The choice of code review process: A survey on the state of the practice, in: International Conference on Product-Focused Software Process Improvement, 2017, pp. 111–127.
[112] F. Ebert, F. Castor, N. Novielli, A. Serebrenik, An exploratory study on confusion in code reviews, Empirical Software Engineering 26 (1) (2021) 1–48.
[113] P. M. Johnson, An instrumented approach to improving software quality through formal technical review, in: Proceedings of 16th International Conference on Software Engineering, 1994, pp. 113–122.
[114] F. Belli, R. Crisan, Towards automation of checklist-based code-reviews, in: Proceedings of ISSRE'96: 7th International Symposium on Software Reliability Engineering, 1996, pp. 24–33.
[115] A. A. Porter, H. P. Siy, C. A. Toman, L. G. Votta, An experiment to assess the cost-benefits of code inspections in large scale software development, IEEE transactions on software engineering 23 (6) (1997) 329–346.
[116] F. Belli, R. Crisan, Empirical performance analysis of computer-supported code-reviews, in: Proceedings The Eighth International Symposium on Software Reliability Engineering, 1997, pp. 245–255.
[117] K. Chan, An agent-based approach to computer assisted code inspections, in: Proceedings 2001 Australian Software Engineering Conference, 2001, pp. 147–152.
[118] D. Kelly, T. Shepard, Task-directed software inspection, Journal of Systems and Software 73 (2) (2004) 361–368.
[119] E. Farchi, S. Ur, Selective homeworkless reviews, in: 2008 1st International Conference on Software Testing, Verification, and Validation, 2008, pp. 404–413.
[120] J. Ratcliffe, Moving software quality upstream: The positive impact of lightweight peer code review, in: Pacific NW software quality conference, 2009, pp. 1–10.
[121] B. Xu, Cost efficient software review in an e-business software development project, in: 2010 International Conference on E-Business and E-Government, 2010, pp. 2680–2683.
[122] V. Balachandran, Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation, in: in Proc. 35th International Conference on Software Engineering, 2013, pp. 931–940.
[123] S. Misra, L. Fernández, R. Colomo-Palacios, A simplified model for software inspection, Journal of software: evolution and process 26 (12) (2014) 1297–1315.
[124] C. Staff, Codeflow: improving the code review process at microsoft, Communications of the ACM 62 (2) (2019) 36–44.
[125] S. Rebai, A. Amich, S. Molaei, M. Kessentini, R. Kazman, Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations, Automated Software Engineering (2020) 1–28.
[126] Z. Xia, H. Sun, J. Jiang, X. Wang, X. Liu, A hybrid approach to code reviewer recommendation with collaborative filtering, in: 2017 6th International Workshop on Software Mining (SoftwareMining), 2017, pp. 24–31.
[127] S. McIntosh, Y. Kamei, B. Adams, A. E. Hassan, An empirical study of the impact of modern code review practices on software quality, Empirical Software Engineering 21 (5) (2016) 2146–2189.
[128] M. M. Rahman, C. K. Roy, J. A. Collins, Correct: code reviewer recommendation in github based on cross-project and technology experience, in: Proceedings of the 38th International Conference on Software Engineering Companion, 2016, pp. 222–231.
[129] Y. Wang, X. Wang, Y. Jiang, Y. Liang, Y. Liu, A code reviewer assignment model incorporating the competence differences and participant preferences, Foundations of Computing and Decision Sciences 41 (1) (2016) 77–91.
[130] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, K.-i. Matsumoto, Who should review my code? a file location-based code-reviewer recommendation approach for modern code review, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 141–150.