

Spring 5-1-2021

Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox

Eman Abdullah AlOmar
Rochester Institute of Technology

Hussein Alrubaye
Xerox Corporation

Mohamed Wiem Mkaouer
Rochester Institute of Technology

Ali Ouni
University of Quebec at Montreal

Marouane Kessentini
University of Michigan - Dearborn

Follow this and additional works at: <https://scholarworks.rit.edu/article>



Part of the [Software Engineering Commons](#)

Recommended Citation

<https://doi.org/10.1109/ICSE-SEIP52600.2021.00044>

This Conference Paper is brought to you for free and open access by the Faculty & Staff Scholarship at RIT Scholar Works. It has been accepted for inclusion in Articles by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox

Eman Abdullah AlOmar*, Hussein AlRubaye†, Mohamed Wiem Mkaouer*, Ali Ouni‡, Marouane Kessentini§

*Rochester Institute of Technology, Rochester, NY, USA

†Xerox Corporation, Rochester, NY, USA

‡ETS Montreal, University of Quebec, Montreal, QC, Canada

§University of Michigan, Dearborn, MI, USA

eman.alomar@mail.rit.edu, hussein.alrubaye@xerox.com, mwmvse@rit.edu, ali.ouni@etsmtl.ca, marouane@umich.edu

Abstract—Modern code review is a common and essential practice employed in both industrial and open-source projects to improve software quality, share knowledge, and ensure conformance with coding standards. During code review, developers may inspect and discuss various changes including refactoring activities before merging code changes in the code base. To date, code review has been extensively studied to explore its general challenges, best practices and outcomes, and socio-technical aspects. However, little is known about how refactoring activities are being reviewed, perceived, and practiced.

This study aims to reveal insights into how reviewers develop a decision about accepting or rejecting a submitted refactoring request, and what makes such review challenging. We present an industrial case study with 24 professional developers at Xerox. Particularly, we study the motivations, documentation practices, challenges, verification, and implications of refactoring activities during code review.

Our study delivers several important findings. Our results report the lack of a proper procedure to follow by developers when documenting their refactorings for review. Our survey with reviewers has also revealed several difficulties related to understanding the refactoring intent and implications on the functional and non-functional aspects of the software. In light of our findings, we recommended a procedure to properly document refactoring activities, as part of our survey feedback.

Index Terms—Refactoring, Code Review, Software Quality

I. INTRODUCTION

The role of refactoring has been growing in practice beyond simply improving the internal structure of the code without altering its external behavior [1] to become a widespread concept for the agile methodologies, and a *de-facto* practice to reduce technical debt [2]. In parallel, contemporary software projects adopt code review, a well-established practice for maintaining software quality and sharing knowledge about the project [3], [4]. Code review is the process of manually inspecting new code changes to verify their adherence to standards and its freedom from faults [3]. Modern code review has emerged as a lightweight, asynchronous, and tool-based process with reliance on a documentation of the inspection process, in the form of a discussion between the code change author and the reviewer(s) [5].

Refactoring, just like any code change, has to be reviewed, before being merged into the code base. However, little is known about how developers *perceive* and *practice* refactoring during the code review process, especially that refactoring, by

definition, is not intended to alter to the system’s behavior, but to improve its structure, so its review may differ from other code changes. Yet, there is not much research investigating how developers review code refactoring. The research on refactoring has been focused on its automation by identifying refactoring opportunities in the source code, and recommending the adequate refactoring operations to perform [6]–[8]. Moreover, the research on code reviews has been focused on automating it by recommending the most appropriate reviewer for a given code change [3]. However, despite the critical role of refactoring and code review, the innate relationship between them is still largely unexplored in practice.

The goal of this paper is to understand how developers review code refactoring, *i.e.*, what criteria developers rely on to develop a decision about accepting or rejecting a submitted refactoring change, and what makes this process challenging. This paper seeks to gain practical insights from the existing relationship between refactoring and code review through the investigation of five main research questions:

RQ1. *What motivates developers to apply refactorings in the context of modern code review?*

RQ2. *How do developers document their refactorings for code review?*

RQ3. *What challenges do reviewers face when reviewing refactoring changes?*

RQ4. *What mechanisms are used by developers and reviewers to ensure the correctness after refactoring?*

RQ5. *How do developers and reviewers assess and perceive the impact of refactoring on the source code quality?*

To address these research questions, we surveyed 24 professional software developers, from the research and development team, at Xerox. Our survey questions were designed to gather the necessary information that can answer the above-mentioned research questions and insights into the review practices of refactoring activities in an industrial setting. Moreover, we perform a pilot study by comparing between code reviews related to refactoring, and the remaining code reviews, in terms of time to resolution and number of exchanged responses. Our findings indicate that refactoring-specific code reviews take longer to be resolved and typically

86 triggers more discussions between developers and reviewers
87 to reach a consensus. The survey with reviewers, has revealed
88 many challenges they are facing when they review refactored
89 code. We report them as part of our survey results, and we
90 provide some guidelines for developers to follow in order to
91 facilitate the review of their refactorings.

92 II. RELATED WORK

93 A. Surveys & Case Studies on Refactoring

94 Prior works have conducted literature surveys on refactoring
95 from different aspects. The focus of these surveys ranges
96 between investigating the impact of refactoring on software
97 quality [13], to comparing refactoring tools [9], and exploring
98 refactoring challenges and practices [10]–[12], [14], [15].
99 These studies are depicted in Table I.

100 Murphy-Hill & Black [9] surveyed 112 Agile Open North-
101 west conference attendees and found that refactoring tools are
102 underused by professional programmers. In an explanatory
103 survey involving 33 developers, Arcoverde et al. [10] studied
104 how developers react to the presence of design defects in
105 the code. Their primary finding indicates that design defects
106 tend to live longer due to the fact that developers avoid
107 performing refactoring to prevent unexpected consequences.
108 Yamashita & Moonen [11] performed an empirical study in
109 commercial software to evaluate the severity of code smells
110 and the usefulness of code smell-related tooling. The authors
111 found that 32% of the interviewed developers are unaware
112 of code smells, and refactoring tools should provide better
113 support for refactoring suggestions. Kim et al. [12] surveyed
114 328 professional software engineers at Microsoft to investigate
115 when and how they do refactoring. When surveyed, the de-
116 velopers cited the main benefits of refactoring to be: improved
117 readability (43%), improved maintainability (30%), improved
118 extensibility (27%) and fewer bugs (27%). When asked what
119 provokes them to refactor, the main reason provided was poor
120 readability (22%). Only one code smell, *i.e.*, code duplication,
121 was reported (13%). Szoke et al. [13] conducted 5 large-scale
122 industrial case studies on the application of refactoring while
123 fixing coding issues; they have shown that developers tend
124 to apply refactorings manually at the expense of a large time
125 overhead. Sharma et al. [14] surveyed 39 software architects
126 asking about the problems they faced during refactoring tasks
127 and the limitations of existing refactoring tools. Their main
128 findings are: (1) fear of breaking code restricts developers
129 to adopt refactoring techniques; and (2) refactoring tools
130 need to provide better support for refactoring suggestions.
131 Newman et al. [15] conducted a survey of 50 developers
132 to understand their familiarity with transformation languages
133 for refactoring. They found that there is a need to increase
134 developer confidence in refactoring and transformation tools.

135 B. Refactoring Awareness & Code Review

136 Research on modern code review topics has been of import-
137 ance to practitioners and researchers. A considerable effort is
138 spent by the research community in studying traditional and
139 modern code review practices and challenges. This literature

has been includes case studies (*e.g.*, [4], [16]), user studies
(*e.g.*, [17]), and surveys (*e.g.*, [3], [18]). However, most of the
above studies focus on studying 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 [3]
observed, and surveyed developers to understand the chal-
lenges 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. [18] interviewed several
teams at Microsoft and conducted a survey to investigate the
human and social factors that influence developers' experi-
ences 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. [16] developed a refactoring-aware code review tool,
called ReviewFactor, that automatically detects refactoring
edits and separates refactoring from non-refactoring changes
with the focus on five refactoring types. The tool was inten-
ded 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 [16], Alves et al. [17]
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.

To summarize, existing studies mainly focus on proposing
and evaluating refactoring tools that can be useful to support
modern code review, but the perception of refactoring in
code review remains largely unexplored. To the best of our
knowledge, no prior studies have conducted case studies in
an industrial setting to explore 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.
Previous studies, however, discussed code review motivations
and challenges in general [3], [4], [18]. To gain more in-depth
understanding of the above-mentioned five dimensions, in this
paper, we surveyed several developers at Xerox.

189 III. STUDY DESIGN

190 A. Research Questions

**RQ1. What motivates developers to apply refactorings
in the context of modern code review?** Several motivations
behind refactoring have been reported in the literature [1],

Table (I) Related work in industrial case study & survey on refactoring.

Study	Year	Research Method	Focus	Single/Multi Company	Subject/Object Selection Criteria	# Participants
Murphy-Hill & Black [9]	2008	Survey	Refactoring tools	Yes/No	programmers	112
Arcoverde et al. [10]	2011	Survey	Longevity of code smells	No/Yes	belongs to development team	33
Yamashita & Moonen [11]	2013	Survey	Developer perception of code smells	No/Yes	developers	85
Kim et al. [12]	2014	Survey & Interview	Refactoring challenges & benefits	Yes/No	has change messages including "refactor" within last 2 years	328
Szoke et al. [13]	2014	Case Study & Survey	Impact of refactoring on quality	No/Yes	developers	40
Sharma et al. [14]	2015	Survey	Challenges & solutions for refactoring adoption	Yes/No	architects	39
Newman et al. [15]	2018	Survey	Developer familiarity of transformation languages for refactoring	No/Yes	has "development" in job title & not students or faculty members	50

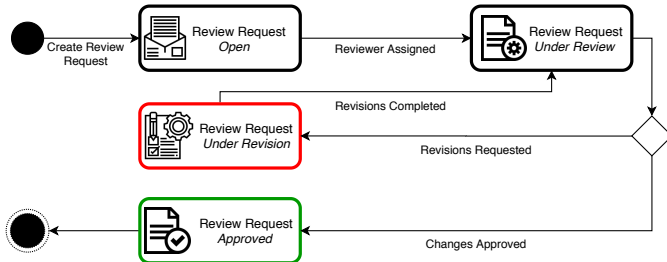


Figure (1) Review process overview.

[12], [19]–[21]. Our first research question seeks to understand what motivations drive code review involving refactoring in various development contexts to augment our understanding of refactorings in theory versus in practice.

RQ2. How do developers document their refactorings for code review? Since there is no consensus on how to formally document refactoring activities [22]–[24], we aim in this research question to explore what information developers have explicitly provided, and what keywords developers have used when documenting refactoring changes for a review. This question aims to capture the taxonomy used and observe whether it is currently helpful in providing enough insights for reviewers to be able to adequately assess the proposed changes to the software design.

RQ3. What challenges do reviewers face when reviewing refactoring changes? We investigate the challenges associated with refactoring, as well as the bad refactoring practices that developers catch when reviewing refactoring changes. This sheds light on how developers should mitigate some of these challenges.

RQ4. What mechanisms are used by developers and reviewers to ensure code correctness after refactoring? We pose this research question to study current approaches for testing behavior preservation of refactoring, and to get an overview of what different criteria are addressed by these approaches.

RQ5. How do developers and reviewers assess and perceive the impact of refactoring on the source code quality? Finally, in our last research question, we are interested in understanding how refactoring connects current research and practice. This helps exploring if the implications or outcomes of refactoring-aware code review match what outlined in the previous research questions.

B. Research Context and Setting

Host Company and Object of Analysis. To answer the above-mentioned research questions, we conducted our survey

with developers from the research and development division, at Xerox Research Center Webster (XRCW), currently Xerox’s largest research center. The research and development division is responsible for implementing and maintaining the software that is currently being shipped with Xerox Printers, (*i.e.*, ConnectKey interface technology¹). The software is directly connected to the hardware and performs various operations going from basic scanning and printing to more complex commands such as exchanging with cloud services. The software is constructed using object-oriented, object-based and markup languages. Despite being a legacy, around 20 years old, lengthy and complex software, the developers in charge have been successfully evolving it to meet business requirements and provide secure and reliable functionality to end users. This reflects the maturity of the engineering process within the research and development division, which raised our interest to understand how they perform code review in general, and how they review refactoring in particular.

Code Review Process at Xerox. The research and development division uses a collaborative code review framework allowing developers to directly tag submitted code changes and request its assignment to a reviewer. Similar to existing modern code review platforms, *e.g.*, Gerrit², a code change author opens a code Review Request (ReR) containing a title, a detailed description of the code change being submitted, written in natural language, along with the current code changes annotated. Once an ReR is submitted, it appears in the requests backlog, open for reviewers to choose. If an ReR remains open for more than 72 hours, a team leader would handle its assignment to reviewers. Once reviewers are assigned to the ReR, they inspect the proposed changes and comment on the ReR’s thread, to start a discussion with the author, just like a forum or a live chat. 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 (*i.e.*, *ship it!*) or decline, and so the proposed code changes are either “Merged” to production or “Abandoned”. An activity diagram, modeling a simplified bird’s view of the code review process, is shown in Figure 1.

C. Pilot Study and Motivation

Rationale. As we were analyzing the review process, to prepare our survey, we had access to the code review platform, containing the team’s history of processed ReRs for

¹<https://www.xerox.com/en-us/innovation/insights/connectkey-interface-technology>

²<https://www.gerritcodereview.com/>

Table (II) Summary of survey questions (the full list is available in [25]).

Category	Question
Background	(1) How many years have you worked in the software industry? (2) How many years have you worked on refactoring? (3) How many years have you worked on code review?
Motivation	(4) As a code change author, in which situation(s) you typically refactor the code?
Documentation	(5) As a code change author, what information do you explicitly provide when documenting your refactoring activity? (6) As a code change author, what phrases (keywords) have you used when documenting refactoring changes for a review?
Challenge	(7) As a code reviewer, what challenges have you face when reviewing refactoring changes? (8) As a code reviewer, what are the bad refactoring practices you typically catch when reviewing refactoring changes?
Verification	(9) As a code change author/code reviewer, what mechanism(s) do you use to ensure the correctness after the application of refactoring?
Implication	(10) As a code reviewer, what implication(s) do you typically experience as software evolves through refactoring? (11) How strongly do you agree with each of the following statements? <ul style="list-style-type: none"> • <i>I have guidelines on how to document refactoring activities.</i> • <i>I have guidelines on how to review refactoring activities while performing code review.</i> • <i>Reviewing refactoring activities slow down the review process.</i> • <i>Reviewing refactoring typically takes longer to reach a consensus.</i>

Table (III) Participant professional development experience in years.

Years of Experience	Industrial Experience (%)	Refactoring Experience (%)	Code Review Experience (%)
1-5	9 (37.5%)	15 (62.5%)	14 (58.33%)
6-10	5 (20.83%)	3 (12.5%)	4 (16.66%)
11-15	4 (16.66%)	1 (4.16%)	2 (8.33%)
16+	6 (25%)	5 (20.83%)	4 (16.66%)

the ConnectKey software system. After reviewing various ReRs, we noticed the existence of a number of refactoring-specific ReRs, *i.e.*, requests to specifically review a refactored code. The existence of such refactoring ReRs raised our curiosity to further study in deeper whether these ReRs are more difficult to resolve than other non-refactoring ReRs. We hypothesize that refactoring ReRs, take longer time and trigger more discussions between developers and reviewers before reaching a decision and closing the ReR. If such hypothesis holds, then it further justifies the need for a more detailed survey targeting these refactoring ReRs.

Extraction of Review Requests Metadata. We aim to identify all recent refactoring ReRs. Similarly to Kim et al. [12], we start with scanning the ReRs repository to distinguish ReRs whose title or description contains the keyword “refactor*”. We only considered recent reviews, which were created between January 2019 and December 2019. We chose to analyze recent ReRs to maximize the chance of developers, who authored or reviewed them, as still within the company. We manually analyze the extracted set to verify that each selected ReR is indeed about requesting the review of a proposed refactoring. This extraction and filtering process resulted in identifying 161 refactoring ReR. To perform the comparison, we need to sample 161 non-refactoring ReR from the remaining ones in the review framework. To ensure the representativeness of the sample, we use the stratified random sampling by choosing ReRs which were (1) created between January 2019 and December 2019; (2) created by the same set of authors of the refactoring ReRs; and (3) created to update the same subsystem(s) that were also updated by the refactoring ReRs.

We then compared both groups based on two factors: (1) review duration (time from starting the review until a decision of close/merge is made), and (2) number of exchanged responses (*i.e.*, review comments) between the author and reviewer(s). Figure 2 reports the boxplots depicting the distribution of each group values, clustered by two above-mentioned factors. To test the significance of the difference between the groups values, we use the Mann-Whitney U test, a non-parametric test that checks continuous or ordinal data for a significant difference between two independent groups. Our hypothesis is formulated to test whether the values of the refactoring ReRs group is significantly higher than the values of the non-refactoring ReRs group. The difference is considered statistically significant if the p-value is less than 0.05.

Pilot Study Results. According to Figure 2, refactoring code reviews take longer to be completed than the non-refactoring code reviews, as the difference was found to be statistically significant (*i.e.*, $p < 0.05$). Similarly, refactoring code reviews were found to significantly trigger longer discussion between the code author and the reviewers before reaching a consensus (*i.e.*, $p < 0.05$). This motivates us to better understand the challenges reviewers face when reviewing refactoring. We designed our survey to ask developers of this team about the kind of problems that triggers them to refactor, and to close the loop, we asked reviewers about what they foresee when they are assigned a refactoring code review, along with the issues they typically face for that type of assignment. The next subsection details our survey design.

D. Research Method

To answer our research questions, we follow a mixture qualitative and quantitative survey questions, as demonstrated in Creswell’s design [26]. The quantitative analysis was performed by the analysis of ReRs metadata, and the comparison between refactoring ReRs and non-refactoring ReRs, in terms of time to completion and number of exchanged responses. Developers survey constitutes the qualitative aspect that we are going to detail in the next section.

Survey Design. For our survey design, we followed the guidelines proposed by Kitchenham and Pfleeger [27]. To

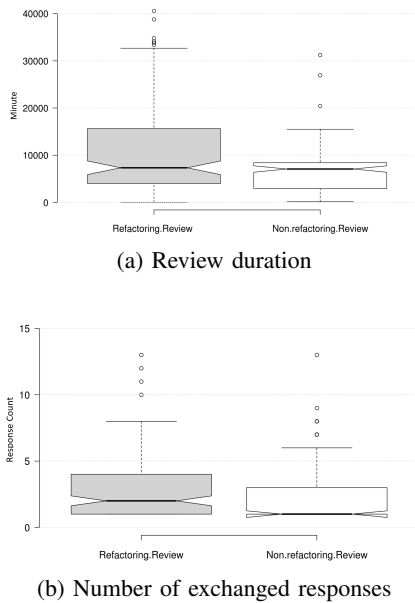


Figure (2) Boxplots of (a) review duration and (b) number of exchanged responses, for refactoring and non-refactoring code review.

344 increase the participation rate, we made our survey anonym-
 345 ous. The survey consisted of 11 questions that are divided
 346 into 2 parts. The first part of the survey includes demo-
 347 graphics questions about the participants. In the second part,
 348 we asked about the (1) motivations behind refactoring, (2)
 349 documentation of refactoring changes, (3) challenges faced
 350 when reviewing refactoring, (4) verification of refactoring
 351 changes, and (5) implications of refactoring on code quality.
 352 As suggested by Kitchenham and Pfleeger [27], we constructed
 353 the survey to use a 5-point ordered response scale (“Likert
 354 scale”) question on the general refactoring-related code review,
 355 2 open-ended questions on the refactoring documentation and
 356 challenges, and 5 multiple choice questions on the refactor-
 357 ing motivations, documentation, mechanisms and implications
 358 with an optional “Other” category, allowing the respondents
 359 to share thoughts not mentioned in the list. Table II contains
 360 a summary of the survey questions; the full list is available
 361 in [25]. In order to increase the accuracy of our survey, we
 362 followed the guidelines of Smith et al. [28], and we targeted
 363 developers who have previously been exposed to refactoring
 364 in the considered project. So instead of broadcasting the
 365 survey to the entire development body, we only intend to
 366 contact developers who have previously authored or reviewed
 367 a refactoring code change. We performed this subject selection
 368 criteria to ensure developers’ familiarity with the concept of
 369 refactoring so that they can be more prepared to answer the
 370 questions. This process resulted in emailing 38 target subjects
 371 who are currently active developers and regularly perform
 372 code reviews. Participation in the survey was voluntary. In
 373 total, 24 developers participated in the survey (yielding a
 374 response rate of 63%, which is considered high for software
 375 engineering research [28]). The industrial experience of the

respondents ranged from 1 to 35 years, their refactoring
 experience ranged from 1 to 30 years, and their experience
 in code review ranged from 1 to 25 years. On average, the
 participants had 10.7 years of experience in industry, 7.5 years
 of experience in refactoring, and 6.97 years of experience in
 code review. Table III summarizes developers’ experience in
 industry, refactoring and code review.

IV. RESULTS & DISCUSSIONS

A. RQ1. What motivates developers to apply refactorings in the context of modern code review?

Figure 3 shows developers’ intentions when they refactor their code. The *Code Smell* and *BugFix* categories had the highest number of responses, with a response ratio of 23.7% and 22.4%, respectively. The category *Functional* was the third popular category for refactoring-related commits with 21.1%, followed by the *Internal Quality Attribute* and *External Quality Attribute*, which had a ratio of 17.1% and 14.5%, respectively. However, we observe that all motivations do not significantly vary as all of them are in the interval 14.5% to 23.7% with no dominant category, as can be seen in Figure 3. Only one participant selected the “other” option stating that, “When i feel it’s painful to fulfill my current task without refactoring”.

If we refer to the Fowler’s refactoring book [1], refactoring is mainly solicited to enforce best design practices, or to cope with design defects. With bad programming practices, *i.e.*, code smells, earning 24% of developer responses, these results do not deviate from the Fowler’s refactoring guide. However, even though the code smell resolution category is prominent, the observation that we can draw is that motivations driving refactoring vary from structural design improvement to feature additions and bug fixes, *i.e.*, developers interleave refactoring with other development tasks. This observation is aligned with the state-of-the-art studies by Kim et al. [12], Silva et al. [19], and AlOmar et al. [21]. The sum of the design-related categories, namely code smell, internal, and external quality attributes represent the majority with 55.3%. These categories encapsulate all developers’ design-improvement changes that range from low level refactoring changes such as renaming elements to increase naming quality in the refactored design, and decomposing methods to improve the readability of the code, up to higher level refactoring changes such as re-modularizing packages by moving classes, reducing class-level coupling, increasing cohesion by moving methods, etc.

Summary: According to the survey, coping with poor design and coding style is the main driver for developers to apply refactoring in their code changes. Yet, functional changes and bug fixing activities often trigger developers to refactor their code as well.

B. RQ2. How do developers document their refactorings for code review?

When we asked developers, “what information do you explicitly provide when documenting your refactoring activity?”, 21

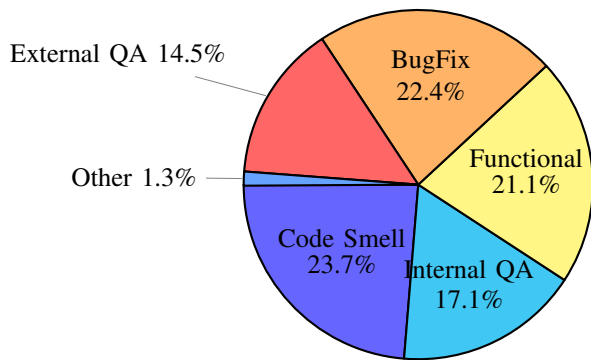


Figure (3) Developers' refactoring motivations for code review.

425 out of the 24 developers (91.3%) indicated that they explicitly
 426 mention the motivation behind the application of refactoring
 427 such as 'improving readability' and 'eliminate code smell'.
 428 Moreover, only 8 out of the 24 developers (34.8%) indicated
 429 their refactoring strategy by stating explicitly the type of
 430 refactoring operation they perform in their submitted code
 431 change description, such as 'move class'. We observe that
 432 developers are eager to explain the rationale of their refact-
 433 oring more than the actual refactoring operations performed.
 434 Due to the nature of inspection, developers need to develop a
 435 "case" to justify the need for refactoring, in order to convince
 436 the reviewers. Therefore, the majority of participants (91.3%)
 437 focus on reporting the *motivation* rather than the *operation*.
 438 Moreover, the identification of the operations can be deduced
 439 by the reviewers when they inspect the code before and after
 440 its refactoring. Finally, only a few respondents (6 participants)
 441 responded that they thoroughly document their refactoring by
 442 reporting both the *motivation* and *operation*. Moreover, when
 443 we asked, "what typical keywords you use when documenting
 444 refactoring changes for a review?", the developers answers
 445 contain various refactoring phrases. Table IV enumerates these
 446 patterns (keywords in bold indicate that the keyword was
 447 mentioned by more than one developer).

448 Table IV is quite revealing in several ways. First, we observe
 449 that developers state the motivation behind refactoring, and
 450 that some of these patterns are not restricted only to fixing
 451 code smells, as in the original definition of refactoring in
 452 Fowler's book [1]. Second, developers tend to use a variety of
 453 textual patterns to document their refactoring activities, such as
 454 'refactor', 'clean up', and 'best practice'. These patterns can
 455 be (1) generic to describe the act of refactoring without giving
 456 any details; or (2) specific to give more insights on how mainly
 457 provide a generic description/motivation of the refactoring
 458 activity such as 'improving readability'. A common trend
 459 amongst developers is that they either report a problem to
 460 indicate that refactoring action is needed (e.g., 'duplicate',
 461 'bugs', 'bad code', etc.), or they state the improvement to the
 462 code after the application of refactoring (e.g., 'best practice',
 463 'ease of use', 'improving code quality', etc.). By looking at
 464 the refactoring discussion (see Figure 2), we realized that
 465 developers do ask for more details to understand the performed

Table (IV) List of refactoring keywords reported by the participants.

Patterns		
(1) allow easier integration with	(16) fix	(31) remove legacy code
(2) bad code	(17) improving code quality	(32) replace hard coded
(3) bad management	(18) loose coupling	(33) reorganiz*
(4) best practice	(19) moderniz*	(34) restructur*
(5) break out	(20) modif*	(35) rewrit*
(6) bugs	(21) modulariz*	(36) risks
(7) cleanup	(22) not documented	(37) simply
(8) cohesion	(23) open close	(38) single responsibility
(9) comment	(24) optimiz*	(39) single level of abstraction
(10) complexity	(25) performance	per function
(11) consistency	(26) readability	(40) splitting logic
(12) decouple	(27) redundancy	(41) strategy pattern
(13) duplicate	(28) refactor*	(42) stress test results
(14) ease of use	(29) regression	(43) testing
(15) extract class	(30) remov*	(44) uncomment

refactoring activities.

Summary: Developers rarely report specific refactoring operations as part of their documentation. Instead, they use general keywords to indicate the motivation behind their refactorings. Nevertheless, several patterns are solicited by developers to describe their refactorings. With the lack of refactoring documentation guidelines, reviewers are forced to ask for more details in order to recognize the need for refactoring.

C. **RQ3.** What challenges do reviewers face when reviewing refactoring changes?

As shown in Figure 4, we report the main challenges faced by reviewers when inspecting a refactoring review request. The majority of the developers (17 respondents (70.8%)) communicated that they were concerned about avoiding the introduction of regression in system's functionality. Interestingly, refactoring by default, ensures the preservation of the system's behavior through a set of pre and post conditions, yet, reviewers main focus was to validate the behavior of the refactored code. In this context, a recent study have shown that developers do not rely on built-in refactoring in their Integrated Development Environments (IDEs) and they perform refactoring manually [19], e.g., when moving a method from one class to another, instead of activating the 'move method' from the refactoring menu, developers prefer to cut and paste the method declaration into its new location, and manually update any corresponding memberships and dependencies. Such process is error prone, and therefore, reviewers tend to treat refactoring like any other code change and inspect the functional aspect of any refactored code.

In Figure 4, 14 developers (58.3%) revealed the need to investigate the impact of refactoring on software quality. Such investigation is not trivial, as it has been the focus of a plethora of previous studies (e.g., [29]), finding that not all refactoring operations have *beneficial* impact on software quality, and so developers need to be careful as various design and coding defects may require different types of refactorings. In this context, we identified, in our previous study [23] which

497 structural metrics (coupling, complexity, etc.) are aligned
498 with the developer’s perception of quality optimization when
499 developers explicitly mention in their commit messages that
500 they refactor to improve these quality attributes. Interestingly,
501 we observed that, not all structural metrics capture developers
502 intentions of improving quality, which indicated the existence
503 of a gap between what developers consider to be a design
504 improvement, and their measurements in the source code.
505 When asked about their quality verification process, developers
506 use, as part of their internal process, the Quality Gate of
507 SonarQube. While SonarQube is a popular, widely adopted
508 quality framework, it suffers, like any other static analysis
509 tools, from the high false positiveness of its findings, when
510 it is not properly tuned.

511 A moderate subset of 11 developers (45.8%) were con-
512 cerned about having inadequate documentation about refact-
513 oring, whereas 10 developers (41.7%) were concerned about
514 understanding the motivations for refactoring changes. 9 de-
515 velopers (37.5%) found that reviewing refactoring changes in a
516 timely manner is difficult, whereas 6 of them (25%) found that
517 the challenge is centered around understanding how refactor-
518 ing changes were implemented. In addition to these challenges,
519 two participants stated, “*The quality of code readability (being*
520 *able to understand what the code author intended to do with*
521 *the logic/algorithm even without documentation*”, and “*Style*
522 *changes or personal preference that the author holds and feels*
523 *strongly about*”.

524 To get a more qualitative sense, we also study bad refactor-
525 ing practices that reviewers catch when reviewing refactoring
526 changes. We analyzed the survey responses to this open ques-
527 tion to create a comprehensive high-level list of bad refactoring
528 practices that are being caught by reviewers. These practices
529 are centered around five main topics: (1) interleaving refact-
530 oring with multiple other development-related tasks, (2) lack
531 of refactoring documentation, (3) avoiding refactoring negative
532 side effects on software quality, (4) inadequate testing, and (5)
533 lack of design knowledge. In the rest of this subsection, we
534 provide more in-depth analysis of these refactoring practices.

535 **Challenge #1: Interleaving refactoring with multiple other**
536 **development-related tasks.** One participant indicated that,
537 “*Refactoring changes are intermixed with bug fix changes*”
538 and another mentioned “*Refactoring after adding to many*
539 *features*”, indicating that these practices are not desirable when
540 performing or reviewing refactoring changes. This suggests
541 that interleaving refactoring with bug fixes and new features
542 could be a challenge from a reviewer’s point of view. Even
543 though we did not ask a specific question concerning interlea-
544 ving refactorings with other development-related context, three
545 participants acknowledged that mixing refactoring with any
546 other activity is a potential problem. This can be explained by
547 the fact that behavior preservation cannot be guaranteed and
548 it may introduce new bugs.

549 **Challenge #2: Lack of refactoring documentation.** In con-
550 trast with how developers document bug fixes and functional
551 changes, the documentation of refactoring seems to be vague

and unstructured. If we refer to our findings in our previ- 552
ous research question, developers lack guidelines on how to 553
describe their refactoring activities, and they refer to their 554
personal interpretation to justify their decisions. To mitigate 555
this ambiguity, there is a need for proper methodology that 556
articulates how developers should document refactoring code 557
changes. Reviewers did explicitly share their concerns during 558
the survey: 559

560 “1. Lack of documentation, 2. Inconsistent variable nam- 560
ing, 3. Unorganized code, 4. No explanation why changes 561
were made [...]”; “[...],no guideline, different guidelines 562
used in the project, bad code practices”; “[...] Not enough 563
comments” 564

565 **Challenge #3: Avoiding refactoring negative side effects on** 565
software quality. The majority of the participants commented 566
that wrongly naming code elements and duplicate code are the 567
common bad refactoring practices that they typically catch. It 568
has been proven by previous studies that a developer may 569
accidentally introduce a design anti-pattern while trying to 570
fix another (e.g., [30]). One mentioned example was how a 571
long method (large in lines of code, and has more than one 572
functionality) can be fixed by splitting the method into two, 573
using the *extract method* refactoring operation. However, if the 574
split does not create two cohesive methods (i.e., segregation 575
of concerns), then the results could be two tightly coupled 576
methods, which one method can envy the other method’s 577
attributes (i.e., feature envy anti-pattern). Thus, it is part of the 578
code review to verify the impact of refactoring on the software 579
design from different perspectives (e.g., code smell removal, 580
adherence to object-oriented design practices such as SOLID 581
and GRASP, etc.). We report samples of the participants’ 582
comments below to illustrate this challenge: 583

584 “*Poorly named methods, poorly named variables, lack of* 584
basic Object Oriented Design principles and concepts, 585
increased complexity, increased coupling.”; “*duplication,* 586
low-cohesion”; “*Code refactoring does not follow the* 587
coding standards set by the project. [...]”; “*Tight coup-* 588
ling, Lack of tests, convoluted logic, inconsistent variable 589
names, outdated comments” 590

591 **Challenge #4: Inadequate testing.** By default, refactoring is 591
supposed to preserve the behavior of the software. Ideally, 592
using the existing unit tests to verify that the behavior is 593
maintained should be sufficient. However, since refactoring 594
can also be interleaved with other tasks, then there might be a 595
change in the software’s behavior, and so, unit tests, may not 596
capture such changes if they were not revalidated to reflect 597
the newly introduced functionality. This can be a concern 598
if developers are unaware of such non behavior preserving 599
changes, and so, deprecated unit tests will not guarantee the 600
refactoring correctness. The following reviewers’ comments 601
illustrate this challenge: 602

603 “1) Not testing refactor code changes on all potential 603
impacted areas 2) Not adding newly named functions to 604
old test suites [...]”; “[...] partial testing process”; “[...] 605

606 *No follow-up testing*"; "[...] *No regression testing*"; "*Tight*
607 *coupling, Lack of tests [...]*"

608 **Challenge #5: Lack of design knowledge.** Developers typ-
609 ically refactor classes and methods that they recently and
610 frequently change. So, the more they change the same code
611 elements, the more confident they become about their design
612 decisions. However, not all team members have access to all
613 software codebase, and so they do not *draw the full picture*
614 of the software design, which makes their decision adequate
615 locally, but not necessarily at the global level. Moreover,
616 developers only reason on the actual *screenshot* of the current
617 design, and there is no systematic way for them to recognize
618 its evolution by, for instance, accessing previously performed
619 refactorings. This may also narrow their decision making, and
620 they may end up *reverting* some previous refactorings. These
621 concerns along others were also raised by participants, for
622 instance, one participant stated:

623 "*Lack of knowledge about existing design patterns in code*
624 *(strategy, builder, etc.) and their context along with lack*
625 *of knowledge about SOLID principles (especially open*
626 *close and dependency inversion). I've seen people claim*
627 *that the code cannot be tested but in reality the problem*
628 *is in the way they've structured their code.*"

629 It is clear that the code review plays also a major role in
630 knowledge transfer between junior and senior developers, and
631 in educating software practitioners about writing clean code
632 that meet quality standards.

Summary: Challenges of reviewing refactored code inherits challenges of reviewing traditional code changes, as refactoring can also be mixed with functional changes. Reviewers also report the lack of refactoring documentation, and inspect any negative side effects of refactorings on design quality. The inadequate testing of such changes hinder the safety of the performed refactoring. Finally, the lack of developer's exposure to whole system design can reduce the visibility of their refactoring decision making.

634 **D. RQ4.** *What mechanisms are used by developers and re-*
635 *viewers to ensure code correctness after refactoring?*

636 Developers reported mechanisms to verify the application
637 of refactoring (see Figure 5). 23 of the participants (95.8%)
638 refer to testing the refactored code; 17 (70.8%) reported
639 doing manual validation; 11 (45.8%) brought up ensuring the
640 improvement of software quality metrics; 9 (37.5%) mentioned
641 using visualization techniques; and 9 (37.5%) selected running
642 static checkers and linters. Besides performing testing, two
643 participants mentioned in the "other" option: "*Automated Test*
644 *Coverage*", and "*Existing Unit tests*".

645 We observe that reviewers treat refactoring like any tradi-
646 tional code change, and they unit-test it for correctness.
647 This eventually minimizes the introduction of faults. However,
648 when developers assume refactoring is preserving the behavior,

649 while it is not, then they may not have updated their unit
650 tests, and so their execution later by reviewers can become
651 unpredictable, *i.e.*, some test cases may or may not fail because
652 of their deprecation. Furthermore, some refactoring operations,
653 such as '*extract method*', do create new code elements that
654 are not covered by unit tests. So reviewers need to enforce
655 developers to write test cases for any newly introduced code.

656 Reviewers also refer to the quality gate to inspect if they
657 refactoring did not introduce any design debt or anti-patterns
658 in the system. Yet, the manual inspection of the code is still the
659 rules, some reviewers refer to visualizing the code before and
660 after refactoring to verify the completeness of the refactoring.

Summary: Since reviewers unit test refactoring, just like any other code change, developers need to add or update unit tests to the newly introduced or refactored code. Furthermore, reviewers are manually inspecting the refactored code to guarantee its correctness.

661
662 **E. RQ5.** *How do developers and reviewers assess and perceive*
663 *the impact of refactoring on the source code quality?*

664 As can be seen from Figure 6, all participants (24, 100%)
665 replied that the code becomes more readable and understand-
666 able. Intuitively, the main purpose of refactoring, is to ease
667 the maintenance and evolution of software. So reviewers,
668 implicitly consider refactoring to be an opportunity to *clean*
669 the code and make it adhere to the team's coding conventions
670 and style. Also, 12 (50%) indicated that it becomes easier to
671 pass Sonar Qube's Quality Gate. So, it is expected that the
672 refactored code does not increase the quality deficit index, if
673 not decreasing it. Finally, 11 (45.8%) stated their expectation
674 that refactored, through better renames, and more modular
675 objects, should reduce the code's proneness to bugs.

Summary: Besides using Quality Gates and static checkers to assess the impact of refactoring on the software design, reviewers rate the success of refactoring to the extent to which the refactored code has improved in terms of readability and understandability.

676 V. RECOMMENDATIONS 677

678 A. Recommendations for Practitioners 679

680 It is heartening for us to realize that developers refactor
681 their code and perform reviews for the refactored code. Our
682 main observation, from developers' responses, is how the
683 review process for refactoring is being hindered by the lack
684 of documentation. Therefore, as part of our survey report to
685 the company, we designed a procedure for documenting any
686 refactoring ReR, respecting three dimensions that we refer to
687 as the three *Is*, namely, *Intent*, *Instruction*, and *Impact*. We
688 detail each one of these dimensions as follows:

689 **Intent.** According to our survey results, (*cf.*, Figure 3), it
690 is intuitive that reviewers need to understand the purpose of
691 the intended refactoring as part of evaluating its relevance.

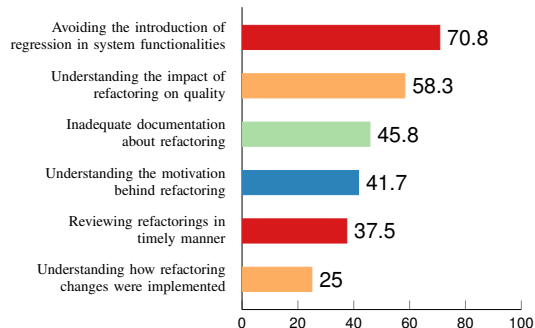


Figure (4) Challenges faced by developers when reviewing refactoring.

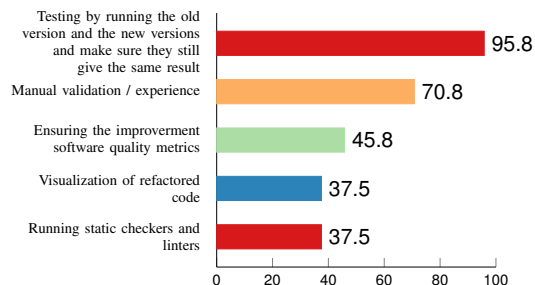


Figure (5) Mechanisms used to ensure the correctness after the application of refactoring.

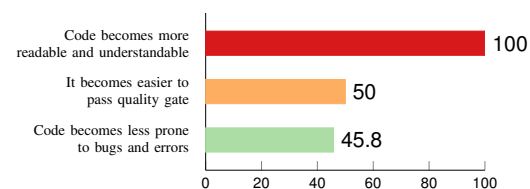


Figure (6) Implications experienced as software evolves through refactoring.

691 Therefore, when preparing the request for review, developers
 692 need to start with explicitly stating the motivation of the
 693 refactoring. This will provide the context of the proposed
 694 changes, for the reviewers, so they can quickly identify how
 695 they can comprehend it. According to our initial investigations,
 696 examples of refactoring intents, reported in Table IV, include
 697 *enforcing best practices*, *removing legacy code*, *improving*
 698 *readability*, *optimizing for performance*, *code clean up*, and
 699 *splitting logic*.

700 **Instruction.** Our second research question shows how rarely
 701 developers report refactoring operations as part of their docu-
 702 mentation. Developers need to clearly report all the refactor-
 703 ing operations they have performed, in order to allow their
 704 reproducibility by the reviewers. Each instruction needs to
 705 state the type of the refactoring (move, extract, rename, etc.)
 706 along with the code element being refactored (*i.e.*, package,
 707 class, method, etc.), and the results of the refactoring (the
 708 new location of a method, the newly extracted class, the new
 709 name of an identifier, etc.). If developers have applied batch or
 710 composite refactorings, they need to be broken down for the

711 reviewers. Also, in case of multiple refactorings applied, they
 712 need to be reported in their execution chronological order.

713 **Impact.** We observe from Figures 4 and 6 that practitioners
 714 care about understanding the impact of the applied refactoring.
 715 Thus, the third dimension of the documentation is the need to
 716 describe how developers ensure that they have correctly imple-
 717 mented their refactoring and how they verified the achievement
 718 of their intent. For instance, if this refactoring was part of a
 719 bug fix, developers need to reference the patch. If developers
 720 have added or updated the selected unit tests, they need to
 721 attach them as part of review request. Also, it is critical to self-
 722 assess the proposed changes using Quality Gate, to report all
 723 the variations in the structural measurements and metrics (*e.g.*,
 724 coupling, complexity, cohesion, etc.), and provide necessary
 725 explanation in case the proposed changes do not optimize the
 726 quality deficit index.

727 Upon its acceptance for trial at Xerox, a set of developers
 728 have adopted the *Is* procedure when submitting any refactoring
 729 related code change. These developers were initially given
 730 support for adopting it by us rewriting samples of their previ-
 731 ous code review requests, using our template. We will closely
 732 monitor its adoption, and perform any necessary tweaking. We
 733 also plan on following up on whether this practice was able
 734 to be beneficial for reviewers by (1) empirically validating
 735 whether refactoring ReRs, using our template, take less time
 736 to be reviewed, in comparison with other refactoring ReRs;
 737 and (2) rescheduling another follow up interview with the
 738 developers have been using it.

739 B. Recommendations for Research and Education

740 **Program Comprehension.** Refactoring for readability was
 741 pointed out by the majority of participants. In contrast with
 742 structural metrics, being automatically generated by the Qual-
 743 ity Gate, reviewers are currently relying on their own inter-
 744 pretation to assess the readability improvement, and such
 745 evaluation can be subjective and time-consuming. There is
 746 a need for a refactoring-aware code readability metrics that
 747 specifically evaluate the code elements that were impacted
 748 by the refactoring. Such metrics help in contextualizing the
 749 measurement to fulfill the developer's intention.

750 **Teaching Documentation Best Practices.** Prospective soft-
 751 ware engineers are mainly taught how to model, develop and
 752 maintain software. With the growth of software communities,
 753 and their organizational and socio-technical issues, it is im-
 754 portant to also teach the next generation of software engineers
 755 the best practices of refactoring documentation. So far, these
 756 skills can only be acquired by experience or training.

757 VI. THREATS TO VALIDITY

758 **Construct & Internal Validity.** Concerning the complete-
 759 ness and correctness of our interpretation of open responses
 760 within the survey, we did not extensively discuss all responses
 761 because some of them are open to various interpretations,
 762 and we need further follow up surveys to clarify them.
 763 Concerning the selection criteria of the participants, we tar-
 764 geted participants whose code review description included the
 765 keyword "refactor*". Since the validity of our study requires

familiarity with the concept of refactoring, we assume that participants who used this keyword know the meaning and the value of refactoring. Another potential threat relates to the communication channel to identify the motivation driving code review involving refactoring. We examined threaded discussions and some situations may not have been easily observable. For example, determining whether the reviewer confusion was primarily caused by the refactoring and not by another phenomenon is not practically easy to assess through discussions. Interviewing developers would be a good direction to consider in the future to capture such motivations.

External Validity. Concerning the representativeness of the results, we designed our study with the goal of better understanding developer perception of code review involving refactoring actions within a specific company. Further research in this regard is needed. As with every case study, the results may not generalize to other contexts and other companies. But extending this survey with the open-source communities is part of our future investigation to challenge our current findings.

VII. CONCLUSION

Understanding the practice of refactoring code review is of paramount importance to the research community and industry. In this work, we aim to understand the motivations, documentation, challenges, mechanisms and implications of refactoring-aware code review by carrying out an industrial case study of 24 software engineers at Xerox. In summary, we found that: (1) refactoring is completed for a wide variety of reasons, going beyond its traditional definition, such as reducing the software's proneness to bugs, (2) refactoring-related patterns mainly demonstrate developer perception of refactoring, but practitioners sometimes provide information about refactoring operations performed in the source code, (3) participants considered avoiding the introduction of regression in system functionality as the main challenge during their review, (4) although participants do use different static checkers, testing is the main driver for developers to ensure correctness after the application of refactoring, and (5) readability and understandability improvement is the primary implications of refactoring on software evolution.

VIII. ACKNOWLEDGEMENTS

We would like to thank the Software Development Manager Wendy Abbott for approving the survey and all Xerox developers who volunteered their time to participate in this research.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and d. Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[3] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *International conference on software engineering*, pp. 712–721, 2013.

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

[5] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 56–75, 2016.

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

[7] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software modularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 1–45, 2015.

[8] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 23, 2016.

[9] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE software*, vol. 25, no. 5, pp. 38–44, 2008.

[10] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 33–36, ACM, 2011.

[11] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Working Conference on Reverse Engineering (WCRE)*, pp. 242–251, 2013.

[12] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring-challenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[13] G. Szöke, C. Nagy, R. Ferenc, and T. Gyimóthy, "A case study of refactoring large-scale industrial systems to efficiently improve source code quality," in *International Conference on Computational Science and Its Applications*, pp. 524–540, Springer, 2014.

[14] T. Sharma, G. Suryanarayana, and G. Samarthyam, "Challenges to and solutions for refactoring adoption: An industrial perspective," *IEEE Software*, vol. 32, no. 6, pp. 44–51, 2015.

[15] C. D. Newman, M. W. Mkaouer, M. L. Collard, and J. I. Maletic, "A study on developer perception of transformation languages for refactoring," in *International Workshop on Refactoring*, pp. 34–41, 2018.

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

[17] E. L. Alves, M. Song, T. Massoni, P. D. Machado, and M. Kim, "Refactoring inspection support for manual refactoring edits," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 365–383, 2017.

[18] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2017.

[19] D. Silva, N. Tsantalis, and 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, FSE 2016*, (New York, NY, USA), pp. 858–870, ACM, 2016.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, pp. 5–18, Jan 2012.

[21] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini, "How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation," *Expert Systems with Applications*, p. 114176, 2020.

[22] E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages," in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWor)*, pp. 51–58, IEEE, 2019.

[23] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and 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)*, pp. 1–11, IEEE, 2019.

[24] E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "Toward the automatic classification of self-affirmed refactoring," *Journal of Systems and Software*, vol. 171, p. 110821, 2020.

[25] AlOmar, <https://smilevo.github.io/self-affirmed-refactoring/>, 2020 (last accessed October 16, 2020).

[26] J. W. Creswell, "Research design: Quantitative, qualitative and mixed methods," 2009.

[27] B. A. Kitchenham and S. L. Pflieger, "Personal opinion surveys," in *Guide to advanced empirical software engineering*, pp. 63–92, Springer, 2008.

[28] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann, "Improving developer participation rates in surveys," in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 89–92, IEEE, 2013.

[29] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[30] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.