An Empirical Study of Quick Remedy Commits

Fengcai Wen, Csaba Nagy, Michele Lanza, Gabriele Bavota
Software Institute, USI Università della Svizzera italiana - Lugano, Switzerland

ABSTRACT

Software systems are continuously modified to implement new features, to fix bugs, and to improve quality attributes. Most of these activities are not atomic changes, but rather the result of several related changes affecting different parts of the code. For this reason, it may happen that developers omit some of the needed changes and, as a consequence, leave a task partially unfinished, introduce technical debt or, in the worst case scenario, inject bugs. Knowing the changes that are mistakenly omitted by developers can help in designing recommender systems able to automatically identify risky situations in which, for example, the developer is likely to be pushing an incomplete change to the software repository.

We present a qualitative study investigating "*quick remedy commits*" performed by developers with the goal of implementing changes omitted in previous commits. With *quick remedy commits* we refer to commits that (i) *quickly* follow a commit performed by the same developer in the same repository, 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). Through a manual analysis of 500 quick remedy commits, we define a taxonomy categorizing the types of changes that developers tend to omit. The defined taxonomy can guide the development of tools aimed at detecting omitted changes, and possibly autocomplete them.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; *Software maintenance tools.*

KEYWORDS

Fixing Commits, Empirical Software Engineering, Mining Software Repositories

ACM Reference Format:
Fengcai Wen, Csaba Nagy, Michele Lanza, Gabriele Bavota. 2020. An Empirical Study of Quick Remedy Commits. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389266

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05.
https://doi.org/10.1145/3387904.3389266

1 INTRODUCTION

In the software life-cycle, change is the rule rather than the exception. Changes are generally performed through commit activities aimed at adding new functionalities, repairing faults, and refactoring code [49]. Some of these commits can involve a substantial part of the source code, with dozens of artifacts impacted [39]. This is often the result of what Herzig and Zeller [41] 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.* [54] 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.* [53], 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.* [54] 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.

We present a qualitative study focusing on "*quick remedy commits*" performed by developers. We define 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* to 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 of introduced errors. 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.

Fig. 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. Note that we only show in Fig. 1 part of the commit diff due to space constraints.

Figure 1: Example of quick remedy commit

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 one from the other. Second, as shown by Park *et al.* [53], it is challenging to prevent omission errors automatically; thus, 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. We present our taxonomy via qualitative examples and discuss implications for researchers and practitioners.

Structure of the paper. In Section 2 we present the design of our empirical study, and discuss its results in Section 3. In Section 4 we discuss the threats that could affect the validity of our study. In Section 5 we discuss the related literature, while in Section 6 we draw our conclusions and outline our future work.

2 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?*

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 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 [1] 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 manually filtered out repositories that do not represent real software systems (*e.g.,* JAVA-DESIGN-PATTERNS [21] and SPRING-PETCLINIC [23]), and checked for projects with shared history (*i.e.,* forked projects).

When we identified a set of forked projects, we only selected among them the one with the longest commit history (*e.g.,* both FindBugs [20] and its successor SpotBugs [22] fall under our search criteria, but we only kept the latter one). Finally, we sorted the projects in descending order based on their number of stars (*i.e.,* the most popular on top), and we selected from the list the top 1,500 projects for our study.

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	1778

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 [24].

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 the developer as a quick remedy fix for c_i. In other words, c_{i+1} must (i) have been 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, by using the author date of each commit. 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).

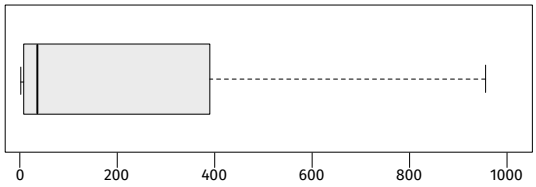

Figure 2: Time differences (in minutes) between subsequent commits (without outliers)

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 performed in rapid succession (*i.e.,* within five minutes). Then, we excluded from the selected pairs of commits all those in which c_i and c_{i+1} had not been performed by the same author, to increase the probability of c_{i+1} actually being a remedy commit for c_i rather than an unrelated change implemented by another author and just by chance being temporally close to c_i. This filtering left us with 1,041,397 candidate commits.

Finally, 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 note 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 1,041,397 commits 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 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 1,577 c_{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*.

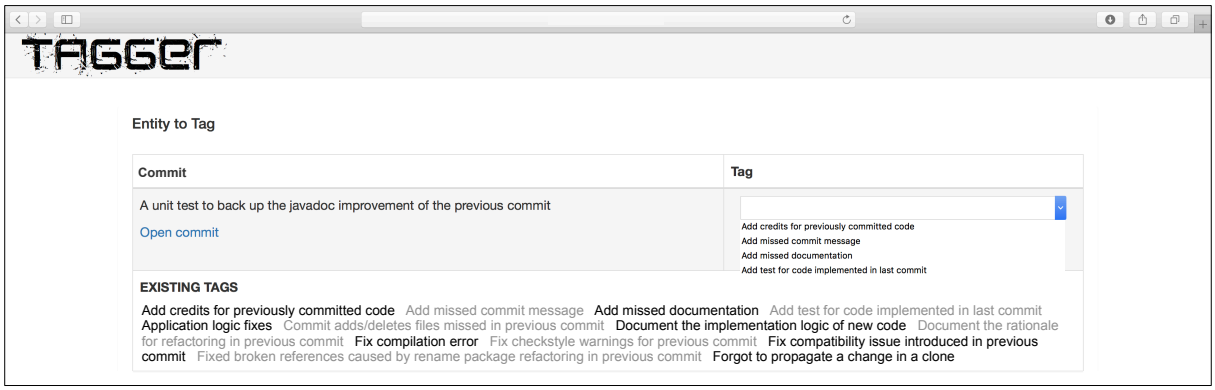

Figure 3: Web application used to run the manual tagging

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.

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 Replication Package

The data used in our study is publicly available [24]. We provide (i) the list of 1,497 subject projects; (ii) the link to the 500 commits we manually analyzed; and (iii) the classification of the manually analyzed commits.

3 RESULTS

We addressed our research question by labeling 500 commits identified as candidates to being quick remedy commits (see Section 2). 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. 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.

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

3.1 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 and causing the test suite to brake. For example, in the `Denominator` project of `Netflix`, a developer reported in the commit message: "*Fix tests broken by former commits*" [4].

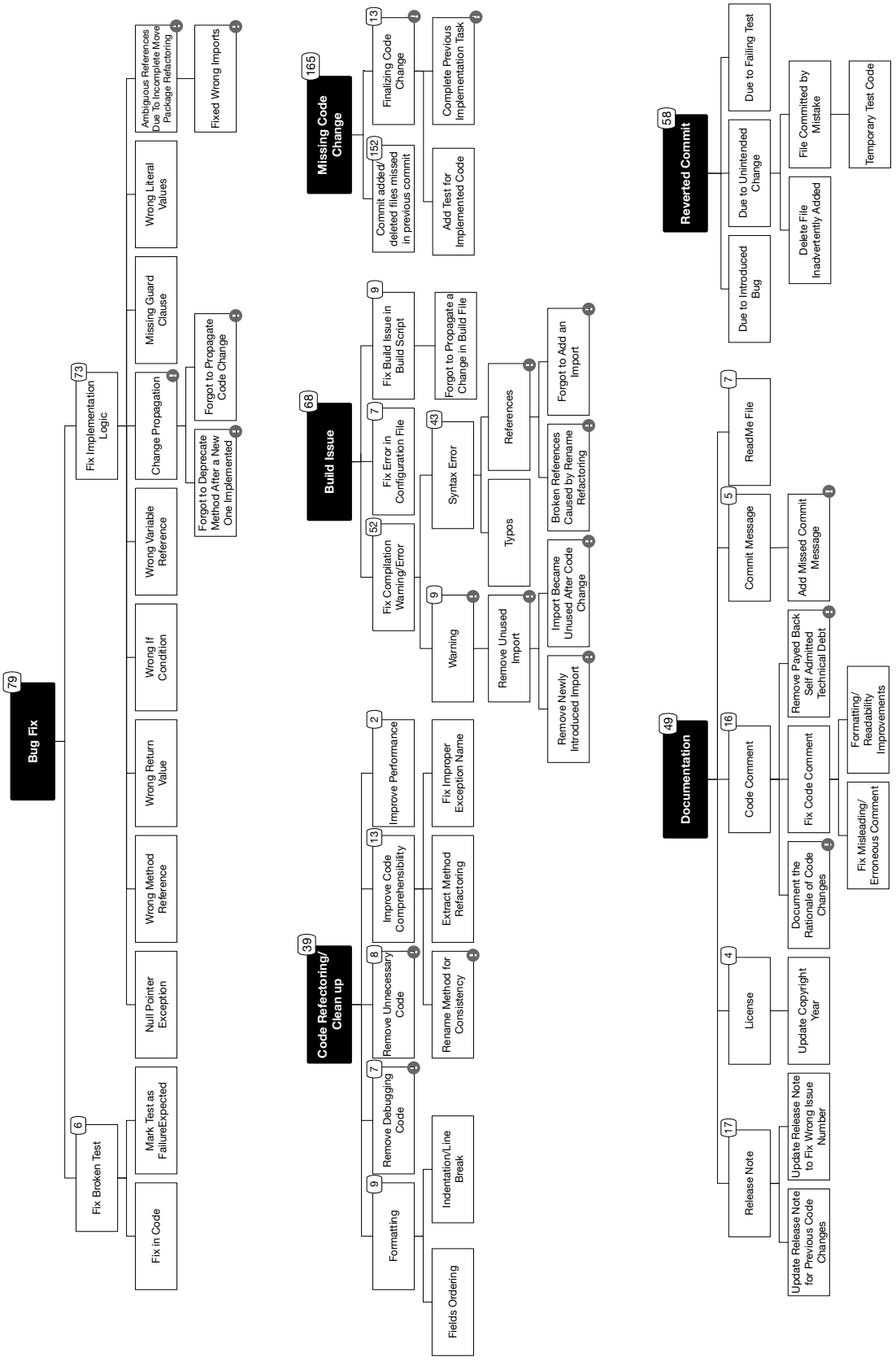

Figure 4: Taxonomy of Quick Remedy Commits

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 [65]. 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 [65]. 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 [46]. An example of this can be seen in the *mathttTomP2P* project. In a commit [15], 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 [16], they fix a conditional statement to check the status of a `Put` object in a new branch. Then, only a few seconds later [17], 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 [11], they update a SQL script by adding a query for the removal of unnecessary data. Then, in the quick subsequent commit [12], 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.* [70]) 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 [2], but still keep the old ones.

In a follow-up commit [3], they realize that they use, however, the wrong references to the migrated classes. ⚗ Code clone detection techniques [60] 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 [2].

3.2 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` [9] 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 `liferay – portal` project [8], developers opened an issue to "*introduce tests to document current behavior*" [19]. 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 [8], 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.* [25] with their NATURALIZE tool able to recommend meaningful identifier names and formatting guidelines. Other approaches focus only on rename refactoring suggestions [47]. 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.

3.3 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 [18] 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 [36]. 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.

3.4 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 [41] 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.* [32] 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.* [43]).

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 [10], 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 interesting case from the `geoserver` project [5], 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 able to compare what described in the commit message and what has been actually implemented in the code change. For example, in the previously discussed example [10], a misalignment between the number of sub-features actually implemented and claimed in the commit message could be spotted and reported to the developer.

3.5 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.

It is also worth mentioning that 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.* [61] 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.* [67] 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.* [61], 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.,* [27, 30, 33] — 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.,* [58, 63]. Empirical studies aimed at assessing the impact of considering (or not) reverted commits on the findings of MSR studies could shed some light on the bias (if any) these commits introduce.

3.6 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 fix of commit messages rarely happens, 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 ⟨*code changes implemented in a commit* c_x, *commit message of* c_x⟩ to train models able to learn how to generate commit notes [43], 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 [6] performed in the `jitsi` project. In a commit [7] they fix a bug due to 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 [6] 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 [55], 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 [66], that could possibly confuse developers comprehending the associated code components. One representative example of this scenario is the commit [13] 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 [14] 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.

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 the manual analysis we performed to identify the reasons behind the quick remedy changes performed by developers. To mitigate subjectivity bias in such a process, 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.

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.* [44] 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. Finally, due to the qualitative nature of our study, 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 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.

5 RELATED WORK

There is a vast literature of empirical studies investigating developers' commits for various purposes. Many studies tackle who/what/when/where/why developers change their source code. However, there has been little research on quick fixes, or consecutive changes performed by software developers. Here we present an overview of the related work close to the topic of our paper.

5.1 Reasons for Changes

Mockus *et al.* [49] 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*). Besides, 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 [39] 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 (forward engineering) or maintenance (reengineering, corrective engineering, and management) categories.

Hindle *et al.* [42] conducted a study on large commits and created a taxonomy of the purpose of large commits. They also 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

5.2.1 Small Changes. Purushothaman and Perry [56] 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.

5.2.2 Large Changes. Sliwerski *et al.* [62] 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.

5.2.3 Social Characteristics. Eyolfson *et al.* [37] 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.* [34] 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.* [31] 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 *et al.* [57] 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.

Gonzales-Barahona *et al.* [38] 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.

5.2.4 Supplementary Patches. Park *et al.* [54] 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 [52, 53] 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.* [26] 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.

5.2.5 Tangled Changes. Herzig *et al.* [41] 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 multiple tangled changes. Later, they also showed that tangled changes could significantly impact the accuracy of defect prediction models as assessed in empirical studies [40].

5.2.6 Consecutive Changes. Dai *et al.* [35] 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 and strong impact on the later file versions in the short term, especially when the length of the change chain is four or five.

5.2.7 Inconsistent Changes. Bettenburg *et al.* [29] 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.

5.2.8 Incorrect Changes. Yin *et al.* [68] 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.

5.2.9 Changes and Refactoring. Palomba *et al.* [50] 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.* [28] 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.* [59] 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 *et al.* [45] 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.* [51] 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.* [69] 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 [48] 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 [64] or to learn how to automatically fix bugs [65].

5.5 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 shedding some light on the phenomenon of *quick remedy commits*. Indeed, while previous studies looked at supplementary bug-fixes [52–54], their focus was limited to bug-fixing activities, while we looked at remedy commits from a broader perspective. For this reason, our work complements previous findings reported in the literature.

6 CONCLUSION

We presented a qualitative empirical study aimed at investigating *quick remedy commits* performed by developers in GitHub projects. We defined *quick remedy commits* as commits performed by developers to remedy to changes omitted or errors introduced in a previous commit, performed just a few minutes before.

Our study is based on the manual analysis of 500 commits, that we classified by looking at the objective of the remedy commit. The output of our 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.

Our future work will target two directions. First, we plan to enlarge the set of commits manually analyzed to test the generalizability and completeness of the defined taxonomy. Second, we will work on some of the research directions discussed in our results section, 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.,* [65]) 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 [25]. Once learned, the coding conventions can be automatically checked on the code to commit, raising a warning in case violations are detected.

On the impact of reverted commits on MSR studies. Studying the impact that reverted commits have on the findings of MSR studies using the change history of software systems as basic information to compute a variety of proxies (*e.g.,* developers' experience, change-proneness of code) is also part of our research agenda.

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).

ACKNOWLEDGMENT

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects PROBE (SNF Project No. 172799) and CCQR (SNF Project No. 175513), and CHOOSE for sponsoring our trip to the conference.

REFERENCES

[1] [n. d.]. About stars (GitHub). https://help.github.com/articles/about-stars/.
[2] [n. d.]. Commit to Accumulo project on GitHub. https://github.com/apache/accumulo/commit/2ad672a
[3] [n. d.]. Commit to accumulo project on GitHub. https://github.com/apache/accumulo/commit/b8859513a
[4] [n. d.]. Commit to denominator project on GitHub. https://github.com/Netflix/denominator/commit/e727b9d
[5] [n. d.]. Commit to geoserver project on GitHub. https://github.com/geoserver/geoserver/commit/22c89ad106
[6] [n. d.]. Commit to jitsi project on GitHub. https://github.com/jitsi/jitsi/commit/6a361bbf6
[7] [n. d.]. Commit to jitsi project on GitHub. https://github.com/jitsi/jitsi/commit/a74af45
[8] [n. d.]. Commit to liferay-portal project on GitHub. https://github.com/liferay/liferay-portal/commit/1b5c378d4785
[9] [n. d.]. Commit to lombok project on GitHub. https://github.com/rzwitserloot/lombok/commit/57f59074
[10] [n. d.]. Commit to openpnp project on GitHub. https://github.com/openpnp/openpnp/commit/aeef4cb0e4
[11] [n. d.]. Commit to spacewalk project on GitHub. https://github.com/spacewalkproject/spacewalk/commit/6df7327
[12] [n. d.]. Commit to spacewalk project on GitHub. https://github.com/spacewalkproject/spacewalk/commit/fec7040
[13] [n. d.]. Commit to tinkerpop project on GitHub. https://github.com/apache/tinkerpop/commit/a4c62be7a5
[14] [n. d.]. Commit to tinkerpop project on GitHub. https://github.com/apache/tinkerpop/commit/aa3d538
[15] [n. d.]. Commit to TomP2P project on GitHub. https://github.com/tomp2p/TomP2P/commit/3db803c
[16] [n. d.]. Commit to TomP2P project on GitHub. https://github.com/tomp2p/TomP2P/commit/8802c5e
[17] [n. d.]. Commit to TomP2P project on GitHub. https://github.com/tomp2p/TomP2P/commit/4bc6e824
[18] [n. d.]. Commit to Tower project on GitHub. https://github.com/DroidPlanner/Tower/commit/72132d049
[19] [n. d.]. Liferay Portal Issue LPS-44476. https://issues.liferay.com/browse/LPS-44476
[20] [n. d.]. Project FindBugs on GitHub. https://github.com/findbugsproject/findbugs
[21] [n. d.]. Project java-design-patterns on GitHub. https://github.com/iluwatar/java-design-patterns
[22] [n. d.]. Project SpotBugs on GitHub. https://github.com/spotbugs/spotbugs
[23] [n. d.]. Project spring-petclinic on GitHub. https://github.com/spring-projects/spring-petclinic
[24] [n. d.]. Replication Package. https://github.com/Em11FW/ICPC2020-quick-remedy-commit.
[25] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 281–293.
[26] Le An, Foutse Khomh, and Bram Adams. 2014. Supplementary Bug Fixes vs. Re-opened Bugs. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM '14)*. IEEE Computer Society, Washington, DC, USA, 205–214. https://doi.org/10.1109/SCAM.2014.29
[27] Mauricio Finavaro Aniche, Gabriele Bavota, Christoph Treude, Marco Aurélio Gerosa, and Arie van Deursen. 2018. Code smells for Model-View-Controller architectures. *Empirical Software Engineering* 23, 4 (2018), 2121–2157.
[28] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 104–113.
[29] Nicolas Bettenburg, Weiyi Shang, Walid M. Ibrahim, Bram Adams, Ying Zou, and Ahmed E. Hassan. 2012. An Empirical Study on Inconsistent Changes to Code Clones at the Release Level. *Sci. Comput. Program.* 77, 6 (June 2012), 760–776. https://doi.org/10.1016/j.scico.2010.11.010
[30] J. M. Bieman, A. A. Andrews, and H. J. Yang. 2003. Understanding change-proneness in OO software through visualization. In *11th IEEE International Workshop on Program Comprehension, 2003*. 44–53.
[31] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 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 (ESEC/FSE '11)*. ACM, New York, NY, USA, 4–14. https://doi.org/10.1145/2025113.2025119
[32] Nick C. Bradley, Thomas Fritz, and Reid Holmes. 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*. 993–1003.
[33] Gemma Catolino and Filomena Ferrucci. 2019. An extensive evaluation of ensemble techniques for software change prediction. *Journal of Software: Evolution and Process* 31, 9 (2019).
[34] Maëlick Claes, Mika V. Mäntylä, Miikka Kuutila, and Bram Adams. 2018. Do Programmers Work at Night or During the Weekend?. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 705–715. https://doi.org/10.1145/3180155.3180193
[35] Meixi Dai, Beijun Shen, Tao Zhang, and Min Zhao. 2014. Impact of Consecutive Changes on Later File Versions. In *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies (EAST 2014)*. ACM, New York, NY, USA, 17–24. https://doi.org/10.1145/2627508.2627512
[36] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 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 (ESEC-FSE '07)*. 185–194.
[37] Jon Eyolfson, Lin Tan, and Patrick Lam. 2011. Do Time of Day and Developer Experience Affect Commit Bugginess?. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, New York, NY, USA, 153–162. https://doi.org/10.1145/1985441.1985464
[38] Jesus M.. Gonzalez-Barahona, Daniel Izquierdo-Cortazar, and Andrea Capiluppi. 2011. Are Developers Fixing Their Own Bugs?: Tracing Bug-Fixing and Bug-Seeding Committers. *Int. J. Open Source Softw. Process.* 3, 2 (April 2011), 23–42. https://doi.org/10.4018/jossp.2011040102
[39] Lile P. Hattori and Michele Lanza. 2008. On the Nature of Commits. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. IEEE Press, Piscataway, NJ, USA, III–63–III–71. https://doi.org/10.1109/ASEW.2008.4686322
[40] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The Impact of Tangled Code Changes on Defect Prediction Models. *Empirical Softw. Engg.* 21, 2 (April 2016), 303–336. https://doi.org/10.1007/s10664-015-9376-6
[41] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 121–130.
[42] Abram Hindle, Daniel M. German, and Ric Holt. 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 (MSR '08)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/1370750.1370773
[43] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 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*. 135–146.
[44] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proc. of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. 92–101.
[45] Sunghun Kim and E. James Whitehead, Jr. 2006. How Long Did It Take to Fix Bugs?. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. ACM, New York, NY, USA, 173–174. https://doi.org/10.1145/1137983.1138027
[46] J. Krinke. 2007. A Study of Consistent and Inconsistent Changes to Code Clones. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. 170–178.
[47] Bin Lin, Simone Scalabrino, Andrea Mocci, Rocco Oliveto, Gabriele Bavota, and Michele Lanza. 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, 2017*. 81–90.
[48] Matias Martinez and Martin Monperrus. 2019. Coming: A Tool for Mining Change Pattern Instances from Git Commits. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 79–82. https://doi.org/10.1109/ICSE-Companion.2019.00043
[49] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00) (ICSM '00)*. IEEE Computer Society, Washington, DC, USA, 120–.
[50] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An Exploratory Study on the Relationship Between Changes and Refactoring. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, 176–185. https://doi.org/10.1109/ICPC.2017.38
[51] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. 2009. Toward an Understanding of Bug Fix Patterns. *Empirical Softw. Engg.* 14, 3 (June 2009), 286–315. https://doi.org/10.1007/s10664-008-9077-5
[52] Jihun Park, Miryung Kim, and Doo-Hwan Bae. 2014. An Empirical Study on Reducing Omission Errors in Practice. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 121–126. https://doi.org/10.1145/2642937.2642956
[53] Jihun Park, Miryung Kim, and Doo-Hwan Bae. 2017. An Empirical Study of Supplementary Patches in Open Source Projects. *Empirical Softw. Engg.* 22, 1

(Feb. 2017), 436–473. https://doi.org/10.1007/s10664-016-9432-x

[54] J. Park, M. Kim, B. Ray, and D. Bae. 2012. An empirical study of supplementary bug fixes. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 40–49. https://doi.org/10.1109/MSR.2012.6224298

[55] A. Potdar and E. Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 91–100.

[56] Ranjith Purushothaman and Dewayne E. Perry. 2005. Toward Understanding the Rhetoric of Small Source Code Changes. *IEEE Trans. Softw. Eng.* 31, 6 (June 2005), 511–526. https://doi.org/10.1109/TSE.2005.74

[57] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, Experience and Defects: A Fine-grained Study of Authorship. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 491–500. https://doi.org/10.1145/1985793.1985860

[58] M. M. Rahman, C. K. Roy, and R. G. Kula. 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)*. 215–226.

[59] Gema Rodriguez-Perez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. 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 (WETSoM '17)*. IEEE Press, Piscataway, NJ, USA, 29–35. https://doi.org/10.1109/WETSoM.2017..6

[60] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.* 74, 7 (May 2009), 470–495.

[61] J. Shimagaki, Y. Kamei, S. McIntosh, D. Pursehouse, and N. Ubayashi. 2016. Why are Commits Being Reverted?: A Comparative Study of Industrial and Open Source Projects. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–311. https://doi.org/10.1109/ICSME.2016.83

[62] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083147

[63] Michele Tufano, Gabriele Bavota, Denys Poshyvanyk, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2017. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process* 29, 1 (2017).

[64] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 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*. 25–36.

[65] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 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*. 832–837.

[66] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 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*. 53–64.

[67] Meng Yan, Xin Xia, David Lo, Ahmed E. Hassan, and Shanping Li. 2019. Characterizing and identifying reverted commits. *Empirical Software Engineering* 24, 4 (01 Aug 2019), 2171–2208. https://doi.org/10.1007/s10664-019-09688-8

[68] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 26–36. https://doi.org/10.1145/2025113.2025121

[69] Yangyang Zhao, Hareton Leung, Yibiao Yang, Yuming Zhou, and Baowen Xu. 2017. Towards an Understanding of Change Types in Bug Fixing Code. *Inf. Softw. Technol.* 86, C (June 2017), 37–53. https://doi.org/10.1016/j.infsof.2017.02.003

[70] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.