Do Developers Introduce Bugs when they do not Communicate? The Case of Eclipse and Mozilla

Mario Luca Bernardi[†], Gerardo Canfora[†], Giuseppe A. Di Lucca[†], Massimiliano Di Penta[†], Damiano Distante[‡]
[†] Dept. of Engineering-RCOST, University of Sannio, Italy
[‡] Fac. of Economics, Unitelma Sapienza Univ., Italy
{mlbernar,canfora,dilucca,dipenta}@unisannio.it, distante@unitelma.it

Abstract—**Developers working on related artifacts often communicate each other to coordinate their changes and to make others aware of their changes. When such a communication does not occur, this could create misunderstanding and cause the introduction of bugs.**

This paper investigates how the level of communication between committers relates to their proneness to introduce faults. This is done by identifying committers likely responsible of bug-introducing changes, and comparing—through social network measures—characteristics of their communication with the characteristics of other committers.

We report results from a study conducted on bugs from Eclipse and Mozilla, indicating that bug-introducing committers have a higher social importance than other committers, although the communication between themselves is significantly lower than for others.

Keywords: Bug Management, Developers' Communication, Social Network Analysis, Empirical Study.

I. Introduction

In the context of open source projects, or more generally of cooperative development, maintenance activities entail interaction among people spread across the world.

In large open source projects, tools such as bug-tracking systems (BTSs) or mailing lists are essential means of communication between developers, as well as between developers and other project contributors, e.g., users who submit bug reports or other requests for changes. This is not necessarily true on all projects: in many industrial projects face-to-face/voice communication prevails, and therefore BTSs and mailing lists do not keep track of all the discussion related to a bug [1].

Previous work on analyzing the intensity of communication in periods when bugs are introduced indicates that bugs tend to be introduced when there are communication peaks [2]. Although it is intuitive to believe that lack of proper communication would induce the introduction of defects, it would be interesting to empirically investigate to what extent does such a common wisdom apply in existing software projects. To the best of our knowledge, nobody has investigated yet whether, in these periods, there has been any communication between developers working on the same artifacts: the lack of (or a limited) communication between these developers could have increased their likelihood of introducing bugs. When multiple developers working in a distributed, cooperative environment modify the same artifacts without agreeing and properly communicating about each other changes, this can cause a limited understanding of the artifacts which, in turn, increases the likelihood of introducing bugs.

This paper investigates the relationship between how developers communicate through BTSs and their proneness to introduce bugs. Specifically, using a technique previously developed by Kim *et al.* [3], we identify bug reports requiring changes to fix them and associate the reported bugs with the committers who performed such changes. Then, we merge this information with a description of the committers communication network recovered from the BTS. Finally, we analyze—using social network analysis measures—the characteristics of the communication involving the committers who likely introduced a bug, as opposed to the communication of other committers who worked on the same artifacts impacted by the bug, but whose changes did not introduce bugs. Specifically, our study aims at answering the following research questions:

- **RQ1:** *Who are the bug-fixing committers? Are they the same persons who introduced the bugs, or, instead, other committers?*
- **RQ2:** *What is the importance—in the bug tracking system communication—of committers that likely introduced bugs, if compared to those that did not?*
- **RQ3:** *What is the level of communication of bug-introducing committers, if compared with other committers?*

Results of a study carried out on 3,873 bugs from Eclipse and 5,050 bugs from Mozilla indicate that committers responsible of bug-introducing changes tend to have a high importance in the BTS communication, higher than other committers and comparable only to committers who fixed the bug. However, they seldom communicate each other, i.e., the connectedness of bug-introducing committers is significantly lower than that of other committers.

The paper is organized as follows. Section II describes the process followed to extract data needed for our study. Section III defines the study and the analyzed variables.

Results are reported and discussed in Section IV, while Section V discusses the study threats to validity. After a discussion of related work (Section VI), Section VII concludes the paper and outlines directions for future work.

II. Analyzing Bug Introducing Changes and Developers' Network

This section describes the process used to extract data needed for our empirical study.

A. Step 1: Identification of Bug-Fixing Commits

First, we download the versioning system (CVS in our study) log and extract, for each commit, the files changed, their revision, the commit timestamp, the committer ID, and the commit note. Then, we cluster together related commits into change sets using the heuristic by Zimmermann *et al.* [4], which groups together the commits performed by the same committer, having the same commit note and a temporal distance shorter than 200 seconds. We restrict our attention to change sets referring to bug fixes, i.e., those matching a pattern such as *bug ID*, *issue ID*, or similar, where *ID* is a valid bug ID from the project BTS. This information—i.e., the bug ID mentioned in the commit note—is also used to link a bug fix change set with the related issue on the BTS.

B. Step 2: Identification of Bug-Introducing Changes

To identify bug-introducing changes, we used an approach inspired by the work of Kim *et al.* [5]. Specifically, we rely on the CVS annotation feature which, given a file revision, indicates for each file line the revision when the last change occurred. In essence, given a bug fix identified by the bug ID k, the approach works as follows:

1) for each file f_i, $i = 1 \ldots m_k$ involved in the bug fix k (m_k is the number of files changed in the bug fix k), and fixed in its revision $rfix_{i,k}$, we extract the file revision just *before* the bug fixing ($rfix_{i,k} - 1$), e.g., if the bug was fixed in revision 13 of the file f_i, we consider its revision 12 as it is the most recent revision still containing the bug k.
2) starting from the revision $rfix_{i,k} - 1$, the *annotate* option of CVS is used to identify for each source line in f_i changed to fix the bug k, the file revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island parser developed in Perl. This produces, for each file f_i, a set of $n_{i,k}$ bug-introducing revisions $rb_{i,j,k}$, $j = 1 \ldots n_{i,k}$
3) among revisions $rb_{i,j,k}$, we identify the oldest one: $rold_{i,k} = minTimeStamp\,(rb_{i,j,k})$, where $minTimeStamp$ returns the revision among the $rb_{i,j,k}$ having the smallest timestamp.

C. Distinguishing Bug-Introducing Committers from Other Committers

We define the set of bug-introducing committers B_k for bug k as follows:

$$B_k = \bigcup_{i=1}^{m_k} \left(\bigcup_{j=1}^{n_{i,k}} comm\,(rb_{i,j,k}) \right)$$

where $comm(rb_{i,j,k})$ returns the committer for $rb_{i,j,k}$. In other words, B_k is given by the union of all committers for bug-introducing changes. Then, we consider all the changes made to each file $f_i, i = 1 \ldots m_k$ in the time interval between $rold_{i,k}$ and $rfix_{i,k}$ to identify the set G_k of committers modifying f_i without introducing the bug k:

$$G_k = \bigcup_{i=1}^{m_k} \left(\bigcup_{j=1}^{n_{i,k}} comm\,(rg_{i,j,k}) \right) - B_k,$$

D. Step 4: Extraction of BTS Contributors Network

For each of the considered projects, we build the contributors social network by analyzing bug reports posted on the associated Bugzilla BTS.

Each bug report in Bugzilla contains a set of structural information fields, a text providing the description of the identified bug, and a list of comments posted by project contributors during the bug life-time. The list of comments associated to the bug report is ordered by the reported date, and each comment is characterized by the name and email of the contributor providing it (*author*), the reported date and time (*timestamp*) and the *description* of the comment itself. The list of comments associated to a bug report represents the discussion emerged on the specific bug and keeps trace of the communication occurred between contributors participating in the bug fixing process. In the following, reporters, assignees and authors are generically referred as "BTS contributors".

To build the contributors network for a given bug fix, we adopt a conservative approach by making the assumption that the author of the j-th comment associated to the report for the bug with ID k ($C_{k,j}$) replies (and thus communicates) to all the contributors that have posted one or more comments on the same bug prior to her. In order to consider in the network only contributors which are likely to cooperate, we filter bug reports based on the *component* bug report field, assuming that developers contributing to the fixing of bugs of a given component are likely to communicate in order to cooperate. We use comments' *timestamps* to determine if the associated communication event has to be considered relevant for the considered bug fixing, as they occurred in the time-frame calculated as described in Section II-F. Also, we resolve case of multiple names associated to the same person using an approach similar to the one used by Bird *et al.* [6] we previously used to analyze mailing lists for a different research [7].

E. Step 5: Mapping Committer IDs to Bug Tracking System Contributor Names

This step aims at linking BTS contributors with committer IDs. We use a process similar to the one used above to unify BTS contributors. Among others, we use heuristics to match IDs composed of initials to contributor names—e.g. mapping, where this does not induce ambiguities, *jfk* to *john fitzgerald kennedy*—or finding BTS contributor names to be linked to CVS committer IDs like *johnsmith* or *jsmith*, trying to compose BTS contributors first and last names, or first/middle name initials and last name. In some cases committer IDs could be email addresses. Specifically, this happened for Mozilla, which uses email addresses where "@" is replaced by "%", thus heuristics for email addresses can be applied as well.

F. Step 6: Extracting Discussions Relevant to a Bug Fixing

The final step of our analysis aims at identifying, for each bug fixing, the portion of contributors network of interest. We filter the network as follows:

1) *component filtering:* first, we restrict our analysis to bug reports related to the specific component on which the bug ID occurred.
2) *time frame filtering:* second, we need to restrict our time frame, and we select a time interval between the two following timestamps:
   - we consider the oldest bug-introducing change for all files f_i involved in the particular bug fixing, i.e., the $minTimeStamp(rold_{i,j}), \forall f_i,\ i = 1 \ldots m_k$ and look back of a period of three months. Since one of our aim is to identify to what extent bug-introducing committers talked each other, we need to start our analysis before the first bug-introducing change occurred—i.e., to see whether bug-introducing committers talked before such change. We consider three months as a reasonable period—also representative of the minimum period between two releases for the two analyzed projects (Eclipse and Mozilla)—in which a communication between two developers could occur before a change was done. Thus, we consider such a date $= minTimeStamp(rold_{i,j}) - 90$ *days* as the starting date of the interval considered. We also tried longer periods without obtaining substantially different results.
   - we consider as ending date of the interval of observation the date when the bug was opened. This is because we are interested to observe whether committers introducing bug-inducing changes (and other changes) communicated each other *before* the bug was discovered.

G. Running Example

Figure 1 reports a running example illustrating our data extraction approach for an hypothetical bug k. Figure 1-b shows, on the right side, the bug fixing commit, with filled boxes representing the changed code lines. The vertical lines before the bug fixing represent bug-introducing commits (when they contain a filled box traced to the fixed lines) or other commits (when they do not contain the filled box). In our example, *Alan* and *Jane* are bug-introducing committers (set B_k), while *Paul* and *Alice* are other committers modifying the fixed files (set G_k). Figure 1-a shows comments posted on bugs of the same component in the period considered (white area). The resulting network is shown in Figure 2, where filled squares represent bug-introducing committers ($\in B_k$), empty squares non bug-introducing committers ($\in G_k$), and dots other BTS contributors. Information on all contributors communication is reported in Table I.

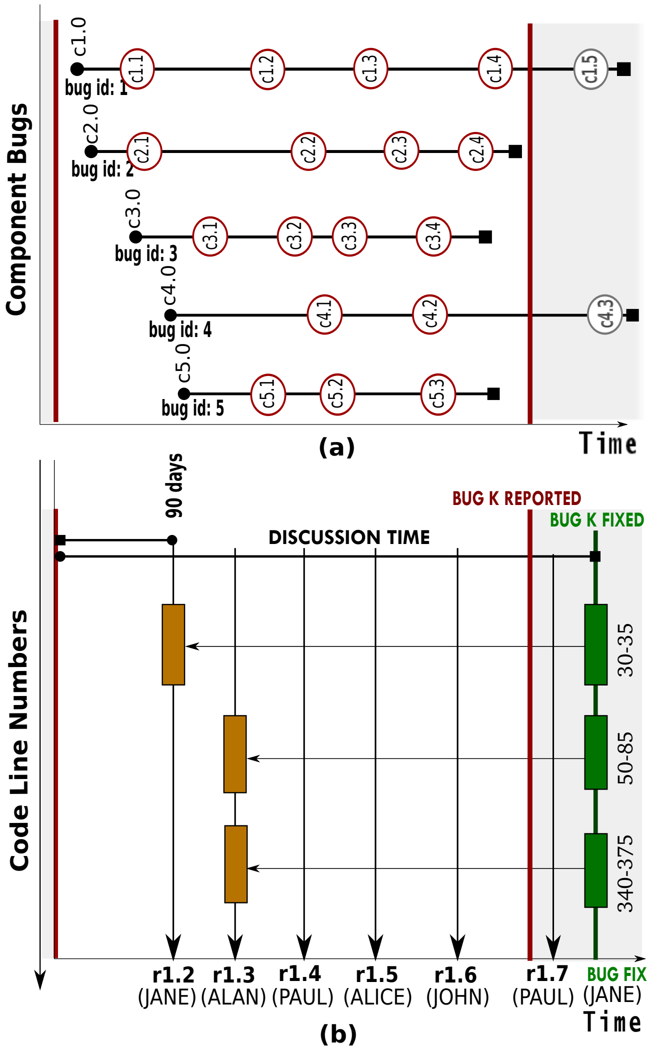


Figure 1. Running example: BTS comments (top) and commits (bottom).

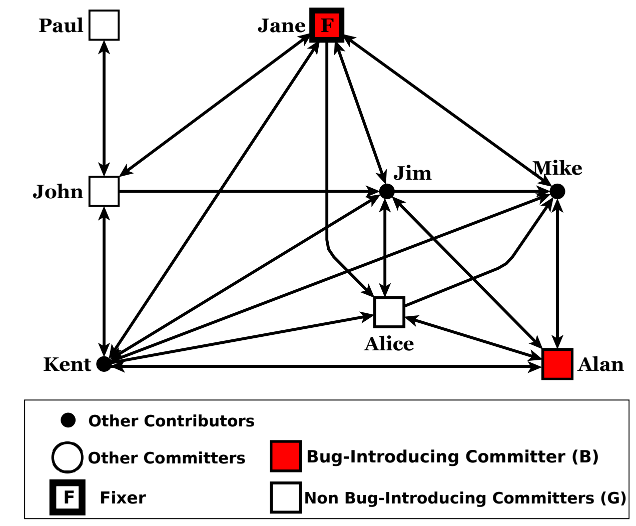


Figure 2. BTS contributors network related to the example of Figure 1.

Table I
INFORMATION ABOUT BTS CONTRIBUTORS FOR THE EXAMPLE OF FIGURE 1.

BTS Contributor	Committer	Reported bug ID	Assigned bug ID	Comment IDs
Alan	yes	n.a.	1	c1.1 c1.5
Alice	yes	n.a.	3	c1.2 c3.1
Jane	yes	2	2	c2.0 c2.4 c3.2
John	yes	4	n.a.	c4.0 c2.1 c4.2 c5.2
Jim	no	3	5	c3.0 c1.3 c3.4 c5.1
Kent	no	5	n.a.	c5.0 c1.4 c2.3 c3.3 c5.3
Mike	no	1	n.a.	c1.0 c2.2
Paul	yes	n.a.	4	c4.1 c4.3

III. EMPIRICAL STUDY

The *goal* of this study is to investigate the level of communication of bug-introducing committers (compared with the level of communication of other committers) with the *purpose* of understanding whether a lack of communication could favor the introduction of faults. The *quality focus* is software reliability and its relation with committers' communication. The *perspective* is mainly of researchers interested to investigate how, in open source projects, the lack of communication could influence the introduction of bugs, but also of project managers interested to promote a better communication, through suitable tools or better project management, with the objective of avoiding bugs due to lack of communication.

The *context* consists of data extracted from versioning systems (CVS) and BTS (Bugzilla) of two open source projects: Eclipse (*Platform* only) and Mozilla. Eclipse[1] is an open-source integrated development environment written mostly in Java. It is a platform used both in open-source communities and in the industry. Mozilla[2] is a suite comprising a Web browser, an email client, and other Internet utilities, written in C/C++. Specifically, we randomly selected 3,873 bugs from Eclipse and 5,050 bugs from Mozilla, spanning each project's lifetime. Table II reports, for the two projects, some overall information relevant for our study.

[1]http://www.eclipse.org

Table II
CHARACTERISTICS OF THE DATA SET USED IN THE STUDY.

	Eclipse	Mozilla
Products considered	Platform	Firefox, Core, Directory, JSS, NSPR, NSS, Plugins, Rhino
Observed period	May 2001-Jan 2009	Apr 1998-Sep 2010
# of bugs in the sample	3,873	5,050
mean # of edges per bug network	2,019	1,585
median # of edges per bug network	587	664
mean# of nodes per bug network	252	169
median # of nodes per bug network	152	124
# of committers	188	638
# of mailing list contributors	4,582	6,212
# of committers ∩ contributors	121	403

A. Research Questions

This study aims at addressing the three research questions we anticipated in the introduction and hereby detailed:

- **RQ1:** *Who are the bug-fixing committers? Are they the same persons who introduced the bugs, or, instead, other committers?* The first research question aims at understanding whether a bug was fixed by: (*i*) one of the committers that introduced the bug (and thus someone who realized his/her own mistake and corrected it); (*ii*) committers who changed the file in the meantime, without however introducing bugs (thus people that, to some extent, could be considered more reliable committers than the previous one); (*iii*) or, instead, by someone else that jumped in the discussion and solved the problem.
- **RQ2:** *What is the importance—in the bug tracking system communication—of committers that likely introduced bugs, if compared to those that did not?* The second research question aims at understanding the level of importance of bug-introducing committers in the BTS communication, if compared with other committers and with the bug-fixing committers. It could either happen that bug-introducing committers have, in general, a very poor role in the communication, and because of that introduce bugs, or that, despite their high communication (and thus importance) they introduce bugs more than others.
- **RQ3:** *What is the level of communication of bug-introducing committers, if compared with other committers?* The third research question aims at investigating to what extent bug-introducing committers *talk each other*, other than with other committers. This would

[2]http://www.mozilla.org

help to capture cases where multiple persons modify the same artifacts without coordinating each other, and thus they introduce a bug.

B. Analysis Method

This section describes the social network analysis measure used to characterize the communication occurring through the BTS, as well as the statistics used to analyze the data. All analyses have been performed using the *R* statistical environment[3] and, specifically, the packages *igraph* and *sna* to perform social network analysis. To address **RQ1**, we compute the proportion of bugs fixed by: (*i*) any of the committers who likely introduced the bug; (*ii*) committers that worked on the file during the same period when the bug was introduced, without however introducing bugs; or (*iii*) other committers not modifying the fixed files. We test whether proportions significantly differ using χ^2 goodness-of-fit test, and also use the odds ratio (OR) [8] effect size measure to compare the chance for a committer belonging to one of the three categories to fix a bug, as opposed to the others. An odd [8] indicates the likelihood that an event will occur as opposed to it not occurring. OR is defined as the ratio of the odds of an event occurring in one group to the odds of it occurring in another group. If the probabilities of the event in each of the groups are indicated as p (experimental group) and q (control group), then the OR is defined as: $OR = (p/(1-p))\,/\,(q/(1-q))$.

To address **RQ2**, we consider the network extracted for each bug (see *Step 6* of the process in Section II), and compute for each BTS contributor a series of social network measures that characterize their importance in the communication. In particular, we compute the following measures:

- *degree:* this is the most obvious measure of the importance of an actor; it is defined as the number of connections a node has. Actors with a high degree have a higher potential of being influential than those with a lower degree. Specializations of the degree are the *in-degree* and *out-degree*, indicating the number of incoming and outgoing edges, respectively. They distinguish whether an actor receives (in-degree) or provides (out-degree) information from/to a wide range of other actors.
- *betweenness:* this is a generalized concept of "centrality" for an actor, and it is defined as the percentage of all geodesic (shortest) paths from neighbor to neighbor that pass through the actor. Betweenness is computed by determining, for each pair of actors (a_1, a_2), the fraction of shortest paths between a_1 and a_2 that pass through the actor in question, and by summing such percentages over all pairs.

[3]http://www.r-project.org

For all bugs considered in the study, we compare the above measures across three groups of committers: (*i*) F: bug-fixing committers; (*ii*) B: bug-introducing committers; and (*iii*) G: committers that worked on files affected by the bug, without introducing the bug. The comparison is first performed using the Kruskal-Wallis test, which is a non-parametric test to compare more than two distributions; then, we perform a pairwise comparison using Mann-Whitney test and finally we correct p-values using the Holm's correction procedure. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on. Finally, in addition to the statistical comparison, we compute the effect size of the difference using the Cliff's delta non-parametric effect size measure [9], defined as the probability that a randomly selected member of one sample has a higher response than a randomly selected member of the second sample, minus the reverse probability. Cliff's delta ranges in the interval $[-1 \dots 1]$, and is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

To address **RQ3**, we identify, from the network of each bug, the following restrictions: (*i*) the network composed of committers $\in B$; and (*ii*) the network composed of committers $\in G$. For each restriction, we compute the Krackhardt's *connectedness*, a social network measure that indicates how well-connected is a set of nodes in a graph. The Krackhardt's connectedness for a digraph DG is defined as the fraction of all actors, $\{a_i, a_j\}$, such that there exists a path from a_i to $a_j \in DG$. The connectedness ranges from 0 (all isolated nodes) to 1 (weakly connected graph). Then, we perform a statistical comparison of the connectedness of the the subsets of committers similarly to what done for **RQ2**.

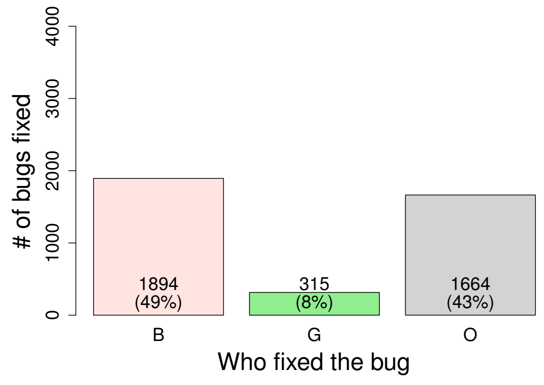
IV. Results

This section reports and discusses results of the empirical study defined in Section III. Raw data and working data sets are available for replication purposes[4].

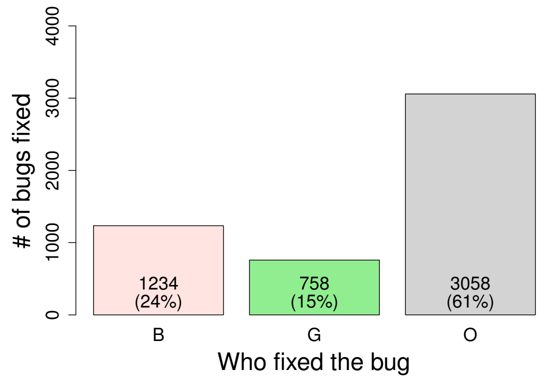
A. RQ1: Who are the bug-fixing committers? Are they the same persons who introduced the bugs, or, instead, other committers?

Figure 3-a shows, for Eclipse, the number and percentage of bugs fixed by (i) B: bug-introducing committers, (ii) G: committers working on the same files without introducing bugs, and (iii) O: other committers. As the figure shows, a large majority of fixings were performed either by people that previously introduced the bug and, thus, corrected their own mistakes (49%), or by other committers (43%) not really involved on that file, at least in a recent period of time. The latter could be representative of people that jumped in just with the purpose of performing a bug fixing. Only a small percentage (8%) of fixings were performed by the

[4]http://www.rcost.unisannio.it/mdipenta/talking-data.tgz

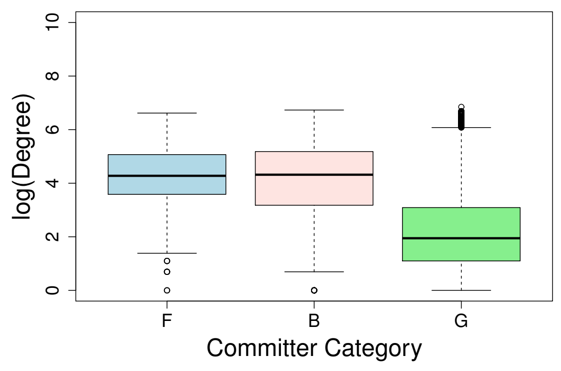


(a) Eclipse

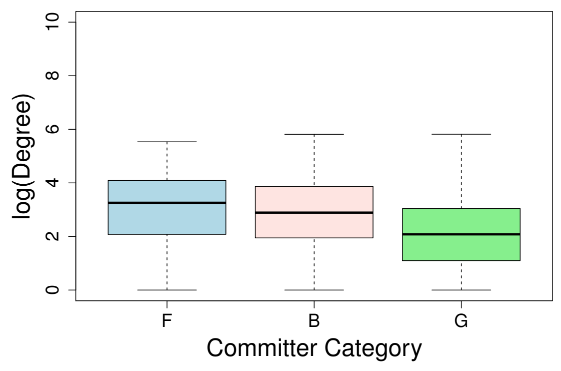


(b) Mozilla

Figure 3. Number and percentage of bug fixings of (B) bug-introducing committers, (G) non bug-introducing committers who changed fixed files, and (O) other committers.



(a) Eclipse



(b) Mozilla

Figure 4. Degree for different kinds of committers. F: bug fixers, B: bug-introducers, G: non-bug introducers.

set of committers who worked on the files impacted by the fixing, without actually introducing the bug.

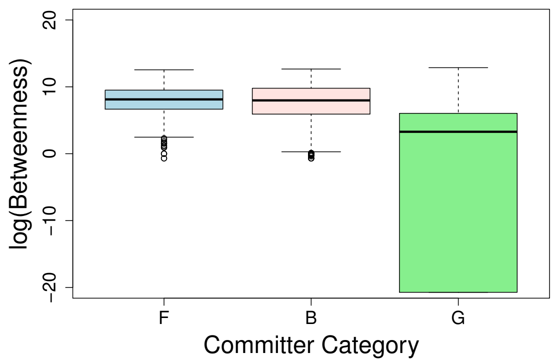
χ^2 test indicated a significant difference, in proportions, among the three sets of fixings (p-value< 0.001). In terms of OR, committers $\in B$ have OR=6 times the chances of fixing a bug than those $\in G$; committers $\in O$ have OR=5.28 times the chances of fixing a bug than those $\in G$, while the OR between B and G is 1.14, i.e., almost even.

Figure 3-b shows results for Mozilla. In this case, the percentage (61%) of bugs fixed by other committers is more than twice the percentage of bugs fixed by bug introducers (24%). χ^2 test indicated a significant difference, in proportions, among the three sets of fixings (p-value< 0.001). Committers $\in B$ have 1.6 times the chances of fixing a bug than those $\in G$; committers $\in O$ have 4 times the chances of fixing a but than those $\in G$, while the OR between B and G is 0.40, this time the percentage is higher of B than for G.

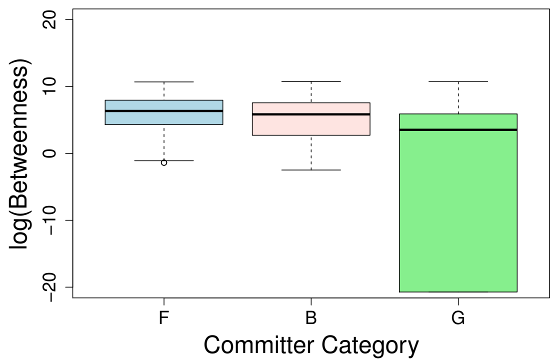
In summary, we can conclude **RQ1** stating that *a non-trivial number of bugs (between 24% and 49%) is fixed by the same committers who introduced them*; however, a high number of bugs (between 43% and 61%) is fixed by people that did not work recently on files affected by the bug.

B. RQ2: What is the importance—in the bug tracking system communication—of committers that likely introduced bugs, if compared to those that did not?

Figure 4 shows a comparison of the degree social network measure for three different kinds of committers, i.e., (F) bug fixers, (B) bug introducers, and (G) committers who worked on the fixed files without introducing bugs. Since distributions are highly skewed, boxplots are in logarithmic scale. We only report the overall degree, while we omit figures related to in-degree and out-degree; however, differences among different groups of committers are perfectly consistent among degree, in-degree and out-degree. The right-side of Table III reports results of a pairwise comparison of degree distributions for various committers,



(a) Eclipse



(b) Mozilla

Figure 5. Betweenness for different kinds of committers. F: bug fixers, B: bug-introducers, G: non-bug introducers.

specifically Mann-Whitney test p-values (adjusted with the Holm's correction) and Cliff's delta effect sizes.

For Eclipse, as Figure 4-a shows, and according to p-values and Cliff's delta in the top-left side of Table III, it can be stated that bug fixers and bug introducers tend to have a significantly higher degree than other committers, with a high effect size ($d > 0.47$). Bug introducers have a significantly lower degree than bug fixers, though the difference, in terms of effect size, is negligible ($d = -0.02$). Considerations made for Eclipse are still valid for Mozilla, as Figure 4-a and values on the bottom-left part of Table III indicate.

Figure 5 shows (again in logarithmic scale) the betweenness computed for different groups of committers, while statistics are reported in the right-side of Table III. As it can be seen, results are consistent with those obtained for the degree, i.e., bug fixers and bug introducers tend to have a significantly higher betweenness than other committers, with a *high* effect size. Again, the effect size is negligible when

Table III
COMPARISON BETWEEN DEGREE AND BETWEENNESS OF DIFFERENT GROUPS OF COMMITTERS: ADJUSTED MANN-WHITNEY P-VALUE AND CLIFF'S DELTA EFFECT SIZE.

ECLIPSE	Degree		Betweenness	
Comparison	adj. p	Cliff's delta	adj. p	Cliff's delta
F vs. B	< 0.01	-0.01	< 0.01	-0.02
F vs. G	< 0.01	0.78	< 0.01	0.72
B vs. G	< 0.01	0.73	< 0.01	0.65

MOZILLA	Degree		Betweenness	
Comparison	adj. p	Cliff's delta	adj. p	Cliff's delta
F vs. B	< 0.01	0.09	< 0.01	0.05
F vs. G	< 0.01	0.56	< 0.01	0.52
B vs. G	< 0.01	0.42	< 0.01	0.40

comparing betweenness of bug-introducing committers with bug fixers.

We can conclude **RQ2** stating that bug-introducing committers tend to have a higher importance (measured in terms of degree and betweenness) than other committers, and comparable only to bug fixers. In essence, it appears that *bug introducers and bug fixers participate to the BTS discussion very actively, in general more than other committers.*

C. RQ3: What is the level of communication of bug-introducing committers, if compared with other committers?

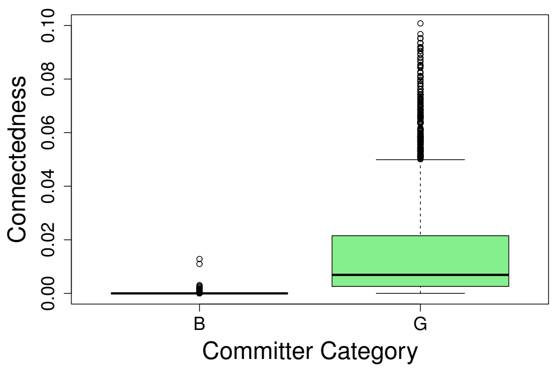
Figure 6, shows boxplots of the connectedness for (B) bug-introducing committers, and (G) committers who worked on the fixed files without introducing a bug. Mann-Whitney test indicate that, for both Eclipse and Mozilla, committers $\in G$ have a significantly higher connectedness (p-value < 0.01) than those $\in B$, with a high effect size (0.89 for Eclipse and 0.62 for Mozilla).

We can conclude **RQ3** stating that the connectedness of bug-introducing committers is significantly lower than the median connectedness of other committers working on the fixed files. This result, if compared with results of **RQ2**, indicates that, despite bug-introducing committers have an important role in the BTS discussion, the communication between them is scarce: *people that work on the same file and induce bugs do not communicate each other, or at least communicate less than others.*

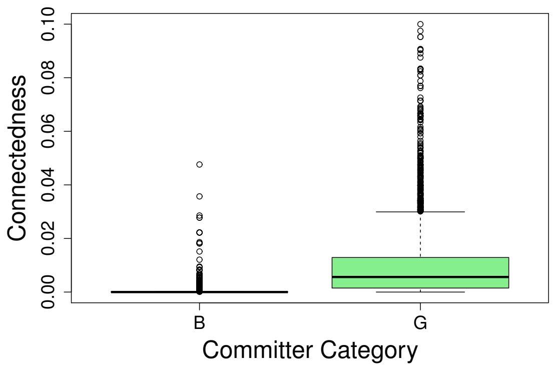
D. An example: Eclipse Bug # 50892

As an example of communication graph, Figure 7 shows the discussion related to Eclipse bug #50892 [5] *"Search empty using Search perspective"*. It has to be recalled that the discussion is not the one related to the bug itself (that instead started when the bug was discovered, thus after the period we observed) but rather the discussion between the

[5]https://bugs.eclipse.org/bugs/show_bug.cgi?id=50892

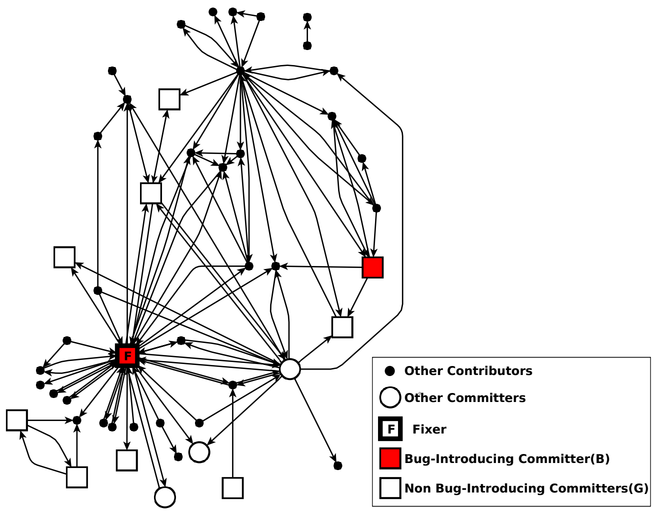


(a) Eclipse



(b) Mozilla

Figure 6. Connectedness for B:bug-introducers, G: non-bug introducers.



F
● Other Contributors
Other Committers
Fixer
Bug-Introducing Committer(B)
Non Bug-Introducing Committers(G)

Figure 7. Discussion graph related to Eclipse Bug #50892. Note: the discussion includes threads on the same component in the time period of interest for the bug.

bug introducing change (May 2002) and the bug fix (Feb 2004) on the component (*Search*) impacted by the bug. As it can be noticed, while *G* committers (white squares) and other committers (circle) mostly communicate each other, the two bug-introducing committers (one of which is also bug fixer) are basically disconnected, except for the link through other BTS contributors (black dots) not considered when computing the network connectedness.

E. Some Anecdotal Evidence

Although this paper does not aim at claiming a cause-effect relation between the lack of communication (through BTS) and bug introducing changes, and although it is very unlikely to find evidence of that in the BTS text, we attempted at mining bug reports of bugs for which the communication between bug-introducers was lower than the communication of other committers. In some cases, we found comments mentioning that the bug was fixed after clarifying the problem with another person, i.e., there was likely lack of communication.

For example, in Eclipse, we found such evidence in bug #108162 *"After talking to John what I think we want to do is remove your need to spin . . ."*, bug #129184 *"I talked to Tod. With the above clarification, I'm happy with the API addition."*, and bug #50131 *"this is not what you think is happening, then we're talking about different . . ."*

In Mozilla, we found examples of evidence in bug # 104075 *"Please talk with me before doing so.*, bug # 124485 *"I know next to nothing about menus. You need to talk to pinkerton."*, bug #159359 *"new patch after talk with bz and jkeiser"*, and bug #163645 *"Okay after talking with jst, I think I have a handle on how to properly fix"*.

V. Threats to Validity

This section discusses the main threats to the validity of our study.

Construct validity threats concern the relationship between theory and observation. There could be imprecisions/omissions in the measurements made in this paper for a series of reasons:

- There is not a perfect correspondence between developers and committers. Often the author of a patch submits it to a quality assurance team and, once the patch is approved, it is committed in the versioning system by an authorized committer, which may or may not correspond to the patch author. Thus, we are aware that committers' activity and communication only represent a limited view of the reality. However, since committers are responsible for the quality of the code they modify, they should coordinate each others, i.e., their communication can still be considered an important factor related to fault-proneness.

- The mapping between committer IDs and contributor names can cause imprecisions, however we have manually validated it.
- in this study we have limited our attention to communication through BTSs only. In future work we plan to also include data from mailing lists, for projects for which they are available. Also, as shown by Aranda and Venolia for industrial projects [1], software repositories do not fully capture the rationale around a bug fixing activity; we found this is some cases to be true also for open source projects. For example, in Eclipse there were bug reports referring to IRC communication, e.g., *"talk over IRC this week"* (bug #154329).
- The way a social network is built from BTSs only represents an approximation of what is the real link between people and—as explained by Bird *et al.* [10]—heuristics used to reconstruct the network often are cause of omissions and imprecisions.
- Finally, we are aware that the linking between commits and bug reports through bug IDs in commit notes is far for being complete, although this represent the techniques mostly adopted in mining software repository studies.

Conclusion validity concerns the relationship between the treatment and the outcome. As explained in Section III-B we use appropriate statistical procedures to support claims for the three research questions, including non-parametric tests and effect-size measures.

Threats to *internal validity* concern factors that can influence our observations. This is an exploratory study and we cannot claim a cause-effect relationship between the lack of communication and bug-introducing changes. Rather, we provide statistically significant evidence that bug-introducing committers communicate between themselves less than others, and qualitative explanations of that phenomenon. Other work investigate different properties, such as code ownership [11], that can also influence fault-proneness.

Threats to *external validity* concern the generalization of our findings. Although we performed our analyses on discussions related to a pretty large set of bugs (3,873 for Eclipse and 5,050 for Mozilla) of two projects belonging to different domains, it must be clear that our conclusions apply to these two projects, and their generalization requires replications on further datasets.

VI. Related Work

To the best of our knowledge, the closest work to ours has been carried out by Abreu and Premraj [2]. They extract a developer network from Eclipse-JDT mailing lists and from information related to bug-introducing changes (also using the approach by Kim *et al.* [5]). The mailing list activity of developers (and of core developers in particular) in a specific time frame is related to the presence of bug-introducing changes in the same time frame. The authors found that bugs are generally introduced in periods where there is a peak of communication, either among core developers or other developers as well. Though there are similarities in the way data is extracted and on the kind of social network analysis measures to define the importance of a developer in a network, there are some substantial differences with respect to our work. Indeed, Abreu and Premraj [2] focus is on *when* peaks of communication and bug-introducing changes occur, while our work specifically investigates the communication level of *who* was likely responsible of bug-introducing changes, and compares it with the communication of other committers.

Other than communication, another very important factor that can relate to fault-proneness is the code *ownership* of developers that modify a given component, i.e., to what extent a component is modified by its major contributor(s) or by other minor contributors. Bird *et al.* [11] investigate such a phenomenon in Microsoft Windows Vista and Windows 7, finding a negative correlation between ownership and fault-proneness and, more important, a high positive correlation between changes made by minor contributors and fault-proneness.

In software development contexts, socio-technical congruence among different systems has been analyzed in the work of Bird *et al.* [12], who adopt communication and development data to detect the network structure of the community of five large open source projects and to detect the centrality of individuals. Pohl and Diehl [13] show how social networks could be used to determine roles in a community of developers within a single project. Canfora *et al.* [7] analyze the role of committers involved in cross-system bug fixings between FreeBSD and OpenBSD, finding that those committers exhibit higher brokerage, degree, and betweenness metrics than others.

Some authors have also used social network measures for the purpose of assessing software quality or to predict failures, as in the work of Bettenburg and Hassan [14] or in the work of Bird *et al.* [15]. In our case, rather than using social network analysis on communication occurring on BTSs or mailing lists to build predictor models, we focus on what happens between committers working on the same file in a same time frame: our conjecture is that, if they do not communicate or they rarely communicate, this could possibly induce a bug.

VII. Conclusion and Work-in-Progress

This paper reported an empirical study aimed at analyzing the "social importance" in bug-tracking system (BTS) communication of committers likely responsible of introducing bugs. Then, the paper analyzes to what extent bug-introducing committers communicate each other while modifying the same artifact (and introducing a bug), compared to the level of communication of other committers.

Results of the study, conducted on 3,873 Eclipse bugs and 5,050 Mozilla bugs, provide different insights about the role of bug-introducing committers. First, the study shows that a substantial percentage of bugs (49% and 24% for Eclipse and Mozilla respectively) are fixed by the same committers who likely introduced them, and only a small minority of bugs (8% and 15%) are fixed by people who previously modified the files without introducing a bug. All other fixings (43% and 61%) are performed by people that, since when the bug was introduced, did not contribute to the source code impacted by the bug. Then, the study shows that, in general, bug-introducing committers have a high importance (in terms of degree and betweenness) in the communication concerning the same component where the bug occurred, and performed in the period since the bug was introduced until the bug was discovered (and its bug report opened). Such a level of importance is only comparable to another category of committers, i.e., those who fixed the bug. Last, and most important, the study shows that bug-introducing committers communicate *between themselves* significantly less than other committers, and in particular significantly less than other people working on the same files in the period when the bug was introduced.

There are several directions for future work. First, we plan to extend the study considering other sources of information, mailing lists in particular, and a larger set of bugs and systems. Second we would better investigate on the nature of bug-introducing committers, for example understanding whether they are people working concurrently on many different components, or people having a given level of experience in modifying particular components or performing specific kinds of changes.

References

[1] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 298–308.

[2] R. Abreu and R. Premraj, "How developer communication frequency relates to bug introducing changes," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 153–158.

[3] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*. IEEE Computer Society, 2006, pp. 81–90.

[4] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 563–572.

[5] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 181–196, 2008.

[6] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 137–143.

[7] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD," in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*. IEEE, 2011, pp. 143–152.

[8] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[9] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[10] R. Nia, C. Bird, P. T. Devanbu, and V. Filkov, "Validity of network analyses in open source projects," in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010, Cape Town, South Africa, May 2-3, 2010, Proceedings*. IEEE, 2010, pp. 201–209.

[11] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 4–14.

[12] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 24–35.

[13] M. Pohl and S. Diehl, "What dynamic network metrics can tell us about developer roles," in *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, ser. CHASE '08. New York, NY, USA: ACM, 2008, pp. 81–84.

[14] N. Bettenburg and A. E. Hassan, "Studying the impact of social structures on software quality," in *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*. IEEE Computer Society, 2010, pp. 124–133.

[15] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. T. Devanbu, "Putting it all together: Using socio-technical networks to predict failures," in *ISSRE 2009, 20th International Symposium on Software Reliability Engineering, Mysuru, Karnataka, India, 16-19 November 2009*. IEEE Computer Society, 2009, pp. 109–119.