Empirical Software Engineering (2021) 26: 12
https://doi.org/10.1007/s10664-020-09909-5

An exploratory study on confusion in code reviews

Felipe Ebert[1] · **Fernando Castor**[2] · **Nicole Novielli**[3] · **Alexander Serebrenik**[1]

Accepted: 23 October 2020 / Published online: 27 January 2021
© The Author(s) 2021

Abstract

Context Code review is a widely used technique of systematic examination of code changes which aims at increasing software quality. Code reviews provide several benefits for the project, including finding bugs, knowledge transfer, and assurance of adherence to project guidelines and coding style. However, code reviews have a major cost: they can delay the merge of the code change, and thus, impact the overall development process. This cost can be even higher if developers do not understand something, i.e., when developers face *confusion* during the code review.

Objective This paper studies the phenomenon of *confusion* in code reviews. Understanding confusion is an important starting point to help reducing the cost of code reviews and enhance the effectiveness of this practice, and hence, improve the development process.

Method We conducted two complementary studies. The first one aimed at identifying the reasons for confusion in code reviews, its impacts, and the coping strategies developers use to deal with it. Then, we surveyed developers to identify the most frequently experienced reasons for confusion, and conducted a systematic mapping study of solutions proposed for those reasons in the scientific literature.

Results From the first study, we build a framework with 30 reasons for confusion, 14 impacts, and 13 coping strategies. The results of the systematic mapping study shows 38 articles addressing the most frequent reasons for confusion. From those articles, we found 13 different solutions for confusion proposed in the literature, and five impacts were established related to the most frequent reasons for confusion.

Conclusions Based on the solutions identified in the mapping study, or the lack of them, we propose an actionable guideline for developers on how to cope with confusion during code reviews; we also make several suggestions how tool builders can support code reviews. Additionally, we propose a research agenda for researchers studying code reviews.

Keywords Code reviews · Confusion · Card sorting · Survey · Systematic mapping study

Communicated by: Massimiliano Di Penta, David C. Shepherd

This article belongs to the Topical Collection: *Software Analysis, Evolution and Reengineering (SANER)*

✉ Felipe Ebert
f.ebert@tue.nl

Extended author information available on the last page of the article.

1 Introduction

Code review is a technique of systematic examination of code changes. It can be conducted before or after the change is integrated into the main code repository (Rigby et al. 2008). Code changes submitted by a developer are reviewed by one or more of their peers. This is why code reviews are also known as *peer reviews* or *peer code reviews*. For the sake of simplicity, we use the term code review in this study.

Code review is an important practice for software quality assurance (Tao and Kim 2015; Bavota and Russo 2015; Boehm and Basili 2001; Mäntylä and Lassenius 2009; Barnett et al. 2015). Several open source projects, e.g., ANDROID,[1] QT,[2] and ECLIPSE,[3] as well as companies, e.g., MICROSOFT,[4] ORACLE,[5] and SAMSUNG,[6] adopt code review as part of their development process. Likewise, studies have also shown that code review can provide multiple benefits in the development process (Bacchelli and Bird 2013; Pangsakulyanont et al. 2014; Morales et al. 2015; Cohen et al. 2006; McIntosh et al. 2015).

The main goals of code reviews are to find bugs in the code change, and verify whether the project guidelines and coding style are being respected (Fagan 1976; Wiegers 2002; Wang et al. 2015; Bacchelli and Bird 2013; Bosu et al. 2017). Furthermore, code reviews help to improve the quality of the code on production, find better ways to implement the change, spread the knowledge about the project, and create awareness of the changes in the code base (Bacchelli and Bird 2013; Pangsakulyanont et al. 2014; Morales et al. 2015; Cohen et al. 2006; McIntosh et al. 2015).

Despite such benefits, code reviews can incur costs on software development projects, as they can delay the merge of a code change in the repository and, consequently, slow down the overall development process (Pascarella et al. 2018; Greiler 2016). The time invested by a developer in reviewing code is non-negligible (Tao and Kim 2015) and may take 10%–15% of the overall time invested in software development activities (Bosu et al. 2017; Cohen et al. 2006). Furthermore, performing a code review is not a trivial task per se. In fact, understanding the code change and its context is one of the major issues reviewers face during code reviews (Bacchelli and Bird 2013; Cohen et al. 2006; Tao et al. 2012; Sutherland and Venolia 2009; LaToza et al. 2006). The merge of a code change in the repository can be further delayed when reviewers experience difficulties in understanding the change, i.e., when they are not certain of its correctness, run-time behaviour and impact on the system (Cohen et al. 2006; Bacchelli and Bird 2013; Tao et al. 2012; Sutherland and Venolia 2009; LaToza et al. 2006).

We believe that *confusion*, i.e., *any situation where a person is uncertain about something or unable to understand something* (Ebert et al. 2017), can affect the artifacts that developers produce and the way they work, and hence, impact the development process (Cohen et al. 2006; Bacchelli and Bird 2013; Tao et al. 2012; Sutherland and Venolia 2009; LaToza et al. 2006). For instance, on the one hand, the code review might take longer than it should, the quality of the review might decrease, more discussions might take place, or even the code change might be blindly accepted or summarily rejected (Ebert et al. 2019). On the other

[1] https://android-review.googlesource.com

[2] https://codereview.qt-project.org

[3] https://git.eclipse.org/r

[4] https://queue.acm.org/detail.cfm?id=3292420

[5] https://smartbear.com/product/collaborator/overview

[6] https://www.perforce.com/case-studies/vcs/samsung

hand, confusion might lead reviewers and authors to reach an improved solution (Ebert et al. 2019). As such, we believe that a proper understanding of the phenomenon of *confusion* in code reviews is a necessary starting point towards reducing the cost of code reviews and enhancing the effectiveness of this practice, thereby improving the overall development process.

In this paper, we extend our previous study of the reasons and impact of confusion in code reviews, as well as the strategies developers adopt to deal with confusion (Ebert et al. 2019). In that study, we built a framework for confusion in code reviews including reasons, impacts, and the coping strategies adopted by developers. To do so, we employed a concurrent triangulation strategy combining a developer's survey and the analysis of code review comments. Our findings show that there are 30 different reasons for confusion, and that the three most prevalent ones relate to the missing rationale for the change, discussion of non-functional aspects of the solution, and the lack of familiarity with the existing project code. Furthermore, we observed that confusion can impact code reviews in 14 different ways. The most popular impacts are delaying, decrease of review quality, and the need for additional discussions. Finally, our framework includes 13 coping strategies developers reported to adopt when dealing with confusion in code reviews. The most prevalent strategies include requesting more information, improving own familiarity with the existing code, and engaging in off-line discussions.

The evidence provided by our previous study has several implications for both tool builders and researchers (Ebert et al. 2019). However, two factors motivated us to follow up on that study. The first factor is the relatively low number of coping strategies for confusion, (13), when compared to the number of reasons for confusion (30). This stems in part from the adopted methodology, since most of the discussion in the code reviews we examined revolves around the reasons for confusion (Ebert et al. 2019). The second factor is related to the contextualization of confusion in the literature, i.e., we want to discover to what extent different aspects of confusion are addressed in scientific studies. Code reviews has been extensively addressed by recent literature, and hence, we intend to identify suggested solutions for confusion in code reviews and, most importantly, summarize existing gaps, i.e., where future research should focus on. To contextualize our findings, we performed a systematic mapping study in order to identify mitigation strategies designed to address confusion, as well negative impacts of these factors going beyond confusion. The strategies might be beneficial for developers facing confusion and complement the currently employed coping mechanisms. To address these issues, we decided to conduct a deeper investigation of the solutions proposed and impacts identified in the scientific literature.

This paper extends our previous study by reporting on a systematic mapping study of the most frequently experienced reasons for confusion and solutions proposed for them. To identify the most frequently experienced reasons for confusion, we conduct a survey with 62 developers. Based on their answers, we selected the five most frequent reasons for confusion and performed a systematic mapping study of the Software Engineering literature to assess to what extent does the scientific literature discuss these reasons and identify solutions proposed in the literature for each one of them. Based on the identified solutions or the lack thereof, we propose an actionable guideline for developers on how to deal with confusion in code reviews. Furthermore, we propose a research agenda for researchers interested in studying how to provide support for developers experiencing confusion.

The remainder of this paper is organized as follows. Section 2 presents the background related to this study. In Section 3, we present our first study aimed at understanding the reasons for confusion, its impacts, and the strategies developers used to deal with it. In

Section 4, we present the preliminary study we conducted in order to identify the most frequent reasons for confusion according to developers. Next, in Section 5, we present the second study we conducted in order to investigate the solutions and impacts of the most frequent reasons for confusion proposed by literature. The discussion is presented in Section 6. The related work is discussed in Section 7. Finally, the conclusions and future work are presented in Section 8.

2 Background

In this section, we provide a background of code reviews in Section 2.1. Then, we present our definition of *confusion* in Section 2.2.

2.1 Code Reviews

Formal code review was first defined by Fagan in 1976 as *software inspections* (Fagan 1976). Software inspection, the most formal type of code review (Rigby and Bird 2013), is a structured process for reviewing source code that relies on rigid roles and steps, with the single goal of finding defects (Fagan 1976). Notwithstanding the initial success of Fagan's inspections with both the industry and research, its formality brings several drawbacks. Indeed, the inspections are very time consuming because the meetings need to be organised and the participants need to do some preparation. Another disadvantage is the chance of turning the inspection meeting into a political or social disaster (Wiegers 2002). Moreover, the formality of the inspection does not fit well with agile development methods (Martin 2003).

As a result, a more *lightweight* code review process with a better fit for test-driven and iterative development processes started to become more popular. Formalising this practice, Bacchelli and Bird (2013) defined the lightweight code review process as a "modern code review", which is a review that is informal (as opposed to Fagan's inspections), supported by code review tools, and occurs regularly in practice. We also use the term *code reviews* as a synonym for modern code reviews in this study.

The code review process is an iterative process and can be instantiated in different ways. As input, a code review receives the original code change and the outcome is the reviewed change, which might be either accepted or rejected. The developer who wrote the code change is the author, and might also be responsible for submitting the change for review. The reviewer is responsible for assuring that the code change is functionally correct, meets the performance requirements, and follows the quality standards of the project.

In general, there are two types of workflow for code reviews, depending on when the review is conducted in the development process:

- **Review-then-commit (pre-commit):** the code is reviewed before it is integrated into the main repository of the Version Control System (VCS) (Tichy 1985);
- **Commit-then-review (post-commit):** the code is reviewed after it is integrated into the main repository of the VCS (Tichy 1985);

Since the most common type of code review is *review-then-commit* (Rigby 2011), it will be the focus of this thesis. We present an example of the code review process within this approach in Fig. 1.

It starts with the author submitting the code change (1). The reviewers are notified and start reviewing the code change (2). They should check and verify it based on several quality

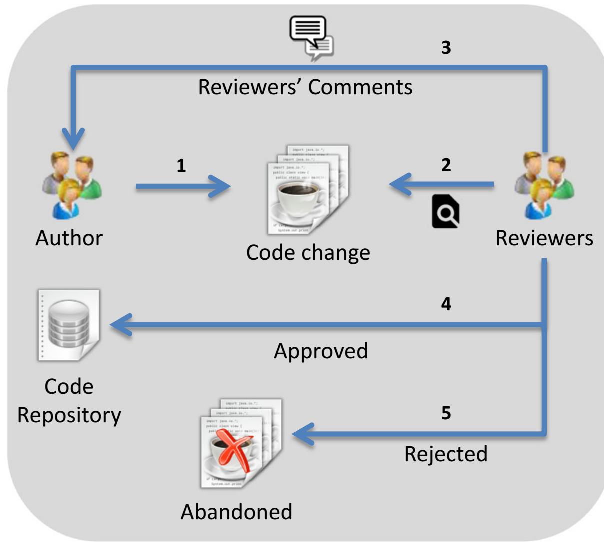

Fig. 1 The code review process

criteria, such as correctness, adherence to the project guidelines, and conventions. If the reviewers believe that the code change does not fulfil those requirements, they ask the author to fix it, or to submit a new one (3). Thus, the author needs to work on the code change and submit it again (1) for review (2). When the reviewers are satisfied that the code change is suitable, it is integrated into the code repository (4). However, if reviewers' quality criteria are not achieved by the code change, it is rejected, and the code review is abandoned (5). There might be several iterations before the reviewers decide to end the process (1 to 3), where the code change might be accepted (i.e., it is merged into the main repository), or rejected (i.e., it is discarded).

2.2 Confusion Definition

There are several studies which tried to model the *affective disequilibrium* related to confusion, uncertainty, and lack of knowledge, especially from the Psychology field. In this section, we discuss some of the most relevant studies on those topics.

The Merriam-Webster dictionary[7] provides the following definitions of the word **confusion**: (1) "a situation in which people are uncertain about what to do or are unable to understand something clearly" and (2) "the feeling that you have when you do not understand what is happening, what is expected, etc.", i.e., confusion is both the situation and a sentiment.

Armour (2000) suggested categorising ignorance into layers based on what we know and what we do not know. He defined the *Five Orders of Ignorance*:

- **0th Order Ignorance - Lack of Ignorance:** when we know something, i.e., it is knowledge;

[7] www.merriam-webster.com/dictionary/confusion

- **1st Order Ignorance - Lack of Knowledge:** when we do not know something, but we can easily identify that fact;
- **2nd Order Ignorance - Lack of Awareness:** when we do not know that we do not know something, i.e., when we are unaware of that fact;
- **3rd Order Ignorance - Lack of Process:** when we do not know a suitably efficient way to find out we do not know that we do not know something;
- **4th Order Ignorance - Meta Ignorance:** when we do not know about the Five Orders of Ignorance.

D'Mello and Graesser (2014) focused on *confusion* and how it impacts learning and problem solving. Similarly to the second definition of Merriam-Webster, D'Mello and Graesser consider confusion to be an affective state. According to the authors, confusion happens when an individual detects new or discrepant information, e.g., there is a conflict with prior knowledge. Jordan et al. (2012) investigated the frequency of *uncertainty expressions* in discussions of students using a computer-mediated environment. The authors introduced their own definition of *uncertainty* and provided a coding scheme to describe and model it. Acknowledging that defining uncertainty was not simple, Jordan et al. (2012) define *uncertainty* as: "*situations when individuals have a sense of wondering, doubt, or unease about how the future will unfold, what the present means, or how to interpret the past*".

We believe that lack of knowledge and confusion, which can also encompass doubt and uncertainty, are strictly linked (e.g., confusion could be determined as lack of knowledge) and are both actionable (D'Mello and Graesser 2014). Thus, we define *confusion* broadly as:

"*a situation where a person is uncertain about or unable to understand something*."

3 Understanding Confusion in Code Reviews (Ebert et al. 2019)

In this section, we summarize our previous study aimed at a framework for confusion in code reviews. Specifically, we investigated what are reasons for confusion (**RQ1**), its impacts (**RQ2**), and the strategies developers are using to deal with it (**RQ3**) (Ebert et al. 2019). To the best of our knowledge, our study on is the first one conducting a deep investigating of the phenomenon of confusion in code reviews. In Section 4 we build upon this study to get further insights in frequently experienced reasons for confusion.

We describe the methodology in Section 3.1. The results are presented in Section 3.2. Finally, we discuss the threats to validity in Section 3.3.

3.1 Methodology

To strengthen the validity of the study we follow the recommendation of Easterbrook et al. (2008) and opt for a concurrent triangulation strategy, which is a combination of different research methods. Firstly, we conduct a survey to understand "what developers say" (Section 3.1.1). Then, we analyze the code review comments to understand "what developers do" (Section 3.1.2). Finally, we compare and contrast the findings of the two analyses (Section 3.1.3): indeed, Easterbrook et al. (2008) observe that "what people say" could be different from "what people do".

3.1.1 Surveys

In the SE literature, a theory is missing to describe what are the reasons for confusion in code reviews, the impact of confusion on the development process, and what coping strategies developers employ to deal with confusion. As such, to answer our **RQs** we opt for grounded theory building (Glaser and Strauss 1967; Stol et al. 2016). We implement an iterative approach. During each iteration, we administer a survey to developers involved in code reviews. We ask developers that already answered the survey during one of the previous iterations to refrain from answering it again.

Survey Design The survey was designed according to the established best practices Groves et al. (2009), Kitchenham and Pfleeger (2008), Singer and Vinson (2002), and Steele and Aronson (1995): prior to asking questions, we explain the purpose of the survey and our research goals, disclose the sponsors of our research and ensure that the information provided will be treated in a confidential way. In addition, we inform the participants about the estimated time required to complete the survey, and obtain their informed consent. The invitation message includes a personalized salutation, a description of the criteria we used for participant selection, as well as the explanation that there would not be any follow up if the respondent did not reply. This last decision also implies that we did not send reminders.

The survey starts with the definition of confusion as provided in Section 2, followed by a question requiring the participants to confirm that they understood the definition. Next, we ask two series of questions: the questions were essentially the same but were first asked from the perspective of the author of the code change, and then from the perspective of the reviewer of the change (cf. Table 1). Each series starts with the Likert-scale question about the frequency of experienced confusion: *never*, *rarely*, *sometimes*, *often*, and *always*. To ensure that the respondents interpret these terms consistently we provide quantitative estimates: 0%, 25%, 50%, 75% and 100% of the time. For respondents who answered anything different from *never*, we pose four open-ended questions (to get the as rich as possible data (Foddy 1993)): i) what are the reasons for confusion, ii) whether they can provide an example of a practical situation where confusion occurred during a code review (**RQ1**), iii) what are the impacts of confusion (**RQ2**), and iv) how do they cope with confusion (**RQ3**). Finally, we ask the participants to provide information about their experience as developers and frequency of reviewing and authoring code changes. We ask these question at the end of the survey rather than at the beginning to reduce the *stereotype threat* (Steele and Aronson 1995). Prior to deploying the survey, we discussed it with other software engineering researchers and clarified it when necessary.

Participants The target population consists of developers who participated in code reviews either as a change author or as a reviewer. During the first iteration we target ANDROID developers who participated in code reviews on GERRIT: 4,645 of their email addresses provided by Ebert et al. (2017) allow us to contact the developers by email and evaluate the response rate. In the subsequent iterations, the survey was announced on FACEBOOK and TWITTER. As the exact number of developers participating in code reviews reached cannot be known we do not report the response rate for the follow-up surveys.

Data Analysis To analyze the survey data, we use a card sorting approach (Zimmermann 2016). We analyze the survey responses from the first iteration using *open card sorting* (Zimmermann 2016), i.e., topics were not predefined but emerged and evolved during

Table 1 Survey questions. The questions marked "*" were only used in the first survey, "+"—only in the second and third surveys

Electronic Consent	
0.	Please select your choice below. Selecting the "yes" option below indicates that: i) you have read and understood the above information, ii) you voluntarily agree to participate, and iii) you are at least 18 years old. If you do not wish to participate in the research study, please decline participation by selecting "No".
Definition of Confusion	
	The remainder of this survey is dedicated to "confusion". We do not make a distinction between lack of knowledge, confusion, or uncertainty. For simplicity reasons, we use the "confusion" to refer to all these terms.
1.	By clicking "next" you declare that you understand the meaning of confusion on this survey.
Review-Then-Commit	
2.+	Have you ever taken part in a "review-then-commit" type of code review (i.e., the code is reviewed before it is integrated into the main repository), either in the role of author or reviewer?
When reviewing code changes	
3.	Developers might feel confused or think that they do not understand the code they review. How often did you feel this way when reviewing code changes?
4.	What usually makes you confused when you are reviewing code changes? Please explain which factors led you to be confused.
5.	Please describe a change you have been reviewing that has confused you.
6.	How does the confusion you experience as a reviewer impact code review?
7.	What do you usually do to overcome confusion in code reviews? Please explain the actions you take when you feel confused.
8.*	When you do not understand a code change, do you usually express this in general comments or in inline comments? Please explain why in the "other" field.
When authoring code changes	
9.	Developers who authored code changes might feel confused or think that they do not understand something when their code is being reviewed. How often did you feel this way when your code has been reviewed?
10.	What usually makes you confused during the code review when you are the author of the code changes? Please explain which factors led you to be confused.
11.	Please describe a change you have been authoring that has confused you.
12.	How does confusion you experience as the code change author impact the code review?
13.	What do you usually do to overcome confusion in code reviews? Please explain the actions you take when you feel confused.
14.*	When you do not understand a code change, do you usually express this in general comments or in inline comments? Please explain why in the "other" field.
Background	
15.	What is your experience as a developer?
16.	What is your experience as a code reviewer?
17.	How often do you submit code changes to be reviewed?
18.	How often do you review code changes?
19.*	Do you have the merge approval right (i.e., the permission to give +2) in Gerrit at least for one software development project?

Table 1 (continued)

20.*	Which option would best describe yourself?
	I contribute to Android voluntarily.
	I'm employed by a company other than Google and I contribute to Android as part of my job.
	I'm employed by Google and I contribute to Android as part of my job.
	Other.
Results	
21.	Would you like to be informed about the outcome of this study and potential publications? Please leave a contact email address.
22.	Would you be willing to be interviewed afterwards?
23.	Please add additional comments below.

the sorting process. After each subsequent survey iteration, we use the results of the previous iteration to perform *closed card sorting* (Zimmermann 2016), i.e., we sort the answers of each survey iteration according to the topics emerging from the previous one. If the closed card sorting succeeds, this means that the saturation has been reached and sampling more data is not likely to lead to the emergence of new topics (Finfgeld-Connett 2014; Lenberg et al. 2017). In such a case the iterations stop. If, however, during the closed card sorting additional topics emerge, another iteration is required.

To facilitate analysis of the data we use axial coding (Kitchenham and Pfleeger 2008) to find the connections among the topics and group them into dimensions. These dimensions emerge and evolve during the final phase of the sorting process, and they represent a higher level of abstraction of the topics.

As we have multiple iterations and multiple surveys answered by different groups of respondents, a priori it is not clear whether the respondents can be seen as representing the same population. Indeed, it could have been the case that, e.g., respondents of the second survey happened to be less inclined to experience confusion than the respondents of the third survey and the reasons of their confusions are very different. This is why we first check similarity of the groups of respondents in terms of their experience as developers and code reviewers, frequency of submitting changes to be reviewed and reviewing changes as well as frequency of experiencing confusion. If the groups of respondents are found to be similar, we can consider them as representing the same population and merge the responses. If the groups of respondents are found to be different, we treat the groups separately. To perform the similarity check we use two statistical methods: i) analysis of similarities (ANOSIM) (Clarke 1993), which provides a way to test statistically if there is a significant difference between two or more groups of sampling units, and ii) permutational multivariate analysis of variance using distance matrices (PERMANOVA) (Anderson 2001; McArdle and Anderson 2001).[8]

3.1.2 Analysis of Code Review Comments

To triangulate the survey findings for the **RQs** we perform an analysis of code review comments. As a dataset we use the one provided by Ebert et al. (2017). Similarly to the

[8]Both methods are available as functions in the R package *vegan*. ANOSIM has been implemented by Jari Oksanen, with a help from Peter R. Minchin. ADONIS (PERMANOVA) has been implemented by Martin Henry H. Stevens and adapted to *vegan* by Jari Oksanen.

developers contacted during the first survey iteration, this dataset pertains to ANDROID. The code reviews of ANDROID are supported by GERRIT, which enables communication between developers during the process by using general and inline comments. The former are posted in the code review page itself, which presents the list of all general comments, while the inline comments are included directly in the source code file. The dataset of Ebert et al. comprises 307 code review comments manually labeled by the researchers as confusing: 156 are general and 151 are inline comments.

Similarly to the analysis of the survey data, we use card sorting to extract topics from the code review comments. We conduct an open card sorting of the general comments to account for the possibility of divergent results, i.e., we did not want to use the results from the surveys because what developers do often differs from what they think they do and the emergent codes might a priori be different from those obtained when analyzing the survey data. To confirm the topics emergent from the general comments we then perform a closed card sorting on the inline comments.

3.1.3 Triangulating the Findings

Recall that the goal of concurrent triangulation is to corroborate the findings of the study, increasing its validity. However, following Easterbrook et al. (2008) we expect to see some differences between 'what people say' (survey) and 'what people do' (code review comments). Hence, if the topics extracted from the surveys and code review comments disagree, we conduct a new card sorting round only on the cards associated with topics discovered on the basis of the survey but not on the basis of the code review comments, or vice versa. In order not to be influenced by the results of the previous card sorting, we perform open card sorting and exclude the researchers who participated in the previous card sorting rounds. Finally, in order to finalize the framework for confusion in code reviews, we perform the consistency check within the topics and deduction of more generic topics, as recommended by Zimmermann (2016), as well as a consistency check across **RQs** (i.e., reasons, impacts, and coping strategies) and emergent dimensions.

3.2 Results

We discuss the application of the research method in practice (Section 3.2.1), and analyze similarity between the responses received at each one of the survey iterations (Section 3.2.2). Then, we present the demographics results from the survey (Section 3.2.3), and discuss reasons for confusion (**RQ1**, Section 3.2.4), its impact (**RQ2**, Section 3.2.5), and the strategies employed to cope with it (**RQ3**, Section 3.2.6).

3.2.1 Implementation of Approach

The implementation of the approach designed in Section 3.1 is shown in Fig. 2. Individuals involved in the card sorting are graduate students in computer science or researchers.

First, following the iterative approach we have performed three iterations since saturation has been reached. Among the 4,645 emails sent during the first iteration, 880 emails have bounced; hence, 17 valid responses correspond to the response rate 0.45%. Such response rate was unexpected[9] and might have been caused by presence of inactive members

[9]The common response rates in Software Engineering range between 15% and 20% Palomba et al. (2015), Vasilescu et al. (2015a, b), Qiu et al. (2019) and sometimes much higher response rates are reported (Palomba et al. 2018).

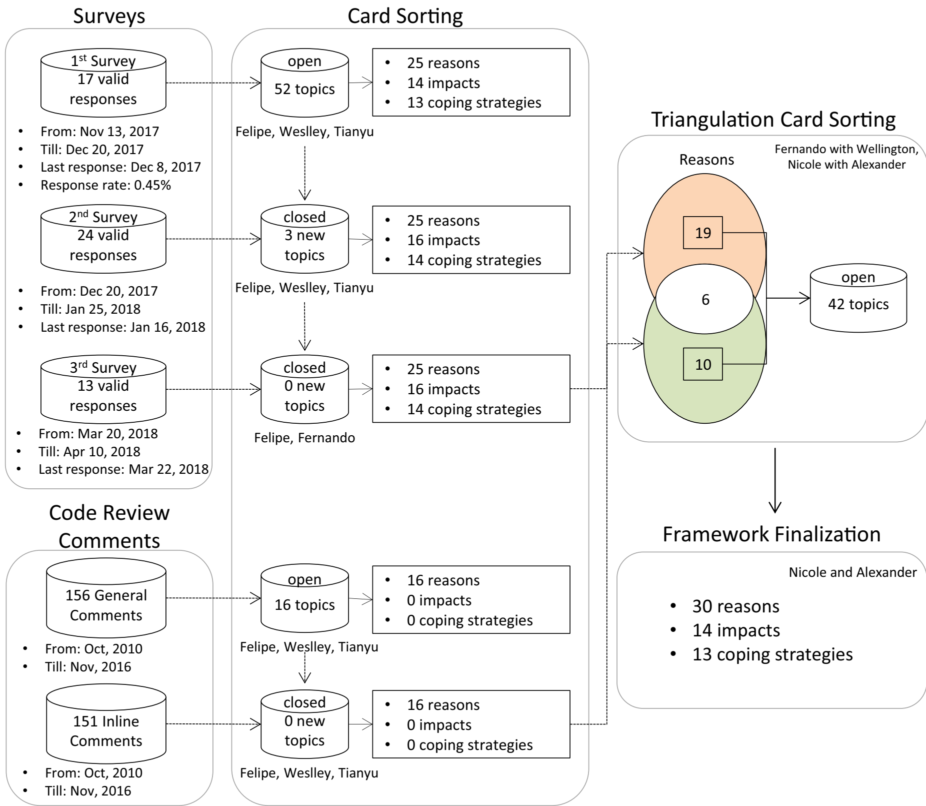

Fig. 2 Implementation of the approach: three survey rounds, general and inline comments, the triangulation, and finalization rounds (Ebert et al. 2019)

or one-time-contributors (Lee et al. 2017). For the second and the third survey rounds, the number of responses are 24 and 13 respectively; the response rate could not be computed.

The open card sorting of the first survey resulted in 52 topics related to the reasons (25), impacts (14) and coping strategies for confusion (13). The closed card sorting of the second survey resulted in three additional topics: two for impacts and one for the coping strategies. Finally, the closed card sorting of the third survey resulted in no new topics. The open card sorting on the general comments resulted in 16 topics related only to the reasons for confusion, i.e., no topics related to the impacts and coping strategies appeared. Then, the closed card sorting on the inline comments resulted in no new topics.

During the triangulation, we verified that what developers said about the reasons for confusion (survey) has a little agreement with what developers did in the code review comments. Only 6 topics were found both among the survey answers and code review comments, 19 topics appeared only in the survey and 10 topics—in the code review comments. Thus, we decided to conduct another card sorting on the divergent 29 topics. This time, since it was an open card sorting, from the cards belonging to divergent topics we identified 42 topics. As the last step, we finalized the framework and obtained a total of 57 topics related to reasons (30), impacts (14), and coping strategies (13). After finalizing the topics we observe that 70% (21/30) of them have cards both from the surveys and from the review

comments. Moreover, the shared topics cover the lion's share of the cards: 94.9% of the survey cards and 90.7% of the code review comments' cards.

As explained above, using axial coding we identified the following dimensions, common for answers to the three **RQs**: **review process** (18 topics): the code review process, including issues that affect the review duration; **artifact** (15 topics): the system prior to change, code change itself and its documentation or the system after change; **developer** (15 topics): topics regarding the person implementing or reviewing the change; **link** (9 topics): the connection between developers and artifacts, e.g., when a developer indicates that they do not understand the code. Examples of topics of different dimensions can be found in Sections 3.2.4, 3.2.5 and 3.2.6.

3.2.2 Analysis of Similarity of the Surveys' Results

First, we verified the similarity of the second and third surveys. Since both were published on FACEBOOK and TWITTER, we expect the values to be similar, i.e., respondents to represent the same population. Using both ANOSIM ($R = -0.0171$ and p-value $= 0.542$) and PERMANOVA (p-value $= 0.975$) we could not observe statistically significant differences between the groups, i.e., the answers can be grouped together. Then, we checked the similarity between the answers to the first survey (ANDROID developers) and the answers to the second and the third surveys taken together. The results of the ANOSIM analysis, $R = 0.126$ and p-value $= 0.01$, showed that the difference between the groups is statistically significant. However, the low R means that the groups are not so different (values closer to 1 mean more of a difference between samples), i.e., the overlap between the surveys is quite high. This observation is confirmed by the outcome of the PERMANOVA test: the p-value $= 0.191$ is above the commonly used threshold of statistical significance (0.05). Based on those results, we conclude that the respondents represent the same population of developers and report the results of all three surveys together.

3.2.3 Demographics of the Survey Respondents

The respondents are experienced *code reviewers*, 80% (38 of 47 respondents that answered questions about demographics) have more than two years of experience reviewing code changes. The experience of our population as *developers*, i.e., authoring code changes, is even higher: 93% (44 respondents) have been developing for more than two years. The number of years of experience as *developers* is higher than the number of years of experience as *reviewers*: this is expected because reviewing tasks are usually assigned only to more experienced individuals (van Wesel et al. 2017). Respondents are active in submitting changes for review, and even more active in reviewing changes: almost 49% (23 developers) submit code reviews several times a week, while for reviewing this percentages reaches 72% (34).

The frequencies with which code change authors and code reviewers experience confusion are summarized in Fig. 3. On the one hand, when reviewing code changes, about 41% (20) of the respondents feel confusion at least half of the time, and only 10% (5) do not feel confusion. On the other hand, when authoring code changes only 12% (6) of the respondents feel confusion at least half of the time, and 35% (17) of the respondents do not feel confusion. Comparing the figures we conclude that confusion when reviewing is very common, and that developers are more often confused when reviewing changes submitted by others as opposed to when authoring the change themselves. We also applied the χ^2 test to check whether experience influences frequency of confusion being experienced. The test was not able to detect differences between more and less experienced developers in terms

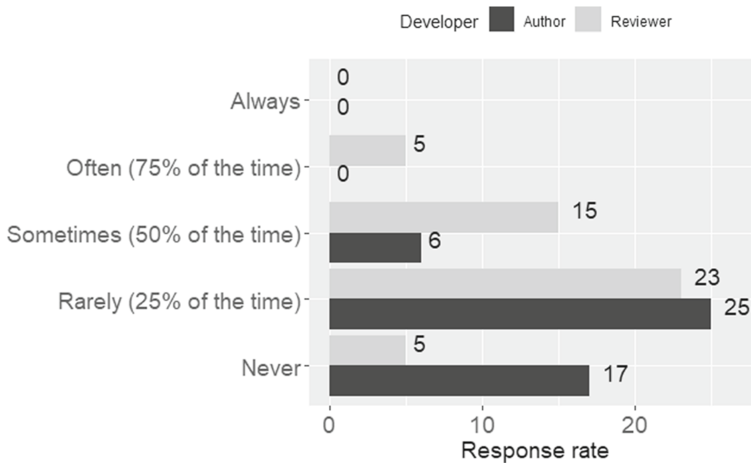

Fig. 3 Frequency of confusion for developers and reviewers

of frequency of confusion being experienced as a developer, nor between more and less experienced reviewers in terms of frequency of confusion being experienced as a reviewer ($p \simeq 0.26$ and 0.09, respectively).

3.2.4 RQ1. What Are the Reasons for Confusion in Code Reviews?

We found 30 reasons for confusion in code review (see Table 2). They are spread over all the dimensions, with the artifact and review process being the most prevalent.

There are seven reasons for confusion related to the code review process. The most common is *organization of work* which comprises reasons such as unclear commit message (e.g., "*when the description of the pull request is not clear*", R50), the status of the change (e.g., " *I'm unsure about the status of your parallel move changes. Is this one ready to be reviewed? [...]*"),[10] or the change addressing multiple issues (e.g., "*change does more than one things*", R31). The second and third reasons most cited are, respectively, confusion about the tools, e.g., "*I don't know why the rebases are causing new CLs*",[11] and the need of the code change, e.g., "*If I understand correctly, this change might not be relevant any more*".[12]

The artifact dimension it is the largest group with 11 topics related to the reasons for confusion. The most popular is the absence of the change rationale, e.g., "*I do not fully understand why the code is being modified*" (R20). Discussion of the solution related to non-functional aspects of the artifact is the second largest topic and it comprises reasons such as poor code readability (e.g., "*Poorly implemented code*" (R43)), and performance (e.g., "*is this true? i can't tell any difference in transfer speed with or without this patch. i still get roughly these numbers from "adb sync" a -B build of bionic: [...]*").[13] The third most frequent reason indicates that developers experience confusion when *unsure about the*

[10] https://android-review.googlesource.com/c/132581
[11] https://android-review.googlesource.com/c/71976
[12] https://android-review.googlesource.com/c/33140
[13] https://android-review.googlesource.com/c/91510

Table 2 The reasons, impacts and coping strategies developers use to deal with confusion; in the parenthesis are the numbers of cards

Reasons 30 topics (507)	Impacts 14 topics (98)	Coping strategies 13 topics (116)
Process		
18 topics (120)		
Organization of work (17)	Delaying (31)	Improved organization of work (5)
Issue tracker, version control (7)	Decreased review quality (11)	
Unnecessary change (6)	Additional discussions (11)	Delaying (2)
Not enough time (3)	Blind approval (8)	Assignment to other reviewers (1)
Dependency between changes (3)	Review rejection (4)	
Code ownership (2)	Increased development effort (4)	Blind approval (1)
Community norms (2)	Assignment to other reviewers (2)	
Artifact		
15 topics (300)		
Missing rationale (66)	Better solution (1)	Small, clear changes (4)
Discussion of the solution: non-functional (49)	Incorrect solution (1)	Improved documentation (4)
Unsure about system behavior (37)		
Lack of documentation (29)		
Discussion of the solution: strategy (29)		
Long, complex change (25)		
Lack of context (19)		
Discussion of the solution: correctness (14)		
Impact of change (11)		
Irreproducible bug (6)		
Lack of tests (5)		
Developer		
15 topics (124)		
Disagreement (18)	Decreased confidence (10)	Information requests (36)
Communicative intention (9)	Abandonment (6)	Off-line discussions (12)
Language issues (3)	Frustration (5)	Providing or accepting suggestions (10)
Propagation of confusion (3)	Anger (2)	
Fatigue (1)	Propagation of confusion (2)	Disagreement resolution (6)
Noisy work environment (1)		
Link		
9 topics (177)		
Lack of familiarity with the existing code (47)		Improved familiarity with the existing code (28)
Lack of programming skills (40)		Testing the change (5)
Lack of understanding of		Improved familiarity with the technology (2)
the problem (21)		
the change (17)		

Table 2 (continued)

Reasons 30 topics (507)	Impacts 14 topics (98)	Coping strategies 13 topics (116)
Lack of familiarity with the technology (14)		
Lack of knowledge about the development process (3)		

system behavior, e.g., "*what is the difference between this path (false == unresolved) and the unresolved path below. [...]*".[14]

Six reasons for confusion are related to the developer dimension. *Disagreement* among the developers is the prevalent topic, e.g., "*[...] If actual change has a big difference from my expectation, I am confused.*" (R11). The second most cited reason is the misunderstanding of the message's intention, e.g., "*Sometimes I don't understand general meaning (need to read several times to understand what person means)*" (R13).

Six reasons are related to the link between the developer and the artifact. The most popular one is the *lack of familiarity with existing code*, e.g., "*Lack of knowledge about the code that's being modified.*" (R37) followed by the *lack of programming skills*, e.g., "*sometimes I'm confused because missing some programming*" (R13), and the *lack of understanding of the problem*, e.g., "*I'm embarrassed to admit it, but I still don't understand this bug.* "[15]

> ***RQ1 Summary - Reasons for confusion:*** We found a total of 30 reasons for confusion. The most prevalent are *missing rationale*, *discussion of the solution: non-functional*, and *lack of familiarity with existing code*. We observe that tools (code review, issue tracker, and version control) and communication issues, such as disagreement or ambiguity in communicative intentions, may also cause confusion during code reviews.

3.2.5 RQ2. What are the Impacts of Confusion in Code Reviews?

The total number of topics related to the impacts of confusion is 14 (see Table 2). They are related to the dimensions of the review process, artifact, and developer. There was no topic related to the link between the developer and the artifact.

We found seven impacts of confusion related to the code review process. *Delaying* the merge decision is the most popular impact, e.g., "*The review takes longer than it should*" (R46). The second and third most cited impact are that confusion makes the code review quality decrease, e.g., "*Well I can't give a high quality code review if I don't understand what I am looking at*" (R5), and an increase in the number of messages exchanged during the discussion, e.g., "*Code reviews take longer as there's additional back and forth*" (R1). One interesting impact of confusion is the blind approval of the code change by the developer, even without understanding it, e.g., "*Blindly approve the change and hope your coworker knows what they're doing (it is clearly the worst; that's how serious bugs end up in production)*" (R16). Confusion may also lead to developers to just reject a code change,

[14] https://android-review.googlesource.com/c/83350

[15] https://android-review.googlesource.com/c/170280

e.g., "*I'm definitely much more likely to reject a 'confusing' code review. Good code, in my experience, is usually not confusing*" (R36).

There are only two impacts of confusion related to the artifact itself. First, the developer may find a better solution because of the confusion, e.g., "*It has not only bad impact but also good impact. Sometimes I can encounter a better solution than my thought*" (R11). Second, the code change might be approved with bugs, as the reviewer is not be able to review it properly due confusion, e.g., "*Sometimes repeated code is committed or even a wrong functionality*" (R24). The *incorrect solution* impact is related to *decrease review quality*, however, the perspective is of the code change containing a bug in production rather than of the reviewing process.

Finally, there are four impacts of confusion related to the developer. The most quoted impact is the decrease of self confidence, either by the author, e.g., "*I can't be confident my change is correct*" (R38), or by the reviewer, e.g., "*I feel less confident about approving it*" (R48). Another impact is the developer giving up, abandoning a code change instead of accounting for the reviewer's comments, e.g., "*other times I just give up*" (R14), or leave the project, e.g., "*dissociated myself a little from the codebase internally*" (R14). We also found emotions being triggered by confusion, such as anger (e.g., "*It pissed me off*", R3) and frustration (e.g., "*Cannot be an effective reviewer—can replace me with a lemur*", R40). And finally, confusion can be contagious, e.g., "*It often causes confusion spreading to other reviewers*" (R12).

RQ2 Summary - Impacts of confusion: We identified 14 different impacts of confusion in code reviews. The most common are *delaying*, *decrease of review quality*, and *additional discussions*. Some developers blindly approve the code change, regardless the correctness of it; other impacts include *frustration*, *abandonment* and *decreased confidence*.

3.2.6 RQ3. How Do Developers Cope with Confusion?

We found 13 topics describing the strategies developers use to deal with confusion in code reviews. Four of them are related to the review process. The most common is to *improve the organization of work*, such as making clearer commit messages, e.g., "*Leave comments on the files with the main changes*" (R50). It is followed by spending more time and *delaying* the code review, e.g., "*I need to spend much more time*" (R13). Assigning other reviewers is also a strategy adopted by developer, e.g., "*Sometimes I completely defer to other reviewers*" (R48). Interestingly, *blind approval* is also a strategy developers use to cope with confusion, i.e., it is not just an impact, e.g., "*assume the best, (of the change)*" (R34).

Two strategies are related to the artifact. Developers make the code change smaller, e.g., "*Also I ask large changes to be broken into smaller*" (R31), and clearer, e.g., "*Try to make the actual code change clear*" (R12). They also improve the documentation by adding code comments, e.g., "*A good description in the commit message describing the bug and the method used to fix the bug is also helpful for reviewers*" (R5).

The dimension with the most quotes is related to the developers themselves. Requesting for information on the code review tool itself is the most quoted among developers, e.g., "*Put comment and ask submitter to explain unclear points*" (R15). Developers also take the discussions off-line, i.e., using other means to reach their peers, e.g., "*schedule meetings*" (R50) or "*ask in person*" (R1). Providing and accepting suggestions is also mentioned as

a good way to cope with confusion. It includes strategies such as being open minded to the comments of their peers, e.g., "*Being open to critical review comments*" (R12), and providing polite criticism, e.g., "*Trying to be 'a nice person'. Gently criticizing the code*" (R3). The use of criticism by developers in code reviews was also found by Ebert et al. (2018), but their study focused on the intention of questions in code reviews. Disagreement resolution is also a good strategy to cope with confusion, e.g., "*I try to explain the reasoning behind the decisions/assumptions I made*" (R31).

Regarding the link between the developer and the artifact, there are three strategies developers use to cope with confusion. Firstly, to study the code or the documentation, e.g., "*It forces me to dig deeper and learn more about the code module to make sure that my understanding is correct (or wrong)*" (R12), and "*Read requirements documentation*" (R24). Secondly, to test the code change, e.g., "*play with the code*" (R9). Finally, developers also use external sources to improve their knowledge about the technology, e.g., "*Sometimes further research on the web [...]*" (R25).

RQ3 Summary - Coping strategies: We have identified 13 coping strategies. Common strategies include *information requests*, *improved familiarity with the existing code*, and *off-line discussions*.

3.3 Threats to Validity

As any empirical study, our work is subject to several threats of validity. We identified three kinds of threats to its validity: construct, internal, and external, all of which are discussed below.

Construct validity is related to the relation between the concept being studied and its operationalisation. In particular, it is related to the risk of respondents misinterpreting the survey questions. To reduce this risk we included our own definition of confusion and requested the respondents to confirm that they understood it. For the same reason, we always anchored the frequency questions and adhered to well-known survey design recommendations (Groves et al. 2009; Kitchenham and Pfleeger 2008; Singer and Vinson 2002; Steele and Aronson 1995).

Internal validity pertains to inferring conclusions from the data collected. The card sorting adopted in our work is inherently subjective because of the necessity to interpret text. To reduce subjectivity every card sorting step has been carried out by several researchers. Moreover, to assure the completeness of the topics related to the reasons, impacts and confusion coping strategies we conducted several survey iterations until the data saturation has been achieved, and augmented the insights from the surveys with those from the code review comments.

External validity is related to the generalizability of the conclusions beyond the specific context of the study. Our first survey targeted only a single project: ANDROID. However, the second and the third ones targeted a general software developer population. Statistical analysis has not revealed any differences between the respondents of the different surveys suggesting that the answers obtained are likely to reflect opinions of the code review participants, in general. To complement the surveys we consider 307 code review comments from GERRIT. While the functionality of GERRIT is typical for most modern code review tools, developers using more advanced code review tools do not necessarily experience confusion

in the same way. For instance, COLLABORATOR[16] supports custom templates and checklists, that if properly configured might require the change authors to indicate rationale of their change, reducing the importance of "missing rationale" from Table 2.

4 Which Reasons for Confusion are Most Frequent? A Preliminary Study

The long-term goal of our research is to help developers combat confusion in code reviews. The main contribution of the study discussed in Section 3 is a framework for confusion in code reviews, presented in Table 2, including 30 reasons, 14 impacts, and 13 coping strategies. The difference in numbers between the reasons, on the one hand, and impacts and coping strategies, on the other hand, suggested a gap between the way confusion is experienced and the ways it impacts software development and is addressed. However, many of these reasons for confusion have been extensively studied in the scientific literature (Bacchelli and Bird 2013; Tao et al. 2012; Kononenko et al. 2015). Hence, we decided to complement the results in Table 2 by investigating the solutions proposed by literature for the most frequent reasons for confusion, as well as the impacts of those reasons. As a preliminary step towards this goal, we survey developers to gauge the frequency with which the 30 reasons for confusion from our framework typically occur in practice. The results of the survey allow us to prioritize the reasons for confusion, i.e. to identify the reasons for confusion to focus on in the literature review discussed in Section 5.

In the remainder of this section, we present the aforementioned survey, involving 62 developers. More specifically, we aim at answering the following research question:

- **RQ4.** Which reasons for confusion do developers perceive as occurring most frequently?

We describe the methodology in Section 4.1. Section 4.2 presents the results, and threats to the validity are discussed in Section 4.3.

4.1 Methodology

We start by discussing the design of our survey (Section 4.1.1). Then, we present the participants selection (Section 4.1.2). Finally, we discuss the data analysis process we used (Section 4.1.3).

4.1.1 Survey Design

We designed a survey to ask code reviewers how often they experience each of the 30 reasons for confusion included in our framework (see Table 2). We design the survey in line with established best practices (Groves et al. 2009; Kitchenham and Pfleeger 2008; Singer and Vinson 2002; Steele and Aronson 1995). We start by explaining the goal of this survey and our research goals, disclose the sponsors of our research and inform that the information provided will be treated in a confidential way. We also inform the respondents about the estimated time to finish the survey, and then, obtain the respondents' informed consent.

The questions of the survey are presented in Table 3. It starts with the same definition of confusion used in the former study and presented in Section 2. Then, we ask the respondents to confirm their understanding of this definition (Q1). Next, Q2–Q29 ask how often do

[16] https://smartbear.com/product/collaborator/overview/

the respondents feel confused when **reviewing** changes due to reasons for confusion from Table 2, i.e., we focused on code reviewers. Frequency is measured on a Likert scale: *not at all*, *less than once a month*, *once a month*, *once a week*, *once a day*, and *more than once a day*. For the sake of readability, we split the 30 questions corresponding to reasons for confusion from Table 2 according to the four dimensions defined in Section 3.2: review process, artifact, developer, and link between the developer and the artifact. We do not include two reasons for confusion in this survey since they are only related to the code change author, and not the reviewer, i.e., the reasons *code ownership* and *community norms*.

Before deploying the survey, we discussed it with other software engineering researchers and clarified it when necessary: e.g., we replaced "unnecessary change" by "a change which is unnecessary for the project".

4.1.2 Participants

As the target population, we considered developers who reviewed code changes in reviews. We sent the survey to two different groups. The first group comprises 33 developers who answered the survey from our first study (cf. Section 3.1.1) and indicated that they would like to be informed about the results of that study. Within ten days after the first mail we sent a reminder. The email message included a personalized salutation, a brief discussion of the results of our first study (Ebert et al. 2019), an explanation about this new study, and the link for the new survey. The second group consists of developers recruited via social media: we published the survey on FACEBOOK and TWITTER and asked developers to answer it. We left the survey open until we received no more responses for two weeks (cf. surveys conducted by German et al. (2018) and Kononenko et al. (2018)).

4.1.3 Data Analysis

Similarly to the analysis of Section 3.1.1, we have a survey with two different groups of respondents. Thus, a priori it is not clear if the responses can be seen as representing the same population. We used the same statistical methods, ANOSIM (Clarke 1993) and PERMANOVA (Anderson 2001; McArdle and Anderson 2001), to perform the similarity check. Again, if the groups of respondents can be said to be similar, we can consider them as representing the same population, and then merge the responses. Otherwise, we would treat the groups separately.

To further analyze the responses of our survey, we applied the Scott-Knott Effect Size Difference (ESD) test (Tantithamthavorn et al. 2017) to group the 28 reasons for confusion into statistically distinct ranks according to their Likert scores in terms of frequency. Scott-Knott ESD is a variant of Scott-Knott test (Scott and Knott 1974), in which there is no normality assumption of the data. The Scott-Knott ESD test merges any two statistically distinct groups that have a negligible effect size into one group. Scott-Knott ESD has been successfully applied in the software engineering context Calefato et al. (2019), Catolino and Ferrucci (2019), and Tantithamthavorn et al. (2017).

4.2 Results

In this section, we present the results of our survey. We start by explaining how we conducted the survey (Section 4.2.1). Then we present the results of the similarity analysis (Section 4.2.2). Finally, we present the results of **RQ4** using Scott-Knott ESD test (Tantithamthavorn et al. 2017) (Section 4.2.3).

Table 3 Survey questions

Electronic Consent	
0.	Please select your choice below. Selecting the “yes” option below indicates that: i) you have read and understood the above information, ii) you voluntarily agree to participate, and iii) you are at least 18 years old. If you do not wish to participate in the research study, please decline participation by selecting “No”.
Definition of Confusion	
	The remainder of this survey is dedicated to “confusion”. We do not make a distinction between lack of knowledge, confusion, or uncertainty. For simplicity reasons, we use the “confusion” to refer to all these terms.
1.	By clicking “next” you declare that you understand the meaning of confusion on this survey.
Topics related to the code review process	
How often do you feel confused when reviewing code changes due to:	
2.	Organization of work (e.g., an unclear commit message, the status of the code review, a change addressing multiple issues)
3.	Any development related tool (e.g., issue tracker, code review or version control system)
4.	A change which is unnecessary for the project
5.	Not having enough time
6.	Dependency between different code changes
Topics related to the code change	
How often do you feel confused when reviewing code changes due to:	
7.	Missing code change rationale (e.g., in the commit message, or in code comments)
8	Discussion of the solution related to non-functional aspects (e.g., maintainability, performance, or poor code readability)
9.	Lack of understanding of the system behavior
10.	Lack of documentation
11.	Disagreement with the strategy proposed in the code change
12.	Long or complex code change
13.	Lack of context
14.	Lack of understanding of the correctness of the code change
15.	The impact of code change
16.	Lack of understanding of how to reproduce the bug
17.	Lack of tests
Topics related to the developer	
How often do you feel confused when reviewing code changes due to:	
18.	Disagreement with the peers
19.	Lack of understanding of the intention of peers’ comments
20.	Language issues in the communication (e.g., due to poor mastery of English)
21.	Propagation of confusion (spreading confusion among the peers)
22.	Fatigue
23.	Noisy work environment
Topics related to the link between the developer and the artifact	
How often do you feel confused when reviewing code changes due to:	
24.	Lack of familiarity with the existing code
25.	Lack of programming skills

Table 3 (continued)

26.	Lack of understanding of the problem
27.	Lack of understanding of the code change
28.	Lack of familiarity with the technology
29.	Lack knowledge about the development or code review process
Results	
30.	Would you like to be informed about the outcome of this study and potential publications? Please leave a contact email address.
31.	Please add additional comments below.

4.2.1 Implementation of the Survey

The first emails were sent on the July 15th, 2019. Among the 33 emails sent for the first group, four emails have bounced. We received 13 responses, i.e., a response rate of 44%. Seven developers answered the survey in the first day, while the remaining six developers answered our survey after the reminder. The survey was published on FACEBOOK and TWITTER on the same day we sent the emails. The response rate could not be computed for this group. We closed the survey after two weeks with no new response in August 21st, i.e., the last response we received was on August 7th. We received 50 responses from the social media but one respondent did not indicate their consent, i.e., we have obtained 49 valid responses.

4.2.2 Analysis of Similarity of the Surveys' Results

The results of the similarity check with ANOSIM, $R = -0.06928$ and p-value $= 0.792$, did not show any statistically significant differences between the two groups. The results for the PERMANOVA method, p-value $= 0.506$, also did not show any statistically significant differences. Based on those results, we conclude that the two groups of respondents represent the same population of developers, and subsequently we merged their responses and report the results pertaining to the combined group. Hence, we have a total of 62 valid responses considered in our analysis.

4.2.3 RQ4. Which Reasons for Confusion do Developers Perceive as Occurring Most Frequently?

The results of the frequency of reasons for confusion are presented in Table 4. Since our goal is to define the most frequent reasons for confusion, we need a fair measure to order them. One possibility is to consider as more frequent the reasons that more developers classified as "More than once a day", normalized by the overall number of classifications for each reason. A similar approach has been employed by previous work (Begel and Zimmermann 2014). However, in our case, every reason has been classified the same number of times, unlike previous work. Furthermore, we do not think that a reason classified just once as "More than once a day" but not as "Once a day" is really more frequent than one that has not been classified as "More than once a day" but received, e.g., ten classifications as "Once a day".

Thus, we used the Scott-Knott Effect Size Difference (ESD) test (Tantithamthavorn et al. 2017) to group reasons with similar frequencies. Table 4 shows the 28 reasons for confusion organized into seven different groups. The first group contains the most frequent five reasons for confusion. Additionally, Table 4 also shows the mean and median Likert scores for the 28 reasons for confusion in terms of frequency, and their respective dimensions.

We can see that the most frequent reasons for confusion are either related to the artifact (i.e., the code change itself) or to the review process. They are: *long or complex code change*, *organization of work* (e.g., an unclear commit message, the status of the code review, a change addressing multiple issues), *dependency between different code changes*, *lack of documentation*, and *missing code change rationale*. The least frequent reasons for confusion accordingly to developers are related to developers themselves and to the link between developer and artifact: *propagation of confusion*, *language issues in the communication*, *lack of programming skills*, and *lack of knowledge about the development or code review process*.

We conjecture that the most frequent reasons for confusion are top ranked because they are related to processing a large amount of information which is spread across different places. For example, *long or complex code change* can be related to many different files (or many places in the same file); *organization of work* can refer to the same code change addressing multiple issues; and *dependency between different code changes* is related to different changes. As for the least frequent reasons for confusion, we conjecture that they are related to self admission of confusion by developers themselves as they pertain to the dimensions related to the developer (and the link between developers and the artifact), such as *lack of knowledge about the development or code review process*, *lack of programming skills*, *language issues in the communication*, and *propagation of confusion*.

RQ4 Summary - Most frequent reasons for confusion: The most frequent reasons for confusion experienced by developers are related to the artifact and the review process. According to the rank based on the developers' answer, the top five reasons are: *long or complex code change*, *organization of work*, *dependency between different code changes*, *lack of documentation*, and *missing code change rationale*. The least frequent reasons for confusion are related to the developer and the link between the developer and the artifact.

4.3 Threats to Validity

Similarly to our first study (see Section 3), this survey is subject to three kind of threats of validity:

Internal validity relates to how conclusions are inferred from the data analyzed. This threat in our survey relates to how developers recollect past events, i.e., when and how they feel confused in code reviews. We acknowledge that the frequency of confusion might also depend on how often the survey respondents perform code review activities (e.g., on a daily basis, weekly, and so on). However, we believe that there is no reason for assuming that some reasons for confusion might be remembered more easily than others, which mitigate such a threat.

Construct validity relates the concept being studied and its operationalisation, i.e., the degree to which we actually measure what we intend to. One threat to the validity of this study is that survey respondents can misinterpret the questions. We followed the same approach presented in Section 3.3 to reduce this threat. Specifically, we presented our definition of confusion and requested the respondents to confirm whether they understood it.

Table 4 The 28 reasons for confusion ranked according to the Scott-Knott Effect Size Difference test in terms of frequency, and the mean and median Likert scores

Group	Reason for confusion	Mean	Median	Dimension
1	Long or complex code change	2.40	2	Artifact
	Organization of work	2.33	2	Review Process
	Dependency between different code changes	2.24	2	Review Process
	Lack of documentation	2.20	2	Artifact
	Missing code change rationale	2.19	2	Artifact
2	Lack of tests	2.14	2	Artifact
	Lack of familiarity with the existing code	2.11	2	Link
	Lack of understanding of the system behavior	2.08	2	Artifact
	Not having enough time	2.03	2	Review Process
	Disagreement with the strategy proposed in the code change	2.01	2	Artifact
	The impact of code change	2.00	2	Artifact
	Lack of understanding of the correctness of the code change	1.96	2	Artifact
3	Lack of context	1.93	2	Artifact
	Discussion of the solution related to non-functional aspects	1.83	2	Artifact
	Lack of understanding of the code change	1.82	2	Link
	Fatigue	1.80	2	Developer
	Lack of understanding of the intention of peers' comments	1.77	2	Developer
	Lack of understanding of the problem	1.77	2	Link
4	Lack of understanding of how to reproduce the bug	1.64	1	Artifact
	Disagreement with the peers	1.61	1	Developer
	A change which is unnecessary for the project	1.59	2	Review Process
5	Noisy work environment	1.41	1	Developer
	Lack of familiarity with the technology	1.38	1	Link
	Any development related tool	1.22	1	Review Process
6	Propagation of confusion	1.08	1	Developer
	Language issues in the communication	1.06	0.5	Developer
	Lack of programming skills	0.96	1	Link
7	Lack knowledge about the development or code review process	0.79	0	Link

Additionally, we designed our survey based on well-known recommendations (Groves et al. 2009; Kitchenham and Pfleeger 2008; Singer and Vinson 2002; Steele and Aronson 1995). Another threat to construct validity pertains to the measure we employed to rank the reasons for confusion in terms of their frequency. In order to reduce such threat, we used a specific test, Scott-Knott ESD test (Tantithamthavorn et al. 2017), for measuring, comparing, and clustering the frequency of the responses for the reasons for confusion. One last threat to construct validity is the use of a survey itself, since it relies on developers' perceptions. Our reason for adopting this approach is the possibility to scale it up, since we can gather information about all the reasons for confusion described in Section 3.2.4 from many developers.

External validity is related to the generalizability of the conclusions of the study. The first group of population of our survey targeted ANDROID developers. The second group

targeted a more general software developer population. Thus, we used statistical analysis to verify similarity between these different populations. The results suggests no difference between the first and the second group, indicating that the responses can be treated as one group.

Another external threat is related to volunteer bias, i.e., when the subjects who volunteered to participate in a research project might differ in some ways from the target population. We tried to reduce such a threat by recruiting participants both by personal invitations and via social media. Furthermore, since the likelihood of volunteer bias increases with the refusal increases, we ensured anonymity and confidentiality of volunteers in order to try to increase participation, and thus, to decrease volunteer bias.

5 A Systematic Mapping Study of Solutions and Impacts of Confusion in Code Reviews

The main contribution of the preliminary study, as reported in the previous section, is an ordered list of the most frequent reasons for confusion according to developers (cf. Table 4). As mentioned before, many of the factors we have identified as possible reasons for confusion have been studied in software engineering literature (Bacchelli and Bird 2013; Tao et al. 2012; Kononenko et al. 2015). To contextualize our findings, we perform a literature review. Based on the results of our survey presented in Section 4, we selected the top five most frequently occurring reasons for confusion, as a starting point to conduct a systematic mapping study of the scientific literature. Our goal is to identify their impacts on code reviews, beyond confusion, and the solutions and mitigation strategies researchers have proposed to cope with them. Such strategies might be beneficial for developers facing confusion and complement the currently employed coping mechanisms.

As such, we designed and ran a systematic mapping study aims to answer the following research questions:

- ***RQ5.*** *What are the solutions proposed by researchers for the most frequent reasons for confusion?*
- ***RQ6.*** *What relationships has previous research established between the most frequent reasons for confusion and their impacts?*

The results of this mapping study allow us to complement the framework presented in Section 3 in three ways:

i. by identifying new coping strategies to address confusion;
ii. by establishing links between the reasons for confusion and the coping strategies proposed by researchers and employed by developers, as identified by previous studies; and
iii. by determining how the reasons for confusion and impacts of confusion are connected.

Section 5.1 describes the methodology of the systematic mapping study. In Section 5.2, we present the results this study, and threats to the validity are discussed in Section 5.3.

5.1 Methodology

The **goal** of the mapping study is to identify, classify, and understand what are the solutions proposed by the research community to *the most frequent reasons for confusion* in

code reviews (**RQ5**), according to the survey described in Section 4. Furthermore, we aim to identify the link between the most frequent reasons of confusion and their impacts in the code review process (**RQ6**). Based on the results of **RQ4**, we chose the most frequent reasons for confusion on the mapping study. Then, we conduct the mapping study, following the guidelines by Petersen et al. (2008) and Petersen et al. (2015).

To perform the systematic mapping study, we used Parsifal,[17] an online tool supporting systematic literature reviews and mapping studies within the context of software engineering. It provides support for all the phases of the mapping studies: planning, conducting, and reporting the mapping.

Kitchenham and Charters (2007) developed PICO (Population, Intervention, Comparison, and Outcomes): a guideline to identify keywords and formulate search strings from research questions in systematic literature reviews. The guidelines of Petersen et al. (2015) suggest that only P (population) and I (intervention) should be used for systematic mapping studies. In our context, the *population* are code reviewers, and the *intervention* are the most frequent reasons for confusion. Due to the large number of reasons for confusion in our framework (30), on the one hand, and the estimated effort required for the mapping study, on the other hand, we consider the most frequent reasons, i.e., the five topics from the first group in Table 4:

- **Reason #1:** Long or complex code change;
- **Reason #2:** Organization of work (e.g., an unclear commit message, the status of the code review, or a change addressing multiple issues);
- **Reason #3:** Dependency between different code changes;
- **Reason #4:** Lack of documentation;
- **Reason #5:** Missing code change rationale (e.g., in the commit message, or in code comments).

Since we have five different reasons for confusion, we created five different search strings to simplify the process. Firstly, we defined the string related to code reviews by including several synonyms to it: *code review OR code inspection OR ((peer code review OR peer review) AND software*. After a few queries, we decided to add the term *software* as a way to exclude secondary studies of different areas, since the string *peer review* is also related to systematic literature reviews. Then, we combined this string with terms related to the specific reason for confusion, e.g., the reason *missing code change rationale* resulted in the search string *((lack OR missing OR omission OR absence OR absent OR unclear OR "not clear" OR bad OR misunderstanding) AND (documentation OR comment OR license))*. We did this for each one of the reasons. Tables 5 show all five search strings:

We search for articles in IEEE XPLORE,[18] ACM DL,[19] SCOPUS,[20] and SPRINGERLINK.[21] All the searches were conducted on September 3, 2019. The searches also include plural forms of the words. For the libraries ACM DL, SCOPUS and SPRINGERLINK, we could group all five search strings into one to run it once. For the IEEE XPLORE, there is a size limit of the string, hence, we needed to run eight search strings (as the search string related to the organization of work needed to be split into three). SPRINGERLINK allow us

[17] https://parsif.al
[18] https://ieeexplore.ieee.org
[19] https://dl.acm.org
[20] https://www.scopus.com
[21] https://link.springer.com

Table 5 The search strings for all the five reasons for confusion

	Long or complex code change
	(“code review” OR “code inspection” OR ((“peer code review” OR “peer review”) AND software))
#1	AND
	((long OR large OR huge OR big OR complex OR decompose OR composite OR cumbersome OR tricky OR intricate OR complicate OR tangled) AND (“code change” OR changeset OR commit OR “patch set” OR patch OR “pull request”))
	Organization of work (e.g., an unclear commit message, the status of the code review, or a change addressing multiple issues
	(“code review” OR “code inspection” OR ((“peer code review” OR “peer review”) AND software))
	AND
	(
#2	((lack OR missing OR omission OR absence OR absent OR unclear OR “not clear” OR bad OR misunderstanding) AND (commit OR description OR details))
	OR
	((status OR rejected OR accepted OR parallel) AND (“code change” OR “changeset” OR “commit” OR “patch set” OR “patch” OR “pull request”))
	OR
	((mixed OR tangential OR multiple OR composite) AND (“code change” OR “changeset” OR “commit” OR “patch set” OR “patch” OR “pull request”))
	)
	Dependency between different code changes
	(“code review” OR “code inspection” OR ((“peer code review” OR “peer review”) AND software))
#3	AND
	((dependency OR dependence OR upstream OR depends OR dependent OR parallel OR concurrent) AND (“code change” OR “changeset” OR “commit” OR “patch set” OR “patch” OR “pull request”))
	Lack of documentation
	(“code review” OR “code inspection” OR ((“peer code review” OR “peer review”) AND software))
#4	AND
	((lack OR missing OR omission OR absence OR absent OR unclear OR “not clear” OR bad OR misunderstanding) AND (documentation OR comment OR license))
	Missing code change rationale
	(“code review” OR “code inspection” OR ((“peer code review” OR “peer review”) AND software))
#5	AND
	((lack OR missing OR omission OR absence OR absent OR unclear OR “not clear” OR bad OR misunderstanding) AND (rationale OR reason OR goal OR purpose OR intention OR motivation))

to filter the articles by discipline, e.g., only computer science related articles. However, we decided not to do so because we wanted to include as many scientific papers as possible

Table 6 Number of articles per library

Digital library	Search results
IEEE Xplore	100
ACM	149
Scopus	159
SpringerLink	19
Total	427

during this step and could not trust the disciplines as recorded by SPRINGERLINK. Additionally, in the ACM DL query we used the ACM DL *Guide to Computing Literature* option, which is the "*most comprehensive bibliographic database focused exclusively on the field of computing*" and it "*includes all of the content from The* ACM DL *Full-Text Collection along with citations, and links where possible, to all other publishers in computing*". Table 6 shows the number of articles returned by each library.

In all digital libraries, except for SPRINGERLINK, the search was conducted on the title, abstract, and keywords. Since SPRINGERLINK does not allow one to restrict the search to title, abstract and keywords only, we have initially performed a full-text search. However, the full-text search retrieved 30,128 articles. Thus, we created a script to query the *html* pages of each of the 30,128 articles to identify the title, abstract, and keywords. Then, we conducted another search round on those fields only. This step resulted in 19 articles. Next, the 427 identified articles were reviewed based on the following criteria:

- Inclusion criteria:
  - Articles available in full-text;
  - Articles discussing code reviews;
  - Articles subject to peer-review.
- Exclusion criteria:
  - Books, chapters, proceedings, and gray literature;
  - Duplicate articles;
  - Articles not in the field of software engineering;
  - Articles not written in English;
  - Secondary studies (e.g., systematic literature reviews).

The first author started with applying the exclusion criteria by removing the duplicate articles with the aid of the Parsifal tool. In total, we found 155 duplicated articles. Next, the same author applied the remaining exclusion criteria and removed 95 additional articles. To determine whether the article belonged to the field of software engineering, was written in English, or constituted a secondary study, titles and abstracts have been used. In order to diminish research bias, the 95 articles excluded at this stage were reviewed and confirmed by the remaining authors. Hence, by applying the exclusion criteria 250 (= 155 - 95) articles have been removed, leaving 177 (= 427 - 250). Then, the first author verified the inclusion criteria on the remaining articles: 49 of them did not pass the inclusion criteria, leaving 128 (= 177 - 49) articles for the last step, the full-text reading. Once again, the remaining authors reviewed and confirmed those excluded by the inclusion criteria. For the full-text reading step, we looked for any of the five reasons for confusion being mentioned in the articles. We split the 128 articles among the four authors in a way that each paper was reviewed by two authors, i.e., each author reviewed 64 papers. All the disagreements were resolved with

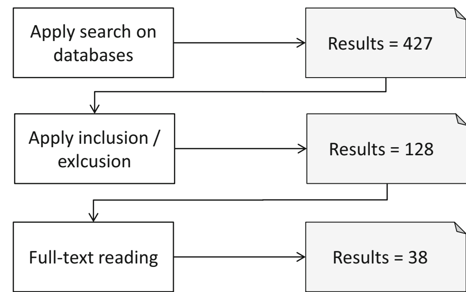

Fig. 4 Number of included articles during the study selection process

online meetings between the authors. Finally, a total of 38 articles have been identified as discussing at least one of the five reasons. The number of included and excluded articles is shown in Fig. 4.

We developed a simple template to extract data from the articles, as shown in Table 7. Each data extraction field has a data item and a value. The extraction was performed by each author during the selection phase. The items ID, title, publication year, and venue were extracted automatically by the Parsifal tool. The remaining items were extracted manually by the authors.

5.2 Results

In this section, we present the results of the our systematic mapping study, which aimed at answering **RQ5** and **RQ6**. Firstly, we provide some general data about the articles selected by the mapping study (Section 5.2.1). Then we discuss the results of **RQ5** (Section 5.2.2) and **RQ6** (Section 5.2.3), respectively.

5.2.1 General Information About the Selected Articles

In Fig. 5 we show the distribution of the 38 articles per year and kind of venue, respectively. We can observe a trend showing an increase of studies related to the most frequent reasons for confusion in code reviews (the size and the color gradient of the circles increases with the number of articles). The data for 2019 is incomplete because the study only considered articles published until September.

Table 7 Data extraction form

Data item	Value
ID	Bibtex ID
Title	Article's title
Publication year	Calendar year
Venue	Name of publication venue
Reasons for confusion	Any of the five reasons for confusion
Solutions for the reasons	Any solutions proposed by the article for the reasons
Relationships between reasons and impacts	Any relations the article established between the reasons for confusion and their impacts?

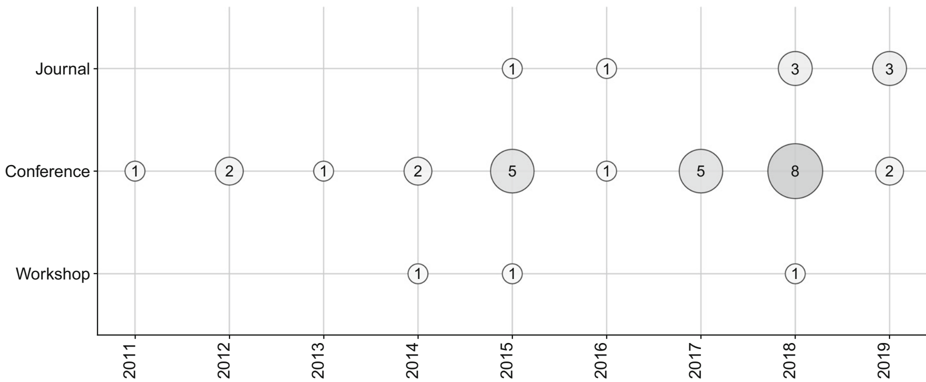

Fig. 5 Distribution of the articles per year according the kind of venue. The data for 2019 is incomplete

We also see that the papers investigating the reasons for confusion cover a broad spectrum of venues including journals (e.g., TSE, EMSE, and JSS), magazines (e.g., IEEE Software), conferences (e.g., ICSE, SANER, MSR, FSE, and ICSME), workshops (e.g., CSD, and MUD). Moreover, we see that these reasons have been discussed at broad-spectrum venues targeting the entire domain of software engineering (e.g., ICSE, APSEC, and FSE), focused events targeting specific activities within software engineering such as maintenance (e.g., ICSME, and SANER), and those dedicated to specific techniques used to analyze software data (e.g., MSR, MUD, and PROMISE). Table 8 provides the complete list of the 38 articles resulting of the mapping study, grouping them by venue.

Table 8 Articles included in the literature study

Venue	Articles
ICSE	Gousios et al. (2014), Huang et al. (2018b), and Barnett et al. (2015)
	Zhang et al. (2015), Sadowski et al. (2018), and Rigby and Storey (2011)
SANER	Zhang et al. (2012), Norikane et al. (2017), and Baysal et al. (2013)
FSE	Tao et al. (2012), Huang et al. (2018a), and Bosu et al. (2014)
APSEC	Wang et al. (2017), An et al. (2018), and Mohamed et al. (2018)
EMSE	Baum et al. (2019) and Baysal et al. (2016)
ICSME	Kononenko et al. (2015) and Baum et al. (2017)
MSR	Tao and Kim (2015) and Hellendoorn et al. (2015)
Others	Baum et al. (2016), Luna Freire et al. (2018), and MacLeod et al. (2018)
	Thompson and Wagner (2017), Zanaty et al. (2018), and Guo et al. (2019)
	Norikane et al. (2018), Kovalenko et al. (2018), and Begel and Vrzakova (2018)
	Pascarella et al. (2019), Guo and Song (2017), and Mishra and Sureka (2014)
	Konopka and Navrat (2015), Izquierdo-Cortazar et al. (2017), and Faragó (2015)
	Yang et al. (2017) and Gerede and Mazan (2018)

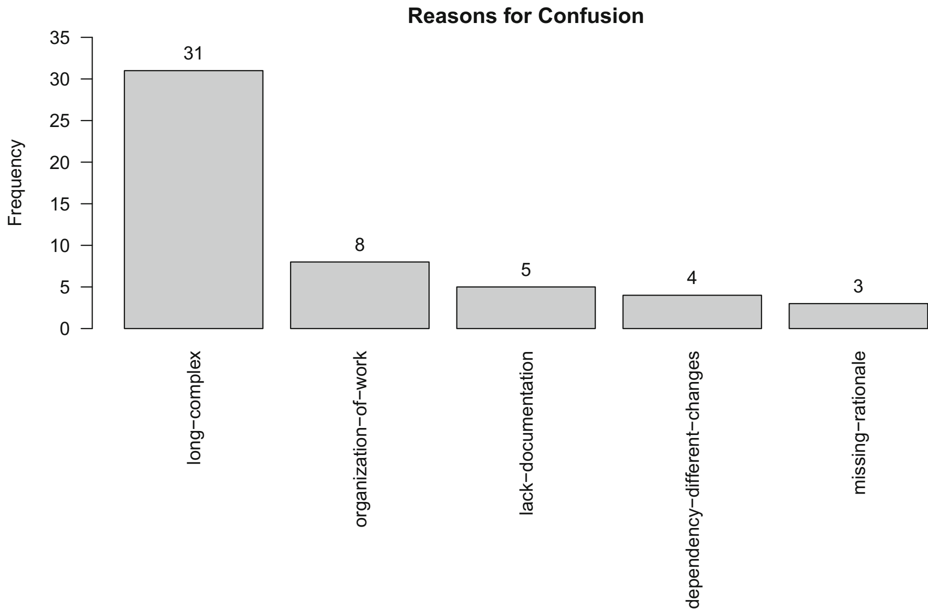

Fig. 6 Number of articles that mentioned each of reason for confusion

5.2.2 RQ5. What are the Solutions Proposed by Researchers for the Most Frequent Reasons for Confusion in Code Reviews?

In Fig. 6, we present the number of articles which address any of the five reasons for confusion. The most common reason is *long or complex code change* with almost all, i.e., a total of 31, articles discussing it. The remaining reasons for confusion were addressed by a much lower number of articles: *organization of work* with eight, *lack of documentation* with five, *dependency between different code changes* with four, and *missing code change rationale* with 3.

In the remainder of this section, we discuss the solutions found in the scientific literature for each one of the five reasons for confusion. It is worth noting that not all articles presented solutions for the reasons for confusion they address.

Long or complex code change: We found a total of five solutions for this reason for confusion proposed by eight different articles in the literature:

1. *Make the change short and simple:* This is the most commonly repeated advice to deal with code changes which are long or complex (Gousios et al. 2014; MacLeod et al. 2018; Sadowski et al. 2018). In fact, GERRIT, a popular code review system, has an option “Show Change Sizes As Colored Bars”: when this option is enabled, the size of the bar indicates the number of changed lines.
2. *Make use of salient files:* Not all files affected by a complex change are equally important and automatic identification of the most important files might reduce the reviewers’ effort. Pascarella et al. (2019) propose an automatic just-in-time identification of defective files in a complex change, while the work of Huang et al. (2018a) introduces the notion of “salient classes”, i.e., the most important class in and the

main reason for the code change, and builds a classification model to automatically identify them.

3. *Improve code review tools:* Code review tools could be expanded to provide functionality that is already present in modern IDEs, such as jumping to definition of an identifier, finding a reference, or exploring a caller/callee tree (Tao et al. 2012).
4. *Make the use of "super reviews":* To allocate the task of reviewing long or complex code changes to the most experienced developers in the team (Kononenko et al. 2015).
5. *Ordering the changes within the code change:* Another way to support developers reviewing long or complex changes is to provide a suggested order of the code change parts in order to reduce the overall cognitive load (Baum et al. 2017).

Organization of work: This is the second most often discussed reason for confusion in the literature. It is a broad topic that gathers different situations related to how work is organized and conducted in a software development project. Even though, we only found two solutions proposed by seven articles in the literature for different aspects of organization of work that may lead to confusion, described below:

1. *Describe the code change:* A reviewer may have a hard time attempting to understand an unclear commit message. It may be unclear for a number of reasons: because it is too short, because it does not include rationale, or because it is poorly written. One of the confusing aspects of the organization of work is lack of clarity in the commit message. To address this problem (MacLeod et al. 2018) stress the importance of describing code changes in an informative way, particularly emphasizing the motivation for the change and the tests associated with it.
2. *Decompose composite code changes:* This is also a common solution proposed for situations when confusion is related to how the code change is organized, i.e., changes addressing multiple issues. Several tools have been proposed to automatically split composite code changes in different changes (Luna Freire et al. 2018; Guo et al. 2019; Barnett et al. 2015; Guo and Song 2017; Tao and Kim 2015; Konopka and Navrat 2015), e.g., a change that implements a new functionality and fixes a bug is split into two changes, one for the new functionality and one for the bug fix.

Lack of documentation: From the five articles discussing this reason for confusion, three of them proposed two solutions for it:

1. *Document well the change:* Developers should properly describe their changes and ensure that all decisions made during the implementation and review are also well-documented (MacLeod et al. 2018).
2. *Support for the placement of code comments:* Code review tools could be expanded to assist developers by suggesting for appropriate locations to place comments in the source code. Huang et al. (2018b) proposes such approach to help developers to decide where to add code comments in the source code by analyzing code context information. Gousios et al. (2014) also suggested that code review tools should provide automated improvement of documentation.

Dependency between different code changes: We identified three solutions to address this reason for confusion on three articles, the third most frequently mentioned in our survey:

1. *Cluster related code changes:* Clustering code changes which are related to each other is a simple solution, however, developers need to be careful to avoid submitting

different issues in the same code change (MacLeod et al. 2018). This is the trade-off between clustering changes and making them composite.

2. *Create tools to summarize similar code changes:* Code review tools could be expanded to find similar changes and detect potential mistakes (based on previous changes) to support reviewers in understanding the impact of related changes. Zhang et al. (2015) developed a tool that summarizes similar code changes and detects potential mistakes to support reviewers' understanding of the impact of related changes.
3. *Use commit-then-review:* In order to avoid longer cycle times when there are dependencies between different code changes so that one has to be committed before another can be started, Baum et al. (2016) suggests to use commit-the-review process, instead of review-then-commit.

Missing code change rationale: This reason for confusion was addressed in three different papers. From those papers, only one proposes a solution for the absence of rationale:

1. *Provide the motivation for the code change:* This is the most basic solution to solve confusion due to missing rationale (MacLeod et al. 2018) in code reviews.

RQ5 Summary - Solutions for most frequent reasons for confusion: We found a total of 13 solutions to five different reasons for confusion in code reviews in the literature. Several solutions are or can be implemented in code review tools. The reasons with the most solutions are *long or complex code change* (5), *dependency between different code changes* (3), *organization of work* (2), and *lack of documentation* (2). We found only one solution proposed in the literature for *missing code change rationale*.

5.2.3 RQ6. What Relationships has Previous Research Established Between the Reasons for Confusion and Their Impacts?

The results of **RQ6** are shown in Table 9. We can observe that *long or complex code change* and *organization of work* have the largest number of impacts described in the literature (4). For the remaining reasons for confusion (*dependency between different code changes*, *lack of documentation*, and *missing code change rationale*) we found they are related to only one impact each in the literature. It is also worth noting that all impacts found in the literature are related to the review process dimension of our framework, exception for *frustration*, which is related to the developer.

We believe that the discrepancy between the number of relationships between reasons for confusion and their impacts can be explained by the number of articles addressing the reasons in the literature: *long or complex code change* and *organization of work* have the largest number of articles. Below we discuss each of the impacts.

– **Delaying** of the code review, i.e., the merge decision, is one of the impacts with the largest number of reasons related to it: *long or complex code change* (Zhang et al. 2012; Gousios et al. 2014; Pascarella et al. 2019; Baysal et al. 2013, 2016; Sadowski et al. 2018; Tao and Kim 2015; Huang et al. 2018a), *organization of work* (Guo and Song 2017), and *dependency between different code changes* (Baum et al. 2016; Zhang et al. 2015; Izquierdo-Cortazar et al. 2017);

- **Decreased review quality** is related to the number of problems identified in the code change during the review, i.e., the review is less effective and potentially identifies less bugs or non-adherences to project guidelines. The literature shows this is caused by *long or complex code change* (Baum et al. 2019; Pascarella et al. 2019; Barnett et al. 2015; Kononenko et al. 2015; Faragó 2015; An et al. 2018; Bosu et al. 2014; Yang et al. 2017), and *organization of work* (Barnett et al. 2015). Some studies also reported that *long or complex code change* can cause the introduction of vulnerabilities issues (Bosu et al. 2014; Yang et al. 2017);
- **Increased development effort** is related to *long or complex code change* and *organization of work*, i.e., the reviewer will have to invest more effort to finish the review (Mishra and Sureka 2014; Huang et al. 2018a; Baysal et al. 2013) , the code change author will need to submit additional revisions if their code change is long or complex (Baysal et al. 2013), as well as the reviewer will not know from which part of the code change they should begin the review in case of *long or complex code changes* (Huang et al. 2018a);
- **Review rejection** was related to three different reasons for confusion: *long or complex code change* (Rigby and Storey 2011; Norikane et al. 2017; Gerede and Mazan 2018; Hellendoorn et al. 2015), *organization of work* (Tao and Kim 2015), and *lack of documentation* (Norikane et al. 2017);
- **Frustration** of the developer is reported in literature as related to *missing code change rationale* (Sadowski et al. 2018).

RQ6 Summary - Impacts of most frequent reasons for confusion: We found that the literature has established the relationship between the five reasons for confusion and five impacts. The reasons for confusion *long or complex code change* and *organization of work* have the largest number of related impacts. Four impacts are related to the review process, while only one is related to the developer (*frustration*).

5.3 Threats to Validity

Following Petersen et al. (2015), the following types of validity should be considered for systematic mapping studies: descriptive validity, theoretical validity, and generalizability.

Descriptive validity is related to the extent to which the observations are described accurately and objectively. We designed a data collection form to support the recording of data, and hence, reduce this threat. We used a spreadsheet to record the data, from which some of the data points were automatically extracted with the aid of the Parsifal tool.

Table 9 Relationships between reasons for confusion and their impacts

Reasons for confusion vs Impacts	Delaying	Decreased review quality	Increased development effort	Review rejection	Frustration
Long or complex change	x	x	x	x	
Organization of work	x	x	x	x	
Dependency between changes	x				
Lack of documentation				x	
Missing rationale					x

Theoretical validity is related to the ability of the authors capture what they intend to capture during the study. Researcher biases might appear during the application of inclusion and exclusion criteria, the selection phase, and extraction of data. Application of the inclusion and exclusion criteria was conducted by the first author, and all excluded articles were reviewed by the remaining authors. The articles remaining for the selection and extraction data phases were split among the four authors in a way that each paper was reviewed by two authors. The authors checked and resolved all disagreements with online meetings. Furthermore, to reduce the bias of the data extraction phase, all the extracted data was reported in a spreadsheet with pre-established fields. The first author reviewed all the data extracted by the other authors and, when necessary, the extracted data was discussed by two or more authors.

External validity concerns the generalizability of the study conclusions. Our results may not apply for to systematic literature reviews as they are different in their goals.

6 Discussion and Implications

The main contribution of this study is fourfold:

i. a improved framework for confusion in code reviews (Section 6.1),
ii. a guideline for developers on how to cope with confusion during code reviews (Section 6.2),
iii. actionable implications for the tool builders (Section 6.3), and
iv. a research agenda for researchers to provide support for confusion (Section 6.4).

6.1 Improved Framework for Confusion in Code Reviews

In this section, we revise the framework for confusion in code reviews presented in Section 3 and augment it with the results of the systematic mapping study (from Section 5). The results of the **RQ6** did not show any new impact related to the most frequent reasons for confusion. The five impacts we found in the literature review are already described in the original framework. From those, all except one are related to the review process. This result suggests literature should also aim at investigating the remaining impacts identified in our first study (Section 3).

Based on the results of the **RQ5**, we could improve our framework as we found new solutions in the literature. From the 13 solutions for confusion we identified in the literature, eight of them are new to our framework. The final improved framework for confusion in code reviews is presented in Table 10 (the new solutions are presented in italics font). We can observe that all new solutions are either related to the review process or to the artifact itself, i.e., the code change. We believe these results highlight the need for more research on the other dimensions related to the developer and the link between developer and artifact.

6.2 Implications for Developers

We found that *long or complex code change* is the most frequently experienced reason for confusion in code reviews according to developers, followed by *a change addressing multiple issues*. These results highlight that to avoid confusion patch authors should aim for changes that are simpler, smaller, and non-composite. Based on the preceding discussion

Table 10 The improved framework for confusion in code reviews

Reasons 30 topics	Impacts 14 topics	Coping strategies 21 topics
Process		
Organization of work	Delaying	Improved organization
Issue tracker, version control	Decreased review quality	of work
Unnecessary change	Additional discussions	Delaying
Not enough time	Blind approval	Assignment to
Dependency between changes	Review rejection	other reviewers
Code ownership	Increased development effort	Blind approval
Community norms	Assignment to	Improve code review tools
	other reviewers	Make the use of super reviews
		Use commit-then-review
		Cluster related code changes
		Create tools to summarize
		similar code changes
Artifact		
Missing rationale	Better solution	Small, clear changes
Discussion of the solution:	Incorrect solution	Improved documentation
non-functional		Make use of salient files
Unsure about system behavior		Ordering of the changes
Lack of documentation		within code change
Discussion of the solution:		Support for the placement
strategy		of code comments
Long, complex change		
Lack of context		
Discussion of the solution:		
correctness		
Impact of change		
Irreproducible bug		
Lack of tests		
Developer		
Disagreement	Decreased confidence	Information requests
Communicative intention	Abandonment	Off-line discussions
Language issues	Frustration	Providing or accepting
Propagation of confusion	Anger	suggestions
Fatigue	Propagation of confusion	Disagreement resolution
Noisy work environment		
Link		
Lack of familiarity with		Improved familiarity with
the existing code		the existing code
Lack of programming skills		Testing the change
Lack of understanding of		Improved familiarity with
the problem		the technology
the change		

Table 10 (continued)

Lack of familiarity with the technology
Lack of knowledge about the development process

we propose the following guideline for developers on how to deal with confusion in code reviews.

1. Before submitting different commits, developers should check and **cluster related code changes** to diminish the chances of creating *dependency between different code changes* (MacLeod et al. 2018), which is the third most frequent reason for confusion.
2. *Long or complex code changes* is the most frequent reason for confusion in code reviews. Even though this is fairly obvious, developers should keep in mind that **making the changes short and simple** will be beneficial for reviewers and also for them, as it improves the chances of their changes being accepted (Gousios et al. 2014; MacLeod et al. 2018; Sadowski et al. 2018). One twist to this formula is that, if changes are simple and strongly related, they should probably be committed together, to reduce reviewing overhead.
3. Developers should also **provide the motivation for the code changes**, as it is important to avoid confusion due to *missing rationale* (MacLeod et al. 2018).
4. Developers should **describe the code changes** to avoid submitting *unclear commit messages* (MacLeod et al. 2018). This will ease the job of reviewers and avoid unnecessary, frustrating, and time consuming requests for additional information.

We believe that our guidelines are complementary to the guidelines proposed by Rigby et al. (2008) as our results derive from different developers of different projects (ANDROID and others) and add new specific instructions on documentation. For instance, Rigby et al. (2008) described APACHE code reviews as: "*(a) early, frequent reviews (b) of small, independent, complete contributions (c) conducted asynchronously by a potentially large, but actually small, group of self-selected experts (d) leading to an efficient and effective peer review technique*". Thus, we can observe that their guideline on (b) relates to two of our guidelines: **making the changes short and simple** and **cluster related code changes**. While the remaining we can say are complementary to each other.

6.3 Implications for Tool Builders

Code reviews are supported by tools such as GERRIT. Currently the only feature of GERRIT that we can relate to confusion reduction is flagging large code changes. Indeed, *long or complex code changes* are among the most popular reasons for confusion in our framework.

Several changes related to *organization of work* can also be addressed by the tools supporting code reviews. For instance, COLLABORATOR[22] supports custom templates and checklists that, if properly configured, might require the change authors to indicate rationale of their change. Similarly, decomposition of composite code changes (Luna Freire et al. 2018; Guo et al. 2019; Barnett et al. 2015; Guo and Song 2017; Tao and Kim 2015; Konopka and Navrat 2015) can be integrated in code review tools: e.g., we envision a bot checking

[22] https://smartbear.com/product/collaborator/overview/

the pull request suggested by a developer, decomposing it when necessary and submitting several pull requests on the developer's behalf. If such an intervention will prove not to be acceptable for developers, functionality of the bot can be restricted to automatic identification of composite changes. Another possibility for code review tools is to provide the code change parts in a specific order to reduce the overall cognitive load of reviewers (Baum et al. 2017). Finally, UPSOURCE code review tool of JETBTRAINS is capable of automatically recommending code reviewers for a given change (Kovalenko et al. 2018). Similar techniques might be integrated in other code review tools. On the same vein, different heuristics to find the best group of reviewers can be integrated into these tools.

6.4 Implications for Researchers

The first item in the agenda for researchers is to invest more on the least addressed reasons for confusion in code reviews: *organization of work*, *dependency between different changes*, *missing code change rationale*, and *lack of documentation*. These are all important reasons for confusion. For example, in the study of Section 3, where we investigated real code reviews and also obtained responses from developers, missing rationale was the most common reason for confusion. Notwithstanding, it is rarely addressed in the scientific literature. These four reasons are in the top five most frequent according to developers. Researchers should aim at exploring more these topics related to code reviews, e.g., by creating automatic approaches to extract the rationale of the change based on code comments or on source code elements.

One the one hand, our findings make it clear that developers should not compose different issues (such as a bug fix and a refactoring) in the same code change, i.e., *decompose composite code changes* (Luna Freire et al. 2018; Guo et al. 2019; Barnett et al. 2015; Guo and Song 2017; Tao and Kim 2015; Konopka and Navrat 2015), since long or composite changes are one of the most frequent reasons for confusion. On the other hand, developers should cluster related changes into a simple solution (MacLeod et al. 2018) to avoid *dependency between different code changes*. This is not an easy trade-off to balance. There has been much investigation into how to break composite changes. However, to the best of our knowledge, there are no papers proposing solutions to balance simple, related changes being clustered together and a change addressing multiple issues being too complex to understand.

Since we found several studies focusing on decomposition of code changes and only one about *dependency between different code changes* (MacLeod et al. 2018), we believe more research is needed to help developers on clustering related changes. For instance, researchers can investigate approaches that analyze code changes before they are integrated and suggest combinations of related commits, thus freeing developers from having to commit multiple small, strongly-connected changes in separate commits.

Another avenue for researchers we see is related to the solution *making use of salient files*, which aims to solve *long or complex code changes*. We found two articles (Huang et al. 2018a; Pascarella et al. 2019) arguing that the use of important files within the code change can help reviewers in the process of conducting reviews by indicating where they should start and how to proceed when reviewing long or complex code changes. In a similar vein, we envision the use of the *task context* (LaToza et al. 2006) of the code change author. This context consists of the set of changed files and also the files and methods the author accessed during the implementation. This information elements can be presented together with the file *diffs* to the reviewer. This approach reduces the need for navigation by providing the reviewer with information that is likely to be necessary to understand the code change.

7 Related Work

In this section, we discuss the related work. Studies related to code reviews are presented in Section 7.1, while studies related to confusion are discussed in Section 7.2.

7.1 Code Review

Code review has been the focus of a plethora of studies (Bavota and Russo 2015; Bacchelli and Bird 2013; Tao et al. 2012; Kononenko et al. 2015; Hentschel et al. 2016; Mukadam et al. 2013; Hamasaki et al. 2013; Thongtanunam et al. 2014; Yang et al. 2016; van Wesel et al. 2017).

Bacchelli and Bird (2013) introduced the term *modern code review* which is supported by tools, is informal, and which happens frequently. They explored the motivations, challenges, and outcomes of code reviews by observing, interviewing, and surveying software developers. Their study shows that finding defects is not the only benefit of code reviews, knowledge transfer and team awareness are also advantages coming from reviews. They also show that the main challenge of code review is understanding the code change and its context.

Tao et al. (2012) investigated how the understanding of code changes affects the development process. They conducted surveys and follow-up emails with software designers, testers, and software managers at MICROSOFT. They shown that *rationale* is the most important information for understanding a code change. However, respondents mentioned that code changes can be easily understood if a good description is provided. They discovered that reviewers could benefit more from the code-exploration features provided by common IDEs (e.g., *call hierarchy* from Eclipse) when they are exploring the change context and estimating its risk.

Bavota and Russo (2015) investigated how code reviews influence the chance of inducing bug fixes, and the quality measured by code coupling, complexity, and readability of the code changes. They showed that commits not reviewed are twice as likely to introduce defects than reviewed commits. Furthermore, the reviewed code changes have a substantially higher readability as compared to unreviewed code changes.

Kononenko et al. (2015) investigate the quality of code reviews in an OSS project by exploring the factors that might affect the reviews. They use the SZZ algorithm to find code changes that introduce defects and then relate them to the code review information. They show that 54% of the code changes that went through the review process introduced defects into the system. Furthermore, personal metrics (reviewer experience and workload) and participation metrics (number of reviewers) are associated with the quality of the code review process. Another interesting result is that the technical properties of the code change (the size, number of files changed, etc.) have a significant impact on the chance of inducing defects in the system.

Pascarella et al. (2018) investigated, by analysing code review comments, what information reviewers need to perform a proper code review. They analysed threads of comments which started from a reviewer's question from a total of 900 code reviews. Additionally, semi-structured interviews and one focus group with developers were conducted to understand the perceptions of the code review needs from developers. They found seven high-level information needs, such as the suitability of an alternative solution, the correct understanding of the code change, rationale, and the context of the code change.

Paixão and Maia (2019) conducted an empirical study to understand the frequency of rebasing operations and their impacts in the code review process by performing a large-scale

investigation of more than 28,000 code reviews of 11 systems. They found that rebasing operations happens in about 75.35% of code reviews, and from those, about 34.21% of rebasing operations tend to tamper with the reviewing process. The authors also propose a methodology to handle rebasing operations in empirical studies that employ code review data.

As for the work related to secondary studies, i.e., systematic literature reviews and systematic mapping studies, we found two articles focused on code reviews. Coelho et al. (2019) focused on refactoring-aware code reviews, in which the reviewers are informed that code change being reviewed contains a refactoring. They conducted a systematic mapping study in order to investigate gather evidence of the studies related to refactoring-aware code reviews in terms of actual support, research trends, and open research topics. Their findings show a lack of proper support when reviewing code change with different types of refactorings and a need for more empirical investigation of the effectiveness of the refactoring-aware solution for code reviews (both in open source and industrial scenarios).

Schettino et al. (2019) conducted a systematic mapping study focusing on code reviewer recommendation, with emphasis on application contexts, the input data, and the empirical validations. They found that several researchers try to validate their work with open source datasets, with GITHUB being the most used. Furthermore, the literature proposed the following data as input for the recommendation systems: social relationships, revision expertize and development. These input were evaluated with Top-k and review activeness metrics.

7.2 Confusion

Confusion has been studied before, also in relation with complex cognitive tasks (D'Mello and Graesser 2014; D'Mello et al. 2014). Approaches to automatic identification of confusion have been recently developed, based on natural language processing (Yang et al. 2015; Jean et al. 2016; Ebert et al. 2017). Yang et al. (2015) used textual content of comments from a forum and its clickstream data to automatically identify posts that express confusion. Their model to identify confusion comprises questions, users' click patterns, and users' linguistic features based on LIWC[23] words. They tried to identify the reasons why users are confused by looking at the recent click behavior. Jean et al. (2016) proposed an approach to detect uncertain expressions based on the statistical analysis of syntactic and lexical features. Ebert et al. (2017) assessed the feasibility of automatic recognition of confusion in code review comments based on linguistic features. They assessed the performance of several classifiers based on supervised training, using a gold standard of 800 comments manually labeled as indicating or not a developer's confusion.

Confusion-related phenomena have been recently investigated in code reviews. Uwano et al. (2006) proposed the use of eye tracking to characterise the performance of developers performing code reviews. They developed a system which captures the source code line number the reviewer's eye is looking at. It is also able to record the transition from a line to another when the reviewer's eyes move, as well as the time spent at each line. Their system was used to perform an experiment with five students reviewing code changes. As result, they identified a specific pattern in reviewer's eyes: "scan". This pattern is characterised by the reviewer's action of reading the entire code before investigating in details each line. Furthermore, reviewers who did not spend sufficient time for the scan tend to take more time for finding defects.

[23] https://liwc.wpengine.com

Ram et al. (2018) aimed to obtain an empirical understanding of what makes a code change easier to review. They empirically defined *reviewability* as how the code change is: i) explained (e.g., in the change description), ii) properly sized and self-contained (e.g., small changes), and iii) aligned with the coding style of the project. They researched academic literature papers, and also blogs and white papers, interviewed professional developers, and evaluated a tool to rate the reviewability of code changes. They found that reviewability is affected by several factors, such as the change description, size, and coherent commit history.

Barik et al. (2017) conducted an eye tracking study to understand how developers use compiler error messages. They found that the difficulty experienced by developers while reading error messages is a significant predictor of task correctness and it also increases the overall hardness of resolving a compiler error

Gopstein et al. (2017) introduced the term *atom of confusion* which is the smallest code pattern that can reliably cause confusion in a developer. Through a controlled experiment with developers, they studied the prevalence and significance of the atoms of confusion in real projects. They shown that the 15 known atoms of confusing occur millions of times in programs like the LINUX kernel and GCC, appearing on average once every 23 lines. They reported a strong correlation between these confusing patterns and bug-fix commits, as well as a tendency for confusing patterns to be eventually commented.

The work presented in this paper is complementary with respect to the ones discussed so far. To the best of our knowledge, these two studies are the first that aim at building a framework of what make developers confused during code reviews, their impacts and what strategies do developers implement to overcome confusion. Additionally, we conducted the first systematic mapping study focused on the reasons for confusion in code reviews.

8 Conclusion

The omnipresence of code reviews calls for a careful attention for obstacles and problems developers experience when reviewing source code or authoring code being reviewed. In this paper, we describe two empirical studies that we conducted to understand the reasons for confusion, its impacts, and the strategies available to deal with it.

We built a confusion framework with 30 reasons for confusion, 14 impacts, and 13 coping strategies adopted by developers. To this aim, we used a concurrent triangulation strategy combining a developer's survey and the content analysis of code review comments in GERRIT. Furthermore, we surveyed developers and identified which ones of the 30 reasons for confusion are experienced most frequently. We found that the five most frequent reasons for confusion are: the presence of *long or complex code change*, poor *organization of work*, *dependency between different code changes*, *lack of documentation*, *missing code change rationale*, and *lack of tests*.

We conducted a systematic mapping study of the scientific literature, which revealed 13 solutions to the most frequent reasons for confusion in code reviews. Moreover, we found that the literature has established the relationship between such reasons for confusion and five impacts in our framework.

Based on our findings we formulated guidelines for developers on how to deal with confusion, suggestions for tool builders on how to support the code review, as well as an agenda for researchers interested in studying code reviews.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give

appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommonshorg/licenses/by/4.0/.

References

An L, Khomh F, Mcintosh S, Castelluccio M (2018) Why did this reviewed code crash? An empirical study of mozilla firefox. In: 2018 25th Asia-Pacific software engineering conference (APSEC), pp 396–405. https://doi.org/10.1109/APSEC.2018.00054

Anderson MJ (2001) A new method for non-parametric multivariate analysis of variance. Austral Ecol 26(1):32–46. https://doi.org/10.1111/j.1442-9993.2001.01070.pp.x. https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1442-9993.2001.01070.pp.x

Armour PG (2000) The five orders of ignorance. Commun ACM 43(10):17–20. https://doi.org/10.1145/352183.352194. http://doi.acm.org/10.1145/352183.352194

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: ICSE. IEEE, pp 712–721

Barik T, Smith J, Lubick K, Holmes E, Feng J, Murphy-Hill E, Parnin C (2017) Do developers read compiler error messages? In: Proceedings of the 39th international conference on software engineering. ICSE '17. IEEE Press, Piscataway, pp 575–585. https://doi.org/10.1109/ICSE.2017.59

Barnett M, Bird C, Brunet J, Lahiri SK (2015) Helping developers help themselves: automatic decomposition of code review changesets. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1, pp 134–144. https://doi.org/10.1109/ICSE.2015.35

Baum T, Kortum F, Schneider K, Brack A, Schauder J (2016) Comparing pre commit reviews and post commit reviews using process simulation. In: 2016 IEEE/ACM international conference on software and system processes (ICSSP), pp 26–35

Baum T, Schneider K, Bacchelli A (2017) On the optimal order of reading source code changes for review. In: 2017 IEEE international conference on software maintenance and evolution (ICSME), pp 329–340. https://doi.org/10.1109/ICSME.2017.28

Baum T, Schneider K, Bacchelli A (2019) Associating working memory capacity and code change ordering with code review performance. Empir Softw Eng 24(4):1762–1798. https://doi.org/10.1007/s10664-018-9676-8

Bavota G, Russo B (2015) Four eyes are better than two: on the impact of code reviews on software quality. In: ICSME, pp 81–90

Baysal O, Kononenko O, Holmes R, Godfrey MW (2013) The influence of non-technical factors on code review. In: 2013 20th working conference on reverse engineering (WCRE), pp 122–131. https://doi.org/10.1109/WCRE.2013.6671287

Baysal O, Kononenko O, Holmes R, Godfrey MW (2016) Investigating technical and non-technical factors influencing modern code review. Empir Softw Eng 21(3):932–959. https://doi.org/10.1007/s10664-015-9366-8

Begel A, Vrzakova H (2018) Eye movements in code review. In: Proceedings of the workshop on eye movements in programming. EMIP '18. Association for Computing Machinery, New York. https://doi.org/10.1145/3216723.3216727

Begel A, Zimmermann T (2014) Analyze this! 145 questions for data scientists in software engineering. In: Proceedings of the 36th international conference on software engineering, ICSE 2014, pp 12–23

Boehm B, Basili VR (2001) Top 10 list [software development]. Computer 34(1):135–137

Bosu A, Carver JC, Hafiz M, Hilley P, Janni D (2014) Identifying the characteristics of vulnerable code changes: an empirical study. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. FSE 2014. Association for Computing Machinery, New York, pp 257–268. https://doi.org/10.1145/2635868.2635880

Bosu A, Carver JC, Bird C, Orbeck J, Chockley C (2017) Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. IEEE Trans Softw Eng 43(1):56–75

Calefato F, Lanubile F, Novielli N (2019) An empirical assessment of best-answer prediction models in technical q&a sites. Empir Softw Eng 24(2):854–901. https://doi.org/10.1007/s10664-018-9642-5

Catolino G, Ferrucci F (2019) An extensive evaluation of ensemble techniques for software change prediction. J Softw: Evol Process 31(9):e2156. https://doi.org/10.1002/smr.2156. https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2156, https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2156

Clarke KR (1993) Non-parametric multivariate analysis of changes in community structure. Austral J Ecol 18:117–143

Coelho F, Massoni T, LG Alves E (2019) Refactoring-aware code review: a systematic mapping study. In: 2019 IEEE/ACM 3rd international workshop on refactoring (IWoR), pp 63–66. https://doi.org/10.1109/IWoR.2019.00019

Cohen J, Teleki S, Brown E (2006) Best kept secrets of peer code review. Smart Bear Inc, Somerville

D'Mello S, Graesser A (2014) Confusion and its dynamics during device comprehension with breakdown scenarios. Acta Psychol 151:106–116

D'Mello S, Lehman B, Pekrun R, Graesser A (2014) Confusion can be beneficial for learning. Learn Instruct 29:153–170. https://doi.org/10.1016/j.learninstruc.2012.05.003

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. In: Shull F, Singer J, Sjøberg DIK (eds) Guide to advanced empirical software engineering. Springer, London, pp 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

Ebert F, Castor F, Novielli N, Serebrenik A (2017) Confusion detection in code reviews. In: ICSME, pp 549–553

Ebert F, Castor F, Novielli N, Serebrenik A (2018) Communicative intention in code review questions. In: ICSME

Ebert F, Castor F, Novielli N, Serebrenik A (2019) Confusion in code reviews: reasons, impacts, and coping strategies. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER), pp 49–60. https://doi.org/10.1109/SANER.2019.8668024

Fagan ME (1976) Design and code inspections to reduce errors in program development. IBM Syst J 15(3):182–211. https://doi.org/10.1147/sj.153.0182

Faragó C (2015) Variance of source code quality change caused by version control operations. Acta Cybern 22(1):35–56. https://doi.org/10.14232/actacyb.22.1.2015.4

Finfgeld-Connett D (2014) Use of content analysis to conduct knowledge-building and theory-generating qualitative systematic reviews. Qual Res 14(3):341–352. https://doi.org/10.1177/1468794113481790

Foddy WH (1993) Constructing questions for interviews and questionnaires: theory and practice in social research. Cambridge University Press, Cambridge

Gerede ÇE, Mazan Z (2018) Will it pass? Predicting the outcome of a source code review, vol 26, pp 1343–135. https://doi.org/10.3906/elk-1707-173. https://www.scopus.com/inward/record.uri?eid=2-s2.0-85048211876&doi=10.39066, cited By 0

German DM, Robles G, Poo-Caamaño G, Yang X, Iida H, Inoue K (2018) "Was my contribution fairly reviewed?": a framework to study the perception of fairness in modern code reviews. In: Proceedings of the 40th international conference on software engineering. ICSE '18. ACM, New York, pp 523–534. https://doi.org/10.1145/3180155.3180217. http://doi.acm.org/10.1145/3180155.3180217

Glaser BG, Strauss AL (1967) The discovery of grounded theory: strategies for qualitative research. Aldine de Gruyter, New York

Gopstein D, Iannacone J, Yan Y, DeLong L, Zhuang Y, Yeh MKC, Cappos J (2017) Understanding misunderstandings in source code. In: ESEC/FSE. ACM, New York, pp 129–139

Gousios G, Pinzger M, Deursen AV (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering. ICSE 2014. ACM, New York, pp 345–355. https://doi.org/10.1145/2568225.2568260. http://doi.acm.org/10.1145/2568225.2568260

Greiler M (2016) On to code review: lessons learned @ microsoft. https://pt.slideshare.net/mgreiler/on-to-code-review-lessons-learned-at-microsoft, keynote for QUATIC 2016—the 10th international conference on the quality of information and communication technology

Groves RM, Fowler FJ, Couper MP, Lepkowski JM, Singer E, Tourangeau R (2009) Survey methodology, 2nd edn. Wiley, New York

Guo B, Song M (2017) Interactively decomposing composite changes to support code review and regression testing. In: 2017 IEEE 41st annual computer software and applications conference (COMPSAC), vol 1, pp 118–127. https://doi.org/10.1109/COMPSAC.2017.153

Guo B, Kwon YW, Song M (2019) Decomposing composite changes for code review and regression test selection in evolving software. J Comput Sci Technol 34(2):416–436. https://doi.org/10.1007/s11390-019-1917-9

Hamasaki K, Kula RG, Yoshida N, Cruz AEC, Fujiwara K, Iida H (2013) Who does what during a code review? Datasets of oss peer review repositories. In: MSR. IEEE, pp 49–52

Hellendoorn VJ, Devanbu PT, Bacchelli A (2015) Will they like this? Evaluating code contributions with language models. In: 2015 IEEE/ACM 12th working conference on mining software repositories, pp 157–167

Hentschel M, Hähnle R, Bubel R (2016) Can formal methods improve the efficiency of code reviews? In: IFM. Springer, pp 3–19

Huang Y, Jia N, Chen X, Hong K, Zheng Z (2018a) Salient-class location: help developers understand code change in code review. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. ESEC/FSE 2018. ACM, New York, pp 770–774. https://doi.org/10.1145/3236024.3264841. http://doi.acm.org/10.1145/3236024.3264841

Huang Y, Jia N, Zhou Q, Chen X, Yingfei X, Luo X (2018b) Guiding developers to make informative commenting decisions in source code. In: 2018 IEEE/ACM 40th international conference on software engineering: companion (ICSE-Companion), pp 260–261

Izquierdo-Cortazar D, Sekitoleko N, Gonzalez-Barahona JM, Kurth L (2017) Using metrics to track code review performance. In: Proceedings of the 21st international conference on evaluation and assessment in software engineering. EASE'17. ACM, New York, pp 214–223. https://doi.org/10.1145/3084226.3084247. http://doi.acm.org/10.1145/3084226.3084247

Jean PA, Harispe S, Ranwez S, Bellot P, Montmain J (2016) Uncertainty detection in natural language: a probabilistic model. In: International conference on web intelligence, mining and semantics. ACM, New York, pp 10:1–10:10

Jordan ME, Schallert DL, Park Y, Lee S, hui Vanessa Chiang Y, Cheng ACJ, Song K, Chu HNR, Kim T, Lee H (2012) Expressing uncertainty in computer-mediated discourse: language as a marker of intellectual work. Discourse Process 49(8):660–692

Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Tech. Rep. EBSE 2007-001, Keele University and Durham University Joint Report

Kitchenham B, Pfleeger SL (2008) Personal opinion surveys. In: Shull F, Singer J, Sjoberg DIK (eds) Guide to advanced empirical software engineering, pp 63–92

Kononenko O, Baysal O, Guerrouj L, Cao Y, Godfrey MW (2015) Investigating code review quality: do people and participation matter? In: 2015 IEEE international conference on software maintenance and evolution (ICSME), pp 111–120. https://doi.org/10.1109/ICSM.2015.7332457

Kononenko O, Rose T, Baysal O, Godfrey M, Theisen D, de Water B (2018) Studying pull request merges: a case study of shopify's active merchant. In: Proceedings of the 40th international conference on software engineering: software engineering in practice. ICSE-SEIP '18. ACM, New York, pp 124–133. https://doi.org/10.1145/3183519.3183542. http://doi.acm.org/10.1145/3183519.3183542

Konopka M, Navrat P (2015) Untangling development tasks with software developer's activity. In: 2015 IEEE/ACM 2nd international workshop on context for software development, pp 13–14. https://doi.org/10.1109/CSD.2015.10

Kovalenko V, Tintarev N, Pasynkov E, Bird C, Bacchelli A (2018) Does reviewer recommendation help developers? IEEE Trans Softw Eng 1–1

LaToza TD, Venolia G, DeLine R (2006) Maintaining mental models: a study of developer work habits. In: ICSE. ACM, New York, pp 492–501

Lee A, Carver JC, Bosu A (2017) Understanding the impressions, motivations, and barriers of one time code contributors to FLOSS projects: a survey. In: Uchitel S, Orso A, Robillard MP (eds) Proceedings of the 39th international conference on software engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017. IEEE/ACM, pp 187–197

Lenberg P, Feldt R, Tengberg LGW, Tidefors I, Graziotin D (2017) Behavioral software engineering—guidelines for qualitative studies. CoRR arXiv:1712.08341

Luna Freire VdC, Brunet J, de Figueiredo JCA (2018) Automatic decomposition of java open source pull requests: a replication study. In: Tjoa AM, Bellatreche L, Biffl S, van Leeuwen J, Wiedermann J (eds) SOFSEM 2018: theory and practice of computer science. Springer International Publishing, Cham, pp 255–268

MacLeod L, Greiler M, Storey MA, Bird C, Czerwonka J (2018) Code reviewing in the trenches: challenges and best practices. IEEE Softw 35(4):34–42. https://doi.org/10.1109/MS.2017.265100500

Mäntylä MV, Lassenius C (2009) What types of defects are really discovered in code reviews? TSE 35(3):430–448

Martin RC (2003) Agile software development: principles, patterns, and practices. Prentice Hall PTR, Upper Saddle River

McArdle BH, Anderson MJ (2001) Fitting multivariate models to community data: a comment on distance-based redundancy analysis. Ecology 82(1):290–297. https://doi.org/10.1890/0012-9658(2001)082[0290:FMMTCD]2.0.CO;2

McIntosh S, Kamei Y, Adams B, Hassan AE (2015) An empirical study of the impact of modern code review practices on software quality. In: ESE, pp 1–44

Mishra R, Sureka A (2014) Mining peer code review system for computing effort and contribution metrics for patch reviewers. In: 2014 IEEE 4th workshop on mining unstructured data, pp 11–15. https://doi.org/10.1109/MUD.2014.11

Mohamed A, Zhang L, Jiang J, Ktob A (2018) Predicting which pull requests will get reopened in github. In: 2018 25th Asia-Pacific software engineering conference (APSEC), pp 375–385. https://doi.org/10.1109/APSEC.2018.00052

Morales R, McIntosh S, Khomh F (2015) Do code review practices impact design quality? A case study of the qt, vtk, and itk projects. In: 2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER), pp 171–180. https://doi.org/10.1109/SANER.2015.7081827

Mukadam M, Bird C, Rigby PC (2013) Gerrit software code review data from android. In: MSR. IEEE, pp 45–48

Norikane T, Ihara A, Matsumoto K (2017) Which review feedback did long-term contributors get on oss projects? In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), pp 571–572. https://doi.org/10.1109/SANER.2017.7884682

Norikane T, Ihara A, Matsumoto K (2018) Do review feedbacks influence to a contributor's time spent on oss projects? In: 2018 IEEE international conference on big data, cloud computing, data science engineering (BCD), pp 109–113

Paixão M, Maia PH (2019) Rebasing considered harmful: a large-scale investigation in modern code review. In: 2019 IEEE 19th international working conference on source code analysis and manipulation (SCAM)

Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2015) Mining version histories for detecting code smells. IEEE Trans Softw Eng 41(5):462–489. https://doi.org/10.1109/TSE.2014.2372760

Palomba F, Tamburri DA, Serebrenik A, Zaidman A, Fontana FA, Oliveto R (2018) How do community smells influence code smells? In: Proceedings of the 40th international conference on software engineering: companion proceedings. ICSE '18. ACM, New York, pp 240–241. https://doi.org/10.1145/3183440.3194950. http://doi.acm.org/10.1145/3183440.3194950

Pangsakulyanont T, Thongtanunam P, Port D, Iida H (2014) Assessing MCR discussion usefulness using semantic similarity. In: 2014 6th International workshop on empirical software engineering in practice (IWESEP), pp 49–54. https://doi.org/10.1109/IWESEP.2014.11

Pascarella L, Spadini D, Palomba F, Bruntik M, Bacchelli A (2018) Information needs in contemporary code review. In: Proceedings of the ACM conference on computer supported cooperative work, CSCW '18

Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction, vol 150, pp 22–36. https://doi.org/10.1016/j.jss.2018.12.001. http://www.sciencedirect.com/science/article/pii/S0164121218302656

Petersen K, Feldt R, Mujtaba S, Mattsson M (2008) Systematic mapping studies in software engineering. In: Proceedings of the 12th international conference on evaluation and assessment in software engineering. EASE'08. BCS Learning & Development Ltd., Swindon, pp 68–77. http://dl.acm.org/citation.cfm?id=2227115.2227123

Petersen K, Vakkalanka S, Kuzniarz L (2015) Guidelines for conducting systematic mapping studies in software engineering: an update. Inf Softw Technol 64:1–18. https://doi.org/10.1016/j.infsof.2015.03.007. http://www.sciencedirect.com/science/article/pii/S0950584915000646

Qiu HS, Nolte A, Brown A, Serebrenik A, Vasilescu B (2019) Going farther together: the impact of social capital on sustained participation in open source. In: ICSE. IEEE

Ram A, Ashok Sawant A, Marco C, Bacchelli A (2018). In: 26th ACM Joint European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE '18

Rigby PC (2011) Understanding open source software peer review: review processes, parameters and statistical models, and underlying behaviours and mechanisms. PhD thesis, University of Victoria, Victoria, B.C., Canada, Canada. http://hdl.handle.net/1828/3258

Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ESEC/FSE, vol 2013. ACM, New York, pp 202–212. https://doi.org/10.1145/2491411.2491444. http://doi.acm.org/10.1145/2491411.2491444

Rigby PC, Storey MD (2011) Understanding broadcast based peer review on open source software projects. In: Taylor RN, Gall HC, Medvidovic N (eds) 2011 33rd International conference on software engineering (ICSE). ACM, pp 541–550

Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: a case study of the apache server. In: Proceedings of the 30th international conference on software engineering. ICSE '08. Association for Computing Machinery, New York, pp 541–550. https://doi.org/10.1145/1368088.1368162

Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern code review: a case study at Google. In: Proceedings of the 40th international conference on software engineering: software engineering in practice. ICSE-SEIP '18. ACM, New York, pp 181–190. https://doi.org/10.1145/3183519.3183525. http://doi.acm.org/10.1145/3183519.3183525

Schettino VJ, Araújo MAP, David JMN, Braga RMM (2019) Towards code reviewer recommendation: a systematic review and mapping of the literature. In: Proceedings of the XXII Iberoamerican conference on software engineering, CIbSE 2019, La Habana, Cuba, April 22–26, 2019, pp 558–571

Scott AJ, Knott M (1974) A cluster analysis method for grouping means in the analysis of variance. Biometrics 30(3):507–512. http://www.jstor.org/stable/2529204

Singer J, Vinson NG (2002) Ethical issues in empirical studies of software engineering. IEEE Trans Softw Eng 28(12):1171–1180. https://doi.org/10.1109/TSE.2002.1158289

Steele CM, Aronson J (1995) Stereotype threat and the intellectual test performance of African Americans. J Pers Social Psychol 69(5):797–811

Stol KJ, Ralph P, Fitzgerald B (2016) Grounded theory in software engineering research: a critical review and guidelines. In: ICSE, pp 120–131. https://doi.org/10.1145/2884781.2884833

Sutherland A, Venolia G (2009) Can peer code reviews be exploited for later information needs? In: ICSE-Companion, pp 259–262

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2017) An empirical comparison of model validation techniques for defect prediction models. IEEE Trans Softw Eng (TSE) 43(1):1–18

Tao Y, Kim S (2015) Partitioning composite code changes to facilitate code review. In: 2015 IEEE/ACM 12th working conference on mining software repositories, pp 180–190. https://doi.org/10.1109/MSR.2015.24

Tao Y, Dang Y, Xie T, Zhang D, Kim S (2012) How do software engineers understand code changes?: an exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering. FSE '12. ACM, New York, pp 51:1–51:11. https://doi.org/10.1145/2393596.2393656. http://doi.acm.org/10.1145/2393596.2393656

Thompson C, Wagner D (2017) A large-scale study of modern code review and security in open source projects. In: Proceedings of the 13th international conference on predictive models and data analytics in software engineering. PROMISE. ACM, New York, pp 83–92. https://doi.org/10.1145/3127005.3127014. http://doi.acm.org/10.1145/3127005.3127014

Thongtanunam P, Yang X, Yoshida N, Kula RG, Cruz AEC, Fujiwara K, Iida H (2014) Reda: a web-based visualization tool for analyzing modern code review dataset. In: ICSME, pp 605–608

Tichy WF (1985) Rcs—a system for version control. Softw: Pract Exp 15:637–654

Uwano H, Nakamura M, Monden A, Matsumoto K (2006) Analyzing individual performance of source code review using reviewers' eye movement. In: Proceedings of the 2006 symposium on eye tracking research & applications. ETRA '06. ACM, New York, pp 133–140. https://doi.org/10.1145/1117309.1117357. http://doi.acm.org/10.1145/1117309.1117357

Vasilescu B, Filkov V, Serebrenik A (2015a) Perceptions of diversity on git hub: a user survey. In: 2015 IEEE/ACM 8th international workshop on cooperative and human aspects of software engineering, pp 50–56. https://doi.org/10.1109/CHASE.2015.14

Vasilescu B, Posnett D, Ray B, van den Brand MGJ, Serebrenik A, Devanbu P, Filkov V (2015b) Gender and tenure diversity in github teams. In: Proceedings of the 33rd annual ACM conference on human factors in computing systems. CHI '15. ACM, New York, pp 3789–3798. https://doi.org/10.1145/2702123.2702549. http://doi.acm.org/10.1145/2702123.2702549

Wang J, Shih PC, Wu Y, Carroll JM (2015) Comparative case studies of open source software peer review practices. Inf Softw Technol 67(C):1–12. https://doi.org/10.1016/j.infsof.2015.06.002

Wang C, Xie X, Liang P, Xuan J (2017) Multi-perspective visualization to assist code change review. In: 2017 24th Asia-Pacific software engineering conference (APSEC), pp 564–569. https://doi.org/10.1109/APSEC.2017.66

van Wesel P, Lin B, Robles G, Serebrenik A (2017) Reviewing career paths of the openstack developers. In: ICSME. IEEE Computer Society, pp 544–548

Wiegers KE (2002) Peer reviews in software: a practical guide. Addison-Wesley Longman Publishing Co., Inc., Boston

Yang D, Wen M, Howley I, Kraut R, Rose C (2015) Exploring the effect of confusion in discussion forums of massive open online courses. In: ACM conference on learning @ scale. ACM, pp 121–130

Yang L, Li X, Yu Y (2017) Vuldigger: a just-in-time and cost-aware tool for digging vulnerability-contributing changes. In: GLOBECOM 2017—2017 IEEE global communications conference, pp 1–7

Yang X, Kula RG, Yoshida N, Iida H (2016) Mining the modern code review repositories: a dataset of people, process and product. In: MSR. ACM, pp 460–463

Zanaty FE, Hirao T, McIntosh S, Ihara A, Matsumoto K (2018) An empirical study of design discussions in code review. In: Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement. ESEM '18. Association for Computing Machinery, New York. https://doi.org/10.1145/3239235.3239525
Zhang F, Khomh F, Zou Y, Hassan AE (2012) An empirical study on factors impacting bug fixing time. In: 2012 19th Working conference on reverse engineering, pp 225–234
Zhang T, Song M, Pinedo J, Kim M (2015) Interactive code review for systematic changes. In: Proceedings of the 37th international conference on software engineering, vol 1. ICSE '15. IEEE Press, Piscataway, pp 111–122. http://dl.acm.org/citation.cfm?id=2818754.2818771
Zimmermann T (2016). In: Menzies T, Williams L, Zimmermann T (eds) Card-sorting: from text to themes. Morgan Kaufmann, Boston, pp 137–141. https://doi.org/10.1016/B978-0-12-804206-9.00027-1. https://www.sciencedirect.com/science/article/pii/B9780128042069000271

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Felipe Ebert is a post-doc researcher at Eindhoven University of Technology, The Netherlands. His research interests are related to how software systems and developers interact with each other. He is main interested in both technical and social aspects of software maintenance, specifically code reviews, mining software repositories, and social development aspects. In the past, he also has worked with error handling and software energy consumption.

Fernando Castor is an Associate Professor at the Informatics Center of the Federal University of Pernambuco, Brazil. His broad research goal is to help developers build more efficient software systems more efficiently. More specifically, he conducts research in the areas of Software Maintenance, Software Energy Efficiency, and Error Handling.

Nicole Novielli is an Assistant Professor at the University of Bari, where she received a PhD in Computer Science in 2010. Her research interests lie at the intersection of software engineering and affective computing with a specific focus on mining emotions and opinions from developers' communication traces and sensor-based recognition of developers' cognitive and affective states. In 2016, she started the ICSE workshop series on Emotion Awareness in Software Engineering.

Alexander Serebrenik is a Full Professor of Social Software Engineering at Eindhoven University of Technology. His research goal is to facilitate evolution of software by taking into account social aspects of software development. He has co-authored a book "Evolving Software Systems" (Springer Verlag, 2014), and more than 100 scientific papers and articles. He has won several distinguished paper and distinguished review awards.

Affiliations

Felipe Ebert[1] · **Fernando Castor**[2] · **Nicole Novielli**[3] · **Alexander Serebrenik**[1]

Fernando Castor
castor@cin.ufpe.br

Nicole Novielli
nicole.novielli@uniba.it

Alexander Serebrenik
a.serebrenik@tue.nl

[1] Eindhoven University of Technology, Eindhoven, The Netherlands

[2] Federal University of Pernambuco, Recife, Brazil

[3] University of Bari, Bari, Italy