Empirical Software Engineering (2022) 27: 14
https://doi.org/10.1007/s10664-021-10051-z

Quick remedy commits and their impact on mining software repositories

Fengcai Wen[1] · Csaba Nagy[1] · Michele Lanza[1] · Gabriele Bavota[1]

Accepted: 13 September 2021 / Published online: 28 October 2021

Abstract

Most changes during software maintenance and evolution are not atomic changes, but rather the result of several related changes affecting different parts of the code. It may happen that developers omit needed changes, thus leaving a task partially unfinished, introducing technical debt or injecting bugs. We present a study investigating "*quick remedy commits*" performed by developers to implement changes omitted in previous commits. With *quick remedy commits* we refer to commits that (i) *quickly* follow a commit performed by the same developer, and (ii) aim at *remedying* issues introduced as the result of code changes omitted in the previous commit (e.g., fix references to code components that have been broken as a consequence of a rename refactoring) or simply improve the previously committed change (e.g., improve the name of a newly introduced variable). Through a manual analysis of 500 quick remedy commits, we define a taxonomy categorizing the types of changes that developers tend to omit. The taxonomy can (i) guide the development of tools aimed at detecting omitted changes and (ii) help researchers in identifying corner cases that must be properly handled. For example, one of the categories in our taxonomy groups the *reverted commits*, meaning changes that are undone in a subsequent commit. We show that not accounting for such commits when mining software repositories can undermine one's findings. In particular, our results show that considering completely reverted commits when mining software repositories accounts, on average, for 0.07 and 0.27 noisy data points when dealing with two typical MSR data collection tasks (i.e., bug-fixing commits identification and refactoring operations mining, respectively).

Keywords Fixing commits · Empirical software engineering · Mining software repositories

Communicated by: Yann-Gaël Guéhéneuc, Shinpei Hayashi and Michel R. V. Chaudron

This article belongs to the Topical Collection: *International Conference on Program Comprehension (ICPC)*

✉ Gabriele Bavota
gabriele.bavota@usi.ch

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

1 Introduction

In the software life-cycle, change is the rule rather than the exception. Changes are generally performed through commit activities to add new functionality, repair faults, and refactor code (Mockus and Votta 2000). Some of these commits can involve a substantial part of the source code, with dozens of artifacts impacted (Hattori and Lanza 2008). This is often the result of what Herzig and Zeller (2013) defined as *tangled commits*: Commits grouping together several unrelated activities, such as fixing a bug and adding a new feature.

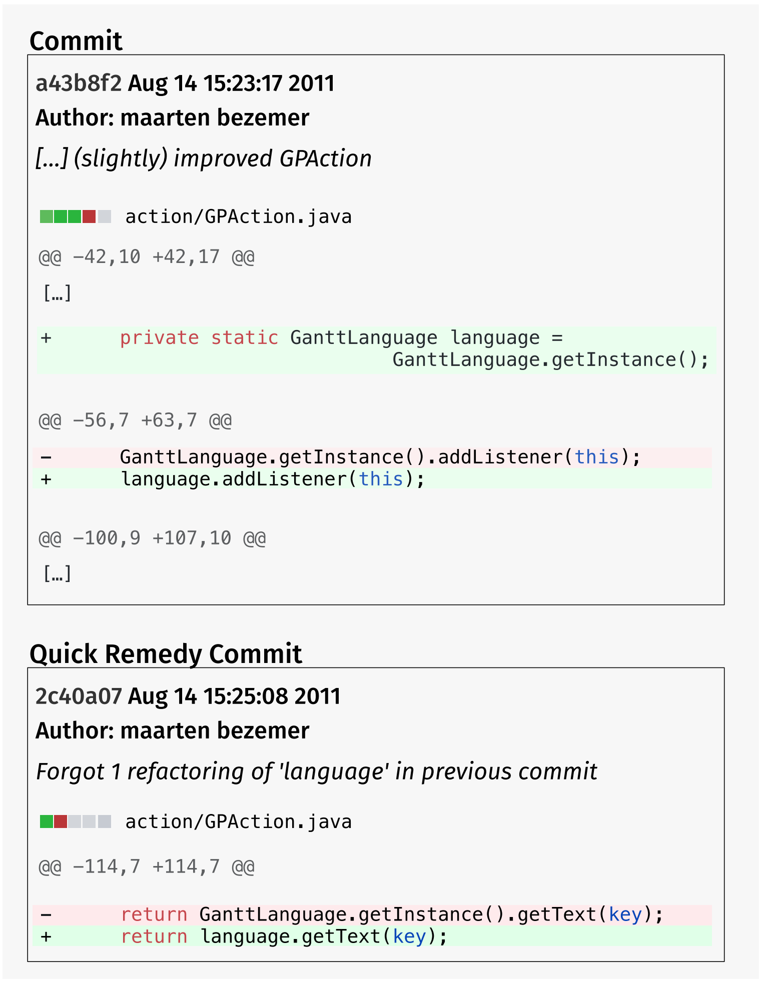
In other cases, a single cohesive change (e.g., a bug fix) is instead split across several commits. This can be due to omitted code changes and/or the need for fixing a mistake done in the first attempt to implement the change. Park et al. (2012) showed that 22% to 33% of bugs require more than one fix attempt (i.e., supplementary patches). Studying supplementary patches can be instrumental in designing recommender systems able to reduce omission errors by alerting software developers, as attempted in a subsequent work by Park et al. (2017), where the authors tried to predict additional change locations for real-world omission errors. Due to the limited empirical evidence about the nature of omitted changes, this is still an open challenge. Indeed, while the work by Park et al. (2012) investigates omitted changes, it explicitly focuses on supplementary patches for bug-fixing activities, ignoring other types of code changes (e.g., implementation of new features, refactoring). Thus, there is no study broadly investigating the types of changes that developers tend to omit during implementation activities.

To fill this gap, in our previous work (Wen et al. 2020) we presented a qualitative study focusing on "*quick remedy commits*" performed by developers. We defined as *quick remedy commits* those commits that (i) quickly succeed a commit performed by the same developer in the same repository; and (ii) aim at remedying the issues introduced as the result of code changes omitted in the previous commit (e.g., fix references to code components that have been broken as a consequence of a rename refactoring) and/or improves suboptimal choices made in the previously committed change (e.g., refactoring code to improve its comprehensibility). In other words, we identified pairs of commits (c_i, c_{i+1}) that are temporally close (i.e., c_{i+1} succeeds c_i by a few minutes), are performed by the same developer, and include in the commit note of c_{i+1} a reference to fixing issues introduced in c_i.

Figure 1 shows an example of a quick remedy commit from our dataset, and in particular from the GitHub project `bardsoftware/ganttproject`. In the commit depicted in the top part of Fig. 1 (i.e., commit `a43b8f2`), the developer implemented, among other changes, a refactoring aimed at simplifying the code of the `GPAction` class. In particular, instead of invoking three times the method `GanttLanguage.getInstance()` in different parts of the class, the `language` variable is instantiated, and reused where needed.

Two minutes later, the same author performs a *quick remedy commit* (bottom part of Fig. 1 — commit `2c40a07`) by reporting in the commit note: *Forgot 1 refactoring of 'language' in previous commit*. The remedy commit propagates the changes introduced by the refactoring to another location of the `GPAction` class, that was missed by mistake in the original commit.

We decided to focus on remedy commits (c_{i+1}) that are temporally close to the original change they fix (c_i) for two reasons. First, it is easier to establish a clear link between two commits by the same developer if they are performed within a few minutes. Second, as shown by Park et al. (2017), it is challenging to prevent omission errors automatically; thus,



Commit

a43b8f2 Aug 14 15:23:17 2011
Author: maarten bezemer
[...] (slightly) improved GPAction

```
🟩🟩🟩🟥⬜ action/GPAction.java

@@ -42,10 +42,17 @@
[…]
+       private static GanttLanguage language =
                                 GanttLanguage.getInstance();

@@ -56,7 +63,7 @@
-       GanttLanguage.getInstance().addListener(this);
+       language.addListener(this);

@@ -100,9 +107,10 @@
[…]
```

Quick Remedy Commit

2c40a07 Aug 14 15:25:08 2011
Author: maarten bezemer
Forgot 1 refactoring of 'language' in previous commit

```
🟩🟥⬜⬜⬜ action/GPAction.java

@@ -114,7 +114,7 @@
-       return GanttLanguage.getInstance().getText(key);
+       return language.getText(key);
```

Fig. 1 Example of quick remedy commit

we decided to focus on omission errors that, since fixed within few minutes, are likely not to be so complex.

This allows gathering empirical knowledge to take a first step in automating the prevention of a basic set of omission errors that, as we show, can be responsible for bugs and major code inconsistencies if not promptly fixed.

We defined heuristics to identify *quick remedy commits* automatically, and mined the commits of interest from the complete change history of 1,497 Java projects hosted on GitHub. This allowed the identification of ∼1,500 candidates quick remedy commits. We manually analyzed 500 of them looking at the changes introduced in the remedy commit (c_{i+1}) and the previous commit (c_i) as well as the summary of changes provided in the commit notes.

The goal of the manual analysis was to identify the rationale of the remedy commits to define a taxonomy categorizing the types of issues introduced by developers during commit activities that trigger a remedy commit, discussing the implications of our taxonomy for researchers and practitioners.

In this work, extending our previous paper (Wen et al. 2020), we further looked into the implications of a specific part of our taxonomy for researchers working in the Mining Software Repositories (MSR) field. In particular, we focused on a category in our taxonomy grouping together *reverted commits*, i.e., remedy commits c_{i+1} in which the developers revert, completely or partially, the code changes they committed in the previous commit c_i. We defined a methodology to automatically identify these commits in a given repository and studied the impact they could have on MSR studies. The decision to focus on such a specific category in our taxonomy is two-fold: (i) as we will explain later in the paper, it is the type of quick remedy commits that is more likely to affect the data collection process in MSR, possibly leading to the inclusion of noisy data points in the study; (ii) it is the only category for which a reliable and automated detection mechanism can be easily devised (i.e., it is relatively easy to detect reverted commits as compared to other categories of commits in our taxonomy).

We took two "data collection tasks" frequently performed in MSR studies, namely the identification of bug-fixing commits (see e.g., Rodriguez-Perez et al. 2017b; Rodríguez-Pérez et al. 2018; Tufano et al. 2018; Wang et al. 2020; Penta et al. 2020) and the mining of refactoring operations performed in the history of a system (see e.g., Penta et al. 2020; Peruma 2019; Mahmoudi et al. 2019; Lin et al. 2019; Fakhoury et al. 2019; AlOmar et al. 2019). Then, we applied these two tasks on 100 long-lived Github repositories; collecting refactoring operations performed in each commit and a set of bug-fixing commits. Finally, we cleaned the collected data by removing completely and partially reverted commits. For example, a researcher may identify a bug-fixing commit in the history of a software system. However, if such a bug-fix is later on reverted by the developer, we argue that, in most of the cases, it should not be considered as a valid data point, since it basically represents noise. We show that, for each completely reverted commit kept in the collected data, there is a .07 increase in the number of detected bug-fixing commits and a 0.27 increase in the number of detected refactoring commits. The methodology we adopt to identify the *reverted commits* can be applied in MSR studies to help researchers in minimizing the impact of these commits on their findings. Clearly, the removal of *reverted commits* is subject to the goal of the study and the data analyses researchers are interested in performing. For example, if the goal of the study is to count the number of bugs introduced by a developer in a system, researchers may be willing to also count bug-introducing commits that have been later on reverted. Instead, if the goal is to assess the logical coupling between code components (i.e., how frequently they co-change), researchers may want to ignore completely reverted changes in the coupling computation. Our study confirms the importance of careful data cleaning when mining software repositories, as highlighted in previous works (Rigby and Robillard 2013).

The data used in both our studies are publicly available (Replication package 2021).

Structure of the paper In Section 2 we present the design and the results of our first empirical study, in which we investigate the types of quick remedy commits performed by developers. Section 3 presents the design and results of our second study, assessing the potential impact of reverted commits in MSR studies. In Section 4 we discuss the threats that could affect the validity of our two studies, while in Section 5 we discuss the related literature. In Section 6 we conclude the paper and outline our future work.

2 Study I: Studying Quick Remedy Commits Performed by Developers

2.1 Study Design

The *goal* of the study is to qualitatively investigate quick remedy commits. The *purpose* is to define a taxonomy of quick remedy commits that developers perform to fix issues introduced in a previous commit and/or finalize an uncompleted implementation task. The study addresses the following research question (RQ):

RQ$_1$: *What types of quick remedy commits are made by developers in Java projects?*

This RQ aims at identifying the types of quick remedy commits that are performed by developers (e.g., documenting through a code comment a piece of code introduced in the previous commit). Knowing the types of quick remedy commits made by developers can guide the development of tools to automatically alert developers when code changes they are committing may require a subsequent remedy commit. In some cases this could even avoid the introduction of bugs (e.g., due to changes not propagated in all code areas where they are required).

2.1.1 Data Collection and Analysis

To answer RQ$_1$ we mined the complete change history of 1,497 open source Java projects hosted on GitHub. These projects represent the context of our study and have been selected from GitHub in November 2018 using the following constraints:

- **Programming language.** We only considered projects written in Java since all the manual evaluators involved in the study (i.e., three of the four authors) have experience in Java, and would be able to understand the reasons behind the quick remedy comments in most of the cases.
- **Change history.** Since we were interested in identifying a good number of quick remedy commits to manually analyze, we only selected projects having a relatively long change history, composed of at least 500 commits.
- **Popularity.** The number of stars (About stars (GitHub) 2021) of a repository is a proxy for its popularity on GitHub. Starring a repository allows GitHub users to express their appreciation for the project. Projects with less than ten stars are excluded from the dataset, to avoid the inclusion of likely irrelevant/toy projects.

A total of 6,563 projects satisfied these constraints. Then, we sorted the projects in descending order based on their number of stars (i.e., the most popular on top), and we manually inspected the ranked list (starting from the top) to filter out repositories that do not represent real software systems (e.g., JAVA-DESIGN-PATTERNS 2021 and SPRING-PETCLINIC 2021). Such a selection was done by checking the projects' names and descriptions (no code analysis was performed). We also checked for projects with shared history (i.e., forked projects). In particular, we considered as forked projects two repositories having in their history at least one commit having the same *SHA* and commit date. When we identified a set of forked projects, we only selected among them the one with the longest commit history (e.g., both FINDBUGS 2021 and its successor SPOTBUGS 2021 fall under our search criteria, but we only kept the latter one). Such a process stopped once we reached 1,500 valid projects for our study.

During the cloning of the 1,500 GitHub repositories, we got a cloning error for three of them. Thus, we extracted the list of commits performed over the change history of the remaining 1,497 projects. Table 1 reports descriptive statistics for size, change history, and popularity of the selected projects. The complete list of considered projects is publicly available in our replication package (2021).

To extract the history of the subject systems, we iterated through the commit history related to all branches of each project with the `git log --topo-order` command. This allowed us to analyze all branches of a project, without intermixing their history and avoiding unwanted effects of merge commits.

Then, given the commit history, our goal was to identify all pairs of subsequent commits (c_i, c_{i+1}) in which c_{i+1} had been performed by a developer D_j as a quick remedy fix for a commit c_i also authored by D_j. In other words, c_{i+1} must (i) have been authored by the same developer of c_i and performed within a relatively short time interval from c_i; (ii) clearly be a "compensatory" fix for c_i.

To identify the (c_i, c_{i+1}) pairs of interest, we adopt the following heuristic-based procedure. First, we computed the time interval between all adjacent (subsequent) commits in each system authored by the same developer. In *git* it is possible to retrieve the *author date* (i.e., the date in which the change has been implemented by the author) or the *committer date* (i.e., the date in which the change has been committed). Given the goal of our work, we considered the *author date*. We analyzed the distribution of these time intervals (see Fig. 2).

We considered the first quartile, exactly *five minutes*, as a candidate threshold to identify remedy commits: c_{i+1} commits performed as quick fixes for their predecessor c_i commit. This allowed us to select pairs of commits meeting our first requirement: They were authored by the same developer and performed in rapid succession (i.e., within five minutes). This filtering left us with 1,041,397candidate commits.

Second, we set up a process to define lexical patterns allowing the identification of c_{i+1} commits in which the developer explicitly indicates in the commit notes the fact that c_{i+1} is a remedy commit for changes introduced in the previous commit (c_i). The first author extracted from all 1,041,397commits output of the previous filtering step the words and 2-grams used in their commit notes. This means that, from a commit note reporting "*Fixes a bug introduced in previous commit*", we would extract *fixes*, *a*, *bug*, etc. as the single words, and *fixes a*, *a bug*, *bug introduced*, etc. as 2-grams. To remove noise, stop words (e.g., articles) and all single words shorter than four characters had been excluded from the set of single words (not from the 2-grams list). The remaining words and all 2-grams had then been sorted by frequency in descending order, excluding the long tail of those appearing in less than ten commits. Indeed, even if useful to identify remedy commits, lexical patterns defined from these words/2-grams are unlikely to retrieve a substantial amount of useful commits and, thus, are excluded *a priori* from reducing the inspection effort. For each remaining word/2-gram, we randomly extracted ten commit notes in which it appears.

This dataset, composed of words/2-grams and related commit notes, had been manually and independently inspected by three authors with the goal of defining the needed lexical

Table 1 Dataset statistics

	Overall	Per Project	
		Mean	Median
Java files	1,599,323	1,068	360
Effective LOC	162,243,714	108,379	31,392
Stars	2,895,219	1,930	762
Commits	7,926,912	5,313	1,778

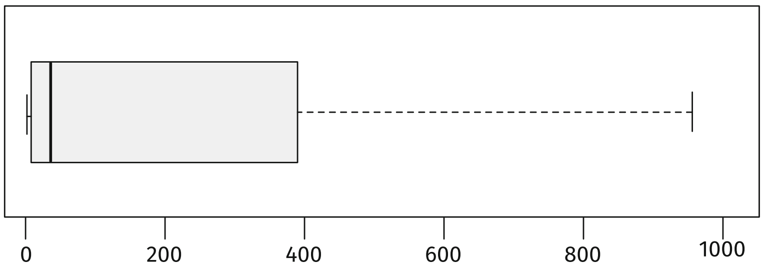


Fig. 2 Time differences (in minutes) between subsequent commits (without outliers)

patterns. After an open discussion in which each author presented his list of patterns, the three evaluators agreed on the following lexical pattern to identify remedy commits:

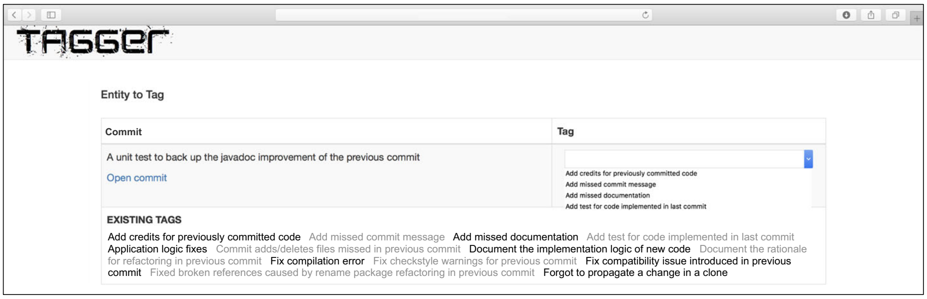
(*former* or *last* or *prev* or *previous*) and *commit*

This means that commit notes including *former commit*, *last commit*, *prev commit*, or *previous commit* would be matched and considered as relevant for our study. While this heuristic is quite strict, our goal was to maximize precision at the expense of recall, considering the fact that our study is qualitative in nature and does not target a large number of manually analyzed commits. At the end of this last filtering step, we obtained 1,577c_{i+1} commits which (i) have been authored within five minutes from the commit c_i previously performed by the same author; and (ii) explicitly mention in the commit note a lexical reference to the previous commit that can be captured by the defined pattern. Given the high cost of the manual analysis process detailed in the following, we decided to focus our analysis on a randomly selected sample of 500 commits, representing a 99% statistically significant sample with a 4.8% confidence interval.

The 500 commits were randomly distributed among three authors, making sure that each commit was classified by two authors. The goal of the process was to identify the exact reason behind the changes performed in the commit. If the commit was unrelated to the previous one, the evaluator classified it as *false positive*.

Otherwise, a tag explaining the reason for the change (e.g., *remove debugging code from the previous commit*) was assigned.

We did not limit our analysis to the reading of the commit message, but we analyzed the source code diff of the changes implemented in the GitHub commits, both in the c_{i+1} commit as well as in its predecessor (c_i). The tagging process was supported by a Web application that we developed to classify the commit and to solve conflicts between the authors. The Web application is shown in Fig. 3. Each author independently tagged the commits assigned to him by defining a tag describing the reason behind the commit. Every time the authors had to tag a commit, the Web application also showed the list of tags created so far, allowing the tagger to select one of the already defined tags (visible in the bottom part of Fig. 3). Although, in principle, this is against the notion of open coding, in a context like the one encountered in this work, where the number of possible tags (i.e., cause behind the commit) is extremely high, such a choice helps using consistent naming and does not introduce substantial bias. In cases for which there was no agreement between the two evaluators (44%of the classified commits), the commit was assigned to an additional evaluator to solve the conflict. While such a percentage may look high, it is worth considering that our task was not to assign commits to a list of predefined categories, but to define the names for such categories during the tagging process. This naturally leads to a higher number of conflicts.



Entity to Tag

Commit	Tag
A unit test to back up the javadoc improvement of the previous commit Open commit	Add credits for previously committed code Add missed commit message Add missed documentation Add test for code implemented in last commit

EXISTING TAGS

Add credits for previously committed code Add missed commit message Add missed documentation Add test for code implemented in last commit Application logic fixes Commit adds/deletes files missed in previous commit Document the implementation logic of new code Document the rationale for refactoring in previous commit Fix compilation error Fix checkstyle warnings for previous commit Fix compatibility issue introduced in previous commit Fixed broken references caused by rename package refactoring in previous commit Forgot to propagate a change in a clone

Fig. 3 Web application used to run the manual tagging

Also, we considered as a conflict cases in which a different but "semantically equivalent" tag was used by the two evaluators (e.g., *remove unnecessary code vs remove unneeded code*). In this case, the third evaluator just made sure that consistent wording was used, and selected the proper tag. In a minority of cases, the two evaluators applied completely different tags and the third evaluator could choose whether to reuse one of the two labels or, instead, define a new tag by discussing and agreeing with the two original evaluators.

After having manually tagged all commits, we defined a taxonomy of quick remedy commits through an open discussion involving all the authors (see Fig. 4). We qualitatively answer our research question by discussing specific categories of commits likely related to the code changes developers often forget to implement and try to immediately remedy. For each category, we present interesting examples and discuss implications for researchers and practitioners.

2.2 Results

We addressed our research question by labeling 500 commits identified as candidates to being quick remedy commits (see Section 2.1). We identified 42 false positives (i.e., commits c_{i+1} that were not related to the preceding c_i commit) and 458 commits actually classifiable as quick remedies.[1] Note that not all these quick remedy commits are compensatory fixes for issues caused by omitted changes. They also include fixes for previously introduced errors (e.g., the developer realizes that her previous commit introduced a bug) as well as commits aimed at simply improving the previously committed change (e.g., improve the name of a newly introduced variable). Finally, our taxonomy also features remedy commits aimed at fixing simple mistakes performed during the c_i commit process itself (e.g., the developer forgot to include a modified file in commit c_i and thus commits it in c_{i+1}).

Overall, we identified 69 types of quick remedy commits made by developers, 20 of which relevant for changes omitted in the previous commit.

Figure 4 presents the results in the form of a hierarchical taxonomy composed by six root categories: *Bug Fix*, *Code Refactoring/Clean Up*, *Build Issue*, *Missing Code Change*, *Documentation*, and *Reverted Commit*. The more specific types of quick remedy commits are represented either as intermediate nodes or leaves, and commits relevant for the fixing of issues caused by omitted changes are marked with a ❗ sign. For each category, we next

[1]Our online appendix features an analysis of common keywords present in the commit message of these commits in comparison to non-quick-remedy commits (Replication package 2021).

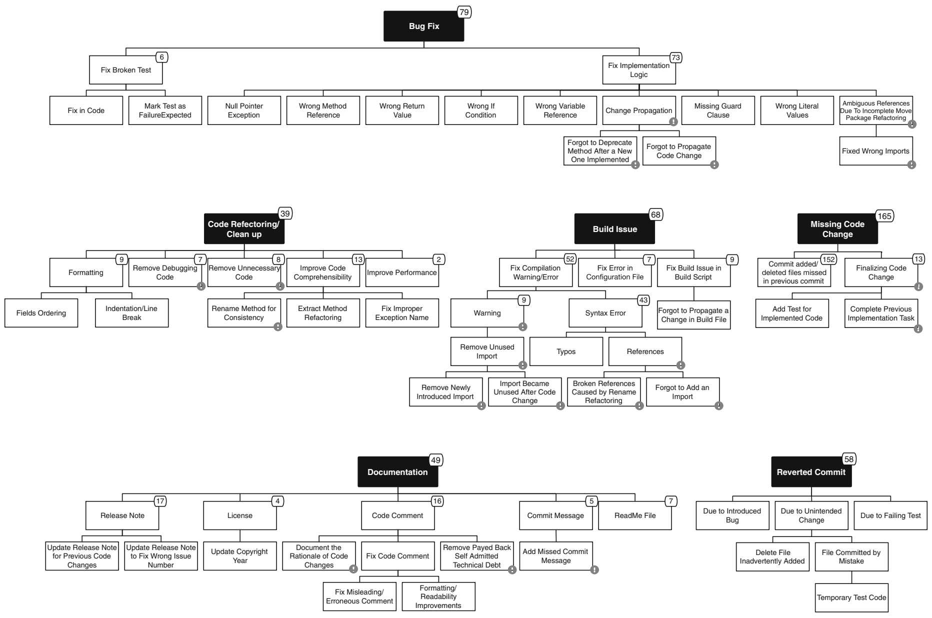

Bug Fix 79
Fix Broken Test 6
Fix Implementation Logic 73
Fix in Code
Mark Test as FailureExpected
Null Pointer Exception
Wrong Method Reference
Wrong Return Value
Wrong If Condition
Wrong Variable Reference
Change Propagation
Missing Guard Clause
Wrong Literal Values
Ambiguous References Due To Incomplete Move Package Refactoring
Forgot to Deprecate Method After a New One Implemented
Forgot to Propagate Code Change
Fixed Wrong Imports
Code Refectoring/ Clean up 39
Formatting 9
Remove Debugging Code 7
Remove Unnecessary Code 8
Improve Code Comprehensibility 13
Improve Performance 2
Fields Ordering
Indentation/Line Break
Rename Method for Consistency
Extract Method Refactoring
Fix Improper Exception Name
Build Issue 68
Fix Compilation Warning/Error 52
Fix Error in Configuration File 7
Fix Build Issue in Build Script 9
Warning 9
Syntax Error 43
Forgot to Propagate a Change in Build File
Remove Unused Import
Typos
References
Remove Newly Introduced Import
Import Became Unused After Code Change
Broken References Caused by Rename Refactoring
Forgot to Add an Import
Missing Code Change 165
Commit added/ deleted files missed in previous commit 152
Finalizing Code Change 13
Add Test for Implemented Code
Complete Previous Implementation Task
Documentation 49
Release Note 17
License 4
Code Comment 16
Commit Message 5
ReadMe File 7
Update Release Note for Previous Code Changes
Update Release Note to Fix Wrong Issue Number
Update Copyright Year
Document the Rationale of Code Changes
Fix Code Comment
Remove Payed Back Self Admitted Technical Debt
Add Missed Commit Message
Fix Misleading/ Erroneous Comment
Formatting/ Readability Improvements
Reverted Commit 58
Due to Introduced Bug
Due to Unintended Change
Due to Failing Test
Delete File Inadvertently Added
File Committed by Mistake
Temporary Test Code

Fig. 4 Taxonomy of quick remedy commits

describe representative examples and discuss implications for researchers (indicated with the icon) and/or practitioners (icon) derived from our findings.

2.2.1 Bug Fix (79)

This category groups pairs of commits (c_i, c_{i+1}) in which the remedy commit (i.e., c_{i+1}) fixes a bug introduced in c_i. We identified two main subcategories: *Fix Broken Test*, in which c_{i+1} has been triggered by test cases failing after the change implemented in c_i, and *Fix Implementation Logic*, in which the developer realized that she introduced a bug in c_i and quickly submits a patch.

The commits in the *Fix Broken Test* category targets the fixing of the production code or the test code modified in c_i that caused a break in the test suite. For example, in the `Denominator` project of `Netflix`, a developer reported in the commit message: "*Fix tests broken by former commits*" (Commit to denominator project on GitHub 2021).

While in the cases we analyzed the issue was spotted and fixed quickly by the developer, there might be non-trivial cases in which only a subset of the test suite is executed for regression testing (e.g., due to a limited testing budget) and a non-executed broken test is not identified by the developer.

For researchers, this is an opportunity to study test breaking-changes and to develop techniques able to alert the developer when a change she implemented might require a double check of (part of) the test suite. For practitioners, continuous integration practices can help in timely spotting these issues in most of the cases.

The fixes to the implementation logic are mostly classic bugs introduced but quickly recognized and fixed by developers (e.g., errors in `if` conditions, wrong literal values, null pointer exceptions, etc.). While these are not related to omitted changes, they are interesting since they represent bugs fixed by developers within five minutes (due to our selection criteria for the commits).

This indicates that these bugs, while prevalent in our taxonomy (73 instances), are likely quite simple to fix. Thus, researchers could investigate the possibility of creating approaches able to learn from this data on how to avoid and/or automatically fix these bugs. For example, recent work applied Neural Machine Translation (NMT) models to automatically fix bugs (Tufano et al. 2018). However, given the complexity of this task and the non-trivial bugs that these models have to fix, they are usually only able to automatically fix a minority of the bugs provided as input (Tufano et al. 2018). Focusing on these simpler but quite frequent bugs could represent a good application scenario for the NMT-based bug fixing approach.

Some of the fixes in the *Fix Implementation Logic* category are related to omitted changes (see Fig. 4). This includes the *Forgot to Propagate Code Change* category in which developers do not consistently propagate a change across all relevant code components. This is typical of cases in which code clones are spread in the system and inconsistent changes are implemented in c_i (Krinke 2007). An example of this can be seen in the $mathttTomP2P$ project. In a commit (Commit to TomP2P project on GitHub 2021b), the developers adapts a builder class (`PutBuilder`) to earlier changes of the original class and they implement new methods such as `isPutConfirm` and `isPutReject`. In a follow-up change (Commit to TomP2P project on GitHub 2021c), they fix a conditional statement to check the status of a `Put` object in a new branch. Then, only a few seconds later (Commit to tomp2p project on GitHub 2021a), they update a conditional check with a similar structure but in another class. For this last commit, the commit message says "*belongs to previous commit*". Another example can be seen in the $mathttspacewalk$ project. In a commit (Commit to spacewalk project on GitHub 2021a), they update a SQL script by adding a query for the removal of unnecessary data. Then, in the quick subsequent commit (Commit to spacewalk project on GitHub 2021b), they propagate the same schema changes into a database upgrade file.

These examples highlight the relevance for practitioners of approaches to guide code changes (see e.g., the seminal work in the area by Zimmermann et al. (2005)) as well as the need for the research community to continue improving these techniques and, possibly, making them easily pluggable into a continuous integration pipeline to foster developers' adoption.

Interesting in this category is also the introduction of *ambiguous references due to incomplete move package refactoring*. We found this case in the `apache/accumulo` project, where they migrate some classes to another package (Commit to Accumulo project on GitHub 2021), but still keep the old ones.

In a follow-up commit (Commit to accumulo project on GitHub 2021), they realize that they use, however, the wrong references to the migrated classes. Code clone detection techniques (Roy et al. 2009) could help in these cases by promptly pointing the developer to the presence of multiple copies of the same classes in the repository. The integration of these approaches in a just-in-time fashion could help in identifying clones introduced in the last commit, thus avoiding mistakes as the one in the discussed commit (Commit to Accumulo project on GitHub 2021).

2.2.2 Code Refactoring/Clean up (39)

This category groups the pairs of commits (c_i, c_{i+1}) in which the remedy commit (i.e., c_{i+1}) implements a refactoring/cleanup of the code changed in c_i (see Fig. 4). In these commits developers are either not satisfied of the code they implemented or are trying to address warnings received by static analyzers.

Some other subcategories include the simple removal of code that was only temporary implemented in c_i (i.e., *Remove Debugging Code*) or that becomes unnecessary after c_i's changes (i.e., *Remove Unnecessary Code*). Also, code formatting issues (e.g., mainly the inconsistencies of indentations and line breaks introduced with code changes) were fixed by developers in the remedy commit (ie *Code Formatting*). Additionally, in 2 cases, developers changed the code implemented in c_i to improve its performance. An example can be seen in project rzwitserloot/lombok (Commit to lombok project on GitHub 2021) where a developer fine tunes a cache clearing mechanism implemented in a previous commit by turning a variable volatile and moving the invocation for the cache clearing after a conditional check.

However, the main purpose of those code refactoring/clean up tasks is to improve the code understandability. Variable and method renaming refactoring (i.e., renaming a variable or method to better reflect its functionality) is the most common way to make the code easier to comprehend. Also popular are code transformations aimed at replacing literal values with variables or splitting long functions through extract method refactoring. The latter allows not only to foster comprehensibility, but also the reusability of small code snippets.

Other interesting cases are the ones in which developers modify the previously committed code to promote consistency with the coding style of the project (see e.g., *Rename Method for Consistency*). For example, in a commit of the project liferay − portal (Commit to liferay-portal project on GitHub 2021), developers opened an issue to "*introduce tests to document current behavior*" (Liferay Portal Issue LPS-44476 2021). Interestingly, in this process they very carefully review the used method names for better readability, and in a commit they say:

[...] where specific method names are NOT accurate, go for a generic name to force the developer to read the code to find what the method actually does.

The developers decided to change a method's name from assertThatSearchResultHasVersion to assertSearchResult. In the next commit (Commit to liferay-portal project on GitHub 2021), to remain consistent, they replace the method invocation of assertThatEverythingButSummaryIsEmpty (in another class) to assertSearchResult. For this last commit, the commit message says "*Match previous commit even though this method name was accurate*".

⚗ The inconsistencies fixed with simple refactorings point to the possibility for the software engineering research community to investigate techniques able to learn coding conventions used in a given system and recommend fixes for possible violations. To the best of our knowledge, the only attempt at date has been made by Allamanis et al. (2014) with their NATURALIZE tool able to recommend meaningful identifier names and formatting guidelines. Other approaches focus only on rename refactoring suggestions (Lin et al. 2017, 2017). While these techniques cover most of the inconsistencies fixed in the *Code Refactoring/Clean up* category (e.g., *Rename Method for Consistency*, *Fix Improper Exception Name*), others are left uncovered (e.g., *Fields Ordering*), indicating more potential for additional research in the area of recommending coding convention fixes.

2.2.3 Build Issue (68)

This category is related to commits fixing build issues introduced as a result of the c_i changes. The main subcategory here is the fix of the compilation errors/warnings issued by the compiler due to the changes in c_i (i.e., *Fix Compilation Warning/Error*). Unused import statements are the main cause for the warnings we identified (see Fig. 4), and the trigger for the remedy commits in this category. The unnecessary import statements are caused either by `import` statements introduced in c_i by the developer and then unused, or by previously existing `imports` becoming unused due to the changes implemented in c_i. These warnings are usually raised by static analysis checks performed at commit time and, thus, are easy to catch for developers.

In the *Syntax Error* category we found many cases of broken references due to rename refactoring operations performed in c_i. These rename refactorings are related to variables, methods, classes, as well as packages. An example can be seen in the commit (Commit to tower project on GitHub 2021) of the `DroidPlanner/Tower` project which followed a renaming of multiple classes. Some other cases were violating the syntax of the programming language due to introduced typos (e.g., missing statement separators).

Considering the good refactoring support provided by modern IDEs, the identification of these broken references as a consequence of refactorings was quite surprising for us. This may indicate either that these refactorings were performed manually, leading to the introduction of broken references, or that bugs might affect refactoring engines, as already found by previous work in the literature (Daniel et al. 2007). Additional investigation focused on these specific types of errors is needed to understand the reasons behind them.

Other subcategories that also caused a build issue include the fix of introduced errors in configuration files (i.e., *Fix Error in Configuration File*) or in a build script (i.e., *Fix Build Issue in Build Script*). For example, in some remedy commits developers fixed broken tags in configuration files or incorrect filepath references in build scripts.

2.2.4 Missing Code Change (165)

This category groups the pairs of commits (c_i, c_{i+1}) in which the remedy commit (i.e., c_{i+1}) adds some missing code changes that should be introduced within previous commit c_i. We divided those commits into two subcategories: *Commit Added/Deleted Files Missed in Previous Commit* and *Finalizing Code Change*.

The first subcategory is related to fixing a previous commit error. In this case, we are not referring to the code changes implemented in c_i, but to the commit process itself. This issue is mainly caused by an incorrect selection of committed files by the developer. Also, sometimes IDE cache issues can lead to a similar situation (e.g., the IDE cached the wrong version of a committed file or lost track of some code changes during the git commit process). While this subcategory is kind of unrelated to artifacts' changes, it still provides hints for interesting research directions. For example, approaches to automatically identify the set of files to commit can be designed to reduce the possibility of missing files or to include unrelated changes. This could also go further and recommend to the developer *when* to commit in such a way to avoid tangled commits (Herzig and Zeller 2013) and committing cohesive sets of code changes. To the best of our knowledge, the only step in this direction has been done by Bradley et al. (2018) with a context-aware developer assistant able to identify the files to push towards the repository when the developer asks. However, more automation can be envisioned, with approaches also able to (i) recommend when to commit (as previously said, to e.g., avoid tangled commits), and (ii) summarize the changes in a meaningful commit message (as attempted by Jiang et al. 2017).

The second subcategory (i.e., *Finalizing Code Change*) refers to code changes forgotten or left incomplete for other reasons in commit c_i that are then finalized in c_{i+1}. This includes cases in which developers add new test cases needed to test the production code introduced in the previous commit, or to complete an implementation task. For example, in a commit of the `openpnp` project (Commit to openpnp project on GitHub 2021), the developer claimed in the commit message that three new sub-features were introduced. However, the developer forgot to actually implement one of those sub-features and added the missing implementation in the following commit. In another case from the `geoserver` project (Commit to geoserver project on GitHub1 2021), the developer introduced a guard clause in commit c_i to check if a processed reference is `null`. Meanwhile, a debugging message was also added saying that "*the reference is null, reset it to default value*". However, the actual implementation for resetting this reference value was missing in commit c_i, and implemented in the remedy commit c_{i+1}. While these issues are of different natures, some of them can be spotted automatically through techniques comparing what is described in the commit message and what has been actually implemented in the change. For example, in the previously discussed example (Commit to openpnp project on GitHub 2021), a misalignment between the number of sub-features actually implemented and claimed in the commit message could be spotted and reported to the developer.

2.2.5 Reverted Commit (58)

This category groups remedy commits c_{i+1} in which the developers revert the code changes they committed in the previous commit c_i. The reasons pushing a developer to revert previous changes through a remedy commit include: (i) introduced bugs spotted after pushing the changes in c_i; (ii) unintended changes, pushed in c_i by mistake; (iii) failing test cases, possibly indicating a bug worth of investigation before applying the c_i's changes. In all these cases, developers prefer to quickly bring the code back to its previous state to double check the implemented changes and understand the causes for the (possible) introduced issues.

In many cases we were not able to understand the reasons behind the reverted changes by manually inspecting the subject commits. These cases are just grouped in the root category *Reverted Commit*. Also, we observed that sometimes the code changes were reverted backward and forward within a few subsequent commits.

Our study is not the first one investigating reverted commits in software repositories. Shimagaki et al. (2016) conducted a study to gain a better understanding of why commits are reverted in large software systems. They found that 1%-5% of the commits from the systems they studies are reverted and this number could be reduced by improving team communication and developers' awareness. However, in some cases, commits are reverted due to external factors (e.g., requirement change by end-users, customers, or remote teams) and, in this case, they are difficult to avoid. Yan et al. (2019) proposed a model to automatically identify commits that will be reverted in the future. They also found that the developer who performs the change is the most important predictive feature among the three they studied (i.e., code change, developer, commit message). Besides the recommendations to developers already provided by Shimagaki et al. (2016), the presence of reverted commits in the history of software systems is also relevant for the mining software repositories (MSR) research community. For example, it could be debated whether studies analyzing the change-proneness of code components (i.e., how frequently code components are subject to changes in software repositories) — e.g., Bieman et al. (2003), Catolino and Ferrucci (2019), and Aniche et al. (2018) — should take into account commits that are quickly reverted or, as currently done, should consider them. The same applies for works using the history of changes implemented by developers as a proxy for the developers' experience — e.g., Rahman et al. (2017) and Tufano et al. (2017). In Section 3 we present an empirical study aimed at assessing the impact of considering (or not) reverted commits for typical MSR data collection tasks.

2.2.6 Documentation (49)

Our last category groups remedy commits related to software documentation. These commits impact a number of documentation artifacts that represent the main subcategories (see Fig. 4), namely: release notes, licensing statements, code comments, commit messages, and readme files.

The errors fixed in release notes, licenses and readme files are mostly minor. For example, some commits just update the copyright year in a previously committed file. Also, the fixes of commit messages rarely happen, and are mostly due to adding a missing commit message for the code changes implemented in the previous commit. Also these cases are interesting for the MSR community. For example, approaches using pairs $\langle$*code changes implemented in a commit* c_x, *commit message of* $c_x\rangle$ to train models able to learn how to generate commit notes (Jiang et al. 2017), could be negatively biased by commit messages in a commit c_{i+1} referring to changes implemented in c_i.

Other remedy commits are related to code comments. In some cases, developers documented the rationale for a code change implemented in the previous commit. This is the case of commit (Commit to jitsi project on GitHub 2021a) performed in the `jitsi` project. In a commit (Commit to jitsi project on GitHub 2021b) they fix a bug due to the wrong generation of a message where they mistakenly set a value of a parameter to an empty string instead of a `null` value.

In the next commit (Commit to jitsi project on GitHub 2021a) they add a comment to explain the otherwise non-trivial difference in the generated message.

Interesting is also the missed removal of Self Admitted Technical Debt (SATD) instances (Potdar and Shihab 2014), meaning technical debt documented by developers in the code with comments such as `TODO`: ..., `TOFIX`: ..., etc. We found cases in which developers payed-back the technical debt instance, but forgot to remove the comment documenting the SATD. This resulted in a code-comment inconsistency (Wen et al. 2019), that could possibly confuse developers comprehending the associated code components. One representative example of this scenario is the commit (Commit to tinkerpop project on GitHub 2021a) performed in the `apache/tinkerpop` project where the developers "*Forgot to remove todo from previous commit*", as their commit message says. Indeed, in the remedy commit they remove a single-line comment which says "*todo: need a test to enforce this condition*", and just right in the previous commit (Commit to tinkerpop project on GitHub 2021b) they had implemented the missing test case, thus paying back the technical debt.

The cases discussed above for the *Documentation* category provide us with some interesting lessons learned. First, identifying code components in which specific types of comments (e.g., to document the rationale for a given implementation and/or to detail the application logic) are needed, can be a promising research direction. Second, automatically classify SATD as payed-back (or not) can help in identifying obsolete and misleading comments in the code. We believe this is another interesting research direction for the software engineering community.

3 Study II: On the Impact of Reverted Commits on MSR Data Collection

3.1 Study Design

The *goal* of the study is to investigate the impact of reverted commits (one subcategory of quick remedy commits) on data collection activities performed in the context of MSR studies. The *purpose* is to show the level of noise introduced by reverted commits in MSR studies collecting specific types of data. Our study addresses the following research question:

RQ$_2$: *What is the impact of reverted commits on data collection tasks when mining Java projects?*

We instantiate RQ$_2$ on two popular "data collection tasks", namely the identification of bug-fixing commits (Rodriguez-Perez et al. 2017b; Rodríguez-Pérez et al. 2018; Tufano et al. 2018; Wang et al. 2020; Penta et al. 2020) and of refactoring operations (Penta et al. 2020; Peruma 2019; Mahmoudi et al. 2019; Lin et al. 2019; Fakhoury et al. 2019; AlOmar et al. 2019) performed in the change history of software systems. We show the impact of filtering-out (or not) reverted commits while mining this data (e.g., a refactoring operation mined in the system's history in commit c_i may have been reverted in commit c_{i+1}, thus questioning its validity as a study data point). The results of our study help to increase the awareness about noisy data points introduced by reverted commits, eventually leading to a better handling of data processing in MSR studies.

3.1.1 Data Collection and Analysis

To answer RQ$_2$, we sorted the 1,497 projects used in the context of RQ$_1$ based on the number of commits in their change history impacting at least one source code (i.e., Java) file. We discarded seven projects having more than 100k of such commits since the data extraction process for refactoring operations on these systems is too costly in terms of time. In particular, we run the data collection process described in the following for two weeks, processing in parallel up to ten systems at a time. At the end of these two weeks, the seven systems we excluded were still far from being processed. We replaced these seven systems with those ranked in positions 101-107, still selecting a total of 100 repositories as context for RQ$_2$. The list of considered projects is available in our replication package (Replication package 2021).

From each of the 100 selected projects we extracted the following information:

- **Bug-fixing commits.** To identify bug-fixing commits in open-source repositories, we mined lexical patterns in commits, as done in previous work (Fischer et al. 2003). In particular, we used the pattern defined by Tufano et al. (2019), who reported a precision of 97.6% (i.e., 97.6% of commits identified by this heuristic as bug-fixes are true positives): The commit message must match the patterns *("fix" or "solve") and ("bug" or "issue" or "problem" or "error")* to classify the related commit as a bug-fix.
- **Refactoring operations.** To mine the refactoring operations in the history of a system at commit level we used the state-of-the-art tool RefactoringMiner (Tsantalis et al. 2018; Tsantalis et al. 2020). If at least one refactoring operation is identified in a given commit, we mark this commit as a "refactoring commit" and store the refactoring-related information (i.e., performed refactoring operations, code lines impacted by the refactoring).

- **Reverted commits.** Before detailing the procedure we adopted to identify reverted commits, it is important to clarify that, in our study, we only focus on identifying commits reverting Java code changes from the previous commit. This means that, as for our previous study, we are still in a scenario in which we are looking at pairs of commits c_i and c_{i+1}, with c_i being the reverted commit and c_{i+1} the reverting one. We implemented an approach similar to the one by Yan et al. (2019). First, we identify reverting commits by scanning commit messages, looking for the pattern *reverts commit* c_i. Second, to identify reverting commits c_{i+1} not explicitly labeled as such in their commit note, we compare the code they change with the one changed in the previous commit c_i. To do this, we stored the changes performed in each commit in a vector having the format: $\langle$*AddedFile, DeletedFile, ModifiedFile, AddedCode, DeletedCode*$\rangle$. We refer to this 5-element vector as a commit change vector V, in which *AddedFile* indicates the added file paths, *DeletedFile* the deleted file paths, *ModifiedFile* the modified file paths, and *AddedCode* and *DeletedCode* refer to the text in the inserted lines and removed lines, respectively, with each line added together with a prefix of the changed file path. Given the commits c_i and c_{i+1}, we mark c_i as a reverted commit and c_{i+1} as a reverting commit if they satisfy all of the following constraints:
  - $addedFile_{i+1} = deletedFile_i$,
  - $deletedFile_{i+1} = addedFile_i$,
  - $modifiedFile_{i+1} = modifiedFile_i$,
  - $addedCode_{i+1} = deletedCode_i$,
  - $deletedCode_{i+1} = addedCode_i$.
- **Partially reverted commits.** Similarly to the identification of completely reverted commits, given two commits c_i and c_{i+1}, we mark c_i as a partially reverted commit and c_{i+1} as a partially reverting commit if they satisfy all of the following constraints:
  - $addedFile_{i+1} \subset deletedFile_i$,
  - $deletedFile_{i+1} \subset addedFile_i$,
  - $modifiedFile_{i+1} \subset modifiedFile_i$
  - $addedCode_{i+1} \subset deletedCode_i$,
  - $deletedCode_{i+1} \subset addedCode_i$.

Once extracted the above described data from the change history of the 100 selected projects, we compute the impact of considering/not-considering completely and partially reverted commits when collecting bug-fixes and refactoring operations from the change history of software projects. In particular, given a task $T \in \{refactorings,\ bugfixes\}$, we compute for each project the average number of noisy data points introduced by a single reverted commit in the following way:

$$\frac{|DataPoints_{T_{all}} - DataPoints_{T_{cleaned}}|}{|reverted|}$$

where $DataPoints_{T_{all}}$ represents the total number of data points collected for the task T (i.e., in our case, number of bug-fixes or number of refactorings); $DataPoints_{T_{cleaned}}$ is the number of data points collected for the same task T when removing reverted commits; and $|reverted|$ is the total number of reverted commits identified in the repository. To make an example, in the case of T = mining of bug-fixing commits, a value for this metric of 0.5 indicates that every reverted commit introduces in the collected data, on average, 0.5

noisy bug-fixing commits. We compute the same metric when considering both reverted and partially reverted commits:

$$\frac{|DataPoints_{T_{all}} - DataPoints_{T_{cleaned'}}|}{|reverted| + |partiallyreverted|}$$

In this case, the only difference is that $DataPoints_{T_{cleaned'}}$ represents the number of data points collected for the task T when removing both reverted and partially reverted commits.

3.2 Results

We start by commenting on the number of fully and partially reverted commits we identified in the 100 systems. Overall, we found 5,083 reverted (avg=51, median=30, Q1=15, Q3=60) and 958 partially reverted (avg=10, median=7, Q1=3, Q3=13) commits. While the number of reverted commits is non-negligible, we only found a limited number of partially reverted commits, with a maximum of 44 observed for `apache/hbase`. Also, fully reverted commits are found in all repositories, while for the partially reverted ones we did not find any instance in six of the analyzed projects. Note that the number of reverted commits found in our paper is substantially lower as compared to the data reported in the work by Shimagaki et al. (2016) and Yan et al. (2019), in which up to 5% of commits in a repo were found to be reverted. However, it is worth noting that in our study, differently from previous work, we only considered reverting commits c_{i+1} that revert changes in c_i (e.g., we do not consider c_{i+1} as reverting commit if it reverts changes performed in c_{i-1}).

Figure 5 shows the results achieved for the data collection task related to bug-fixing commits. The 100 projects are sorted from the left to the right in ascending order by the absolute number of completely reverted commits. For example, the first project on the left is `hibernate/hibernate-search` with only one reverted commit in its change history, while the last is `apache/hbase` with 617. The stacked bar chart shows the number of non-impacted bug-fixing commits (i.e., commits that are not fully nor partially reverted)—blue

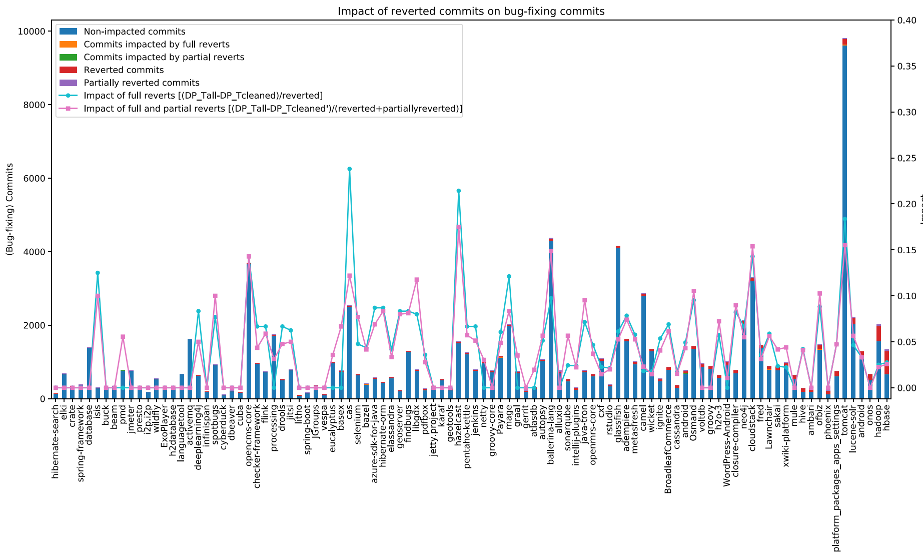


Fig. 5 Impact of reverted commits on bug-fixing commits

bar, of fully reverted bug-fixing commits (orange bar) and of partially reverted bug-fixing commits (green bar), using the scale on the left y-axis. The partially reverted commits are hardly visible in the chart due to their low number.

The line chart in Fig. 5 shows instead the average impact of fully reverted commits (cyan line) and of both fully and partially reverted commits (pink line) using the formulas presented at the end of Section 3.1. In this case, the reference y-axis is the one on the right. Since the number of partially reverted commits is very low, we limit our discussion to the impact of fully reverted commits on the collected bug-fixes. However, as it can be seen in the line chart in Fig. 5, the trend of the two lines is very similar.

Ignoring reverted commits from the data collection has an impact, in terms of collected data points, on 57 out of the 100 analyzed systems. The average impact goes from a minimum of 0.02 (i.e., a reverted commit results, on average, in 0.017 "wrong" bug fixes collected) to a maximum of 0.24, with an average of 0.07 and a median of 0.06. The system resulting in the highest number of noisy data points for this task is `apache/tomcat`, in which the 147 reverted commits cause the collection of 27 reverted bug-fixes (on average, each reverted commit contributes 0.18 noisy data points).

We discuss a few examples of commits that were identified as a bug-fixing commit but had been reverted in the subsequent commit.

One commit of the `apache/hadoop` project was marked bug-fixing https://github.com/apache/hadoop/commit/cb64e8eb192 as the log message said: "*Fix synchronization issues …*" The changes, however, were reverted by the next commit with the message "*Revert "Fix synchronization issues …" because forgot to add JIRA Number*." In this case, the reverted commit is indeed a bug-fixing commit, but the reverting commit should not be considered a valid bug-fix even though it contains the expression "Fix issues." In the worst case, a mining study might believe that there are already two bug-fixes in the change history after the revert. While in reality, the code does not implement the bug-fix after the revert.

In another commit of the `apache/tomcat` project https://github.com/apache/tomcat/commit/f711963768, the author claimed in the commit message that a reported issue had been fixed. However, the fix was reverted in the subsequent commit as they noticed that "*it fixes the reported issue but introduces other issues*." Again, the fix was reverted, and the reverting commit should not be counted as a bug-fix.

Another interesting example can be seen in https://github.com/aosp-mirror/platform_packages_apps_settings/commit/d3dcce029d. The bug-fix was reverted because the issue had been fixed before by someone else: "*Revert […] Bug: 27700406" Framework bug was fixed by ag/900274, so this is no longer needed.*"

It is important to highlight that, while there is an impact of the reverted commits on the collected bug-fixes (and, as such, excluding them from the data analysis might be preferred), such an impact is overall limited. However, it is also worth reminding that in our study design we favored the precision in the identification of reverted commits rather than recall. Thus, the number of reverted commits we identify is certainly an underestimation of the real ones. Also, in case these reverted bug-fixes are used to compute additional data (e.g., are provided as input to an SZZ algorithm as done in previous works (Penta et al. 2020)), such an error can further propagate and results in additional noisy data points. Basically, a cleaning of reverted commits when collecting bug-fixes is usually desirable, even though for specific study designs (e.g., collection of bug-fixing commits for qualitative manual analysis) it might not be needed.

Figure 6 shows the same data discussed before for the refactoring-related task, with the only difference that, in this case, the reverted and partially reverted commits are "refactoring

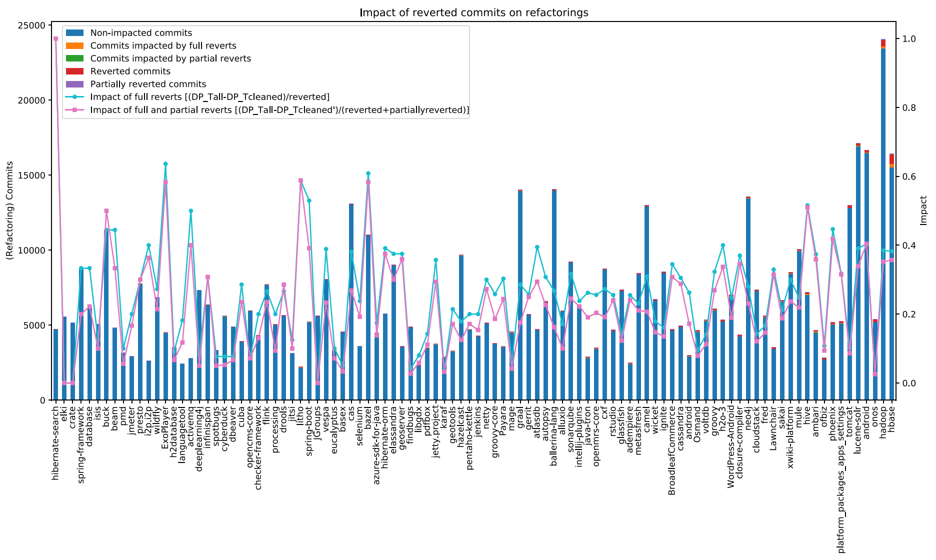


Fig. 6 Impact of reverted commits on refactoring commits

commits", meaning commits featuring at least one refactoring operation. Also in this case we focus our discussion on the completely reverted commits.

Considering reverted commits during the data collection has an impact on 97 out of the 100 systems, with an average impact for a single reverted commit of 0.27 noisy data points (i.e., reverted refactoring commits), median=0.26. The average impact goes from a minimum of 0.08 to a maximum of 1.00. The latter is a sort of outlier, since it refers to the `hibernate/hibernate-search` that, as said before, does only have one reverted commit that is indeed a refactoring commit.

In this case, the system that would be mostly affected by the presence of noisy refactoring commits collected when not handling reverted commits is `apache/hbase` with a total of 236 reverted refactoring commits that would be wrongly considered (result of the overall 617 reverted commits in this system).

An example of refactoring-related commits that have been reverted is the one commit performed in the `metasfresh/metasfresh` project https://github.com/metasfresh/metasfresh/commit/7875c81632. The developer performed some refactoring operations (e.g., rename parameter, change return type, rename method), but the commit message claimed that the refactoring was only partially. The subsequent commit reverted this partial refactoring. Thus, specific types of empirical studies mining refactoring operations may consider ignoring the refactorings detected in the first commit, since the refactorings were implemented and quickly reverted by the developer.

In another commit performed in the `wordpress − mobile/WordPress − Android` project,[2] one of the private inner classes has been moved to a public outer class through a move class refactoring. However, the author said that this refactoring was only for testing purpose and reverted the change in the subsequent commit.

[2]https://github.com/wordpress-mobile/WordPress-Android/commit/c363f2ff2d

As compared to the collection of bug-fix commits, reverted commits seem to have a higher impact when mining refactoring operations, with an overall of 1,447 reverted (noisy) refactoring commits that are identified across the 100 analyzed systems. Considering our conservative approach to identify reverted commits, we believe its cleaning is highly recommended when studying refactoring operations over the history of software systems.

3.3 Summing Up

Both the quantitative and qualitative results of this study point to an opportunity to obtain cleaner data by considering reverted commits: Reverted commits are noise in the recorded history of a system, and while it looks like a negligible phenomenon, we argue that the cleaner the data the better the analyses. In the spirit of the work by Kawrykow and Robillard (2011) on cleaning out non-essential changes from any mining software repositories research, detecting and removing reverted commits could thus also become a part of the cleaning preprocessing before starting an actual analysis.

4 Threats to Validity

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to (Study I) the manual analysis we performed to identify the reasons behind the quick remedy changes performed by developers, and (Study II) the heuristics used to identify bug-fixing commits and reverted commits as well as to imprecisions introduced by the tool used to mine refactoring operations.

To mitigate subjectivity bias in the manual analysis (Study I), every commit was assigned to two authors who manually analyzed it independently. Then, in the case of disagreement, a third author was assigned to the commit to solve the conflict. In addition to that, we used lexical patterns to identify candidate remedy commits. While these lexical patterns can return false positives, these have been excluded in our study through manual validation, and thus do not influence our findings.

Concerning Study II, the identification of bug-fixing commits was based on a heuristic defined and validated in previous work (Tufano et al. 2019). As for the reverted commits, we combined two types of heuristics based on the analysis of the commit message and of the code changes. Also, we limited the identification of reverted commits only to pairs of subsequent commits to increase the precision in our analysis. While this likely reduces the number of reverted commits we can identify (i.e., recall), considering the analysis we performed (i.e., assessing the average "cost" in terms of noisy data of a single reverted commit) our findings should not be substantially affected. Finally, refactoring operations have been mined by relying on the state-of-the-art tool RefactoringMiner (Tsantalis et al. 2020).

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. One aspect could be related to the selection of projects being considered. As explained by Kalliamvakou et al. (2014) mining GitHub can be risky because projects may contain very few commits. To mitigate this threat, we applied strict criteria (i.e., more than 500 commits, more than ten stars) when selecting the context of our study. Also, we manually looked into the set of retrieved projects to exclude repositories that do not represent real software systems (e.g., tutorials, collections of code examples) and forked projects. Also, in Study I all considered data points (i.e., commits) have been manually checked, strengthening its internal validity.

Threats to *external validity* concern the generalizability of our findings. Our analysis in Study I is limited to a specific set of 500 commits we randomly selected as the output of a keyword-based mechanism used for the pre-selection of commits likely to be "remedy" commits. Because of this procedure, our taxonomy inevitably omits types of remedy commits we did not analyze and/or documented in diverse data sources. Also, we set a 5-minute threshold to identify the *quick remedy commits* subject of our study. While our choice is justified by the temporal distribution plotted in Fig. 2, changing this threshold value may result in different findings. This investigation is part of our future research agenda.

As for Study II, the reported findings are related to a set of 100 Java open source projects, which do not allow us to generalize our results to projects written in other languages which require additional investigations.

5 Related Work

There is a vast literature of empirical studies investigating developers' commits for various purposes. Many studies tackle research questions related to when, where, or why developers change source code. However, there has been little research on quick fixes, or consecutive changes performed by software developers, as well as on the impact of specific types of commits in the data collection of MSR studies. Here we present an overview of the related work close to the topic of this article.

5.1 Reasons for Changes

Mockus and Votta (2000) studied a large legacy telecommunication system to identify reasons for software changes. Using an automatic classification algorithm, they discovered three primary reasons for changes according to maintenance activities: adding new functionality (*adaptive*), repairing faults (*corrective*), and restructuring the code to accommodate future changes (*perfective*). They noticed that several changes fall under the fourth category of inspection rework changes, i.e., changes to implement the recommendations of code inspections. They also found a strong relationship between the type and size of a change and the difficulty of a change.

Hattori and Lanza (2008) conducted an empirical study on nine large open source systems. They defined the size of a commit based on the number of files. They classified commits according to the comments information into development or maintenance (reengineering, corrective engineering, and management).

Hindle et al. (2008) conducted a study on large commits, created a taxonomy of their purpose. They found that large commits are more focused on perfective maintenance, while small commits are more related to corrective maintenance.

5.2 Effects of a Change on Quality

Small Changes Purushothaman and Perry (2005) investigated small source code changes (i.e., one-line changes) during the development process. An interesting finding of their work is that there is less than a four percent probability that a one-line change introduces a fault in the code.

Large Changes Śliwerski et al. (2005) studied fix-inducing changes, i.e., changes that lead to problems indicated by fixes. In particular, they investigated the day of the week and the size of commits in Eclipse and Mozilla. They found that the commits performed on Friday and large commits have higher chances of introducing bugs.

Social Characteristics Eyolfson et al. (2011) investigated the bug-fix time as the time from the earliest commit that introduced the bug to the bug-fixing commit. Their findings suggest that the time and date of a code update may affect the quality of the code.

In an earlier study, Claes et al. (2018) also studied developers' working hours by investigating the timestamps of commit activities. They found that developers mainly work in regular office hours, and they did not find support that project maturation would decrease irregular working hours.

Bird et al. (2011) mined commits in Windows Vista and Windows 7 to investigate the relationship between code ownership and software quality. They found that high levels of ownership, specifically high values for the proportion of ownership for the top owners, or high values for major, and low values of minor contributors, are associated with fewer defects.

Rahman and Devanbu (2011) found that implicated code is more closely related to the contribution of a single developer. Their findings also indicate that an author's specialized experience in the target file is more important than general experience.

Gonzalez-Barahona et al. (2011) investigated in FLOSS projects from the Mozilla community whether contributors fixing a bug are the same introducing and seeding them in the first place. Their results show that in 80% of the cases, the bug-fixing activity involves source code modified by at most two developers. Hence, in most of the cases, the bug fixing process is not carried out by the same developers.

Supplementary Patches Park et al. (2012) studied bugs whose initial patches were later considered incomplete and to which programmers applied supplementary patches. They examined three open source projects: Eclipse JDT core, Eclipse SWT, and Mozilla. They found that a significant portion of bugs fall in this category while their causes are often diverse, e.g., missed port changes, incorrect handling of conditional statements, or incomplete refactoring. In their follow-up work (Park et al. 2014; 2017) they further investigated supplementary patches, and the results showed that only 7% to 17% of supplementary patches had content similar to their initial patches, which implies that a separate code clone analysis could not predict the supplementary patch location.

An et al. (2014) found that supplementary bug fixes accounted for 10.3% to 26.9% of total bug reports. Also, in the subject systems, a high percentage of the supplementary fixes (i.e., from 21.6% to 33.8%) had been re-opened.

Consecutive Changes Dai et al. (2014) investigated the relationship between consecutive changes and software quality. They studied two concepts of consecutive changes: chain of consecutive bug-fixing file versions, and chain of consecutive file versions where each pair of adjacent versions has different authors. They found that those consecutive changes have a negative impact on the later file versions in the short term, especially when the length of the change chain is four or five.

Inconsistent Changes Bettenburg et al. (2012) conducted an empirical study on inconsistent changes to code clones in two large open source software systems. They observed that the number of defects caused by inconsistent changes to code clones was substantially lower at the release level, compared to the revision level. Their findings suggest that developers can effectively manage and control the evolution of cloned code at the release level.

Incorrect Changes Yin et al. (2011) presented a comprehensive characteristic study on incorrect bug-fixes from large operating system code bases, including Linux, OpenSolaris, and FreeBSD. They found that at least 14.8%-24.4% of sampled fixes for post-release bugs in these large operating systems were incorrect.

Changes and Refactoring Palomba et al. (2017) conducted a quantitative investigation of the relationship between different types of code changes and different refactoring types. They found that developers tend to apply a higher number of refactoring operations when they are fixing bugs.

Bavota et al. (2012) presented a study aimed at investigating to what extent refactoring activities induce faults. They showed that refactorings involving hierarchies (e.g., *pull down method*) induce faults very frequently. Conversely, other kinds of refactorings are likely to be harmless in practice.

5.3 Changes and Time

Rodriguez-Perez et al. (2017a) conducted two case studies and studied the *Time To Notify* (TNN) metric which describes how much time it takes for a bug to be notified/reported since the bug was introduced into the source code. They examined how this metric is related to software maintenance and evolution. Interestingly, they found relatively high mean values of TTN in the projects: 312 and 431 days.

Kim and Whitehead (2006) studied the bug-fix time of files in ArgoUML and PostgreSQL. Their statistics showed that fixing 50% of the bugs requires 100 to 300 days, while the median bug-fix time is about 200 days.

5.4 Change Patterns

Pan et al. (2009) presented an automatic approach in which software history data is mined to find patterns in bug fix changes and automatically categorize bugs. They defined bug fix patterns (e.g., method call with different actual parameter values) which covered 45-63% of bug fixes in seven open source projects.

Zhao et al. (2017) conducted an empirical study to investigate the characteristics of change types in bug fixing code. They proposed a change classification schema and developed an automatic classification tool to categorize changes into five change types. They found that interface-related code changes are the most frequent bug-fixing changes.

In a related research thread, Martinez and Monperrus (2019) presented Coming, a tool to mine change pattern instances from git commits.

Change patterns have also been exploited recently to train neural networks in order to automatically reproduce code changes implemented by developers in pull requests of open source projects (Tufano et al. 2019) or to learn how to automatically fix bugs (Tufano et al. 2018).

5.5 Bias and Noise in Mining Change Histories

Many approaches and studies depend on the quality of the dataset produced by mining change histories. Discussions about the bias in data collected by mining repositories have gained more attention recently. As Bird et al. (2009) say in a study on bias in bug-fix datasets: "*bias is a critical problem that threatens both the effectiveness of processes that rely on biased datasets to build prediction models and the generalizability of hypotheses* *tested on biased data*". Here, we overview the potential causes and impact of bias and noise in mining studies.

Impact of Non-Essential Changes Kawrykow and Robillard (2011) observed that software changes are often accompanied by non-essential modifications, such as local variable refactorings, or textual differences induced as part of a rename refactoring. They studied code changes in over 24,000 changesets of seven open-source systems and observed non-essential changes in their history. They found that up to 15.5% of a system's method updates were due to non-essential differences among interesting observations.

The authors also investigated the impact of non-essential changes on change-based analyses in their same research work (Kawrykow and Robillard 2011). They implement a method-pair association rule mining analysis similar to the approach of Zimmermann et al. (2004). This approach, given a set of changes, suggests and predicts likely further changes. They found that removing non-essential method updates improved the precision of the recommendations by 10.5% and decreased their recall by 4.2%.

Impact of Tangled Changes Herzig and Zeller (2013) defined a tangle change as a single commit which consists of separate changes (e.g., fixing a bug and adding a new feature). They found that up to 15% of all bug fixes include tangled changes.

Later, they also showed that tangled changes could significantly impact the accuracy of defect prediction models assessed in empirical studies (Herzig et al. 2016).

Impact of Untracked Changes Hora et al. (2018) claimed that changes affecting code entities' names (untracked changes) present a potential threat to MSR studies. For example, a method rename could be misinterpreted as the deletion and the addition of a method, thus, splitting its history. Based on an empirical analysis of 15 Java systems, they found that between 10 and 21% of the method level changes are untracked, hence, should be systematically considered by MSR studies.

Bias in Bug Localization and Prediction Kochhar et al. studied biases in bug localization (Kochhar et al. 2014). They identified potential causes that can impact the validity of the results reported in studies. One of the main reasons is that files modified in commits that fix the bugs might not contain the bug. Instead, files are often changed because of refactorings or modifications to program comments.

Kim et al. (2011) measured the impact of noise on defect prediction models built using historical defect data obtained by mining software repositories. They consider false positives and false negatives as noise in such dataset. They found that, for large defect datasets, noises alone do not lead to substantial performance differences. However, their prediction performance decreased significantly when the dataset contained 20%-35% of both FP and FN noises.

Rahman et al. (2013) assessed whether the size of the dataset or bias affects the performance of defect prediction approaches. Similar to the findings of Kim et al. (2011), they conclude that size matters at least as much as bias.

Noise in History Slicing Li et al. (2016) presented a semantic history slicing approach to extract changes related to a particular functionality. As they say, state-of-the-art techniques tend to over-approximate the inferred changes, and their slice histories may contain irrelevant changes. Their approach implements a method to untangle unrelated changes introduced in a single commit.

Threats in Aggregating Software Repository Data Robillard et al. (2018) investigated potential threats to validity associated with metrics that summarize software repository data. They conducted a case study in which they retrieved and analyzed every file considered abandoned to investigate the files' properties, including size, file type, and amount of comments. As a result, they identified eight major threats that can generalize to software process metrics derived from repository data. These threats are fragility, file content, file role, comment, contributor involvement, quantization, architectural sensitivity, and exceptional action.

5.6 Summing Up

As discussed above, previous work investigated code changes from several different points of view. However, to the best of our knowledge, our study is the first to investigate the impact of reverted commits on data collected by mining the versioning system (and, in particular, big-fixing commits and refactoring operations).

6 Conclusion

We presented two empirical studies related to *quick remedy commits*. In the first, we qualitatively investigate *quick remedy commits* performed by developers in GitHub projects. We defined *quick remedy commits* as commits performed by developers to remedy changes omitted or errors introduced in a previous commit, performed just a few minutes before. This study (Study I) is based on the manual analysis of 500 commits, that we classified by looking at the objective of the remedy commit. The output of this study is represented by the taxonomy depicted in Fig. 4. We used several qualitative findings to distill lessons learned resulting in actionable items for both researchers and practitioners, which are summarized in Fig. 7.

Then, we investigated the impact of a specific type of quick remedy commits, namely reverted commits, on the data extracted for MSR studies. In particular, we focused on two data collection tasks performed in many previous works: (i) the identification of bug-fixing commits and (ii) the mining of refactoring operations over the change history of a system. Our analysis disclosed the amount of potential noise brought by reverted commits for these two data collection tasks.

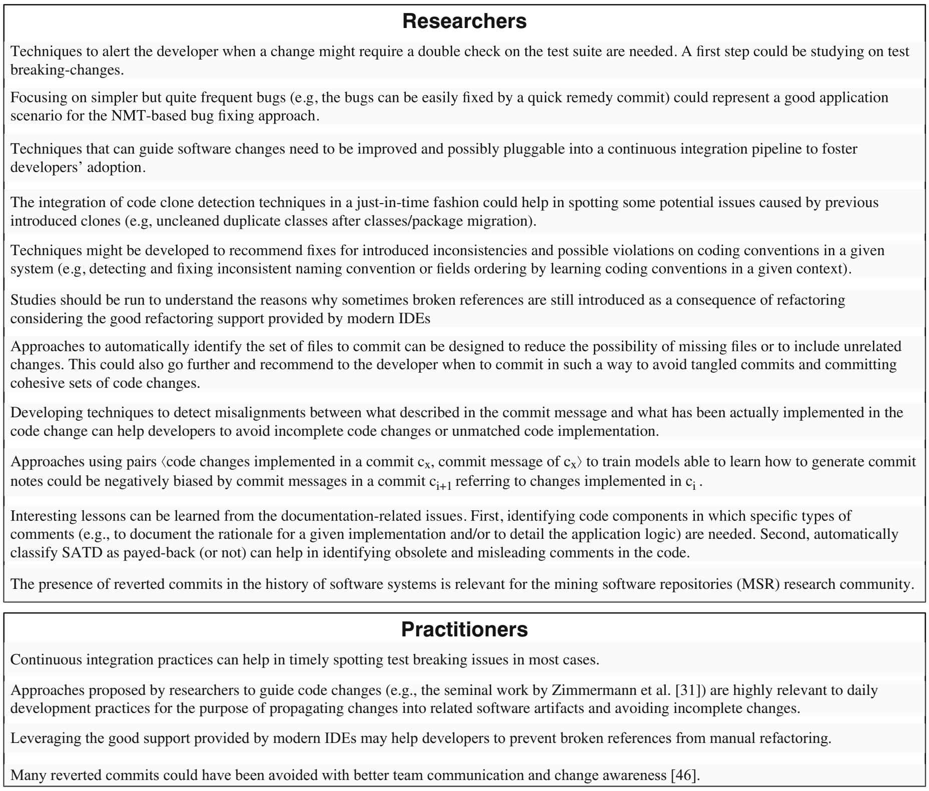
6.1 Future Work

Our future work will target two directions. First, we will work on some of the research directions discussed in the results section of Study I, and summarized in the following:

Automatic bug fixing Developing approaches able to learn how to automatically fix the "simple" bugs that, as shown in our study, are fixed by developers within a few minutes from their introduction. We believe that approaches based on deep learning (see e.g., Tufano et al. 2018) can be particularly performant in this specific context.

Automatic identification of omitted changes Integrating approaches to identify locations for missed code changes in a continuous integration pipeline, to alert developers when changes they are committing are likely to be incomplete.

Learning coding conventions Investigating novel techniques to learn coding conventions, enlarging the set of conventions that are currently supported by state-of-the-art techniques



Researchers

Techniques to alert the developer when a change might require a double check on the test suite are needed. A first step could be studying on test breaking-changes.

Focusing on simpler but quite frequent bugs (e.g, the bugs can be easily fixed by a quick remedy commit) could represent a good application scenario for the NMT-based bug fixing approach.

Techniques that can guide software changes need to be improved and possibly pluggable into a continuous integration pipeline to foster developers' adoption.

The integration of code clone detection techniques in a just-in-time fashion could help in spotting some potential issues caused by previous introduced clones (e.g, uncleaned duplicate classes after classes/package migration).

Techniques might be developed to recommend fixes for introduced inconsistencies and possible violations on coding conventions in a given system (e.g, detecting and fixing inconsistent naming convention or fields ordering by learning coding conventions in a given context).

Studies should be run to understand the reasons why sometimes broken references are still introduced as a consequence of refactoring considering the good refactoring support provided by modern IDEs

Approaches to automatically identify the set of files to commit can be designed to reduce the possibility of missing files or to include unrelated changes. This could also go further and recommend to the developer when to commit in such a way to avoid tangled commits and committing cohesive sets of code changes.

Developing techniques to detect misalignments between what described in the commit message and what has been actually implemented in the code change can help developers to avoid incomplete code changes or unmatched code implementation.

Approaches using pairs ⟨code changes implemented in a commit c_x, commit message of c_x⟩ to train models able to learn how to generate commit notes could be negatively biased by commit messages in a commit c_{i+1} referring to changes implemented in c_i .

Interesting lessons can be learned from the documentation-related issues. First, identifying code components in which specific types of comments (e.g., to document the rationale for a given implementation and/or to detail the application logic) are needed. Second, automatically classify SATD as payed-back (or not) can help in identifying obsolete and misleading comments in the code.

The presence of reverted commits in the history of software systems is relevant for the mining software repositories (MSR) research community.

Practitioners

Continuous integration practices can help in timely spotting test breaking issues in most cases.

Approaches proposed by researchers to guide code changes (e.g., the seminal work by Zimmermann et al. [31]) are highly relevant to daily development practices for the purpose of propagating changes into related software artifacts and avoiding incomplete changes.

Leveraging the good support provided by modern IDEs may help developers to prevent broken references from manual refactoring.

Many reverted commits could have been avoided with better team communication and change awareness [46].

Fig. 7 Summary implications for researchers and practitioners

(Allamanis et al. 2014). Once learned, the coding conventions can be automatically checked on the code to commit, raising a warning in case violations are detected.

Automatic software documentation Developing techniques able to (i) identify code components in which specific types of comments (e.g., rationale for implementation choices) are needed; and (ii) automatically classify SATD as payed-back (or not).

Second, we will analyze the impact of other types of *quick remedy commits* on the outcome of MSR studies. For example, the results of works mining logical coupling between components (i.e., how often specific files co-change), can be impacted by considering a *quick remedy commit* c_{i+1} as part of its previous commit c_i, since they basically represent the same implementation activity.

Acknowledgments We gratefully acknowledge the financial support of the projects PROBE (SNF Project 172799), INSTINCT (SNF-FNRS Project 190113), and SENSOR (SNF-JSPS Project 183587).

Funding Open Access funding provided by Università della Svizzera italiana.

References

About stars (GitHub) (2021) https://help.github.com/articles/about-stars/

AlOmar E, Mkaouer MW, Ouni A (2019) Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In: 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), pp 51–58

Allamanis M, T Barr E, Bird C, Sutton C (2014) Learning natural coding conventions. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014, pp 281–293

An L, Khomh F, Adams B (2014) Supplementary bug fixes vs. re-opened bugs. In: Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, ser. SCAM '14. IEEE Computer Society, Washington, pp 205–214

Aniche MF, Bavota G, Treude C, Gerosa MA, van Deursen A (2018) Code smells for model-view-controller architectures. Empir Softw Eng 23(4):2121–2157

Bavota G, Carluccio BD, Lucia AD, Penta MD, Oliveto R, Strollo O (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, pp 104–113

Bettenburg N, Shang W, Ibrahim WM, Adams B, Zou Y, Hassan AE (2012) An empirical study on inconsistent changes to code clones at the release level. Sci Comput Program 77(6):760–776

Bieman JM, Andrews AA, Yang HJ (2003) Understanding change-proneness in oo software through visualization. In: 11th IEEE International Workshop on Program Comprehension, 2003, pp 44–53

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (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 Symposium on the Foundations of Software Engineering, ser. ESEC/FSE '09. Association for Computing Machinery, New York, pp 121–130. https://doi.org/10.1145/1595696.1595716

Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't touch my code!: Examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ser. ESEC/FSE '11. ACM, New York, pp 4–14

Bradley NC, Fritz T, Holmes R (2018) Context-aware conversational developer assistants. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp 993–1003

Catolino G, Ferrucci F (2019) An extensive evaluation of ensemble techniques for software change prediction. J Softw Evol Process 31(9)

Claes M, Mäntylä MV, Kuutila M, Adams B (2018) Do programmers work at night or during the weekend? In: Proceedings of the 40th International Conference on Software Engineering, ser. ICSE '18. ACM, New York, pp 705–715

Commit to Accumulo project on GitHub (2021) [Online]. Available: https://github.com/apache/accumulo/commit/2ad672a

Commit to denominator project on GitHub (2021) [Online]. Available: https://github.com/Netflix/denominator/commit/e727b9d

Commit to liferay-portal project on GitHub (2021) [Online]. Available: https://github.com/liferay/liferay-portal/commit/1b5c378d4785

Commit to lombok project on GitHub (2021) [Online]. Available: https://github.com/projectlombok/lombok/commit/57f59074

Commit to tomp2p project on GitHub (2021) [Online]. Available: https://github.com/tomp2p/TomP2P/commit/4bc6e824

Commit to TomP2P project on GitHub (2021) [Online]. Available: https://github.com/tomp2p/TomP2P/commit/3db803c

Commit to TomP2P project on GitHub (2021) [Online]. Available: https://github.com/tomp2p/TomP2P/commit/8802c5e

Commit to accumulo project on GitHub (2021) [Online]. Available: https://github.com/apache/accumulo/commit/b8859513a

Commit to jitsi project on GitHub (2021) [Online]. Available: https://github.com/jitsi/jitsi/commit/6a361bbf6

Commit to jitsi project on GitHub (2021) [Online]. Available: https://github.com/jitsi/jitsi/commit/a74af45

Commit to spacewalk project on GitHub (2021) [Online]. Available: https://github.com/spacewalkproject/spacewalk/commit/6df7327

Commit to spacewalk project on GitHub (2021) [Online]. Available: https://github.com/spacewalkproject/spacewalk/commit/fec7040

Commit to tinkerpop project on GitHub (2021) [Online]. Available: https://github.com/apache/tinkerpop/commit/a4c62be7a5

Commit to tinkerpop project on GitHub (2021) [Online]. Available: https://github.com/apache/tinkerpop/commit/aa3d538

Commit to tower project on GitHub (2021) [Online]. Available: https://github.com/DroidPlanner/Tower/commit/72132d049

Commit to openpnp project on GitHub (2021) [Online]. Available: https://github.com/openpnp/openpnp/commit/aeef4cb0e4

Commit to geoserver project on GitHub1 (2021) [Online]. Available: https://github.com/geoserver/geoserver/commit/22c89ad106

Dai M, Shen B, Zhang T, Zhao M (2014) Impact of consecutive changes on later file versions. In: Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies, ser. EAST 2014. ACM, New York, pp 17–24

Daniel B, Dig D, Garcia K, Marinov D (2007) Automated testing of refactoring engines. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ser. ESEC-FSE '07, pp 185–194

Eyolfson J, Tan L, Lam P (2011) Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th Working Conference on Mining Software Repositories, ser. MSR '11. ACM, New York, pp 153–162

Fakhoury S, Roy D, Hassan A, Arnaoudova V (2019) Improving source code readability: Theory and practice. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp 2–12

Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: 19th International Conference on Software Maintenance (ICSM 2003), the Architecture of Existing Systems, 22-26 September, 2003, Amsterdam, Netherlands, p 23

Gonzalez-Barahona JM, Izquierdo-Cortazar D, Capiluppi A (2011) Are developers fixing their own bugs?: Tracing bug-fixing and bug-seeding committers. Int J Open Source Softw Process 3(2):23–42

Hattori LP, Lanza M (2008) On the nature of commits. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, ser. ASE'08. IEEE Press, Piscataway, pp III–63–III–71

Herzig K, Just S, Zeller A (2016) The impact of tangled code changes on defect prediction models. Empir Softw Eng 21(2):303–336

Herzig K, Zeller A (2013) The impact of tangled code changes. In: Proceedings of the 10th Working Conference on Mining Software Repositories, ser. MSR '13. IEEE Press, Piscataway, pp 121–130

Hindle A, German DM, Holt R (2008) What do large commits tell us?: A taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, ser. MSR '08. ACM, New York, pp 99–108

Hora A, Silva D, Valente MT, Robbes R (2018) Assessing the threat of untracked changes in software evolution. In: Proceedings of the 40th International Conference on Software Engineering, ser. ICSE '18. ACM, pp 1102–1113

Jiang S, Armaly A, McMillan C (2017) Automatically generating commit messages from diffs using neural machine translation. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp 135–146

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ser. MSR 2014, pp 92–101

Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: Proceedings of the 33rd International Conference on Software Engineering, ser. ICSE '11. ACM, pp 351–360

Kim S, Whitehead EJ Jr (2006) How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories, ser. MSR '06. ACM, New York, pp 173–174

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Proceedings of the 33rd International Conference on Software Engineering, ser. ICSE '11. ACM, pp 481–490

Kochhar PS, Tian Y, Lo D (2014) Potential biases in bug localization: Do they matter? In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ser. ASE '14. ACM, pp 803–814

Krinke J (2007) A study of consistent and inconsistent changes to code clones. In: 14th Working Conference on Reverse Engineering (WCRE 2007), pp 170–178

Li Y, Zhu C, Rubin J, Chechik M (2016) Precise semantic history slicing through dynamic delta refinement. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ser. ASE 2016. ACM, pp 495–506

Liferay Portal Issue LPS-44476 (2021) [Online]. Available: https://issues.liferay.com/browse/LPS-44476

Lin B, Scalabrino S, Mocci A, Oliveto R, Bavota G, Lanza M (2017, 2017) Investigating the use of code analysis and NLP to promote a consistent usage of identifiers. In: 17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, pp 81–90

Lin B, Nagy C, Bavota G, Lanza M (2019) On the impact of refactoring operations on code naturalness. In: Wang X, Lo D, Shihab E (eds) 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER. IEEE, pp 594–598

Mahmoudi M, Nadi S, Tsantalis N (2019) Are refactorings to blame? an empirical study of refactorings in merge conflicts. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 151–162

Martinez M, Monperrus M (2019) Coming: A tool for mining change pattern instances from git commits. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ser. ICSE '19. IEEE Press, Piscataway, pp 79–82

Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: Proceedings of the International Conference on Software Maintenance (ICSM'00), ser. ICSM '00. IEEE Computer Society, Washington, pp 120–

Palomba F, Zaidman A, Oliveto R, De Lucia A (2017) An exploratory study on the relationship between changes and refactoring. In: Proceedings of the 25th International Conference on Program Comprehension, ser. ICPC '17. IEEE Press, pp 176–185

Pan K, Kim S, Whitehead EJ Jr (2009) Toward an understanding of bug fix patterns. Empir Softw Eng 14(3):286–315

Park J, Kim M, Bae D-H (2014) An empirical study on reducing omission errors in practice. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ser. ASE '14. ACM, New York, pp 121–126

Park J, Kim M, Bae D-H (2017) An empirical study of supplementary patches in open source projects. Empir Softw Eng 22(1):436–473

Park J, Kim M, Ray B, Bae D (2012) An empirical study of supplementary bug fixes. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), pp 40–49

Penta MD, Bavota G, Zampetti F (2020) On the relationship between refactoring actions and bugs: a differentiated replication. In: Devanbu P, Cohen MB, Zimmermann T (eds) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp 556–567

Peruma A (2019) A preliminary study of android refactorings. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp 148–149

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp 91–100

Project FindBugs on GitHub (2021) [Online]. Available: https://github.com/findbugsproject/findbugs

Project SpotBugs on GitHub (2021) [Online]. Available: https://github.com/spotbugs/spotbugs

Project java-design-patterns on GitHub (2021) [Online]. Available: https://github.com/iluwatar/java-design-patterns

Project spring-petclinic on GitHub (2021) [Online]. Available: https://github.com/spring-projects/spring-petclinic

Purushothaman R, Perry DE (2005) Toward understanding the rhetoric of small source code changes. IEEE Trans Softw Eng 31(6):511–526

Rahman F, Devanbu P (2011) Ownership, experience and defects: A fine-grained study of authorship. In: Proceedings of the 33rd International Conference on Software Engineering, ser. ICSE '11. ACM, New York, pp 491–500

Rahman F, Posnett D, Herraiz I, Devanbu P (2013) Sample size vs. bias in defect prediction. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013. ACM, pp 147–157

Rahman MM, Roy CK, Kula RG (2017) Predicting usefulness of code review comments using textual features and developer experience. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp 215–226

Replication package (2021) https://github.com/USI-INF-Software/EMSE-ICPC2020-quick-remedy-commit

Rigby PC, Robillard MP (2013) Discovering essential code elements in informal documentation Notkin D, Cheng BHC, Pohl K (eds), IEEE Computer Society

Robillard MP, Nassif M, McIntosh S (2018) Threats of aggregating software repository data. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 508–518

Rodriguez-Perez G, Robles G, Gonzalez-Barahona JM (2017) How much time did it take to notify a bug?: Two case studies: Elasticsearch and nova. In: Proceedings of the 8th Workshop on Emerging Trends in Software Metrics, ser. WETSoM '17. IEEE Press, Piscataway, pp 29–35

Rodriguez-Perez G, Robles G, Gonzalez-Barahona JM (2017) How much time did it take to notify a bug? two case studies: Elasticsearch and nova. In: 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM), pp 29–35

Rodríguez-Pérez G., Zaidman A, Serebrenik A, Robles G, González-Barahona J. M. (2018) What if a bug has a different origin? making sense of bugs without an explicit bug introducing change. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ser. ESEM '18. Association for Computing Machinery, New York. https://doi.org/10.1145/3239235.3267436

Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Sci. Comput Program 74(7):470–495

Shimagaki J, Kamei Y, McIntosh S, Pursehouse D, Ubayashi N (2016) Why are commits being reverted?: A comparative study of industrial and open source projects

Śliwerski J., Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories, ser. MSR '05. ACM, New York, pp 1–5

Tsantalis N, Ketkar A, Dig D (2020) Refactoringminer 2.0 IEEE Transactions on Software Engineering

Tsantalis N, Mansouri M, M Eshkevari L, Mazinanian D, Dig D (2018) Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th International Conference on Software Engineering, ser. ICSE '18. ACM, New York, pp 483–494. https://doi.org/10.1145/3180155.3180206

Tufano M, Bavota G, Poshyvanyk D, Penta MD, Oliveto R, Lucia AD (2017) An empirical study on developer-related factors characterizing fix-inducing commits. J Softw Evol Process 29(1)

Tufano M, Pantiuchina J, Watson C, Bavota G, Poshyvanyk D (2019) On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp 25–36

Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D (2018) An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, pp 832–837

Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D (2019) An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans Softw Eng Methodol 28(4):19,1–19,29

Wang C, Li Y, Chen L, Huang W, Zhou Y, Xu B (2020) Examining the effects of developer familiarity on bug fixing. J Syst Softw 169:110667. http://www.sciencedirect.com/science/article/pii/S0164121220301266

Wen F, Nagy C, Bavota G, Lanza M (2019) A large-scale empirical study on code-comment inconsistencies. In: Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019, pp 53–64

Wen F, Nagy C, Lanza M, Bavota G (2020) An empirical study of quick remedy commits. In: ICPC '20: 28th International Conference on Program Comprehension. ACM, pp 60–71

Yan M, Xia X, Lo D, Hassan AE, Li S (2019) Characterizing and identifying reverted commits, vol 24, pp 2171–2208. https://doi.org/10.1007/s10664-019-09688-8

Yin Z, Yuan D, Zhou Y, Pasupathy S, Bairavasundaram L (2011) How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ser. ESEC/FSE '11. ACM, New York, pp 26–36

Zhao Y, Leung H, Yang Y, Zhou Y, Xu B (2017) Towards an understanding of change types in bug fixing code. Inf Softw Technol 86(C):37–53

Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, ser. ICSE '04. IEEE Computer Society, pp 563–572

Zimmermann T, Zeller A, Weissgerber P, Diehl S (2005) Mining version histories to guide software changes. IEEE Trans Softw Eng 31(6):429–445

Affiliations

Fengcai Wen[1] · Csaba Nagy[1] · Michele Lanza[1] · Gabriele Bavota[1]

Fengcai Wen
fengcai.wen@usi.ch

Csaba Nagy
csaba.nagy@usi.ch

Michele Lanza
michele.lanza@usi.ch

1 Software Institute, USI Università della Svizzera italiana, Lugano, Switzerland