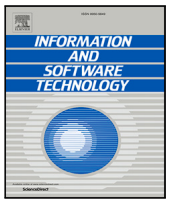
Information and Software Technology 177 (2025) 107596

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Deciphering refactoring branch dynamics in modern code review: An empirical study on Qt

Eman Abdullah AlOmar

Stevens Institute of Technology, Hoboken, NJ, USA

ARTICLE INFO

Dataset link: https://smilevo.github.io/self-affirmed-refactoring/

Keywords:
Refactoring
Code review
Developer perception
Software quality

ABSTRACT

Context: Modern code review is a widely employed technique in both industrial and open-source projects, serving to enhance software quality, share knowledge, and ensure compliance with coding standards and guidelines. While code review is extensively studied for its general challenges, best practices, outcomes, and socio-technical aspects, little attention has been paid to how refactoring is reviewed and what developers prioritize when reviewing refactored code in the 'Refactor' branch.
Objective: The goal is to understand the review process for refactoring changes in the 'Refactor' branch and to identify what developers care about when reviewing code in this branch.
Method: In this study, we present a quantitative and qualitative examination to understand the main criteria developers use to decide whether to accept or reject refactored code submissions and identify the challenges inherent in this process.
Results: Analyzing 2154 refactoring and non-refactoring reviews across Qt open-source projects, we find that reviews involving refactoring from the 'Refactor' branch take significantly less time to resolve in terms of code review efforts. Additionally, documentation of developer intent is notably sparse within the 'Refactor' branch compared to other branches. Furthermore, through thematic analysis of a substantial sample of refactoring code review discussions, we construct a comprehensive taxonomy consisting of 12 refactoring review criteria.
Conclusion: Our findings underscore the importance of developing precise and efficient tools and techniques to aid developers in the review process amidst refactorings.

1. Introduction

Refactoring is a crucial practice for maintaining code quality as software evolves. Its significance has expanded beyond mere code cleanup to become a cornerstone of modern software development. This has attracted significant attention from researchers, evident in the numerous research papers dedicated to the topic [1]. Another key practice in maintaining software quality is code review [2]. It has become another important to reduce technical debt, and to detect potential coding errors [2–4]. Code review represents the manual inspection of any newly performed changes to the code, for the purpose of verifying integrity, compliance with standards, and error-freedom [5]. Today's Modern Code Review (MCR) process is typically lightweight and tool-based, relying heavily on discussions between authors and reviewers to decide whether to merge or discard a code change [6].

Refactoring changes, like any other code modifications, must undergo review before being merged. Failure to apply refactoring properly can lead to adverse effects, including compromised software quality [7–10] and inducing bugs [11,12] making refactoring changes more challenging to review. However, little is known about how reviewers *examine* refactoring related code changes, especially when it is intended to serve the same *purpose* of improving software quality. According to the industrial case study, AlOmar et al. [13] has found that reviewing refactoring related code changes takes a significantly longer time, in comparison with other code changes, demonstrating the need for refactoring review *culture*. Yet, little is known about what criteria reviewers consider when they review refactoring. Most of refactoring studies focus on its automation by recommending refactoring opportunities in the source code [14–16], or mining performed refactorings in change histories of software repositories [17]. Moreover, while research on code reviews has concentrated on automation, such as recommending the most suitable reviewer for a given code change [2], the review process for refactoring changes in 'Refactor' branch remains largely unexplored.

Building upon our previous research [13,20], which revealed that refactoring code reviews often take longer to be approved, we now delve deeper into the specific practices of refactoring review within an ecosystem featuring a dedicated 'Refactor' branch. This investigation aims to provide insights into addressing the challenges identified in

E-mail address: ealomar@stevens.edu.

https://doi.org/10.1016/j.infsof.2024.107596
Received 20 May 2024; Received in revised form 10 August 2024; Accepted 1 October 2024
Available online 5 October 2024

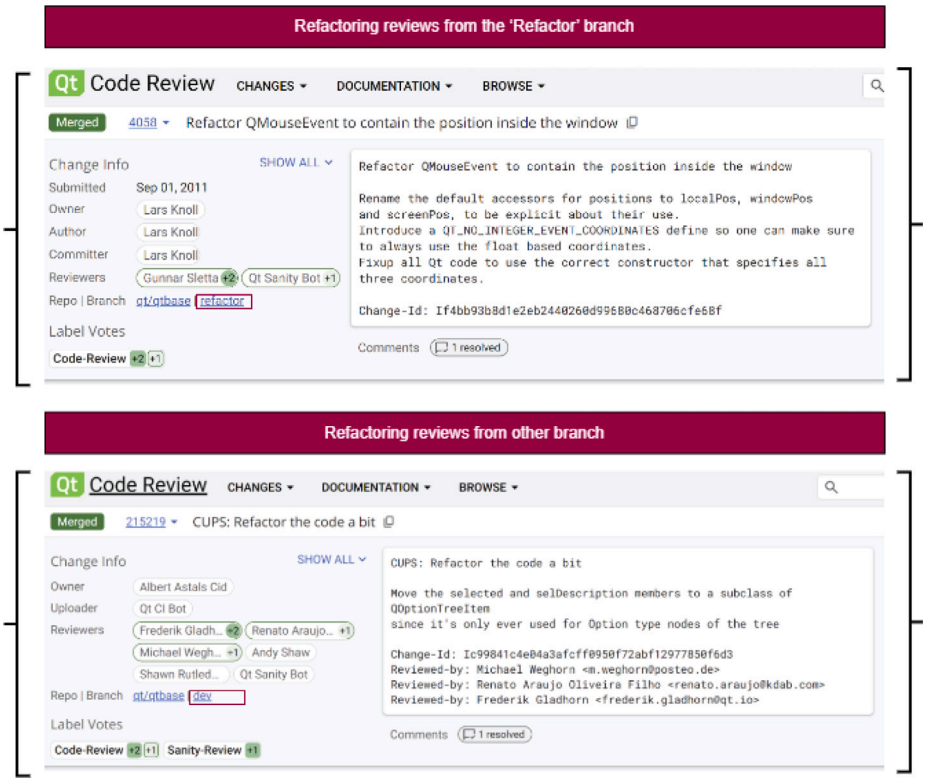


Fig. 1. Example of a code review from Qt project using Gerrit [18,19].

our previous studies, thus contributing to a more efficient and effective refactoring review process. Specifically, the goal of this paper is to understand the criteria developers use when reviewing refactored code in the 'Refactor' branch, focusing on what influences their decisions to accept or reject submissions.

In our study, we explore two distinct venues where refactoring can occur: the typical branch and the refactoring branch. The typical branch refers to the standard development workflow where incremental changes and features are integrated. In contrast, the refactoring branch is specifically dedicated to restructuring and improving the codebase without altering its external behavior. Fig. 1 illustrates a segment of the refactoring review from the 'Refactor' branch and other branches within the Qt system. The top example shows a code review that was created to refactor `QMouseEvent` to contain the position inside the window. The developers submitted the code, while explicitly stating that the refactoring was intended to rename the default accessors for positions to `localPos`, `windowPos` and `screenPos`, to be explicit about their use. This can also be seen in the final subject and description of the review that later merged the modified code into production. Based on this example, we can see that *rename* was one of the refactoring operations that developers consider for code optimization, and author confirms its quality improvement as follows "*There is no behavioral change. All I did was add the windowPos(), and make sure it is set correctly everywhere. Old code will continue to work just as before.*" In contrast, the bottom example reveals changes to the `QtBase` module. It focuses on enhancing the code by removing unnecessary elements, improving readability, and making the code more maintainable. The review includes detailed comments from developers discussing the impact and necessity of these changes, aiming to ensure that the refactoring maintains the module's functionality while improving its structure. This review had a longer review duration and more extensive discussion compared to the 'Refactor' branch reviews. We believe that early clarification of the 'Refactor' and the typical branch is crucial for understanding their roles in the development process. The typical branch often prioritizes feature development and bug fixes, while the refactoring branch focuses on enhancing code quality, maintainability, and performance. By distinguishing between these two branches, we aim to highlight the unique challenges and considerations associated with each, thereby providing a comprehensive understanding of their impact on the software development lifecycle.

Therefore, we conduct our study using the following overarching question: *How do developers approach and evaluate refactoring tasks compared to non-refactoring tasks during code reviews, and what patterns, quality attributes, and topics are emphasized in these reviews?*

To answer our research questions, we first extracted a set of 718 refactoring-related code reviews in the 'Refactor' branch, from the Qt ecosystem. Then, we compared this set of refactoring-related code reviews in 'Refactor' branch, with another two sets of code reviews, in terms of the number of reviewers, number of review comments, number of inline comments, number of revision, number of changed files, review duration, discussion and description length, and code churn. Our empirical investigation indicates that refactoring-related code reviews from 'Refactor' branch take significantly shorter to be resolved and typically trigger fewer discussions between developers and reviewers to reach a consensus. To understand the key characteristics of reviewing refactored code, we perform a thematic analysis on a significant sample of these reviews. This process resulted in a hierarchical taxonomy composed of four categories, and 12 sub-categories.

We provide our experiments package [21] to further replicate and extend our study. The package contains raw data, analyzed data, statistical test results, survey questions, and custom-built scripts used in our research.

The remainder of this paper is organized as follows. Section 2 provides background on the Gerrit-based code review process. Section 3 reviews the existing studies related to refactoring awareness and code review. Section 4 outlines our empirical setup in terms of data collection, analysis and research question. Section 5 discusses our findings, while the research implication is discussed in Section 6. Section 7 captures any threats to the validity of our work, before concluding with Section 8.

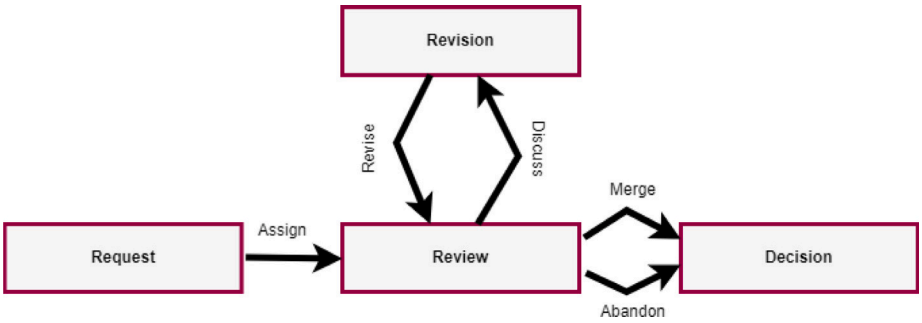


Fig. 2. Gerrit-based code review process overview.

2. Background

Code review involves the manual assessment of source code by humans to identify defects and quality issues [22]. However, traditional code review practices have limitations when applied to globally distributed software development [23]. In recent years, Modern Code Review (MCR) has emerged as a tool-based system that is less formal than traditional methods. MCR has become popular and is widely used in both proprietary software (*e.g.,* Google, Microsoft) and open-source software (*e.g.,* OpenStack, Qt) [2]. In this study, we selected Gerrit as it provides robust code review functionality that is essential to maintain code quality and ensure thorough review processes. This feature is crucial for our research's needs, where code integrity and peer review are paramount. In the following, we provide a brief overview of the Gerrit-based code review system, a prominent tool frequently used in previous studies [24–27].

The code review process of the systems studied is based on Gerrit,[1] a collaborative code review framework. Gerrit facilitates developers in tagging submitted code changes directly and requesting their assignment to a reviewer. Generally, a code change author opens a code review request containing a title, a detailed description of the code change being submitted, written in natural language, and the current code changes annotated. Once the review request is submitted, it appears in the requests backlog, and be open to reviewers to choose from. Once reviewers are assigned to the review request, they inspect the proposed changes and comment on the review request's thread, to start a discussion with the author. This way, the authors and reviewers can discuss the submitted changes, and reviewers can request revisions to the code being reviewed. Following up discussions and revisions, a review decision is made to either accept or decline, and so the proposed code changes are either "*Merged*" to production or "*Abandoned*".

A diagram, modeling a simplified bird's view of the Gerrit-based code review process, is shown in Fig. 2. It begins with a "Request" for a review, which is then assigned to a reviewer. The "Review"" phase follows, where the code is assessed and discussed. If changes are needed, the code enters the "Revision" phase, where it is revised based on feedback and can be discussed further. This cycle continues until a decision is made to either "Merge" the code into the main branch or "Abandon" the changes.

3. Related work

Research on code review has been of importance to practitioners and researchers. A considerable effort has been spent by the research community in studying traditional and modern code review practices and challenges. The literature has included case studies (*e.g.,* [3,13,24,25,28–30]), user studies (*e.g.,* [31–35]), surveys (*e.g.,* [2,13,36,37]), and empirical experiments (*e.g.,* [25,30,32,38,39]). However, most of the above studies focus on studying and improving the effectiveness of modern code review in general, as opposed to our work that focuses on understanding developers' perception of code review involving refactoring. In this section, we are only interested in research related to refactoring-aware code review.

In a study performed at Microsoft, Bacchelli and Bird [2] observed, and surveyed developers to understand the challenges faced during code review. They pointed out purposes for code review (*e.g.,* improving team awareness and transferring knowledge among teams) along with the actual outcomes (*e.g.,* creating awareness and gaining code understanding). In a similar context, MacLeod et al. [37] interviewed several teams at Microsoft and conducted a survey to investigate the human and social factors that influence developers' experiences with code review. Both studies found the following general code reviewing challenges: (1) finding defects, (2) improving the code, and (3) increasing knowledge transfer. Ge et al. [28,29] developed a refactoring-aware code review tool, called ReviewFactor, that automatically detects refactoring edits and separates refactoring from non-refactoring changes with a focus on five refactoring types. The tool was intended to support developers' review process by distinguishing between refactoring and non-refactoring changes, but it does not provide any insights on the quality of the performed refactoring. Inspired by the work of [28,29], Alves et al. [34,40] proposed a static analysis tool, called RefDistiller, that helps developers inspect manual refactoring edits. The tool compares two program versions to detect refactoring anomalies' type and location. It supports six refactoring operations, detects incomplete refactorings, and provides inspection for manual refactorings.

Coelho et al. [41] performed a systematic literature mapping study on refactoring tools to support modern code review. They raised the need for more tools to explain composite refactorings. They also reported the need for more surveys to assess the existing refactoring tools for modern code review in both open-source and industrial projects. Pascarella et al. [42] investigated the effect of code review on bad programming practices (*i.e.,* code smells). Their approach mainly focused on comparing code smells at the file level before and after the code review process. Additionally, they manually investigated whether the severity of code smells was reduced in a code review or not. Their results show that in 95% of the cases, the severity of code smells does not decrease with a review. The reduction in code smells in the remaining few cases was impacted by code insertion and refactoring-related changes.

Paixão et al. [43] explored if developers' intents influence the evolution of refactorings during the review of a code change by mining 1780 reviewed code changes from 6 open-source systems. Their main findings show that refactorings are most often used in code reviews that implement new features, accounting for 63% of the code changes we studied. Only in 31% of the code reviews that employed refactorings the developers had the explicit intent of refactoring. Uchôa et al. [44] reported the multi-project retrospective study that characterizes how the process of design degradation evolves within each review and across multiple reviews. The authors utilized software metrics to observe the influence of certain code review practices on combating design degradation. The authors found that the majority of code reviews

[1] https://www.gerritcodereview.com/

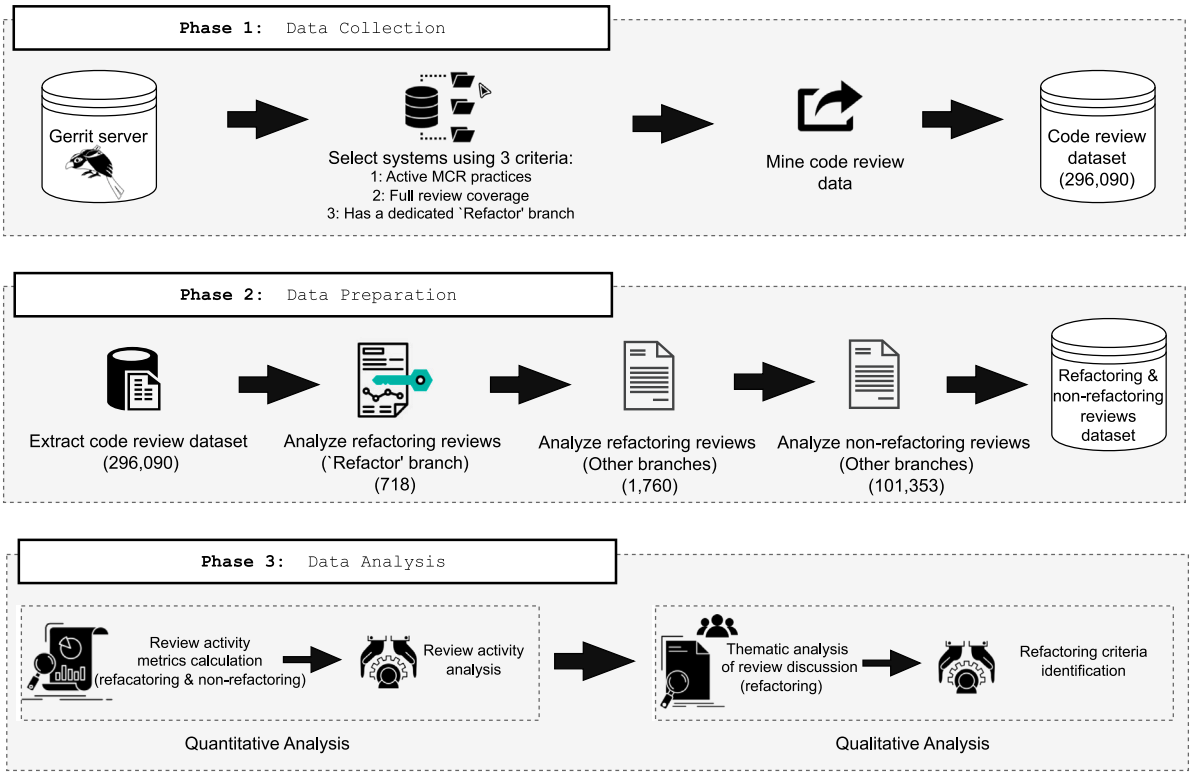


Fig. 3. Overview of our experiment design.

had little to no design degradation impact in the analyzed projects. Additionally, the practices of long discussions and the high proportion of review disagreement in code reviews were found to increase design degradation. In their study on predicting design impactful changes in modern code review with technical and/or social aspects, Uchôa et al. [45] analyzed reviewed code changes from seven open source projects. By evaluating six machine learning algorithms, the authors found that technical features result in more precise predictions and the use of social features alone also leads to accurate predictions.

A couple of studies considered pull requests as a main source of the study code review process. Pantiuchina et al. [46] presented a mining-based study to investigate why developers are performing refactoring in the history of 150 open source systems. Particularly, they analyzed 551 pull requests implemented refactoring operations and reported a refactoring taxonomy that generalizes the ones existing in the literature. Coelho et al. [47] performed a quantitative and qualitative study exploring code reviewing-related aspects intending to characterize refactoring-inducing pull requests. Their main finding show that refactoring-inducing pull requests take significantly more time to merge than non-refactoring-inducing pull requests.

AlOmar et al. [13] conducted a case study in an industrial setting to explore refactoring practices in the context of modern code review from the following five dimensions: (1) developers motivations to refactor their code, (2) how developers document their refactoring for code review, (3) the challenges faced by reviewers when reviewing refactoring changes, (4) the mechanisms used by reviewers to ensure the correctness after refactoring, and (5) developers and reviewers assessment of refactoring impact on the source code's quality. Their findings show that refactoring code reviews take longer to be completed than non-refactoring code reviews. In a follow-up work, AlOmar et al. [20] performed an empirical study on OpenStack to understand the challenges developers faced when reviewing refactoring changes. Their findings corroborate the results of their industrial case study, indicating that refactoring changes require more time for acceptance compared to non-refactoring changes. Brito and Valente [48] introduced RAID, a refactoring-aware and intelligent diff tool to alleviate the cognitive effort associated with code reviews. The tool relied on RefDiff [49] and is fully integrated with the state-of-the-art practice of continuous integration pipelines (GitHub Actions) and browsers (Google Chrome). The authors evaluated the tool with eight professional developers and found that RAID indeed reduced the cognitive effort required for detecting and reviewing refactorings. In another study, Kurbatova et al. [50] presented RefactorInsight, a plugin for IntelliJ IDEA that integrates information about refactorings in diffs in the IDE, auto folds refactorings in code diffs in Java and Kotlin, and shows hints with their short descriptions.

To summarize, the study of open source projects that use either the Gerrit tools or GitHub pull requests has been extensively studied (*e.g.,* [24,46,51,52]). Since notable open-source organizations such as Eclipse, OpenStack, and Qt adopted Gerrit as their code review management tool, we chose to analyze refactoring practices in modern code review from projects that adopted Gerrit as their code review tool. Although there are recent studies that explored the motivation behind refactoring in pull requests [46,47], to the best of our knowledge, no prior studies have manually extracted all the criteria developers are facing when submitting their refactored code for review from a dedicated 'Refactor' branch. To gain a more in-depth understanding of factors mostly associated with refactoring review and to advance the understanding of refactoring-aware code review, in this paper, we performed an empirical study on a rapidly evolving open-source project. This study complements the existing efforts that are done in an industrial environment [13] and open-source systems [20,46,47] using Gerrit and GitHub pull-based development.

4. Study design

The main goal of our study is to understand the practice of refactoring in the context of Modern Code Review (MCR) to characterize the criteria that influence decision making when reviewing refactoring changes. Thus, we aim to answer the following research questions:

- **RQ$_1$**. *How do refactoring reviews compare to non-refactoring reviews in terms of code review efforts?*
- **RQ$_2$**. *What textual patterns do developers use to describe their refactoring needs in the 'Refactor' branch?*
- **RQ$_3$**. *What quality attributes do developers consider when describing refactoring in the 'Refactor' branch?*
- **RQ$_4$**. *What topics do developers discuss when reviewing refactoring tasks?*

According to the guidelines reported by Runeson and Höst [53], we designed an empirical study that consists of three steps, as depicted in Fig. 3, and discussed in the next subsections. Since our research questions are both quantitative and qualitative, we used tools/scripts along with manual activities to investigate our data. Furthermore, the dataset utilized in this study is available on our project website [21] for extension and replication purposes.

4.1. Data collection

4.1.1. Studied systems

In line with [30,54,55], to select the subject systems, we identified three important criteria:

Criterion #1: Active MCR practices. Our goal is to study a system that actively examines code changes through a code review tool. Therefore, we focus on systems where a number of reviews are performed using a code review tool (*i.e.,* systems which have review procedures in place), similar to [30,54,55].

Criterion #2: Full review coverage. Since we investigate the practice of refactoring-related code reviews, we focus on systems that have many files with 100% review coverage (*i.e.,* files where every change made to them is reviewed before they are merged into the repositories), similar to studies that explored code review practices in defective files [25,54,55].

Criterion #3: Has a dedicated 'Refactor' branch. Since we want to study refactoring practices in MCR, we need to ensure that the subject systems have sufficient refactoring-related instances to help us perform our statistical analysis. So, we selected the project with the highest number of refactoring reviews.

To satisfy criterion 1, we started by considering five systems (*i.e.,* OpenStack,[2] Qt,[3] LibreOffice,[4] VTK,[5] ITK[6]) that use the Gerrit code review tool and have been widely studied in previous research in MCR, *e.g.,* [56–59]. We then discarded VTK and ITK since Thongtanunam et al. [55] reported that the linkage rate of code changes to the reviews for VTK is too low and ITK does not satisfy criterion 2. As for criterion 3, after mining the code review data, we found that Qt is the only system with a 'Refactor' branch. Due to the human-intensive nature of carefully studying and analyzing refactoring practice in MCR, we opt for performing an in-depth study on a single system. With the above-mentioned criteria in mind, we select Qt, a cross-platform application and user interface framework developed by Digia Corporation.

4.1.2. Mining code review data

We mined code review data using the `RESTful API`[7] provided by Gerrit, which returns the results in a `JSON` format. We used a script to automatically mine the review data and store them in the `SQLite` database. All collected reviews are closed (*i.e.,* having a status of either "*Merged*" or "*Abandoned*"). In total, we mined 296,372 code changes between December 2012 and April 2021 from Qt projects. An overview of the project's statistics is provided in Table 1.

[2] https://review.opendev.org/
[3] https://codereview.qt-project.org/
[4] https://gerrit.libreoffice.org/
[5] http://vtk.org/
[6] http://itk.org/
[7] https://gerrit-review.googlesource.com/Documentation/rest-apichanges.html

Table 1
Overview of the Qt studied system.

Item	Count
Version	4.7 to 5.11
Line of code	21,256,665
No. of commits	1,659,190
No. of code changes	296,372
No. of developers	3,264
No. of files	351,387
Reviews in 'Refactor' branch	718
Reviews with keyword '*refactor**' in title and description	1,760
Non-refactoring reviews from other branches	101,353

4.2. Data preparation

Our main goal is to explore refactoring review *culture* in 'Refactor' branch. However, to make a comparison, we select refactoring reviews containing the keyword '*refactor*'. Similarly to previous work on identifying refactoring changes or defect-fixing or defect-inducing changes [5,8,46,47,55,60–68], we utilize a keyword-based mechanism to extract refactoring code review data from other branches. The keyword-based approach was chosen for the manual inspection, which required not only non-trivial efforts but also a deep knowledge of the domain. Specifically, we start by searching for the term '*refactor**' in the title or description (we use * to capture extensions like refactors, refactoring etc.). The choice of '*refactor*', besides being used by various related studies, is intuitively the first term to identify refactoring-related code review. However, since related work on refactoring documentation shows that developers may use other synonymous terms/phrases [64,66,69,70], we ensure to exclude these synonymous terms/phrases when selecting non-refactoring reviews. In summary, we have extracted the following reviews.

- *Refactoring reviews from the 'Refactor' branch.* These reviews are specifically chosen from a dedicated 'Refactor' branch.
- *Refactoring reviews containing the keyword 'refactor'.* These reviews include the keyword '*refactor*' in their title and description, and are selected from branches other than the 'Refactor' branch.
- *Non-refactoring reviews from other branches.* These reviews are selected from branches other than the 'Refactor' branch, and they do not involve refactoring or any synonymous terms/phrases commonly found in the literature.

To extract the set of refactoring-related code reviews, we follow a two-step procedure: (1) automatic filtering, and (2) manual filtering.

(1) Automatic Filtering. In the first step, we extract all of the 718 review instances in the 'Refactor' branch. We notice that the ratio of these reviews is very small in comparison with the total number of the mined reviews, *i.e.,* 296,372.

(2) Manual Filtering. To ensure the correctness of the data, we manually inspected and read all these refactoring reviews.

Our goal is to have a *gold set* of reviews in which the developers explicitly reported the refactoring activity. This *gold set* will serve to check later criteria that are mostly associated with refactoring review discussion. Furthermore, since related work on refactoring documentation shows that developers may use synonymous terms/phrases [64,66,69,70], we ensure to exclude these synonymous terms/phrases and manually inspect them when selecting non-refactoring reviews.

4.3. Data analysis

To address our research questions, a structured mixed-method study was designed to combine elements of both quantitative and qualitative research.

Table 2
Statistics of code review activity efforts.

Metrics	*Refactoring code review ('refactor' branch)*						*Non-refactoring code review*						*Statistical difference*	
	Min	Q1	Median	Mean	Q3	Max	Min	Q1	Median	Mean	Q3	Max	*p*-value	Cliff's delta (δ)
Number of reviewers	0	2	3	2.93	4	7	0	1	2	3.01	4	8	**0**	small (0.11)
Number of review comments	1	3	3	4.67	5	8	1	3	6	9.20	11	22	**0**	medium (0.3)
Number of inline comments	0	0	0	1.12	0	0	0	0	0	3.72	2	5	**0**	medium (0.34)
Number of revisions	1	1	1	1.87	2	3	1	1	1	2.44	2	3	**0.000211**	small (0.09)
Number of changed files	0	1	2	49.84	4	8	0	1	2	9.08	4	8	0.6369	small (0.01)
Review duration (seconds)	0	0.15	1.08	90.58	20.66	48.41	0	16.96	163.67	2356.32	1515.01	3719.12	**0**	medium (0.5)
Length of discussion (characters)	22	150	190	427.32	298	518	9	135	341.50	2160.06	1008	2312	**1.621e−14**	small (0.2)
Length of description (characters)	64	96	123	172.12	200	355	55	102	163	260.2	295	584	**1.239e−8**	small (0.15)
Code churn	0	4	14	1366.49	67	161	0	5	22	364.82	81	191	**0.005045**	small (0.07)
Metrics	*Refactoring code review ('refactor' branch)*						*Refactoring code review (Other branches)*						*Statistical difference*	
	Min	Q1	Median	Mean	Q3	Max	Min	Q1	Median	Mean	Q3	Max	*p*-value	Cliff's delta (δ)
Number of reviewers	0	2	3	2.93	4	7	1	3	4	3.82	5	8	**0**	small (0.27)
Number of review comments	1	3	3	4.67	5	8	1	4	8	12.84	16	34	**0**	medium (0.43)
Number of inline comments	0	0	0	1.12	0	0	0	0	0	5.55	4	10	**0**	medium (0.38)
Number of revisions	1	1	1	1.87	2	3	1	2	3	4.13	5	9	**0**	medium (0.45)
Number of changed files	0	1	2	49.84	4	8	0	1	3	7.82	8	18	**2.187e−14**	small (0.2)
Review duration (seconds)	0	0.15	1.08	90.58	20.66	48.41	0	21.37	126.09	1342.24	1917.12	3719.12	**0**	large (0.56)
Length of discussion (characters)	22	150	190	427.32	298	518	36	198	513	6571.11	1528	3469	**0**	medium (0.37)
Length of description (characters)	64	96	123	172.12	200	355	68	115	206.5	304.15	356	715	**0**	medium (0.31)
Code churn	0	4	14	1366.49	67	161	0	39	127.5	425.26	356	822	**0**	medium (0.44)

4.3.1. Quantitative data analysis

We leverage the data collected to compare refactoring and non-refactoring reviews using review efforts, *i.e.,* code review metrics. As we calculate the metrics of refactoring and non-refactoring code reviews, we want to distinguish, for each metric, whether the variation is statistically significant. We first test for normality using the Shapiro–Wilk normality test [71] and observe that the distribution of code review activity metrics does not follow a normal distribution. Therefore, we use the Mann–Whitney U test [72], a non-parametric test, to compare between the two groups, since these groups are independent of one another. The null hypothesis is defined by no variation in the metric values of refactoring and non-refactoring code reviews. Thus, the alternative hypothesis indicates that there is a variation in the metric values. Additionally, the variation between values of both sets is considered significant if its associated *p*-value is less than 0.05. Furthermore, we use the Cliff's Delta (δ) [73], a non-parametric effect size measure, to estimate the magnitude of the differences between refactoring and non-refactoring reviews. As for its interpretation, we follow the guidelines reported by Romano et al. [74]:

- Negligible for $|\delta| < 0.147$
- Small for $0.147 \leq |\delta| < 0.33$
- Medium for $0.33 \leq |\delta| < 0.474$
- Large for $|\delta| \geq 0.474$

To measure the extent of the relationship between these metrics, we conducted a Spearman rank correlation test (a non-parametric measure) [75]. We chose a rank correlation because this type of correlation is resilient to data that is not normally distributed.

4.3.2. Qualitative data analysis

To answer RQ_2, RQ_3, and RQ_4, we perform the analysis of the data. The author manually inspects refactoring review subject, description, and discussions by considering both the general comments and the inline comments. Next, we describe the methodology for building and refining the taxonomy.

Taxonomy Building and Refinement. The goal of the manual analysis was to categorize the topics discussed in the 'Refactor' branch within each of the refactoring review instances. The entire process was supported by a spreadsheet application equipped with tagging capabilities. For each instance, the evaluator was presented with: (i) the metadata as returned by Gerrit (*e.g.,* Gerrit_Id, Gerrit_URL); (ii) the branch that was matched in that specific instance; and (iii) the subject and description in Gerrit for easier inspection.

The categorization required the assignment of one or more labels to an instance, describing the topics discussed. In case manual inspection revealed that reviews were not actually used for refactoring tasks, the instance was discarded.

When analyzing the review discussions, we adopted a thematic analysis approach based on the guidelines provided by Cruzes et al. [76]. Thematic analysis is one of the most used methods in Software Engineering literature (*e.g.,* [77]), which is a technique for identifying and recording patterns (or "themes") within a collection of descriptive labels, which we call "codes". For each refactoring review, we proceeded with the analysis using the following steps: (i) Initial reading of the review discussions; (ii) Generating initial codes (*i.e.,* labels) for each review; (iii) Translating codes into themes, sub-themes, and higher-order themes; (iv) Reviewing the themes to find opportunities for merging; (v) Defining and naming the final themes, and creating a model of higher-order themes and their underlying evidence.

The above-mentioned steps were performed independently by two annotators. One annotator performed the labeling of review discussions independently of the other author who was responsible for reviewing the taxonomy currently drafted. By the end of each iteration, the authors met and refined the taxonomy.

It is important to note that the approach is not a single-step process. As the codes were analyzed, some of the first cycle codes were subsumed by other codes, relabeled, or dropped altogether. As the author progressed with the translation to themes, there was some rearrangement, refinement, and reclassification of data into different or new codes. For example, we aggregated, into "*Refactoring*", the preliminary categories "*move method*", "*refactoring operations*", and "*rename*" that were analyzed. We used the thematic analysis technique to address RQ_4.

Taxonomy Validation. In addition to the iterative process of building the taxonomy, we need to externally validate it from a practitioner's point of view [78,79]. The aim of this validation is to investigate whether it reflects actual MCR practices. To do so, we validated the taxonomy with a senior developer, with 8 years of industrial experience, and with 4 years of experience in code review. The survey contained 9 questions related to the correctness and representativeness of our taxonomy and proposed guidelines.

5. Results and discussion

5.1. How do refactoring reviews compare to non-refactoring reviews in terms of code review efforts?

Motivation. The first research question aims to explore whether reviewing refactoring in 'Refactor' branch takes longer compared to refactoring reviews containing the keyword 'refactor' and non-refactoring reviews from other branches. Understanding the differences in review efforts helps identify the unique challenges and requirements of refactoring reviews compared to non-refactoring ones.

Approach. To address RQ1, we intend to compare *refactoring reviews* with *non-refactoring reviews*, to see whether there are any differences

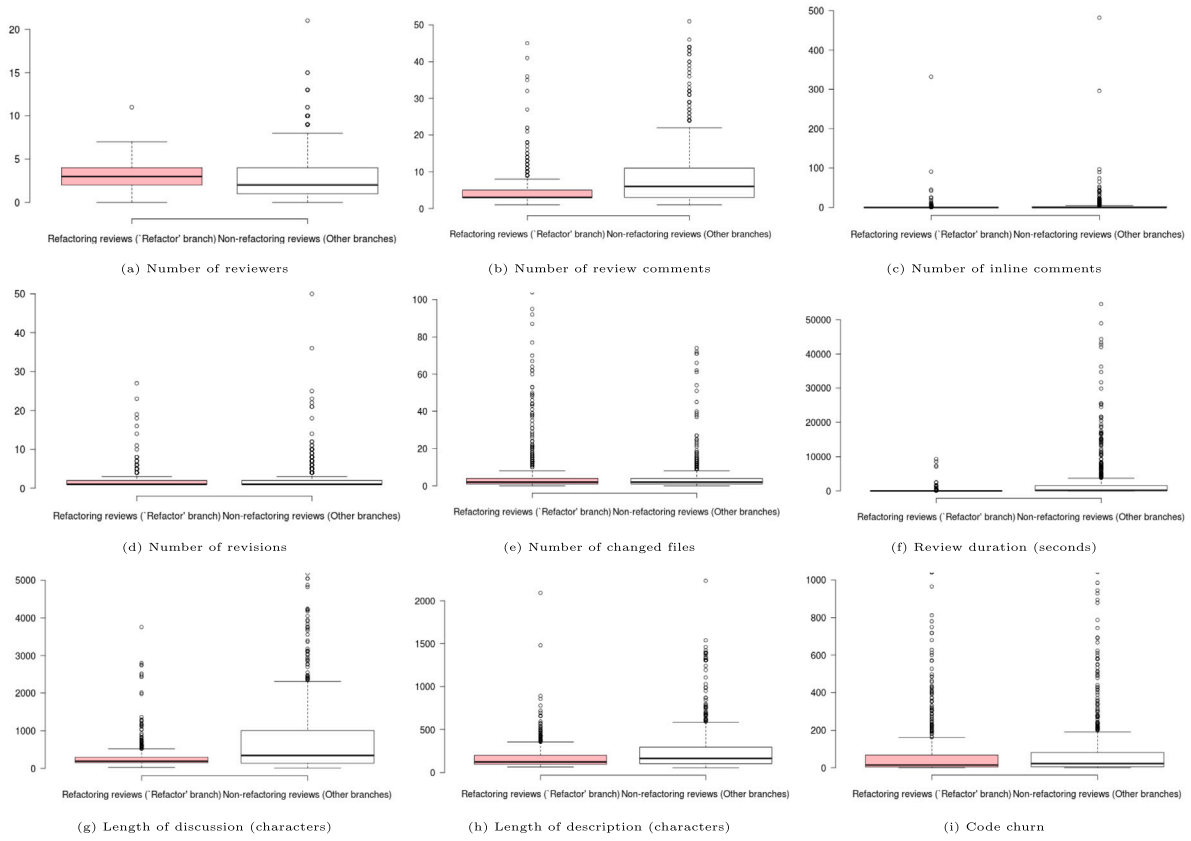


Fig. 4. Boxplots of metrics values of refactoring reviews ('Refactor') branch and non-refactoring reviews (Other branches).

in terms of code review efforts or metrics listed in Table 2, and Figs. 4, 5. Since our refactoring set in the 'Refactor' branch contains 718 reviews, we sampled 718 non-refactoring reviews and refactoring reviews containing the keyword 'refactor*' from the remaining ones in the review framework. This size provides a comprehensive view of the refactoring review practices within the 'Refactor' branch, capturing a diverse range of scenarios and developer interactions. To ensure the representativeness of the sample [80], we use stratified random sampling by choosing reviews from the rest of the reviews.

Results. By looking at the statistical summary in Table 2, and Figs. 4, 5, we found that reviewing refactoring changes in 'Refactor' branch differs, with fewer reviewers (μ = 2.93), fewer review comments (μ = 4.67), fewer inline comments (μ = 1.12), fewer revisions (μ = 1.87), shorter review time (μ = 90.58), and fewer discussions and descriptions (μ = 427.32, μ = 172.12, respectively) compared to reviewing non-refactoring changes. However, reviewing refactoring changes in the 'Refactor' branch shows more file changes (μ = 49.84), and more added and deleted lines between revisions (μ = 1366.49). As shown in Table 2, we performed a non-parametric Mann–Whitney U test and we obtained a statistically significant p-value when the values of these two groups were compared (p-value < 0.05 for all review efforts, except number of changed files), and accompanied with a small, medium, or large effect size depending on the review effort/metric.

Regarding the comparison between reviewing refactoring changes from the 'Refactor' branch and reviewing refactoring changes with the keyword 'refactor*' from other branches, we observed similar patterns as in the previous comparison. Reviewing refactoring changes in the 'Refactor' branch significantly differs, with fewer reviewers, fewer review comments, fewer inline comments, fewer revisions, shorter review time, and fewer discussions and descriptions compared to reviewing refactoring changes from other branches. However, in this comparison, there are more changed files and more added and deleted lines between revisions when reviewing refactoring changes from other branches. The only difference between this set and the previous comparison is that the difference in the number of changed files is significant.

We speculate that the observed differences in reviewing refactoring changes from the 'Refactor' branch and reviewing refactoring changes with the keyword 'refactor*' from other branches and non-refactoring reviews can be attributed to several factors:

Branch Isolation. Reviewing refactoring changes in the 'Refactor' branch took significantly less time compared to the non-refactoring changes. This indicates that the isolation of refactoring activities could streamline the review process, as reviewers can focus solely on refactoring tasks without being distracted by other types of changes. The statistically significant shorter review time (p-value < 0.05) in the 'Refactor' branch compared to other branches supports the idea that a dedicated branch for refactoring helps reduce the overall review effort.

- Example. In review ID 4058 (see Fig. 6), the refactoring change was reviewed in just 2.41 s with only 4 comments, highlighting how the isolated branch facilitated a quick and focused review.

Visibility. Fewer review comments and inline comments in the 'Refactor' branch suggest that reviewers quickly understand the changes and provide focused feedback. This lower number of comments, supported by a statistically significant p-value, implies that changes in the 'Refactor' branch might be receiving more targeted and effective attention from developers specializing in refactoring.

- Example. Review ID 3245 (see Fig. 6) had 0 inline comments and was approved by a single reviewer, showing how dedicated visibility in the 'Refactor' branch helps in quick and clear review processes.

Quality. The number of revisions needed and fewer discussions and descriptions in the 'Refactor' branch compared to other branches indicate a higher initial code quality. The lower number of revisions

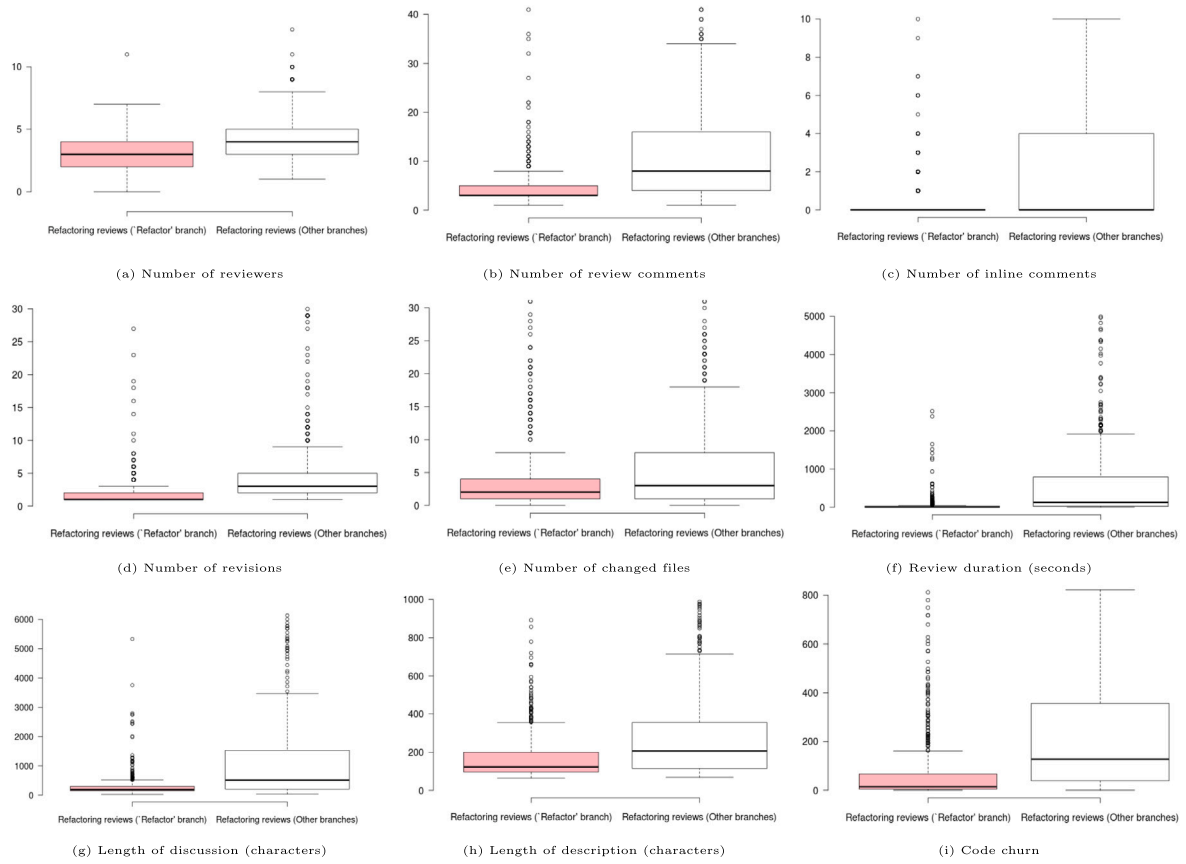


Fig. 5. Boxplots of metrics values of refactoring reviews ('Refactor') branch and refactoring reviews (Other branches).

and the fewer extensive discussions, with statistically significant differences, suggest that the refactoring changes are better prepared and understood, aligning with the higher quality hypothesis.

- Example. Review ID 1290 (see Fig. 6) required just one revision and had minimal back-and-forth discussion, suggesting that the initial refactoring proposal was of high quality.

Developer Expertise. A lower number of reviewers is required for refactoring changes in the 'Refactor' branch compared to non-refactoring changes. The significant difference in the number of reviewers, supported by p-value < 0.05, indicates that specialized knowledge in the 'Refactor' branch allows for more efficient and effective reviews, with fewer people needed to reach a consensus.

- Example. In review ID 1733 (see Fig. 6), two experienced developers quickly approved the change with minimal comments, demonstrating the impact of specialized expertise on the efficiency of the review process.

Collaboration Dynamics. Concise discussions and descriptions in the 'Refactor' branch suggest a more focused and collaborative environment. The statistically significant reduction in discussion length supports the notion that a collaborative environment in the 'Refactor' branch leads to more efficient communication and quicker resolution of review comments.

- Example. Review ID 123377 (see Fig. 6) had a concise discussion with 96 characters and was finalized quickly in 0.22 s, reflecting the collaborative and focused nature of the 'Refactor' branch.

Moreover, we conjecture that reviewing refactoring from other branches triggers longer discussions between the code change authors and the reviewers as we notice that several refactoring-related actions are being extensively discussed before reaching an agreement. While previous studies have found a similar pattern in GitHub's pull requests in open-source systems [47], Gerrit [20], and using code review tools in industry [13], there is no study that looked at the main reasons for refactoring-related discussions in 'Refactor' branch to take significantly less effort to be reviewed. Therefore, the findings of RQ1 have motivated us to manually analyze these reviews and extract the main criteria related to reviewing refactored code (RQ_4).

Further, we observe that refactoring-related code reviews from 'Refactor' branch impact larger code churn and more changes across files than non-refactoring code changes. These results are expected and agree with previous work [47,81,82], which found that refactored code has higher size-related metrics and larger changes promote refactorings. We also noticed that the number of developers who participated in the code review process is lower due to the high number of lines added, modified, or deleted between revisions. However, unlike a previous finding [47], no evidence of the correlation between the number of reviewers and refactoring was detected.

> ***Summary.*** *In the 'Refactor' branch, the review process is more focused due to the branch's dedicated purpose. This allows developers to concentrate more on refactoring activities. Further, reviews in the 'Refactor' branch tend to have shorter review times. This can be attributed to the specialized knowledge of the reviewers who are familiar with refactoring techniques, leading to quicker consensus and fewer iterations.*

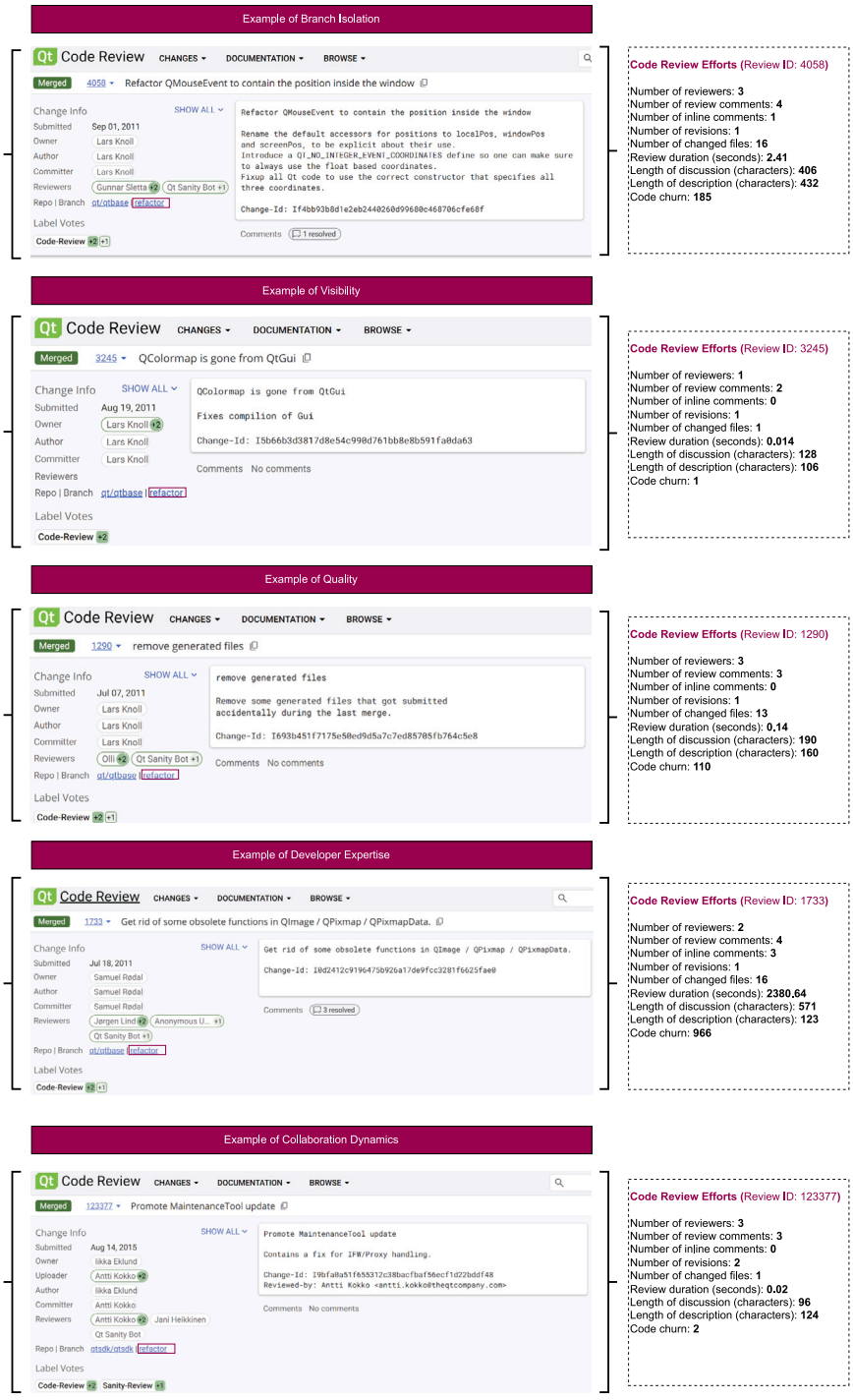


Fig. 6. Example of refactoring reviews from the 'Refactor' branch in the Qt project.

5.2. What textual patterns do developers use to describe their refactoring needs in the 'Refactor' branch?

Motivation. Since there is no consensus on how to formally document the act of refactoring code [65,66], research question two identifies what textual patterns developers have used to describe their refactoring activities in 'Refactor' branch. Identifying these patterns can reveal how developers communicate their intentions and needs, which is crucial to improving documentation and review processes.

Approach. We manually inspect Qt's subject, and description to identify refactoring documentation patterns in refactoring reviews from 'Refactor' branch and refactoring reviews from other branches containing the keyword 'refactor*' in the subject and description. These patterns are represented as a keyword or phrase that frequently occurs in refactoring reviews.

Results. Our in-depth inspection resulted in a list of 29 and 43 refactoring documentation patterns for refactoring reviews from 'Refactor' branch and other branches, respectively, as shown in Table 3. Our findings show that the names of refactoring operations (*e.g.*, 'mov*', 'renam*', 'extract*') occur in the top frequently occurring patterns, and these patterns are mainly linked to code elements at different levels of granularity such as classes, methods, and variables. These specific terms are well-known software refactoring operations and indicate

Table 3
List of refactoring documentation ('*' captures the extension of the keyword).

Patterns				
'Refactor' branch				
Chang* (890)	Fix* (340)	Add* (226)	Mov* (226)	Creat* (139)
Remov* (121)	Refactor* (107)	Merg* (74)	Renam* (33)	Dependenc* (28)
Replac* (24)	Get rid of (24)	Improv* (18)	Introduc* (18)	Cleanup* (13)
Modif* (6)	Split* (5)	Extend* (4)	Extract* (4)	Polish* (4)
Reduc* (4)	Remov* unused (4)	Inlin* (3)	Simplif* (3)	Encapsulat* (2)
Code clean* (2)	Organiz* (1)	Housekeeping (1)	Fix* regression (1)	
Other branches				
Refactor* (3669)	Chang* (2113)	Mov* (599)	Add* (519)	Fix* (427)
Remov* (342)	Creat* (253)	Simplif* (96)	Introduc* (93)	Renam* (89)
Replac* (71)	Split* (70)	Cleanup (68)	Improv* (64)	Extract* (50)
Modif* (48)	Merg* (40)	Reduc* (33)	Extend* (26)	Rewrit* (23)
Get rid of (22)	Remov* unused (17)	Dependenc* (14)	Inlin* (13)	Organiz* (12)
Encapsulat* (9)	Restructur* (8)	Remov* redundant (8)	Enhanc* (7)	Code clean* (6)
Reformat* (5)	Rework* (4)	Reorder* (3)	Modulariz* (2)	Polish* (2)
Reorganiz* (2)	Fix regression (3)	Cosmetic chang* (2)	Customiz* (2)	Re-writ* (1)
Less code (1)	Chang* the name (1)	Code clarity (1)		

developers' knowledge of the catalog of refactoring operations. We also observe that the top-ranked refactoring operation-related keywords include 'mov*', 'renam*', and 'extract*'. Moreover, we observe the occurrences of refactoring specific terms such as 'cleanup', 'get rid of', and 'remov* unused'.

RQ_2 indicates that developers tend to use limited textual patterns to document their refactorings in the 'Refactor' branch compared to refactoring reviews from other branches. These patterns can provide either (1) a generic description of problems developers encounter, or (2) a specific refactoring operation name following Fowler's names [83]. Although previous studies show that rename refactorings are a common type of refactoring, *e.g.*, [84], we notice that 'mov*' and 'extract*' are also among the topmost documented refactorings in 'Refactor' branch and other branches. This can be explained by the fact that developers tend to make many design improvement decisions, including remodularizing packages by moving classes, reducing class-level coupling, and increasing cohesion by moving methods. This information can provide valuable references for the practice of refactoring documentation. For example, whether refactoring-related reviews have relevant information is a critical indicator of the quality of refactoring reviews.

> ***Summary.*** *Within the 'Refactor' branch, developers tend to use fewer keywords in their documentation. This could be attributed to their specialized knowledge and expertise in refactoring techniques, which might result in more precise and less confusing documentation for reviewers.*

5.3. What quality attributes do developers consider when describing refactoring in the 'Refactor' branch?

Motivation. Various studies have explored the bad programming practices that trigger refactoring and the potential quality attributes that are optimized when restructuring the code. In this research question, we investigate whether developers explicitly mention the purpose of their refactoring activity, *e.g.,* improving structural metrics to fix code smells. Knowing the quality attributes prioritized by developers helps to understand their focus areas and can guide the creation of better refactoring guidelines and tools.

Approach. After identifying refactoring documentation patterns, we categorize the patterns into main categories (similar to previous studies [65,66]): (1) internal quality attributes, (2) external quality attributes, and (3) code smells.

Results. Table 4 provides the list of refactoring documentation patterns, ranked based on their frequency, which we identify in 'Refactor' branch and other branches. We observe that developers mention key internal quality attributes (such as 'inheritance', 'complexity', etc.), a few external quality attributes (such as 'compatability' and 'performance'), and code smells (such as 'dead code') that might impact code quality. To improve internal design, optimization of the structure of the system with respect to its dependency and inheritance appears to be the dominant focus that is consistently mentioned in the review. Concerning external quality attributes, we observe the mention of refactorings to enhance nonfunctional attributes. Patterns such as 'compatability', 'flexibility', and 'performance' represent the main focus in 'Refactor' branch, with 70%, 20%, and 10%, respectively. Finally, for code smells, developers mentioned a few antipatterns such as 'dead code'.

From RQ_3, we observe that developers in the 'Refactor' branch tend to provide less explicit documentation of their intent compared to other branches. This could suggest that the 'Refactor' branch prioritizes higher code quality, allowing reviewers to grasp the context and purpose of changes more easily even without detailed documentation. For instance, developer discussed fixing design issues by putting common functionalities into a superclass to eliminate duplicate code, breaking up lengthier methods to make the code more readable, and avoiding nested complex data structure to reduce code complexity. Moreover, we observe that code smell is rarely documented in 'Refactor' branch with only 3.22%. Similarly, developers tend to report few external quality attributes, focusing mainly on fixing *compatability* of the code.

> ***Summary.*** *Documentation of intent by developers is notably limited within the 'Refactor' branch when compared to other branches. However, this observation may signify higher code quality within the 'Refactor' branch, as changes are clearer and more understandable to reviewers without explicit documentation. This suggests that developers in the 'Refactor' branch prioritize key quality attributes such as compatibility, performance, complexity, and inheritance.*

5.4. What topics do developers discuss when reviewing refactoring tasks?

Motivation. We pose this research question to develop a taxonomy of all refactoring contexts, where reviewers discuss refactoring in the 'Refactor' branch. Analyzing discussion topics provides insights into the critical issues and considerations during refactoring reviews, highlighting areas for potential improvement.

Table 4
Summary of refactoring patterns in 'Refactor' branch, clustered by refactoring related categories.

Internal QA (%)	External QA (%)	Code Smell (%)
'Refactor' branch		
Dependency (90.32%) Inheritance (3.22%) Abstraction (3.22%) Complexity (3.22%)	Compatibility (70%) Flexibility (20%) Performance (10%)	Dead Code (3.22%)
Other branches		
Inheritance (30.43%) Dependency (30.43%) Coupling (19.56%) Abstraction (10.86%) Complexity (6.52%) Composition (2.17%)	Accessibility (35.63%) Performance (24.13%) Readability (18.39%) Compatibility(6.89%) Correctness (6.89%) Modularity (2.29%) Stability (1.14%) Usability (1.14%) Flexibility (1.14%) Robustness (1.14%) Testability (1.14%)	Code duplication (79.16%) Dead Code (10.41%) Switch Statement (4.16%) Antipattern (4.16%) Data Class (2.08%)

Approach. To get a more qualitative sense, we manually inspect the Qt ecosystem using a thematic analysis technique [76], to study the topics that reviewers discuss when reviewing refactoring changes, so we understand the main reasons for which refactoring reviews does not take shorter compared to non-refactoring reviews or refactoring reviews from other branches.

Results. Upon analyzing the review discussions, we create a comprehensive high-level categories of review criteria. Fig. 7 shows the proposed taxonomy of the criteria related to reviewing the refactored code. The taxonomy is composed of two layers: The top layer contains 4 categories that group activities with similar purposes, whereas the lower layer contains 12 subcategories that essentially provide a fine-grained categorization. These refactoring review criteria are centered on four main categories, as shown in the figure: (1) quality, (2) refactoring, (3) objective, and (4) testing. It is worth noting that our categorization is not mutually exclusive, meaning that a review can be associated with more than one category. An example of each category is provided in Table 5. In the rest of this subsection, we provide a more in-depth analysis of these categories.

Quality. The quality of design emerges as a crucial aspect of the refactoring review process. As per the submitted reviews, developers delve into optimizing *internal* and *external quality attributes* while striving to avoid *code smells*. They offer recommendations on coding practices and suggest ways to enhance both internal and external quality attributes. This attention to detail is essential because developers may not always grasp the full scope of the software design. Additionally, developers often focus their refactoring efforts on classes and methods that undergo frequent changes. This pattern is evident in the reviews, where recurrent files are frequently mentioned. By repeatedly modifying the same code elements, developers become more intimately acquainted with the system, thereby enhancing their design decisions.

Refactoring. This category gathers reviews with a focus on evaluating the correctness of the code transformation and checking whether or not the submitted changes lead to a safe and trustworthy refactoring. These refactoring reviews discuss refactoring operation-related responses such as *simplifying method calls*, *method composition*, and *features move*. As developers often interleave refactoring with other tasks, developers mentioned that combining refactoring with other changes could potentially result in overshadowing errors, thus increasing the likelihood of introducing bugs.

Objective. In this category, we have gathered cases where developers document the *feature-related*, *bug fix-related*, and *clean up-related* activities to better understand the rationale of the submitted code changes. This reveals how developers keep proposing areas of improvement, pertaining to the perception and the rationale of the change. It appears that the clarity of the documented changes is of paramount importance in accepting the submitted refactoring changes. We realized that the clarity of the explanation of what is being changed and why affects review time and decision.

Testing. Refactoring is intended to maintain the behavior of the software. Ideally, utilizing existing unit tests to confirm that the behavior remains unchanged should suffice. However, refactoring tasks may sometimes be interleaved with other activities, leading to potential alterations in the software's behavior. In such cases, existing unit tests may not capture these changes if they have not been revalidated to reflect the newly introduced functionality. Upon analyzing these discussions, reviewers have proposed several recommendations. They suggest incorporating unit tests before initiating the refactoring process to instill greater confidence that the code remains intact. Additionally, they recommend adding test cases when refactored code results in decreased test coverage, such as when extracting new methods. Furthermore, when developers submit their changes for review, including the results of running the tests can enhance transparency and provide assurance to reviewers. To encompass these scenarios, we have identified the following sub-categories under the *Testing* category: *additional test suite*, *auto-test fix*, and *test file refactoring*.

Our taxonomy builds on and extends existing refactoring taxonomies by focusing specifically on the unique characteristics of the 'Refactor' branch. Unlike general refactoring taxonomies that encompass a wide range of refactoring activities across various contexts, our taxonomy is designed to capture specific practices, challenges, and review dynamics within a dedicated refactoring branch. This taxonomy can be compared with existing ones in the literature, such as those proposed by Pantiuchina et al. [46,85] and Paixao et al. [43], which classify refactorings based on reasons for refactoring rejection and rationale in the GitHub pull requests and Gerrit, respectively. For example, previous studies found that lack of clear goals and poorly documented proposals are the main causes of rejection after code review, while design improvement and test quality are key motivations. Also, they found that non-explicit or mixed intent refactorings tend to have high interactivity during the review, observations we shared when creating the taxonomy in the 'Refactor' branch.

Unlike traditional taxonomies, ours is tailored to the unique context of the 'Refactor' branch, reflecting its specific practices and priorities. By focusing on the 'Refactor' branch, the taxonomy provides a more granular and context-specific understanding of refactoring practices. It offers insights into how branch isolation and focused collaboration can streamline the review process, reduce review time, and improve code quality. These findings are particularly valuable for organizations considering the adoption of dedicated refactoring branches as a strategy

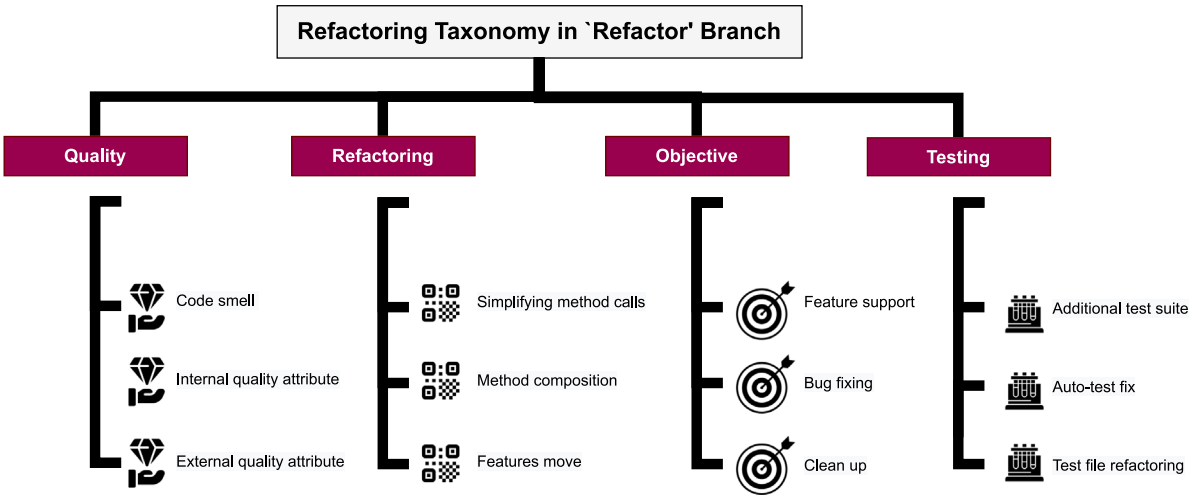


Fig. 7. Refactoring review criteria from 'Refactor' branch in modern code review.

to enhance their development workflows. The outcome of the survey with a senior developer shows the existence of these types of review criteria in the 'Refactor' branch.

Furthermore, we conducted a manual analysis to identify the factors that contribute to successful reviews in the refactoring branch compared to other branches. These insights were drawn from developer documentation and our observations of best practices. The following best practices were identified:

Focused Scope of Changes. Reviews in the 'Refactor' branch often involve changes that are narrowly focused on improving code structure without altering functionality. This clear separation may help reviewers quickly grasp the intent and impact of the changes. For example, in review ID 4058 (see Fig. 6), the developer focused solely on renaming variables to improve code readability. The review had only two inline comments and was approved within 2.41, demonstrating the efficiency of focused changes.

Clear and Concise Documentation. Reviews in the 'Refactor' branch often include clear, concise documentation explaining the purpose of the refactoring, the specific changes made, or the expected benefits. For instance, in review ID 1733 (see Fig. 6), the developer provided a clear explanation of the purpose of refactoring, and a succinct summary of the changes, which led to a smooth review process with minimal comments.

Consistency with Coding Standards. Reviews in the 'Refactor' branch often adhere to Gerrit Qt's established coding standards and guidelines, making it easier for reviewers to evaluate changes without debating style or format issues. For example, we noticed in review ID 4058 (see Fig. 6) that the developer used Qt specific naming convention when naming classes.

Summary. *Discussions within the 'Refactor' branch primarily revolve around topics related to code quality, objectives and testing. In particular, developers actively engage in conversations to ensure that the refactoring efforts adhere to the coding standards, discussions center on clarifying the objectives of the refactoring task, and there is a focus on rigorous testing practices to verify that the refactoring changes do not introduce new issues and that the improved code performs as expected.*

6. Implications

6.1. Implications for practitioners

Establishing continuous improvement culture for refactoring-related reviews. Our RQ_1 findings show that developers are more inclined to accept refactoring changes quickly in refactoring branch compared to the refactoring changes in other branches or to the non-refactoring changes. To emphasize the value of refactoring changes, managers can cultivate a culture by prioritizing continuous improvement and investing in refactoring. Therefore, establishing guidelines for submitting refactoring changes for review can be beneficial. For instance, refactoring branches should involve smaller, focused changes aimed at improving specific areas of the code. Moreover, these branches should emphasize the improvement of software quality, aligning with the shared goal among developers. It is essential for refactoring branches to clearly document the primary goals of the proposed changes. Additionally, thorough code reviews and testing should be conducted to ensure adherence to best practices and to prevent the introduction of regressions or breaking existing functionality. In summary, the recommended checklists are as follows:

Guidelines for submitting refactoring changes to the 'Refactor' branch

☐ **Nature of Changes:**

- ✓ Are the changes primarily focused on improving the structure, readability, or maintainability of the code without altering its external behavior?
- ✓ Do the changes involve renaming variables, methods, or classes to improve clarity?
- ✓ Are the changes aimed at reducing technical debt by simplifying complex code or removing redundancies?

☐ **Scope of Changes:**

- ✓ Do the changes impact multiple files or modules, indicating a broader structural improvement?
- ✓ Are the changes part of a planned refactoring effort as outlined in the project's roadmap or guidelines?

☐ **Documentation and Communication:**

- ✓ Have you documented the intent, instruction, impact, and scope of the refactoring changes in the commit message or accompanying documentation?

Table 5
A taxonomy of the refactoring review criteria from 'Refactor' branch in modern code review.

Category	Sub-category	Example (Excerpts from a related refactoring review)
Quality	Code smell	"*Qt Creator: remove dead patching code We are using a VCS. No need to keep dead code.*"
	Internal quality attribute	"*Checking in text control and editor classes what of those related attributes have exactly changed before calling the update increases code complexity unnecessarily for a little benefit.*"
	External quality attribute	"*This is simpler than the existing texture cache in QtOpenGL, as it only serves the GL paint engine. There is one per context group, to simplify the design and to prevent performance degradations.*"
Refactoring	Simplifying method calls	"*Refactor QMouseEvent to contain the position inside the window. Rename the default accessors for positions to localPos, windowPos and screenPos, to be explicit about their use.*"
	Method composition	"*Adding 10s delay after extracting 7z.*"
	Feature move	"*Add QtUiTools and uilib This code used to live in qtbase. It does not belong there however and since there are no dependencies in qtbase left that require it move it here. This significantly simplifies the build system with regards to the code in uilib.*"
Objective	Feature support	"*Restore feature compatibility with QPrinter in QTextDocument::print Add a margin method to QPagedPaintDevice.*"
	Bug fixing	"*fix foundImportantUpdate and rename it to essential – there was bug that unselected updates are removed in case there is an important update – to avoid the error with old installation we are renaming Important tag to Essential which means the same but only works with this fix.*"
	Clean up	"*Refactor the input framework Results of the ongoing workshop in Oslo: QInputPanel will be the application facing interface for controlling the input context as well as querying things like the position of the virtual keyboard. QInputContext is significantly cleaned up and only there as a compatibility API for existing code.*"
Testing	Additional test suite	"*Added new QOpenGLPaintDevice test case in tst_QOpenGL.*"
	Auto-test fix	"*Fixed auto-test failure in tst_QOpenGL. QOpenGLFramebufferObject::height() was returning the width.*"
	Test file refactoring	"*Move tests to more logical positions. Before, all the auto tests were jumbled together in a huge mess in tests/auto, now they are organized after which module/submodule/class they belong to. I have also started separating out unit tests from integration tests.*"

✓ Have you communicated with the team about the planned refactoring and received approval to proceed with the changes?

☐ **Testing and Validation:**

✓ Have you ensured that all existing tests pass after the refactoring changes?
✓ Have you added or updated tests to cover the refactored code, ensuring no functionality is broken?

☐ **Review Process:**

✓ Have you considered the feedback from previous refactoring reviews and incorporated best practices into your changes?

Guidelines for submitting changes to the main branch

☐ **Functional Changes:**

✓ Are the changes introducing new features, fixing bugs, or modifying the external behavior of the application?
✓ Do the changes involve implementing new functionality or altering existing functionality to meet new requirements?

☐ **Isolated Updates:**

✓ Are the changes isolated to a single file or a small set of related files?
✓ Do the changes address a specific issue or feature request without requiring broader structural modifications?

By following these checklists, developers can make informed decisions about where to submit their changes, ensuring that the refactoring branch is used appropriately for structural improvements while functional changes are directed to the main branch. We believe this approach can enhance the clarity and effectiveness of the development process, benefiting both individual developers and the overall project. It is worth noting that this checklist has been validated by an external developer for its relevance and completeness. However, we plan to apply this checklist in practice to further assess its applicability and impact in real-world scenarios.

Establishing guidelines for refactoring-related reviews. Our taxonomy shows that reviewing refactoring goes beyond improving the code structure. To improve the practice of reviewing refactored code, and contribute to the quality of reviewing code in general, managers can collaboratively work with developers to establish customized guidelines for reviewing refactoring changes which could establish beneficial and long-lasting habits or themes to accelerate the process of reviewing refactoring. Additionally, since our RQ_4 findings show that testing is one of the topics discussed by developers when reviewing refactoring changes, it is recommended to utilize continuous integration to keep the testing suite in sync with the code base during and after refactoring.

6.2. Implications for researchers

Understanding how refactoring changes in the 'Refactor' branch tend to be reviewed. From RQ_1, we observe that refactoring changes in the 'Refactor' branch are completed in a shorter timeframe compared to changes from other branches and non-refactoring changes. Researchers should further investigate the underlying reasons why refactoring changes in the 'Refactor' branch tend to be reviewed more efficiently compared to changes from other branches. Understanding the factors contributing to this efficiency, such as branch isolation, developer expertise, or collaboration dynamics, can provide valuable information on optimizing code review processes and enhancing software quality in development environments where refactoring is a common practice. Furthermore, exploring the impact of streamlined refactoring review practices on code quality, developer productivity, and overall project success could offer practical guidance for software development teams aiming to improve their refactoring workflows.

Supporting for the refactoring of non-source code artifacts. From RQ_4, we discover that refactoring operations are not limited to source code files. Artifacts such as databases and log files are also susceptible to refactoring. Similarly, we also observed discussions about refactoring test files. While it can be argued that test suites are source code files, recent studies [46,66] show that the types of refactoring operations applied to test files are frequently different from those applied to production files. Hence, future research on refactoring is encouraged to introduce refactoring mechanisms and techniques exclusively geared to refactoring non source code artifact types and test suites.

6.3. Implications for tool builders

Developing next generation refactoring-related code review tools. Finding that reviewing refactoring changes from other branches takes longer than non-refactoring changes reaffirms the necessity of developing accurate and efficient tools and techniques that can assist developers in the review process in the presence of refactorings. Refactoring toolset should be treated in the same way as the CI/CD tool set and integrated into the tool-chain. Researchers could use our findings with other empirical investigations of refactoring to define, validate, and develop a scheme to build automated assistance for reviewing refactoring considering the refactoring review criteria as review code becomes an easier process if the code review dashboard augmented with the factors to offer suggestions to better document the review.

Furthermore, to accelerate the code review process and limit having a back-and-forth discussion for clarity on the problem faced by the developer, tool builders can develop *bots* for the integration, testing, and management categories. Additionally, it would be interesting to use a popular and widely adopted quality framework, *e.g.,* Quality Gate of SonarQube [86], as part of quality verification process by embedding its results in the code review. This might facilitate convincing the reviewer about the impact and the correctness of the performed refactoring.

7. Threats to validity

In this section, we describe potential threats to validity of our research method, and the actions we took to mitigate them.

Internal Validity. Concerning the identification of refactoring related code review, we select reviews from a dedicated 'Refactor' branch. As for the other group of refactoring reviews, we analyze reviews with the keyword '*refactor**' in their title and description. Such selection criteria may have resulted in missing refactoring-related reviews, and there is the possibility that we may have excluded synonymous terms/phrases. However, even though this approach reduces the number of reviews in our dataset, it also decreases the false positiveness of our selection. While our data collection may result in missing some reviews, our approach ensures that we analyze reviews that are explicitly geared toward refactoring. In other words, these are reviews where developers were explicitly documenting a refactoring action and they wanted it to be reviewed. Additionally, after performing the manual inspection on review discussions, we realized that refactoring is heavily emphasized in discussions that start with a title or a description containing the keyword '*refactor**'. Yet, this does not prevent other discussions from bringing refactoring into the picture, and these will be missed by our selection (*i.e.,* false negatives). Hence, we excluded potential refactoring synonymous terms/phrases when selecting non-refactoring reviews. We opted for such picky selection to only consider discussions when code authors explicitly wanted their refactored code to be reviewed, and so reviewers eventually propose a refactoring-aware feedback, which is what we are aiming for in this study. Therefore, it is interesting to consider scenarios where reviewers have raised concerns about refactoring a code change that was not intended to be associated with the 'Refactor' branch. Since refactoring can easily be interleaved with other functional changes, it would be interesting to extract scenarios where reviewers thought it was misused. The study can also help developers better understand not only how to refactor their code, but also how to document it properly for easier review.

Furthermore, we focus on the code review activity that is reported by the tool-based code review process, *i.e.,* Gerrit, of the systems studied due to the fact that other communication media (*e.g.,* in-person discussion [22], a group IRC [87], or a mailing list [88]) do not have explicit links to code changes and recovering these links is a daunting task [55,89].

Construct Validity. About the representativeness and the correctness of our refactoring review criteria, we derive these criteria from a manual analysis of refactoring-related reviews. This approach may not cover the whole spectrum of all the review criteria done with refactoring in mind. Additionally, to avoid personal bias during the manual analysis, each step in the manual analysis was conducted twice, and the results were always cross-validated. Another potential threat to validity relates to refactoring reviews. Since refactorings could interleave with other changes [90] (*i.e.,* developers performed changes together with refactorings), we cannot claim that the selected refactoring reviews are exclusively related to refactoring. However, during our qualitative analysis, we identified this activity as one of the challenges that contributed to slowing down the review process from other branches.

External Validity. We focus our study on one open-source system, due to the low number of systems that satisfied our eligibility criteria (see Section 4). Therefore, our results may not be generalized to all other open-source systems or to commercially developed projects. However, the goal of this paper is not to build a theory that applies to all systems, but rather to show that refactoring can have an impact on code review process. Another potential threat relates to the proposed taxonomy. Our taxonomy may not generalize to other open source or commercial projects since the refactoring review criteria may be different for another set of projects (*e.g.,* outside the Qt community). Consequently, we cannot claim that the results of refactoring review criteria (see Fig. 7) can be generalized to other software systems where the need for improving the design might be less important.

Conclusion Validity. To compare the two groups of code review requests, we used appropriate statistical procedures with *p*-value and effect size measures to test the significance of the differences and their magnitude. A statistical test was implemented to measure the significance of the observed differences between group values. This test makes no assumption that the data is normally distributed. Also, it assumes the independence of the groups under comparison. We cannot verify whether code review requests are completely independent, as some can be re-opened, or one large code change can be treated using several requests. To mitigate this, we verified all the reviews we sampled for the test.

8. Conclusion

Understanding the practice of refactoring code review holds significant importance for both the research community and industry. Despite the widespread adoption of modern code review practices in open-source and industrial projects, the correlation between code review and refactoring practices in the 'Refactor' branch remains largely unexplored. In our study, we conducted a comprehensive quantitative and qualitative analysis to investigate the review criteria discussed by the developers during the review of refactorings. Our findings highlight several key points: refactoring changes in the 'Refactor' branch are completed in a shorter timeframe compared to changes from other branches and non-refactoring changes; documentation of developer intent within the 'Refactor' branch is limited in comparison to other branches; and developers rely on a specific set of criteria to guide their decisions regarding the acceptance or rejection of submitted refactoring changes.

For future work, we plan on conducting a structured survey with software developers from both open-source and industry. The survey will explore their general and specific review criteria when performing refactoring activities in code review. This survey will complement and validate our current study to provide the software engineering community with a more comprehensive view of refactoring practices in the context of modern code review. Another interesting research direction is to link refactoring-related reviews to refactoring detection tools such as Refactoring Miner [17] or RefDiff [91] to better understand the impact of these reviews on refactoring types specifically.

CRediT authorship contribution statement

Eman Abdullah AlOmar: Writing – review & editing, Writing – original draft, Visualization, Validation, Project administration, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The dataset is publically available: https://smilevo.github.io/self-affirmed-refactoring/.

References

[1] C. Abid, V. Alizadeh, M. Kessentini, T. do Nascimento Ferreira, D. Dig, 30 years of software refactoring research:A systematic literature review, 2020, arXiv:2007.02194.

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

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

[4] Y. Kashiwa, R. Nishikawa, Y. Kamei, M. Kondo, E. Shihab, R. Sato, N. Ubayashi, An empirical study on self-admitted technical debt in modern code review, Inf. Softw. Technol. 146 (2022) 106855.

[5] S. McIntosh, Y. Kamei, B. Adams, A.E. Hassan, An empirical study of the impact of modern code review practices on software quality, Empir. Softw. Eng. 21 (5) (2016) 2146–2189.

[6] X. Yang, R.G. Kula, N. Yoshida, H. Iida, Mining the modern code review repositories: A dataset of people, process and product, in: Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 460–463.

[7] O. Hamdi, A. Ouni, E.A. AlOmar, M.O. Cinnéide, M.W. Mkaouer, An empirical study on the impact of refactoring on quality metrics in android applications, in: IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), 2021, pp. 28–39.

[8] E.A. AlOmar, M.W. Mkaouer, A. Ouni, M. Kessentini, On the impact of refactoring on the relationship between quality attributes and design metrics, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, IEEE, 2019, pp. 1–11.

[9] O. Hamdi, A. Ouni, M.Ó. Cinnéide, M.W. Mkaouer, A longitudinal study of the impact of refactoring in android applications, Inf. Softw. Technol. 140 (2021) 106699.

[10] A. Peruma, C.D. Newman, M.W. Mkaouer, A. Ouni, F. Palomba, An exploratory study on the refactoring of unit test files in android applications, in: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, 2020, pp. 350–357.

[11] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, O. Strollo, When does a refactoring induce bugs? an empirical study, in: IEEE 12th International Working Conference on Source Code Analysis and Manipulation, 2012, pp. 104–113.

[12] M. Di Penta, G. Bavota, F. Zampetti, On the relationship between refactoring actions and bugs: A differentiated replication, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 556–567.

[13] E.A. AlOmar, H. AlRubaye, M.W. Mkaouer, A. Ouni, M. Kessentini, Refactoring practices in the context of modern code review: An industrial case study at xerox, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP, IEEE, 2021, pp. 348–357.

[14] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of type-checking bad smells, in: 2008 12th European Conference on Software Maintenance and Reengineering, IEEE, 2008, pp. 329–331.

[15] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using NSGA-III, ACM Trans. Softw. Eng. Methodol. (TOSEM) 24 (3) (2015) 1–45.

[16] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: An industrial case study, ACM Trans. Softw. Eng. Methodol. (TOSEM) 25 (3) (2016) 23.

[17] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: 2018 IEEE/ACM 40th International Conference on Software Engineering, ICSE, IEEE, 2018, pp. 483–494.

[18] https://codereview.qt-project.org/c/qt/qtbase/+/4058.

[19] https://codereview.qt-project.org/c/qt/qtbase/+/215219.

[20] E.A. AlOmar, M. Chouchen, M.W. Mkaouer, A. Ouni, Code review practices for refactoring changes: An empirical study on openstack, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 689–701.

[21] https://smilevo.github.io/self-affirmed-refactoring/.

[22] M. Beller, A. Bacchelli, A. Zaidman, E. Juergens, Modern code reviews in open-source projects: Which problems do they fix? in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 202–211.

[23] L.G. Votta Jr., Does every inspection need a meeting? in: Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, 1993, pp. 107–114.

[24] P.C. Rigby, C. Bird, Convergent contemporary software peer review practices, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 202–212.

[25] S. McIntosh, Y. Kamei, B. Adams, A.E. Hassan, The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects, in: Working Conference on Mining Software Repositories, 2014, pp. 192–201.

[26] P. Thongtanunam, X. Yang, N. Yoshida, R.G. Kula, A.E.C. Cruz, K. Fujiwara, H. Iida, Reda: A web-based visualization tool for analyzing modern code review dataset, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 605–608.

[27] R.G. Kula, A.E.C. Cruz, N. Yoshida, K. Hamasaki, K. Fujiwara, X. Yang, H. Iida, Using profiling metrics to categorise peer review types in the android project, in: 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops, IEEE, 2012, pp. 146–151.

[28] X. Ge, S. Sarkar, E. Murphy-Hill, Towards refactoring-aware code review, in: International Workshop on Cooperative and Human Aspects of Software Engineering, 2014, pp. 99–102.

[29] X. Ge, S. Sarkar, J. Witschey, E. Murphy-Hill, Refactoring-aware code review, in: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, 2017, pp. 71–79.

[30] R. Morales, S. McIntosh, F. Khomh, Do code review practices impact design quality? a case study of the qt, vtk, and itk projects, in: International Conference on Software Analysis, Evolution, and Reengineering, SANER, 2015, pp. 171–180.

[31] M. Barnett, C. Bird, J. Brunet, S.K. Lahiri, Helping developers help themselves: Automatic decomposition of code review changesets, in: International Conference on Software Engineering, vol. 1, 2015, pp. 134–144.

[32] Y. Tao, S. Kim, Partitioning composite code changes to facilitate code review, in: Working Conference on Mining Software Repositories, 2015, pp. 180–190.

[33] T. Zhang, M. Song, J. Pinedo, M. Kim, Interactive code review for systematic changes, in: International Conference on Software Engineering, vol. 1, 2015, pp. 111–122.

[34] E.L. Alves, M. Song, T. Massoni, P.D. Machado, M. Kim, Refactoring inspection support for manual refactoring edits, IEEE Trans. Softw. Eng. 44 (4) (2017) 365–383.

[35] A. Peruma, S. Simmons, E.A. AlOmar, C.D. Newman, M.W. Mkaouer, A. Ouni, How do i refactor this? An empirical study on refactoring trends and topics in stack overflow, Empir. Softw. Eng. 27 (1) (2022) 1–43.

[36] Y. Tao, Y. Dang, T. Xie, D. Zhang, S. Kim, How do software engineers understand code changes?: an exploratory study in industry, in: ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, p. 51.

[37] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, J. Czerwonka, Code reviewing in the trenches: Challenges and best practices, IEEE Softw. 35 (4) (2017) 34–42.

[38] B. Guo, M. Song, Interactively decomposing composite changes to support code review and regression testing, in: Annual Computer Software and Applications Conference, COMPSAC, vol. 1, 2017, pp. 118–127.

[39] A. Peruma, M.W. Mkaouer, M.J. Decker, C.D. Newman, Contextualizing rename decisions using refactorings and commit messages, in: 2019 19th International Working Conference on Source Code Analysis and Manipulation, SCAM, IEEE, 2019, pp. 74–85.

[40] E.L. Alves, M. Song, M. Kim, RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 751–754.

[41] F. Coelho, T. Massoni, E.L. Alves, Refactoring-aware code review: A systematic mapping study, in: International Workshop on Refactoring, 2019, pp. 63–66.

[42] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, A. Bacchelli, Information needs in contemporary code review, Proc. ACM Hum.-Comput. Interact. 2 (CSCW) (2018) 135.

[43] M. Paixão, A. Uchôa, A.C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, E. Arvonio, Behind the intents: An in-depth empirical study on software refactoring in modern code review, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 125–136.

[44] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenílio, R. Lima, A. Garcia, C. Bezerra, How does modern code review impact software design degradation? an in-depth empirical study, in: 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2020, pp. 511–522.

[45] A. Uchôa, C. Barbosa, D. Coutinho, W. Oizumi, W.K. Assunçao, S.R. Vergilio, J.A. Pereira, A. Oliveira, A. Garcia, Predicting design impactful changes in modern code review: A large-scale empirical study, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR, IEEE, 2021, pp. 471–482.

[46] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, M.D. Penta, Why developers refactor source code: A mining-based study, ACM Trans. Softw. Eng. Methodol. (TOSEM) 29 (4) (2020) 1–30.

[47] F. Coelho, N. Tsantalis, T. Massoni, E.L. Alves, An empirical study on refactoring-inducing pull requests, in: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, 2021, pp. 1–12.

[48] A. Brito, A. Hora, M.T. Valente, Refactoring graphs: Assessing refactoring over time, 2020, arXiv preprint arXiv:2003.04666.

[49] D. Silva, J. Silva, G.J.D.S. Santos, R. Terra, M.T.O. Valente, RefDiff 2.0: A multi-language refactoring detection tool, IEEE Trans. Softw. Eng. (2020).

[50] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, T. Bryksin, RefactorInsight: Enhancing IDE representation of changes in git with refactorings information, 2021, arXiv preprint arXiv:2108.11202.

[51] 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, IEEE, 2015, pp. 141–150.

[52] X. Zhang, Y. Chen, Y. Gu, W. Zou, X. Xie, X. Jia, J. Xuan, How do multiple pull requests change the same code: A study of competing pull requests in github, in: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2018, pp. 228–239.

[53] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (2009) 131–164.

[54] P. Thongtanunam, S. McIntosh, A.E. Hassan, H. Iida, Investigating code review practices in defective files: An empirical study of the qt system, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 168–179.

[55] P. Thongtanunam, S. McIntosh, A.E. Hassan, H. Iida, Revisiting code ownership and its relationship with software quality in the scope of modern code review, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 1039–1050.

[56] P. Thongtanunam, A.E. Hassan, Review dynamics and their impact on software quality, IEEE Trans. Softw. Eng. (2020).

[57] A. Ouni, R.G. Kula, K. Inoue, Search-based peer reviewers recommendation in modern code review, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2016, pp. 367–377.

[58] Y. Fan, X. Xia, D. Lo, S. Li, Early prediction of merged code changes to prioritize reviewing tasks, Empir. Softw. Eng. 23 (6) (2018) 3346–3393.

[59] M. Chouchen, A. Ouni, M.W. Mkaouer, R.G. Kula, K. Inoue, WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review, Appl. Soft Comput. 100 (2021) 106908.

[60] S. Kim, E.J. Whitehead, Y. Zhang, Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34 (2) (2008) 181–196.

[61] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Softw. Eng. 39 (6) (2012) 757–773.

[62] A. Mockus, L.G. Votta, Identifying reasons for software changes using historic databases, in: Icsm, 2000, pp. 120–130.

[63] A.E. Hassan, Automated classification of change messages in open source projects, in: Proceedings of the 2008 ACM Symposium on Applied Computing, 2008, pp. 837–841.

[64] J. Ratzinger, T. Sigmund, H.C. Gall, On the relation of refactorings and software defect prediction, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, ACM, New York, NY, USA, 2008, pp. 35–38, http://dx.doi.org/10.1145/1370750.1370759, URL http://doi.acm.org/10.1145/1370750.1370759.

[65] E.A. AlOmar, M.W. Mkaouer, A. Ouni, Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages, in: International Workshop on Refactoring, IEEE, 2019, in press.

[66] E.A. AlOmar, A. Peruma, M.W. Mkaouer, C. Newman, A. Ouni, M. Kessentini, How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation, Expert Syst. Appl. 167 (2021) 114176.

[67] K. Stroggylos, D. Spinellis, Refactoring–does it improve software quality? in: Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), IEEE, 2007, 10–10.

[68] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, A. Raja, An empirical study of refactorings and technical debt in machine learning systems, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, IEEE, 2021, pp. 238–250.

[69] E.A. AlOmar, J. Liu, K. Addo, M.W. Mkaouer, C. Newman, A. Ouni, Z. Yu, On the documentation of refactoring types, Autom. Softw. Eng. 29 (1) (2022) 1–40.

[70] D. Zhang, B. Li, Z. Li, P. Liang, A preliminary investigation of self-admitted refactorings in open source software, 2018, http://dx.doi.org/10.18293/SEKE2018-081.

[71] D. Taeger, S. Kuhnt, Statistical Hypothesis Testing with SAS and R, John Wiley & Sons, 2014.

[72] W.J. Conover, Practical Nonparametric Statistics, vol. 350, John Wiley & Sons, 1998.

[73] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions, Psychol. Bull. 114 (3) (1993) 494.

[74] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data, in: Proceedings of the Annual Meeting of the Florida Association of Institutional Research, 2006, pp. 1–3.

[75] C. Wissler, The spearman correlation formula, Science 22 (558) (1905) 309–311.

[76] D.S. Cruzes, T. Dyba, Recommended steps for thematic synthesis in software engineering, in: 2011 International Symposium on Empirical Software Engineering and Measurement, IEEE, 2011, pp. 275–284.

[77] D. Silva, N. Tsantalis, M.T. Valente, Why we refactor? Confessions of GitHub contributors, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: FSE 2016, ACM, New York, NY, USA, 2016, pp. 858–870, http://dx.doi.org/10.1145/2950290.2950305, URL http://doi.acm.org/10.1145/2950290.2950305.

[78] L. Pascarella, F.-X. Geiger, F. Palomba, D. Di Nucci, I. Malavolta, A. Bacchelli, Self-reported activities of android developers, in: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft, IEEE, 2018, pp. 144–155.

[79] E. Doğan, E. Tüzün, Towards a taxonomy of code review smells, Inf. Softw. Technol. 142 (2022) 106737.

[80] K.L. Clarkson, P.W. Shor, Applications of random sampling in computational geometry, II, Discrete Comput. Geom. 4 (5) (1989) 387–421.

[81] P. Hegedűs, I. Kádár, R. Ferenc, T. Gyimóthy, Empirical evaluation of software maintainability based on a manually validated refactoring dataset, Inf. Softw. Technol. 95 (2018) 313–327.

[82] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, M. Harman, The impact of code review on architectural changes, IEEE Trans. Softw. Eng. (2019).

[83] M. Fowler, K. Beck, J. Brant, W. Opdyke, d. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999, URL http://dl.acm.org/citation.cfm?id=311424.

[84] E.A. AlOmar, A. Peruma, M.W. Mkaouer, C.D. Newman, A. Ouni, An exploratory study on refactoring documentation in issues handling, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 107–111.

[85] J. Pantiuchina, B. Lin, F. Zampetti, M. Di Penta, M. Lanza, G. Bavota, Why do developers reject refactorings in open-source projects? ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (2) (2021) 1–23.

[86] O. Gaudin, Continuous inspection a paradigm shift in software quality management, third ed., in: 10, vol. 4, SonarSource, 2013.

[87] E. Shihab, Z.M. Jiang, A.E. Hassan, Studying the use of developer IRC meetings in open source projects, in: 2009 IEEE International Conference on Software Maintenance, IEEE, 2009, pp. 147–156.

[88] P.C. Rigby, M.-A. Storey, Understanding broadcast based peer review on open source software projects, in: 2011 33rd International Conference on Software Engineering, ICSE, IEEE, 2011, pp. 541–550.

[89] A. Bacchelli, M. Lanza, R. Robbes, Linking e-mails and source code artifacts, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, 2010, pp. 375–384.

[90] E. Murphy-Hill, C. Parnin, A.P. Black, How we refactor, and how we know it, IEEE Trans. Softw. Eng. 38 (1) (2012) 5–18, http://dx.doi.org/10.1109/TSE.2011.41.

[91] D. Silva, M.T. Valente, Refdiff: detecting refactorings in version histories, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR, IEEE, 2017, pp. 269–279.