On the Relationship between Refactoring Actions and Bugs: A Differentiated Replication

Massimiliano Di Penta
dipenta@unisannio.it
University of Sannio
Benevento, Italy

Gabriele Bavota
gabriele.bavota@usi.ch
Università della Svizzera italiana
Lugano, Switzerland

Fiorella Zampetti
fiorella.zampetti@unisannio.it
University of Sannio
Benevento, Italy

ABSTRACT

Software refactoring aims at improving code quality while preserving the system's external behavior. Although in principle refactoring is a behavior-preserving activity, a study presented by Bavota *et al.* in 2012 reported the proneness of some refactoring actions (*e.g.,* pull up method) to induce faults. The study was performed by mining refactoring activities and bugs from three systems. Taking profit of the advances made in the mining software repositories field (*e.g.,* better tools to detect refactoring actions at commit-level granularity), we present a differentiated replication of the work by Bavota *et al.* in which we (i) overcome some of the weaknesses that affect their experimental design, (ii) answer the same research questions of the original study on a much larger dataset (3 *vs* 103 systems), and (iii) complement the quantitative analysis of the relationship between refactoring and bugs with a qualitative, manual inspection of commits aimed at verifying the extent to which refactoring actions trigger bug-fixing activities. The results of our quantitative analysis confirm the findings of the replicated study, while the qualitative analysis partially demystifies the role played by refactoring actions in the bug introduction.

CCS CONCEPTS

• **Software and its engineering → Software reliability**; *Designing software.*

KEYWORDS

refactoring, bug introduction, mining software repositories

ACM Reference Format:
Massimiliano Di Penta, Gabriele Bavota, and Fiorella Zampetti. 2020. On the Relationship between Refactoring Actions and Bugs: A Differentiated Replication. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409695

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409695

1 INTRODUCTION

Software refactoring has been extensively studied by the research community, through empirical studies investigating how and why developers perform refactoring [32, 37, 39, 43, 48, 49], how refactoring relates with other development tasks (*e.g.,* merge conflicts [35]), with software quality indicators (*e.g.,* quality metrics) [5, 17, 45, 46], and with developers' productivity [36]. Some studies (*e.g.,* Kim *et al.* [32]) indicated that often developers are concerned about performing refactoring activities as it may cause the introduction of bugs.

The relationship between refactoring and bugs has been the subject of several studies, that analyzed software repositories to understand the extent to which refactoring activities introduce bugs [8, 24, 50]. Weißgerber and Diehl [50] studied the correlation between refactoring activities and bug reports opened in the subsequent days, finding no strong correlation. However, their study did not link refactoring activities in a specific file with bug-fixes performed on that same file.

In a previous work, some of the authors[1] [8] presented a study overcoming this limitation, showing that refactoring actions involving hierarchies (*e.g.,push-down method*) induce bug-fixing commits more frequently than other refactoring types. They used the RefFinder [40] tool to create a dataset of 12,922 manually-validated refactoring actions, detected comparing subsequent releases (63 in total) of three Java systems. By comparing releases, Bavota *et al.* [8] assumed that a specific refactoring was performed on a file F_j between releases R_i and R_{i+1} of a given system, while the exact refactoring-related commit was unknown. Then, by mining the change history of the three systems, the authors identified bug-fixing commits by linking commit messages and issue tracker data using a keyword-based approach [25] (*e.g.,* "*fixed issue #ID*", where ID was the id of an issue on the issue tracker of the mined system). Finally, for each bug-fixing commit, they identified its fix-inducing commits using the SZZ algorithm [44]. Using such data, Bavota *et al.* assumed that a refactoring action performed on file F_j between R_i and R_{i+1} induced a fix if a bug-inducing commit c identified by the SZZ was performed on F_j between R_i and R_{i+1}. Thus, there is a strong assumption made in the experimental design: Since the refactoring actions were captured between releases, it is not possible to know whether the refactoring was actually implemented in the bug-inducing commit c. Also, some refactoring actions may not be detected because of the large differences that may occur between two releases.

This, together with the small size (three projects) are the main limitations of this study.

[1] In the following we refer previous work as Bavota *et al.* because the set of authors only partially overlaps.

More recently, Ferreira *et al.* [24] reported preliminary results of a mining-based study performed on five systems and overcoming the main design issue of the work by Bavota *et al.* [8]. Ferreira *et al.* mined both refactoring actions and bug-inducing changes at commit-level, looking for how "close" the refactoring actions were to bug-inducing changes. They confirmed the relationship between refactoring actions and bugs showing, however, that many bugs are not the direct consequence of the refactoring action, but of changes implemented later on the refactored code. By using a tool-chain similar to the one adopted by Ferreira *et al.* [24], we present a differentiated replication of the study by Bavota *et al.* [8]. We overcome several limitations of that study by:

Taking profit of the recent advances made in the mining software repositories field. This reflects in (i) better refactoring miner tools able to precisely identify refactoring actions at commit-level granularity [47], thus avoiding the assumption made in the original study done at release-level; (ii) enhanced implementations of the SZZ algorithm, overcoming some of the limitations of the original algorithm [19]; (iii) a line-level linking between refactoring actions and bug-fixing activities (as compared to the file-level linking done in previous studies).

Considering the possible impact of the size confounding factor on the achieved results. While the original study indicated a relationship between specific refactoring actions and the introduction of bugs, the authors ignored the possible impact of the size confounding factor on this finding (*e.g.,* refactoring is usually performed in larger commits and larger commits are more likely to introduce bugs).

Complementing the quantitative analysis with a systematic qualitative evaluation. We manually analyze a statistically significant sample of 384 commits identified as fix-inducing refactoring actions (*i.e.,* those that induced a bug-fixing activity) to study whether the performed refactoring actions actually induced the bug-fix. This analysis provides more confidence in the reported quantitative findings.

Answering the same research questions presented in [8], but on a larger scale. We answer the research questions presented in [8] both on the same three systems used in the original study, as well as, on a set of 100 open source Java projects. This increases the generalizability of the findings.

Despite the different experimental design adopted, our quantitative analysis confirms most of the findings of the original study. However, we also unveil the significant role played by the size confounding factor in inducing bug-fixing activities. Also, our qualitative analysis shows that, while the SZZ can identify the commit implementing the refactoring(s) as the last one modifying the code then subject to bug-fixing activities, in many cases the bug was already in the system before the refactoring even happened.

The obtained results trigger further research in the area of automated refactoring, but also warns developers about possible risks associated with refactoring activities, if the latter are not accompanied by suitable verification & validation.

Paper structure. Section 2 describes the study design. Results are discussed in Section 3, while their threats to validity in Section 4. After a discussion of related work (Section 5), Section 6 concludes the paper.

2 STUDY DESIGN

The *goal* of the study is to perform a differentiated replication of the work by Bavota *et al.* [8], in which the authors investigated the extent to which refactoring actions trigger bug-fixing activities. The *context* is represented by the history of 103 Java projects, and in particular by the refactoring operations and bug-fixes performed by their developers.

We address the following research questions (RQs):

RQ$_1$ ***Are refactoring-related commits more likely to induce fixes than other commits?*** This RQ mirrors the RQ$_1$ from the original work of Bavota *et al.* [8]. They answered this RQ by mining refactoring actions and fix-inducing changes performed between subsequent releases of three systems. Using this data, Bavota *et al.* investigated whether refactoring operations are likely to induce bug-fixes. However, as also acknowledged [8], the strong (unverified) assumption behind the study is that there is an overlap between the fix-inducing commits and the commits that implemented the refactoring actions. Instead of performing our replication at release-level, we use a commit-level granularity. This means that we know the exact commits in which refactoring operations have been performed in a specific file F_j and, as a consequence, we can check whether those commits induced a fix or not. We also improved other aspects on top of the original experimental design. Finally, while we answer RQ$_1$ by using the same three systems adopted in the original study [8], we also answer RQ$_1$ in a large-scale study involving 100 open source projects.

RQ$_2$ ***To what extent is the relationship between refactoring actions and fix induced changes influenced by the effect of size?*** Bavota *et al.* did not consider the size of the code change as a possible confounding factor in their analysis. However, it is well-known that large commits (*i.e.,* commits impacting a large number of files/lines/code churns) have a higher probability of inducing a bug [33]. It is possible that commits implementing refactoring operations are more likely to induce bug-fixes simply because they are larger than commits implementing other types of changes (*e.g.,* bug-fixes, enhancements). RQ$_2$ aims at investigating the role played by the commit size co-factor in the relationship between refactoring actions and fix-inducing changes.

RQ$_3$ ***What kinds of refactoring types are more likely to induce fixes?*** RQ$_3$ mirrors the RQ$_2$ of the original study, and analyzes the likelihood that different types of refactoring (*e.g., extract class*, *pull up method*) trigger bug-fixing activities.

RQ$_4$ ***To what extent does refactoring actually trigger bug-fixing activities?*** RQ$_4$ is a qualitative analysis we perform on a sample of the fix-inducing commits we identified in our quantitative study as responsible for both (i) implementing a refactoring, and (ii) inducing a bug-fixing activity. In other words, these should be the commits where there is a cause-effect relationship between refactoring and bug introduction.

2.1 Context Selection

We answer our research questions by mining the change history of 103 projects. Three of them, namely Apache Ant, ArgoUML, and Apache Xerces-J, are the Java projects used in the replicated study [8], while the remaining 100 were selected from GitHub through the following procedure.

Our initial idea was to mine popular and large projects from GitHub, excluding forked projects, coding tutorials, and personal projects, as well as projects having less than 100 issues and 1,000 commits, to ensure the availability of a long change history to study.

Also, we decided to ignore projects having less than 80% of their code written in Java since the refactoring detector used in our study [47] only works with Java. Finally, since in our study it is of crucial importance to identify bug-fixing commits, we also wanted to exclude repositories not using a clear label for bugs and those not consistently referencing in commit notes the id(s) of the issue(s) closed by the commit. Concerning the first point (*i.e.,* label for bugs), in GitHub every project can define its own set of labels to "tag" the opened issues, thus indicating bugs, feature requests, *etc.* As for the second point, having an explicit link between commits and bugs allows to precisely identify the bug-fixing commits needed for our study.

To this aim, we used the GitHub API [3] to extract the list of projects having at least 100 issues and Java as their "first language". The latter criterion means that Java is the most used language in the project, but does not guarantee that the vast majority of the code is written in Java. Since the GitHub API returns at most 1,000 results per search, we generated several requests, each having a specific size range. We used the `size:min..max` argument to retrieve only projects within a specific size range. In this way, we increased the number of returned results to up $1{,}000 \times n$, where n is the number of considered size ranges. Note that, while such a search heuristic does not allow to identify all possible GitHub projects having at least 100 issues and Java as their primary language, this is not important for the sake of our study. Here the goal was to just collect a set of candidate projects that then we can manually validate to decide which ones to include in our study. We collected 2,538 projects, and two of the authors inspected them to check the selection criteria previously mentioned. After analyzing the first 1,000 projects (by sorting them in descending order of stars), it became clear that most of these projects were not suitable for our study. In particular, out of these 1,000, we found only 40 projects to match all our selection criteria. Then, upon further inspection, other problems were found also for most of these 40 projects. Some of them, while having defined an explicit label for bugs, had very few labeled issues in the issue tracker. For others, while in the manual inspection of the change-log we observed commits linked to closed issues, the number of these links turned out to be very low even in projects having a very high number of commits and issues. This likely indicated the non-consistent adoption of a linking methodology between issues and commits.

For these reasons, we decided to adopt a different process for project selection. However, before describing it, we want to stress the challenges and perils of automatically selecting projects from GitHub. Indeed, while we applied some strong selection criteria on the number of issues (at least 100) and sorted projects based on their popularity as indicated by the number of stars (the most popular projects in our dataset had ~67k stars), we obtained as result many tutorial-like projects (*e.g.,* `Snailclimb/JavaGuide`), repositories collecting quiz for job interviews (*e.g.,* `kdn251/interviews`) or, as previously said, repositories making a very limited use of methodologies to link commits and issues and/or to consistently label issues. We believe this is an important warning for our research community when dealing with large-scale studies in which project selection is not manually curated.

We decided to focus on projects managed by the Apache Software Foundation (ASF) [1], because these are well-used projects managed by a known open source foundation. Also, a large chunk of these projects consistently used through their entire change history a single bug-tracking system, namely JIRA [2]. The issues are always classified based on their types (*e.g.,* bug) and, as a best practice, the Apache projects reference the issue id(s) in the note of commits closing issues. We used the GitHub API to extract the list of GitHub projects managed by the ASF. Then, we filtered out projects not having at least 80% of their code written in Java, obtaining a list of 554 candidate projects. Finally, we sorted them by the number of forks (as a proxy for popularity), and two of the authors manually inspected this list from the top with the goal of selecting 100 projects to use for the study. The selection was done based on two criteria: 1) the project used the JIRA issue tracker for its entire change history; 2) the project was not a subproject representing a "component" of a bigger project (*e.g.,* we excluded `fineract-cn-portfolio`). If these two criteria were met, the authors annotated the name of the projects from the Apache JIRA installation [2] that were referenced in the change-log of the repositories (*i.e.,* in the commit notes). Indeed, the Apache JIRA installation hosts several projects, each one identified by a specific name. For example, the `apache/hadoop` project references in its change-log issues from the following projects hosted in Apache JIRA: HADOOP, HDFS, MAPREDUCE, and YARN. The two authors stopped when the set of 100 projects was collected (available in our online appendix [21]).

For what concerns the three projects used in the replicated study, two of them (*i.e.,* ArgoUML and Xerces-J) use JIRA as well in their whole change history. Ant, instead, uses a mix of Bugzilla and JIRA and, thus, we had to manage this case in a different way as explained in the next section.

2.2 Data Extraction

Once cloned the 103 repositories we used RMiner [47] to identify commits containing refactoring operations. RMiner has been estimated to achieve a precision of 98% and a recall of 87%. For each project, we run RMiner on all commits of all branches impacting Java files, excluding merge commits.

RMiner outputs, for each commit, the list of refactoring actions detected, with the files and lines affected on the left-hand-side (before) and right-hand-side (after) of the change.

For the three projects studied by Bavota *et al.* [8], we considered two different observation periods. The first considers the same history they analyzed *i.e.,* analyzing all commits preceding the releases they studied (identified from release tags or commit messages), and bug fixes limited within their observation period, *i.e.,* by December 31, 2011. Specifically, we considered the following release intervals: ArgoUML (0.11, 0.34], Ant (1.1, 1.8.2], and Xerces (1.0.3, 2.9.1].

The second observation period considers the whole evolution of the three projects up to January 15, 2020. Similarly, for the 100 Apache projects, we considered the entire history on GitHub until January 15, 2020.

To identify fix-inducing changes, we first download the issue reports of the mined projects by using the JIRA project names previously extracted during the project selection. For the 100 Apache projects, we download issue reports using the Perceval tool [4]. As for the three projects from the replicated study [8], they use a heterogeneous way of reporting issues. While Xerces uses the Apache JIRA server, and ArgoUML uses its own JIRA installation, Ant is the trickiest case because it used Bugzilla at the beginning of its history, and JIRA later. Also, Ant has several cases of bugs reported directly in the commit message. Therefore, for these projects, we identified regular expressions in commit messages referring to (i) JIRA issues, (ii) Bugzilla issues, and (iii) bugs fixed without an issue. For the first two cases, we downloaded the issue reports using the wget Unix utility, rendering them as free-text using the Lynx browser, and extracted the relevant content using a Python script. For fixes without an issue report, we assumed the reporting and closing timestamp to match the commit timestamp.

Once downloaded the relevant issues, we linked them to commits using a regular expression-based approach [25]. For the Apache projects, the regular expression is of type `ISSUEPROJECT-#` (where ISSUEPROJECT is the name of the project on the issue tracker), whereas for the three other projects we used all possible regular expressions identified through the manual analysis explained above. We considered as bug-fixing commits those (i) linked to an issue of type "Bug" or, for Bugzilla (Ant), of priority at least "Normal" and not being an "Enhancement"; (ii) where the issue was in status "Closed" and Resolution "Fixed", except for 12 Apache projects where the Closed status was not used, and we kept those with a "Resolved" status. For the Ant fixes without an issue, as explained before, we simply relied on the commit message regular expression. Finally, we noticed that some of the mined commits included commits reverting previous bug fixes (thus, they were matching our regular expressions since mentioning the issue for which they were reverting the fixing). We excluded these cases from the analysis.

While we are aware that software projects may contain fixes with no explicit link to issues [12] and that approaches to propose candidate links for such fixes exist [52], we preferred to avoid such a solution in order to limit false positives.

More important, as explained in Section 2.1, one criterion for the selection of projects was the careful usage of issue trackers (the only exception was Ant, which has several non-tracked issues, which we handled as explained above). We could have identified bugs from commits to mitigate the bias described by Bird *et al.* [12], but this would have introduced false positives in the bug datasets and, also, would not have provided us with information about the issue opening date. For this reason, we limited this approach to untracked commits from Ant.

After having the set of bug-fixing commits and related issue metadata available, we were ready to apply the SZZ.

At first, we tried to use already available tools, and in particular, SZZ Unleashed [13]. However, by experimenting it and by discussing with its authors, we discovered that sometimes it tracks to wrong file version and line numbers, due to issues with the used Python git library. Thus, we implemented our own version of SZZ, capable of (i) ignoring cosmetic changes (*i.e.,* formatting, using the git blame -w option), changes to comments, and changes to non-Java files; and (ii) relying on the native Unix `git diff`, renaming and line mapping. Our SZZ does not ignore semantically-equivalent changes because, indeed, we are interested in analyzing refactoring actions. Our SZZ implementation first identifies the lines changed by the fix. Then, starting from the file version before the fix, and considering only the fixed lines, it uses `git blame -w -p` to identify the last change before the fix to these lines, along with the file name, and the line number mapping. In summary, for each changed line of fixed files, the algorithm outputs a candidate introduction location (commit, file name and line number). We discard candidate fix-inducing changes that occurred after the issue opening date. As for fixes without an issue (only for the Ant project), this heuristic was not used as a filter.

Recent work suggests that for an accurate fix-inducing change identification, bulk commits as well as the first commit of the project should also be ignored [19], although the work also points out that such commits can still introduce fixes. For such reasons, we decided to keep them, also considering that (i) the first commit of the analyzed projects does not contain refactoring actions, and therefore false positives in those commits do not affect the experimental group; (ii) refactoring actions could occur in bulk commits because these can be commits aimed at performing a general restructuring of the projects. At the same time, in RQ_2 we control the effect of the change' size on the observed results. Furthermore, some SZZ implementations [20] only consider the most recent blame from each fix as a fix-inducing change, while we consider all possible blames as we want to be conservative. Indeed, we keep track of these changes and we show how results change if limiting the analysis only to those.

As a final step of our data extraction approach, we merge the SZZ output with the RMiner output. Specifically, for each commit considered by RMiner, we report:

(1) whether it contains at least a refactoring;
(2) whether it induces a fix;
(3) whether there is at least one fix inducing change and refactoring action occurring in the same file;
(4) whether there is at least one fix inducing change and refactoring action occurring on the same line;
(5) detailed information for each refactoring action, *i.e.,* refactoring type and whether the refactoring occurs in a file and in a line with fix-inducing changes.

Finally, to control for the size of the change, we compute using `git diff`, for each analyzed commit, the number of changed Java files and the number of churns and of lines added and deleted in these files.

2.3 Analysis Methodology

The analyses described below have been performed using the R statistical environment [41]. To address $\mathbf{RQ}_1$, we first use a methodology similar to the one applied by Bavota *et al.* [8]. That is, we use Fisher's exact test [26] and Odds Ratio (OR) effect size to check whether commits containing at least one refactoring induce fixes in a higher proportion with respect to other commits. An OR $x > 1$ indicates that the odds for refactoring-related commits to induce fixes are x times greater than other commits. Note that for a refactoring-related commit we assume that the refactoring induces a fix if (1) at last a refactoring occurs on the same file where the fix is induced;

or (2) the refactoring impacts the same lines changed in the bug-fixing commit. We analyze results for both options (1) and (2). For option (1) it is possible that refactoring and bug fixing occur in different lines of the same file. As it will be explained in Section 2.2, since the relationship between the refactoring and the bug fix is determined using a re-implementation of the SZZ algorithm [44], the fix must occur after the refactoring. We perform the analysis on each project separately, and then we adjust p-values using the Benjamini–Hochberg procedure [10].

To address $\mathbf{RQ}_2$, we first identify the change size indicator to be used, by analyzing the presence of a correlation (using Spearman's rank correlation) between different size indicators. Then, we test the null hypotheses H_{0r}: refactoring-related commits do not have a significantly different size from other commits, and H_{0f}: fix-inducing commits do not have a significantly different size from other commits. We first test such null hypotheses using Wilcoxon rank-sum test [51]. We then consider all possible combinations of the two factors (*e.g.,* fix-inducing and refactoring-related, fix-inducing but not refactoring related, *etc.*), using Kruskal-Wallis test [34] followed by a Dunn post hoc analysis [22]. We also report Cliff's delta effect size values [28].

Finally, we study whether the size of the change and refactoring actions interact with respect to inducing a fix, by using a logistic regression model with mixed-effect (*glmer* function of the R *lme4* package [7]). The dependent variable is a dichotomous variable indicating if a commit is fix-inducing or not; the independent variables are dichotomous variables indicating whether a commit contains at least a refactoring which impacted a fix inducing file or line, the commit size, and their interaction. The random effect is the project.

To address $\mathbf{RQ}_3$, we perform, on data from all projects, an analysis similar to the one of RQ_1, but by refactoring type. That is, we consider whether commits containing at least one refactoring of a given type have higher odds to induce a fix (again considering as positive cases when the refactoring overlaps with the fix at file or line level) than commits not containing that kind of refactoring. Since the test is repeated for 41 refactoring types, p-values are adjusted as before.

To address $\mathbf{RQ}_4$, we firstly extracted from our dataset the 17,985 bug-fixing commits for which a match with one or more refactoring was found at line level in the fix-inducing commit. This means that the source code lines impacted by the bug-fixing commit were also impacted, completely or in part, by refactoring operations performed in the fix-inducing commit.

Once obtained this set, we extracted from it a statistically significant sample ensuring a 95% confidence level ± 5%.

This resulted in the selection of 384 bug-fixing commits with their related refactoring operations. The selection of the 384 instances was performed in the following way. First, we analyzed the distribution of refactoring types (*e.g., extract class, extract method, etc.*) in the entire population of fix-inducing commits implementing refactoring actions. In this way, we found out the percentage of fix-inducing commits in which each refactoring type appears. Then, we also computed the number of fix-inducing commits in each of the 103 systems considered in our study. The system and the refactoring type were used as strata to randomly select the 384 commits for manual validation. This means that the higher the number of fix-inducing commits in a system S, the higher the number of fix-inducing commits from S that will be in our sample. Similarly, the higher the number of fix-inducing commits containing a certain refactoring type T, the higher the number of commits implementing T in our sample.

The selected sample was manually analyzed by three authors (from now on evaluators) with the goal of classifying them as false positive (*i.e.,* the refactoring in the fix-inducing commit was not responsible for the bug introduction) or as a true positive (*i.e.,* the refactoring introduced the bug). In the latter case, the evaluator could also briefly describe the reason why the refactoring induced the bug-fixing activity.

The manual analysis was supported by a Web app that we developed for this task. Each author independently inspected the commits randomly assigned to her by the Web app. Each commit was assigned to two evaluators by the Web app, that showed for a given commit: (i) the link to the bug-fixing commit in GitHub, highlighting the code line(s) modified in it that was also impacted by the refactoring; (ii) the link to the fix-inducing commit in GitHub, highlighting the code line(s) impacted by the refactoring that was also modified in the bug-fix; (iii) a list of the refactoring actions detected by RMiner that were implemented in the fix-inducing commit and matched the lines in the bug-fix. Each author roughly classified 270 commits to obtain the two evaluations needed for each of the 384 commits. At the end of this process, the authors performed an open discussion to solve the 117 conflicts (30%) that have occurred.

To answer RQ_4, we report the percentage of analyzed commits in which we found an actual link between refactoring and bug introduction. Also, we discuss interesting cases identified in our manual analysis.

3 STUDY RESULTS

We discuss the results accordingly to the defined RQs.

3.1 RQ_1: Are refactoring-related commits more likely to induce fixes than other commits?

We report the comparison of the proportion of fix-inducing changes occurring in commits with a refactoring — overlapping at the file(s) or the lines(s) level — and in other changes. In particular, Table 1 reports results on the same systems and on the same history studied by Bavota *et al.* [8].

As the table shows, commits with refactoring always have significantly higher odds to induce a fix than other changes. Looking at the top-side of the table, the ORs are between 3.46 and 3.87 when considering a matching at the file level. This is the closest comparison to Bavota *et al.* [8]: compared to our results, they reported OR at release-level, with the following OR ranges computed for significant differences: Ant [3.50,6.65], ArgoUML 5.17 (only one release showed significant results) and between 8.79 and 157.69 for Xerces2-J. Note that they counted proportions on refactoring instances detected on a single release, and because of that for many releases (13 out of 17 for Ant, 13 out of 14 for ArgoUML, 23 out of 29 for Xerces2-J) they did not obtain statistically significant results. However, such a lack of significance seems to be due more to a limited statistical power rather than to other reasons. Similarly, the

Table 1: RQ_1: Replication on the same systems and history of Bavota *et al.*. (NRNI: no refactoring, no inducing fix; NRI: no refactoring, inducing fix; RNI: refactoring, no inducing fix; RI: refactoring, inducing fix).

File matching						
System	**NRNI**	**NRI**	**RNI**	**RI**	**OR**	**p adj**
Ant	11,823	288	1,981	187	3.87	<**0.001**
ArgoUML	19,413	458	3,979	344	3.66	<**0.001**
Xerces2-J	4,206	614	672	340	3.46	<**0.001**
Line matching						
System	**NRNI**	**NRI**	**RNI**	**RI**	**OR**	**p adj**
Ant	11,823	288	2,085	83	1.63	<**0.001**
ArgoUML	19,413	458	4,144	179	1.83	<**0.001**
Xerces2-J	4,206	614	846	166	1.34	<**0.001**

Table 2: RQ_1: Replication on the same systems of Bavota *et al.*, history up to date. (NRNI: no refactoring, no inducing fix; NRI: no refactoring, inducing fix; RNI: refactoring, no inducing fix; RI: refactoring, inducing fix).

File matching						
System	**NRNI**	**NRI**	**RNI**	**RI**	**OR**	**p adj**
Ant	13,762	371	2,248	257	4.24	<**0.001**
ArgoUML	22,526	590	4,366	410	3.59	<**0.001**
Xerces2-J	5,747	686	970	392	3.38	<**0.001**
Line matching						
System	**NRNI**	**NRI**	**RNI**	**RI**	**OR**	**p adj**
Ant	13,762	371	2,384	121	1.88	<**0.001**
ArgoUML	22,526	590	4,564	212	1.77	<**0.001**
Xerces2-J	5,747	686	1,168	194	1.39	<**0.001**

157 OR they observed for ArgoUML was computed on a release with only 2 refactoring actions and 2 fix inducing commits.

Looking at the bottom side of the table, if we consider that a refactoring induces a fix only if a line affected by the refactoring is also modified in the bug-fixing commit, odds are reduced by 60% or more, and vary between 1.34 and 1.83. Still, changes involving refactoring actions have higher odds to induce a fix. Also, note this is a very conservative analysis because a refactoring might still impact a fix without directly affecting a line modified in the bug fix.

Considering the complete history of the projects, as Table 2 shows, results are quite consistent with the ones of Table 1.

When performing the Fisher's exact test for the 100 Apache projects, at file-level, 85 p-values are statistically significant (< 0.05, before and after the adjustment). At line-level, only 34 p-values are statistically significant, 28 after the adjustment. Figure 1 shows the distribution of OR at file- and line-level matching for the 100 Apache projects. An OR greater than one indicates that a commit where a refactoring occurs has more chances than other commits to induce a fix. For the file-level matching, the median OR is 2.13 (it reaches 2.36 if considering only the projects where the difference in proportion is statistically significant). For the line-level matching, the OR decreases dramatically to a level at which the difference

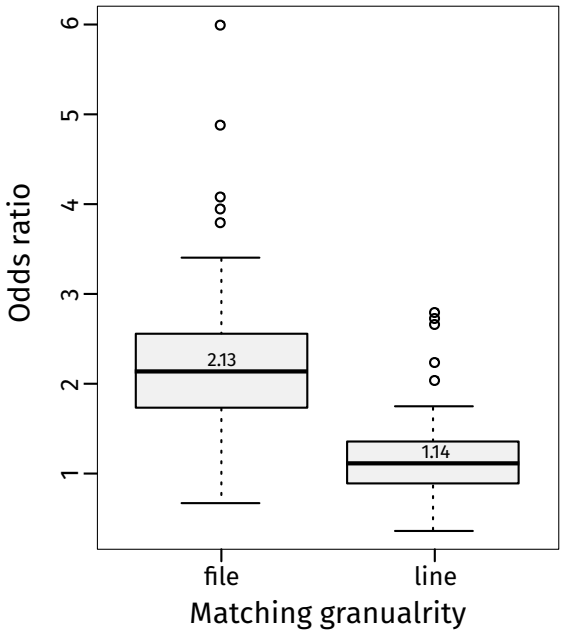

Figure 1: RQ_1: Odds that refactoring actions induce fixes. Boxplot of OR for the 100 Apache projects.

between a commit with refactoring actions and other commits is smaller (OR=1.13, while it reaches 1.46 if considering statistically significant cases only).

What if considering as fix-inducing only the most recent blame [20]? We performed the analysis (details in the replication package [21]), and results did not change dramatically. For the three projects of Bavota *et al.*, odds were still above 3 at file-level and above 1.5 at line-level. For the 100 Apache projects the median OR was 2.07 and 1.17 at file- and line-level, respectively.

RQ_1 Summary: Our results confirm the main findings of Bavota *et al.* [8]. Commits implementing refactoring actions have higher odds to induce a fix than other changes. This finding is also confirmed when working at line-level granularity, even though the difference between refactoring and other types of changes is less marked.

3.2 RQ_2: To what extent is the relationship between refactoring actions and fix induced changes influenced by the effect of size?

We found a moderate to strong correlation (0.59) between the number of changed files and the number of added lines, between the number of added and deleted lines (0.46), between the number of added lines and added churns (0.79), and between the number of added lines and deleted churns (0.50). Therefore, we only report our analysis considering, as the size of a change, the number of added lines. We also performed the same analysis for the other factors, obtaining similar results.

Table 3: Mixed-effect logistic regression relating refactoring, lines added, and their interaction with fix-inducing changes

AIC	BIC	logLik	deviance	df.residuals
300,772.4	300,828.6	-150,381.2	300,762.4	562,671
Scaled residuals:				
Min	**1Q**	**Median**	**3Q**	**Max**
-3.4245	-0.3249	-0.2389	-0.1491	15.0509
Random effects:				
Groups			**Variance**	**Std.Dev.**
Project (Intercept)			0.7136	0.8448
Number of obs: 562,676, Groups: Project: 103				
Fixed effects:				
	Estimate	**Std. Error**	**z value**	**Pr(>|z|)**
(Intercept)	-3.24	0.08	-39.81	**<0.001**
Ref.	0.83	0.01	61.97	**<0.001**
Lines Added	0.01	0.00	80.96	**<0.001**
Ref.:Lines Added	-0.00	0.00	-16.95	**<0.001**

The Wilcoxon rank-sum test indicates that commits with fix-inducing changes are bigger than other commits (p-value < 0.001) with a medium effect size (d=0.40). At the same time, commits in which refactoring actions occur are significantly bigger than others (p-value < 0.001), with a large effect size (d=0.50). In our dataset the conditions for ANOVA application were not met (residuals not normally distributed and variance not homogeneous). Therefore, we verified the presence of interaction between the two factors (*i.e.,* refactoring and fix-inducing) using a Kruskal-Wallis test followed by a post hoc Dunn's test with Benjamini-Hochberg correction. The test indicates that all possible combinations are statistically different from each other, and that (i) changes with refactoring actions and fix-inducing changes are larger than all other changes; (ii) changes with refactoring actions but not fix-inducing are larger than changes with no refactoring but fix-inducing, and (iii) changes with no refactoring actions and no fix-inducing are smaller than any other group.

Finally, we use a mixed-effect logistic regression model to evaluate whether, even in presence of the "size" effect, refactoring actions still correlate with fix-inducing changes. As Table 3 shows, the occurrence of refactoring actions, the commit size in lines added, and their interaction have a statistically significant effect on the likelihood that the commit induces a fix. By observing the estimates, the presence of a refactoring increases by $e^{0.83} = 2.29$ times the odds that a commit induces a fix, while a unity increment of the added lines increases the odds by $e^{0.01} = 1.01$, and a similar effect size is observed for the interaction between refactoring actions and lines added.

Similar results have been obtained considering, as a change size indicator, the number of added churns.

RQ_2 Summary: When controlling for size, the refactoring actions still play a role in inducing bug-fixing activities, thus supporting the RQ_1 findings.

Table 4: Fix-inducing proneness by type of refactoring.

Name	#	(%)	Buggy	OR	p adj
Extract Subclass	910	0.31	232	2.07	**<0.001**
Move And Inline Method	1,962	0.68	422	1.65	**<0.001**
Extract Class	4629	1.60	994	1.65	**<0.001**
Extract And Move Method	6,633	2.29	1,331	1.52	**<0.001**
Move And Rename Method	3,672	1.27	735	1.51	**<0.001**
Push Down Method	744	0.26	143	1.44	**<0.001**
Split Attribute	413	0.14	73	1.30	0.07
Extract Superclass	5,272	1.82	916	1.27	**<0.001**
Merge Variable	838	0.29	140	1.21	0.06
Move Method	5,767	1.99	930	1.15	**<0.001**
Parameterize Variable	3,870	1.33	618	1.15	**<0.001**
Merge Parameter	738	0.25	117	1.14	0.25
Replace Attribute	203	0.07	32	1.13	0.52
Split Parameter	331	0.11	52	1.13	0.48
Extract Interface	2,011	0.69	312	1.11	0.14
Split Variable	149	0.05	23	1.10	0.65
Inline Method	5,056	1.74	782	1.10	**0.03**
Push Down Attribute	575	0.20	88	1.09	0.48
Pull Up Attribute	1,310	0.45	198	1.08	0.42
Pull Up Method	1,571	0.54	230	1.03	0.65
Move Attribute	4,614	1.59	656	1.00	1.00
Move And Rename Class	2,999	1.03	389	0.90	0.09
Replace Variable With Attr.	3,469	1.20	443	0.88	**0.02**
Extract Method	27,371	9.44	3,509	0.86	**<0.001**
Merge Attribute	576	0.20	70	0.84	0.22
Move And Rename Attr.	216	0.07	25	0.79	0.40
Inline Variable	5,957	2.05	611	0.69	**<0.001**
Rename Attribute	15,435	5.32	1,535	0.66	**<0.001**
Rename Variable	23,327	8.05	2,310	0.65	**<0.001**
Change Parameter Type	15,098	5.21	1,446	0.63	**<0.001**
Change Variable Type	20,484	7.07	1,929	0.61	**<0.001**
Rename Parameter	19,011	6.56	1764	0.60	**<0.001**
Change Return Type	16,868	5.82	1,493	0.58	**<0.001**
Change Attribute Type	20,064	6.92	1,721	0.56	**<0.001**
Rename Method	20,938	7.22	1,798	0.54	**<0.001**
Extract Variable	25,328	8.74	2,150	0.54	**<0.001**
Rename Class	8,459	2.92	609	0.46	**<0.001**
Extract Attribute	2785	0.96	197	0.46	**<0.001**
Move Class	6,345	2.19	373	0.37	**<0.001**
Change Package	1,149	0.40	60	0.33	**<0.001**
Move Source Folder	2,750	0.95	58	0.13	**<0.001**

3.3 RQ_3: What kinds of refactoring types are more likely to induce fixes?

Table 4 reports, for each refactoring type, the odd that a commit containing at least a refactoring of that type has to induce a fix. For this RQ, for space reasons, we consider only the case in which the refactoring overlaps with the bug-fix at line level. This also because non-overlapping lines in the same file could be subject to other refactoring types. Refactoring types are ordered by decreasing OR.

Most of the refactoring types having a high odd to induce fixes are those involving refactoring big chunks of code (*extract class*/*subclass*, *move and inline method*/*extract and move methods*), as well as those involving inheritance (*extract subclass*/*superclass*, *push down method*).

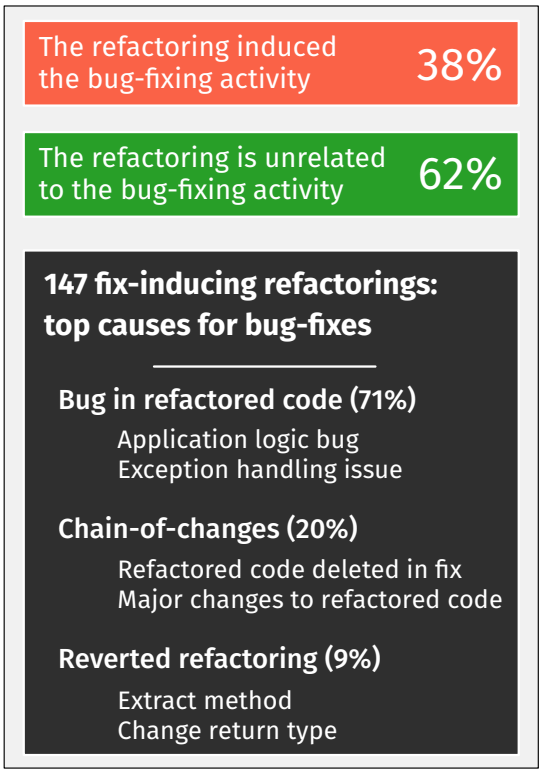

Figure 2: RQ$_4$: Manual validation of 384 fix-inducing commits implementing refactoring actions.

The latter confirms previous findings [8], which also found such types of refactoring to be particularly concerning, and literature highlighting the difficulties to test class hierarchies [29].

We can also notice how some refactoring actions not involving large changes, *e.g., split attribute* and *merge variable* have a relatively high OR (1.30 and 1.20, respectively). Instead, renaming changes are largely harmless, despite being among the most frequent refactoring actions we found. Surprisingly, *extract method*, another very frequent refactoring has an OR (0.86) smaller than similar refactoring types (*e.g., extract and move method*, 1.52). It is possible that extracting a method within the same class creates fewer problems than an *extract and move method* (due to the need for context adjustment).

RQ$_3$ Summary: Twenty refactoring types confirm their higher chances to induce fixes as compared to other types of changes, with ten of them being statistically significant. As compared to the work by Bavota *et al.* [8], we confirm the high odds to induce fixes for refactoring types related to inheritance.

3.4 RQ$_4$: To what extent does refactoring actually trigger bug-fixing activities?

Figure 2 shows the results of the manual validation we performed to verify whether the refactoring actions detected in 384 fix-inducing commits identified through the SZZ algorithm were actually responsible for triggering the bug-fixing activity. Before commenting on the results, a number of clarifications must be made. First, we noticed that in some cases what was labeled as "bug" in the issue tracker of the subject systems was not a *functional* bug, but rather an issue with non-functional aspects of source code (*e.g.,* performance) or, in a few cases, minor issues (*e.g.,* a wrong logging message).

We do not make any distinction among these types of issues in our study, assuming that what was labeled by the original developers as a "bug" should be considered as such. Second, while the authors involved in the manual validation have a strong experience in Java (*i.e.,* the language used in all subject systems), they are not the developers of the subject systems. In some cases, while we managed to identify the refactored code as responsible (or not) for triggering the fixing activity, we found extremely difficult to distill the exact code change that caused the bug. For example, let us assume that a method created through an *extract method* refactoring in the fix-inducing commit was the target of changes in the bug-fix, and that the impacted code was created during the *extract method* refactoring (*i.e.,* did not previously exist in the system). We labeled this refactoring as fix-inducing even if we did not manage to locate the actual bug in the code.

For 147 (38%) of the analyzed fix-inducing commits, we classified the refactoring as responsible for triggering the bug-fixing activity. This means that in 62% of cases (237 commits), while the refactoring actions were part of the changes implemented in the fix-inducing commit, the manual analysis did not show any evidence about their implication in the bug introduction. The main reasons for not considering the refactoring as the trigger for the fixing commit were three. In 31% of cases (74), the refactored code was unrelated to the bug introduction meaning that, while the bug was actually introduced in the commit indicated by the SZZ (*i.e.,* the one implementing the refactoring) and the bug-fixing commit also modified lines of code impacted by the refactoring, the fixed bug concerned other lines modified in the same commit that were not subject of any refactoring activity. In 29% of cases (68 out of 237), the fixed bug already affected the system before the refactoring. An example of this scenario is the case in which an *extract class* refactoring grouped together a number of existing statements and one of them was already buggy (*e.g.,* the condition in an `if` statement, then fixed in the bug-fixing commit). The subsequent *extract class* did not change the statements but was identified by the SZZ as responsible for triggering the fix since it was the last change impacting on the buggy statement. Finally, in the remaining 40% of cases (95), the refactoring and or the bug-fixing were part of tangled commits (such a percentage is smaller of the proportion of floss refactoring indicated in previous literature [37], *i.e.,* about 60%, but not particularly small), often of huge size, that made extremely difficult to identify the actual triggering of the bug-fix. However, in all those cases, the authors agreed on the unlikely link between the refactoring and the bug introduction.

For what concerns the 147 "true positive" instances, Figure 2 shows the three causes we identified for the triggering of bug-fixing activities, *i.e., Bug in refactored code*, *Chain-of-changes*, and *Reverted refactoring*. Each of these categories contains sub-categories better detailing the reason behind the bug. Due to space limitations, we

only report in Figure 2 the top-2 subcategories for each of these main categories. The complete categorization is available in our online appendix [21]. In the following, we describe each category and present one representative example for each of them.

Bug in refactored code. This is the "obvious" and expected reason for which a refactoring should trigger a bug-fixing commit and, indeed, this category accounts for 71% of the true positive cases. Most of the bugs in this category are related to application logic bugs, to the handling of exceptions, and to wrong initialization of variables. An example of this category is commit `c20ac05` from `apache/karaf`, in which an *extract method* refactoring is implemented. In particular, part of the `doExecute` method from the `DisplayLog` class is extracted into the newly created `display` method, which is then invoked in `doExecute` through the statement `display(cnv, event, out)`. The bug-fixing commit (`d9ecb3d`), which commit note mentions "*[KARAF-546] Added NPE check inside DisplayLog*", adds a Null Pointer Exception (NPE) guard in an `if` statement preceding the invocation of the extracted method (*i.e.,*`if(event != null)`) and avoids possible NPE. The changes introduced due to the performed *extract method* have induced the bug-fixing commit.

Chain-of-changes. In 20% of cases, while we were not able to precisely identify bugs in the refactored code, we observed a "chain-of-changes" triggered by the refactoring and resulting in the bug-fixing commit. For example, in 10% of cases, the refactored code (*e.g.,* an extracted variable/class/method) was deleted in the bug-fixing commit. In the remaining cases, the bug-fixing commit implemented major changes in the previously refactored code. For example, we found seven commits in which the refactoring changed the type of a parameter, a variable, or the return type of a method and then, the bug-fixing commit changed that same type again — not to the original type (*i.e.,* the one before the refactoring) but to a new one. An example of these cases is commit `6a1ced0` from `apache/felix`, in which the developers performed a *change parameter type* refactoring, changing `Source sourceDirectory` to `List sourceDirectories`. The bug-fixing commit changed again the parameter to `File outputDirectory`, with a consequent impact on the application logic of the method. Note that in these cases the link between refactoring and bug-introduction is less strong as compared to the previous category. However, we still see the refactoring as at least one of the causes of the changes implemented in the bug-fix.

Reverted refactoring. In 9% of cases, the bug-fixing commit reverted the changes implemented by the refactoring. Differently from the *Chain-of-changes* category, in this case the refactored code was reverted to its status before the refactoring. The most reverted refactoring actions are those related to the changes of types. In commit `ae008b7` of `apache/hive`, a *change variable type* converts the type of a variable `t` from `TimestampWritable` to `TimestampWritableV2`. Such a change is reverted in the bug-fixing commit `bd95a2f` with the following comment added in the source code `//Use old timestamp writable hash- code for backwards compatibility`.

It is important to note that, while we found 9% of reverted refactoring, none of them belong to an explicitly reverted commit (previous research indicate how reverted commits are used to undo changes throughout a project's history [42].)

RQ_4 Summary: Our manual validation, while confirming the possible role played by refactoring in the introduction of bugs, partially debunks the findings of our quantitative analysis and of previous studies [8]. Indeed, in 62% of cases, while the SZZ reports the commit implementing refactoring as the one inducing the bug-fixing activity, we did not find evidence of the linking between the refactored code and the bug-fix.

4 THREATS TO VALIDITY

Construct validity. Imprecisions in the detected refactorings could have affected our results. However, we used a highly precise state-of-the-art tool (RMiner [47]), reported to have a 98% precision and 87% recall. Another threat is related to the approximations and the granularity of the SZZ algorithm [13] used for identifying fix-inducing changes. As detailed in Section 2.2, we used appropriate heuristics to mitigate this issue, *e.g.,* filter out commented code and cosmetic changes. Although we did not compute the accuracy for our SZZ re-implementation, we mitigate this threat (i) by testing our implementation on a set of ~ 20 bug introduction instances, and (ii) through the manual analysis performed in the context of RQ_4.

Finally, links between commits and issues may be missing and biased [12], or issues improperly tagged [6, 30]. This is one of the reasons why we decided to use as subject systems a set of projects adopting well-defined practices to label issues and to link them to commits.

Conclusion validity. As already detailed in Section 2.3, wherever possible we used appropriate statistical procedures with p-value correction and effect size measures to test the significance of the differences and their magnitude.

Internal validity. Those are mainly related to a missing causation link between refactorings and bug fixes and to possible confounding factors that may influence such a relationship. We controlled for the size of implemented changes as confounding factors. Other co-factors not considered in our study may play a role in the reported findings (*e.g.,* floss refactoring activities). However, (i) in our observational study we do not claim causation, and (ii) at least, we complemented the quantitative analysis with a qualitative one, which helped in better understanding the refactoring-bug relationship.

External validity. While we considered over 100 projects in our study, we only considered Java projects belonging to the Apache ecosystem. In Section 2.1, we explained the reasons of this choice, *i.e.,* availability of reliable-enough defect data. Our findings may not generalize to other languages or to systems outside of this ecosystem. Also, we only considered the refactoring operations currently supported by RMiner.

5 RELATED WORK

As reported in the introduction, many studies have investigated software refactoring practices [32, 37, 39, 43]. In this section, we focus on the ones aimed at investigating the impact of refactoring on code quality, since being the most related to our work.

Bavota *et al.* [9], mined the evolution history of three open source projects looking at whether refactoring operations usually involve

code components with specific characteristics in terms of quality metrics and presence of smells.

Their results highlight that (i) very often quality metrics do not show a clear relationship with refactoring; (ii) only 42% of refactoring involves code components affected by code smells; and (iii) only 7% of the performed operations actually remove the code smells from the affected class.

Cedrim *et al.* [15] conducted a longitudinal study aimed at characterizing the beneficial and harmful effects of refactoring on code smells. Their results show that even if in ≃ 80% of cases refactoring activities involve smelly elements, only ≃ 10% of the refactoring actions results in the removal of code smells from the affected code. Moreover, they found that while applying refactoring developers tend to introduce new code smells (33%), *e.g.,* ≃ 30% of *move method* and *pull up method* refactoring operations introduce a God Class.

Chávez *et al.* [18] analyzed the impact of refactoring on internal quality attributes by looking at 29k refactoring actions occurred in the history of 23 projects. They found that often the refactoring touches code components showing at least one critical internal quality attribute. Furthermore, they show that 55% of these operations improve internal quality attributes against a 10% of code quality decline.

Eposhi *et al.* [23] studied, among other things, the relationship between refactoring and code quality issues. Their findings show that (i) the density of code smells is more than 8 times higher in refactored classes and (ii) refactoring actions usually do not reduce the density of quality issues.

Bibiano *et al.* [11] looked at refactoring operations applied in batches rather than in isolation to analyze their effect on code smells. Their study is based on the assumption that a single refactoring rarely suffices to remove a code smell. Surprisingly, their results show that batches mostly ended up introducing (51%) or not fully removing (38%) smells.

Vassallo *et al.* [48] mined 200 systems to quantitatively investigate factors correlating with refactoring, looking at when, why, and by whom refactoring is performed. Their results show that refactorings (i) are rarely performed close to a new release; (ii) are mainly performed while improving existing features; and (iii) are mainly done by the owners of the code components being refactored.

All the aforementioned work relate refactoring actions to quality attributes, such as metrics, code smells, or to process indicators (as Vassallo *et al.* [48] did), whereas our study relates refactoring actions to bug introduction, while considering the effect of some change metrics (*i.e.,* change size) as a co-factor. Our study allowed to (partially) corroborate previous findings reported in the literature [8].

A close-related work to ours is the one by Ferreira *et al.* [24], who conducted a study on five Java projects, 20,689 refactoring actions and 1,033 bug reports, looking at the distance between the commit in which the refactoring was performed and the commit in which the bug emerged in the refactored code element. They found that (i) many bugs are introduced in the refactored code as soon as the first immediate change is made on it, and (ii) code elements affected by refactoring actions performed in conjunction with other changes (*i.e.,* floss refactoring) are more prone to have bugs compared to root-canal refactoring actions.

Indeed, we used a similar toolchain (*e.g.,*RMiner to detect refactoring actions, SZZ to identify fix-inducing commits). However, the study design, the answered RQs, and the scale of the studies are different.

6 CONCLUSIONS

This paper reported a differentiated replication of a previous study by Bavota *et al.* [8], using a different and up-to-date tool chain, finer granularity and more precise matching between refactoring actions and fix-inducing changes, and being conducted at a larger scale (103 projects in total).

The data extraction itself posed several challenges and highlighted important lessons for researchers conducting similar studies. First, carefully test the tool chain (including third-party tools) being used. Second, refrain to perform an indiscriminate, large-scale mining from GitHub. While previous studies already advised about the risks of mining GitHub [31] and ranking projects by stars [14], or provided means to identify a diverse and representative set of projects [38], for our study we found that only relying on a set of project belonging to a well-disciplined ecosystem (*i.e.,* the Apache Software Foundation projects) allowed us to have enough confidence to mine projects with a good linking between commits and issues and issue classification.

The quantitative study results were surprising. Albeit the tool chain and the analysis methodology (*i.e.,* commit-level of granularity and matching of lines affected by refactoring actions and fix inducing changes) was completely different, and although we found how the size of a change played a significant role, results of the replicated study were generally confirmed, and the effect of refactoring appeared even more evident. Noteworthy, results hold both on the three systems analyzed in the original study and on a larger set including 100 additional Apache projects. These findings also support the observations reported in previous qualitative studies with developers [32], indicating their concerns about possible bugs introduced in the refactoring process.

However, a deep, manual analysis on a sample of 384 fix-inducing changes overlapping with refactoring partially debunked the quantitative results, revealing that a quantitative analysis may "scratch-the surface" and miss details on how exactly the source code changed over time. At the same time, there is still a good proportion of cases in which refactoring actions indeed induce fixes, and there are recurring patterns in such cases. Often such recurring patterns highlight latent implications, *e.g.,* reverted changes might imply that some refactoring actions were not carefully planned.

The obtained results entail implications for both researchers and practitioners. As for researchers, the study highlights the need for better refactoring support, in particular for better planning/pondering it (*e.g.,* in the direction of identifying its possible impact [16]), or automatically testing/verifying the change made, or further work in the direction of supporting refactoring review [27]. As for practitioners, this study warns them by pointing out that refactoring is only in theory behavior-preserving, therefore it must be planned with appropriate verification & validation activities aimed at reducing its risks.

We believe that our study, together with the previously published research on the same topic [8, 24, 50], provides substantial quantitative evidence of the relationship between refactoring and bug-fixing activities. However, we still see the need for more qualitative studies unveiling the mechanisms through which refactoring operations introduce bugs. Our future work will point in this direction.

The data and scripts used in our study are publicly available [21].

ACKNOWLEDGMENTS

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851720).

REFERENCES

[1] [n.d.]. *The Apache Software Foundation.* https://www.apache.org
[2] [n.d.]. *Apache's JIRA issue tracker.* https://issues.apache.org/jira
[3] [n.d.]. *GitHub REST API v3.* https://developer.github.com/v3/
[4] [n.d.]. *Perceval.* https://github.com/chaoss/grimoirelab-perceval
[5] Mohammad Alshayeb. 2009. Empirical investigation of refactoring effect on software quality. *Information and Software Technology* 51, 9 (2009), 1319 – 1326.
[6] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research.* IBM, 23.
[7] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software* 67, 1 (2015), 1–48.
[8] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012.* 104–113.
[9] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1 – 14.
[10] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300.
[11] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. 2019. A quantitative study on characteristics and effect of batch refactoring on code smells. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, 1–11.
[12] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar T. Devanbu. 2009. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009.* 121–130.
[13] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE@ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 27, 2019.* 7–12.
[14] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *J. Syst. Softw.* 146 (2018), 112–129.
[15] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017).* 465–475.
[16] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. 2014. On the Impact of Refactoring Operations on Code Quality Metrics. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014.* 456–460.
[17] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How Does Refactoring Affect Internal Quality Attributes?: A Multi-project Study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17).* 74–83.
[18] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes? A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering.* 74–83.
[19] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Trans. Software Eng.* 43, 7 (2017), 641–657.
[20] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process* 26, 1 (2014), 107–139.
[21] Massimiliano Di Penta, Gabriele Bavota, and Fiorella Zampetti. 2020. *On the Relationship between Refactoring Actions and Bugs: A Differentiated Replication – Replication Package.* https://doi.org/10.5281/zenodo.4018691
[22] Olive Jean Dunn. 1961. Multiple Comparisons among Means. *J. Amer. Statist. Assoc.* 56, 293 (1961), 52–64.
[23] Andre Eposhi, Willian Oizumi, Alessandro Garcia, Leonardo Sousa, Roberto Oliveira, and Anderson Oliveira. 2019. Removal of design problems through refactorings: are we looking at the right symptoms?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC).* IEEE, 148–153.
[24] Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, Anderson Uchôa, Ana Carla Bibiano, Alessandro Garcia, João Lucas Correia, Filipe Santos, Gabriel Nunes, Caio Barbosa, and et al. 2018. The Buggy Side of Code Refactoring: Understanding the Relationship between Refactorings and Bugs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE ?18).* 406?407.
[25] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. In *19th International Conference on Software Maintenance (ICSM 2003).* 23–.
[26] R. A. Fisher. 1922. On the Interpretation of X-square from Contingency Tables, and the Calculation of P. *Journal of the Royal Statistical Society* 85, 1 (1922), 87–94.
[27] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson R. Murphy-Hill. 2017. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017.* 71–79.
[28] Robert J. Grissom and John J. Kim. 2005. *Effect sizes for research: A broad practical approach* (2nd edition ed.). Lawrence Earlbaum Associates.
[29] Mary Jean Harrold, John D. McGregor, and Kevin J. Fitzpatrick. 1992. Incremental Testing of Object-Oriented Class Structures. In *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia, May 11-15, 1992.* 68–80.
[30] Kim Herzig, Sascha Just, and Andreas Zeller. 2015. It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction. In *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany.* 103–104.
[31] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.
[32] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering* (Research Triangle Park, NC, USA).
[33] Sunghun Kim, E. James Whitehead Jr., and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Trans. Software Eng.* 34, 2 (2008), 181–196.
[34] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621.
[35] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019.* 151–162.
[36] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. 2008. Balancing Agility and Formalism in Software Engineering. Chapter A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, 252–266.
[37] Emerson Murphy-Hill, Chris Parnin, and Andreaw P. Black. 2011. How We Refactor, and How We Know It. *Transactions on Software Engineering* 38, 1 (2011), 5–18.
[38] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in software engineering research. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013.* 466–476.
[39] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. 2018. An Empirical Investigation of How and Why Developers Rename Identifiers. In *Proceedings of the 2Nd International Workshop on Refactoring (IWoR 2018).* 26–33.

[40] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. 1–10.

[41] R Core Team. 2012. *R: A Language and Environment for Statistical Computing*. http://www.R-project.org ISBN 3-900051-07-0.

[42] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, David Pursehouse, and Naoyasu Ubayashi. 2016. Why are Commits Being Reverted?: A Comparative Study of Industrial and Open Source Projects. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. 301–311.

[43] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*. 858–870.

[44] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*.

[45] Konstantinos Stroggylos and Diomidis Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Proceedings of the 5th International Workshop on Software Quality (WoSQ '07)*. IEEE Computer Society, Washington, DC, USA, 10–.

[46] Gábor Szoke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 95–104.

[47] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 483–494.

[48] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180, 1 (2019), 1–15.

[49] Yi Wang. 2009. What motivate software engineers to refactor source code? evidences from professional developers. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. 413 –416.

[50] Peter Weißgerber and Stephan Diehl. 2006. Are refactorings less error-prone than other changes?. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*. 112–118.

[51] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[52] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. ReLink: recovering links between bugs and changes. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 15–25.