Information and Software Technology xxx (xxxx) xxx

Contents lists available at ScienceDirect

Information and Software Technology

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

Why is my code change abandoned?

Qingye Wang [a], Xin Xia [b,*], David Lo [c], Shanping Li [a]

[a] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[b] Faculty of Information Technology, Monash University, Melbourne, Australia
[c] School of Information Systems, Singapore Management University, Singapore

ARTICLE INFO

Keywords:
Code review
Empirical study
Abandoned change

ABSTRACT

Context: Software developers contribute numerous changes every day to the code review systems. However, not all submitted changes are merged into a codebase because they might not pass the code review process. Some changes would be abandoned or be asked for resubmission after improvement, which results in more workload for developers and reviewers, and more delays to deliverables.

Objective: To understand the underlying reasons why changes are abandoned, we conduct an empirical study on the code review of four open source projects (Eclipse, LibreOffice, OpenStack, and Qt).

Method: First, we manually analyzed 1459 abandoned changes. Second, we leveraged the open card sorting method to label these changes with reasons why they were abandoned, and we identified 12 categories of reasons. Next, we further investigated the frequency distribution of the categories across projects. Finally, we studied the relationship between the categories and time-to-abandonment.

Results: Our findings include the following: (1) *Duplicate* changes are the majority of the abandoned changes; (2) the frequency distribution of abandoned changes across the 12 categories is similar for the four open source projects; (3) 98.39% of the changes are abandoned within a year.

Conclusion: Our study concluded the root causes of abandoned changes, which will help developers submit high-quality code changes.

1. Introduction

Code review, a manual inspection of changes by developers other than authors, is recognized as an effective way to reduce software defects and improve development quality [1,2,43]. Bavota and Russo [7] found that unreviewed changes (i.e., changes that did not undergo a review process) have over two times more chances of introducing bugs than reviewed changes (i.e., changes that underwent a review process). Moreover, they found that code committed after review has a substantially higher readability than unreviewed code. In 1976, Fagan formalized a highly structured process for code inspection, reducing errors in software development [15]. Over the years, many researchers have done much work on code inspection [3,13,26,35,42]. Unfortunately, while effective in identifying defects, the time-consuming and cumbersome nature of traditional code inspection has been shown to limit its adoption [25,44,51].

In contrast, Modern Code Review (MCR) provides a lightweight, informal and tool-based code review practice, and it has been adopted both in industrial and open source projects [4,5,32,37], e.g., at companies such as Microsoft [11], Facebook, Google, etc. In the process of MCR, a developer submits a change to a code review system (e.g., Gerrit), then the code review system assigns reviewers to review this change. After review and discussion, the reviewers decide the outcome of the change, which would be either merged, abandoned or resubmitted after modification. Typically, there are a fair proportion of changes that are not merged into a codebase after review. Rigby and German [38] found that 56% of patches were rejected in an Apache project. Weißgerber et al. [52] found that about 60% of patches were abandoned in OpenAFS and FLAC projects.

Change rejection wastes time of the contributors and the reviewers, and reduces development efficiency. Such wasted time could be used to contribute more changes that are eventually integrated into the codebase. Moreover, a high change rejection rate might indicate problems with software development process [45].

In this paper, we investigate why changes are abandoned. We focus on analyzing changes in Gerrit, a popular code review management system. The system is widely used in a large body of open source projects

* Corresponding author.
E-mail addresses: wqyy@zju.edu.cn (Q. Wang), xin.xia@monash.edu (X. Xia), davidlo@smu.edu.sg (D. Lo), shan@zju.edu.cn (S. Li).

https://doi.org/10.1016/j.infsof.2019.02.007
Received 10 July 2018; Received in revised form 23 February 2019; Accepted 25 February 2019
Available online xxx
0950-5849/

such as Eclipse and OpenStack. Moreover, there are many studies investigating code changes in Gerrit [9,18,27,32,34,49].

We conduct an empirical study to investigate why a change is abandoned. To do so, we analyzed code changes of the four projects, i.e., Eclipse, Libreoffice, OpenStack, and Qt. We manually analyzed 1459 abandoned changes in total from the studied projects by reading the comments of these changes, and then we applied open card sorting to classify all the reasons. Our study aims to answer the following research questions:

RQ1. Why were changes abandoned? What are some categories of rationales behind this phenomenon?

RQ2. What are the frequency distributions of abandoned changes across the rationales and projects?

RQ3. How much time is spent to review changes before they were abandoned?

There are some studies related to ours: Rigby and German [38] studied code review process used by open source projects. Weißgerber et al. [52] extracted patches from emails and found their application in repositories. Tao et al. [47] analyzed 300 patches from Eclipse and Mozilla by manually inspecting their patch review comments to understand why they were rejected. Our work is related but different from theirs: (1) The first two studies did not investigate specific reasons why changes were abandoned. (2) Tao et al. manually investigated only 300 rejected patches from two projects; also, they focused on patches in Bugzilla which is a bug tracking system. On the other hand, we investigated 1459 abandoned changes from four projects and we analyzed changes in Gerrit which is a code review system.

The main contributions of our work are as follows:

1. We are the first to perform a large-scale empirical study in Gerrit to manually categorize 1459 abandoned changes of the four open source projects into various categories by using the open card sorting method.
2. We analyze the frequency distributions of the categories across different projects.
3. We investigate the relationship between the categories and the abandonment time.
4. We recommend five aspects to help developers submit high-quality changes in practice.

The rest of this paper is organized as follows. In Section 2 we introduce our empirical study setup. Section 3–5 describe the empirical study results. Section 6 discusses the implications of our study and the threats to validity. Section 7 briefly presents related work. Section 8 concludes the paper and mentions future work.

2. Empirical study setup

2.1. Research questions

RQ1. Why were changes abandoned? What are some categories of rationales behind this phenomenon?

Categorizing reasons why changes were abandoned can provide a good reference for developers and help them push better changes.

RQ2. What are the frequency distributions of abandoned changes across the rationales and projects?

Different projects are developed under different requirements and intend to accomplish different tasks. Do they show similar frequency distribution of categories of the abandoned changes? If the frequency distributions show similar trends across projects, we could rank the categories identified in our answer to RQ1 from most to least common to help developers not to forget the checks with respect to more common categories, and help them to focus their attention on these common categories.

RQ3. How much time is spent to review changes before they were abandoned?

We investigate the relationship between time-to-abandonment and the categories. The result may indicate that more attention should be paid to a specific category of abandoned changes.

2.2. Data collection

Retrieving representative open source projects. We investigated four popular open source projects, i.e., Eclipse, LibreOffice, OpenStack, and Qt. The reasons we choose the four projects are as follows: First, their code review systems contain a large number of changes (i.e.,60,000–630,000 changes). That is, they are popular in open source projects. Second, there are at least dozens of changes submitted into these projects every day. In other words, the developers are active in these projects. Third, the four projects represent different programming languages. Eclipse project refers to Java. Qt project refers to C++. OpenStack project refers to Python. LibreOffice project refers to C++ and Java. To summarize, these four projects are popular and represent the diversity of different programming languages. In code review system, there are three main status labels: “Open”, “Merged”, and “Abandoned”. For the Qt project, there are other status labels: “Staged”, “Integrating”, etc. In our study, we want to know why a change is abandoned, so we only collect changes with the “Abandoned” status.

The process we collected data. Take LibreOffice project as an example: we first downloaded all the abandoned changes from the website of https://gerrit.libreoffice.org/#/q/status:abandoned, then we randomly selected the changes from our collected data. That is, initially, we randomly selected 1000 changes from each of the project. And then we manually read the discussion of these projects to identify the reasons of these abandoned changes. We removed the changes which are hard to identify the reasons when reading the comments. Finally, we collected 309, 590, 346, 214 changes for Eclipse, LibreOffice, OpenStack and Qt, respectively. In total, we analyzed 1459 abandoned changes. Statistics of our dataset is shown in Table 1.

Analyzing code change comments. Code change comments are one of the main parts of code review where reviewers can add their feedback and suggestions for changes. These comments play a significant role in code review practice. Comments point out bugs, provide suggestion or identify violations of team common practice, coding convention and standard. It can help contributors submit a higher quality change to the codebase and improve authors’ development skills. Through these comments, developers exchange their ideas with others and put forward better solutions for solving problems. With code review comments, Ebert et al. [14] did a study to identify the factors that confuse code reviewers and understand how confusion impacts the efficiency and effectiveness of code review(er)s. In our study, we extract the reasons that cause changes to be abandoned by analyzing change comments in code review.

Validation survey. To confirm our study, we sent emails to the developer who submitted the code change. In these emails, we asked two simple questions: (1) *Why was the following change abandoned?* (We attached a URL of an abandoned change created by the developer.) (2) *Are there any other reasons why changes are abandoned in general?* Totally, we sent out 203 emails and we received 80 replies from developers who contributed to the four projects.

Table 1
Dataset.

Project	# Changes
Eclipse	309
LibreOffice	590
Opentack	346
Qt	214
In total	1459

Table 2
Classification scheme.

Category	Description	Abbreviation
Duplicate	Changes that were similar to other changes.	Dupl
Lack of Feedback	Changes that were abandoned because the contributors did not respond to the reviewers' comments or no reviewer wanted to review the changes.	Lacf
Contributor Operation	Changes that were abandoned due to contributor's wrong operation in the process of pushing commit.	Cono
New Work	The contributors continued the work with a new change.	Newo
Incomplete/Wrong Fix	Changes that were wrong or imperfect.	Inco
Superfluous	Changes that were not worthwhile to make.	Supf
Test	Changes that were created for testing purposes and will never be merged.	Test
Branch Transfer	Changes that were transferred from one branch to another branch.	Brtr
Complicated Change	Changes that needed to be split into smaller and independent ones.	Comc
Merge Conflict	Changes that caused merge conflicts.	Merc
Give Up	Contributors gave up on improving changes because they had no time or they cannot fix the issues highlighted in comments.	Givp
Other	Other reasons that resulted in changes to be abandoned.	Othe

Table 3
Subcategories of *Duplicate*.

Category	Subcategory	Description
Duplicate	Already Done	The issue in the change was already done in other change.
	Suboptimal Solution	There were better solutions in other changes.
	Integrated	The change was a part of another change or had been integrated into another change.

Table 4
Subcategories of *Contributor Operation*.

Category	Subcategory	Description
Operation	Wrong Branch	Contributors pushed changes to a wrong branch.
	Accidental Push	Contributors accidentally pushed changes which were not ready to be reviewed.
	Update Change	Contributors accidentally created a new change when they intended to update an existing one.
	Other Wrong Operation	Changes related to wrong operation but did not belong to any of the above subcategories.

2.3. Methodology

To analyze the 1459 abandoned changes, we extracted the title and comments of each change and followed a card sort process [46].

Step 1: Card sorting. For each change, we create one card. The card includes change information extracted from change title and change comments. The first author and one graduate student jointly performed this card sort process. The specific steps are as follows:

Iteration 1. We first randomly select the Qt project and manually check its changes. Then we put these changes into different sets according to their root causes. Next, for each set, we discuss and label it by referring to the categories that were defined in Tao et al.'s study [47]. The primary classification scheme contains nine categories as shown in Table 2 (except *Merge Conflict, Give Up and Other* categories). Our categories are based on Tao et al.'s study [47], and among the nine categories above, there are three categories (i.e., *Duplicate, Incomplete/Wrong Fix, Complicated Change*) same to the categories of Tao et al.'s study.

Iteration 2. We manually inspect changes in the other three projects (Libreoffice, OpenStack, and Eclipse), and we encounter some new reasons. Thus we create three new categories (i.e., *Merge Conflict, Give Up and Other*) as described in Table 2.

Iteration 3. We find that *Duplicate* category accounts for a large proportion of changes. So we further decompose it into three subcategories shown in Table 3.

Iteration 4. We find that the reasons in the *Contributor Operation* category are various. So we decompose it into four subcategories shown in Table 4.

Step 2: Labeling. The first author and one graduate student independently labeled the 1459 changes of the four open source projects. We measure the agreement between the two labelers with Fleiss Kappa [17]. Fleiss Kappa is used for measuring the reliability of agreement between a number of raters when categorical ratings are assigned to many items or classifying items. Table 5 shows the interpretation of Kappa values. The overall Kappa value between the two labelers on all changes is 0.68. It indicates substantial agreement between the labelers. After completing the manually labeling process, the two labelers discussed their disagreements, and at last, they reached a common decision.

Table 5
Interpretation of Kappa values.

Kappa value	Interpretation
< 0	Poor agreement
[0.01, 0.20]	Slight agreement
[0.21, 0.40]	Fair agreement
[0.41, 0.60]	Moderate agreement
[0.61, 0.80]	Substantial agreement
[0.81, 1.00]	Almost perfect agreement

3. Category

3.1. Category overview

This section answers RQ1. The reasons why changes are abandoned are various. In our reply emails, developers pointed out various reasons for abandoned changes. Some examples are listed below:

* *"In general changes are abandoned by various reasons - sometimes they just are not good, sometimes a better patch is proposed, and sometimes patches are just examples of some behavior which are shared with other developers."*
* *"Reasons: 1. It is hard to fix. 2. Core reviewers do not agree with the method in your posted patch. 3. If the patch is not updated by committer or reviewed by others, in OpenStack, this will be abandoned by PTL."*
* *"Reasons for abandoning code reviews in our project feature: duplicate patches, testing and invalidating the chosen approach, changes*

Table 6
Reason categories.

Category	Count	Percentage
Duplicate	595	40.78%
Already Done	*300*	*21.73%*
Suboptimal Solution	*181*	*12.40%*
Integrated	*97*	*6.65%*
Lack of Feedback	213	14.60%
Contributor Operation	146	10.01%
Wrong Branch	*47*	*3.22%*
Accidental Push	*37*	*2.54%*
Update Change	*20*	*1.37%*
Other Wrong Operation	*42*	*2.88%*
New work	123	8.43%
Incomplete/Wrong Fix	110	7.54%
Superfluous	65	4.46%
Test	64	4.39%
Branch Transfer	43	2.95%
Merge Conflict	40	2.74%
Give Up	29	1.99%
Complicated Change	28	1.90%
Other	3	0.21%

in developer and company agendas, splitting patches into several units when too large, and probably others."

* *"Sometimes we decide something was a bad idea, or someone creates a different review with a better approach. Sometimes someone contributes a review that is not ready to be merged and then does not have time to finish it or does not respond to our comments, so we abandon it."*

We totally analyzed 1459 changes in the code review systems from four open source projects. The overall distribution of reasons based on 12 categories is shown in Table 6. We found that the top three categories of the highest percentage are *"Duplicate"* (40.78%), *"Lack of Feedback"*(14.60%) and *"Contributor Operation"*(10.01%).

The 1459 changes from the studied projects could be classified into 12 categories. Duplicate*is the dominant reason.*

3.2. Category detail

In this section, we present representative change samples for each category.

3.2.1. Duplicate

It refers to changes that were abandoned because they were similar to other changes. Because duplicate changes accounted for a large proportion of changes, so we divided it into three subcategories. The subcategories are as follows:

"Already done": It refers to changes for which other similar changes had been made and reviewed already. Much effort was wasted in open source projects since there were some duplicate changes to do more or less the same things in different ways. Many changes implemented something that had already been done by other developers. In our reply emails, some comments mentioned this problem:

* *"This is duplicated with others. In OpenStack, we may not notice others' patch, so if we do the same job with others, we should abandon."*
* *"The patch fix a coding defect caused by my previous commit. But someone had done that before I did. So I abandoned it after I had discovered that."*
* *"Duplicate patch, someone else committed the same thing before me."*
* *"The mentioned change was abandoned, because the problem it was fixing was already fixed by another change I did not see. So my change was not needed anymore."*

In our data set, 21.73% of the changes were abandoned due to *Already Done*. Some representative samples are as follows:

- LibreOffice Change 32399: This was made at the same time by another contributor.
- OpenStack Change 303542: The issue had been fixed. We should abandon this one.
- Eclipse Change 86155: In the meantime this has been fixed by someone else.
- LibreOffice Change 21336: Already been done, didn't notice!

"Suboptimal solution": It refers to changes that were replaced by another change which proposed a better solution. If multiple approaches for the same feature or bug fix were provided, the best one would be merged, and all others would be abandoned. Here is a comment mentioning this problem in our reply emails:

* *"Solved a problem in one way while someone else has another solution in mind. Maybe I agree that the other solution is better."*

In our data set, 12.40% of the changes were abandoned due to *Suboptimal Solution*. Some representative samples are as follows:

- Eclipse Change 89938: Replaced with another one, which is indeed a better patch.
- Qt Change 187516: In favor of https://codereview.qt-project.org/187527.
- Qt Change 172067: Better fix: https://codereview.qt-project.org/187259.
- OpenStack Change 439769: Dims beat you to it. Abandoned. Dims was faster.

"Integrated": It refers to the changes that were a part of another change or had been integrated into another change. In our reply emails, some comments mentioned this problem:

* *"The change was already submitted as a part of a different commit."*
* *"Squashed means that the patch content has been merged with another patch, and this patch has been abandoned."*

In our data set, 6.65% of the changes were abandoned due to *Integrated*. Some representative samples are as follows:

- Eclipse Change 87860: Integrated to 872985.
- LibreOffice Change 24091: https://gerrit.libreoffice.org/#/c/24119/ included this change.
- LibreOffice Change 12166: It's now part of another patch.
- LibreOffice Change 31220: I merged two dependent commits into one. That's why it is no longer needed.
- Qt Change 182192: Integrated in another patch.
- OpenStack Change 436433: Squashed into another one.

As for duplicate changes, we did a further investigation, and the findings are as follows:

1. Many duplicate changes are only processed by the continuous integration (CI) tools(e.g., Hudson, Jenkins and Qt Sanit Bot), which are used for automatic validation in the Gerrit review system. Apart from the CI tool, there was not any other reviewer in this kind of changes. 55 of the 162 duplicate changes (34%) in our Eclipse dataset are only processed by the CI tool.
2. Many duplicate changes are abandoned by the change owners. For example, there are 162 duplicate changes in our Eclipse dataset. Among these changes, 88 changes are abandoned by the change owners, and 19 changes are abandoned because reviewers rejected the changes then the changes were abandoned by their owners. And the rest changes were abandoned by reviewers.
3. Many duplicate changes are duplicate of the changes submitted by the same owner. For example, in our Eclipse dataset, 20.37% of the changes are duplicate of another change submitted by the same author, and 8.64% of the changes are duplicate of the reviewers' changes. It indicates that some changes are abandoned because the

author submitted two similar changes. They are duplicated of each other, and one change is abandoned while the other is merged.
An intriguing finding is: reviewers may abandoned a contributor's change, and then, the reviewer submits a new change with similar function. In our dataset, we found in most of the cases the new one submitted by the reviewer was merged while the original one submitted by the contributor was abandoned. The minority of new changes were abandoned in favor of the original changes submitted by the contributor. For example, for #87120 change in Eclipse, a reviewer of this change submitted a better one (i.e., #87132 change) after this change, so the original change was abandoned in favor of the new one.

4. The time interval between two duplicate changes can be a few seconds to several years. For example, a contributor submitted # 89938 change in Eclipse, and after 34 seconds, a reviewer of this change submitted a new change (i.e., #89939 change) which is better. Finally, # 89938 change was abandoned and # 89939 change was merged. As another example, # 52524 change in Eclipse was submitted by a contributor in July, 2015. This was an incomplete change while the contributor could not improve it. In January, 2017, a reviewer of this change picked it up, then incorporated and completed it into another change(i.e., #88222) submitted in January, 2017. Finally, the # 52524 change was abandoned.

3.2.2. Lack of feedback

This could happen when reviewers added comments but contributors did not respond, or when contributor uploaded a change but nobody reviewed it. *Lack of Feedback* was a common reason why a change was abandoned. In our reply emails, some comments mentioned this problem:

* *"Dead patches, which patches still has some problems but did not be maintained by the author for a long time, and abandoned by the core reviews."*
* *"Another common reason for abandoning a review is lack of feedback, when reviewers add comments but the original uploader does not respond."*
* *"People with committer status, typically submit a patch and forget it, so the system catches it and they typically abandon it."*

In our data set, 14.60% of the changes were abandoned due to *Lack of Feedback*. Some representative samples are as follows:

- LibreOffice Change 15259: Abandoning this due to lack of response from submitter to review comments.
- OpenStack Change 400085: Abandoned due to inactivity.
- LibreOffice Change 15274: No activity on this since months, let's abandon.
- LibreOffice Change 13058: Abandoning since there are no replies from submitter.
- OpenStack Change 436775: I am abandoning because nobody wants to review cute text files.

We found that if a change was lack of feedback neither from the contributors or reviewers for a period time, the change would be abandoned by Project Team Lead (PTL). In view of this, we deduce if there is a tool to automatically detect these changes, maybe it would improve the efficiency of PTLs.

In addition, some changes were picked up after being abandoned. For example, #367629 change in OpenStack, was abandoned on Mar 8th, 2017 due to inactivity over five months from the contributor, and then the contributor picked it up and resurrected it on Mar 31th, 2017. After resurrection, the change was reviewed again, and finally, it was merged.

However, there are some changes abandoned not only due to *Lack of Feedback*, but also some other reasons (e.g., some problems needed to fix). For the *Lack of Feedback* changes in our dataset, we think the main reason leading them to be abandoned is lack of feedback. For example, the # 318930 in LibreOffice, it was abandoned due to *"A polite ping. Are you still working on this patch? There is a merge conflict, would you like to help solve that? Abandoned. Work seems abandoned. Remark patch can anytime be reopened."* It was abandoned because of not only lack of feedback, but also "merge conflict", but the main reason in this case is that the contributor could not fix the merge conflict in time, that is, lack of feedback.

3.2.3. Contributor operation

It refers to changes that were abandoned due to erroneous operations of contributors. For example, for #91766 change in Eclipse, the contributor forgot adding Change-Id to the commit message, so the change was abandoned. Then the contributor added Change-Id to the commit message and submitted it as a new change (#91767 change). There were various types of erroneous operations. We divided this category into four subcategories. The subcategories are as follows:

"Wrong branch": It refers to the changes that contributors pushed to a wrong branch. In our reply emails, some comments mentioned this problem:

* *"Pushed to wrong branch."*
* *"I accidentally submitted that patch on the wrong branch, which caused it to have a dependency on a different patch, which it was not supposed to have. I re-submitted the same patch to a different branch and without that dependency."*

In our data set, 3.22% of the changes were abandoned due to pushing to a wrong branch. Some representative samples are as follows:

- Eclipse Change 89775: Based on wrong branch. Re-pushing.
- LibreOffice Change 7922: Wrong branch, I'm sorry for the noise.
- OpenStack Change 444312: Wrong branch, I wanted to do it into stable/newton since it's a trivial fix.

"Accidental push": It refers to the changes accidentally pushed by contributors. In our reply emails, some comments mentioned this problem:

* *"I had accidentally pushed the same change patch multiple times."*
* *"Patches uploaded just because of contributor's mistake."*
* *"My patch was a local modification that I stopped working on and that should never have reached git. It was a mistake pushing it."*

In our data set, 2.54% of the changes were abandoned due to accidental push of contributors. Some representative samples are as follows:

- Eclipse Change 88996: This was an accidental push to gerrit.
- OpenStack Change 429882: Please ignore this change. This was pushed by mistake. I have uploaded a new patch with change id 439305.
- Qt Change 187927: Pushed by mistake.

"Update change": It refers to the changes that contributors accidentally created a new change when they meant to update an existing one. Some contributors did not know that some tools or commands would allow them to update the original commit. In our reply emails, some comments mentioned this problem:

* *"I had not configured my Git commit-hooks to always append a Change-Id. Using this development, I had a chain of changes and had made a fix to an earlier change in the chain."*
* *"In this particular case, the code submitted was functional but then re-worked to use a different method. However, when the update was submitted a new patch was created instead of being applied to this particular patch, and so this one was abandoned in favor of the newly created one."*
* *"It was abandoned because I accidentlly uploaded new CR instead of pushing changes to existed one."*

In our data set, 1.37% of the changes were abandoned due to *Update Change*. Some representative samples are as follows:

- LibreOffice Change 33016: Please do not submit an additional patch, when you correct the previous one, use "git commit –amend" to add a new patch set to the same patch.
- LibreOffice Change 31415: Accidentally created a new gerrit when I meant to update an existing one.
- LibreOffice Change 28419: It seems you have submitted 2 patches for the same file. If updating a patch, please remember to use "git commit –amend" to update the patch instead of making a new patch.
- LibreOffice Change 23242: Meant to update patch revision, not create separate one.
- LibreOffice Change 23197: To amend a patch that you have already submitted to Gerrit, you just need to git commit –amend locally. That way, the Change-Id line in your commit message will be the same, and thus Gerrit will recognize that it belongs to an existing change set.

From the comments in the reply emails and the change samples in our data set, we note that Gerrit code review system allows contributors to stack patches on the same functionality development. But if contributors forget to add the correct Change-Id into commit message and it would not properly stack rather than create a new change. When a developer amend the commit, the same Change-Id is used and code review system identifies the change as an update of the existing change. Change-Id is what uniquely identifies the change in code review. If contributors wanted to update one change, but they do not amend the existing change, they would create a new change, the new change should be abandoned, and contributors should resubmit to amend the old one.

"Other wrong operations": It refers to the changes related to wrong operation but did not belong to any of the above three subcategories. In our reply emails, some comments mentioned this problem:

* *"A commit id is supposed to be included in the commit, but it was not. This was abandoned."*
* *"I abandoned this patch by myself. I am new to code in LibreOffice and I just got to know the workflow of patch-updates. So this abandonment was a mistake."*

In our data set, 2.88% of the changes were abandoned due to *Other Wrong Operations*. Some representative samples are as follows:

- Eclipse Change 90632: Wrong bug number.
- Eclipse Change 85133: Forgot to add Gerrit change id in commit message.
- Eclipse Change 91697: Issues with local git checkout. Wrong files pushed.
- Qt Change 180997: Wrong message & description. Also unnecessary patch upload. Will do proper commit.
- Qt Change 180290: Incorrect commit message.

3.2.4. New work

It refers to the changes that contributors continued the work on a new change. Maybe they built a new version and created a new commit or they proposed a different approach in a new change. In our reply emails, some comments mentioned this problem:

* *"We build a new version and I created a new commit."*
* *"I abandon the request and make (in the future) a new one."*
* *"We were planning a new version of the EEF project."*

In our data set, 8.43% of the changes were abandoned due to *New Work*. Some representative samples are as follows:

- Eclipse Change 62061: Work has continued in change #64007, abandoning this one.
- Eclipse Change 87496: Will proceed in another way. Abandon this fix.
- LibreOffice Change 25032: Remove all this and propose a new way.
- LibreOffice Change 15363: Let's abandon this, I'm working on a proper fix.

3.2.5. Incomplete/wrong fix

It refers to changes that were wrong or imperfect. Such changes did not work as desired or had several issues discovered during review, or still had some problems. In our reply emails, some comments mentioned this problem:

* *"The change I proposed was abandoned because it would have created more trouble than without it."*
* *"Change does not work as desired or has too many issues."*
* *"In my case code reviews/patches are abandoned mainly when I realize that fix for bug or feature implementation is done in completely incorrect way and I need to start from the beginning with implementation."*
* *"Proposed change introduces severe problems which cannot be fixed or can only be fixed with large effort no one wants to spend."*
* *"Sometimes because the whole approach is wrong."*

In our data set, 7.54% of the changes were abandoned due to *Incomplete/Wrong fix*. Some representative samples are as follows:

- Eclipse Change 91271: Depends on previous abandoned change.
- OpenStack Change 318707: This is a wrong solution.
- Qt Change 178628: It doesn't work.
- LibreOffice Change 17596: Abandoned due to lower performance than the previous code.
- LibreOffice Change 16339: Brings more problems than it solves.

3.2.6. Superfluous

It refers to changes that were not worthwhile to make. In general, such changes made unwanted features or aimed to fix an issue while the issue was not worth to fix. For example, the bug that a change aimed to fix was marked invalid or closed. In our reply emails, some comments mentioned this problem:

* *"Invalid patches, the patches want to fix an issue, but the issue or cannot be confirmed or is not worth to fix."*
* *"Failed to persuade some reviewer that a change is worthwhile."*

In our data set, 4.46% of the abandoned changes belonged to the category of *Superfluous*. Some representative samples are:

- LibreOffice Change 30045: No need to waste time making unwanted features.
- LibreOffice Change 15551: I don't think is worthwhile, since it didn't seem to find any real problems.
- OpenStack Change 414863: The change itself doesn't bring any value to OpenStack projects. You just spend test resources and there is no profit in the change. Let's keep everything as is.

3.2.7. Test

It refers to the changes that were created for testing purposes. In our reply emails, some comments mentioned this problem:

* *"The particular change was never intended to be merged, as you may see in the patch title(DO NOT MERGE)."*
* *"This particular Gerrit patch was abandoned because it was only a test build, it was never meant to be merged. "*

In our data set, 4.39% of the changes were abandoned due to *Test*. Some representative samples are as follows:

- Eclipse Change 90268: This was just a test and didn't provide any insights. Abandoning this change.
- LibreOffice Change 23500: Was just testing the feature branch.
- OpenStack Change 430487: Was just a test.

3.2.8. Branch transfer

Code review systems allow transferring changes from one branch to another branch(e.g., cherry-picked). It would create a new change on the selected destination branch. Here is a comment mentioning this problem in our reply email:

* *"It was abandoned because it was merged (by me) previously to a different, better-suited branch. The submitter had presumably cherry-picked it to be able to do their work without having to worry about crashes."*

In our data set, 2.95% of the changes were abandoned due to *Branch Transfer*. Some representative samples are as follows:

- Eclipse Change 86142: Cherry picked on moka-master.
- LibreOffice Change 25778: Cherry-picked to private/Rosemary/change-tracking.

3.2.9. Merge conflict

It refers to the changes that caused merge conflicts. Some errors were generated when contributors merged the code. In our reply emails, some comments mentioned this problem:

* *"Because of a merge conflict, the change needs to be rewritten completely."*
* *"The reasons why this review was abandoned: Merge issues since we were merging some other code and some errors were generated."*

In our data set, 2.74% of the changes were abandoned due to *Merge Conflict*. Some representative samples are as follows:

- Eclipse Change 90565: Too many merge conflicts. I will submit a separate patch.
- LibreOffice Change 19745: Need to resolve the merge conflict, and resubmit the patch. If someone wants to deal with the merge conflict that's fine, else it can be abandoned.
- OpenStack Change 442869: Merge conflict.

3.2.10. Give up

It refers to the changes that contributors gave up on improving changes because they had no time or they could not fix the issues highlighted in comments. Sometimes, some changes introduced severe problems which could hardly be fixed or could only be fixed with a large effort so that no one wanted to do. In our reply emails, some comments mentioned this problem:

* *"Author has no more time to finish change and nobody else shows interest to finish the change."*
* *"Developer does not have the time to do it."*
* *"New people do a good job of making their patch work, but get review comments e.g. due to the user experience not being correct, and stop working."*
* *"Contributor underestimated the effort to get a change right to meet the quality requirements and gives up after negative reviews, or lacks time to complete it."*
* *"There is more work to be done than anticipated and the author loses interest."*

In our data set, 1.99% of the changes were abandoned because of *Give Up*. Some representative samples are as follows:

- Eclipse Change 84689: No time to keep iterating on this. Sorry.
- LibreOffice Change 4993: I will abandon this change as I do not have the time to work on it currently.
- LibreOffice Change 15046: I have no idea how to fix this. So let us abandon this one.
- LibreOffice Change 18056: Abandon this for now until I have time to pick it up again.

3.2.11. Complicated change

It refers to the changes that needed to be split into smaller and independent ones, which were easier to review. In our reply emails, some comments mentioned this problem:

* *"If the patch is very large and difficult to review and could benefit from being submitted in multiple patches to make reviewing the code easier."*
* *"After submitting this change 85635 I got a requested from committer to split it to several smaller reviews(to make it easier for review), I took code change (of the original patch) and committed it as 3 separate code reviews which were then accepted so I then abandoned original all-in-one review."*
* *"To make the patch easy to review and build successfully, the patch was split as two small patches and was submitted and hence this large change was abandoned."*

In our data set, 1.90% of the changes were abandoned due to *Complicated Change*. Some representative samples are as follows:

- Eclipse Change 89987: This change will be split in two individual changes.
- LibreOffice Change 23839: This has been broken up into smaller pieces and committed.
- Qt Change 179650: Splitted into several changes.

In many open source projects, smaller changes tended to be reviewed faster as they were easier for reviewers to inspect. Hence, breaking changes into small and concise changes was a better choice. It would be more likely to get reviewed faster. In addition, it also helped to reach a clean history. When contributors were tracing a bug in the code which was introduced two years ago, if the history was built from small and concise commits, it was easier to find the change which introduced the problem and understand the motivation why it was implemented in that way. This was consistent with Rigby et al. study [39] which suggested that dividing changes into smaller, independent and complete pieces may reduce the burden placed on any individual change.

3.2.12. Other

It refers to the changes that were abandoned except the above 11 categories. For example, some changes were just examples to be shared with other developers. It was just a way to easily share an idea with other team members and start a discussion. Once it had finished its task, the change itself might not be any valuable to keep around. In our reply emails, some comments mentioned this problem:

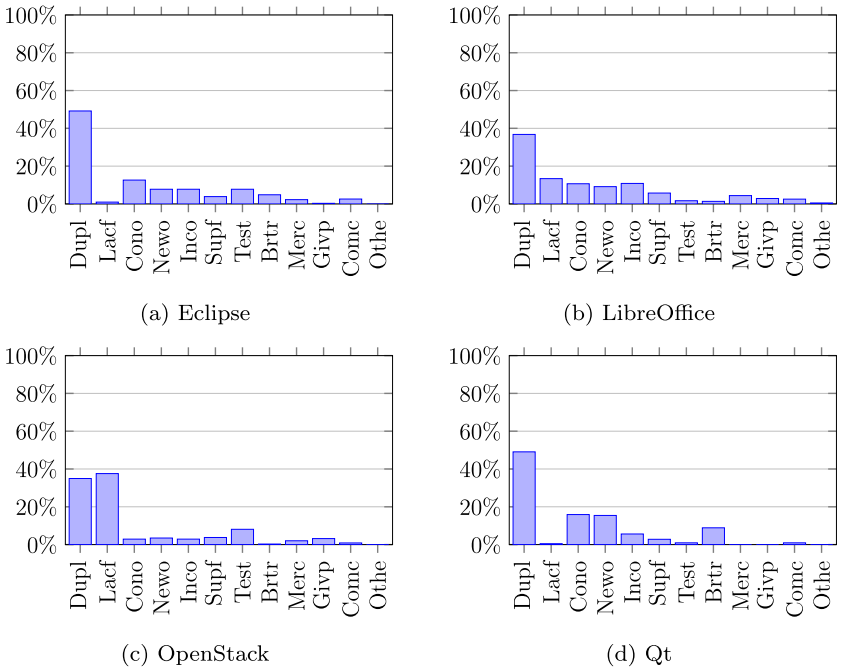
* *"I have many patches that are abandoned because I use Gerrit as a way to share prototypes quickly and express ideas in a clearer way than a giant text email. Often after they are abandoned."*
* *"We also use code review to discuss ideas and code prototype alternatives. Those are often not meant to become part of the code base from the beginning (i.e., probably what Gerrit's 'draft' feature is meant for.)"*
* *"Sometimes I am using the code review tool just for sharing the information and saving draft work."*

In our data set, 0.21% of the changes belonged to this category. Here is a sample:

- LibreOffice Change 32730: Not intending to submit this, but seems like a good place to keep work I've spent too much time on.

4. Distribution

In order to answer RQ2, we analyzed the frequency distribution of abandoned changes across the rationales and projects. All the four open source projects were analyzed, and the frequency distribution of every project is depicted in Fig. 1 (The acronyms in subfigures (X-axes) are



(a) Eclipse (b) LibreOffice

(c) OpenStack (d) Qt

Fig. 1. Distribution of reason categories for each project.

from Table 2). We observed similar frequency distribution of reason categories across four studied open source projects. For instance, *Duplicate* changes account for the majority in most studied projects.

To check differences in frequency distributions is statistically significant, we further apply the Wilcoxon signed-rank test [53] at 95% significance level on 12 paired categories. In the 6 Wilcoxon signed-rank tests of the four open source projects, all the p-values were greater than 0.05 (min: 0.20, max: 0.91, median: 0.63). That is, the differences in the frequency distribution of the reason categories across projects were not significant.

However, although the results of Wilcoxon signed-rank tests show the differences were not significant, we could see there is an unusual category (i.e., *Lack of Feedback*) whose values are very low in Qt and Eclipse project but high in OpenStack project. As for this phenomenon, qualitative analysis is required. By our detailed investigation, the following reasons could explain this question:

1) In Qt project, there is a status of change called "Deferred", so almost all changes of "Lack of Feedback" and "Give Up" belong to this one, such as the following changes:

- Qt Change 222726: Deferred. Most of the skipped tests have been fixed, and some new are failing. I don't have time to look into it right now, though.
- Qt Change 154637: Automatic cleanup after prolonged inactivity.
- Qt Change 232840: Deferred. I don't have time to look at this anytime soon.
- Qt Change 233844: I am working on other stuff right now, but I'd like to come back to this the next time I add a ListView test.
- Qt Change 232999: Deferred. I don't have the energy to fight this, if someone wants to, please go ahead.

Such changes were deferred because contributors did not respond to the comments by reviewers for a long time or contributors could not fix the problem or they have no time to fix it. There is an automatic cleanup system that abandons changes (as determined by seeing no activity at all within three months). Besides, the percentage of *Lack of Feedback* is not zero in our Qt dataset. It is because there is one change (Qt # 161878) in Qt project (in our dataset) that no reviewers wanted to review this one spite that it was assigned with three reviewers (except for the owner and an automatic verification reviewer). This case also belongs to *Lack of Feedback*.

2) The percentage of *Lack of Feedback* category is so low in our Eclipse dataset while it is high in our OpenStack dataset. One possible explanation is that OpenStack is a larger project where the number of changes is over 600,000 than Eclipse where the number of changes is over 100,000. So we may speculate that the percentage of core developers in OpenStack are relatively less than that in Eclipse. Another plausible explanation is that these changes in OpenStack will be abandoned by Project Team Lead (PTL) if contributors do not respond to the comments for a long time.

> *The frequency distributions of reason categories shared similar trends across studied projects.*

5. Duration

In order to answer RQ3, we investigated the relationship between categories and time-to-abandonment. We measured the time interval by the number of days from `Created` time when a change was submitted and `Abandoned` time when a change was abandoned. Table 7 shows

Table 7
Change abandoning durations in terms of days.

Reason categories	Min	Max	Mean	Median
Duplicate	0.0005	848.2421	34.5591	2.6662
Lack of Feedback	0.0028	744.8853	191.8712	161.7100
Contributor Operation	0.0003	111.7033	3.6336	0.0347
New Work	0.0049	408.3261	40.3557	4.2419
Incomplete/Wrong Fix	0.0032	590.1685	29.5627	2.3161
Superfluous	0.0043	939.9169	46.0039	1.8521
Test	0.0018	301.8160	13.6013	0.4729
Branch Transfer	0.0361	178.8673	43.7281	17.8710
Merge Conflict	0.0203	68.6853	11.9476	0.3403
Give Up	1.7536	423.1277	108.1894	97.6660
Complicated Change	0.0104	166.1683	24.0878	4.0059
Other	0.1419	27.6982	9.9646	2.0536

Table 8
Numbers of change abandoned within various durations.

Reason category	Duration	Count	Proportion
Duplicate	within a month	465	78.15%
	within a year	584	98.15%
	more than a year	11	1.85%
Lack of Feedback	within a month	3	1.41%
	within a year	192	90.14%
	more than a year	21	9.86%
Contributor Operation	within a month	141	96.58%
	within a year	146	100.00%
	more than a year	0	0.00%
New work	within a month	91	73.98%
	within a year	121	98.37%
	more than a year	2	1.63%
Incomplete/Wrong Fix	within a month	88	80.00%
	within a year	108	98.18%
	more than a year	2	1.82%
Superfluous	within a month	56	86.15%
	within a year	61	93.85%
	more than a year	4	6.15%
Test	within a month	55	85.94%
	within a year	64	100.00%
	more than a year	0	0.00%
Branch Transfer	within a month	23	53.49%
	within a year	43	100.00%
	more than a year	0	0.00%
Merge Conflict	within a month	35	87.50%
	within a year	40	100.00%
	more than a year	0	0.00%
Give Up	within a month	7	24.14%
	within a year	28	96.55%
	more than a year	1	3.45%
Complicated Change	within a month	23	82.14%
	within a year	28	100.00%
	more than a year	0	0.00%
Other	within a month	3	100.00%
	within a year	3	100.00%
	more than a year	0	0.00%

the minimum, maximum, mean and the median of days that changes took up to be abandoned.

We noticed that the minimum period for all changes in our studied projects was just a few minutes between `Created` time and `Abandoned` time. The maximum was close to three years. The top five categories of maximum were *Superfluous, Duplicate, Lack of Feedback, Incomplete/Wrong Fix* and *Give Up*, whose maximum surpassed 420 days. In terms of mean time, *Lack of Feedback, Give Up, Superfluous, Branch Transfer* and *New Work* took the longest time to be abandoned (more than 40 days).

Table 8 gives further breakdowns of durations into a month, year, and more than a year for each category. In addition, we manually investigated these changes that were abandoned after many years, and found most of these changes were *Lack of Feedback* changes. It is meaningless to calculate its duration. So we removed the category of *Lack of Feedback*, and then calculated the following information: 42.86% of the changes were abandoned within a day, 62.28% of the changes were abandoned within a week, 79.21% of the changes were abandoned within a month, 98.39% of the changes were abandoned within a year, and only 1.61% of the changes were abandoned over one year.

6. Discussion

6.1. Recommendation for submitting high-quality changes in practice

From the results of our empirical study, we recommend developers to submit code changes according to the following aspects:

1. **Before submitting a change, developers should make sure the issue is worthwhile to fix or the feature is useful.** Automated techniques should be proposed to predict whether a code change is worthwhile to fix before it goes to the code reviewer. Such a tool can help code reviewers to review low-quality or meaningless changes, to save their effort. Fan et al. [16] proposed 34 features to predict whether a code change will be merged. Our work is consistent with Fan et al.'s study.
2. **Before submitting a change, developers could investigate whether this issue has been solved already to avoid pushing duplicate changes.** *Duplicate* is the dominant reason why changes are abandoned. Table 6 shows that *Duplicate* changes are the majority of the studied changes, accounting for 40.78% in the studied projects. It takes a large percentage, so we divided it into three subcategories (i.e., *Already Done, Suboptimal Solution, Integrated*). The characteristics of the three sub-categories are different: changes of *Already Done* are easier to detect before submitting them than other two sub-categories, because the change title and commit message of a change could summarize the function of the change, so developers only need to compare their change title and commit message with committed changes. However, for changes of *Suboptimal Solution*, it is difficult for developers to verify whether their changes are better than other changes, since developers need to have a good understanding of other changes. For changes of *Integrated*, it is impossible to detect because it is abandoned just due to being integrated which happens after being submitted.
   In general, much effort is wasted in open source projects due to duplicate changes. There are many studies on duplication in the past. Hordijk et al. [20] thought that duplication of source code is an essential factor suspected to affect the quality of systems, so they conducted a literature survey to investigate how duplication affects quality. To summarize, this finding indicates that code review systems(e.g., Gerrit) need to avoid duplicated implementation, and before this, contributors had better investigate whether the issue has been solved already before submitting a change.
   To solve the problem, in the future, we will do a tool for code review systems with which developers could investigate whether the issue they want to solve are already done by measuring text (e.g., description of the changes) or/and code (e.g., diff files) similarity between the new change and the changes in the historical repository.
3. **Before pushing a change, developers should make sure the change addresses only one issue, which is easy to review.** *Complicated Change* accounts for 1.90% of the studied changes. *Complicated Change*, accurately speaking, are changes that include unnecessary changes or solve not only one problem. Hence, smaller changes are easier to review. Our conclusion is consistent with the work of Weißgerber et al. [52]. They found that patch size impacts the likelihood of the patch to be accepted: for small patches with at most four lines changed, the possibility to get accepted are higher, while for very large patches they are less likely to be accepted. Bosu et al. [11] drew a similar conclusion. They found reviewers take more time and effort to inspect changes with more files. This might bring more problems to understand the change, causing lower usefulness. Other studies also reported that change size is a good indicator of change acceptance [24,49,52]. Baysal et al. [10] found that large changes tend to have more revisions than small changes in the code review processes of WebKit and Blink projects. The size of change gives an estimate of how long a review of this change may take, generally, in many projects, smaller changes tend to be reviewed faster because they are easier for review.
   To decompose a composite change, Barnett et al. [6] developed a static analysis technique for decomposing changesets. However, with this technique developers are still burdened with the task of understanding and applying partitioned changesets to the original version [19]. Guo and Song [19] developed a change decomposition technique, called CHGCUTTER. It could decompose a composite change and identifying a subset of related atomic changes.
4. **When developers push a change, they should be more careful, e.g., push to a correct branch, remember to add Change-Id to commit message, and keep the commit message**

description consistent with change function. *Contributor Operation* is one of the major categories. Table 6 shows that *Contributor Operation* changes take up 10.01% of the reported changes in the studied projects. Some changes are abandoned due to contributors' wrong operation, such as pushing to a wrong branch, accidental push and so on. This finding indicates that before pushing a change, contributors should carefully scrutinize the change. For example, do not forget to add the Change-Id to the commit message, make sure choosing the correct branch, deleting redundancy files, and using git command to update an existing change rather than create a new one. Otherwise, those wrong operations would lead the changes to be abandoned.

In addition, Gerrit requires that every change must have a unique Change-Id. In general, there are two situations leading to missing Change-Id in commit message: (1) Developers forget to add Change-Id in commit message; (2) Developers add Change-Id in commit message, but the Change-Id is not in the last paragraph of the commit message. For the first case, if the commit message of a change does not contain a Change-Id, developers have to update its commit message and insert a Change-Id. For the second case, the Change-Id must be contained in the last paragraph of commit message. Otherwise, the Change-Id could be mistaken as other part of the commit message (e.g., change title) by the code review system. If this happens, developers have to update the commit message and move the Change-Id into the last paragraph.

By default, Gerrit will prevent pushing for review if no Change-Id is provided. However, repositories can be configured to allow commits without Change-Ids in the commit message by setting "Require Change-Id in commit message" to "FALSE". Our finding shows that changes without Change-Id tend to be abandoned. Thus, to avoid this situation, a standard 'commit-msg' hook is provided by Gerrit and can be installed in the local Git repository to automatically generate and insert a Change-Id line when committing a change. Hence, developers are suggested to install it. In such a case, if a Change-Id line is not present in the commit message, Gerrit will automatically generate its own Change-Id and display it on the web. This line can be manually copied and inserted into an updated commit message if additional revisions to a change are required.

Moreover, developers could use tools which can automatically generate commit messages to help them write high-quality commit messages [12,22,30,31]. For example, Cortés-Coy et al. [12] proposed an approach, coined as *ChangeScribe*, to automatically generate editable commit message for a given change-set. Jiang and McMillan [23] proposed a method that can generate short commit messages that convey the key ideas of commits. Liu et al. [31] proposed a simple approach called *NNGen* which leverages the nearest neighbor (NN) algorithm to generate commit messages.

5. **After pushing a change, developers should reply to reviewers' comments in time.** *Lack of Feedback* changes take the longest median time and average time to be abandoned. In the review process, some reviewers' comments may describe how to improve the change. Contributor of the change should modify the change according to the review comments. However, such comments may be lack of feedback, and in this case, the change is likely to be abandoned. This can happen when reviewers add some comments but contributors do not respond. In addition, lack of feedback also happens when a change is submitted but no reviewers give response. For *Lack of feedback* changes, reviewers spend long time to decide the fate of the change, until there is long time that no feedback to the change, and the change will be abandoned. In the LibreOffice project, it generally takes four weeks to abandon a change which has no feedback. We notice that *Lack of Feedback* changes take about 190 days and 160 days to be abandoned in terms of average time and median time respectively (see Table 7).

A code review system should have a module for automatically reminding contributors and reviewers when there is no activity. For example, the code review system could regularly(e.g., one month) send contributors and reviewers email to remind them to continue the work in the code review system. In addition, reviewers and contributors should regularly see the status of changes they are working on.

6.2. Similar study

Tao et al. [47] did similar work. They analyzed 300 patches from Eclipse and Mozilla by manually inspecting their patch review comments to understand why they were rejected. They summarized 12 categories of reasons as shown in Table 9. However, there are so many differences between our work:

1) The research object is different. The patches they studied are bug reports while our research object is changes in code review systems. Bug reports are mainly aimed to fix bugs, but changes in code review not only fix bugs, but also enhance new features or modify documentation and so on.
2) Data source is different. They collected data from Bugzilla platform while we collected data from Gerrit platform. Bugzilla is a bug report tool, while Gerrit is a code review tool. In addition, there is no automatic verification mechanism in Bugzilla, but in Gerrit code review system, there is automatic verification mechanism. Verification is taken on the process of code compiling, unit tests etc. Verification is usually done by an automated build server rather than a person, such as a Jenkins/Hudson build server.
3) As for duplicate changes, we did more in-depth investigation and summarized four findings(listed in Section 3.2.1) while Tao et al. did not.
4) Tao et al. summarized twelve categories as shown in Table 9, while we summarized twelve categories as shown in Table 2 and seven subcategories in total as shown in Tables 3 and 4.
5) There are both twelve categories in Tao et al.'s paper and our paper. There are three categories are the same, that is, *Duplicate, Incomplete/Wrong Fix* and *Complicated Change*. They are the same because they both focus on code changes or patches, which aim to improve the quality of software. The rest nine categories are different. The explicit description is as follows:

(1) The different nine categories in Tao et al.'s work:
   - Compilation errors: there is no automatic verification mechanism in Bugzilla while there is such mechanism in Gerrit. It could lead to compilation errors without automatically verification mechanism.
   - Suboptimal solution: this is included in *Duplicate* in our paper.
   - Including unnecessary changes: this is included in *Complicate Change* in our paper.
   - Bad naming: this is included in *Incomplete/Wrong Fix* in our paper.
   - Missing documentation: this is included in *Contributor Operation* in our paper.
   - Introducing new bugs: this is included in *Incomplete/Wrong Fix* in our paper.
   - Inconsistent of misleading documentation: this is included in *Contributor Operation* in our paper.
   - Violating coding style guidelines: this is included in *Incomplete/Wrong Fix* in our paper.
   - Test failures: this is included in *Incomplete/Wrong Fix* in our paper.

(2) The new findings of category in our work: *Lack of Feedback, New work, Superfluous, Test, Branch Transfer, Give Up* and *Other*.

6.3. Threats to validity

Internal validity. The classification process in RQ1 involves manual examination. The classification process was conducted by the first author and one graduate who are not involved in the code review process

Table 9
Patch-rejection reasons.

Patch-rejection reason	Example of patch review comments	Project-BugID
Compilation errors	I will not review a patch that causes errors in my workspace. As said before: make sure you have API tools enabled and a R3.5 baseline set.	Eclipse-78522
Test failures	The provided patch causes about 20 tests to fail. Either the change really breaks something, or it has side-effects that need the tests to be changed, that means that it changes the expected behavior of the generator.	Eclipse-331875
Introducing new bugs	The patch fixes the CCE but introduces a new bug: the returned key string is wrong in the normal case i.e., it includes the '" at the end.	Eclipse-247012
Inconsistent or misleading documentation	The note is unclear. "As per..." sounds like we follow the spec. But since we don't, this should be stated explicitly ("Note: This deviates from JLS314.3..."). Furthermore, it's confusing that you use differing terms "anonymous type" and "anonymous inner classes" for the same thing.	Eclipse-339337
Suboptimal solution	I honestly don't want all this complexity for this user pref ... Much easier will be to add a link from the Email Preferences tab pointing to email-related user prefs once bug 589138 is implemented	Mozilla-589128
Duplication	What I now don't like is that we have two methods which almost do the same thing but have different names: *#packageChanged()* and *#getPackageStatus(packName)*.	Eclipse-393161
Including unnecessary changes	Removed this unnecessary check from *#getNextElseOffset: if(then == null)return -1;*	Eclipse-377141
Incomplete fix	I couldn't test this patch, as it seems to be missing the change to browser.inc that adds secondaryToolbarButtons.	Mozilla-877335
Violating coding style guidelines	Per our Bugzilla guideline, we never leave —*if (*— alone on its own line.	Mozilla-637981
Bad naming	*JavaCompareUtilities.getActiveEditor(IEditorPart)* has wrong name as it simply works with the given part (doesn't matter if active or not).	Eclipse-260531
Missing documentation	In the OverviewRuler class Javadoc I would mention that it uses non-saturated colors unless *setUseSaturatedColorPreference(...)* gets called.	Eclipse-341808
Patch size too large	Here is a patch smaller than 250 line.	Eclipse-344125

of the studied systems. The results of manual classification by a domain expert might be different.

In addition, we analyzed the reasons why changes are abandoned from review comments and survey. There might be other suitable materials to analyze the reasons why changes are abandoned. In the future, we plan to analyze more materials to summarize the reasons.

Moreover, the survey brings some noise, e.g., some developers from the response emails did not answer our question. In order to reduce this threat, we have manually removed such emails, and they were excluded from the analysis why changes are abandoned.

External validity. We pick up four mainstream open source projects in Gerrit, but they cannot be on behalf of all code review systems and projects. Because the four studied systems are open source projects, thus our results may not generalize commercial code review systems. In the future, we are going to analyze more code review systems and projects to reduce the threat.

7. Related work

In this section, we briefly introduce some related research studies. First, we introduce some previous empirical studies on code review. Next, we describe studies on the influencing factors of accepted changes.

Empirical study on code review. There are some empirical studies on code review. Thongtanunam et al. [50] conducted an empirical investigation to identify how an open source software developer's reputation affects the outcome of his/her code review requests. Their result suggested that core developers receive quicker first feedback on their review request, complete the review process in shorter time, and are more likely to have their code accepted into the project codebase. McIntosh et al. [33] conducted an empirical study on the relationship between post-release defects (a popular proxy for long-term software quality) and code review coverage, participation, and expertise. They found that code review coverage, participation, and expertise share a significant link with software quality. Their work confirmed that poorly-reviewed code has a negative impact on software quality in large systems which use modern code review tools. To improve code review effectiveness and quality in projects, Bosu et al. [11] conducted an empirical study to identify factors that lead to useful code reviews. Ruangwan et al. [41] empirically studied 230,090 patches and found that a large number of patches (i.e., 16%–66%) have at least one invited reviewer who did not respond to the review invitation.

Baysal et al. [10] conducted an empirical study to investigate technical and non-technical factors influencing modern code review. Their findings suggested that non-technical factors can significantly impact code review outcomes. To evaluate the impact that characteristics of modern code review practices have on software quality, Thongtanunam et al. [48] conducted an empirical study to investigate defective and clean source code files. They found that both future-defective files and risky files tend to be reviewed less rigorously than their clean counterparts. Also, they found that the concerns addressed during the code reviews of both defective and clean files tend to enhance evolvability. Kononenko et al. [28] conducted a study to explore the code review practices of a large open source project in order to understand the developers' perception of code review quality. They surveyed 88 core contributors to the Mozilla project. Their qualitative analysis of the survey responses provided insights into the factors that affect the time and decision of a review, and the challenges developers face, when conducting code review tasks. They found that code review quality is mainly associated with thoroughness of the feedback, the reviewers' familiarity with the code and the perceived quality of code itself.

Bacchelli and Bird [4] conducted an empirical study to explore the motivations, challenges, and outcomes of modern code review. They found that, although finding defects is still the main motivation for review, the output of reviews brings fewer defects than expected. The review activities are also used to provide additional benefits such as increasing team awareness, knowledge transfer and so on. Kononenko et al. [29] did an empirical study to explore code review quality by investigating various factors. Their findings suggested that developer participation in discussions on bug fixes and developer-related characteristics (e.g., review experience and review loads) were promising predictors of code review quality. Different from these studies, we focus on the issue why changes are abandoned in code review system.

Research on the influencing factors of accepted changes. Rigby et al. [39] analyzed 2603 patches of Apache HTTP server project and they found that small, complete and independent patches are more likely to be accepted. Weißgerber et al. [52] performed a case study on email archives of two open source projects. They found that small patches (at most four lines changed) have a higher possibility to get accepted while the very large patches get significantly lower. They also found that while patch size is an outstanding factor on the chances of patch being accepted, it does not significantly affect the duration until the patch is accepted. Jiang et al. [24] studied the relation of patch characteristics

with the possibility of patch acceptance and the time taken for patches to be merged into the codebase. They took Linux kernel as an example. They found that patches developed by more experienced contributors are easier to be accepted and faster reviewed and integrated. Baysal et al. [9] conducted an empirical study of the code review process for WebKit which is a large open source project. Their findings indicated that non-technical factors could significantly impact the outcomes of code review. It is worth mentioning that they found most influential factors on patch acceptance as well as on review time are the organization that a patch contributor is affiliated with and their level of participation. In short, the more active role a contributor decides to play, the faster and more likely his contribution will be integrated into the codebase.

Jeong et al. [21] observed the review process of two open source projects, the Mozilla Core and the Firefox. Then they proposed two improvements, one is to predict whether a given patch is acceptable and another is to suggest reviewers for a patch. Another work finished by Baysal et al. [8] found that the patches submitted by casual contributors are disproportionately more likely to be abandoned compared with core contributors. They suggested that patches from casual contributors should receive extra care in order to both ensure quality and encourage future community contributions. Rigby and Storey [40] summarized six technical reasons and six non-technical reasons that changes are abandoned. Similarly, Rigby et al. [36] did another work to examine the reasons why commits on GitHub pull requests are rejected. Different from the studies listed above, we conduct a large-scale study to investigate the reasons why changes are abandoned in Gerrit.

8. Conclusion and future work

The paper studied the reasons why changes were abandoned in code review system. We manually inspected 1459 changes of four open source projects and utilized open card sorting to categorize reasons. We further investigated the frequency distribution of different categories across different projects. Moreover, we investigated the relationship between categories and duration of abandoned changes. In addition, we recommended five aspects for contributors to submit high-quality changes in practice.

In the future, we plan to use text mining and machine learning techniques to automatically classify reasons that changes are abandoned, in order to reduce the manual effort in categorizing reasons. We also would like to investigate more code review systems and changes. Moreover, we plan to investigate the approaches that could help developers to avoid duplicate changes.

References

[1] A.F. Ackerman, L.S. Buchwald, F.H. Lewski, Software inspections: an effective verification process, Softw. IEEE 6 (3) (1989) 31–36.
[2] A.F. Ackerman, R.G. Ebenau, R.G. Ebenau, Software Inspections and the Industrial Production of Software, Elsevier North-Holland, Inc., 1984.
[3] Ö. Albayrak, D. Davenport, Impact of maintainability defects on code inspections, in: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, 2010, p. 50.
[4] A. Bacchelli, C. Bird, Expectations, outcomes, and challenges of modern code review, in: International Conference on Software Engineering, 2013, pp. 712–721.
[5] V. Balachandran, Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation, in: Software Engineering (ICSE), 2013 35th International Conference on, IEEE, 2013, pp. 931–940.
[6] M. Barnett, C. Bird, J. Brunet, S.K. Lahiri, Helping developers help themselves: automatic decomposition of code review changesets, in: Proceedings of the 37th International Conference on Software Engineering-Volume 1, IEEE Press, 2015, pp. 134–144.
[7] G. Bavota, B. Russo, Four eyes are better than two: on the impact of code reviews on software quality, in: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, 2015, pp. 81–90.
[8] O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, The secret life of patches: a firefox case study, in: Reverse Engineering (WCRE), 2012 19th Working Conference on, IEEE, 2012, pp. 447–455.
[9] O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, The influence of non-technical factors on code review, in: Reverse Engineering (WCRE), 2013 20th Working Conference on, IEEE, 2013, pp. 122–131.
[10] O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, Investigating technical and non-technical factors influencing modern code review, Empir. Softw. Eng. 21 (3) (2016) 932–959.
[11] A. Bosu, M. Greiler, C. Bird, Characteristics of useful code reviews: an empirical study at microsoft, in: Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, 2015, pp. 146–156.
[12] L.F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, D. Poshyvanyk, On automatically generating commit messages via summarization of source code changes, in: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, IEEE, 2014, pp. 275–284.
[13] A. Dunsmore, M. Roper, M. Wood, Practical code inspection techniques for object-oriented systems: an experimental comparison, IEEE Softw. 20 (4) (2003) 21–29.
[14] F. Ebert, F. Castor, N. Novielli, A. Serebrenik, Confusion detection in code reviews, in: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE, 2017, pp. 549–553.
[15] M.E. Fagan, Advances in software inspections to reduce errors in program development, IBM Syst. J. 15 (3) (1976) 182–211.
[16] Y. Fan, X. Xia, D. Lo, S. Li, Early prediction of merged code changes to prioritize reviewing tasks, Empir. Softw. Eng. (2018) 1–48.
[17] J.L. Fleiss, Measuring nominal scale agreement among many raters., Psychol. Bull. 76 (5) (1971) 378.
[18] J.M. Gonzalez-Barahona, D. Izquierdo-Cortazar, G. Robles, A. del Castillo, Analyzing gerrit code review parameters with bicho, Electronic Communications of the EASST, 2014.
[19] B. Guo, M. Song, Interactively decomposing composite changes to support code review and regression testing, in: Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual, vol. 1, IEEE, 2017, pp. 118–127.
[20] W. Hordijk, L. Ponisio, R.J. Wieringa, Harmfulness of code duplication-a structured review of the evidence, in: 13th International Conference on Evaluation and Assessment in Software Engineering, EASE 2009, British Computer Society, 2009.
[21] G. Jeong, S. Kim, T. Zimmermann, K. Yi, Improving code review by predicting reviewers and acceptance of patches, in: Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO 2009-006), 2009, pp. 1–18.
[22] S. Jiang, A. Armaly, C. McMillan, Automatically generating commit messages from diffs using neural machine translation, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, 2017, pp. 135–146.
[23] S. Jiang, C. McMillan, Towards automatic generation of short summaries of commits, in: Proceedings of the 25th International Conference on Program Comprehension, IEEE Press, 2017, pp. 320–323.
[24] Y. Jiang, B. Adams, D.M. German, Will my patch make it? And how fast? Case study on the linux kernel, in: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, IEEE, 2013, pp. 101–110.
[25] P.M. Johnson, Reengineering inspection, Commun. ACM 41 (2) (1998) 49–52.
[26] E. Kantorowitz, T. Kuflik, A. Raginsky, Estimating the required code inspection team size, in: Software-Science, Technology & Engineering, 2007. SwSTE 2007. IEEE International Conference on, IEEE, 2007, pp. 104–115.
[27] N. Kitagawa, H. Hata, A. Ihara, K. Kogiso, K. Matsumoto, Code review participation: game theoretical modeling of reviewers in gerrit datasets, in: Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering, ACM, 2016, pp. 64–67.
[28] O. Kononenko, O. Baysal, M.W. Godfrey, Code review quality: how developers see it, in: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, IEEE, 2016, pp. 1028–1038.
[29] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, M.W. Godfrey, Investigating code review quality: do people and participation matter? in: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, 2015, pp. 111–120.
[30] M. Linares-Vásquez, L.F. Cortés-Coy, J. Aponte, D. Poshyvanyk, Changescribe: a tool for automatically generating commit messages, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2, IEEE, 2015, pp. 709–712.
[31] Z. Liu, X. Xia, A.E. Hassan, D. Lo, Z. Xing, X. Wang, Neural-machine-translation-based commit message generation: how far are we? in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 373–384.
[32] S. McIntosh, Y. Kamei, B. Adams, A.E. Hassan, The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and itk projects, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 192–201.
[33] S. McIntosh, Y. Kamei, B. Adams, A.E. Hassan, An empirical study of the impact of modern code review practices on software quality, Empir. Softw. Eng. 21 (5) (2016) 2146–2189.
[34] M. Mukadam, C. Bird, P.C. Rigby, Gerrit software code review data from android, in: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, IEEE, 2013, pp. 45–48.
[35] N. Ramasubbu, R. Subramanyam, S. Mithas, M.S. Krishnan, On the value of code inspections for software project management: an empirical analysis, in: AMCIS 2006 Proceedings, 2006, p. 459.
[36] P.C. Rigby, A. Bacchelli, G. Gousios, M. Mukadam, A mixed methods approach to mining code review data: examples and a study of multi-commit reviews and pull requests, The Art and Science of Analyzing Software Data. Morgan Kaufmann, 2015.
[37] P.C. Rigby, C. Bird, Convergent contemporary software peer review practices, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 202–212.
[38] P.C. Rigby, D.M. German, A Preliminary Examination of Code Review Processes in Open Source Projects, Technical Report, Technical Report DCS-305-IR, University of Victoria, 2006.

[39] P.C. Rigby, D.M. German, M.A. Storey, Open source software peer review practices: a case study of the apache server, in: International Conference on Software Engineering, 2008, pp. 541–550.
[40] P.C. Rigby, M.A. Storey, Understanding broadcast based peer review on open source software projects, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 541–550.
[41] S. Ruangwan, P. Thongtanunam, A. Ihara, K. Matsumoto, The impact of human factors on the participation decision of reviewers in modern code review, Empir Softw. Eng. (2018) 1–44.
[42] C.B. Seaman, V.R. Basili, An empirical study of communication in code inspections, in: Proceedings of the 19th international conference on Software engineering, ACM, 1997, pp. 96–106.
[43] J. Shimagaki, Y. Kamei, S. Mcintosh, A.E. Hassan, N. Ubayashi, A study of the quality-impacting practices of modern code review at sony mobile, in: Ieee/acm International Conference on Software Engineering Companion, 2017, pp. 212–221.
[44] F. Shull, C. Seaman, Inspecting the history of inspections: an example of evidence-based technology diffusion, IEEE Softw. 25 (1) (2008) 88–90.
[45] R.R. Souza, C.F. Chavez, R.A. Bittencourt, Patch rejection in firefox: negative reviews, backouts, and issue reopening, J. Softw. Eng. Res.Dev. 3 (1) (2015) 1–22.
[46] D. Spencer, Card Sorting: Designing Usable Categories, Rosenfeld Media, 2009.
[47] Y. Tao, D. Han, S. Kim, Writing acceptable patches: an empirical study of open source project patches, in: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, 2014, pp. 271–280.
[48] P. Thongtanunam, S. McIntosh, A.E. Hassan, H. Iida, Investigating code review practices in defective files: an empirical study of the qt system, in: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, 2015, pp. 168–179.
[49] P. Thongtanunam, S. McIntosh, A.E. Hassan, H. Iida, Review participation in modern code review, Empir. Softw. Eng. 22 (2) (2017) 768–817.
[50] P. Thongtanunam, C. Tantithamthavorn, R.G. Kula, N. Yoshida, H. Iida, K.-i. Matsumoto, Who should review my code? A file location-based code-reviewer recommendation approach for modern code review, in: Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, IEEE, 2015, pp. 141–150.
[51] L.G. Votta, Does every inspection need a meeting? in: Acm Sigsoft Symposium on Foundations of Software Engineering, 1993, pp. 107–114.
[52] P. Weißgerber, D. Neu, S. Diehl, Small patches get in!, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, ACM, 2008, pp. 67–76.
[53] F. Wilcoxon, Individual comparisons by ranking methods, Biom.Bull. 1 (6) (1945) 80–83.