Towards a Classification of Bugs Based on the Location of the Corrections: An Empirical Study

Mathieu Nayrolles*, Abdelwahab Hamou-Lhadj*, Emad Shihab†, Alf Larsson‡, Sigrid Eldh‡

*Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University, Montréal, Canada
{mathieu.nayrolles, wahab.hamou-lhadj}@ece.concordia.ca

† Data-driven Analysis of Software (DAS) Research Lab
CS, Concordia University, Montréal, Canada
.shihab@concordia.ca

‡Ericsson
Stockholm, Sweden
{alf.larsson, sigrid.eldh}@ericsson.com

Abstract—The abstract goes here.

Index Terms—Bug taxonomy, Bug tracking systems, Empirical studies, Software maintenance

1 Introduction

There have been several studies (e.g., [1], [2]) that study of the factors that influence the bug fixing time. These studies empirically investigate the relationship between bug report attributes (description, severity, etc.) and the fixing time. Other studies take bug analysis to another level by investigating techniques and tools for bug prediction and reproduction (e.g., [3], [4], [5]). These studies, however, treat all bugs as being the same. For example, a bug that requires only one fix is analysed the same way as a bug that necessitates multiple fixes. Similarly, if multiple bugs are fixed by modifying the exact same locations in the code, then we should investigate how these bugs are related in order to predict them in the future. Note here that we do not refer to duplicate bugs. Duplicate bugs are marked as duplicate (and not fixed) and only the master bug is fixed. As a motivating example, consider Bugs #AMQ-5066 and #AMQ-5092 from the Apache Software Foundation bug report management system (used to build one of the datasets in this paper). Bug #AMQ-5066 was reported on February 19, 2014 and solved with a patch provided by the reporter. The solution involves a relatively complex patch that modifies `MQTTProtocolConverter.java`, `MQTTSubscription.java` and `MQTTTest.java` files. The description of the bug is as follows:

When a client sends a SUBSCRIBE message with the same Topic/Filter as a previous SUBSCRIBE message but a different QoS, the Server MUST discard the older subscription, and resend all retained messages limited to the new Subscription QoS.

A few months later, another bug, Bug #AMQ-5092 was reported:

MQTT protocol converters does not correctly generate unique packet ids for retained and non-retained publish messages sent to clients. [...] Although retained messages published on creation of client subscriptions are copies of retained messages, they must carry a unique packet id when dispatched to clients. ActiveMQ re-uses the retained message's packet id, which makes it difficult to acknowledge these messages when wildcard *topics are used. ActiveMQ also sends the same non-retained message multiple times for every matching subscription for overlapping subscriptions. These messages also re-use the publisher's message id as the packet id, which breaks client acknowledgment.*

This bug was assigned and fixed by a different person than the one who fixed bug #AMQ-5066. The fix consists of modifying slightly the same lines of the code in the exact files used to fix Bug #AMQ-5066. In fact, Bug #5092 could have been avoided altogether if the first developer provided a more comprehensive fix to #AMQ-5066 (a task that is easier said than done). These two bugs are not duplicates since, according to their description, they deal with different types of problems and yet they can be fixed by providing a similar patch. In other words, the failures are different while the root causes (faults in the code) are more or less the same. From the bug handling perspective, if we can develop a way to detect such related bug reports during triaging then we can achieve considerable time saving in the way bug reports are processed, for example, by assigning them to the same developers. We also conjecture that detecting such related bugs can help with other tasks such as bug reproduction. We can reuse the reproduction of an already fixed bug to reproduce an incoming and related bug.

The objective of this paper is not to propose a way to detect such related bug reports or how we can take advantage of them to improve the bug handling process, but it is to introduce a new way of grouping bugs into types that we believe can facilitate the bug handling process. We are interested in bugs that share similar fixes. By a fix, we mean a modification (adding or deleting lines of code) to an exiting file that is used to solve the bug. With this in mind, the relationship between bugs and fixes can be modeled using the UML diagram in Figure 1. The diagram only includes bugs that are fixed. From this figure, we can think of four instances of this diagram, which we refer to as bug taxonomy or simply bug types (see Figure 2).

Figure 1. Class diagram showing the relationship between bugs and fixed

The first and second types are the ones we intuitively know about. Type 1 refers to a bug being fixed in one single location (i.e., one file), while Type 2 refers to bugs being fixed in more than one location. In Figure 2, only two locations are shown for the sake of clarity, but many more locations could be involved in the fix of a bug. Type 3 refers to multiple bugs that are fixed in the exact same location. Type 4 is an extension of Type 3, where multiple bugs are resolved by modifying the same set of locations. Note that Type 3 and Type 4 bugs are not duplicates, they may occur when different features of the system fail due to the same root causes (faults). We conjecture that knowing the proportions of each type of bugs in a system may provide insight into the quality of the system. Knowing, for example, that in a given system the proportion of Type 2 and 4 bugs is high may be an indication of poor system quality since many fixes are needed to address these bugs. In addition, the existence of a high number of Types 3 and 4 bugs calls for techniques that can effectively find bug reports related to an incoming bug during triaging. This is similar to the many studies that exist on detection of duplicates (e.g., [6], [7], [8]), except that we are not looking for duplicates but for related bugs (bugs that are due to failures of different features of the system, caused by the same faults). To our knowledge, there is no study that empirically examines bug data with these types in mind, which is the main objective of this section. By analogy, we can look at the proposed bug taxonomy in a similar way as the clone detection taxonomy presented by Kapser and Godfrey [9]. The authors proposed seven types of source code clones and then conducted a case study, using their classification, on the file system module of the Linux operating system. This clone taxonomy continues to be used by researchers to build better approaches for detecting a given clone type and being able to effectively compare approaches with each other.

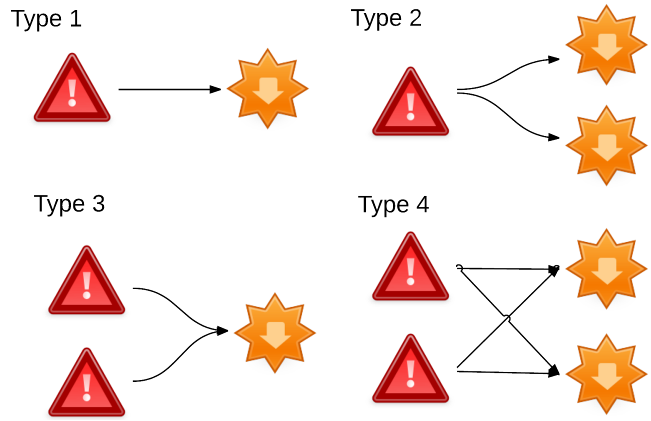


Figure 2. Proposed Taxonomy of Bugs

In this paper, we are interested in addressing the following research questions:

- RQ1: What are the proportions of different types of bugs?
- RQ2: How complex is each type of bugs?
- RQ3: How pertinent is a bug taxonomy? [WAHAB: I think you should remove this and keep it for a

journal. i suggest to send this paper, without RQ3 to MSR]

2 Study Design

The goal of this study is to analyse the location of bug fixes, with the purpose of classifying bug fixes into types. More specifically, this study aims to answer the following two research questions:

- **RQ$_1$:** *What are the proportions of different types of bugs?* This research question aims to identify the extent to which bugs can be classified according to their fix-locations and the proportion of each types. Specifically, we investigate if different types of bugs exist at all and if the proportion of different types in non-negligible. As discussed earlier [WAHAB: Please run a spell checker], knowing, for example, that bugs of Type 3 and 4 are the most predominant ones suggests that we need to investigate techniques to help detect whether an incoming bug is of Types 3 and 4 by examining historical data. Similarly, if we can automatically identify a bug that is related to another one that has been fixed then we can reuse the results of reproducing the first bug in reproducing the second one.
- **RQ$_2$:** *How complex is each type of bugs?* This second research question aims to investigate the complexity of the different types of bug. More specifically, we analyse the complexity of different types of bugs using code and process metrics both. For the code aspect of the complexity, we compute the number of different files impacted by the fix and the number of hunks and churns. We do not compute any statical complexity metrics such as cyclomatic complexity [10]. For the process aspect of complexity, we analyse the severity of a bug, the number of duplicate bug reports submitted, the number of times a bug report is reopened, the number of commits [WAHAB: Do you mean commits?], and the time required to fix the bug.

2.1 Version control systems

Version control consists of maintaining the versions of files — such as source code and other software artifacts [11]. This activity is a complex task and cannot be performed manually on real world project. To this end, numerous tools have been created to help practitioners manage the version of their software artifacts. Each evolution of a software is a version (or revision) and each version (revision) is linked to the one before through modifications of software artifacts. These modifications consist of updating, adding, or deleting software artifacts. They can be referred as `diff`, `patch` or `commit`[1]. Each `diff`, `patch` or `commit` have the following characteristics:

- Number of Files: The number of software files that have been modified, added, or deleted.
- Number of Hunks: The number of consecutive code blocks of modified, added or deleted lines in textual files. Hunks are used to determine, in each file, how many different places the developer has modified.
- Number of Churns: The number of lines modified. However, the churn value for a line change should be at least two as the line has to be deleted first and then added back with the modifications.

Figure 3 presents the data structure used to store a commit. Each commit is represented as a tree. The root leaf (green) contains the commit, tree and parent hashes as well as the author and the description associated with the commit. The second leaf (blue) contains the leaf hash and the hashes of the files of the project.

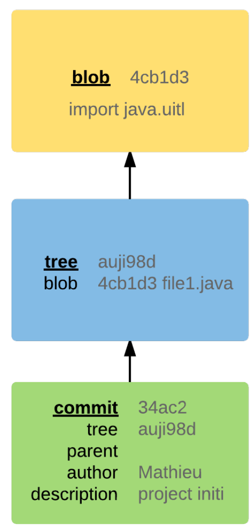


Figure 3. Data structure of a commit.

In this example, we can see that author "Mathieu" has created the file $file1.java$ with the message "project init".

2.2 Project Tracking Systems

Project tracking systems allow end users to create bug reports (BRs) to report unexpected system behavior, manager can create tasks to drive the evolution forward and crash report (CRs) can be automatically created. These systems are also used by development teams to keep track

1. These names are not to be used interchangeably as difference exists.

of the modification induced by bug and to crash reports, and keep track of the fixes.

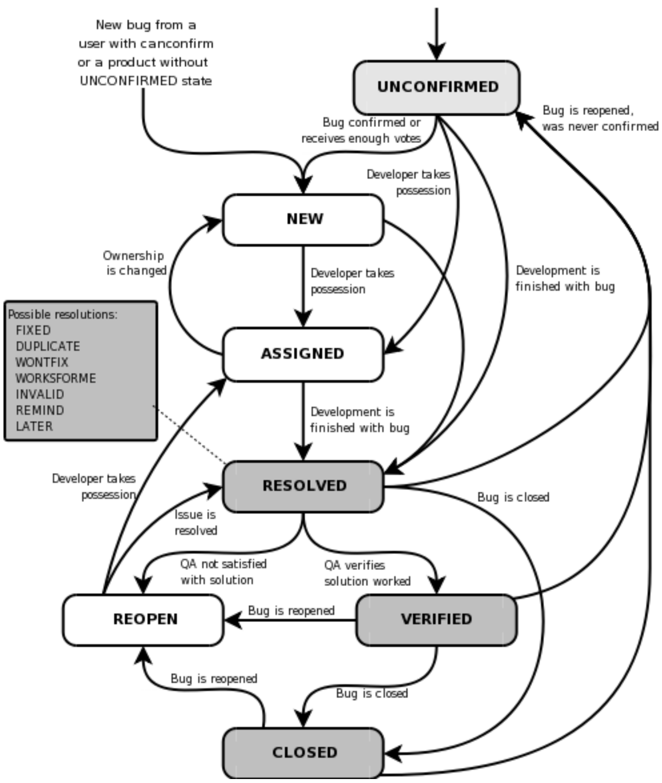


Figure 4. Lifecyle of a report [12]

Figure 4 presents the life cycle of a report. When a report is submitted by an end-user, it is set to the `UNCONFIRMED` state until it receives enough votes or that a user with the proper permissions modifies its status to `NEW`. The report is then assigned to a developer to be fixed. When the report is in the `ASSIGNED` state, the assigned developer(s) starts working on the report. A fixed report moves to the `RESOLVED` state. Developers have five different possibilities to resolve a report: `FIXED`, `DUPLICATE`, `WONTFIX`, `WORKSFORME` and `INVALID` [13].

- `RESOLVED/FIXED`: A modification to the source code has been pushed, i.e., a changeset (also called a patch) has been committed to the source code management system and fixes the root problem described in the report.
- `RESOLVED/DUPLICATE`: A previously submitted report is being processed. The report is marked as duplicate of the original report.
- `RESOLVED/WONTFIX`: This is applied in the case where developers decide that a given report will not be fixed.
- `RESOLVED/WORKSFORME`: If the root problem described in the report cannot be reproduced on the reported OS / hardware.
- `RESOLVED/INVALID`: If the report is not related to the software itself.

Finally, the report is `CLOSED` after it is resolved. A report can be reopened (sent to the `REOPENED` state) and then assigned again if the initial fix was not adequate (the fix did not resolve the problem). The elapsed time between the report marked as the new one and the resolved status are known as the *fixing time*, usually in days. In case of task branching, the branch associated with the report is marked as ready to be merged. Then, the person in charge (quality assurance team, manager, ect...) will be able to merge the branch with the mainline. If the report is reopened: the days between the time the report is reopened and the time it is marked again as `RESOLVED/FIXED` are cumulated. Reports can be reopened many times.

Tasks follow a similar life cycle with the exception of the `UNCONFIRMED` and `RESOLVED` states. Tasks are created by management and do not need to be confirmed in order to be `OPEN` and `ASSIGNED` to developers. When a task is complete, it will not go to the `RESOLVED` state, but to the `IMPLEMENTED` state. Bug and crash reports are considered as problems to eradicate in the program. Tasks are considered as new features or amelioration to include in the program.

Reports and tasks can have a severity [14]. The severity is a classification to indicate the degree of impact on the software. The possible severities are:

- blocker: blocks development and/or testing work.
- critical: crashes, loss of data, severe memory leak.
- major: major loss of function.
- normal: regular report, some loss of functionality under specific circumstances.
- minor: minor loss of function, or other problem where easy workaround is present.
- trivial: cosmetic problems like misspelled words or misaligned text.

The relationship between a report or a task and the actual modification can be hard to establish and has been a subject of various research studies (e.g., [15], [16], [17]). This reason is that they are in two different systems: the version control system and the project tracking system. While it is considered a good practice to link each report with the versioning system by indicating the report $\#id$ on the modification message, more than half of the reports are not linked to a modification [17].

2.3 Context Selection

The context of this study consists of the change history of 388 projects belonging to two software ecosystems, namely, Apache and Netbeans. Table 1 reports, for each of them, (i) the number of projects analyzed, (ii) size ranges in terms of the number of classes and KLOC, (iii) the overall number of commits and issues analyzed, and (iv) the average, minimum, and maximum length of the projects' story (in years).

Dataset	R/F BR	CS	Files	Projects
Netbeans	53,258	122,632	30,595	39
Apache	49,449	106,366	38,111	349
Total	102,707	229,153	68,809	388

Table 1
Datasets

All the analyzed projects are hosted in *Git* or *Mercurial* repositories and have either a *Jira* or a *Bugzilla* issue tracker associated with it. The Apache ecosystem consists of 349 projects written in various programming languages (C, C++, Java, Python, ...) and uses *Git* and *Jira*. These projects represent the Apache ecosystem in its entirety; no system has been excluded from our study. The complete list can be found online[2]. The Netbeans ecosystem consists in 39 projects, mostly written in Java. Similarly to the Apache ecosystem, we did not select any of the projects belonging to the Netbeans ecosystem but all of them. The Netbeans community uses *Bugzilla* and *Mercurial*.

The choice of the ecosystems to analyze is not random, but rather driven by the motivation to consider projects having (i) different sizes, (ii) different architectures, and (iii) different development bases and processes. Indeed, Apache projects are extremely various in terms of size of the development team, purpose and technical choices [18]. On the other side, Netbeans has a relatively stable list of core developer and a common vision shared through the 39 related projects [19].

Cumulatively, these datasets span from 2001 to 2014. In summary, our consolidated dataset contains 102,707 bugs, 229,153 changesets, 68,809 files that have been modified to fix the bugs, 462,848 comments, and 388 distinct systems. We also collected 221 million lines of code modified to fix the bugs, identified 3,284 sub-projects, and 17,984 unique contributors to these bug report and source code version management systems. The cumulated opening time for all the bugs reaches 10,661 working years (3,891,618 working days).

2.4 Data Extraction and Analysis

This subsection describes the data extraction and analysis process that we followed to answer our research questions.

2. https://projects.apache.org/projects.html?name

2.4.1 What are the proportions of different types of bugs?

To answer $\mathbf{RQ}_1$, we cloned the 349 *git* repositories belonging to the Apache ecosystem and the 39 *mercurial* repositories belonging to the Netbeans ecosystem. The raw size of the cloned source code alone, excluding binaries, images and other non-text file, is 163 GB. Then, we extracted all the 102,707 closed issues that have been resolved using the *RESOLVED/FIXED* tags. Indeed, this study aims to classify bugs according to their fix locations. If an issue is fixed by other means than *fixing* the source code, then, it falls outside the scope our study. In order to assign commits to issues we used is the regular expression-based approach by Fischer et al. [20] matching the issue ID in the commit note. Using this technique, we were able to link almost 40% (40,493 out of 102,707) of our resolved/fixed issues to 229,153 commits. An issue can be fixed with several commits.

We choose not to use more complex technique like ReLink, an approach proposed by Wu et al. [17], which considers the following constraints: (i) matching the committer/authors with issue tracking contributor name/e-mail; (ii) the time interval between the commit and the last comment posted by the same author/contributor on the issue tracker must be less than seven days; and (iii) Vector Space Model (VSM) cosine similarity between the commit note and the last comment referred above or greater than 0.7 because we belive that mining more than forty thousands issues is enough to be significant.

Using our generated consolidated dataset, we extracted the files f_i impacted by each commit c_i for each one of our 388 projects. Then, we classify the bugs according to the following:

- **Type 1:** A bug is tagged Type 1 if it is fixed by modifying a file f_i and f_i is not involved in any other bug fix.
- **Type 2:** A bug is tagged Type 2 if it is fixed by modifying several files $f_{i..n}$ and the files $f_{i..n}$ are not involved in any other bug fix.
- **Type 3:** A bug is tagged Type 3 if it is fixed by modifying a file f_i and the file f_i is involved in fixing other bugs.
- **Type 4:** A bug is tagged Type 4 if it is fixed by modifying several files $f_{i..n}$ and the files $f_{i..n}$ are involved in any other bug fix.

To answer this question, we analyze whether any type is predominant in the studied ecosystem, by testing the null hypothesis:

- H_{01} : The proportion of types does not change significantly across the studied ecosystems.

We test this hypothesis by observing both a "global" (across ecosystem) and a "local" predominance (per ecosystem) of the different types of bugs. We must observe these two aspects to ensure that the predominance of a particular type of bug is not circumstantial (in few given systems only) but is also not due to some other, unknown factors (in all systems but not in a particular ecosystem).

We answer $\mathbf{RQ}_1$ in two steps. The first step is to use descriptive statistics. We compute the ratio of each types to the total number of bugs in the dataset.

In the second step, we compare the proportions of the different types of bugs with respect to the ecosystem where the bugs were found. We build the contingency table with these two qualitative variables (the type and studied ecosystem) and test the null hypothesis $\mathbf{H}_{01A}$ to assess whether the proportion of a particular type of bugs is related to a specific ecosystem or not.

We use the Pearson's chi-squared test to reject the null hypothesis H_{01A}. Pearson's chi-squared independence test is used to analyze the relationship between two qualitative data, in our study the type bugs and the studied ecosystem. The results of Pearson's chi-squared independence tests are considered statistically significant at α = 0.05. If p-value $\leq$ 0.05, we reject the null hypothesis H_{01A} and conclude that the proportion of each types is different for each ecosystem.

Overall, the data extraction and manipulation for $\mathbf{RQ}_1$ (i.e., cloning repositories, linking commits to issues and tagging issues by type) took thirteen weeks on two Linux servers having 1 quadcore 3.10 GHz CPU and 12 GB of RAM each.

2.4.2 *How complex is each type of bugs?*

To answer $\mathbf{RQ}_2$ we went through the 40,493 resolved/-fixed issues and the linked 229,153 commits in order to compute code and process metrics for each of them. These metrics will then be used to assess the complexity of a bug. The computed process metrics are:

- The time t it took to resolve issue i.
- The number of issues dup tagged as duplicate of issue i.
- The number of time issue i got reopen $reop$.
- The number of comments $comment$ on issue i.
- The severity sev of the issue i.

The computed code metrics are:

- The number of files f impacted by issue i.
- The number of commit c required to fix the issue i.
- The number of hunks h required to fix the issue i.
- The number of churns ch required to fix the issue i.

We address the relation between types and the complexity of the bugs in using our metrics. We analyze whether Types 2 and 4 bugs are more complex to handle than Types 1 and 3 bugs, by testing the null hypotheses:

- H_{02}: The complexity of bug types is not significantly different from type to type.

To test our hypothesis, we build a contingency table with the qualitative variables and the dependent variable for each type.

We use the Pearson's chi-squared test to reject the null hypothesis H_{02}. The results of Pearson's chi-squared independence tests are considered statistically significant at α = 0.05. If a p-value $\leq$ 0.05, we reject the null hypothesis H_{02} and conclude that the complexity of bug is related to its type.

3 Analysis of the Results

This section reports the analysis of the results aiming at answering our two research questions.

3.1 What are the proportions of different types of bugs?

Table 2 presents a contingency table and the results of the Pearson's chi-squared tests we performed on each types of bug. In addition to presenting bug types 1 to 4, Table 2 also presents grouping of bug types: Types 1 and 2 versus Types 3 and 4.

Types 3 (22.6% and 54%) and 4 (31.3% and 64.9%) are predominant compared to types 1 (14.3% and 9.1%) and 2 (6.8% and 3.7%) for the Apache and the Netbeans ecosystems, respectively. Overall, the proportion of different types of bug is as follows: 6.8%, 3.7%, 28.3%, 61.2% for types 1, 2, 3 and 4, respectively. The result of the Pearson's test is below 0.01. As a reminder, we consider results of Pearson's tests statistically significant at α<0.05. Consequently, we reject to null hypothesis H_{01} and conclude that there is a predominance of Types 3 and 4 in all different ecosystems and this observation is not related to a specific ecosystem. When combined into our first group, Types 1 & 2 versus Types 3 & 4, there are significantly more Types 3 and 4 (89.5 %) than Types 1 and 2 (10.5 %).

3.2 How complex is each type of bugs?

To answer $\mathbf{RQ}_2$, we analyze the complexity of each bug in terms of duplication, fixing time, comments, reopenning, files impacted, severity, changesets, hunks and chunks.

Figure 5 presents nine boxplots describing our complexity metric for each type of each ecosystem. In each sub-figure, the book plates are organized as follows: (a) Types 1 to 4 bugs for the Apache ecosystem, (b) Types 1 to 4 bugs for the Netbeans ecosystem and (c) Types 1 to 4 bugs for both ecosystems combined. For all the metrics, except the severity, the median is close to zero and we can observe many outliers. Tables 3, 4 and 5 present

Table 2
Contingency table and Pearson's chi-squared tests

Ecosystem	T1	T2	T3	T4	Pearson's chi-squared p-Value
Apache	1968 (14.3 %)	1248 (9.1 %)	3101 (22.6 %)	7422 (54 %)	
Netbeans	776 (2.9 %)	240 (0.9 %)	8372 (31.3 %)	17366 (64.9 %)	<0.01
Overall	2744 (6.8 %)	1488 (3.7 %)	11473 (28.3 %)	24788 (61.2 %)	
	Types 1 and 2		Types 3 and 4		
Apache	3216 (23.4 %)		10523 (76.6 %)		
Netbeans	1016 (3.8 %)		25738 (96.2 %)		<0.01
All	4232 (10.5 %)		36261 (89.5 %)		

descriptive statistics about each metric for each type for the Apache ecosystem, the Netbeans ecosystem, and both ecosystems combined, respectively. The descriptive statistics used are μ:mean, $\sum$:sum, $\hat{x}$:median, σ:standard deviation and $\%$:percentage. In addition, to the descriptive statistics, these tables present matrices of Mann-Whitney test for each metric and type. We added the ✓ symbol to the Mann-Whitney tests results columns when the value is statistically significant (e.g. α<0.05) and ✗ otherwise.

Finally, Table 6 presents the Pearson's chi-squared test results for each complexity metric for Types 1 to 4 and our two types combination. In what follows, we present our findings for each complexity metric. Complexity metrics are divided into two groups: (a) process and (b) code metrics. Process metrics refer to metrics that have been extracted from the project tracking system (i.e., fixing time, comments, reopening and severity). Code metrics are directly computed using the source code used to fix a given bug (i.e., files impacted, changesets required, hunks and chunks). We acknowledge that these complexity metrics only represent an abstraction of the actual complexity of a given bug as they cannot account for the actual thought process and expertise required to craft a fix. However, we believe that they are an accurate abstraction. Moreover, they are used in several studies in the field to approximate the complexity of a bug [1], [21], [22], [23], [24].

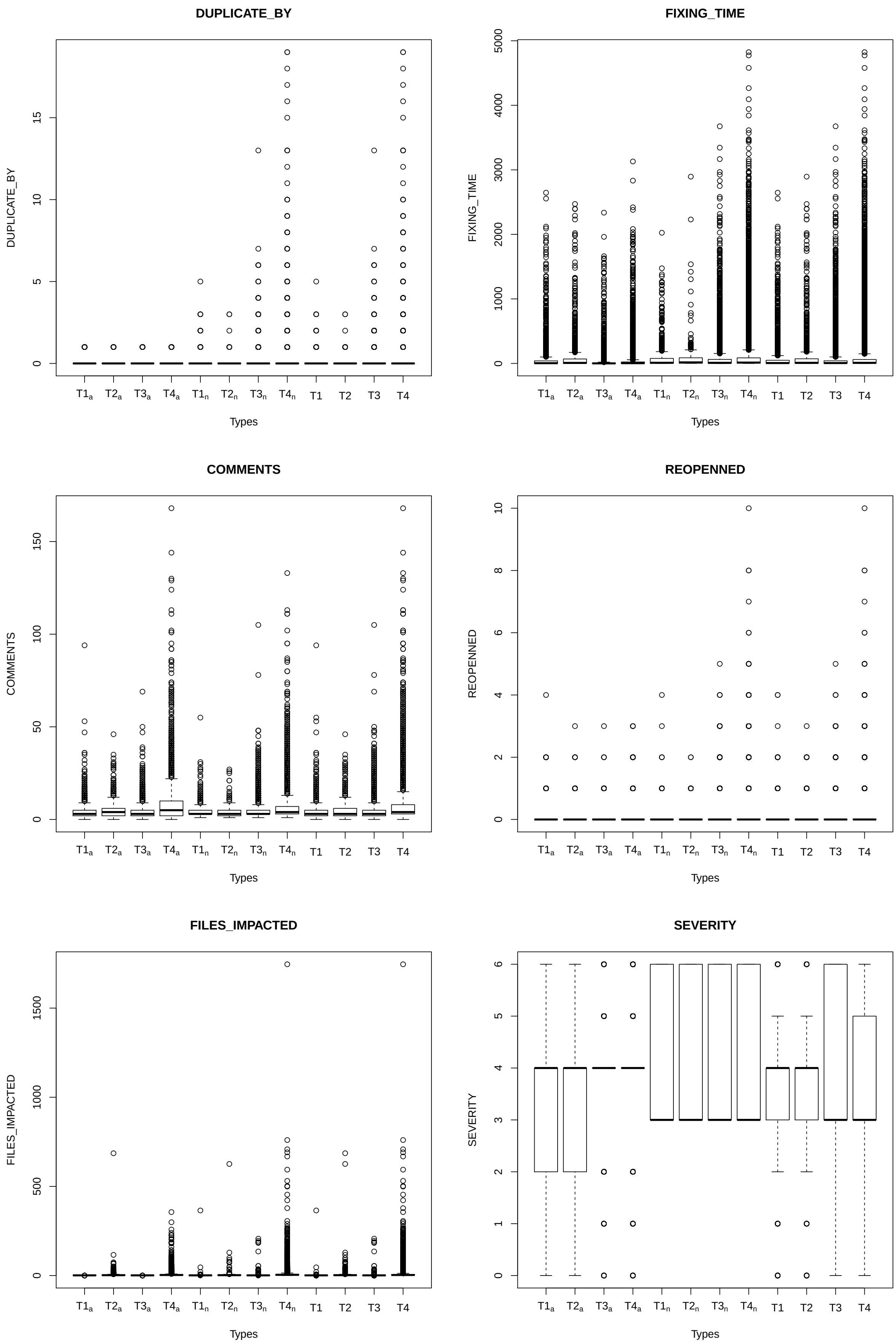
Duplicate: The duplicate metric represents the number of times a bug gets resolved using the *duplicate* label while referencing one of the *resolved/fixed* bug of our dataset. The process metric is useful to approximate the impact of a given bug on the community. For a bug to be resolved using the *duplicate*, it means that the bug has been reported before. The more a bug gets reported by the community, the more people are impacted enough to report it. Note that, for a bug$_a$ to be resolved using the *duplicate* label and referencing bug$_b$, bug$_b$ does not have to resolved itself. Indeed, bug$_b$ could be under investigation (i.e. *unconfirmed*) or being fixed (i.e. *new* or *assigned*). Automatically detecting duplicate bug report is a very active research field [6], [8], [25], [26], [27], [28] and a well-known measure for bug impact.

In the Apache ecosystem, the types that are most likely to get duplicated, ordered by ascending mean duplication rate, are T3 (0.016) < T2 (0.022) < T1 (0.026) < T4 (0.029) and they represent 14.8%, 8.1%, 14.5% and 62.6% of the total duplications, respectively. The differences between duplication means by types, however, are only significant in 33.33% (4/12) of the case. Indeed, the mean duplication is only significant in the following cases: T1 vs. T3, T3 vs. T4. For the Apache ecosystem, we can conclude that $T4^1_{dup} \gg T1^2_{dup} \gg T3^4_{dup}$. We use the notation $x^r_m \gg y^r_m$ ($x^r_m \ll y^r_m$) to represent that x, along the metric m, is significantly greater (lower) than y, along the same metric, according to the mann-whitney tests (α<0.05). r represents the rank of x (y) according to m from 1 (higher percentage) to 4 (lower percentage). In the netbeans ecosystem, we have a different order with T2 (0.067) <T3 (0.074) <T1 (0.086) <T4 (0.113) and they represent 0.6%, 23.3%, 2.5% and 73.6% of the overall duplication, respectively. Also, we have $T4^1_{dup} \gg T3^2_{dup}$ for the netbeans ecosystem.

Overall, the complexity of bug types in terms of the number of duplicates is as follows: $T4^1_{dup} \gg T1^3_{dup} > T3^2_{dup} \gg T2^4_{dup}$.

Fixing time: The fixing time metric represents the time it took for the bug report to go from the *new* state to the *closed* state. If the bug report is reopenned, then the time it took for the bug to go from the *assigned* state to the *closed* state is added to the first time. A bug report can be reopened several times and all the times are added. In this section, the time is expressed in days [2], [29], [30].

In the Apache ecosystem, the types that take the most time to fix are $T2^3_{time} \gg T1^2_{time} \gg T4^1_{time} \gg T3^4_{time}$. The results for the Apache ecosystem might appear surprising at first sight. Indeed, the types requiring the fewer fix location to take longer to fix. However, this is concordant to the finding of Saha *et al.* on long lived bugs [21] where the authors discovered that bugs that stay open the longest are, in fact, bugs that take the fewest locations to fix. In the Netbeans ecosystem, however, the order of bug type along the fixing time metric is different: $T4^1_{time} > T2^4_{time} \gg T1^3_{time} > T3^2_{time}$. This contradicts the finding of Saha *et al.*, however, they did not study the Netbeans ecosystem in their paper [21]. When


DUPLICATE_BY
FIXING_TIME
COMMENTS
REOPENNED
REOPENED
FILES_IMPACTED
SEVERITY
Types
T1a T2a T3a T4a T1n T2n T3n T4n T1 T2 T3 T4

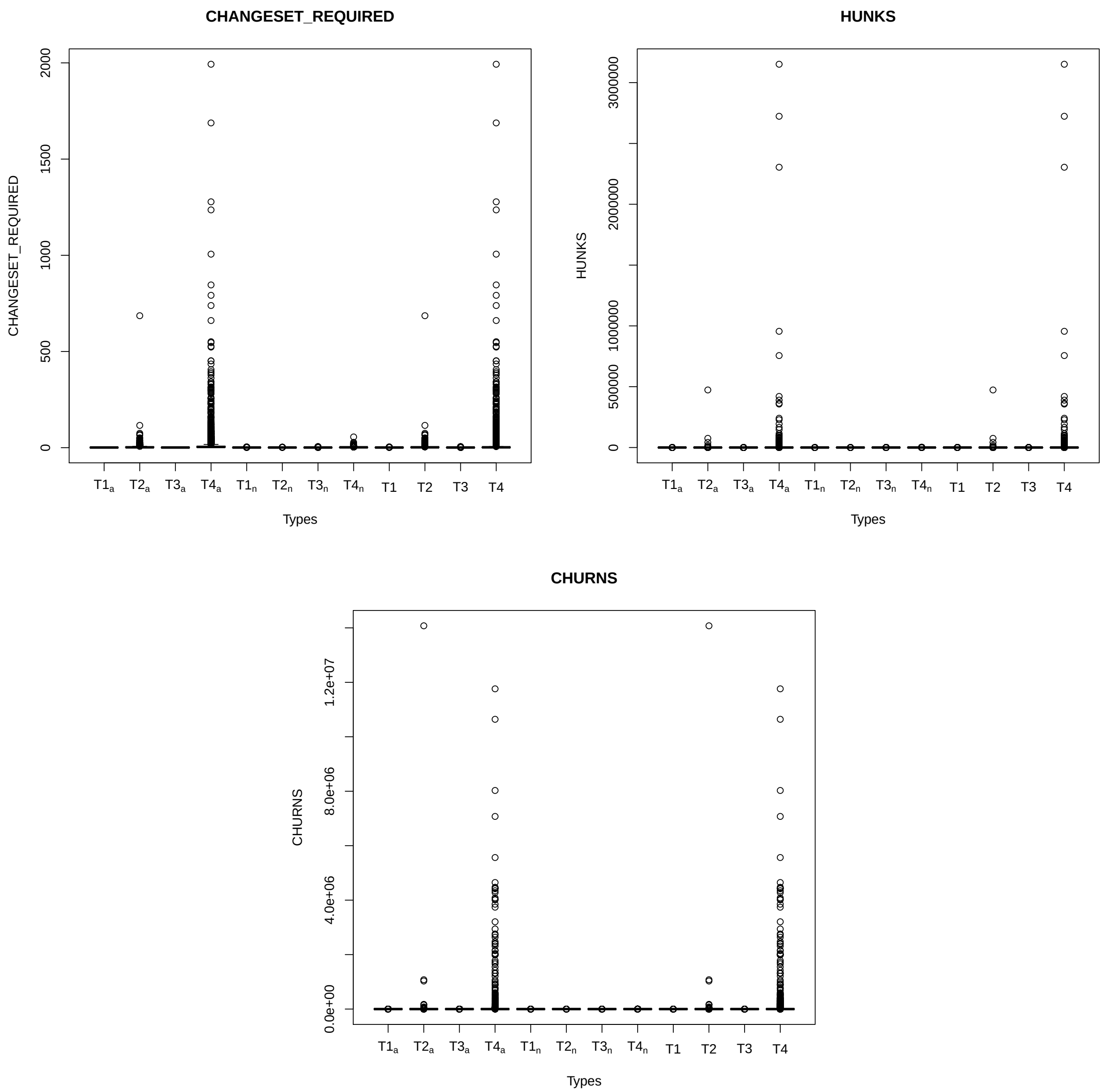


Figure 5. Complexity metrics boxplots. From left to right and top to bottom: Duplicate, Fixing time, Comments, Reopening, Files impacted, Severity, Changesets, Hunks and Chunks.

combined, both ecosystem amounts in the following order $T2^4_{time} > T4^1_{time} \gg T1^3_{time} \gg T3^2_{time}$.

Comments: The number of comments metric refers to the comments that have been posted by the community on the project tracking system. This third process metric evaluates the complexity of a given bug in a sense that if it takes more comments (explanation) from the reporter or the assignee to provide a fix, then the bug must be more complex to understand. The number of comments has been shown to be useful in assessing the complexity of bugs [2], [30]. It is also used in bug prediction approaches [31], [32].

The analysis of the Mann-Whitney test matrix, in respect of comments, for the Apache ecosystem provides the following results: $T4^1_{comment} \gg T2^4_{comment} \gg T3^2_{comment} > T1^3_{comment}$. In the Netbeans ecosystem, the bug types follows a different result: $T4^1_{comment} \gg T3^2_{comment} > T1^3_{comment} \gg T2^4_{comment}$. When combining both ecosystems, the results are: $T4^1_{comment} \gg T2^4_{comment} > T3^2_{comment} \gg T1^3_{comment}$.

Table 3
Apache Ecosystem Complexity Metrics Comparison and Mann-whitney test results.
μ:mean, $\sum$:sum, $\hat{x}$:median, σ:standard deviation, %:percentage

Types	Metric	μ	$\sum$	$\hat{x}$	σ	%	T1	T2	T3	T4
T1	Dup.	0.026	51	0	0.2	14.8	n.a	✗(0.53)	✓(<0.05)	✗(0.45)
	Tim.	91.574	180217	4	262	21.8	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Com.	4.355	8571	3	4.7	9.5	n.a	✓(<0.05)	✗(0.17)	✓(<0.05)
	Reo.	0.062	122	0	0.3	13.8	n.a	✗(0.29)	✓(<0.05)	✓(<0.05)
	Fil.	0.991	1950	1	0.1	3.7	n.a	✓(<0.05)	✗(0.28)	✓(<0.05)
	Sev.	3.423	6737	4	1.3	13.2	n.a	✗(0.18)	✓(<0.05)	✓(<0.05)
	Cha.	1	1968	1	0	1.9	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Hun.	3.814	7506	3	2.4	0	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Chur.	18.761	36921	7	48.6	0	n.a	✓(<0.05)	✗(0.09)	✓(<0.05)
T2	Dup.	0.022	28	0	0.1	8.1	✗(0.53)	n.a	✗(0.16)	✗(0.19)
	Tim.	115.158	143717	8	294.1	17.4	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Com.	5.041	6291	4	4.7	7	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Reo.	0.071	89	0	0.3	10.1	✗(0.29)	n.a	✓(<0.05)	✗(0.59)
	Fil.	4.381	5468	2	20.4	10.5	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Sev.	3.498	4365	4	1.2	8.6	✗(0.18)	n.a	✓(<0.05)	✓(<0.05)
	Cha.	4.681	5842	2	20.4	5.5	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Hun.	561.995	701370	14	13628.2	3.9	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Chur.	14184.869	17702716	88	400710.2	8	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
T3	Dup.	0.016	50	0	0.1	14.5	✓(<0.05)	✗(0.16)	n.a	✓(<0.05)
	Tim.	35.892	111300	1	151.8	13.5	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Com.	4.422	13712	3	4.4	15.2	✗(0.17)	✓(<0.05)	n.a	✓(<0.05)
	Reo.	0.033	101	0	0.2	11.5	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Fil.	0.994	3081	1	0.1	5.9	✗(0.28)	✓(<0.05)	n.a	✓(<0.05)
	Sev.	3.644	11300	4	1.1	22.2	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Cha.	1	3101	1	0	2.9	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Hun.	4.022	12472	3	3.4	0.1	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Chur.	16.954	52573	6	49.8	0	✗(0.09)	✓(<0.05)	n.a	✓(<0.05)
T4	Dup.	0.029	216	0	0.2	62.6	✗(0.45)	✗(0.19)	✓(<0.05)	n.a
	Tim.	52.76	391586	4	182.2	47.4	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Com.	8.313	61701	5	10.2	68.3	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Reo.	0.077	570	0	0.3	64.6	✓(<0.05)	✗(0.59)	✓(<0.05)	n.a
	Fil.	5.633	41805	3	14	79.9	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Sev.	3.835	28466	4	1	56	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Cha.	12.861	95455	4	52.2	89.7	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Hun.	2305.868	17114149	30	58094.7	96	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Chur.	27249.773	202247816	204	320023.5	91.9	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a

Bug Reopening: The bug reopening metric counts how many times a given bug gets reopened.If a bug report is reopened, it means that the fix was arguably hard to come up with or the report was hard to understand [33] [34] [35]. In the Apache and Netbeans ecosystems, we found that the order bug types of the bugs that are reopened is the same: $T4^1_{reop} > T2^4_{reop} \gg T3^3_{reop} \gg T1^2_{reop}$. and $T4^1_{reop} > T2^4_{reop} > T3^2_{reop} > T1^3_{reop}$, respectively. When combined, however, the order does change: $T4^1_{reop} > T2^4_{reop} > T1^3_{reop} \gg T3^2_{reop}$.

Severity: The severity metric reports the degree of impact of the report on the software. Predicting the severity of a given report is an active research field [36], [37], [38], [39], [40], [41] and it helps to prioritization of fixes [42]. The severity is a textual value (blocker, critical, major, normal, minor, trivial) and the Mann-Whitney test only accepts numerical input. Consequently, we had to assign numerical values to each severity. We chose to assign values from 1 to 6 for trivial, minor, normal, major, critical and blocker severities, respectively. The bug type ordering according to the severity metrics is: $T4^1_{sev} \gg T3^2_{sev} \gg T2^4_{sev} > T1^3_{sev}$, $T2^4_{sev} > T1^3_{sev} \gg T3^2_{sev} \gg T4^1_{sev}$ and $T4^1_{sev} \gg T3^2_{sev} > T1^3_{sev} > T2^4_{sev}$ for Apache, Netbeans, and both combined, respectively.

Files impacted: The number of files impacted measures how many files have been modified for the bug report to be closed. Unsurprisingly, Types 4 and 2 are the ones with the most files impacted. Indeed, according to their definitions, presented in Figure 2, Types 1 and 3 only need a modification in one location. This metric is therefore applicable to bug Types 2 and 4 only. In Apache, type 4 structures are wider than type 2. (

Table 4
Netbeans Ecosystem Complexity Metrics Comparison and Mann-whitney test results.
μ:mean, $\sum$:sum, $\hat{x}$:median, σ:standard deviation, %:percentage

Types	Metric	μ	$\sum$	$\hat{x}$	σ	%	T1	T2	T3	T4
	Dup.	0.086	67	0	0.4	2.5	n.a	✗(0.39)	✗(0.24)	✗(0.86)
	Tim.	92.759	71981	10	219.1	2.3	n.a	✓(<0.05)	✗(0.15)	✓(<0.05)
	Com.	4.687	3637	3	4.1	2.4	n.a	✓(<0.05)	✗(0.83)	✓(<0.05)
	Reo.	0.054	42	0	0.3	1.9	n.a	✗(0.1)	✗(0.58)	✓(<0.05)
T1	Fil.	1.735	1346	1	13.2	0.8	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Sev.	4.314	3348	3	1.5	3.1	n.a	✗(0.66)	✓(<0.05)	✓(<0.05)
	Cha.	1.085	842	1	0.4	2	n.a	✗(0.99)	✗(0.26)	✓(<0.05)
	Hun.	4.405	3418	3	7	0.5	n.a	✓(<0.05)	✗(0.13)	✓(<0.05)
	Chur.	5.089	3949	2	12.5	0.3	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Dup.	0.067	16	0	0.3	0.6	✗(0.39)	n.a	✗(0.73)	✗(0.39)
	Tim.	111.9	26856	16	308.6	0.9	✓(<0.05)	n.a	✓(<0.05)	✗(0.41)
	Com.	4.433	1064	3	4	0.7	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Reo.	0.079	19	0	0.3	0.9	✗(0.1)	n.a	✗(0.11)	✗(0.97)
T2	Fil.	8.804	2113	2	42.7	1.3	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Sev.	4.362	1047	3	1.5	1	✗(0.66)	n.a	✓(<0.05)	✓(<0.05)
	Cha.	1.075	258	1	0.3	0.6	✗(0.99)	n.a	✗(0.5)	✓(<0.05)
	Hun.	21.887	5253	8	62.7	0.7	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Chur.	32.263	7743	8	125.8	0.7	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Dup.	0.074	620	0	0.4	23.3	✗(0.24)	✗(0.73)	n.a	✓(<0.05)
	Tim.	87.033	728642	9	233.6	23.8	✗(0.15)	✓(<0.05)	n.a	✓(<0.05)
	Com.	4.73	39599	3	4.3	26.5	✗(0.83)	✓(<0.05)	n.a	✓(<0.05)
	Reo.	0.06	499	0	0.3	22.7	✗(0.58)	✗(0.11)	n.a	✓(<0.05)
T3	Fil.	1.306	10932	1	5.1	6.8	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Sev.	4.021	33666	3	1.4	31.4	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Cha.	1.065	8917	1	0.3	21	✗(0.26)	✗(0.5)	n.a	✓(<0.05)
	Hun.	5.15	43115	3	12.4	5.8	✗(0.13)	✓(<0.05)	n.a	✓(<0.05)
	Chur.	6.727	56317	2	22	4.9	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Dup.	0.113	1959	0	0.7	73.6	✗(0.86)	✗(0.39)	✓(<0.05)	n.a
	Tim.	128.833	2237319	13	332.8	73	✓(<0.05)	✗(0.41)	✓(<0.05)	n.a
	Com.	6.058	105202	4	6.7	70.4	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Reo.	0.094	1639	0	0.4	74.5	✓(<0.05)	✗(0.97)	✓(<0.05)	n.a
T4	Fil.	8.408	146019	4	25.1	91	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Sev.	3.982	69159	3	1.4	64.5	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Cha.	1.871	32494	2	1.2	76.4	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Hun.	40.195	698022	13	98.3	93.1	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Chur.	61.893	1074830	15	178.6	94	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a

$T4^1_{files} \gg T2^2_{files} \gg T3^3_{files} <=> T1^4_{files}$) while in Netbeans, type 2 are wider ($T2^3_{files} \gg T4^1_{files} \gg T3^2_{files} <=> T1^4_{files}$). Overall, types 4 impacts more files than types 2 while types 1 and 2 impacts only 1 file ($T4^1_{files} \gg T2^3_{files} \gg T3^2_{files} <=> T1^4_{files}$).

Changesets: The changeset metrics registers how many changesets (or commits/patch/fix) have been required to close the bug report. In the project tracking system, changesets to resolve the bug are proposed and analysed by the community, automated quality insurance tools and the quality insurance team itself. Each changeset can be either accepted and applied to the source code or dismissed. The number of changesets (or versions of a given changeset) it takes before an integration can hint us about the complexity of the fix. In case the bug report gets reopen and new changesets proposed, the new changesets (after the reopening) are added to the old ones (before the reopening). For the Apache ecosystem, we found the following: $T4^1_{changesets} \gg T2^2_{changesets} \gg T1^4_{changesets} <=> T3^3_{changesets}$. In the Netbeans ecosystem, the order stays the same at the exception of Types 1 and 2 that switch position from 3 to 2 and 2 to 3, respectively. $T4^1_{changesets} \gg T1^3_{changesets} > T2^4_{changesets} > T3^2_{changesets}$. Overall, Type 4 bugs are the most complex bugs in terms of the number of submitted changesets ($T4^1_{changesets} \gg T2^3_{changesets} \gg T3^2_{changesets} \gg T1^4_{changesets}$).

[WAHAB: I don't see how the following sentences are related to your work] While results have been published on the bug-fix patterns [43], smell introduction [44], [45], to the best of our knowledge, no one interested themselves in how many iterations of a patch were required to close

Table 5
Apache and Netbeans Ecosystems Complexity Metrics Comparison and Mann-whitney test results.
μ:mean, $\sum$:sum, $\hat{x}$:median, σ:standard deviation, %:percentage

Types	Metric	μ	$\sum$	$\hat{x}$	σ	%	T1	T2	T3	T4
T1	Dup.	0.043	118	0	0.3	3.9	n.a	✗(0.09)	✗(0.16)	✓(<0.05)
	Tim.	91.909	252198	6	250.6	6.5	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Com.	4.449	12208	3	4.5	5.1	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Reo.	0.06	164	0	0.3	5.3	n.a	✗(0.07)	✓(<0.05)	✓(<0.05)
	Fil.	1.201	3296	1	7	1.5	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Sev.	3.675	10085	4	1.4	6.4	n.a	✗(0.97)	✗(0.17)	✓(<0.05)
	Cha.	1.024	2810	1	0.2	1.9	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Hun.	3.981	10924	3	4.3	0.1	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
	Chur.	14.894	40870	5	42.2	0	n.a	✓(<0.05)	✓(<0.05)	✓(<0.05)
T2	Dup.	0.03	44	0	0.2	1.5	✗(0.09)	n.a	✓(<0.05)	✓(<0.05)
	Tim.	114.632	170573	9	296.4	4.4	✓(<0.05)	n.a	✓(<0.05)	✗(0.15)
	Com.	4.943	7355	3	4.6	3.1	✓(<0.05)	n.a	✗(0.72)	✓(<0.05)
	Reo.	0.073	108	0	0.3	3.5	✗(0.07)	n.a	✓(<0.05)	✗(0.47)
	Fil.	5.095	7581	2	25.4	3.6	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Sev.	3.637	5412	4	1.3	3.4	✗(0.97)	n.a	✗(0.44)	✗(0.1)
	Cha.	4.099	6100	2	18.7	4.1	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Hun.	474.881	706623	12	12481.7	3.8	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
	Chur.	11902.19	17710459	62	366988	8	✓(<0.05)	n.a	✓(<0.05)	✓(<0.05)
T3	Dup.	0.058	670	0	0.4	22.3	✗(0.16)	✓(<0.05)	n.a	✓(<0.05)
	Tim.	73.21	839942	6	215.8	21.6	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Com.	4.647	53311	3	4.3	22.2	✓(<0.05)	✗(0.72)	n.a	✓(<0.05)
	Reo.	0.052	600	0	0.3	19.5	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Fil.	1.221	14013	1	4.4	6.6	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Sev.	3.919	44966	3	1.4	28.4	✗(0.17)	✗(0.44)	n.a	✓(<0.05)
	Cha.	1.048	12018	1	0.3	8.1	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Hun.	4.845	55587	3	10.7	0.3	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
	Chur.	9.491	108890	3	32.3	0	✓(<0.05)	✓(<0.05)	n.a	✓(<0.05)
T4	Dup.	0.088	2175	0	0.6	72.3	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Tim.	106.056	2628905	9	297.9	67.6	✓(<0.05)	✗(0.15)	✓(<0.05)	n.a
	Com.	6.733	166903	4	8	69.6	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Reo.	0.089	2209	0	0.4	71.7	✓(<0.05)	✗(0.47)	✓(<0.05)	n.a
	Fil.	7.577	187824	3	22.4	88.3	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Sev.	3.938	97625	3	1.3	61.8	✓(<0.05)	✗(0.1)	✓(<0.05)	n.a
	Cha.	5.162	127949	2	29	85.9	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Hun.	718.58	17812171	16	31804.5	95.8	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a
	Chur.	8202.463	203322646	28	175548.3	91.9	✓(<0.05)	✓(<0.05)	✓(<0.05)	n.a

a bug report beside us.

Hunks: The hunks metric counts the number of consecutive code blocks of modified, added or deleted lines in textual files. Hunks are used to determine, in each file, how many different places a developer has modified. This metric is widely used for bug insertion prediction [46], [47], [48] and bug-fix comprehension [43]. In our ecosystems, there is a relationship between the number of files modified and the hunks. The number of code blocks modified is likely to rise as to the number of modified files as the hunks metric will be at least 1 per file. We found that Types 2 and 4 bugs, that requires many files to get fixed, are the ones that have significantly higher scores for the hunks metric; Apache ecosystem: $T4^1_{hunks} \gg T2^2_{hunks} \gg T3^3_{hunks} \gg T1^4_{hunks}$, Netbeans ecosystem: $T4^1_{hunks} \gg T2^3_{hunks} \gg T3^2_{hunks} \gg T1^4_{hunks}$, and overall $T4^1_{hunks} \gg T2^2_{hunks} \gg T1^4_{hunks} \gg T3^3_{hunks}$.

Churns: The last metrics, churns, counts the number of lines modified. The churn value for a line change should be at least two as the line has to be deleted first and then added back with the modifications. Once again, this is a widely used metric in the field [43], [46], [47], [48]. Once again, Types 4 and 2 are the ones with the most churns; Apache ecosystem $T4^1_{churns} \gg T2^2_{churns} \gg T1^4_{churns} > T3^3_{churns}$, Netbeans ecosystem: $T4^1_{churns} \gg T2^3_{churns} \gg T3^2_{churns} \gg T1^4_{churns}$. and overall : $T4^1_{churns} \gg T2^2_{churns} \gg T1^4_{churns} \gg T3^3_{churns}$.

[WAHAB: I will edit this late] Assuming that the complexity metrics are equal in terms of assessing the complexity of a given bug, we scored each type with a simple system. We counted how many times each bug type obtained each position in our nine rankings and

Table 6
Pearson's chi-squared tests for complexity metrics

Eco.	Metric	T1 vs. T2 vs. T3 vs. T4	T1T2 v. T3T4
Apache	Dup.	<0.01	<0.01
	Tim.	<0.01	<0.01
	Com.	<0.01	<0.01
	Reo.	<0.01	<0.01
	Fil.	<0.01	<0.01
	Sev.	<0.01	<0.01
	Cha.	<0.01	<0.01
	Hun.	<0.01	<0.01
	Chur.	<0.01	<0.01
Netbeans	Dup.	<0.01	<0.01
	Tim.	<0.01	<0.01
	Com.	<0.01	<0.01
	Reo.	<0.01	<0.01
	Fil.	<0.01	<0.01
	Sev.	<0.01	<0.01
	Cha.	<0.01	<0.01
	Hun.	<0.01	<0.01
	Chur.	<0.01	<0.01
Overall	Dup.	<0.01	<0.01
	Tim.	<0.01	<0.01
	Com.	<0.01	<0.01
	Reo.	<0.01	<0.01
	Fil.	<0.01	<0.01
	Sev.	<0.01	<0.01
	Cha.	<0.01	<0.01
	Hun.	<0.01	<0.01
	Chur.	<0.01	<0.01

multiply them by 4 for the first place, 3 for the second, 2 for the third and 1 for the fourth place. We did the same simple analysis of the rank of each type for each metric, to take into account the frequency of bug types in our calculation, and multiply both values. The complexity scores we calculated are as follows: 1330, 1750, 2580 and 7120 for bug types 1, 2, 3 and 4, respectively. According to these complexity scores, types 3 and 4 are more complex than types 1 and 2. In order to confirm or infirm the validity of our complexity scores, we ran our experiments again. This time, we combined types 1 & 2 and types 3 & 4 for the two ecosystems. As shown by Table 7, our complexity scores are meaningful. Indeed, Types 3 & 4 are statistically more complex ($\gg$) than Types 1 & 2 according to the duplicate, fixing time, comments, files impacted, changesets, hunks and churns complexity metrics. Also, Types 3 & 4 get reopen more than types 1 & 2, in average, but the result of the mann-whitney test is not conclusive (i.e. $\alpha > 0.05$). Out of our nine complexity metrics, the only one where Types 1 & 2 perform *worst* than Types 3 & 4 is the severity.

Consequently, we reject to null hypothesis H_{02} and conclude that the complexity of bug is related to its type. Moreover, Types 3 and 4 bugs are more complex than Types 1 and 2 bugs across the ecosystems we studied.

3.3 How pertinent is a bug taxonomy?

4 Dicussion

In this section, we discuss the answers of our two research questions.

4.1 RQ_1: What are the proportions of different types of bugs?

One important finding of this study is that there are significantly more Types 3 and 4 bugs than Types 1 and 2 in all studied systems. Moreover, this observation is not system-specific. The traditional one-bug/one-fault (i.e., Type 1) way of thinking about bugs only accounts for 6.8% of the bugs.

We believe that, triaging algorithms (e.g., [27], [49], [50], [51]) can benefit from these findings by developing techniques that can detect Type 2 and 4 bugs. This would result in better performance in terms of reducing the cost, time and efforts required by the developers in the bug fixing process.

4.2 RQ_2: How complex is each type of bugs?

To evaluate the complexity of each types of bug, we have computed five process metrics (time to close, duplications, reopenings, comments and severity) and four code metrics (files, commit, hunks and churns).

Process complexity: For four out the five process metrics we used, we found that Types 3 and 4 bugs combined performed significantly worse than Types 1 and 2 bugs. [WAHAB: What do you mean by 'performed', This is not clear. You should state this differently] The only process metric where Types 3 and 4 bugs do not perform significantly worse than Types 1 and 2 bugs is severity. Although clear guidelines exist on how to assign bug severity, it remains a manual process done by bug reporter. In addition, previous studies, notably those by Khomh et al. [50], showed that severity is not a consistent/trustworthy characteristic of a bug report. This led to the emergence of studies for predicting bug severity (e.g., [38], [39], [52]). Nevertheless, we discovered that, in our ecosystems, types 3 and 4 are have an higher severity than types 1 and 2.

Code complexity: All our code metrics (files, commit, hunks and churns) have shown similar results. In all cases, Types 3 and 4 bugs have been shown to be more complex than Types 1 and 2 bugs.

While current approaches that aim to predict which bug will be reopen use the number of modified files [33], [34], [35], we believe that they can be improved by taking into account the type of a bug. For example, if we can

Table 7
Types 1& 2 versus Types 3 & 4 Complexity Metrics Comparison and Mann-whitney test results.
μ:mean, $\sum$:sum, $\hat{x}$:median, σ:standard deviation, %:percentage

Ecosystem	Metric	Types 1 and 2					Types 3 and 4					Mann-Whitney p-value
		μ	$\sum$	$\hat{x}$	σ	%	μ	$\sum$	$\hat{x}$	σ	%	
Apache	Dup	0.025	79	0	0.2	22.9	0.025	266	0	0.2	77.1	✗(0.82)
	Time	100.726	323934	6	275.1	39.2	47.789	502886	3	17 3.9	60.8	✓(<0.05)
	Com	4.621	14862	3	4.7	16.5	7.166	75413	4	9.1	83.5	✓(<0.05)
	Reop	0.066	211	0	0.3	23.9	0.064	671	0	0.3	76.1	✗(0.74)
	Files	2.307	7418	1	12.8	14.2	4.266	44886	2	11.9	85.8	✓(<0.05)
	Severity	3.452	11102	4	1.2	21.8	3.779	39766	4	1	78.2	✓(<0.05)
	Change	2.428	7810	1	12.8	7.3	9.366	98556	3	44.2	92.7	✓(<0.05)
	Hunks	220.422	708876	4	8491.9	4	1627.542	17126621	15	48799.9	96	✓(<0.05)
	Churns	5516.056	17739637	15	249654.4	8.1	19224.593	202300389	72	269046.2	91.9	✓(<0.05)
Netbeans	Dup	0.082	83	0	0.4	3.1	0.1	2579	0	0.6	96.9	✗(0.92)
	Time	97.281	98837	11	243.2	3.2	115.237	2965961	12	304.8	96.8	✗(0.76)
	Com	4.627	4701	3	4	3.1	5.626	144801	4	6.1	96.9	✓(<0.05)
	Reop	0.06	61	0	0.3	2.8	0.083	2138	0	0.4	97.2	✗(0.08)
	Files	3.405	3459	1	23.9	2.2	6.098	156951	2	21.1	97.8	✓(<0.05)
	Severity	4.326	4395	3	1.5	4.1	3.995	102825	3	1.4	95.9	✓(<0.05)
	Change	1.083	1100	1	0.4	2.6	1.609	41411	1	1.1	97.4	✓(<0.05)
	Hunks	8.534	8671	3	31.9	1.2	28.795	741137	8	82.7	98.8	✓(<0.05)
	Churns	11.508	11692	3	63.1	1	43.949	1131147	8	149.5	99	✓(<0.05)
Overall	Dup	0.038	162	0	0.2	5.4	0.078	2845	0	0.5	94.6	✓(<0.05)
	Time	99.899	422771	7	267.8	10.9	95.663	3468847	8	275	89.1	✓(<0.05)
	Com	4.623	19563	3	4.6	8.2	6.073	220214	4	7.1	91.8	✓(<0.05)
	Reop	0.064	272	0	0.3	8.8	0.077	2809	0	0.3	91.2	✗(0.21)
	Files	2.57	10877	1	16.2	5.1	5.566	201837	2	18.9	94.9	✓(<0.05)
	Severity	3.662	15497	4	1.4	9.8	3.932	142591	3	1.3	90.2	✓(<0.05)
	Change	2.105	8910	1	11.2	6	3.86	139967	2	24.1	94	✓(<0.05)
	Hunks	169.553	717547	4	7403	3.9	492.754	17867758	9	26297.9	96.1	✓(<0.05)
	Churns	4194.548	17751329	10	217637.4	8	5610.202	203431536	13	145192.5	92	✓(<0.05)

detect that an incoming bug if of Type 3 or 4 then it is more likely to reopened than a bug of Type 1 or 2. Similarly, approaches aiming to predict the files in which a given bug should be fixed could be categorized and improved by knowking the bug type in advance [53], [54]. Similarly to reopening, we believe that approaches targeting the identification of duplicates [7], [26], [27], [28] could leverage this taxonomy to achieve even better performances in terms of recall and precision. Finally, we believe that, approaches that aim to predict the fixing time of a bug (e.g., [2], [32], [55]) can highly benefit from accurately predicting the type of a bug and therefore better plan the required man-power to fix the bug.

5 Related Works

Researchers have been studying the relationships between the bug and source code repositories for more than two decades. To the best of our knowledge the first ones who conducted this type of study on a significant scale were Perry and Stieg [56]. In these two decades, many aspects of these relationships have been studied in length. For example, researchers were interested in improving the bug reports themselves by proposing guidelines [14], and by further simplifying existing bug reporting models [57].

Another field of study consist of assigning these bug reports, automatically if possible, to the right developers during triaging [23], [49], [51], [58]. Another set of approaches focus on how long it takes to fix a bug [2], [21], [32] and where it should be fixed [53], [59]. With the rapidly increasing number of bugs, the community was also interested in prioritizing bug reports [60], and in predicting the severity of a bug [38]. Finally, researchers proposed approaches to predict which bug will get re-opened [33], [35], which bug report is a duplicate of another one [26], [27], [28] and which locations are likely to yield new bugs [44], [46], [61]. However, to the best of our knowledge, there are not many attempts to classify bugs the way we present in this paper. In her PhD thesis [62], Sigrid Eldh discussed the classification of trouble reports with respect to a set of fault classes that she identified. Fault classes include computational logical faults, ressource faults, function faults, etc. She conducted studies on Ericsson systems and showed the distributions of trouble reports with respect to these fault classes. A research paper was published on the topic in [62]. or safety critical [63]. Hamill et al. [63] proposed a classification of faults and failures in critical safety systems. They proposed several types of faults and show how failures in critical safety systems relate to these classes. They found that only a few fault types were responsible for the majority of failures. They also compare on pre-release and post-release faults and showed that the distributions of fault types differed for pre-release and post-release failures. Another finding is that coding faults are the most predominant ones.

Our study differs from theses studies in the way that we focus on the bugs and their fixes across a wide range of systems, programming languages, and purposes. This is done independtly from a specific class of faults (such as coding faults, resource faults, etc.). This is because our aim is not to improve testing as it is the case in the work of Eldh [62] and Hamill et al. [63]. Our objective is to propose a classification that can allow researchers in the filed of mining bug repositiories to use the taxonomy as a new criterion in triaging, prediction, and reproduction of bugs. By analogy, we can look at the proposed bug taxonomy in a similar way as the clone taxonomy presented by Kapser and Godfrey [9]. The authors proposed seven types of source code clones and then conducted a case study, using their classification, on the file system module of the Linux operating system. This clone taxonomy continues to be used by researchers to build better approaches for detecting a given clone type and being able to effectively compare approaches with each other.

6 Conclusion

7 Reproduction Package

We provide a reproduction package that is publicly available at [link]. All the instructions needed to reproduce our results are self contained in the provided archive.

References

[1] C. Weiß, T. Zimmermann, and A. Zeller, "How Long will it Take to Fix This Bug ?" in *Fourth International Workshop on Mining Software Repositories (MSR'07)*, no. 2, 2007, p. 1.

[2] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: an empirical study of commercial software projects," in *International Conference on Software Engineering*. IEEE Press, may 2013, pp. 1042–1051. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486931

[3] N. Chen, "Star: stack trace based automatic crash reproduction," Ph.D. dissertation, The Hong Kong University of Science and Technology, 2013.

[4] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, may 2007, pp. 489–498. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4222610

[5] M. Nayrolles, A. Hamou-Lhadj, T. Sofiene, and A. Larsson, "JCHARMING : A Bug Reproduction Approach Using Crash Traces and Directed Model Checking," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering, IEEE, 2015*, 2015, pp. 101–110.

[6] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *29th International Conference on Software Engineering*. IEEE, may 2007, pp. 499–510. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4222611

[7] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 1. New York, New York, USA: ACM Press, may 2010, p. 45. [Online]. Available: http://dl.acm.org/citation.cfm?id=1806799.1806811

[8] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. New York, New York, USA: ACM Press, 2012, p. 70. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6494907

[9] C. Kapser and M. W. Godfrey, "Toward a Taxonomy of Clones in Source Code: A Case Study," in *International Workshop on Evolution of Large Scale Industrial Software Architectures*, 2003, pp. 67–78. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.6056

[10] T. J. McCabe and C. W. Butler, "Design complexity measurement and testing," *Communications of the ACM*, vol. 32, no. 12, pp. 1415–1425, dec 1989. [Online]. Available: http://dl.acm.org/citation.cfm?id=76380.76382

[11] A. Zeller, *Configuration management with version sets: A unified software versioning model and its applications*, 1997.

[12] Bugzilla, "Life Cycle of a Bug," 2008.

[13] T. Koponen, "Life cycle of defects in open source software projects," in *Open Source Systems*. Springer, 2006, pp. 195–200.

[14] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, New York, USA: ACM Press, 2008, p. 308. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1453101.1453146

[15] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, oct 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=630830.631291

[16] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*. New York, New York, USA: ACM Press, nov 2010, p. 97. [Online]. Available: http://dl.acm.org/citation.cfm?id=1882291.1882308

[17] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.*, 2011, pp. 15–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=2025120

[18] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013, pp. 280–289. [Online]. Available: http://dl.acm.org/citation.cfm?id=2550526.2550583

[19] X. O. Wang, E. Baik, and P. T. Devanbu, "System compatibility analysis of Eclipse and Netbeans based on bug data," in *Proceeding of the 8th working conference on Mining software repositories - MSR '11*. New York, New York, USA: ACM Press, may 2011, p. 230. [Online]. Available: http://dl.acm.org/citation.cfm?id=1985441.1985479

[20] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from version control and bug tracking systems," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE Comput. Soc, pp. 23–32. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1235403

[21] R. K. Saha, S. Khurshid, and D. E. Perry, "An empirical study of long lived bugs," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, feb 2014, pp. 144–153. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6747164

[22] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th International Conference on Software Engineering (ICSE)*. Ieee, may 2013, pp. 382–391. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6606584

[23] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceeding of the 28th international conference on Software engineering - ICSE '06*. New York, New York, USA: ACM Press, may 2006, p. 361. [Online]. Available: http://dl.acm.org/citation.cfm?id=1134285.1134336

[24] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings. 27th International Conference on Software Engineering, 2005.* IEEe, 2005, pp. 284–292. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1553571

[25] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 253–262, nov 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6100061

[26] N. Bettenburg, R. Premraj, and T. Zimmermann, "Duplicate bug reports considered harmful ... really?" in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 337–345. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4658082

[27] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 52–61. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4630070

[28] Y. Tian, C. Sun, and D. Lo, "Improved Duplicate Bug Report Identification," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2012, pp. 385–390. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6178884

[29] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How Long Will It Take to Fix This Bug?" in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. IEEE, may 2007, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268983.1269017

[30] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An Empirical Study on Factors Impacting Bug Fixing Time," in *2012 19th Working Conference on Reverse Engineering*. IEEE, oct 2012, pp. 225–234. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6385118

[31] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, may 2010, pp. 31–41. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5463279

[32] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *Proceeding of the 8th working conference on Mining software repositories - MSR '11*. New York, New York, USA: ACM Press, may 2011, p. 207. [Online]. Available: http://dl.acm.org/citation.cfm?id=1985441.1985472

[33] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Proceedings of the 34th International Conference on Software Engineering, IEEE*. IEEE, jun 2012, pp. 1074–1083. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6227112

[34] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting Re-opened Bugs: A Case Study on the Eclipse Project," in *2010 17th Working Conference on Reverse Engineering*. IEEE, oct 2010, pp. 249–258. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5645566

[35] D. Lo, "A Comparative Study of Supervised Learning Algorithms for Re-opened Bug Prediction," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2013, pp. 331–334. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6498482

[36] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *2008 IEEE International Conference on Software Maintenance*. IEEE, sep 2008, pp. 346–355. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4658083

[37] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 1. New York, New York, USA: ACM Press, 2010, p. 495. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6062117

[38] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, may 2010, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5463284

[39] Y. Tian, D. Lo, and C. Sun, "Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction," in *2012 19th Working Conference on Reverse Engineering*. IEEE, oct 2012, pp. 215–224. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6385117

[40] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. New York, New York, USA: ACM Press, may 2014, pp. 72–81. [Online]. Available: http://dl.acm.org/citation.cfm?id=2597073.2597099

[41] K. Havelund, G. Holzmann, and R. Joshi, Eds., *NASA Formal Methods*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, vol. 9058. [Online]. Available: http://link.springer.com/10.1007/978-3-319-17524-9

[42] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012, pp. 25–35. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6227209

[43] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, aug 2008. [Online]. Available: http://link.springer.com/10.1007/s10664-008-9077-5

[44] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," pp. 403–414, may 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818805

[45] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess," in *Proceeding of the 8th working conference on Mining software repositories - MSR '11*. New York, New York, USA: ACM Press, may 2011, p. 153. [Online]. Available: http://dl.acm.org/citation.cfm?id=1985441.1985464

[46] S. Kim, T. Zimmermann, K. Pan, and E. Jr. Whitehead,

"Automatic Identification of Bug-Introducing Changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 81–90. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4019564

[47] Y. Jung, H. Oh, and K. Yi, "Identifying static analysis techniques for finding non-fix hunks in fix revisions," in *Proceeding of the ACM first international workshop on Data-intensive software management and mining - DSMM '09*. New York, New York, USA: ACM Press, nov 2009, p. 13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1651309.1651313

[48] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. New York, New York, USA: ACM Press, aug 2015, pp. 966–969. [Online]. Available: http://dl.acm.org/citation.cfm?id=2786805.2803183

[49] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York, New York, USA: ACM Press, aug 2009, p. 111. [Online]. Available: http://dl.acm.org/citation.cfm?id=1595696.1595715

[50] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, "An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox," in *2011 18th Working Conference on Reverse Engineering*. IEEE, oct 2011, pp. 261–270. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6079850

[51] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, New York, USA: ACM Press, 2011, p. 884. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6032543

[52] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing Mining Algorithms for Predicting the Severity of a Reported Bug," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2011, pp. 249–258. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5741332

[53] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering, IEEE*. IEEE, jun 2012, pp. 14–24. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6227210

[54] D. Kim, Y. Tao, S. Member, S. Kim, and A. Zeller, "Where Should We Fix This Bug? A Two-Phase Recommendation Model," *Transaction on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[55] L. D. Panjer, "Predicting Eclipse Bug Lifetimes," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. IEEE, may 2007, pp. 29–29. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268983.1269043

[56] Perry, Dewayne E. and C. S. Stieg., "Software faults in evolving a large, real-time system: a case study." in *Software EngineeringESEC*, 1993, pp. 48–67.

[57] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "Towards a simplification of the bug report form in eclipse," in *Proceedings of the 2008 international workshop on Mining software repositories - MSR '08*. New York, New York, USA: ACM Press, may 2008, p. 145. [Online]. Available: http://dl.acm.org/citation.cfm?id=1370750.1370786

[58] G. Bortis and A. van der Hoek, "PorchLight: A tag-based approach to bug triaging," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, may 2013, pp. 342–351. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6606580

[59] A. Zeller, "Where Should We Fix This Bug? A Two-Phase Recommendation Model," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, nov 2013. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6517844

[60] D. Kim, X. Wang, S. Member, S. Kim, A. Zeller, S. C. Cheung, S. Member, and S. Park, "Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts," *TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 37, no. 3, pp. 430–447, 2011.

[61] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *29th International Conference on Software Engineering*. IEEE, may 2007, pp. 489–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=1248820.1248881

[62] S. Eldh, "On Test Design," Ph.D. dissertation, Mälardalen, 2001.

[63] M. Hamill and K. Goseva-Popstojanova, "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system," *Software Quality Journal*, vol. 23, no. 2, pp. 229–265, apr 2014. [Online]. Available: http://link.springer.com/10.1007/s11219-014-9235-5