Noname manuscript No.
(will be inserted by the editor)

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

1 Introduction

Large software systems are becoming increasingly important in the daily lives of many people. A large portion of the cost of these software systems is attributed to their maintenance. In fact, previous studies show that more than 90% of the software development cost is spent on maintenance and evolution activities (Erlikh 2000). A plethora of previous research was dedicated to addressing issues related to software bugs. For example, there have been several studies (e.g., (Weiß, Zimmermann, and Zeller 2007; Zhang, Gong, and Versteeg 2013)) that study 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., (Chen 2013; S. Kim et al. 2007a; Nayrolles et al. 2015)).

All these studies, however, treat bug as equal in a sense that they do not assume any underlying classification of bugs. From practice and experience, we know that bug are indeed different and, in this study, we classify different bugs

Mathieu Nayrolles · Abdelwahab Hamou-Lhadj ·
SBA Lab, ECE Dept, Concordia University
Montréal, QC, Canada
E-mail: {mathieu.nayrolles,wahab.hamou-lhadj}@concordia.ca

Emad Shihab

Data-driven Analysis of Software (DAS) Research Lab
Montréal, QC, Canada
E-mail: emad.shihab@concordia.ca

Alf Larsson

Sigrid Eldh
Ericsson
Stockolm, Sweden
E-mail: {alf.larsson, sigrid.eldh}@ericsson.com

based on the location of their fixes. By a fix, we mean a modification (adding or deleting lines of code) to an existing file that is used to solve the bug. With this in mind, the relationship between bugs and fixes can be modelled using the UML diagram in Figure 1. The diagram only includes bug reports that are fixed and not, for example, duplicate reports. 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).

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

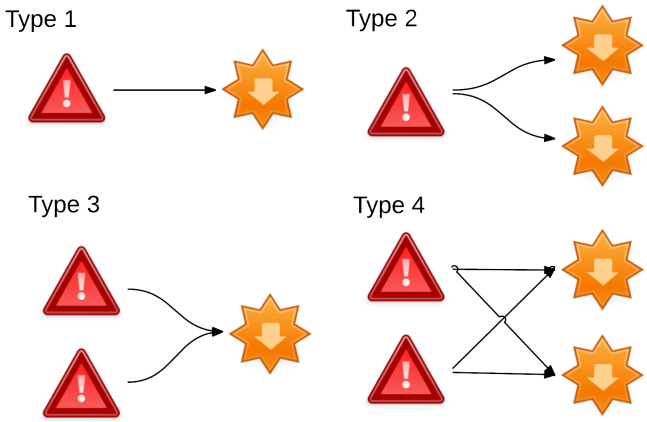

Fig. 2 Proposed Taxonomy of Bugs

The first and second types are the ones we intuitively know about. T1 refers to a bug being fixed in one single location (i.e., one file), while T2 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. T3 refers to multiple bugs that are fixed in the same location. T4 is an extension of T3, where multiple bugs are resolved by modifying the same set of locations. Note that T3 and T4 bugs are not duplicates, they may occur when different features of the system fail due to the same root causes (faults).

In our dataset, composed of 388 projects and presented in section 3.1, the proportion of each type of bug is as follows: T1 6.8 %, T2 3.7 %, T3 28.3 % and T4 61.2 %. Also, classical measures of complexity such as duplication, fixing time, number of comments, number of time a bug is reopened, files impacted, severity, changesets, hunks, and chunks, also presented in section 3.1, show that type 4 are significantly more complex than types 1, 2 and 3.

The existence of a high number of T4 bugs and the fact that they are more complex call for techniques that can effectively tag bug report as T4 at submission

time for enhanced triaging. More particularly, we are interested in the following research questions:

- **RQ1:** *Are T4 bug predictable at submission time?* In this research question, we investigate if and how to predict the type of a bug report at submission time. Being able to build accurate classifiers predicting the bug type at submission time will allow improving the triaging and the bug handling process.
- **RQ2:** *What are the best predictors of type 4 bugs ?* This second research question aims to investigate what are the markup that allows for accurate prediction of type 4 bugs.

Our objective is to propose a classification that can allow researchers in the field of mining bug repositories 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 similarly as the clone taxonomy presented by Kapser and Godfrey (Kapser and Godfrey 2003). 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 compare approaches with each other effectively. Moreover, we build upon this proposed classification and predict the type of incoming bugs with a 65.40% precision 94.16% recall for f_1 measure of 77.19%.

2 Preliminaries

In this section, we present some preliminaries about version control systems and project tracking systems.

2.1 Version control systems

Version control consists of maintaining the versions of files — such as source code and other software artefacts (Zeller 1997). Version control tools have been created to help practitioners manage the version of their software artefacts. Each evolution of software is a version (or revision), and each version (revision) is linked to the one before through modifications of software artefacts. These modifications consist of updating, adding or deleting software artefacts. 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.

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

In modern version control systems, modifications to the source code is done through the commit mechanism. The commit operation will version the modifications applied to one or many files.

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 same 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.

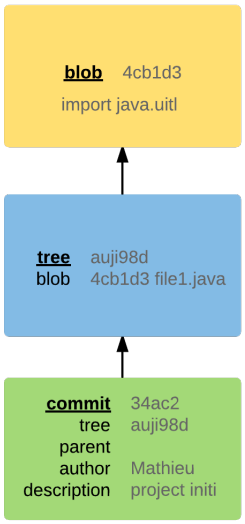

Fig. 3 Data structure of a commit.

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 behaviour. These systems are also used by development teams to keep track of the modifications induced by a bug and bug reports and keep track of the fixes.

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* (Koponen 2006).

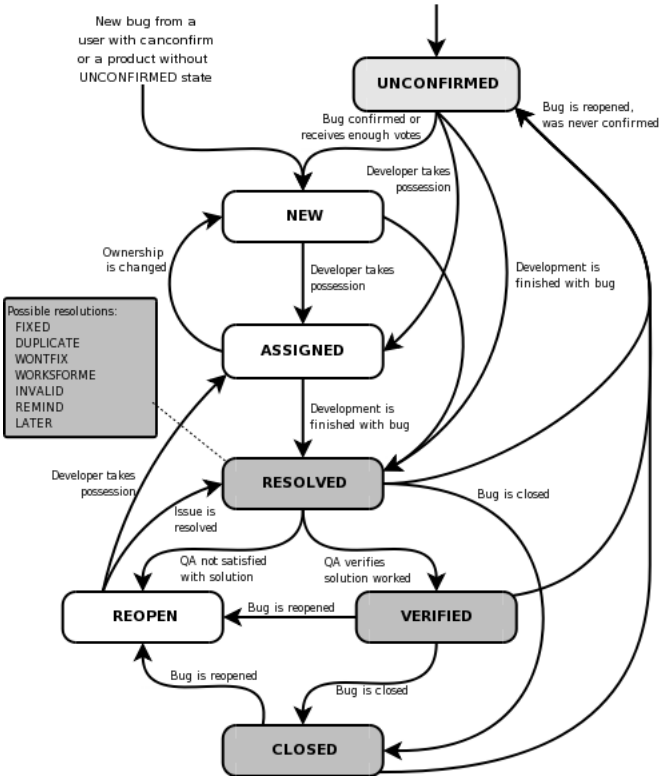

Fig. 4 Lifecyle of a report [@Bugzilla2008]

- 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 a 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 the 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, etc...) will be able to merge the branch with the main line. 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 except the *UNCONFIRMED* and *RESOLVED* states. Tasks are created by management and do not need to be confirmed 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 (Bettenburg et al. 2008). 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 misspelt words or misaligned text.

The relationship between a report or a task and the actual modification can be hard to establish, and it has been a subject of various research studies (e.g., (Antoniol et al. 2002; Bachmann et al. 2010; Wu et al. 2011)). 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, it has been shown that more than half of the submitted reports are not linked to their corresponding source code changes (Wu et al. 2011).

3 Dataset

In this section, we present our datasets in length. In this presentation, we explore the proportion of each type of bug as well as the complexity of each type.

3.1 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 resolved-fixed reports, (ii) the number of commits, (iii) the overall number of files, and (iv) the number of projects analysed.

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 analysed projects are hosted in *Git* or *Mercurial* repositories and have either a *Jira* or a *Bugzilla* issue tracker associated with them. The Apache ecosystem

consists of 349 projects written in various programming languages (C, C++, Java, Python, ...) and uses *Git* with *Jira*. These projects represent the Apache ecosystem in its entirety. We did not exclude any system from our study. The complete list can be found online[2]. The Netbeans ecosystem consists of 39 projects, mostly written in Java. Similar to the Apache ecosystem, we selected all the projects belonging to the Netbeans ecosystem. The Netbeans community uses *Bugzilla* with *Mercurial*.

The choice of these two ecosystems is driven by the motivation to consider projects having (i) different sizes, (ii) different architectures, and (iii) different development bases and processes. Apache projects vary significantly in terms of the size of the development team, purpose and technical choices (Bavota et al. 2013). On the other side, Netbeans has a relatively stable list of core developers and a common vision shared by the 39 related projects (Wang, Baik, and Devanbu 2011).

Cumulatively, these datasets span from 2001 to 2014. In summary, our consolidated dataset contains 102,707 closed issues, 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 bugs, identified 3,284 sub-projects, and 17,984 unique contributors to these bug reports and source code version management systems. The cumulated opening time for all the bugs reaches 10,661 working years (3,891,618 working days). The raw size of the cloned source code alone, excluding binaries, images, and other non-text file, is 163 GB.

To assign commits to issues, we used the regular expression based approach proposed by Fischer et al. (Fischer, Pinzger, and Gall, n.d.), which matches the issue ID in the commit note to the commit. 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. Note that an issue can be fixed using several commits.

3.2 Dataset Analysis

Using our dataset, we extracted the files f_i impacted by each commit c_i for each one of our 388 projects. Then, we classified the bugs according to each type, which we formulate as follows:

- **Type 1:** A bug is tagged T1 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 T2 if it is fixed by modifying by n files, $f_{i..n}$, where n > 1, and the files $f_{i..n}$ are not involved in any other bug fix.
- **Type 3:** A bug is tagged T3 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 T4 if it is fixed by modifying several files $f_{i..n}$ and the files $f_{i..n}$ are involved in any other bug fix.

Table 2 presents the contingency table and the results of the Pearson's chi-squared tests we performed on each type of bug. We can see that the proportion of T4 (61.2%) largely higher than that of T1 (6.8%), 2 (3.7%) and 3 (28.3%) and that the difference is significant according to the Pearson's chi-squared test.

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

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 %)	

Pearson's chi-squared independence test is used to analyse the relationship between two qualitative data, in our study the type bugs and the studied ecosystem. The results of Pearson's chi-square independence tests are considered statistically significant at $= 0.05$. If p-value ≤ 0.05, we can conclude that the proportion of each type is significantly different.

we analyse the complexity of each bug regarding duplication, fixing time, number of comments, number of time a bug is reopened, files impacted, severity, changesets, hunks, and chunks.

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 (Weiß, Zimmermann, and Zeller 2007; Saha, Khurshid, and Perry 2014; Nam, Pan, and Kim 2013; Anvik, Hiew, and Murphy 2006; Nagappan and Ball 2005).

Tables 3, 4 and 5 present descriptive statistics about each metric for each bug type per ecosystem and for both ecosystems combined. The descriptive statistics used are μ:mean, $\sum$:sum, $\hat{x}$:median, σ:standard deviation and %:percentage. We also show the results 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.

3.2.1 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 be 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 (Sun et al. 2011; Bettenburg, Premraj, and Zimmermann 2008; Nguyen et al. 2012; Jalbert and Weimer 2008; Tian, Sun, and Lo 2012; Runeson, Alexandersson, and Nyholm 2007) and a well-known measure for bug impact.

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

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
	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)
T1	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)
	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)
T2	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)
	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)
T3	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)
	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
T4	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

3.2.2 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 reopened, 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 (Weiss et al. 2007; Zhang et al. 2012; Zhang, Gong, and Versteeg 2013).

When combined, both ecosystem amounts in the following order $T2^4_{time} > T4^1_{time} \gg T1^3_{time} \gg T3^2_{time}$. These findings contradicts the finding of Saha *et al.*, however, they did not study the Netbeans ecosystem in their paper (Saha, Khurshid, and Perry 2014).

3.2.3 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 (Zhang, Gong, and Versteeg 2013;

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

Zhang et al. 2012). It is also used in bug prediction approaches (D'Ambros, Lanza, and Robbes 2010; Bhattacharya and Neamtiu 2011).

When combining both ecosystems, the results are: $T4^1_{comment} \gg T2^4_{comment} > T3^2_{comment} \gg T1^3_{comment}$.

3.2.4 Bug Reopening

The bug is 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 (Zimmermann et al. 2012; Shihab et al. 2010; Lo 2013).

When combined, however, the order does change: $T4^1_{reop} > T2^4_{reop} > T1^3_{reop} \gg T3^2_{reop}$.

3.2.5 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 (Menzies and Marcus 2008; Guo2010; Lamkanfi et al. 2010; Tian, Lo, and Sun 2012; Valdivia Garcia and Shihab 2014; Havelund, Holzmann, and Joshi 2015) and it helps to

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
	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)
T1	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)
	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)
T2	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)
	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)
T3	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)
	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
T4	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

prioritization of fixes (Xuan et al. 2012). 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.

3.2.6 Files impacted

The number of files impacted measures how many files have been modified for the bug report to be closed.

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

3.2.7 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).

Overall, T4 bugs are the most complex bugs regarding the number of submitted changesets ($T4^1_{changesets} \gg T2^3_{changesets} \gg T3^2_{changesets} \gg T1^4_{changesets}$).

While results have been published on the bug-fix patterns (Pan, Kim, and Whitehead 2008), smell introduction (Tufano et al. 2015; Eyolfson, Tan, and Lam 2011), to the best of our knowledge, no one interested themselves in how many iterations of a patch was required to close a bug report beside us.

3.2.8 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 (Kim et al. 2006; Jung, Oh, and Yi 2009; Rosen, Grawi, and Shihab 2015) and bug-fix comprehension (Pan, Kim, and Whitehead 2008). 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 T2 and T4 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 $T4^1_{hunks} \gg T2^2_{hunks} \gg T1^4_{hunks} \gg T3^3_{hunks}$.

3.2.9 Churns

The last metric, 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 (Kim et al. 2006; Pan, Kim, and Whitehead 2008; Jung, Oh, and Yi 2009; Rosen, Grawi, and Shihab 2015).

Once again, T4 and T2 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 : $T4^1_{churns} \gg T2^2_{churns} \gg T1^4_{churns} \gg T3^3_{churns}$.

To determine which type is the most complex, we counted how many times each bug type obtained each position in our nine rankings and 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 T1, T2, T3 and T4 bugs, respectively.

Considering that type 4 bugs are (a) the most common, (b) the most complex and (c) not a type we intuitively know about; we decided to kick start our research into the different type of bugs and their impact by predicting wether an incoming bug report type 4 or not.

4 Experimentations

In this section, we present the results of our experiences and interpret them to answer our two research questions.

4.1 Are T4 bug predictable at submission time?

To answer this question, we used as features the words in the bug description contained in a bug report. We removed he stopwords (i.e. the, or, she, he) and truncated the remaining words to their roots (i.e. writing becomes write, failure becomes fail and so on). We experimented with 1-gram, 2-gram, and 3-gram words weighted using tf-idf. To build the classifier, we examined three machine learning techniques that have shown to yield satisfactory results in related studies: SVM, Random forest and linear regression (Weiß, Zimmermann, and Zeller 2007; Alencar, Abebe, and Mcintosh 2014; Nam, Pan, and Kim 2013).

To answer $\mathbf{RQ}_1$, we analyse the accuracy of predictors aiming at determining the type of a bug at submission time (i.e. when the bug report is submitted by someone).

Tables 6, 7 and 8 presents the results obtained while building classifiers for the most complex type of bug. According to the complexity analysis conducted in section 3.2, the most complex type of bug, in terms of duplicate, time to fix, comments, reopening, files changed, severity, changesets, churns, and hunks, is T4.

To answer our research question, we built nine different classifiers using three different machine learning techniques: Linear regression, support vector machines and random forest for ten different projects (5 from each ecosystem).

We selected the top 5 projects of each ecosystem with regard to their bug report count (Ambari, Cassandra, Flume, HBase and Hive for Apache; Cnd, Editor, Java, JavaEE and Platform for Netbeans). For each machine learning techniques, we built classifiers using the text contained in the bug report and the comment of the first 48 hours as they are likely to provide additional insights on the bug itself. We eliminate the stop-words of the text and trim the words to their semantical roots using wordnet. We experimented with 1-gram, 2-gram, and 3-gram words, weighted using tf/idf.

The feature vectors are fed to the different machine learning techniques in order to build a classifier. The data is separated into two parts with a 60%-40% ratio. The 60% part is used for training purposes while the 40% is used for testing purposes. During the training process we use the ten-folds technique iteratively and, for each iteration, we change the parameters used by the classifier building process (cost, mtry, etc). At the end of the iterations, we select the best classifier and exercise it against the second part of 40%. The results we report in this section are the performances of the nine classifiers trained on 60% of the data and classifying the remaining 40%. The performances of each classifier are examined in terms of

true positive, true negative, false negative and false positive classifications. True positives and negative numbers refer to the cases where the classifier correctly classify a report. The false negative represents the number of reports that are classified as non-T4 while they are and false positive represents the number of reports classified as T4 while they are not. These numbers allow us to derive three common metrics: precision, recall and f_1 measure.

$$precision = \frac{TP + FN \cap TP + FP}{TP + FP} \tag{1}$$

$$recall = \frac{TP + FN \cap TP + FP}{TP + FN} \tag{2}$$

$$f_1 = \frac{2TP}{2TP + FP + FN} \tag{3}$$

The performances of each classifier are compared to a tenth classifier. This last classifier is a random classifier that randomly predicts the type of a bug. As we are in a two classes system (T4 and non-T4), 50% of the reports are classified as T4 by the random classifier. The performances of the random classifier itself are presented in table 9.

Finally, we compute the Cohen's Kappa metric (Fleiss and Cohen 1973) for each classifier. The Kappa metric compares the observed accuracy and the expected accuracy to provide a less misleading assessment of the classifier performance than precision alone.

$$kappa = \frac{(observedaccuracy - expectedaccuracy)}{1 - expectedaccuracy} \tag{4}$$

The observed accuracy represents the number of items that were correctly classified, according to the ground truth, by our classifier. The expected accuracy represents the accuracy obtained by a random classifier.

For the first three classifiers (SVM, linear regression and random forest with a 1-gram grouping of stemmed words) the best classifier the random forest one with 77.63% F_1 measure. It is followed by SVM (77.19%) and, finally, linear regression (76.31%). Regardless of the technique used to classify the report, there is no significant difference between ecosystems. Indeed, the p-values obtained with chi-square tests are above 0.05, and a p-value below 0.05 is a marker of statistical significance. While random forest emerges as the most accurate classifier, the difference between the three classifiers is not significant (p-value = 0.99).

For the second three classifiers (SVM, linear regression and random forest with 2-grams grouping of stemmed words) the best classifier is once again random forest with 77.34% F_1 measure. It is followed by SVM (76.91%) and, finally, linear regression (76.25%). As for the first three classifiers, the difference between the classifiers and the ecosystems are not significant. Moreover, the difference in performances between 1 and 2 grams are not significant either.

Finally, the last three classifiers (SVM, linear regression and random forest with 3-grams grouping of stemmed words) the best classifier is once again random forest with 77.12% F1-measure. It is followed by SVM (76.72%) and, finally, linear regression (75.89%). Again, the difference between the classifiers and the ecosystems are not significant. Neither are the difference in results between 1, 2 and 3 grams.

Table 6 Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 1 gram. TP: True positive, TN: True Negative, FN: False Negative, FP: False Positive

Project	Reports	T4 Reports	TP	TN	FN	FP	Precision	Recall	F1
Support Vector Machine									
Ambari	829	540	539	4	1	285	65.41%	99.81%	79.03%
Cassandra	340	199	193	5	6	136	58.66%	96.98%	73.11%
Flume	133	80	79	9	1	44	64.23%	98.75%	77.83%
HBase	357	215	213	4	2	138	60.68%	99.07%	75.27%
Hive	272	191	191	0	0	81	70.22%	100.00%	82.51%
Cnd	1105	805	753	25	52	275	73.25%	93.54%	82.16%
Editor	666	478	455	16	23	172	72.57%	95.19%	82.35%
Java	1090	693	676	37	17	360	65.25%	97.55%	78.20%
JavaEE	585	287	258	52	29	246	51.19%	89.90%	65.23%
Platform	969	573	467	110	106	286	62.02%	81.50%	70.44%
Total	6346	4061	3824	262	237	2023	65.40%	94.16%	77.19%
Linear Regression									
Ambari	829	540	514	14	26	275	65.15%	95.19%	77.35%
Cassandra	340	199	194	5	5	136	58.79%	97.49%	73.35%
Flume	133	80	60	17	20	36	62.50%	75.00%	68.18%
HBase	357	215	212	5	3	137	60.74%	98.60%	75.18%
Hive	272	191	103	40	88	41	71.53%	53.93%	61.49%
Cnd	1105	805	762	26	43	274	73.55%	94.66%	82.78%
Editor	666	478	459	16	19	172	72.74%	96.03%	82.78%
Java	1090	693	683	13	10	384	64.01%	98.56%	77.61%
JavaEE	575	287	271	30	16	258	51.23%	94.43%	66.42%
Platform	969	573	486	102	87	294	62.31%	84.82%	71.84%
Total	6336	4061	3744	268	317	2007	65.10%	92.19%	76.31%
Random Forest									
Ambari	829	540	514	13	26	276	65.06%	95.19%	77.29%
Cassandra	337	199	191	12	8	126	60.25%	95.98%	74.03%
Flume	133	80	76	8	4	45	62.81%	95.00%	75.62%
HBase	357	215	212	9	3	133	61.45%	98.60%	75.71%
Hive	272	191	190	3	1	78	70.90%	99.48%	82.79%
Cnd	1105	805	803	4	2	296	73.07%	99.75%	84.35%
Editor	666	478	476	3	2	185	72.01%	99.58%	83.58%
Java	1090	693	682	26	11	371	64.77%	98.41%	78.12%
JavaEE	575	287	252	59	35	229	52.39%	87.80%	65.63%
Platform	969	573	437	154	136	242	64.36%	76.27%	69.81%
Total	6333	4061	3833	291	228	1981	65.93%	94.39%	77.63%

Each one of our nine classifiers improves upon the random one on all projects and by a large margin ranging from 20.73% to 22.48% regarding F-Measure.

The last measure of performance for our classifier is the computation of the Cohen's Kappa metric presented in table 10.

The table presents the results of the Cohen's kappa metric for each of our nine classifiers. The metric is computed using the observed accuracy and the expected accuracy. The observed accuracy, in our bi-class system (i.e. T4 or not), is the number of correctly classified type 4 added to the number of correctly classified non-T4 bugs over the total of reports. The expected accuracy follows the same principle but using the classification from the random classifier. The expected accuracy is constant as the random classifier predicts 50% of the reports as T4 and 50% as non-T4. Finally, the obtained Cohen's Kappa measures range from 0.27 to 0.32. While there is no unified way to interpret the result of the Cohen's kappa statistic, Landis and Koch considers 0-0.20 as slight, 0.21-0.40 as fair, 0.41-0.60 as moderate, 0.61-0.80 as substantial, and 0.81-1 as almost perfect (Landis and Koch 1977). Consequently, all of our classifiers show a fair improvement over

Table 7 Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 2 grams. TP: True positive, TN: True Negative, FN: False Negative, FP: False Positive

Project	Reports	T4 Reports	TP	TN	FN	FP	Precision	Recall	F1
Support Vector Machine									
Ambari	829	540	525	12	15	277	65.46%	97.22%	78.24%
Cassandra	323	199	189	11	10	113	62.58%	94.97%	75.45%
Flume	133	80	74	15	6	38	66.07%	92.50%	77.08%
HBase	357	215	205	23	10	119	63.27%	95.35%	76.07%
Hive	272	191	171	15	20	66	72.15%	89.53%	79.91%
Cnd	1105	805	731	34	74	266	73.32%	90.81%	81.13%
Editor	666	478	455	30	23	158	74.23%	95.19%	83.41%
Java	1090	693	664	58	29	339	66.20%	95.82%	78.30%
JavaEE	575	287	238	69	49	219	52.08%	82.93%	63.98%
Platform	969	573	461	110	112	286	61.71%	80.45%	69.85%
Total	6319	4061	3713	377	348	1881	66.37%	91.43%	76.91%
Linear Regression									
Ambari	829	540	510	19	30	270	65.38%	94.44%	77.27%
Cassandra	340	199	140	55	59	86	61.95%	70.35%	65.88%
Flume	142	89	59	23	30	30	66.29%	66.29%	66.29%
HBase	357	215	90	100	125	42	68.18%	41.86%	51.87%
Hive	272	191	176	8	15	73	70.68%	92.15%	80.00%
Cnd	1105	805	745	26	60	274	73.11%	92.55%	81.69%
Editor	666	478	453	27	25	161	73.78%	94.77%	82.97%
Java	1090	693	606	106	87	291	67.56%	87.45%	76.23%
JavaEE	575	287	245	70	42	218	52.92%	85.37%	65.33%
Platform	815	573	449	121	124	121	78.77%	78.36%	78.57%
Total	6191	4070	3473	555	597	1566	68.92%	85.33%	76.25%
Random Forest									
Ambari	829	540	511	20	29	269	65.51%	94.63%	77.42%
Cassandra	340	199	176	22	23	119	59.66%	88.44%	71.26%
Flume	133	80	72	21	8	32	69.23%	90.00%	78.26%
HBase	351	215	208	12	7	124	62.65%	96.74%	76.05%
Hive	272	191	190	0	1	81	70.11%	99.48%	82.25%
Cnd	1105	805	794	9	11	291	73.18%	98.63%	84.02%
Editor	666	478	471	6	7	182	72.13%	98.54%	83.29%
Java	1099	702	673	43	29	354	65.53%	95.87%	77.85%
JavaEE	575	287	238	86	49	202	54.09%	82.93%	65.47%
Platform	1002	606	444	163	162	233	65.58%	73.27%	69.21%
Total	6372	4103	3777	382	326	1887	66.68%	92.05%	77.34%

a random classification regarding accuracy and a major improvement regarding F1-measure.

4.2 What are the best predictors of type 4 bugs?

In this section, we answer our second research question: *What are the best predictors of type 4 bugs*. To do so, we extracted the best predictor of type 4 bugs for each one of the extracted grams (1, 2 and 3) for each of our ten test projects (Five Apache, Five Netbeans). Then, we manually investigated the source code and the reports of these ten software projects to determine why a given word is a good predictor of type 4 bug. In the remaining of this section, we present our findings by project and then provide a conclusion on the best predictors of type 4 bugs.

Table 8 Support Vector Machine, Linear Regression and Random Forest based classifiers performances while using 3 grams. TP: True positive, TN: True Negative, FN: False Negative, FP: False Positive

Project	Reports	T4 Reports	TP	TN	FN	FP	Precision	Recall	F1
Support Vector Machine									
Ambari	829	540	520	15	20	274	65.49%	96.30%	77.96%
Cassandra	340	199	193	11	6	130	59.75%	96.98%	73.95%
Flume	133	80	74	8	6	45	62.18%	92.50%	74.37%
HBase	357	215	208	24	7	118	63.80%	96.74%	76.89%
Hive	272	191	175	14	16	67	72.31%	91.62%	80.83%
Cnd	1105	805	725	34	80	266	73.16%	90.06%	80.73%
Editor	666	478	454	22	24	166	73.23%	94.98%	82.70%
Java	1090	693	662	61	31	336	66.33%	95.53%	78.30%
JavaEE	575	287	256	45	31	243	51.30%	89.20%	65.14%
Platform	969	573	461	111	112	285	61.80%	80.45%	69.90%
Total	6336	4061	3728	345	333	1930	65.89%	91.80%	76.72%
Linear Regression									
Ambari	829	540	505	26	35	263	65.76%	93.52%	77.22%
Cassandra	340	199	176	21	23	120	59.46%	88.44%	71.11%
Flume	133	80	68	18	12	35	66.02%	85.00%	74.32%
HBase	357	215	91	99	124	43	67.91%	42.33%	52.15%
Hive	272	191	185	5	6	76	70.88%	96.86%	81.86%
Cnd	1105	805	747	22	58	278	72.88%	92.80%	81.64%
Editor	666	478	448	31	30	157	74.05%	93.72%	82.73%
Java	1090	693	667	55	26	342	66.11%	96.25%	78.38%
JavaEE	575	287	256	51	31	237	51.93%	89.20%	65.64%
Platform	969	573	468	102	105	294	61.42%	81.68%	70.11%
Total	6336	4061	3611	430	450	1845	66.18%	88.92%	75.89%
Random Forest									
Ambari	829	540	500	22	40	267	65.19%	92.59%	76.51%
Cassandra	340	199	188	14	11	127	59.68%	94.47%	73.15%
Flume	133	80	70	23	10	30	70.00%	87.50%	77.78%
HBase	357	215	206	24	9	118	63.58%	95.81%	76.44%
Hive	272	191	189	1	2	80	70.26%	98.95%	82.17%
Cnd	1105	805	755	27	50	273	73.44%	93.79%	82.38%
Editor	666	478	453	32	25	156	74.38%	94.77%	83.35%
Java	1090	693	665	77	28	320	67.51%	95.96%	79.26%
JavaEE	575	287	241	73	46	215	52.85%	83.97%	64.87%
Platform	969	573	443	132	130	264	62.66%	77.31%	69.22%
Total	6336	4061	3710	425	351	1850	66.73%	91.36%	77.12%

Table 9 Random classifier. TP: True positive, TN: True Negative, FN: False Negative, FP: False Positive

Project	Reports	T4 Reports	TP	TN	FN	FP	Precision	Recall	F1
Ambari	828	540	249	158	291	131	65.53%	46.11%	54.13%
Cassandra	339	199	111	68	88	73	60.33%	55.78%	57.96%
Flume	132	80	32	31	48	22	59.26%	40.00%	47.76%
HBase	356	215	105	68	110	74	58.66%	48.84%	53.30%
Hive	271	191	85	40	106	41	67.46%	44.50%	53.63%
Cnd	1104	805	393	159	412	141	73.60%	48.82%	58.70%
Editor	665	478	230	94	248	94	70.99%	48.12%	57.36%
Java	1089	693	365	205	328	192	65.53%	52.67%	58.40%
JavaEE	574	287	122	148	165	140	46.56%	42.51%	44.44%
Platform	968	573	277	194	296	202	57.83%	48.34%	52.66%
Total	6335	4061	1969	1165	2092	1110	63.95%	48.49%	55.15%

Table 10 Cohen's Kappa for each classifier

Type	Gram	TP T4	TN T4	Observed Accuracy	Expected Accuracy	Kappa	Interpretation
SVM	1	3824	262	0.64	0.49	0.30	Fair
	2	3713	377	0.64	0.49	0.30	Fair
	3	3728	345	0.64	0.49	0.29	Fair
Linear Regression	1	3744	268	0.63	0.49	0.27	Fair
	2	3473	555	0.63	0.49	0.28	Fair
	3	3611	430	0.64	0.49	0.28	Fair
Random Forest	1	3833	291	0.65	0.49	0.31	Fair
	2	3777	382	0.66	0.49	0.32	Fair
	3	3710	425	0.65	0.49	0.31	Fair

4.2.1 Ambari

Ambari is aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop clusters. One of the most acclaimed features of Ambari is the ability to visualise clusters' health, according to user-defined metric, with heat maps. These heat maps give a quick overview of the system.

Figure 5 shows a screenshot of such a heat map.

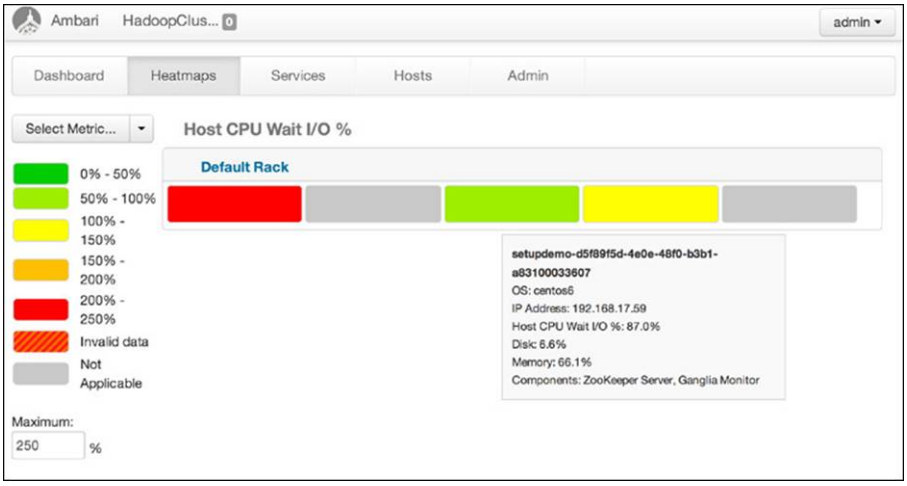

Fig. 5 Ambari heatmap

At every tested gram (i.e. 1, 2 and 3) the word "heat map" is a strong predictor of type 4 bugs. The heat map feature is a complex feature as it heavily relies on the underlying instrumentation of Hadoop and the consumption of many log format too, for example, extracts the remaining free space on a disk or the current load on a CPU.

Another word that is a strong predictor of type 4 bug is "nagio". Nagio is a log monitoring server belonging to the Apache constellation. It is used as an optional add-on for Ambari and, as for the heat map, is very susceptible to log format change and API breakage.

Versions of the "nagio" and "heatmap" keywords include: "heatmap displai", "ambari heatmap", "fix nagio", "nagio test", "ambari heatmap displai", "fix nagio test".

4.2.2 Cassandra

Cassandra is a database with high scalability and high availability without compromising performance. While extracting the unique word combinations from the report of Cassandra, one word which is a strong predictor of type 4 bug is "snapshot".

As described in the documentation, in Cassandra terms, *a snapshot first flushes all in-memory writes to disk, then makes a hard link of the SSTable files for each keyspace. You must have enough free disk space on the node to accommodate making snapshots of your data files. A single snapshot requires little disk space. However, snapshots can cause your disk usage to grow more quickly over time because a snapshot prevents old obsolete data files from being deleted. After the snapshot is complete, you can move the backup files to another location if needed, or you can leave them in place.*

The definition gives the reader an insight into how complex this feature used regarding integration with the host system and how coupled it is to the Cassandra, data model.

Other versions of the "snapshot" keyword include "snapshot sequenti", "make snapshot", "snapshot sequenti repair", "make snapshot sequenti".

4.2.3 Flume

Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data.

One word which is a good predictor of type 4 in flume is "upgrad" and the 2-grams (upgrad flume) and the 3-grams ("upgrad flume to") versions. Once again for the Apache dataset, a change in the software that induce a change in the underlying data model or data store, which is often the case when you upgrade flume to a new version, is a good indicator of the report complexity and the impact of said report on the sourcecode in terms of number of locations fixed.

On the reports manually analysed, Flume's developers and users have a hard time upgrading to new versions in a sense that logs and dashboard get corrupted or disappear post-upgrade. Significant efforts are then made to prevent such losses in the subsequent version.

4.2.4 HBase

HBase is a Hadoop database, a distributed, scalable, big data store provided by Apache. The best predictor of type 4 bug in HBase is "bloom" as in "bloom filters". Bloom filters are a probabilistic data structure that is used to test whether an element is a member of a set (Broder and Mitzenmacher 2004). Such a feature is hard to implement and hard to test because of its probabilistic nature. Much feature commits (i.e. commit intended to add a feature) and fix commits (i.e. commit intended to fix a bug) belonging to the HBase source code are related to the bloom filters. Given the nature of the feature, it is not surprising to find the word

"bloom" and its 2-, 3-grams counterparts ("on Bloom", "Bloom filter", "on Bloom filter") as a good predictor of type 4 bug.

4.2.5 Hive

Hive is a data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage using SQL. Hive is different from its Apache counterpart as the words that are the best predictors of type 4 bugs do not translate into a particular feature of the product but are directly the name of the incriminated part of the system: thrift. Thrift is a software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages. While Thrift is supposed to solve many compatibility issues when building clients for a product such as Hive, it is the cause of many major problems in Hive. The top predictors for type 4 bugs in Hive are "thrifthttpcliservic" and "thriftbinarycliservic".

As Hive, and its client, are built on top of Thrift it makes sense that issues propagating from Thrift induce major refactoring and fixed across the whole Hive source code.

4.2.6 Cnd

The CND projects is a part of the Netbeans IDE and provide support for C/C++. The top two predictors of type 4 bugs are (1) parallelism and (2) observability of c/c++ code. In each gram, we can find reference to the parallel code being problematic while developed and executed via the Netbeans IDE: "parallel comput", "parallel", "parallel comput advis". The other word, related to the observability of c/c++ code inside the Netbeans IDE is "Gizmo". "Gizmo" is the codename for the C/C++ Observability Tool built on top of D-Light Toolkit. We can find occurrences of "Gizmo" in each gram : "gizmo" and "gizmo monitor" for example.

Once again, a complex cross-concern feature with a high impact on the end-user (i.e., the ability to code, execute and debug parallel code inside Netbeans) is the cause of most of the type 4 bugs and mention of said feature in the report is a bug predictor of types of the bug.

4.2.7 Editor

The Editor component of Netbeans is the component which is handling all the textual edition, regardless of the programming language, in Netbeans. For this component, the type 4 bugs are most likely related to the "trailing white spaces" and "spellcheck" features.

While these features do not, at first sight, be as complex as, for example, parallelism debugging, they have been the cause of the majority of type 4 bugs. Upon manual inspection of the related code[3] in the Editor component of Netbeans the complexity of these feature becomes evident. Indeed, theses features behave differently for almost each type of text-file and textboxes inside Netbeans. For

[3] https://netbeans.org/projects/editor/

example, the end-user expects the spellchecking feature of the IDE to kick in while typing a comment inside a code file but not on the code itself. A similar example can be described for the identification and removing of trailing white spaces where users wish the trailing white spaces to be deleted in c/c++ code but not, for example, while typing HTML or a commit message.

Each new language supported or add-on supported by the Netbeans IDE and leveraging the features of the Editor component is susceptible to be the cause of a major refactoring to have a coherent behaviour regarding "trailing white spaces" and "spell checking".

4.2.8 Java

The Java component of Netbeans is responsible for the Java support of Netbeans in the same fashion as CND is responsible for c/c++ support. For this particular component, the set of features that are a good predictor of type 4 are the ones related to the Java autocompletion and navigation optimisation. The autocompletion has to be able to provide suggestions in a near-instantaneous manner if it is to be useful to the developer. To provide near-instantaneous suggestion on modest machines and despite the depth of the Java API, Netbeans developers opted of a statistical autocompletion. The autocompletion *remembers* which of its suggestions you used before and only provide the ones you are the most likely to want to be based on your previous usage. Also, each suggestion is companioned with a percentage which describes the number of time you pick a given a suggestion over the other. One can envision a such a system can be tricky to implement on new API being added in the Java language at each upgrade. Indeed, when a new API comes to light following a Java upgrade on the developer's machine, then, the autocompletion has to make these new API appears in the autocompletion despite their 0% chosen rate. The 0% being linked to the fact that this suggestion was not available thus far and not to the fact that the developer never picked it. When the new suggestion, related to the new API, has been ignored a given number of time, then, it can be safely removed from the list of suggestions.

Implementation of optimisations related to autocompletion and navigations are the root causes of many type 4 bugs, and we can find them in the gram extracted words that are good predictor: "implement optim", "move otim", "optim import implement", "call hierarchy implement".

4.2.9 JavaEE

The JavaEE component of Netbeans is responsible for the support of the JavaEE in Netbeans. This module is different from the CND and JAVA module in a sense that it uses and expands many functionalities from the JAVA component. For the JavaEE component, the best predictor of type 4 bugs is the hibernate and webscoket features which can be found in many gram forms: "hibern revers", "websocket endpoint", "hibern", "websocket", "implement hibern revers", "hibern revers engin".

Hibernate is an ORM that enables developers to write applications whose data outlives the application process more easily. As an Object/Relational Mapping (ORM) framework, Hibernate is concerned with data persistence as it applies to relational databases (via JDBC).

The shortcoming of Netbeans leading to most of the type 4 bugs is related to the annotation based persistence of Hibernate where developers can annotate their class attributes with the name of the column they wish the value of the attribute to be persisted. While the annotation mechanism is supported by Java, it is not possible *compile* annotation and makes sure that their statically sound. Consequently, much tooling around annotation has to be developed and maintained accordingly to new databases updates. Such tooling, for example, is responsible for querying the database model to make sure that the annotated columns exists and can store the attribute data type-wise.

4.2.10 Platform

The last netbeans component we analyzed is the one named Platform. *The NetBeans Platform is a generic framework for Swing applications. It provides the "plumbing" that, before, every developer had to write themselves—saving state, connecting actions to menu items, toolbar items and keyboard shortcuts; window management, and so on.* (https://netbeans.org/features/platform/)

The best predictor of type 4 bug in the platform component is the "filesystem" word which refers to the ability of any application built atop of Platform to use the filesystem for saves and such.

What we can conclude for this second research question is that the best predictor of type 4 bugs is the mention of a cross-concern, complex, widely used feature in the targeted system. Reports mentioning said feature are likely to create a type 4 structure with many bugs being fixed in the same set of files. One noteworthy observation is that the 2- and 3-grams extraction do not add much to the precision about the 1-gram extraction as seen the first research question. Upon the manual analysis required for this research question, we can deduct why. Indeed, the problematic features of a given system are identified with a single word (i.e. hibernate, filesystem, spellcheck, ...). While the 2- and 3-grams classifiers do not provide an additional performance in the classification process, they still become handy when trying to target which part of the feature a good predictor of type 4 ("implement optim", "gizmo monitor", "heatmap displai", ...)

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 (Perry, Dewayne E. and Stieg. 1993). 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 (Bettenburg et al. 2008), and by further simplifying existing bug reporting models (Herraiz et al. 2008).

Another field of study consist of assigning these bug reports, automatically if possible, to the right developers during triaging (Anvik, Hiew, and Murphy 2006; Jeong, Kim, and Zimmermann 2009; Tamrawi et al. 2011; Bortis and Hoek 2013). Another set of approaches focus on how long it takes to fix a bug (Zhang, Gong, and Versteeg 2013; Bhattacharya and Neamtiu 2011; Saha, Khurshid, and Perry

2014) and where it should be fixed [Zhou, Zhang, and Lo (2012); Zeller2013a]. With the rapidly increasing number of bugs, the community was also interested in prioritizing bug reports (Kim et al. 2011), and in predicting the severity of a bug(Lamkanfi et al. 2010). Finally, researchers proposed approaches to predict which bug will get reopened [Zimmermann et al. (2012); Lo2013], which bug report is a duplicate of another one (Bettenburg, Premraj, and Zimmermann 2008; Tian, Sun, and Lo 2012; Jalbert and Weimer 2008) and which locations are likely to yield new bugs (S. Kim et al. 2007b; Kim et al. 2006; Tufano et al. 2015). 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 (Eldh 2001), 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 (Eldh 2001). or safety critical(Hamill and Goseva-Popstojanova 2014). Hamill et al.(Hamill and Goseva-Popstojanova 2014) 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 (Eldh 2001) and Hamill et al. (Hamill and Goseva-Popstojanova 2014). 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 (Kapser and Godfrey 2003). 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.

8 Appendices

Lists all the top-level projects we analysed for this study.

Parsers

- Mime4j: Apache James Mime4J provides a parser, MimeStreamParser, for e-mail message streams in plain rfc822 and MIME format
- Xerces: XML parsers for c++, java and perl
- Xalan:XSLT processor for transforming XML documents into HTML, text, or other XML document types.
- FOP:Print formatter driven by XSL formatting objects (XSL-FO) and an output independent formatter.
- Droids: intelligent standalone robot framework that allows to create and extend existing droids (robots).
- Betwit: XML introspection mechanism for mapping beans to XML

Databases

- Drill: Schema-free SQL Query Engine for Hadoop, NoSQL and Cloud Storage
- Tez: Frameword for complex directed-acyclic-graph of tasks for processing data.built atop Apache Hadoop YARN.
- HBase: Apache HBase is the Hadoop database, a distributed, scalable, big data store.
- Falcon: Falcon is a feed processing and feed management system aimed at making it easier for end consumers to onboard their feed processing and feed management on hadoop clusters.
- Cassandra: Database with high scalability and high availability without compromising performance
- Hive: Data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage using SQL
- Sqoop: Tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.
- Accumulo: Sorted, distributed key/value store is a robust, scalable, high performance data storage and retrieval system.
- Lucene: Full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.
- CouchDB: Store your data with JSON documents. Access your documents and query your indexes with your web browser, via HTTP.
- Phoenix: OLTP and operational analytics in Hadoop for low latency applications
- OpenJPA: Java persistence project that can be used as a stand-alone POJO persistence layer or integrated into any Java EE
- Gora: Provides an in-memory data model and persistence for big data
- Optiq: framework that allows efficient translation of queries involving heterogeneous and federated data.
- HCatalog: Table and storage management layer for Hadoop that enables users with different data processing tools
- DdlUtils: Component for working with Database Definition (DDL) files
- Derby: Relational database implemented entirely in Java
- DBCP: Supports interaction with a relational database
- JDO: Object persistence technology

Web and Services

- Wicket: Server-side Java web framework
- Service Mix: The components project holds a set of JBI (ava Business Integration) components that can be installed in both the ServiceMix 3 and ServiceMix 4 containers.
- Shindig: Apache Shindig is an OpenSocial container and helps you to start hosting OpenSocial apps quickly by providing the code to render gadgets, proxy requests, and handle REST and RPC requests.
- Felix: Implement the OSGi Framework and Service platform and other interesting OSGi-related technologies under the Apache license.

- Trinidad: JSF framework including a large, enterprise quality component library.
- Axis: Web Services / SOAP / WSDL engine.
- Synapse: Lightweight and high-performance Enterprise Service Bus
- Giraph: Iterative graph processing system built for high scalability.
- Tapestry: A component-oriented framework for creating highly scalable web applications in Java.
- JSPWiki: WikiWiki engine, feature-rich and built around standard JEE components (Java, servlets, JSP).
- TomEE: Java EE 6 Web Profile certified application server extends Apache Tomcat.
- Knox: REST API Gateway for interacting with Apache Hadoop clusters.
- Flex: Framework for building expressive web and mobile applications
- Lucy: Search engine library provides full-text search for dynamic programming languages
- Camel: Define routing and mediation rules in a variety of domain-specific languages, including a Java-based Fluent API, Spring or Blueprint XML Configuration files, and a Scala DSL.
- Pivot: Builds installable Internet applications (IIAs)
- Celix: Implementation of the OSGi specification adapted to C
- Traffic Server: Fast, scalable and extensible HTTP/1.1 compliant caching proxy server.
- Apache Net: Implements the client side of many basic Internet protocols. The purpose of the library is to provide fundamental protocol access, not higher-level abstractions.
- Sling: Innovative web framework
- Axis: Implementation of the SOAP ("Simple Object Access Protocol") submission to W3C.
- Shale: Web application framework, fundamentally based on JavaServer Faces.
- Rave: web and social mashup engine that aggregates and serves web widgets.
- Tuscany: Simplifies the task of developing SOA solutions by providing a comprehensive infrastructure for SOA development and management that is based on Service Component Architecture (SCA) standard.
- Pluto: Implementation of the Java Portlet Specification.
- ODE: Executes business processes written following the WS-BPEL standard
- Muse: Java-based implementation of the WS-ResourceFramework (WSRF), WS-BaseNotification (WSN), and WS-DistributedManagement (WSDM) specifications.
- WS-Commons: Web Services Commons Projects
- Geronimo: Server runtime that integrates the best open source projects to create Java/OSGi
- River: Network architecture for the construction of distributed systems in the form of modular co-operating services
- Commons FileUpload: Makes it easy to add robust, high-performance, file upload capability to your servlets and web applications.
- Beehive: Java Application Framework that was designed to simplify the development of Java EE based applications.
- Aries: Java components enabling an enterprise OSGi application programming model.
- Empire Db: Relational database abstraction layer and data persistence component
- Commons Daemon: Java based daemons or services
- Click: JEE web application framework
- Stanbol: Provides a set of reusable components for semantic content management.
- CXF: Open-Source Services Framework
- Sandesha2: Axis2 module that implements the WS-ReliableMessaging specification published by IBM, Microsoft, BEA and TIBCO
- Neethi: Framework for the programmers to use WS Policy
- Rampart: Provides implementations of the WS-Sec* specifications for Apache Axis2.
- AWF: web server
- Nutch: Web crawler
- HttpAsyncClient: Designed for extension while providing robust support for the base HTTP protocol
- Portals Bridges: Portlet development using common web frameworks like Struts, JSF, PHP, Perl, Velocity and Scripts such as Groovy, JRuby, Jython, BeanShell or Rhino JavaScript.
- Stonehenge: set of example applications for Service Oriented Architecture that spans languages and platforms and demonstrates best practices and interoperability.

Cloud and Big data

- Whirr: Set of libraries for running cloud services

- Ambari: Aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop clusters.
- Karaf: Karaf provides dual polymorphic container and application bootstrapping paradigms to the Enterprise.
- Hadoop: Software for reliable, scalable, distributed computing.
- Hama: framework for Big Data analytics which uses the Bulk Synchronous Parallel (BSP) computing model.
- Twill: Abstraction over Apache Hadoop YARN that reduces the complexity of developing distributed applications
- Hadoop MapReduce and Framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.
- Tajo: Big data relational and distributed data warehouse system for Apache Hadoop
- Sentry: System for enforcing fine grained role based authorization to data and metadata stored on a Hadoop cluster.
- Oozie: Workflow scheduler system to manage Apache Hadoop jobs.
- Solr: Provides distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration
- Airavata: Software framework that enables you to compose, manage, execute, and monitor large scale applications
- JClouds: Multi-cloud toolkit for the Java platform that gives you the freedom to create applications that are portable across clouds while giving you full control to use cloud-specific features.
- Impala: Native analytic database for Apache Hadoop.
- Libcloud: Python library for interacting with many of the popular cloud service providers using a unified API.
- Slider: deploy existing distributed applications on an Apache Hadoop YARN cluster
- MRUNIT: Java library that helps developers unit test Apache Hadoop map reduce jobs.
- Stratos: Framework that helps run Apache Tomcat, PHP, and MySQL applications and can be extended to support many more environments on all major cloud infrastructures
- Mesos: Abstracts CPU, memory, storage, and other compute resources away from machines
- Helix: A cluster management framework for partitioned and replicated distributed resources
- Argus: Centralized approach to security policy definition and coordinated enforcement
- DeltaCloud: API that abstracts differences between clouds
- MRQL: Query processing and optimization system for large-scale, distributed data analysis, built on top of Apache Hadoop, Hama, Spark, and Flink.
- Provisionr: create and manage pools of virtual machines on multiple clouds
- Curator: A ZooKeeper Keeper.
- ZooKeeper: Open-source server which enables highly reliable distributed coordination
- Bigtop: Infrastructure Engineers and Data Scientists looking for comprehensive packaging, testing, and configuration of the leading open source big data components.
- Yarn: split up the functionalities of resource management and job scheduling/monitoring into separate daemons.

Messaging and Logging

- Activemq: Messaging queue
- Qpid: Messaging queue
- log4cxx: Logging framework for C++
- log4j: Logging framework for Java
- log4net: Logging framework for .Net
- Flume: Distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data.
- Samza: The project aims to provide a near-realtime, asynchronous computational framework for stream processing.
- Pig: Analyzing large data sets that consists of a high-level language for expressing data analysis programs.
- Chukwa: Data collection system for monitoring large distributed systems
- BookKeeper: Replicated log service which can be used to build replicated state machines.

- Apollo: Faster, more reliable, easier to maintain messaging broker built from the foundations of the original ActiveMQ.
- S4: Processes continuous unbounded streams of data.

Graphics

- Commons Imaging: Pure-Java Image Library
- PDFBox: Java tool for working with PDF documents.
- Batik: Java-based toolkit for applications or applets that want to use images in the Scalable Vector Graphics (SVG)
- XML Graphics Commons: consists of several reusable components used by Apache Batik and Apache FOP
- UIMA: UIMA frameworks, tools, and annotators, facilitating the analysis of unstructured content such as text, audio and video.

Dependency Management and build systems

- Tentacles: Downloads all the archives from a staging repo, unpack them and create a little report of what is there.
- Ivy: Transitive dependency manager
- Rat: Release audit tool, focused on licenses.
- Ant: drive processes described in build files as targets and extension points dependent upon each other
- EasyAnt: Improved integration in existing build systems
- IvyIDE: Eclipse plugin which integrates Apache Ivy's dependency management into Eclipse
- NPanday: Maven for .NET
- Maven: software project management and comprehension tool

Networking

- Mina:100% pure java library to support the SSH protocols on both the client and server side.
- James:Delivers a rich set of open source modules and libraries, written in Java, related to Internet mail communication which build into an advanced enterprise mail server.
- Hupa:Rich IMAP-based Webmail application written in GWT (Google Web Toolkit).
- Etch:cross-platform, language and transport-independent framework for building and consuming network services
- Commons IO: Library of utilities to assist with developing IO functionality.

File systems and repository

- Tika: detects and extracts metadata and text from over a thousand different file types
- OODT: Apache Object Oriented Data Technology (OODT) is a smart way to integrate and archive your processes, your data, and its metadata.
- Commons Virtual File System: Provides a single API for accessing various different file systems.
- Jackrabbit Oak: Scalable and performant hierarchical content repository
- Directory: Provides directory solutions entirely written in Java.
- SANDBOX: Subversion repository for Commons committers to function as an open workspace for sharing and collaboration.

Misc

- Harmony: Modular Java runtime with class libraries and associated tools.
- Mahout: Machine learning applications.
- OpenCMIS: Apache Chemistry OpenCMIS is a collection of Java libraries, frameworks and tools around the CMIS specification.
- Apache Commons: Apache project focused on all aspects of reusable Java components
- Shiro: Java security framework
- Cordova: Mobile apps with HTML, CSS & JS
- XMLBeans: Technology for accessing XML by binding it to Java types

- State Chart XML: Provides a generic state-machine based execution environment based on Harel State Tables
- excalibur: lightweight, embeddable Inversion of Control
- Commons Transaction: Transactional Java programming
- Velocity: collection of POJO
- BCEL: analyze, create, and manipulate binary Java class files
- Abdera: Functionally-complete, high-performance implementation of the IETF Atom Syndication Format
- Commons Collections: Data structures that accelerate development of most significant Java applications.
- Java Caching System: Distributed caching system written in Java
- OGNL: Object-Graph Navigation Language; it is an expression language for getting and setting properties of Java objects, plus other extras such as list projection and selection and lambda expressions.
- Anything To Triples: library that extracts structured data in RDF format from a variety of Web documents.
- Axiom: provides an XML Infoset compliant object model implementation which supports on-demand building of the object tree
- Graft: debugging and testing tool for programs written for Apache Giraph
- Hivemind: Services and configuration microkernel
- JXPath: defines a simple interpreter of an expression language called XPath

References

Alencar, Daniel, Surafel Lemma Abebe, and Shane Mcintosh. 2014. "An Empirical Study of Delays in the Integration of Addressed Issues." In *Software Maintenance and Evolution (ICSME)*.

Antoniol, G., G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. 2002. "Recovering traceability links between code and documentation." *IEEE Transactions on Software Engineering* 28 (10). IEEE Press: 970–83. doi:10.1109/TSE.2002.1041053.

Anvik, John, Lyndon Hiew, and Gail C Murphy. 2006. "Who should fix this bug?" In *Proceeding of the 28th International Conference on Software Engineering - ICSE '06*, 361. New York, New York, USA: ACM Press. doi:10.1145/1134285.1134336.

Bachmann, Adrian, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. 2010. "The missing links." In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE '10*, 97. New York, New York, USA: ACM Press. doi:10.1145/1882291.1882308.

Bavota, Gabriele, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. "The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache." In *2013 IEEE International Conference on Software Maintenance*, 280–89. IEEE. doi:10.1109/ICSM.2013.39.

Bettenburg, Nicolas, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. "What makes a good bug report?" In *Proceedings of the 16th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, 308. New York, New York, USA: ACM Press. doi:10.1145/1453101.1453146.

Bettenburg, Nicolas, Rahul Premraj, and Thomas Zimmermann. 2008. "Duplicate bug reports considered harmful ... really?" In *2008 IEEE International Conference on Software Maintenance*, 337–45. IEEE. doi:10.1109/ICSM.2008.4658082.

Bhattacharya, Pamela, and Iulian Neamtiu. 2011. "Bug-fix time prediction models: can we do better?" In *Proceeding of the 8th Working Conference on Mining Software Repositories - MSR '11*, 207. New York, New York, USA: ACM Press. doi:10.1145/1985441.1985472.

Bortis, Gerald, and Andre van der Hoek. 2013. "PorchLight: A tag-based approach to bug triaging." In *2013 35th International Conference on Software Engineering (ICSE)*, 342–51. IEEE. doi:10.1109/ICSE.2013.6606580.

Broder, Andrei, and Michael Mitzenmacher. 2004. "Network Applications of Bloom Filters: A Survey." *Internet Mathematics* 1 (4). A.K. Peters: 485–509. doi:10.1080/15427951.2004.10129096.

Chen, Ning. 2013. "Star: stack trace based automatic crash reproduction." PhD thesis, The Hong Kong University of Science; Technology.

D'Ambros, Marco, Michele Lanza, and Romain Robbes. 2010. "An extensive comparison of bug prediction approaches." In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 31–41. IEEE. doi:10.1109/MSR.2010.5463279.

Eldh, Sigrid. 2001. "On Test Design." PhD thesis, Mälardalen.

Erlikh, L. 2000. "Leveraging legacy system dollars for e-business." *IT Professional* 2 (3): 17–23. doi:10.1109/6294.846201.

Eyolfson, Jon, Lin Tan, and Patrick Lam. 2011. "Do time of day and developer experience affect commit bugginess." In *Proceeding of the 8th Working Conference on Mining Software Repositories - MSR '11*, 153. New York, New York, USA: ACM Press. doi:10.1145/1985441.1985464.

Fischer, M., M. Pinzger, and H. Gall. n.d. "Populating a Release History Database from version control and bug tracking systems." In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 23–32. IEEE Comput. Soc. doi:10.1109/ICSM.2003.1235403.

Fleiss, J. L., and J. Cohen. 1973. "The Equivalence of Weighted Kappa and the Intraclass Correlation Coefficient as Measures of Reliability." *Educational and Psychological Measurement* 33 (3). Sage PublicationsSage CA: Thousand Oaks, CA: 613–19. doi:10.1177/001316447303300309.

Hamill, Maggie, and Katerina Goseva-Popstojanova. 2014. "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system." *Software Quality Journal* 23 (2): 229–65. doi:10.1007/s11219-014-9235-5.

Havelund, Klaus, Gerard Holzmann, and Rajeev Joshi, eds. 2015. *NASA Formal Methods*. Vol. 9058. Lecture Notes in Computer Science. Cham: Springer International Publishing. doi:10.1007/978-3-319-17524-9.

Herraiz, Israel, Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. 2008. "Towards a simplification of the bug report form in eclipse." In *Proceedings of the 2008 International Workshop on Mining Software Repositories - MSR '08*, 145. New York, New York, USA: ACM Press. doi:10.1145/1370750.1370786.

Jalbert, Nicholas, and Westley Weimer. 2008. "Automated duplicate detection for bug tracking systems." In *2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN)*, 52–61. IEEE. doi:10.1109/DSN.2008.4630070.

Jeong, Gaeul, Sunghun Kim, and Thomas Zimmermann. 2009. "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*, 111. New York, New York, USA: ACM Press. doi:10.1145/1595696.1595715.

Jung, Yungbum, Hakjoo Oh, and Kwangkeun Yi. 2009. "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*, 13. New York, New York, USA: ACM Press. doi:10.1145/1651309.1651313.

Kapser, Cory, and Michael W Godfrey. 2003. "Toward a Taxonomy of Clones in Source Code: A Case Study." In *International Workshop on Evolution of Large Scale Industrial Software Architectures*, 67–78. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.6056`.

Kim, Dongsun, Xinming Wang, Student Member, Sunghun Kim, Andreas Zeller, S C Cheung, Senior Member, and Sooyong Park. 2011. "Which Crashes Should I Fix First?: Pre-

dicting Top Crashes at an Early Stage to Prioritize Debugging Efforts." *TRANSACTIONS ON SOFTWARE ENGINEERING* 37 (3): 430–47.

Kim, Sunghun, Thomas Zimmermann, Kai Pan, and E. Jr. Whitehead. 2006. "Automatic Identification of Bug-Introducing Changes." In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 81–90. IEEE. doi:10.1109/ASE.2006.23.

Kim, Sunghun, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007a. "Predicting Faults from Cached History." In *29th International Conference on Software Engineering (ICSE'07)*, 489–98. IEEE. doi:10.1109/ICSE.2007.66.

———. 2007b. "Predicting Faults from Cached History." In *29th International Conference on Software Engineering*, 489–98. IEEE. doi:10.1109/ICSE.2007.66.

Koponen, Timo. 2006. "Life cycle of defects in open source software projects." In *Open Source Systems*, 195–200. Springer.

Lamkanfi, Ahmed, Serge Demeyer, Emanuel Giger, and Bart Goethals. 2010. "Predicting the severity of a reported bug." In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 1–10. IEEE. doi:10.1109/MSR.2010.5463284.

Landis, J. Richard, and Gary G. Koch. 1977. "The Measurement of Observer Agreement for Categorical Data." *Biometrics* 33 (1): 159. doi:10.2307/2529310.

Lo, D. 2013. "A Comparative Study of Supervised Learning Algorithms for Re-opened Bug Prediction." In *2013 17th European Conference on Software Maintenance and Reengineering*, 331–34. IEEE. doi:10.1109/CSMR.2013.43.

Menzies, Tim, and Andrian Marcus. 2008. "Automated severity assessment of software defect reports." In *2008 IEEE International Conference on Software Maintenance*, 346–55. IEEE. doi:10.1109/ICSM.2008.4658083.

Nagappan, N., and T. Ball. 2005. "Use of relative code churn measures to predict system defect density." In *Proceedings. 27th International Conference on Software Engineering, 2005.*, 284–92. IEEe. doi:10.1109/ICSE.2005.1553571.

Nam, Jaechang, Sinno Jialin Pan, and Sunghun Kim. 2013. "Transfer defect learning." In *2013 35th International Conference on Software Engineering (ICSE)*, 382–91. Ieee. doi:10.1109/ICSE.2013.6606584.

Nayrolles, Mathieu, Abdelwahab Hamou-Lhadj, Tahar Sofiene, and Alf Larsson. 2015. "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*, 101–10.

Nguyen, Anh Tuan, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. 2012. "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*, 70. New York, New York, USA: ACM Press. doi:10.1145/2351676.2351687.

Pan, Kai, Sunghun Kim, and E. James Whitehead. 2008. "Toward an understanding of bug fix patterns." *Empirical Software Engineering* 14 (3): 286–315. doi:10.1007/s10664-008-9077-5.

Perry, Dewayne E., and Carol S. Stieg. 1993. "Software faults in evolving a large, real-time system: a case study." In *Software Engineering—ESEC*, 48–67.

Rosen, Christoffer, Ben Grawi, and Emad Shihab. 2015. "Commit guru: analytics and risk prediction of software commits." In *Proceedings of The10th Joint Meeting on Foundations of Software Engineering*, 966–69. New York, New York, USA: ACM Press. doi:10.1145/2786805.2803183.

Runeson, Per, Magnus Alexandersson, and Oskar Nyholm. 2007. "Detection of Duplicate Defect Reports Using Natural Language Processing." In *29th International Conference on Software Engineering*, 499–510. IEEE. doi:10.1109/ICSE.2007.32.

Saha, Ripon K., Sarfraz Khurshid, and Dewayne E. Perry. 2014. "An empirical study of long lived bugs." In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 144–53. IEEE. doi:10.1109/CSMR-WCRE.2014.6747164.

Shihab, Emad, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto. 2010. "Predicting Re-opened Bugs: A Case Study on the Eclipse Project." In *2010 17th Working Conference on Reverse Engineering*, 249–58. IEEE. doi:10.1109/WCRE.2010.36.

Sun, Chengnian, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. "Towards more accurate retrieval of duplicate bug reports." *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, November. Ieee, 253–62. doi:10.1109/ASE.2011.6100061.

Tamrawi, Ahmed, Tung Thanh Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2011. "Fuzzy set-based automatic bug triaging." In *Proceeding of the 33rd International Confer-*

ence on Software Engineering - ICSE '11, 884. New York, New York, USA: ACM Press. doi:10.1145/1985793.1985934.

Tian, Yuan, David Lo, and Chengnian Sun. 2012. "Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction." In *2012 19th Working Conference on Reverse Engineering*, 215–24. IEEE. doi:10.1109/WCRE.2012.31.

Tian, Yuan, Chengnian Sun, and David Lo. 2012. "Improved Duplicate Bug Report Identification." In *2012 16th European Conference on Software Maintenance and Reengineering*, 385–90. IEEE. doi:10.1109/CSMR.2012.48.

Tufano, Michele, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. "When and why your code starts to smell bad," May. IEEE Press, 403–14. `http://dl.acm.org/citation.cfm?id=2818754.2818805`.

Valdivia Garcia, Harold, and Emad Shihab. 2014. "Characterizing and predicting blocking bugs in open source projects." In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, 72–81. New York, New York, USA: ACM Press. doi:10.1145/2597073.2597099.

Wang, Xinlei (Oscar), Eilwoo Baik, and Premkumar T. Devanbu. 2011. "System compatibility analysis of Eclipse and Netbeans based on bug data." In *Proceeding of the 8th Working Conference on Mining Software Repositories - MSR '11*, 230. New York, New York, USA: ACM Press. doi:10.1145/1985441.1985479.

Weiss, Cathrin, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. "How Long Will It Take to Fix This Bug?" In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 1–1. IEEE. doi:10.1109/MSR.2007.13.

Weiß, Cathrin, Thomas Zimmermann, and Andreas Zeller. 2007. "How Long will it Take to Fix This Bug ?" In *Fourth International Workshop on Mining Software Repositories (MSR'07)*, 1. 2.

Wu, Rongxin, Hongyu Zhang, Sunghun Kim, and SC Cheung. 2011. "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.*, 15–25. `http://dl.acm.org/citation.cfm?id=2025120`.

Xuan, Jifeng, He Jiang, Zhilei Ren, and Weiqin Zou. 2012. "Developer prioritization in bug repositories." In *2012 34th International Conference on Software Engineering (ICSE)*, 25–35. IEEE. doi:10.1109/ICSE.2012.6227209.

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

Zhang, Feng, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. 2012. "An Empirical Study on Factors Impacting Bug Fixing Time." In *2012 19th Working Conference on Reverse Engineering*, 225–34. IEEE. doi:10.1109/WCRE.2012.32.

Zhang, Hongyu, Liang Gong, and Steve Versteeg. 2013. "Predicting bug-fixing time: an empirical study of commercial software projects." In *International Conference on Software Engineering*, 1042–51. IEEE Press. `http://dl.acm.org/citation.cfm?id=2486788.2486931`.

Zhou, Jian, Hongyu Zhang, and David Lo. 2012. "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*, 14–24. IEEE. doi:10.1109/ICSE.2012.6227210.

Zimmermann, Thomas, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. 2012. "Characterizing and predicting which bugs get reopened." In *Proceedings of the 34th International Conference on Software Engineering, IEEE*, 1074–83. IEEE. doi:10.1109/ICSE.2012.6227112.